1. Preparation

A solid background in discrete mathematics, probability, programming, basic algorithms, and data structures are necessary prerequisites for this course.

Please read the Appendix: Mathematical Background in the textbook (Corman et al., 2009).

Common Functions

- Polynomials:

$$f(n) = 1, \quad f(n) = n, \quad f(n) = n^2, \quad f(n) = n^3, \cdots$$

$$f(n) = 1, \quad f(n) = n, \quad f(n) = n(n-1), \quad f(n) = n(n-1)(n-2), \cdots$$

- Poly-Logarithmic:

$$f(n) = \lg(n), \quad f(n) = n\lg(n), \quad f(n) = n^2 \lg n, \cdots$$

- Exponential:

$$f(n) = 2^n, \quad f(n) = n!, \quad f(n) = n^n, \cdots$$

Notes:

1. $\lg(n)$ is the logarithm of n base 2.

$$\lg(n) = \frac{\log(n)}{\log(2)} \approx 3.3219 \cdots \log(n)$$

2. We are most interested in *running time functions* f that satisfy
   - The domain of f is restricted to the natural numbers (non-negative integers). The function f is often defined on a larger domain, but for us, it is only evaluated on natural numbers.
   - f maps the natural numbers into the real numbers
   - f is positive and eventually monotonically increasing
   - f is computable. That is, there is Turing machine that computes $f(n)$.

Sample growth rates

Function	$n = 1$	$n = 10$	$n = 100$	$n = 1000$
$\lg n$	0	3.32	6.64	9.96
n	1	10	100	1000
$n \lg n$	0	33.2	664	9966
n^2	1	100	10000	10^6
n^3	1	1000	10^6	10^9
2^n	2	1024	10^{30}	10^{300}
n^n	1	10^{10}	10^{200}	10^{3000}

Big-O Notation

Definition 1: Big-O notation

$f : \mathbb{N} \mapsto \mathbb{R}$ *is big-O of* $g : \mathbb{N} \mapsto \mathbb{R}$, *denoted*

$$f = O(g) \quad f \text{ is big-O of } g$$

if and only if there exists a natural number $m > 0$ *and a constant* $c > 0$ *such that*

$$f(n) \leq c \cdot g(n)$$

for all $n > m$. *The value* m *and* c *are called* witnesses.
Informally, "for sufficiently large n, f *grows no faster than* g."

To show that f is big-O of g, find *witnesses*: a natural number m and a positive constant c, such that the inequality $f(n) \leq c \cdot g(n)$ is True for all $n > m$. Can you find the witnesses or deduce that there are none for the following?

1. $f(n) = 5n$ is $O(n)$
2. $f(n) = 4n + 3$ is $O(n^2)$
3. $f(n) = 4n^2 + n - 1$ is $O(n^2)$
4. $f(n) = \sqrt{n}$ is $O(n)$
5. $f(n) = 5n$ is not $O(1)$
6. $f(n) = 4n + 3$ is not $O(\sqrt{n})$
7. $f(n) = 4n^2 + n - 1$ is not $O(n)$
8. $f(n) = \sqrt{n^3}$ is not $O(n)$

Theorem 1: Big-O Properties

Pretend f, g and h are running time functions. Then the following properties hold:

1. $f = O(f)$, big-O is reflexive
2. If $f = O(g)$ and $g = O(h)$, then $f = O(h)$, big-O is transitive
3. If f is $O(g)$, then $cf(n)$ is $O(g)$ for any constant $c > 0$.
4. If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then

$$f_1 + f_2 = O(\max(g_1, g_2))$$

$$f_1 f_2 = O(g_1 g_2)$$

Omega Notation

To show that f is $\Omega(g)$, we must find an integer m and a positive constant c, such that the inequality $f(n) \geq c \cdot g(n)$ is true for all $n > m$.

Definition 2: Big-Ω notation

$f : \mathbb{N} \mapsto \mathbb{R}$ *is* big-Omega *of* $g : \mathbb{N} \mapsto \mathbb{R}$, *denoted*

$$f = \Omega(g)$$

if and only if there exists witnesses: a natural number $m > 0$ *and a constant* $c > 0$ *such that*

$$f(n) \geq c \cdot g(n), \quad \forall n > m$$

Informally "function f *grows no slower than a function* g*."*

Can you find the witnesses or deduce that there are no witnesses for the following?

1. $f(n) = 5n$ is $\Omega(n)$
2. $f(n) = 4n^2 + 3$ is $\Omega(n)$
3. $f(n) = 4n^2 + n - 1$ is $\Omega(n^2)$
4. $f(n) = n$ is $\Omega(\sqrt{n})$

Theta Notation

To show that f is $\Theta(g)$, we must find an integer m and positive constants c_1 and c_2, such that the inequalities $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ is true for all $n > m$.

Definition 3: Big-Θ notation

$f : \mathbb{N} \mapsto \mathbb{R}$ *is* big-theta *of* $g : \mathbb{N} \mapsto \mathbb{R}$, *denoted*

$$f = \Theta(g)$$

if and only if there exists witnesses: a natural number $m > 0$ *and real constants* $c_1 > 0$, $c_2 > 0$ *such that for all* $n > m$,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

In words "a function f *grows at the same rate as a function* g*."*

Can you find the witnesses or deduce that there are no witnesses for the following?

1. $f(n) = 5n$ is $\Theta(n)$
2. $f(n) = 4n^2 + 3$ is $\Theta(n^2)$
3. $f(n) = 7n^2$ is $\Theta(n)$
4. $f(n) = n \log_{10}(n)$ is $\Theta(n \lg n)$

Theorem 2: Big-Θ is an equivalence relation

Suppose f, g and h are running time functions. Then the following properties hold:

1. $f(n)$ is $\Theta(f(n))$ (reflexive)
2. $f(n)$ is $\Theta(g(n))$ if and only if $g(n)$ is $\Theta(f(n))$ (symmetric)
3. if f is $\Theta(g)$ and g is $\Theta(h)$, then f is $\Theta(h)$ (transitive)

Therefore, $\Theta(f(n))$ defines an *equivalence relation* on the class of running time functions.

Little-o Notation

Definition 4: Little-o notation

$f : \mathbb{R} \mapsto \mathbb{R}$ *is* little-o *of* $g : \mathbb{R} \mapsto \mathbb{R}$, *denoted*

$$f(x) = o(g(x))$$

means for all $c > 0$ *there exists some* $m > 0$ *such that* $0 \leq$

$f(x) < cg(x)$ *for all* $x \geq m$.

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

The value of m *must be independent of* x*, but may depend on* c.
Informally "a function f *grows much more slowly than the function* g*."*

Little-o notation is primarily used in approximation algorithms. For instance, under relatively weak assumptions, the recurrence

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

converges to a root x where $f(x) = 0$ at a *quadratic* rate. That is, the error at stage n is about the square of the error at stage $(n-1)$. For instance, if the error at stage $(n-1)$ is about 10^{-k}, then it will be about 10^{-2k} at the n-th step, double the number of accurate digits.

The recurrence is known as Newton's method.

The Floor and Ceiling Functions

Conversion among natural, integer, rational and real numbers is common in algorithms. Floors and ceilings map real numbers to integers.

- The *floor* of x is the largest integer n less than or equal to x.

$$\lfloor x \rfloor$$

- The *ceiling* of x is the smallest integer n greater than or equal to x.

$$\lceil x \rceil$$

- Examples:

$$\lfloor \pi \rfloor = 3$$
$$\lceil \pi \rceil = 4$$
$$\lfloor e \rfloor = 2$$
$$\lceil e \rceil = 3$$
$$\lfloor n/2 \rfloor = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (n-1)/2 & \text{if } n \text{ is odd} \end{cases}$$
$$\lfloor x/y \rfloor = (x - x \bmod y)/y$$
$$\lfloor \log_b k \rfloor + 1 = \lceil \log_b (k+1) \rceil = \text{base } b \text{ numerals needed to write } k$$

Factorial Functions

- Given a positive integer n, define

$$n! = 1 \cdot 2 \cdots n$$

- Gauss's trick to show $n!$ is pretty big

$$(n!)^2 = (1 \cdot 2 \cdots n)(n \cdots \cdot 2 \cdot 1) = \prod_{k=1}^{n} k(n-k+1)$$

For all integer k between 1 and n, we have

$$n \le k(n+1-k) \le \frac{1}{4}(n+1)^2,$$

since the quadratic

$$k(n+1-k) = (n+1)^2/4 - (k-(n+1)/2)^2$$

has its smallest value at $k = 1$ and its largest value at $k = (n+1)/2$. Therefore

$$\prod_{k=1}^{n} n \le (n!)^2 \le \prod_{k=1}^{n} \frac{(n+1)^2}{4}$$

that is,

$$n^{n/2} \le n! \le \left(\frac{n+1}{2}\right)^n$$

- Stirling's formula for $n!$ is more accurate:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

This is just the dominant term in Stirling's formula.

- Carrying more terms in Stirling's formula gives

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{5140n^3} + O\left(\frac{1}{n^4}\right)\right)$$

- Stirling's formula is called an *asymptotic* formula for $n!$

Binomial Coefficients

Please read Appendix C: Counting and Probability in the textbook (Corman et al., 2009).

- The binomial coefficient $\binom{n}{k}$ denoted the number of ways k objects can be selected (without regard to order) from a set of n objects.

- It can be shown that

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k!} \\ &= \frac{n!}{k!(n-k)!}\end{aligned}$$

- Pascal's Triangle is an array of binomial coefficients

n\k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

In the following we assume that n and k are non-negative integers, although some of the formulas can be extended to other values for n and k

- Binomial Theorem:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

- Addition Formula:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- Symmetry Identity:

$$\binom{n}{k} = \binom{n}{n-k}$$

- Absorption Identity:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{n-k}$$

- Parallel Summation:

$$\sum_{k \le n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

- Summation on Upper Index

$$\sum_{m \le k \le n} \binom{k}{m} = \binom{n+1}{m+1}$$

- $2^n = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n}$.
- $0^n = \binom{n}{0} - \binom{n}{1} + \cdots + (-1)^n \binom{n}{n}$.
- $(1+z)^r = \sum \binom{r}{k} z^k, \quad |z| < 1$.
- $1 + 2 + \cdots + n = \binom{1}{1} + \binom{2}{1} + \cdots + \binom{n}{1} = \binom{n+1}{2}$

Harmonic Numbers

- Define the n Harmonic number to be

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

- The name is derived from music: The kth harmonic of a string is the tone produced by a string that is $1/k$ times as long as the first string.
- Here is an asymptotic formula for Harmonic numbers

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} + \cdots = O(\lg(n))$$

  where

$$\gamma \approx 0.577215664901\cdots$$

  is called Euler's constant
- You might recognize the Bernoulli numbers in this expansion of H_n

Logarithmic Functions

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b(x^n) = n \log_b x$
- $\log_b x = \frac{1}{\log_x b}$
- $\log_b x = \log_b a \log_a x$
- $x^{\log_b y} = y^{\log_b x}$
- This last formula is especially useful for rewriting functions such as

$$f(n) = a^{\log_b n}$$

  in the form

$$f(n) = n^{\log_b a}$$

Mathematical Induction

- The principle of mathematical induction states:
  - If proposition $P(n)$ is true for $n = k$ and
  - If whenever $P(n)$ is true for some $n \geq k$, then $P(n+1)$ is also true
  - Then $P(n)$ is true for all $n \geq k$

- Proving $P(k)$ is called the *basis for induction*
- Proving $P(n) \Rightarrow P(n+1)$ is called the *inductive step*

Consider proposition "$P(n)$: the sum of the first n squares is $\frac{1}{3}n\left(n+\frac{1}{2}\right)(n+1)$" or in symbols

$$1^2 + 2^2 + \cdots + n^2 = \frac{1}{3}n\left(n+\frac{1}{2}\right)(n+1)$$

- The basis is $P(1)$:

$$1^2 = \frac{1}{3}1\left(1+\frac{1}{2}\right)(1+1)$$

- Suppose $P(n)$ for some $n \geq 1$.
- Then consider $P(n+1)$

$$\begin{aligned} & 1^2 + 2^2 + \cdots + n^2 + (n+1)^2 \\ = \quad & \frac{1}{3}n\left(n+\frac{1}{2}\right)(n+1) + (n+1)^2 \\ = \quad & \frac{1}{3}(n+1)\left(n+\frac{3}{2}\right)(n+2) \end{aligned}$$

- Thus $P(n)$ is true for all $n \geq 1$

  Consider proposition,

  "If $n = 2^p$ and
  $$T(n) = 2T(n/2) + n$$
  for $n > 1$ and $T(1) = 0$, then $T(n) = n \lg n$"

- The basis is $P(1)$: $n = 2^0 = 1$ and $T(1) = \lg 1 = 0$
- Suppose $P(n)$ for some $n = 2^p$, $p \geq 0$
- Then consider $P(2n)$

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\lg(n) + 2n \\ &= 2n[\lg(n) + 1] \\ &= 2n[p+1] \\ &= 2n\lg(2n) \end{aligned}$$

- Thus $P(n)$ is true for all $n \geq 1$, i.e. $T(n) = n\lg n$

Summation Formulas

Please read Appendix A: Summations in the textbook (Corman et al., 2009).

- Arithmetic sum

$$\sum_{k=0}^{n-1}(a+kd) = na + d\binom{n}{2}$$

- Geometric sum $(r \neq 1)$:

$$1 + r + r^2 + r^3 + \cdots + r^{n-1} = \sum_{k=0}^{n-1} r^k = \frac{1-r^n}{1-r}$$

- Geometric series: for $|r| < 1$

$$1 + r + r^2 + r^3 + \cdots = \sum_{k=0}^{\infty} r^k = \frac{1}{1-r}$$

- Alternating geometric series

$$1 - r + r^2 - r^3 + \cdots = \sum_{i=0}^{\infty} (-r)^i = \frac{1}{1+r}$$

What value of r is disallowed in a geometric series?

- Derivative of geometric series

$$\sum_{i=0}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}$$

- Integral of geometric series

$$\sum_{i=1}^{\infty} \frac{r^i}{i} = -\ln(1-r)$$

Recursion

The function $T(n) = \lg(n)$ satisfies the recursion

$$T(n) = T(n/2) + 1, \quad n \in \{m | m \pmod 2 = 0,\ m \geq 2\}$$

The function $T(n) = n\lg(n)$ satisfies the recursion

$$T(n) = 2T(n/2) + n, \quad n \in \{m | m \pmod 2 = 0,\ m \geq 2\}$$

What function $T(n)$ satisfies the recursion

$$T(n) = 2T(n/2) + 1, \quad n \in \{m | m \pmod 2 = 0,\ m \geq 2\}$$

Theorem 3: The Master Theorem for Recursion

Let $a \geq 1$ and $b > 1$ be constants and let $f(n)$ be a function. Let $T(n)$ be defined by

$$T(n) = aT(n/b) + f(n)$$

Then,

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = O(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and

   $$af(n/b) \leq cf(n)$$

   for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Generating Functions

Let $\vec{S} = \langle s_0, s_1, s_2, \ldots \rangle$ be a sequence. The generating function for $\vec{S}$ is

$$G(\vec{S}) = \sum_{k=0}^{\infty} s_k z^k$$

Generating functions are usually first studied as series in calculus. Here are some examples for important sequences.

$$G(\langle 1, 1, 1, \ldots \rangle) = \sum_{k=0}^{\infty} z^k = \frac{1}{1-z} = (1-z)^{-1}$$

$$G(\langle 0, 1, 2, \ldots \rangle) = \sum_{k=0}^{\infty} k z^k = z(1-z)^{-2}$$

$$G(\langle 1, 2, 4, \ldots \rangle) = \sum_{k=0}^{\infty} 2^k z^k = \frac{1}{1-2z}$$

$$G(\left\langle \binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots \right\rangle) = \sum_{k=0}^{\infty} \binom{n}{k} z^k = (1+x)^n$$

I recommend the book by Wilf (Wilf, 2006) for learning about generating functions.

Data Structures

Know thy complexities! Below is a partial list from here.

Stacks

12a ⟨*Stack operations* 12a⟩≡

```
isEmpty :: Stack a -> Bool
push    :: a -> Stack a -> Stack a
pop     :: Stack a -> (a, Stack a)
top     :: Stack a -> a
```

Queues

12b ⟨*Queue operations* 12b⟩≡

```
enqueue :: a -> Queue a -> Queue a
dequeue :: Queue a -> (a, Queue a)
```

Lists

12c ⟨*List operations* 12c⟩≡

```
(++)   :: [a] -> [a] -> [a]
head   :: [a] -> a
last   :: [a] -> a
tail   :: [a] -> [a]
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
```

Let's use HASKELL notation to define types.

- "x :: a" says "x" is an expression of type "a"
- "f :: a -> b" says "f" is a function mapping expressions of type "a" to expressions of type "b"
- "g :: a -> b -> c" says "g" is a function mapping expressions of type "a" to functions of type "b -> c"

Rooted trees

Graphs

Please read the Appendix B.4 & B.5 : Graphs & Trees in the textbook (Corman et al., 2009).

Definition 5: Directed Graph

A directed graph *(*digraph*)* G *is a pair* $(\mathbb{V}, \mathbb{E})$*, where* $\mathbb{V}$ *is a finite set of* vertices *(or* nodes*), and* $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ *is a* binary relation*, called* adjacency *on* $\mathbb{V}$*. This relations defines the* edges *of* G*.*

Describe other types of graphs.

Hash tables

See Hash Tables in these notes.

Exercises

1. Prove that $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ for all integers n.
2. What is a formula for the nearest integer to a given real number x? In case of ties, when x is exactly halfway between two integers, give an expression that rounds (a) up — that is to $\lceil x \rceil$; (b) down — that is, to $\lfloor x \rfloor$.
3. Prove that
$$n = \lfloor n/m \rfloor + \lfloor (n+1)/m \rfloor + \cdots + \lfloor (n+m-1)/m \rfloor$$
4. Prove the *Dirichlet box principle*: If n objects are put into m boxes, some box must contain $\geq \lceil n/m \rceil$ objects, and some box must contain $\leq \lfloor n/m \rfloor$.
5. Estimate the size of $100!$.
6. $\lg n!$ is $O(g(n))$ for what function $g(n)$?
7. In how many ways can 6 people be arranged in a line? around a circle?
8. What is 11^4? Why is this number easy to compute, for a person who knows binomial coefficients?
9. For what value(s) of k is $\binom{n}{k}$ a maximum, when n is a given positive integer?
10. How many n-bit binary numbers have k bits sets to 1?
11. Prove the hexagon property
$$\binom{n-1}{k-1}\binom{n}{k+1}\binom{n+1}{k} = \binom{n-1}{k}\binom{n}{k-1}\binom{n+1}{k+1}$$

12. Define $\binom{n}{k}$ when n is a negative integer.

13. Show that $\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$

14. Show that
$$\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

15. Show that
$$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$$

16. Prove the following identities with harmonic numbers.

    (a) The sum of harmonic numbers:
    $$\sum_{k=1}^{n} H_k = (n+1)H_n - n$$

    (b) The sum of harmonics weighted by binomial coefficients:
    $$\sum_{k=m}^{n} \binom{k}{m} H_k = \binom{n+1}{m+1} \left(H_{n+1} - \frac{1}{m+1} \right)$$

17. Find a simple expression for the generating function of each of the following discrete numeric sequences.

    (a) $1, -2, 3, -4, 5, -6, \ldots$

    (b) $1, 2/3, 3/9, 4/27, \ldots, (n+1)/3^n, \ldots$

    (c) $1, 1, 2, 2, 3, 3, 4, 4 \ldots$

    (d) $0, 5, 50, \ldots, n5^n, \ldots$

18. Determine a discrete numeric sequence for the following generating functions.

    (a) $\frac{1}{1+2z}$

    (b) $\frac{1}{5-6z+z^2}$

    (c) $\frac{7z^2}{(1-2z)(1+3z)}$

    (d) $\frac{z}{(1+z)^2}$

    (e) $(1+z)^n + (1-z)^n$

19. What is the generating function $G(z)$ for the sequence t_i, where the sequence is given by
$$2, 6, 12, \ldots, (i+2)(i+1), \ldots$$
Write the generating function both as a series and in closed form.

20. Rank the following functions by order of growth. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg(n))$	$2^{\lg(n)}$	$(\sqrt{2})^{\lg(n)}$	n^2	$n!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	$\log(\log(n))$
$n * 2^n$	$\lg n$	1	$\ln n$	$2^{\lg n}$
e^n	$5n^3 + 4n^2$	$\sqrt{n^2}$	$\lg(2^n)$	$2^{\ln n}$
$4^{\lg n}$	$(n+1)!$	$n^2 + n + 3$	n	2^n

21. Define the iterated logarithm, $\lg^* n$, as follows.

$$\lg^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0 \\ \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0. \end{cases}$$

The iterated logarithm function is defined as

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

Compute:$\lg^* 2$, $\lg^* 4$, $\lg^* 16$, $\lg^* 65536$, $\lg^*(2^{65536})$

22. Evaluate

$$\sum_{i=3}^{8} i$$

23. Find a formula for

$$\sum_{i=m}^{n} i$$

24. Evaluate

$$\sum_{i=3}^{8} 2^i$$

25. Find a formula for

$$\sum_{i=m}^{n} 2^i$$

26. Find a formula for

$$\sum_{i=m}^{n} i2^i$$

27. Find a formula for

$$\sum_{i=0}^{\infty} a^i$$

28. Find a formula for

$$\sum_{i=0}^{\infty} ia^i$$

29. Find a formula for
$$\sum_{i=0}^{\infty} a^i/(i+1)$$

30. Riemann's zeta function $\zeta(k)$ is defined to be the infinite sum
$$\zeta(k) = 1 + \frac{1}{2^k} + \frac{1}{3^k} + \cdots = \sum_{j=1}^{\infty} \frac{1}{j^k}$$
Show that
$$\sum_{k=2}^{\infty} (\zeta(k) - 1) = 1$$

31. Let

n	0	1	2	3	4	5	6	$\cdots$
F_n	0	1	1	2	3	5	8	$\cdots$

be the Fibonacci sequence. Use mathematical induction on the variable k to prove that
$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$$
(Hint: Begin by showing the formula is true for $k = 1$ and $k = 2$.)

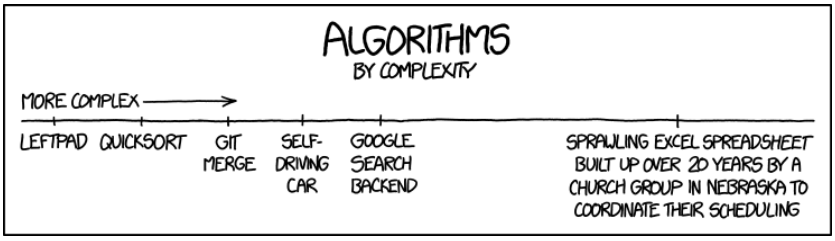
32. A Morse code message, sent by a telegraph, consists of a sequence of dots ($\cdot$) and dashes ($-$). A dot can be sent in 1 second, and a dash can be sent in 2 seconds. Thus, in 2 seconds two different messages *could* be sent ($\cdot\cdot$ or $-$).
    (a) How many different messages could be sent in 3 seconds?
    (b) How many different messages could be sent in 4 seconds?
    (c) Write a recurrence equation for the number of different messages that could be sent in n seconds.
    (d) How many different messages could be sent in n seconds?

33. What are the time complexities of the list operations on page 12? Answer the question for both an array and a linked list implementation.

2. Introduction

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid (Hofstadter, 1999)



Algorithms by Complexity

Goals

These are several lofty goals for those enroll in this course. If you study its content and complete its assignments, then you will be:

- Able to construct an algorithm that solves a problem
- Prove that the algorithm actually solves the problem
- Analyze resources consumed when an algorithm is executed
- Design optimal algorithms for the problem
- Express algorithms in a program language
- Verify the program is correctly implement the algorithm
- Collect statistics on the program's execution that verify the analysis.

What is an Algorithm?

An algorithm should have the following properties:

- It should proceed by discrete steps
- It should be deterministic (maybe not)
- Each step should be elementary
- It should be clear how to proceed from step to step
- It should have a finite description
- It should be possible to approximately implement using some technology
- It should be possible to duplicate the results

There are several formal models of what constitutes an algorithm. They are logically equivalent, but appear quite different on first inspection. A few computational models are:

- Turing machines are, perhaps, most well known
- Random access machines (RAM) correspond well to concept of a physical computer and imperative languages. This is the common model used to analyze algorithms.
- λ calculus is the formalism behind functional programming languages.

Algorithmic Problem Solving

Algorithms can often be classified by broad design problem solving paradigms. Basic problem solving techniques include:

- Brute force: Try every possible solution to find the best one.
- Divide and conquer: Divide a large problem into easier to solve smaller ones whose solutions can be combined to solve the large problem.
- Dynamic programming: Find globally optimal solutions by expressing a solution in terms of sub-problems whose answers have been previously computed and *memoized*.
- Greedy: (Hopefully) Find globally optimal solutions by making locally optimal choices.

The classic reference on problem solving is (Polya, 1945). In a nutshell, Polya's advice is: Understand the problem, devise a plan, carry out the plan, and look back at your answer.

Measuring Time and Space Complexity

Let n be the *size* of an algorithm's input data.

- The size may be the number of data items
- The size may be the number of bits needed to represent the data

To measure an algorithm's *time complexity* you must determine the number of *fundamental steps* required by the algorithm.

- You may want to count every instruction
- You may want to only count major instructions, e.g. compares, swaps, adds, multiplies, etc.
- You may want to *amortize* the time cost over a series of calls. Initial gains may be small, but over time gains may become larger.
- You may want to weigh an instruction count based on the number of bits needed to implement it.

A debt is amortized when a series of payments reduce the principal by increasing amounts.

The time complexity $T(n)$ is a function that counts these fundamental steps.

To measure an algorithm's *space complexity* you must determine the amount of auxiliary storage needed as the algorithm executes. The space complexity $S(n)$ is a functions that counts the memory requirements of an algorithm.

Best, Worst, Average, and Amortized Time Complexity

Like children, an algorithm can sometimes act well, sometimes poorly, and usually performs on an even keel. A problem P of size n will have many *instances*, for example, the decision problem "Are the n integers in a list sorted?" has numerous instances. Every list of n integers is an instance.

Let $I_0, I_1, \ldots, I_k$ denote all size n instances of problem P. Let $T_0, T_1, \ldots, T_k$ denote the times required to solve the instances $I_0, I_1, \ldots, I_k$ using algorithm A.

The number of instances may be infinite, but let's assume it is countable. In fact, let's assume it is finite.

- The *worst case* time complexity of algorithm A is

$$T_w(n) = \max\{T_0, T_1, \ldots, T_k\}$$

  Most of the time we are interested in the worst case behavior of an algorithm (we expect and plan for the worst to happen)

- The *best case* time complexity of algorithm A is

$$T_b(n) = \min\{T_0, T_1, \ldots, T_k\}$$

  The best case behavior is usually of little interest (we don't expect the best to happen often)

- The *average case* time complexity of algorithm A is

$$T_a(n) = \sum_{i=0}^{k} p_i T_i$$

where p_i is the probability of instance I_i occurring. For example, if each instance has equally likely time complexity, then

$$T_a(n) = \frac{1}{k+1} \sum_{i=0}^{k} T_i$$

- The *amortized* time complexity of algorithm A is the average of its (worst case) running times over a sequence of inputs.

$$T_{am}(n) = \frac{1}{n} \sum_{j=0}^{n-1} T_{\pi(j)}$$

where $I_{\pi(j)}$, $j = 0, \ldots, (n-1)$ is some sequence of instances. The idea is that instances can become easier to solve as the algorithm is repeatedly executed.

Models of Computation

Turing machines are one abstract model of computing any computable function.

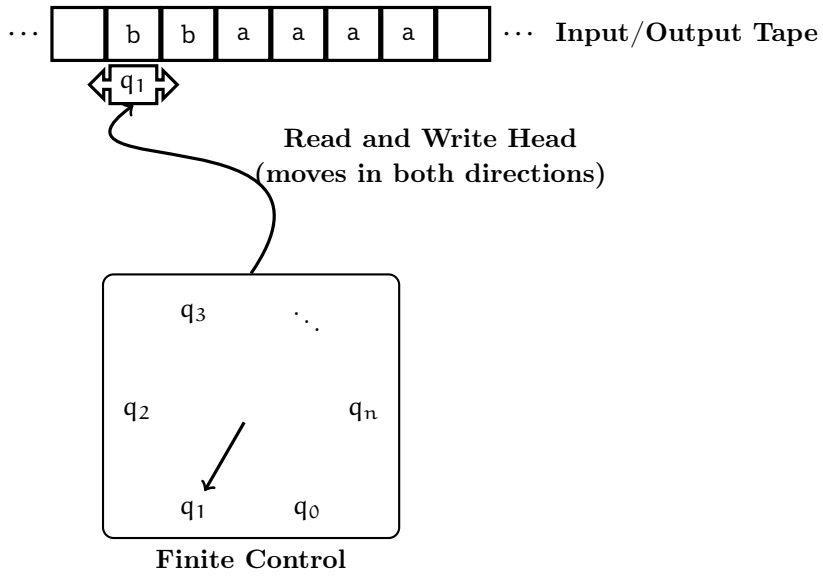


Figure 1: Sketch of Turing machine by Sebastian Sardina

Church's λ calculus is another system equivalent in computing power to Turing machines. The basic ideas in λ calculus are simple, you can:

- β reductions: Substitute a value for a variable in a function's definition. The notation

$$(\lambda\ x.f)s$$

says, for every x that occurs (bound) in expression f, substitute s for x. The notation

$$(\lambda\ x.f)s \overset{\beta}{\mapsto} f(x := s)$$

is used to represent a β substitutions.

- α reductions: Rename variables while avoiding conflicting names. α reductions are useful for name resolution in programming languages with static scope. The precise rules for α reductions are non-trivial, beyond the scope of these notes. As an analogy, in calculus you learn *change of variables* rules that are useful. α reductions are similar.

- η reductions: Eliminate unused expressions. The notation used to represent η reductions is:

$$(\lambda\ x.f)s \overset{\eta}{\mapsto} f \quad \text{when } f \text{ does not contain } x \text{ as a free variable}$$

Random access machines (RAM) are a third computational model equivalent to Turing machines and the λ-calculus. They seem to be a more concrete computational models because they map naturally to imperative programming languages. A RAM consists of:

- a read-only input tape from which integers may be read
- a write-only output tape on which integers may be written
- an unbounded memory
- one accumulator
- one control unit where a RAM's program is stored

RAM instructions are not permitted to modify themselves. Memory can hold integers of arbitrary sizes. Data can be addressed in multiple modes, e.g., immediate, direct, and indirect.

Explain memory addressing methods.

Sample Instructions for a RAM

Here are instructions that could be defined for a RAM. They are similar to instructions you find in assembly-level languages.

Op Code	Op Name	Address
1	LOAD	operand
2	STORE	operand
3	ADD	operand
4	SUB	operand
5	MULT	operand
6	DIV	operand
7	READ	operand
8	WRITE	operand
9	JUMP	address
10	JGTZ	address
11	WRITE	address
12	HALT	

Time and Space Complexity for the RAM Model

To specify the time and space complexity, we must specify the time required to execute each instruction and the space used by each register.

- The *uniform cost model* treats each instruction as requiring 1 unit of time and each register as 1 unit of space
- The *logarithmic cost model* assumes that the time and space required for an operation depends on the size of the operand(s), where size is measured by the number of bits needed to represent the number

We will almost always use the uniform cost model. We will not write algorithms in a low-level RAM language, instead we rely on the fact that compilers allow us solve problems at a higher level of abstraction.

Implementations

The techniques used to implement data structures and algorithms has a profound effect on their run-time statistics. Issues such as hardware, operating systems, sequential, concurrent, parallel, or distributed execution, compilers, optimization's, and programming languages all change the real world behavior of an algorithm.

The Computer Language Benchmark Game is a collection of interesting benchmarks written in many languages. They compare timing results to a C "standard implementation." There are many ideas presented on that site where you can delve into practical matters related to code performance.

I write code using noweb, a literate programming system developed by Norman Ramsey (Ramsey, 1994). Literate programming is an idea championed by Knuth (Knuth, 1984). It is simple idea: Write programs people want to read. I cannot claim to have reached this

Good old fashion computers, whose architectures are widely known, see (Patterson and Hennessy, 1996), are considered here. Newfangled machines based on optical, quantum, or DNA computing are beyond scope.

he noweb source for these notes contains document chunks and program chunks. The source can be woven into this document or tangled into a program.

I have not compiled, tested, nor verified much of the code in these notes. You will find errors. I welcome fixes.

Haskell

goal. However, I would like to promulgate the idea to others who can. Noweb is a neat idea because it supports writing in almost any programming language.

The code in these notes is written in either C, JAVA, HASKELL or Pseudocode. I recommend (Kernighan and Ritchie, 1988), (Sedgewick, 2004), and (Lipovaca, 2011) as language references. The C programming language is used because it exposes much of the architecture of common computers today. C maps naturally to developed algorithm analysis concepts. And, just as importantly, it is the root of a tree of many widely used programming languages. The JAVA programming language is an offshoot of C. It builds object-oriented programming ideas into a C-like syntax. Both C and JAVA are imperative languages: They instruct state changes as the machine completes its calculation. In all but trivial problems possible state paths grow exponentially. Keeping track of state is often intricate if not intractable.

Another programming language model is provided by HASKELL: A functional programming language. Today's computing students almost always learn imperative programming first. I strongly believe functional programming has many advantages and will almost always be the model of choice for expressing higher-level computing abstractions. Let me recommend (Rabhi and Lapalme, 1999) as one source for analysis of functional algorithms. I believe further research into algorithm analysis using pure functional languages with lazy evaluation is necessary.

Finally, some of the algorithms in my notes are written using pseudocode taken more or less directly from the textbook (Corman et al., 2009). In any event, when you write code your source file should have a standard header, something like this, from Haskell programming guidelines.

Listing 1: A Sample Header for Code

23 ⟨*Header for a code project* 23⟩≡

```
{- |
Module      :  <File name replaced automagically>
Description :  <Short text for contents page>
Copyright   :  <(c) Authors or Affiliations>
License     :  <License or use requirements>

Maintainer  :  <Name and email address>
Stability   :  <unstable|provisional|stable|frozen>
Portability :  <portable|non-portable (<reason>)>
Sources     :  <Giants on whose shoulders you stood>
-}
```

I've added the Sources line. I believe it is important to recognize those from whom you have gained. In school and elsewhere, it can be crucial not to plagiarize. I try not to. It is hard to always acknowledge

"If I have seen further it is by standing on the shoulders of Giants." Issac Newton in a 1676 letter to Robert Hooke.

those from whom you have learn.

3. Algorithmic Design

This section is about *algorithmic design*. To gain a deeper understanding, read Jon Bentley's Programming Pearl "Algorithm Design Techniques," (Bentley, 1984) and §4.1 The maximum sub-array problem, in the text (Corman et al., 2009).

Maximum Subsequence Sum Problem

Consider the Dow Jones Industrial Average: It goes up and down daily. What contiguous run of days has the highest gain? Consider your weight: It goes up and down daily. What contiguous run of days has the largest weight gain or loss?

Pretend you are given sequence of integers, say

$$X = [-1, -2, 3, 5, 6, -2, -1, 4, -4, 2, -1] \quad \text{of length } n = 11.$$

By inspection you may notice the largest gain is 15 over the (contiguous) subsequence

$$[3, 5, 6, -2, -1, 4]$$

Let's design some algorithms that solves the maximum subsequence sum problem.

Problem 1: Maximum Subsequence Sum

Given a list of integers $X[k]$, $k = 0, \ldots, (n-1)$, $n \geq 0$, *find the maximal value of*

$$\sum_{k=s}^{e} X[k] \quad \text{for } 0 \leq s \leq e \leq (n-1).$$

In case all values in X *are negative, the maximum subsequence sum is* 0*, from the empty subsequence.*

The empty sequence $[]$ is a contiguous subsequence of X and the sum of elements in $[]$ is 0.

For $n = 11$ there are 11 sums with 1 term, 10 sums with 2 terms, 9 sums with 3 terms, ..., and 1 sum with 11 terms.

Computing these sums can take up to

$$11 \cdot 1 + 10 \cdot 2 + 9 \cdot 3 + \cdots + 1 \cdot 11 = 286$$

additions.

$$\sum_{k=0}^{n} (n-k)k = \frac{(n+2)(n+1)n}{6} = O(n^3)$$

Brute Force

The brute-force approach computes the sum of every possible subsequence and remember the largest.

Listing 2: Cubic algorithm

26a ⟨*Cubic time maximum subsequence sum* 26a⟩≡
```
int maxSubseqSum(int X[], int n) {
  int MaxSoFar =  0; // local state
  ⟨For each start of a subsequence 26b⟩ {
    ⟨For each end of the subsequence 26c⟩ {
    int Sum = 0; // More local state
        ⟨For each subsequence 26d⟩ {
          ⟨Compute partial sum; Check MaxSoFar 26e⟩
      }
    }
    return MaxSoFar;
  }
}
```

The start of a sequence ranges from the first (0) to the last (n-1) index.

26b ⟨*For each start of a subsequence* 26b⟩≡
```
for (int start = 0; start < n; start++)
```

The end of a sequence ranges from the first start to the last (n-1) index.

26c ⟨*For each end of the subsequence* 26c⟩≡
```
for (int end = start; end < n; end++)
```

Compute each partial sum, keeping track of the maximum seen so far.

26d ⟨*For each subsequence* 26d⟩≡
```
for (int k = start; k <= end; k++)
```

26e ⟨*Compute partial sum; Check* MaxSoFar 26e⟩≡
```
Sum = Sum + X[k];
MaxSoFar = (Sum > MaxSoFar) ? Sum : MaxSoFar;
```

The time complexity of this brute-force algorithm is $O(n^3)$, as can be seen by computing the expression

$$T(n) = \sum_{s=0}^{n-1} \sum_{e=s}^{n-1} \sum_{k=s}^{e} c$$

In listing 2, the cost of ⟨*Compute partial sum; Check* `MaxSoFar` 26e⟩ take constant time, call this constant c. Therefore, the inside `for` loop on `k` starting has time complexity

$$\sum_{k=s}^{e} c = (e - s + 1)c$$

Next, the time complexity of the middle `for` loop on e is modeled by the sum

As e goes from s to $n-1$ the value $k = (e - s + 1)$ goes from 1 to $n - s$.

$$\sum_{e=s}^{n-1} (e - s + 1)c = \sum_{k=1}^{n-s} kc = c\frac{(n - s + 1)(n - s)}{2}$$

Finally, the time complexity of the outer `for` loop on s can be computed by

$$\begin{aligned}\sum_{s=0}^{n-1} c\frac{(n - s + 1)(n - s)}{2} &= c\sum_{s=0}^{n-1} \binom{n - s + 1}{2} \\ &= c\binom{n + 2}{3} \\ &= c\frac{(n + 2)(n + 1)n}{6}\end{aligned}$$

A Linear Time Algorithm

Suppose we've solved the problem for `x[0..(k-1)]`. How can we extend that to a solution for `x[0..k]`? The maximum sum in the first `k` elements is either the maximum sum in the first `k-1` elements, which we'll call `MaxSoFar`, or it is the subsequence that ends in position `k`.

Listing 3: Linear time/constant space algorithm

27 ⟨*Imperative linear time algorithm* 27⟩≡

```
int maxSubseqSum(int x[], int n) {
  ⟨Local state 28a⟩

  for (⟨Every end position 28b⟩) {
```

```
    ⟨Compute the maximum that ends here 28c⟩
    ⟨Compute the maximum so far 28d⟩
  }
  return MaxSoFar;
}
```

We need to keep track of the maximum so far and the maximum that ends at some position. Both can be initialized to 0.

28a ⟨*Local state* 28a⟩≡
```
int MaxSoFar      =  0;
int MaxEndingHere =  0;
```

Let a dummy index k iterate from the start to the end of the sequence.

28b ⟨*Every end position* 28b⟩≡
```
(int k = 0; k < n; k++)
```

The maximum at position k is the maximum at k-1 plus x[k], unless that sum is less than 0. In that case, reset MaxEndingHere to 0.

28c ⟨*Compute the maximum that ends here* 28c⟩≡
```
MaxEndingHere = max(0, MaxEndingHere + x[k]);
```

Then the maximum at this point is the maximum so far or the maximum that ends here, whichever is larger.

28d ⟨*Compute the maximum so far* 28d⟩≡
```
MaxSoFar = max(MaxSoFar, MaxEndingHere);
```

Functional Implementation

This functional implementation takes a list [a] and returns the maximum subsequence sum and the subsequence that *witnesses* it.

Two helper functions are useful. A helper function snd, returns the second element in a pair.

Listing 4: The second of a pair

28e ⟨*The second of a pair function* 28e⟩≡
```
snd       :: (a,b) -> b
snd (x,y) = y
```

The other helper function is foldl. It applies a function to an initial value and a list to *recursively reduce* the list to a value. For example, the sum and product functions can be defined by folding lists.

Listing 5: The foldl functions

29a ⟨*Folding a from the left* 29a⟩≡

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

sum [a]     = foldl (+) 0 [a]
product [a] = foldl (*) 1 [a]
```

Let's define a function `f` that acts on two ordered pairs and a value `x` and returns an order pair. Each ordered pair contains a value and a list. The first ordered pair is the value and the list of the maximum to here. The second ordered pair is the value and the list of the maximum so far. The value `x` is the next value in the sequence. (The code was found on Rosetta Code. In fact, it does not execute in linear time as claimed. There is a simple fix. Do you see it?)

Listing 6: Functional algorithm for maximum subsequence sum

29b ⟨*Linear time algorithm* 29b⟩≡

```
maxsubseq :: (Ord a, Num a) => [a] -> (a, [a])
maxsubseq = snd . foldl f ((0,[]),(0,[])) where
  f ((maxToHere,witnessToHere),sofar) x = (a,b) where
    a = max (0,[]) (maxToHere+x, witnessToHere++[x])
    b = max sofar a
```

Exercises

1. Complete project 1.
2. As an exercise, show that

$$\sum_{k=0}^{n}(n-k)k = \frac{(n+1)n(n-1)}{6} = \binom{n+1}{3}$$

   Where was this identity used in the notes?

3. What is the solution to the recurrence equation

$$T(n+1) = T(n) + n \quad \text{with initial condition } T(1) = 1$$

4. What is the solution to the recurrence equation

$$T(n+1) = T(9n/10) + n \quad \text{with initial condition } T(1) = 1$$

4. Euclid, Fibonacci, Lamé & Lucas

All the problems of the world could be settled easily, if men were only willing to *think*.

Thomas Watson Sr.

Please read §31.2 Greatest Common Divisor in the textbook (Corman et al., 2009).

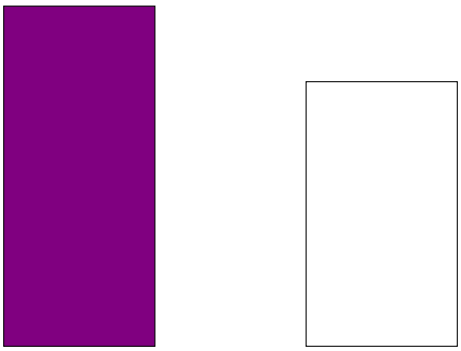
Wikipedia states one flagon is about 1.1 liters. My exchange range is made up for its simple arithmetic properties.

This experiment may be more fun if you drink the wine while performing it!

Imagine riding a time machine back to 457 BC. Traveling around this long-ago world you'll need to exchange money. Perhaps you'll need to convert lengths, volumes, and weights from modern to ancient units. For illustration, consider calculating the conversion factor from liters to flagons. Pretend 9 liters is equivalent to 7 flagons, but we don't know this yet! We have to calculate it. The Euclidean algorithm provides an efficient way to calculate conversion ratios. It finds a common measure (the greatest common divisor) for liters and flagons and uses this common unit to measure both.

$$\begin{aligned} 9 \text{ liters} &= 7 \text{ flagons} \\ 1 \text{ liter} &= 0.7777\cdots \text{ flagons} \\ 1.285714\cdots \text{ liters} &= 1 \text{ flagon} \end{aligned}$$

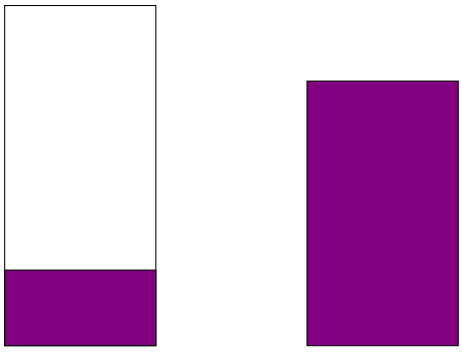
View the stylized drawing below: It show a flagon full of wine on the left and an empty liter on the right.



Full flagon Empty liter

When the wine is poured into the liter, a residue remains in the flagon. We'd like to know what fraction of a liter is left.

A flagon is one liter plus a small residue.

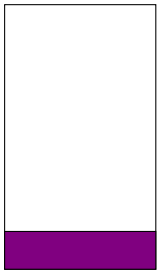


Flagon-residue Full liter

So, pour the full liter into containers each holding the amount remaining in the flagon. This takes three containers and leaves a smaller residue.

A liter is three flagon residues plus a smaller residue.

Flagon-residue Flagon-residue Flagon-residue



Liter-residue

A flagon residue is two liter residues with nothing remaining.

Now, we'd like to know the fraction of the flagon-residue is this liter-residue. So, we pour the flagon-residue into a containers each equal to the amount of the liter-residue. This takes two containers and leaves no residue.

Empty-flagon-residue Liter-residue Liter-residue

To recapitulate, a liter residue is a common measure for liters and flagons. A liter is 7 liter residues; A flagon is 9 liter residues;

$$\begin{aligned} 1 \text{ flagon} &= 1 \text{ liter} + \text{ flagon-residue} & 9 &= 7 + 2 \\ 1 \text{ liter} &= 2 \text{ flagon-residue} + \text{ liter-residue} & 7 &= 2 \cdot 3 + 1 \\ 1 \text{ flagon-residue} &= 2 \text{ liter-residue} & 2 &= 1 \cdot 2 + 0 \end{aligned}$$

Now, share the wine with your friends as you do the math. To compute the ratio $9 : 7$, run the above equations backwards. From the last equation,

$$1 \text{ flagon-residue} = 2 \text{ liter-residue}.$$

and

$$1 \text{ liter-residue} = \tfrac{1}{2} \text{ flagon-residues}$$

Therefore,

$$1 \text{ liter} = 3 \text{ flagon-residue plus } 1 \text{ liter-residue} = \tfrac{7}{2} \text{ flagon-residues}$$

and

$$1 \text{ flagon-residue} = \tfrac{2}{7} \text{ liters}$$

Finally,

$$1 \text{ flagon} = 1 \text{ liter plus } 1 \text{ flagon-residue} = \tfrac{9}{7} \text{ liters}$$

A theorem helps to explain the algorithm.

Theorem 4: Euclidean division

Given two integers a and m, with $m \neq 0$, there exist unique integers q and r such that

$$a = mq + r \quad \text{and} \quad 0 \leq r < |m|.$$

The dividend a equals the divisor m times the quotient q plus the remainder r.

Here's how the Euclidean algorithm computes $\gcd(34, 21)$ occurs.

In example , see how the divisor m and remainder r values shift down and left (southwest) at each step.

Example: Compute $\gcd(34, 21)$

Dividend a	$=$	*Divisor* m	$\cdot$	*Quotient* q	$+$	*Remainder* r
34	$=$	21	$\cdot$	1	$+$	13

The last divisor, where the remainder is 0, is the greatest common divisor. In this case, $\gcd(34, 21) = 1$.

Definition 6: Greatest Common Divisor

The greatest common divisor of two integers a and m is the

largest integer that divides them both.

$$\gcd(a, m) = \max\{k \mid k \backslash a \text{ and } k \backslash m\}$$

The Euclidean algorithm iterates the Euclidean division equation using the recursion: Let $r_0 = a$ and $r_1 = m > 0$, and assume $m \leq a$. Euclid's algorithm computes

$$\begin{aligned}
r_0 &= r_1 q_1 + r_2 && 0 \leq r_2 < r_1 \\
r_1 &= r_2 q_2 + r_3 && 0 \leq r_3 < r_2 \\
&\vdots = \vdots \\
r_{n-2} &= r_{n-1} q_{n-1} + r_n && 0 \leq r_n < r_{n-1} \\
r_{n-1} &= r_n q_n
\end{aligned}$$

The iteration halts when $r_{n+1} = 0$, and the last divisor (non-zero remainder) r_n is the greatest common divisor of a and m

Coding the Euclidean algorithm

The code for the Euclidean algorithm is often based on the identity in theorem 5.

Theorem 5: Greatest Common Divisor Recurrence

Let $0 \leq m < a$. Then,

$$\begin{aligned}
\gcd(a, 0) &= a \\
\gcd(a, m) &= \gcd(m, a \bmod m), \quad \text{for } m > 0
\end{aligned}$$

In C, the code might look something like this:

Listing 7: Imperative GCD algorithm

34a ⟨*Imperative GCD algorithm* 34a⟩≡

```
int gcd(int a, int m) {
  ⟨GCD local state 34b⟩
  while (⟨The divisor m is not 0 35b⟩) {
    ⟨Move m to a and a mod m to m 35a⟩
  }
  ⟨Return the absolute value of a 35c⟩
}
```

To change the values: m goes to a, and $a \bmod m$ goes to m, a local temporary value `t` is used.

34b ⟨*GCD local state* 34b⟩≡

```
int t;
```

35a ⟨*Move* m *to* a *and* a mod m *to* m 35a⟩≡
```
t = a;
a = m;
m = t % a;
```

C is not type safe. Instead of a Boolean test (m == 0) you can use the (wrong type) integer m itself.

35b ⟨*The divisor* m *is not 0* 35b⟩≡
```
m
```

The greatest common divisor is a positive integer. So, just in case the negative value was computed, change the answer to the absolute value of a before returning it.

35c ⟨*Return the absolute value of* a 35c⟩≡
```
return a < 0 ? -a : a;
```

Function GCD algorithm

Here's a functional implementation written in Haskell. It is from the standard Prelude for Haskell. It uses the idea that there is no largest integer that divides 0: All integers divide 0. Therefore, gcd(0, 0) is undefined and raises an error.

HASKELL supports elegant methods for handling errors, but here we just raise our hands and give up.

Listing 8: Functional GCD algorithm

36 ⟨*Functional GCD algorithm* 36⟩≡

```
gcd     :: (Integral a) => a -> a -> a
gcd 0 0 =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y =  gcd' (abs x) (abs y) where
  gcd' a 0  =  a
  gcd' a m  =  gcd' m (a `rem` m)
```

Analyzing the Euclidean algorithm

It is fitting that this early algorithm was also one of the first to have its complexity analyzed (Shallit, 1994). The result is called Lame's theorem, which incorporates the Fibonacci sequence, later widely popularized by Édouard Lucas.

The complexity of the Euclidean algorithm can be measured by the number of quotient–remainder steps taken. Seven steps are taken when computing gcd(34, 21), (see example).

The time complexity of the Euclidean algorithm is reduced least when each quotient is 1, except the last. For instance, when the greatest common divisor is 1, the last quotient is 2, and all other quotients are 1, terms in the Fibonacci sequence is produced. Running the Euclidean algorithm equations backwards, see Fibonacci sequence:

Recall, the Fibonacci sequence is

$$\vec{F} = \langle 0, 1, 1, 2, 3, 5, 8, 13, \ldots \rangle$$

and indexed from 0, that is $f_0 = 0$.

$$\begin{aligned} 2 &= 1 \cdot 2 + 0 \\ 3 &= 2 + 1 \\ 5 &= 3 + 2 \\ 8 &= 5 + 3 \\ 13 &= 8 + 5 \\ &\vdots \quad \vdots \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 3. \end{aligned}$$

Here's another Fibonacci-like example that show the worst-case time complexity of the Euclidean algorithm. In this case, the greatest com-

mon divisor is 2 and the last quotient is 5.

$$\begin{aligned} 10 &= 2 \cdot 5 + 0 \\ 12 &= 10 + 2 \\ 22 &= 12 + 10 \\ 34 &= 22 + 12 \\ &\vdots \\ 10F_n + 2F_{n-1} &\text{for } n \geq 3. \end{aligned}$$

In the general case, when g is the greatest common divisor and q is the last quotient and all other quotients are 1, the steps come from a Fibonacci-like sequence.

$$\begin{aligned} gq &= g \cdot q + 0 \\ gq + g &= gq + g \\ 2gq + g &= (gq + g) + gq \\ 3gq + 2g &= (2gq + g) + (gq + g) \\ 5gq + 3g &= (3gq + 2g) + (2gq + q) \\ gqF_n + gF_{n-1} &\text{for } n \geq 3. \end{aligned} \qquad \vdots \qquad \vdots$$

The asymptotic growth of F_n is $O(\varphi^n)$ where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio. Therefore, when m and a are consecutive terms in a Fibonacci-like sequence the Euclidean algorithm takes n steps where

$$n = O(\log_\varphi(a))$$

Theorem 6: Lamé's Theorem

Let $a, m \in \mathbb{Z}^+$ with $a \geq m > 0$. Let n be the number of quotient–remainder steps in Euclidean algorithm. Then

$$n \leq 1 + 3 \lg m$$

Proof: Lamé's Theorem

Let $r_0 = a$ *and* $r_1 = m$. *Euclid's algorithm computes*

$$\begin{aligned} r_0 &= r_1 q_1 + r_2 \qquad 0 \le r_2 < r_1 \\ r_1 &= r_2 q_2 + r_3 \qquad 0 \le r_3 < r_2 \\ &\vdots \\ r_{n-2} &= r_{n-1} q_{n-1} + r_n \qquad 0 \le r_n < r_{n-1} \\ r_{n-1} &= r_n q_n \end{aligned}$$

using n *divisions to compute* $r_n = \gcd(a, m)$. *Note that*

- $q_i \ge 1$, $i = 1, 2, \ldots, (n-1)$
- $(r_n < r_{n-1}) \Rightarrow (q_n \ge 2)$

Let F_i *denote the* i^{th} *Fibonacci number. Then*

$$\begin{aligned} r_n &\ge 1 = F_2 \\ r_{n-1} &= r_n q_n \ge 2r_n \ge 2 = F_3 \\ r_{n-2} &\ge r_{n-1} + r_n \ge F_3 + F_2 = F_4 \\ &\vdots \\ r_2 &\ge r_3 + r_4 \ge F_{n-1} + F_{n-2} = F_n \\ r_1 &\ge r_2 + r_3 \ge F_n + F_{n-1} = F_{n+1} \end{aligned}$$

Using the growth rate of the Fibonacci numbers $F_{n+1} \approx \varphi^{n-1}$, *we find*

$$m = r_1 \ge F_{n+1} > \varphi^{n-1}$$

Take the logarithm base φ *of both sides and use the change of base formula for logarithms to derive the inequality*

$$\log_\varphi m = \frac{\lg m}{\lg \varphi} > n - 1$$

Since $(\lg \varphi)^{-1} < 3$ *we have*

$$3 \lg m > \frac{\lg m}{\lg \varphi} > n - 1$$

Another way to state the result is that if m can be represented in k bits, then the number of divisions in Euclid's algorithm is less than 3 times the number of bits in m's binary representation.

Application

The least common multiple (lcm) is a number related to the greatest common divisor (gcd). You've learned of it, if not by name, when learning to add fractions: To add $5/3$ and $8/5$ use a common denominator. In this case $3 \cdot 5 = 15$.

Note the cross multiplication and addition of numerators and denominators: $5 \cdot 5 + 3 \cdot 8$.

$$\frac{5}{3} + \frac{8}{5} = \frac{25}{15} + \frac{24}{15} = \frac{49}{15}.$$

Coding the extended Euclidean algorithm

Bézout's identity provides the link between the greatest common divisor and solving linear congruence equations.

Theorem 7: Bezout's identity

Let $0 < m \leq a$. Then, there exists integers s and t such that

$$\gcd(a, m) = at + ms$$

That is, the greatest common divisor $\gcd(a, m)$ can be written as a *linear* combination of a and m.

Proof: Bezout's identity

Let $\mathbb{L} = \{ax + my > 0 : x, y \in \mathbb{Z}\}$ *be the set of all positive* linear combinations *of* a *and* m*, and let*

$$d = \min\{ax + my > 0 : x, y \in \mathbb{Z}\}$$

be the minimum value in $\mathbb{L}$*, Let* t *and* s *be values of* x *and* y *that give the* minimum *value* d*. That is,*

$$d = at + ms > 0$$

Let a *divided by* d *give quotient* q *and remainder* r*.*

$$a = dq + r, \quad 0 \leq r < d$$

Then

$$\begin{aligned} r &= a - dq \\ &= a(1 - tq) + m(sq) \\ &\in \{ax + my \geq 0 : x, y \in \mathbb{Z}\} \text{ and } 0 \leq r < d. \end{aligned}$$

But since d *is the smallest positive linear combination,* r *must be* 0 *and* d *divides* a*. A similar argument shows* d *divides* m*. That is,* d *is a common divisor of* a *and* m*. Finally, if* c *is any common divisor of* a *and* m*, then* c *divides* $d = at + ms$*. That is,* d *is the greatest common divisor of* a *and* m*.*

I found the code for the extended Euclidean algorithm here. The original is by by Trevor Dixon.

Listing 9: Haskell Extended Euclidean Algorithm

40 ⟨*Haskell Extended Euclidean algorithm* 40⟩≡

```
extendedEu :: Integer -> Integer -> (Integer, Integer)
extendedEu a 0 = (1, 0)
extendedEu a m = (t, s - q * t)
  where (q, r) = quotRem a m
        (s, t) = extendedEu m r
```

Exercises

1. Prove theorem 4.
2. What is the time complexity of the algorithm in listing 9.
3. Show that $\mathrm{lcm}(a, m)\gcd(a, m) = am$.
4. Write an least common multiple (lcm) algorithm in HASKELL and C.

5. Numerics

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

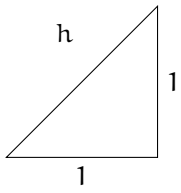
John Tukey

Much of early computing revolved around numerical algorithms that approximate values really sought. Ancient records show algorithms to approximate the value of $\sqrt{2}$, π and other useful numbers. Algorithms that compute numbers like $\sqrt{2}$ or π cannot terminate: There is no finite positional notation for these numbers. However, these algorithms can be terminated once a computed value is determined to be *good enough*. To illustrate numerical algorithms, let's develop Newton's method for computing $\sqrt{m}$.

Newton's Method

Although Newton's name is attached to this method, initial knowledge of it was known to ancient mathematicians of Mesopotomia, the region of modern day Iraq and Iran. Clay tablets dated from around 1800 B.C. to 1600 B.C. have been found in Mesopotomia that show how to approximate $\sqrt{2}$ and perform other arithmetic operations.

These early mathematicians considered an isosceles right triangle with legs of length 1



They knew from the Pythagorean theorem theorem that

$$1^2 + 1^2 = h^2$$

so the *hypotenuse* h has length $h = \sqrt{2} \approx 1.41421356\cdots$ The Babylonians knew how to approximate the value of $h = \sqrt{2}$ to many decimals places. They used a sexigesimal notation. Their calculations indicate this is what they did:

In Western society, the method presented here to compute $\sqrt{m}$ is called Newton's method. However, it was known in many cultures prior to Newton. He did generalize the idea to a broad class of functions, not just $f(m) = x^2 - m$.

The idea is: Let x be a zero of function f. Use Taylor's theorem. Solve for x, and discard the second order error.

$$0 = f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(\xi)}{2}(x - x_k)^2$$

$$x = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{f''(\xi)}{2f'(x_k)}(x - x_k)^2$$

$$x \approx x_k - \frac{f(x_k)}{f'(x_k)}$$

The found artifacts showing computational dexterity date from the time when the Babylonians lived in Mesopotomia.

- Start with $h_0 = 1$ as an initial approximation to $\sqrt{2}$
- Clearly 1 is too small, as the Babylonians could easily measure
- But, if $1 = \sqrt{2}$, then $1 \cdot 1 = \sqrt{2} \cdot \sqrt{2} = 2$ and so $2/1 = \sqrt{2}$
- As it is, $2/1$ is too large
- The average of the under estimate 1 and the over estimate $2/1$ provides a better approximation to $\sqrt{2}$, call this

$$h_1 = \frac{1}{2}\left(h_0 + \frac{2}{h_0}\right) = \frac{1}{2}\left(1 + \frac{2}{1}\right) = \frac{3}{2}$$

- But $h_1 = 3/2$ is too large, as the Babylonians could measure
- But, if $3/2$ were the exact square root, then $2/(3/2) = 4/3$ would equal $\sqrt{2}$
- As it is, $4/3 \approx 1.333\cdots$ is too small
- The average of the over estimate $3/2$ and the under estimate $4/3$ will provide a better approximation

$$h_2 = \frac{1}{2}\left(h_1 + \frac{2}{h_1}\right) = \frac{1}{2}\left(\frac{3}{2} + \frac{4}{3}\right) = \frac{17}{12} \approx 1.41166\cdots$$

- But $h_2 = 17/12$ is too small
- The Babylonians carried out this iteration more times computing the $\sqrt{2}$ accurately to at least 9 decimal places
- That is, they next computed the average of $h_2 = 17/12$ and $2/h_2 = 24/17$

$$\begin{aligned} h_3 &= \frac{1}{2}\left(h_2 + \frac{2}{h_2}\right) \\ &= \frac{1}{2}\left(\frac{17}{12} + \frac{24}{17}\right) \\ &= \frac{1}{2}\left(\frac{289 + 288}{17 \times 12}\right) \\ &\approx 1.41421568628\cdots \end{aligned}$$

To generalize the $\sqrt{2}$ method, pretend you want to compute $\sqrt{m}$. This is equivalent to computing a solution to the equation

$$x^2 - m = 0$$

Consider the recurrence equation

$$x_k = \frac{x_{k-1} + m/x_{k-1}}{2}, \quad k \geq 1 \tag{1}$$

Given an initial value x_0, equation 1 can be used to generate a sequence

$$\langle x_0, x_1, x_2, \ldots \rangle .$$

If you pretend that x_k converges to x as k goes to infinity, that is,

$$\lim_{k \to \infty} x_k = x$$

Then x satisfies the equations

$$\begin{aligned} x &= \frac{x + m/x}{2} \\ 2x &= x + m/x \\ x &= m/x \\ x^2 &= m \\ x &= \sqrt{m} \end{aligned}$$

What happens when m is negative?

A first step in implementing Newton's method for computing $\sqrt{m}$ is to define the function that maps m and x_{k-1} to the next value x_k.

Listing 10: Newton's square root recurrence

43a ⟨*Newton's square root recurrence* 43a⟩≡

```
next :: Double -> Double -> Double
next m 0 = error "Division by zero"
next m x = (x + m/x)/2
```

We want to repeatedly apply `next` to some initial value and generate a list of `Doubles`. Let's define `repeatedly` to be a function that applies a function `f :: Double -> Double` to itself repeatedly. An initial value (seed) `a` for `f` starts the iteration, generating an infinite list.

Listing 11: Repeatedly Apply a Function

43b ⟨*Repeatedly apply a function* 43b⟩≡

```
repeatedly :: (Double -> Double) -> Double -> [Double]
repeatedly f a = a : repeatedly f (f a)
```

Although `repeatedly` does not terminate, it can be terminated once a computed value is *close enough*. A common way to do this is to define a tolerance usually the machine epsilon and declare that the last computed approximation is *good enough* once it and the previous approximation are within the tolerance.

The *absolute difference* $|x_k - x_{k-1}|$ between succssive iterates is a measure of closeness. One way to terminate Newton's iteration is to stop when the absolute difference is within the tolerance.

$$|x_k - x_{k-1}| \leq \tau$$

Computer arithmetic on floating point numbers is not exact. The absolute difference can be small because the numbers x_k and x_{k-1} themselves are small. The absolute difference may never be small because the numbers themselves are large.

Instead of computing until the difference of successive approximations approaches 0, it is often better to compute until the ratio of successive approximations approach 1. This measure of closeness is the *relative difference* $|x_{k-1}/x_k - 1|$. Newton's method terminates when

$$\left|\frac{x_{k-1}}{x_k} - 1\right| \leq \tau$$

The `relative` function maps a tolerance τ and a sequence to the first value in the sequence where the relative error is within tolerance.

Some define the relative error as

$$re_k = \frac{x_k - x}{x}$$

where x root being sought. But this requires knowledge of x to compute.

Listing 12: Convergence of Relative Error

44a ⟨*Test if successive values meet a relative tolerance* 44a⟩≡

```
relative :: Double -> [Double] -> Double
relative tau (a:b:rest)
   | abs (a/b-1) <= tau = b
   | otherwise          = relative tau (b:rest)
```

Now we can express Newton's method to compute the square root of m to within a relative error tolerance τ starting with an initial guess x_0 as the function `mysqrt`.

Explain how you can generalize this code to apply Newton's method to recurrences other than `next`.

Listing 13: Newton's Square Root Method

44b ⟨*Newton's square root* 44b⟩≡

```
mysqrt x0 tau m = relative tau (repeatedly (next m) x0)
```

Convergence of Newton's Method

The number of times the `(next m)` function is evaluated measures the time complexity of the `mysqrt` algorithm. It is not obvious what this number is. What can be shown is that Newton's method converges quadratically, under certain assumptions that are often `True`.

Consider the function $f(x) = x^2 - m$. Using Taylor's theorem, you can derive the equation

$$x^2 - m = (x_{k-1}^2 - m) + 2x_{k-1}(x - x_{k-1}) + (x - x_{k-1})^2$$

There is a more elegant, more general proof of the quadratic convergence of Newton's method. But this is beyond the scope of these notes.

Pretend that $x = \sqrt{m}$ so that both sides of the above equation are zero. Divide by $2x_{k-1}$ to get

$$0 = (x_{k-1}^2 - m)/2x_{k-1} + (x - x_{k-1}) + (x - x_{k-1})^2/2x_{k-1}$$

Notice that

$$(x_{k-1}^2 - m)/2x_{k-1} - x_{k-1} = -(x_{k-1}^2 + m)/2x_{k-1} = -x_k$$

Therefore,

$$x_k - x = (x_{k-1} - x)^2/2x_{k-1} \quad \text{or}$$
$$e_k = \frac{e_{k-1}^2}{2x_{k-1}}$$

That is, the absolute error $x_k - x$ at step k is proportional to the square of the error at step $k-1$. When the error is less than 1, the number of correct digits doubles with each iteration.

In the general case, assume that function f has a continuous second derivative. Assume x is a root of f, that is $f(x) = 0$. By Taylor's theorem

$$0 = f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(\xi)}{2}(x - x_k)^2$$
$$x = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{f''(\xi)}{2f'(x_k)}(x - x_k)^2$$
$$x \approx x_k - \frac{f(x_k)}{f'(x_k)}$$

Exercises

1. The convergence decribed above is for the absolute error. What can you say about the rate of converengce for the relative error?

2. Simpson's rule is a simple quadrature algorithm. It is defined by the equation

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f(m) + f(b)\right] \quad \text{where } m = \left(\frac{a+b}{2}\right) \text{ is the midpoint.}$$

   (a) Explain the idea behind Simpson's rule.

   (b) Write a program that implements Simpson's rule.

   (c) Over a large interval $[a, b]$ you would apply Simpson's rule over many short intervals. Explain how this would be done.

6. Pattern Matching

But pattern-matching doesn't equal comprehension.

Peter Watts, Blindsight

Problem 2: Pattern Matching Problems

Decision Problem: *Given a pattern* p *and text* t, *does* p *occur in* t*?*
Function Problem: *Given a pattern* p *and text* t, *where does* p *occur in* t, *if at all?*

Imagine applications where pattern matching could be useful.

Prerequisite concepts

Please read Chapter 32 Pattern Matching in in the textbook (Corman et al., 2009). I also recommend (Apostolico and Galil, 1997) and (Gusfield, 1997) as references for pattern matching algorithms.

Definition 7: Alphabets, Strings, . . .

- *An* alphabet Σ *is a finite set of* symbols, *often called* characters *or* letters.
- *A* string s over Σ *is a finite sequence of* symbols *from* Σ.
- Σ^* *(the Kleene closure) is the set of all strings over* Σ.
- Σ^+ *is the set of all non-empty strings over* Σ.
- *A* language L *is a subset of* Σ^*.
- *A language* L *is* decidable *if there is an algorithm that correctly answers the question: Is* $s \in L$*? for all strings* $s \in \Sigma^*$.
- *An* algorithm *is . . . , well if it works like an algorithm and stops like an algorithm, then it is an algorithm* for some problem.

Pattern matching

We will study several algorithms for *pattern match*. We are interested in their implementation and their time and space complexities. Some auxiliary functions will be introduced as needed. Here is the outline for these notes.

How well these algorithms parallelize or distribute is also interesting, but beyond the scope of these notes.

48 ⟨*Pattern match* 48⟩≡
⟨*Brute-force pattern matching with left-to-right scan* 50a⟩
⟨*Morris–Pratt pattern matching* 52⟩
⟨*Knuth–Morris–Pratt pattern matching* 61⟩
⟨*Brute-force pattern matching with right-to-left scan* 65a⟩
⟨*Boyer-Moore pattern matching* 66⟩
⟨*Auxiliary functions* 60a⟩
⟨*Test the pattern matching algorithms* 75c⟩

Let's start with the brute force approach: It solves the problem and leads to optimization ideas.

Brute force pattern matching

The brute force approach is simple. As long as the pattern has not been found continue searching the text until the last possible text position has been exceeded.

Consider the initial configuration for some text and a pattern:

The sample `txt` and `pat` come from (Knuth et al., 1977).

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern a b c a b c a c a b
        ↑ 1 compare
```

Since there a mismatch at the up-arrow, the pattern is shifted one position right in the text to yield the configuration:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern   a b c a b c a c a b
              ↑ 4 compares
```

Now, the up-arrow shows a mismatch in this configuration at index 3 in the pattern and index 4 in the test. The brute force approach shifts the pattern one position to the right and restarts matching.

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern     a b c a b c a c a b
            ↑ 1 compare
```

I went through the compares and shifts and got this sequence counting compares.

Did you get the same sequence?

$$\langle 1, 4, 1, 1, 1, 8, 1, 1, 5, 1, 1, 2, 8, 1, 1, 10 \rangle$$

for a total of 47 compares.

Here is a C implementation of the brute force algorithm.

- It imports some header files for string and Boolean manipulations
- Initializes local state
- And, as long as there is text to search
  - It scans the text left-to-right trying to match the pattern
  - `True` is returned when a match is found
  - `False` is returned otherwise

The program is written using `noweb` to illustrate literate programming concepts.

Listing 14: Brute-force left-to-right pattern matching

50a ⟨*Brute-force pattern matching with left-to-right scan* 50a⟩≡

```
#include <string.h>
#include <stdbool.h>

bool bruteForce (const char *txt, const char *pat) {
⟨Initialize local state 50b⟩
  while (⟨text index is legal 50c⟩) {
    ⟨scan left-to-right 51a⟩
    if ⟨pattern found in text 51b⟩ return true;
    ⟨shift the pattern one place in the text 51c⟩
  }
  ⟨pattern not found in text 51d⟩
}
```

The local state consists of lengths for `txt` and `pat`, and indices `i` and `j` into them. The `txt` and `pat` are global values and not to be changed.

50b ⟨*Initialize local state* 50b⟩≡

```
int n = strlen(txt);
int m = strlen(pat);
int i = 0, j = 0;
```

Provided the `txt` index `i` has not gone beyond `n-m`, the `pat` has not been shifted too far.

50c ⟨*text index is legal* 50c⟩≡

```
(i <= n-m)
```

Let's also define a predicate that tests if the `pat` index `j` is legal too.

50d ⟨*pattern index is legal* 50d⟩≡

```
(j < m)
```

Matching starts at some `txt` index `i` and `pat` index `0`. The test for a match at index `j` in the `pat` and index `i+j` in the `txt` is simple.

50e ⟨`pat[j]` *matches* `txt[i+j]` 50e⟩≡

```
(pat[j] == txt[i+j])
```

All these conditions for matching are neatly handled in a while statement, whose body simply increments the pat index.

51a ⟨*scan left-to-right* 51a⟩≡
```
while (⟨pattern index is legal 50d⟩ && ⟨pat[j] matches txt[i+j] 50e⟩) {
  j++;
}
```

There are two exits for left-to-right scan. One is when the pat does match txt. This occurs when the pat index j reaches the length m of the pat.

51b ⟨*pattern found in text* 51b⟩≡
```
(j == m)
```

The second exit from the left-to-right scan occurs when a mismatch occurs. In this case we start the search over moving the pat one position in the txt. That is, increment the txt index i and reset the pat index to 0.

51c ⟨*shift the pattern one place in the text* 51c⟩≡
```
i++;
j = 0;
```

The scan of the txt stops when the pat has been been tested at each legal position and not been found.

51d ⟨*pattern not found in text* 51d⟩≡
```
return false;
```

A functional brute force pattern matcher

Here is a functional implementation for brute force pattern matching. The bf function returns True when pat occurs at the start of txt. As base cases: The empty pattern [] matches any txt and no non-empty pat matches empty text. For non-empty patterns and texts, if the first characters match and their tails match, then there is a match.

Listing 15: Functional Brute Force Algorithm

51e ⟨*Functional brute force pattern matching* 51e⟩≡
```
bf :: String -> String -> Bool
bf [] txt = True
bf pat [] = False
bf (p:ps) (t:ts) = (p == t) && (bf ps ts)
```

```
bfscan :: String -> String -> Bool
bfscan p t
   | bf p t        = True
   | bf p (tail t) = True
   | otherwise     = False
```

The `bfscan` function simply moves the pattern along the text until a match occurs or there is no match.

Brute-force Pattern Matching Analysis

There are several things to notice about brute force pattern matching.

1. Brute-force has optimal space complexity: It solves *pattern match* in constant space needing only a few registers for indices `i`, `j`, and lengths `m` and `n`. Space complexity usually ignores memory used for input to and output from the algorithm.
2. Brute-force has worst case $O((n - m + 1)m) = O(nm)$ time complexity and this may not be very good!
3. The average case time complexity may not be nearly as bad. Its depend on statistical properties of the text and pattern. These topics are beyond this initial discussion.

The Morris–Pratt pattern matcher

The Morris–Pratt algorithm is exactly the same as the brute force algorithm except how shifts are made once a mismatch is found.

Listing 16: Morris–Pratt Pattern Matching

52 ⟨*Morris–Pratt pattern matching* 52⟩≡

```
#include <string.h>
#include <stdbool.h>

bool morrisPratt (const char *txt, const char *pat)
{
  ⟨Initialize local state 50b⟩
  while ⟨text index is legal 50c⟩ {
    ⟨scan left-to-right 51a⟩
    if ⟨pattern found in text 51b⟩ return true;
    ⟨Morris–Pratt shift 58a⟩
  }
  ⟨pattern not found in text 51d⟩
}
```

Importantly, each character in the text is compared at most twice to a pattern character.

The brute force pattern matcher does not use any information gained before a mismatch occurs. It throws away any knowledge of a matching prefix where `pat[0..(j-1)] = txt[i..(i+j-1)]` and `pat[j] /= txt[i+j]` for some `j`. It simply restarts comparing `pat[0]` with `txt[i+1]`. Recall the configurations from the brute force example from page 62.

Consider the initial configuration for some text and a pattern:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern a b c a b c a c a b
        ↑ 1 compare
```

Since there a mismatch at the up-arrow, the pattern is shifted one position right in the text to yield the next configuration:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern   a b c a b c a c a b
                ↑ 4 compares
```

Now, the up-arrow shows a mismatch in this configuration at pattern index $j = 3$ and text index $i = 4$. The Morris–Pratt approach recognizes that shifting the pattern one or two positions to the right will result in immediate mismatches: We know, from matching `abc` that those text characters are `b` and `c`. So the pattern is shifted three places:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern         a b c a b c a c a b
                ↑ 1 compare
```

After the immediate mismatch between `pat[0] = a` and `txt[4] = b`, the pattern is shifted one place right:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern           a b c a b c a c a b
                              ↑ 8 compares
```

The mismatch occurs with the partial match `abcabca` and and mismatch between `c` and `a` pattern index $j = 7$ and text index $i = 12$.

Below, we will define the period to be the number of matching characters minus the length of the longest border.

The partial match `abcabca` has matching prefix and suffix `abca`. A shift by the *period* $p = 7 - 4 = 3$ will align these matching *borders* of `abcabca`. Importantly, the matches do not need to be re-checked!

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern               a b c a b c a c a b
                              ↑ 1 compares
```

Next, the string matched is `abca`. It has an identical prefix and suffix: The border `a`. A shift by the period $p = 4 - 1 = 3$ will align borders.

The text index becomes and the pattern index `j` is reset to and the matching a's do not need to compared.

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                       a b c a b c a c a b
                                ↑ 1 compares
```

Shift the pattern one place to find:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                         a b c a b c a c a b
                                                  ↑ 8 compares
```

Again, the pattern can be shifted $7-4=3$ places get the configuration:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                               a b c a b c a c a b
                                                        ↑ 6 compares
```

The sequence of Morris–Pratt compares in this instance is:

$$\langle 1, 4, 1, 8, 1, 1, 8, 6 \rangle$$

for a total of 30 compares.

Safe and Feasible Shifts

We will show how to preprocess the `pat` so that this matching information is not wasted. There will be a small increase in space, but the worst case time complexity for *pattern match* will be reduced. The preprocessing exploits the invariant

```
pat[0..(j-1)] = txt[i..(i+(j-1))]
```

Later the mismatch information will also be used to improve the time complexity.

After a mismatch, suppose `pat` is shifted from position `i` in `txt` to position `i+k`. There are three conditions such a shift `k` must or should satisfy:

1. Shifting `pat` to the right by `k` positions must be *safe*.
2. Shifting `pat` to the right by `k` positions should be *feasible*,
3. The shift `k` must be *at least one* position.

Definition 8: Safe and Feasible Shifts

1. *A pattern shift from* `i` *to* `i+k` *is* safe *if* `pat` *can* not occur *at any position in between. That is,* `pat` *cannot occur in the* `txt` *at positions* `i+1` *to* `i+k-1`.
2. *A safe shift is* feasible *if a match* could occur *at* `i+k` *(based on our knowledge of matching characters).*

For the example configuration

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern    a b c a b c a c a b
                   ↑
```

`pat[0..2] = txt[1..3]`. There is a mismatch between `pat[3]=a` and `txt[4]=b`. A shift of `k=1` is safe: This is the brute force approach. But, a shift one place right is not feasible. Likewise, based on match and mismatch knowledge, shifts of `k=2` and `k=3` are safe, but not feasible. We can often safely make larger shifts until a feasible shift is found. In this example, a shift `k=4` is safe and feasible.

Prefixes, Suffixes, Borders, and Periods

Before going farther let's introduce some terms and ideas that are helpful in pattern matching algorithms. These ideas show how to compute safe and feasible shifts.

Definition 9: Prefixes and Suffixes

Let w be a string. Then,

1. λ, *the empty string, is a prefix and suffix of* w.
2. *If* p *is a prefix of* w *and* $w = (p : a : ws)$, *then* pa *is prefix of* w.
3. *If* s *is a suffix of* w *and* $w = (ws : a : s)$, *then* as *is suffix of* w.

For example, the magic word $w = \mathtt{abracadabra}$, has non-empty proper prefixes:

$\mathtt{a}, \mathtt{ab}, \mathtt{abr}, \mathtt{abra}, \mathtt{abrac}, \mathtt{abraca}, \mathtt{abracad}, \mathtt{abracada}, \mathtt{abracadab}, \mathtt{abracadabr}$

and, non-empty proper suffixes:

$\mathtt{a}, \mathtt{ra}, \mathtt{bra}, \mathtt{abra}, \mathtt{dabra}, \mathtt{adabra}, \mathtt{cadabra}, \mathtt{acadabra}, \mathtt{racadabra}, \mathtt{bracadabra}$

For the our sample `pat=abcabcacab`, the non-empty proper prefixes are:

$\mathtt{a}, \mathtt{ab}, \mathtt{abc}, \mathtt{abca}, \mathtt{abcab}, \mathtt{abcabc}, \mathtt{abcabca}, \mathtt{abcabcac}, \mathtt{abcabcaca}$

and the non-empty proper suffixes are:

$$b, ab, cab, acab, cacab, bcacab, abcacab, cabcacab, bcabcacab$$

Definition 10: Border of a string

A border *of a word w is any string that is both a prefix and suffix of w. A proper border is non-empty.*

For example, the `pat=abcabca` has proper borders `a` and `abca`.

Let's explore how to define a function `border` that will map a string `w` to the *longest* proper border. Interestingly, upon exploring we find `border` is recursive. Consider the sample `pat=abcabca`.

`pat=abcabca`

$$\texttt{border abcabca} = \texttt{abca}$$
$$\texttt{border abca} = \texttt{a}$$
$$\texttt{border a} = \lambda$$

Borders have a dual notion called *periods*, which are integers `p` such that $0 < \texttt{p} <= |w|$ and prefix `w[0..k]` equals suffix `w[p..(|w|-1)]` where $\texttt{p} = |w| - \texttt{k} - 1$. That is, the periods of a string are how far to shift it to align borders.

For instance, the periods of `pat=abcabcacab` are 8 and 10 since

index	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
p = 8	a	b							a	b
p = 10										

where the last period $\texttt{p} = 10$ corresponds to the empty border.

As another example, the periods of `pat=abaabaaabaaba` are 7, 10, 12, and 13 since

index	0	1	2	3	4	5	6	7	8	9	10	11	12
pattern	a	b	a	a	b	a	a	a	b	a	a	b	a
p=7	a	b	a	a	b	a		a	b	a	a	b	a
p=10	a	b	a								a	b	a
p=12	a												a
p=13													

Notice that the period plus the length of any border equals the word's length

$$\texttt{p} + \texttt{length}\ (\texttt{border}\ w) = \texttt{length}\ w$$

The key idea is:

Shifting a pattern by its periods aligns borders.

Prefixes	*Borders*
λ	-1
a	0
ab	0
aba	1
abaa	1
abaab	2
abaaba	3

Such shifts are feasible, if they are safe. As we will see below in § , you can compute the lengths of borders. Suppose we compute the

length `border pat[0..(j-1)]` for each index `j=1..m` and store this length in an array `border`. We'll see below that `border[0]` should be set to -1. For the pattern abaaba, the `border` array is:

$$\texttt{border} = \langle -1, 0, 0, 1, 1, 2, 3 \rangle$$

When a mismatch occurs at `pat` index `j`, the `pat` is shifted by its period `p=j-border[j]`. Here's an example:

```
pat index j    0 1 2 3 4 5
pat            a b a a b a
txt          a a b a c a b a a ...
txt index i  0 1 2 3 4 5 6 7 8 ...
                     ↑
```

The mismatch occurs at `j=3` once we've matched aba, which has border a of length 1. The period of aba is `2=3-1`. Shifting the `pat` 2 positions right produces the alignment of borders.

```
pat index j          0 1 2 3 4 5
pat                  a b a a b a
txt          a a b a c a b a a ...
txt index i  0 1 2 3 4 5 6 7 8 ...
```

We can restart matching at pattern index `j = border[3] = 1`. We don't need to recheck the match at `pat[0]` with `txt[3]`. Of course, there is an immediate mismatch at `j = 1`, which we'll handle as an optimization soon.

A Morris–Pratt shift by period `p` is safe. Here's why. The period `p` is defined the equation

```
p = j - border[j]
```

where `border[j]` is the length of the longest border of the prefix `pat[0..(j-1)]`. Consider the boundary (base) case where `j=0`. This happens when `pat[0] /= txt[i]`. A shift of `p = 1` is warranted, and so

For instance, if `j=0`, `pat[0]=a` and `txt[i]=b`, then shifting the pattern one place right is safe.

```
p = 1 = 0 - border[0]
```

which implies `border[0] = -1`.

In the general case, when a mismatch occurs at `pat` index `j>0`, a shift by period `p` is *safe*.

To see this, pretend to the contrary that a shift `p` is *unsafe*. That is, for some index $i < k < p$,

Pretend $j = 6$, pattern$[0..5] = abaaba$ and `txt[i..(i+5)]=abaabc`. Shifting `pat p=6-3` places right is safe.

```
pat[0..(m-1)] = txt[k..(k+m-1)]
```

We know `pat[0..(j-1)] = txt[i..i+(j-1)]` and `pat[j] /= txt[i+j]`.

But, `pat[j] = txt[k+j]`.

Figure this out.

Morris–Pratt Shift

After a mismatch, the matching process restarts from `pat` index `j = max(0, border[j])`. These are the conditions of the Morris–Pratt shift.

58a ⟨*Morris–Pratt shift* 58a⟩≡

```
i = i + j - pattern.border[j];
j = (0 < pattern.border[j]) ? pattern.border[j] : 0;
```

Computation of borders

We want to compute `border[0..(m-1)]` for `pat` of length `m`. Recall `border[j]` is the length of the longest border of prefix `pat[0..(j-1)]`. A shift by `p = j - border[j]` aligns border prefix of `pat[0..(j-1)]` with its suffix.

In an object-oriented paradigm, a `pat` object calls `computeBorders()`, that is, the call `pat.computeBorders()` is made before the call `txt.MorrisPratt(pattern)`.

We know `border[0] = -1` and so we can set this initial value. Also, we'll store the length of the current border in a local variable `b`.

58b ⟨*Define border[0]* 58b⟩≡

```
border[0] = -1;
int b = -1;
```

We want to compute `border[1],..,border[m]` where `m` is the length of `pat`. This is done in a `for` loop over each non-empty prefix of `pat`. The Boolean condition for the loop is:

58c ⟨*each non-empty prefix* 58c⟩≡

```
(int j = 1; j <= m; j++)
```

Let's pretend we've computed `b=border[j-1]` of `pat[0..(j-2)]` for some `j >= 1`. We want to compute `border[j]`. Figure 2 shows that if `pat[0..b] = pat[(j-b-1)..(j-1)]` and `pat[b+1]=pat[j]`, then `border[j] = b+1`. That is, when the character after the prefix matches the character after the right border, then the border can be extended by 1.

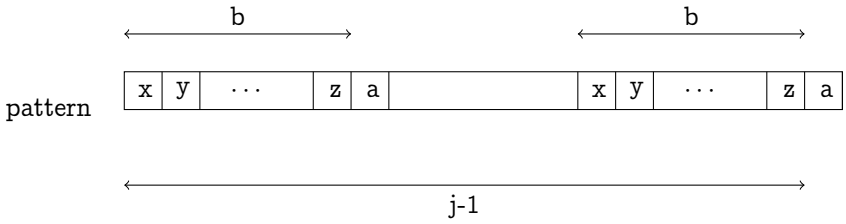


Figure 2: Morris–Pratt: Extend border

59a ⟨*Extend the border by one character when next characters match* 59a⟩≡

```
++b;
border[j] = b;
```

Now let's consider what happens when there is a mismatch after the current border. That is, `pat[b] != pat[j-1]`. This case is a more complex, but Figure 3 shows that while mismatches occur, the border length `b` must be reduced to `border[b]` until there is a match or no border is found (`b = -1`).

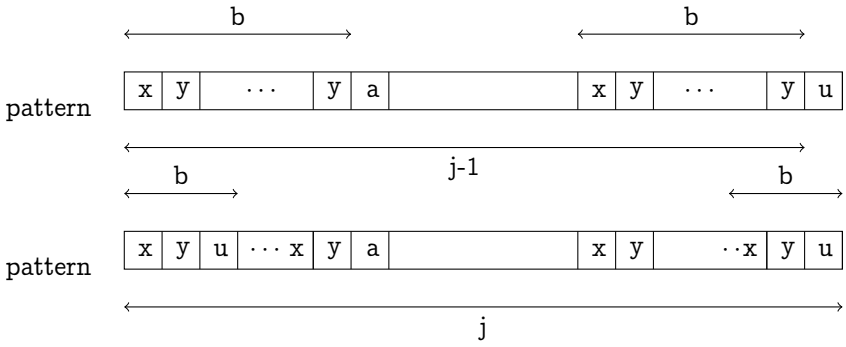


Figure 3: Morris–Pratt: Shrink border

59b ⟨*Reduce the border until a match is found or no border exists* 59b⟩≡

```
while ((b >= 0) && (pattern.charAt(b) != pattern.charAt(j-1)))
{
  b = border[b];
}
```

Example: Computing Borders

Consider the length 8 pattern

p=abaababa

We want to fill out the array `border[0..8]` *of border lengths.*

j	*Prefix*	*Border String*	*Border Length*
0	λ	-	*-1*
1	a	λ	*0*
2	ab	λ	*0*
3	aba	a	*1*
4	abaa	a	*1*
5	abaab	ab	*2*
6	abaaba	aba	*3*
7	abaabab	ab	*2*
8	abaababa	aba	*3*

Using the above code chunks we can put together the `computeBorders()` function easily.

60a ⟨*Auxiliary functions* 60a⟩≡

```
public int* computeBorders()
{
  ⟨Define border[0] 58b⟩
  for ⟨each non-empty prefix 58c⟩ {
    ⟨Reduce the border until a match is found or no border exists 59b⟩
    ⟨Extend the border by one character when next characters match 59a⟩
  }
}
```

One final act, so the compiler does not complain, declare the integer array `border[]` used in the Morris–Pratt algorithm

60b ⟨*Pattern matching instance properties* 60b⟩≡

```
public int[] border;
```

Analysis of the Morris–Pratt pattern matcher

The need for the array `border[0..m]` increases the space complexity from the constant space required for the brute force algorithm. The benefit is a linear worst case time complexity.

Theorem 8: Morris–Pratt Shifts

The maximum number of compares in the Morris–Pratt algorithm is 2n-m.

Proof: Morris–Pratt Shift

There is at most one unsuccessful comparison for each index `i`, *which ranges from* `0` *to* `n-m`. *We can give an upper bound for the number of successful compares by considering the sum* `i+j`. *The least value for* `i+j` *is* `0` *and the greatest value is* `n-1`. *Each time a successful compare is made* `i+j` *increases by* `1` *and it never decreases, thus there are at most* `n` *successful compares. Finally not both successful and unsuccessful compares can attain their maximum, thus there are at most* `n + (n-m+1) - 1 = 2n-m` *compares in the Morris–Pratt algorithm.*

The Knuth–Morris–Pratt pattern matcher

The Morris–Pratt algorithm can be improved by using additional information known at the time a mismatch occurs. In particular, the complete invariant is:

```
pat[0..(j-1)] = txt[i..i+(j-1)]
                and
pat[j] /= txt[i+j].
```

The Knuth–Morris–Pratt (KMP) algorithm makes use of this additional mismatch information to allow longer shifts of the pattern in the text.

Listing 17: Knuth–Morris–Pratt Pattern Matching

61 ⟨*Knuth–Morris–Pratt pattern matching* 61⟩≡

```
public boolean KMP (Text pattern)
{
  ⟨Initialize Knuth–Morris–Pratt local state 63c⟩
  while (⟨text index is legal 50c⟩)
  {
    ⟨scan left-to-right 51a⟩
    if (⟨pattern found in text 51b⟩) { return true;}
    ⟨Knuth–Morris–Pratt shift 63a⟩
  }
  ⟨pattern not found in text 51d⟩
}
```

Consider the initial configuration for some text and a pattern:

```
text    b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern a b c a b c a c a b
        ↑ 1 compare
```

Since there a mismatch at the up-arrow, the pattern is shifted one position right in the text to yield the configuration:

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern    a b c a b c a c a b
                 ↑ 4 compares
```

Now, the up-arrow shows a mismatch in this configuration at index 3 in the pattern and index 4 in the test. The Knuth–Morris–Pratt approach recognizes that shifting the pattern one, two, or three positions to the right will result in immediate mismatches and so the pattern is shifted four places:

The match abc has border length 0. Morris–Pratt would shift only by $3-0$. Knuth–Morris–Pratt shifts by $3-(-1)=4$.

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern            a b c a b c a c a b
                                 ↑ 8 compares
```

After the mismatch on the c in the pattern, abcabca has been matched and abca is its border. The character after the border is b. The character after the matching prefix is c. Because of the mismatch, we know the mismatching character in the text is not c. It could be b or some other character. Therefore, the safe feasible shift is by $7-4=3$ giving the configuration:

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                  a b c a b c a c a b
                                 ↑ 1 compares
```

Now, the string matched is abca which has border a. The character after the border is b. The character after the matching prefix is also b. Because of the mismatch, we know the mismatching character in the text is not b. Shifting by the period $4-1=3$ will lead to an immediate mismatch, Therefore, a shift of $4-0$ is safe and feasible, where the 0 is the length λ, the border of a.

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                        a b c a b c a c a b
                                                ↑ 8 compares
```

The string matched is abcabca. Therefore, as before, a shift of $7-4=3$ is safe and feasible.

```
text     b a b c b a b c a b c a a b c a b c a b c a c a b c
pattern                              a b c a b c a c a b
                                                      ↑ 5 compares
```

The sequence of Knuth–Morris–Pratt compares is:

$$\langle 1, 4, 8, 1, 8, 5\rangle$$

for a total of 27 compares.

One might call the idea used in the KMP shift *strict borders*. The length of these strict borders are stored in an array `strictBorder[]`. The updates to indices `i` and `j` are computed just as in Morris–Pratt.

63a ⟨*Knuth–Morris–Pratt shift* 63a⟩≡

```
i += j-pat.strictBorder[j];
j = (0 < pat.strictBorder[j]) ? pat.strictBorder[j] : 0;
```

Declare `strictBorder[]` and initialize its elements to `-1`.

63b ⟨*Pattern matching instance properties* 60b⟩+≡

```
public int[] strictBorder;
```

63c ⟨*Initialize Knuth–Morris–Pratt local state* 63c⟩≡

```
int n = strlen(txt);
int m = strlen(pat);
int i = 0, j = 0;
for (int j = 0; j < pat.length(); j++) { strictBorder[j] = -1; }
```

To fill out the `strictBorder[]` array, consider what we know:

`pat[0..j-1]= txt[i..i+j-1]` and `pat[j] /= txt[i+j]`

If `pat[0..k]` is the border of `pat[0..j-1]` and `pat[k+1] = pat[j] /= txt[i+j]` then there will be an immediate mismatch when we shift to align borders. In this case we can safely shift farther aligning the border of `pat[0..k]` with the tail of `pat[0..j-1]`.

On the other hand, if `pat[k+1] != pat[j]` then perhaps `pat[k+1] = txt[i+j]` and we can only safely shift the length of border `pat[0..j-1]`.

Example: Strict Borders Example

Consider the length 8 pattern

abaababa.

We want to fill out the array `strictBorder[0..8]` *of border lengths. The index* `j` *in the left column of the table below denotes the number of characters that have been matched. For the Knuth–Morris–Pratt (KMP) algorithm, when the character after the border does not equal the character after the matching prefix, the strict border is the Morris–Pratt border. While the character after a border equals the character after the matching prefix, borders of borders are followed until there is a mismatch.*

j	*Prefix*	*Border String*	*Border Length*	*Char after Prefix*	*Char after Border*
0	λ	-	*-1*	*a*	-
1	a	λ	*0*	*b*	*a*
2	ab	λ	*0*	*a*	*a*
3	aba	a	*1*	*a*	*b*
4	abaa	a	*1*	*b*	*b*
5	abaab	ab	*2*	*a*	*a*
6	abaaba	aba	*3*	*b*	*a*
7	abaabab	ab	*2*	*a*	*a*
8	abaababa	aba	*3*	λ	*a*

MP Shift	*Strict Border*	*KMP Shift*
1	*-1*	*1*
1	*0*	*1*
2	*-1*	*3*
2	*1*	*2*
3	*-1*	*5*
3	*-1*	*6*
3	*3*	*3*
5	*-1*	*8*
5	*0*	*8*

Here's an example showing the shift that occurs on a mismatch.

```
index 0 1 2 3 4 5 6
pat   a b a a b a a          pat[6] != txt[0+6] and the
txt   a b a a b a c a b a a b
```

border of abaaba is aba, which has length 3, implying a Morris–Pratt shift of 6-3=3. But, a shift of 3 leads to an immediate mismatch since `pat[3]=a` will be compared with `txt[6]=c`. Thus, we can consider the border of the border of abaaba, that is the border of aba, or a, which has length 1 and shift by 6-1=5.

Analysis of the Knuth–Morris–Pratt pattern matcher

The KMP pattern matcher has space complexity $S(n+m) = O(m)$. This reflects the storage for array `strictBorder[0..m]` and the constant space required for indices and lengths. The time complexity $T(n+m) = O(n+m)$ is generally better than the Morris–Pratt algorithm, but may be no better than it.

Right-to-left scanning for pattern matching

Now we want to develop a brute force algorithm that will match `pat` against `txt` using a right-to-left scan of `pat`. The previous left-to-right algorithm is modified in these ways:

1. Index `j` starts at the end of `pat`.
2. The inner scan decrements `j`.
3. When `j` runs off the left-end (`j = -1`) a complete match has occurred.
4. The shift moves the `pat` right one place and resets `j` to the end of `pat`.

65a ⟨*Brute-force pattern matching with right-to-left scan* 65a⟩≡

```
public boolean rightLeftBruteForce (Text pattern)
{
  ⟨Initialize right-to-left local state 65b⟩
  while ⟨text index is legal 50c⟩ {
    ⟨right-to-left scan 65c⟩
    if (j == -1) return true;
    ⟨right-to-left brute-force shift of pattern 65d⟩
  }
  ⟨pattern not found in text 51d⟩
}
```

The right-to-left scan starts at the end of `pat`.

65b ⟨*Initialize right-to-left local state* 65b⟩≡

```
int j = m-1;
```

And decrements `pat`'s index `j` so long as the `txt` and `pat` match.

65c ⟨*right-to-left scan* 65c⟩≡

```
while ((j > -1) && (pat[j] == txt[i+j])) {
  j = j - 1;
}
```

When a mismatch occurs, slide `pat` one position right (`i = i+1`) and reset `j` to point to the end of `pat`.

65d ⟨*right-to-left brute-force shift of pattern* 65d⟩≡

```
++i;
j = m - 1;
```

The Boyer-Moore pattern matcher

The Boyer-Moore algorithm (Boyer and Moore, 1977) is exactly the same as the right-to-left brute force algorithm except how shifts are made once a mismatch is found.

Listing 18: Boyer-Moore algorithm

66 ⟨*Boyer-Moore pattern matching* 66⟩≡
```
public boolean BoyerMoore (Text pat)
{
  ⟨Initialize Boyer-Moore local state 67b⟩
  while ⟨text index is legal 50c⟩ {
    ⟨right-to-left scan 65c⟩
    if (j == -1) return true;
    ⟨Boyer-Moore shift of pattern 67a⟩
  }
  ⟨pattern not found in text 51d⟩
}
```

The Boyer-Moore algorithm uses knowledge gained from the brute force algorithm to leverage an improved pattern matcher. What do we know? When a mismatch occurs, we know the invariant

```
pat[j] /= txt[i+j]
```

and

```
pat[(j+1)..(m-1)] = txt[(i+j+1)..(i+m-1)]
```

This invariant is shown in figure 4, where `j = 6`, `i = 0` and `m-1 = 9`, and `pat[7..9] = txt[7..9]` and `pat[6] /= txt[6]` Let's

Figure 4: Alignment with match using right-to-left scan

```
pat index  0  1  2  3  4  5  6  7  8  9
pat        a  b  c  a  b  c  a  c  a  b
txt        b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a  b  c  a  b  c  a  c  a  b  c
txt index  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
                        ↑
```

pretend, after a mismatch, `pat` is shifted `s` positions to the right where $1 \leq s \leq j$. This aligns `pat[0..m-1-s]` and `pat[s..m-1]` as shown in Figure 5. In particular, `pat[j-s]` aligns with `pat[j]` and `txt[i+j]`. Thus, we require that the shift `s` satisfy three conditions:

1. `pat[j-s] /= pat[j]`. If they were equal a mismatch between `pat[j-s]` and `txt[i+j]` would occur; such a shift is not feasible.
2. A border of the reversed prefix `pat[(m-1)..(j-s+1)]` has length `m-1-j`, that is, `pat[(m-1)..(j+1)] = pat[(m-1-s)..(j+1-s)]`.

Notice condition 2 is a statement about *some* border of a prefix of the reversed string; this border may not be *the* border.

Place drawing to showing shift here.

Figure 5: Boyer-Moore shift between 1 and `j` positions.

Now let's pretend that we shift `j+1 <= s < m` characters. Such a shift aligns `pat[0..m-1-s]` and `pat[s..m-1]` with `txt[i+s..i+m-1]`, see figure 6. Such a shift `s` satisfies the third condition:

3. `pat` has a border `pat[0..m-1-s]` of length `m-j-1` or less.

Place drawing to showing shift here.

Figure 6: Boyer-Moore shift between `j+1` and `m` positions.

When either the first two conditions or the third condition fail to hold we can safely shift `pat` the maximal amount `m`. Given the `pat` we can compute if there are shifts satisfying the conditions above. Such shifts are safe and feasible. For each `j` the longest safe and feasible shift is stored in a look-up table which is used when a mismatch occurs. The algorithm is identical to the brute force right-to-left scan, except for this look-up of the shift.

We'll call the table of shifts `goodSuffix`. Then when a mismatch occurs on pattern index `j`, the shift `goodSuffix[j]` is added to `i` and `pat` index `j` is reset to the end of `pat`.

67a ⟨*Boyer-Moore shift of pattern* 67a⟩≡

```
i += pat.goodSuffix[j];
j = m-1;
```

67b ⟨*Initialize Boyer-Moore local state* 67b⟩≡

```
i = m - 1;
j = m - 1;
```

Example: Good Suffix Shift

j	*Suffix*	*Border*	*Length*	*Char Before Suffix*	*Char Before Border*	*Good Suffix Shift*
0	λ	-	*-1*	*a*	-	*5*
1	a	λ	*0*	*b*	*a*	*5*
2	ba	λ	*0*	*a*	*a*	*5*
3	aba	a	*1*	*a*	*b*	*5*
4	baba	a	*1*	*b*	*b*	*2*
5	ababa	ab	*2*	*a*	*a*	*7*
6	aababa	aba	*3*	*b*	*a*	*4*
7	baababa	ab	*2*	*a*	*a*	*1*
8	abaababa	aba	*3*	λ	*a*	*2*

Computing the goodSuffix *array*

We start by declaring an instance of the `goodSuffix[]` array. It's length will be set to `m} when [[pat` is created and each element will be initialized to zero, the Java default. (Remember `Text` variable `pat` has an internal representation with a `String` named `txt` of length `n`. The variable `m` will be used when discussing the algorithm, but `n` will be used in the code.)

68a ⟨*Pattern matching instance properties* 60b⟩+≡

```
public int[] goodSuffix;
```

The code for `goodSuffix[]` is abstruse. Essentially, we want to test the conditions discussed above. Our implementation is driven more by a need for clarity than efficiency in time and space. Here is the complete algorithm for `computegoodSuffix()`.

Let's start with an auxiliary routine that reverses the character is a string. This will be useful in testing condition 2.

68b ⟨*Auxiliary functions* 60a⟩+≡

```
public Text reverse()
{
  int n = txt.length();
  StringBuffer reverse = new StringBuffer();
  for (int i = n-1; i > -1; i-) {
    reverse.append(txt.charAt(i));
  }
  return new Text(reverse.toString());
}
```

Borders for both the pat and its reverse are used, and a variable called s will denote the shift.

69a ⟨*Declarations and initializations* 69a⟩≡
```
int n = txt.length();
Text reverse = reverse();
computeBorders();
reverse.computeBorders();
int s;
```

We'll start by being opportunistic and set, for each j, goodSuffix[j] = m the largest possible shift. As we find that shorter shifts are safe and feasible we'll reset goodSuffix[j] to these smaller values.

69b ⟨*Set each shift to the maximal value* 69b⟩≡
```
for (int j = 0; j < n; j++) {
  goodSuffix[j] = n;
}
```

Boundary conditions

To develop the shift table for the Boyer-Moore algorithm , we'll consider boundary cases first.

First compare is a mismatch. When there is an immediate mismatch between pat[m-1] and txt[i+m-1], a shift of 1 is appropriate, but so is a shift by the smallest value s such that pat[m-1-s] != pat[m-1]. This is condition 1 for the case j=m-1. The requirement is that s be the smallest value satisfying reverse[0..s+1] = 0.

No compare is a mismatch. Here it must be that j, our pat position index, has fallen off the left end of pat, that is j == -1. Our decision algorithm simply return true when this occurs.

Mismatch on the last compare. Now let's consider the case that pat[0] != txt[i]. That is, we've match all characters except the first. I hope it is obvious that a shift by the period of pat, that is m-border[m], is both safe and feasible. The border is a *good suffix* where a shift by the period will produce a potential pattern–text match; no shorter shift can.

Non-boundary cases

We restrict our attention to the case where a mismatch occurs at pat[j] and $0 < j < m - 1$. This scenario is shown in figure 4, and there are two cases to consider. These are illustrated in figures 5 and 6. In figure 5, the proposed shift s is no more than j. In figure 6, s is larger than j, but less than m-1.

A shift `1 <= s <= j` Let's pretend that the situation of figure 5 occurs to determine how to build the code that enforces the situation. Thus `s` is between 1 and `j` where a mismatch occurs at `pat[j]`. Figure 5 illustrates that two conditions must hold:

$$\text{pat[j-s]} \not= \text{pat[j]} \tag{2}$$

and

$$\text{pat[j+1-s..m-1-s]} = \text{pat[j+1..m-1]}. \tag{3}$$

Condition 2 is the Knuth–Morris–Pratt strict border condition and condition 3 is the Morris–Pratt border condition for the *reversed* pattern. With `reverse` being the reversal of `pat`, condition 3 says that prefix `reverse[0..m-j+s-2]` has length `m-j-1}`.

`For a fixed [[j` between `1` and `n-1` and `n` we'll start the shift `s` at `1` and increment `s` until both conditions hold or `s` exceeds `j`. So, for a for each proposed shift `s`, we'll test if the strict border condition 1 holds and when it does we'll determine if the border condition 2 holds on the `reverse pattern`; variable `k` is the length of a border.

70a ⟨*Search for a safe shift between 1 and* `j` 70a⟩≡

```
s = 1;
while (s <= j) {
   if ⟨Strict border condition 70b⟩ {
     ⟨Initialize border of reverse[0..(n-j-1+s)] 70c⟩
     while ⟨Border greater than tail to match 71a⟩ {
       ⟨Reset to smaller border 71b⟩
     }
     if (k == n-j-1) { // border condition satisfied
       ⟨Set goodSuffix[j] and exit while loop 71c⟩
     }
   }
    ++s;
 }
```

The strict border condition is:

70b ⟨*Strict border condition* 70b⟩≡

```
(txt.charAt(j-s) != txt.charAt(j))
```

The prefix of `reverse`, whose borders we want to test, is `reverse[0..(n-j-1+s)]`. We'll start by setting variable `k` to the border of this prefix; `k` will be decremented while it is larger than the length of the tail of `pat` we want to match, that is, `n-1-j`.

70c ⟨*Initialize border of* `reverse[0..(n-j-1+s)]` 70c⟩≡

```
int k = reverse.border[n-j-1+s];
```

The length of the tail that has been matched when a mismatch occurs at `j` is `n-1-(j+1)-1 = n-1-j`.

71a ⟨*Border greater than tail to match* 71a⟩≡
```
(k > n-j-1)
```

The next smaller border is found by looking at the border of the border.

71b ⟨*Reset to smaller border* 71b⟩≡
```
k = reverse.border[k];
```

At this point a shift `s` that satisfies conditions 1 and 2 has been found. We can array to exit the `while (s <= j)` loop by setting `s = j`; it will then be incremented forcing an exit of the loop.

71c ⟨*Set* `goodSuffix[j]` *and exit* `while` *loop* 71c⟩≡
```
goodSuffix[j] = s;
s = j;
```

Putting all of these pieces together gives the code below.

71d ⟨*Reset shifts when* `1 <= goodSuffix[j] <= j` 71d⟩≡
```
for (int j=1; j < n; j++) {
  ⟨Search for a safe shift between 1 and j 70a⟩
}
```

A shift `j < s`

Now let's develop the code when no shift between 1 and `j` can be found, but perhaps a shift greater than `j` exists. Figure 6 depicts the situation that is represented by the equation

$$\text{pat}[0..m\text{-}1\text{-}s] = \text{pat}[s..m\text{-}1]. \tag{4}$$

Here's the outline of what we need to do.

72a ⟨*Reset shifts when* goodSuffix[j] > j 72a⟩≡
```
⟨Set the border length 72b⟩
⟨Initialize the index where the search starts 72e⟩
while (⟨There is a non-empty border 72c⟩) {
  ⟨Set shift to the period of the current border 72d⟩
  for (int j = start; j < s; j++) {
    goodSuffix[j] = (s < goodSuffix[j]) ? s : goodSuffix[j];
  System.out.println("**gs["+j+"]="+goodSuffix[j]);
  }

  ⟨Reset the start index 73⟩
  ⟨Reset to smaller border 71b⟩
}
```

The shortest shift of the type under consideration is determined by the period of `pat`. We'll initialize `k` to the length of `pat`'s border and let `k` become successive (shorter) border lengths as we search for longer shifts.

72b ⟨*Set the border length* 72b⟩≡
```
int k = border[n];
```

```
The search continues as long as there is a non-empty border.
After each search for a shift with one border, we reset the
border length [[k]] to the length of the next border.
```

72c ⟨*There is a non-empty border* 72c⟩≡
```
(k > 0)
```

With a border of length `k` the period to shift aligning `pat` borders is `n-k`.

72d ⟨*Set shift to the period of the current border* 72d⟩≡
```
s = n - k;
```

A placeholder `start` will be used to control the search over `j`. The first time through `pat` index `j` starts at 0.

72e ⟨*Initialize the index where the search starts* 72e⟩≡
```
int start = 0;
```

Once we've searched over a range `start <= j < s`, the next search can be over a range that begins with `start = s`.

73 ⟨*Reset the start index* 73⟩≡

```
start = s;
```

And that is the code which enforces condition 3.

Concerns about the derivation of `computeGoodSuffix()`

The above derivation of `computeGoodSuffix()` is not very efficient, but it may be more clear than other developments of the code.

The Last Occurrence Function

The classical Boyer-Moore algorithm uses what is known as the *last occurrence* or *bad character* heuristic. It says, when a mismatch occurs between `pat[j]` and `txt[i+j]`, find the right-most (last) occurrence of `txt[i+j]` in `pat` and shift to align these, see figure 7 which shows this shift.

Place drawing last occurrence or bad character.

Figure 7: The last occurrence or bad character heuristic.

When the last occurrence of `b` in `pat` is at index `k`, the last occurrence shift on a mismatch at `j` is `lastOccurrence[j] = j-k`. Notice that when `k > j` this is a negative (leftward) shift! Also when `b` does not occur in `pat` a shift of `j+1` characters is appropriate, thus we'll define `lastOccurrence[b] = -1` when `b` does not occur in `pat`. To create a `lastOccurrence[` table requires $|A|$ space (A is the alphabet and $|A|$ is its cardinality).

Some authors eschew the use of a `lastOccurrence[]` table, other extol it. It does require space that is dependent on the alphabet, something we've not seen before. It's utility depends on the alphabet size and distribution of characters in `pat`.

Analysis of the Boyer-Moore pattern matcher

Establishing a tight upper bound on the number of comparisons is beyond the scope of these notes. A bound of $4n$ is fairly simple to prove, although $3n$ is a better approximation. When `pat` is relatively long and the alphabet is large, Boyer-Moore is likely to be the most efficient pattern matcher. Empirically, in the average case, the number of compares is often sub-linear, that is, the number of compares is cn where $c < 1$.

Finishing up

To complete the code for the `Text` class we'll define a constructor and some auxiliary functions. The constructor has one `String` argument, which is set to the `txt`. We also initialize the tables (arrays) used to look up shifts required by the various algorithms.

74 ⟨*Auxiliary functions* 60a⟩+≡

```
public Text(String t)
```

```
  {
    txt = t;
    border = new int[txt.length()+1];
    strictBorder = new int[txt.length()+1];
    goodSuffix = new int[txt.length()];
  }
```

Another useful method returns the length of the `txt` string.

75a ⟨*Auxiliary functions* 60a⟩+≡

```
  public int length()
  {
    return txt.length();
  }
```

And, another useful method returns the character at a position `k` in the `txt` string.

75b ⟨*Auxiliary functions* 60a⟩+≡

```
  public char charAt(int k)
  {
    return txt.charAt(k);
  }
```

Test the Algorithms

Now we'll do one last, but important thing. We'll write some test cases that helps us to believe that no defects occur in our code.

The `main` routine will read two strings from command line and then perform various tests to see that our algorithms work correctly (at least on the test cases). The first string is the `txt` and the second is the `pat`.

75c ⟨*Test the pattern matching algorithms* 75c⟩≡

```
  public static void main(String[] args)
  {
    Text txt = new Text(args[0]);
    Text pat = new Text(args[1]);
    ⟨Test the left-to-right scan brute-force patternMatcher() 76a⟩
    ⟨Test computeBorders() 76b⟩
    ⟨Test MorrisPratt() 76c⟩
    ⟨Test computeStrictBorders() 76d⟩
    ⟨Test KnuthMorrisPratt() 76e⟩
    ⟨Test the right-to-left scan brute-force patternMatcher2() 76f⟩
    ⟨Test computeGoodSuffix() 76g⟩
    ⟨Test BoyerMoore() 77⟩
  }
```

The first test will be of the brute force left-to-right scan pattern matcher.

76a ⟨*Test the left-to-right scan brute-force* `patternMatcher()` 76a⟩≡
```
System.out.println(txt.patternMatcher(pat));
```

One thing to test is that the `border[]` array is correctly computed.

76b ⟨*Test* `computeBorders()` 76b⟩≡
```
pat.computeBorders();
for (int j = 0; j <= pat.length(); j++) {
  System.out.println("border[" + j + "] = " + pat.border[j]);
}
```

Now let's test that our implementation of the Morris–Pratt algorithm works correctly.

76c ⟨*Test* `MorrisPratt()` 76c⟩≡
```
System.out.println(txt.MorrisPratt(pat));
```

We can not test the KMP algorithm since we've left its completion as an exercise.

76d ⟨*Test* `computeStrictBorders()` 76d⟩≡
```
//  pat.computeStrictBorders();
//  for (int j = 0; j <= pat.length(); j++) {
//    System.out.println("border[" + j + "] = " + pat.border[j]);
//  }
```

76e ⟨*Test* `KnuthMorrisPratt()` 76e⟩≡
```
//  System.out.println(txt.KnuthMorrisPratt(pat));
```

76f ⟨*Test the right-to-left scan brute-force* `patternMatcher2()` 76f⟩≡
```
System.out.println(txt.patternMatcher2(pat));
```

Before testing Boyer-Moore we see if `goodSuffix[]` is calculated properly.

76g ⟨*Test* `computeGoodSuffix()` 76g⟩≡
```
pat.computeGoodSuffix();
for (int j = 0; j < pat.length(); j++) {
  System.out.println("goodSuffix[" + j + "] = " + pat.goodSuffix[j]);
}
```

And now our test of `BoyerMoore()`.

77 ⟨*Test* `BoyerMoore()` 77⟩≡

```
System.out.println(txt.BoyerMoore(pat));
```

Exercises

1. An alternate to terminating the search in the brute force sequential search algorithm is to continue looking for a second or more occurrences of the pattern. An on-line algorithm which continually accepts input until an end of input marker is found would usually do this. Re-write the code to handle a continuous stream of characters. It will output a stream of 0's and 1's indicating the pattern was not or was found.

2. Show that in the worst case, `bruteForce`'s the inner `while` loop can execute `m` times and the outer `while` loop can execute `n-m+1` times. Show that the maximum number of comparisons is $(n+1)^2/4$ and give example strings for `pat` and `txt` where this worst case is realized. Hint: maximize the quadratic expression $nm - m^2 + m$ as a function of m.

3. Turn the brute force algorithm given in these notes into a working program with input and output. Test the average case behavior of the code. Use the words in a dictionary (for example `usr/dict/words` on a Unix system) as patterns for which to search. Find a large text document on the World Wide Web (for example the The Gutenberg Project has a large collection of great books that can serve as text files).

4. Define *Fibonacci words* over the alphabet $\{a, b\}$ by

$$F_0 = e,\ F_1 = b,\ F_2 = a,\ \text{and} F_n = F_{n-1}F_{n-2}\ \text{for}\ n >= 2$$

   Determine the length of F_n. Find the periods and borders of F_n.

5. Develop an algorithm that computes the strict border of a pattern. You may find it useful to know that `strictBorder[j] = border[j]` if `pat[border[j-1]+1] != pat[j]`, while when this inequality does not hold we set `j = border[j]` until it does or `j` becomes negative. Show that your algorithm is correct and estimate its time complexity.

6. Provide a time and space complexity analysis of the presented code for `computeGoodSuffix()`.

7. Develop an alternative more efficient (in time and space) algorithm for `computeGoodSuffix()`. Some things to consider. Declaring the `reverse` of `pat` requires significant extra space; it can be eliminated. The time spend of computing `goodSuffix[n-1]` is large; this computation can be folded into the computation when `goodSuffix[j] <= j`.

8. Write a program `computeLastOccurrence()`, which when given an alphabet `A` and a `pat` determines the last occurrence (rightmost) of each character in `A` in `pat`. Use this algorithm to *improve* the

Boyer-Moore algorithm. Empirically compare the time and space complexity of Boyer-Moore with and without this improvement by using a large `txt` and multiple `pats`.

7. All Keywords in Text

The Aho-Corasick Algorithm

The AhoâĂŞCorasick string matching algorithm formed the basis of the original Unix command fgrep.

This section studies the problem of finding all key words in a string of text. To gain a deeper understanding, read the paper (Aho and Corasick, 1975).

Describe applications where an algorithm for this problem could be useful.

Problem 3: All Keywords Problem

Function Problem: *Given a set of keywords* $\{k_0, \ldots, k_{n-1}\}$ *and a text string* T, *find all occurrence of the keywords in* T.

Example: Find all Keywords in a Text

Find keywords: {`he, she, his, hers`} *in the text:*

$T =$ *"the time for this ushers ashes"*

The keywords occur at positions

Keyword	*Index*
`he`	1 20 27
`she`	19 26
`his`	14
`hers`	20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t	h	e		t	i	m	e		f	o	r		t	h

15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
i	s		u	s	h	e	r	s		a	s	h	e	s

Given the keywords, the Aho-Corasick algorithm constructs a finite state machine that recognizes each keyword. The machine consists of three functions that Aho and Corasick call *goto*, *failure*, and *output*.

The *goto* function for the keywords, `he`, `she`, `his`, and `hers` is shown as a finite state machine in Figure 8 below. When a match is found, the *goto* function maps a (state, character) pair forward to a next state. For instance, if the machine is in state 2 and the next character is `r`, then the machine moves to state 8. When the next character in the `text` does not move the search forward, the `failure`

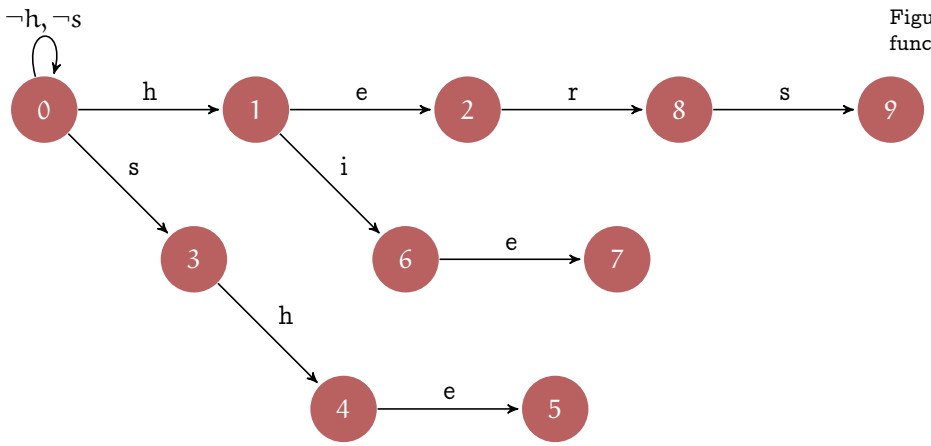


Figure 8: The Aho-Corasick `goto` function

function is called. For instance, if the machine is in state 1 and the next character is neither `e` nor `i`, the machine fails back to the start state 0. Likewise, in states 2, 3, 6, or 8, if the next character is not `r`, `h`, `s`, or `s`, respectively, fail back to the start state 0. In state 4 where an `h` was just seen, fail to state 1, In state 5 where `he` was just seen, fail to state 2, And, in state 7 or 9, where an `s` was just seen, fail to state 3.

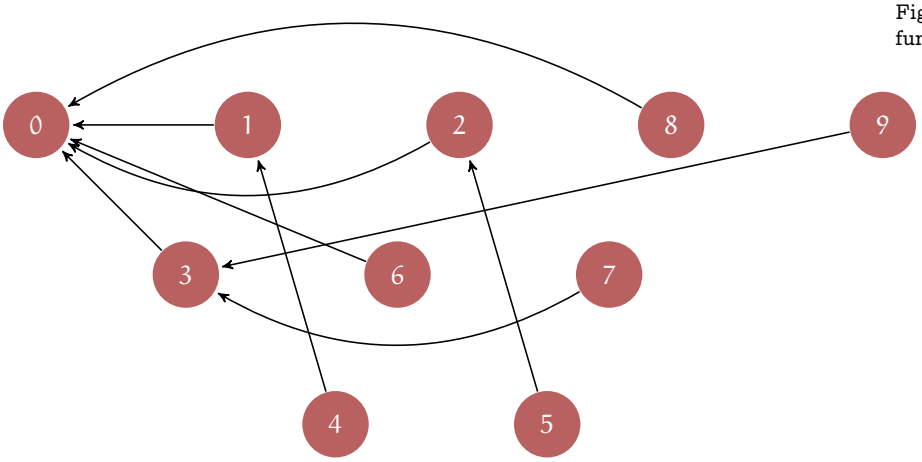


Figure 9: The Aho-Corasick `failure` function

Finally, there is an *output* function that prints keywords and where they were found in the `text`.

State	Output
2	he
5	{she, he}
7	his
9	hers

With these functions, the Aho-Corasick algorithm can be written as follows. The code is *object-oriented*. It assumes a *class* called `State` that has auxiliary `goto`, `failure`, and `output` functions.

The general structure of the algorithm is shown in the code below.

Listing 19: Aho-Corasick All Keywords Search

83a ⟨*Aho–Corasick Algorithm* 83a⟩≡
```
⟨Auxiliary functions 84a⟩

public void ahoCorasick(String text) {
  State state = start;
  for (int i = 0; i < text.length; i++)  {
    ⟨While goto fails follow the failure function 83b⟩
    ⟨Otherwise, follow the goto function 83c⟩
    ⟨If in an output state, print information 83d⟩
  }
}
```

83b ⟨*While* `goto` *fails follow the* `failure` *function* 83b⟩≡
```
while (state.goto(text.charAt(i)) == fail) {
  state = state.failure();
  ⟨If in an output state, print information 83d⟩
}
```

Once a state where the label on an outgoing edge matches the character in the text, follow that edge to the next state.

83c ⟨*Otherwise, follow the* `goto` *function* 83c⟩≡
```
state = state.goto(text.charAt(i));
```

Pretend the class `State` has an auxiliary function `hasOutput` that decides if a state has output to be displayed.

83d ⟨*If in an* `output` *state, print information* 83d⟩≡
```
if (state.hasOutput()) {
  state.output();
}
```

Analysis of Aho-Corasick Algorithm

The `ahoCorasick()` algorithm makes fewer than $2n$ state transitions in processing a text string of length n. To understand this note the following:

1. The algorithm makes exactly n `goto` (forward) transitions where n is the length of the `text` string.
2. For each character in the `text` the algorithm makes zero or more failure (backward) transitions.
3. To reach a state of depth d requires d `goto` transitions.
4. If we reach a state of depth d, then no more than d failure transitions can occur afterwards.

Therefore, the number of failure transitions is no more than the number of goto transitions. (The number of gotos is greeter than or equal to number of failures) The total the number of state transitions is therefore bound above by

$$\text{goto's} + \text{failure's} \leq n + n = 2n$$

and the algorithm makes $2n$ or fewer transitions.

The `goto` and `failure` functions can be constructed in time $O(m)$ where m is the length of the concatenated keywords.

Aho-Corasick `goto` *Function*

Can you help find all the mistakes in this code? I'm certain there are many.

An array of keywords is passed into the `buildGoTo` function: The `goto` function is constructed. And the `output` function is partially defined.

Assume `output(state)` is empty when `state` is first created. Also, assume `state.goto(c) = fail` if `c` is undefined or if `state.goto(c)` has not yet been defined.

84a ⟨*Auxiliary functions* 84a⟩≡

```
public void buildGoTo(String[] keyword) {
  State start = new State();
  ⟨Enter each keyword into the goto table 84b⟩
  ⟨Add self-loops to start for characters not starting keywords 85a⟩
}
```

84b ⟨*Enter each keyword into the* `goto` *table* 84b⟩≡

```
for (int i = 0; i < keyword.length; i++) {
  ⟨Enter keyword[i] 85c⟩
}
```

85a ⟨*Add self-loops to* `start` *for characters not starting keywords* 85a⟩≡
```
⟨For each character c in the alphabet 85b⟩
if (start.goto(c) == fail) start.goto(c) = start;
```

The alphabet and its implementation are not fleshed out here.

85b ⟨*For each character* `c` *in the alphabet* 85b⟩≡

To enter a new keyword, first follow its already matched prefix.
Then add states for its remaining suffix.

85c ⟨*Enter keyword[i]* 85c⟩≡
```
State state = start;
int j = 0;
⟨Follow existing path prefix of keyword[i] 85d⟩
⟨Construct new path for suffix of keyword[i] 85e⟩
⟨Save keyword[i] as output 85f⟩
```

85d ⟨*Follow existing path prefix of keyword[i]* 85d⟩≡
```
while (state.goto(keyword[i].charAt(j)) != fail) {
  state = state.goto(keyword[i].charAt(j));
  ++j;
}
```

85e ⟨*Construct new path for suffix of keyword[i]* 85e⟩≡
```
for (int k = j; k < keyword[i].length; k++) {
  State newState = new State();
  state.setGoto(keyword[i].charAt(k)) = newState;
  state = newState;
}
```

85f ⟨*Save* `keyword[i]` *as* `output` 85f⟩≡
```
state.saveOutput(keyword[i]);
```

Aho-Corasick `failure` *Function*

Now let's build the `failure` function. First, add each state of depth 1 to a queue. Each such state fails back to the `start` state. Then, use the states stored in the queue to compute failures for states of greater depth.

86a ⟨*Auxiliary functions* 84a⟩+≡
```
public void buildFailureTranstions() {
  ⟨Add each state of depth one to a queue 86b⟩
  ⟨Compute failure for states of depth d from those of depth (d − 1) 86c⟩
}
```

86b ⟨*Add each state of depth one to a queue* 86b⟩≡
```
Queue queue = new Queue;
⟨For each character c in the alphabet 85b⟩ {
  State state = start.goto(c);
  if (state != start) {
    queue.enqueue(state);
    state.setFailure(start);
  }
}
```

Next, get a `state` from the queue. For every character `c` that moves this `state` forward, queue up that `nextState`. Then, follow failures from `state` until they end. Set `nextState` to fail to where the `goto` function moves `c`.

Here is an example using figure 8: Pretend you are in state 8, a state of depth 3, having matched *her*. On the character `s`, move to `nextState = 9`. The `failure` from state 8 has already been computed to be the `start` state 0. From this `start` state, the `goto` function moves to state 3 on `s`. Therefore, failure in `nextState = 9` moves to state 3.

86c ⟨*Compute* `failure` *for states of depth* d *from those of depth* (d − 1) 86c⟩≡
```
while (queue.notEmpty()) {
  State state = queue.dequeue();
  ⟨For each character c in the alphabet 85b⟩ {
    if (state.goto(c) != fail) {
      State nextState = state.goto(c);
      queue.enqueue(nextState);
      ⟨Follow failures from state until they end 87a⟩
      nextState.setFailure(failState.goto(c));
      state.saveOutput(nextState.output());
    }
  }
}
```

87a ⟨*Follow failures from* `state` *until they end* 87a⟩≡

```
failState = state.failure();
while (failState.goto(c) == fail) {
  failState = failState.failure();
}

\subsection{Aho-Corasick [[output] Function}
```

87b ⟨*Auxiliary functions* 84a⟩+≡

```
public bool hasOutput() {
  // to be determined
}

public bool output() {
  // to be determined
}
```

Exercises

1. Construct the *goto*, *failure*, and *output* functions for the keywords

   sofa, soft, take, tame, sort, fast

8. Dictionaries and Hash Tables

Please read Chapter 11 Hash Tables in in the textbook (Corman et al., 2009).

Many applications require a data structure with only dictionary operations: `insert`, `delete`, and `search`.

Definition 11: Dynamic Sets and Dictionaries

- *A* dynamic set $\mathbb{X}$ *can change over time: It supports* `insert` *and* `delete` *operations.*
- *A* dictionary *is a dynamic set with a* `search` *operation.*
- *A* dictionary *is typically implemented as an* associative array *of (*`key`, `value`*) pairs.*
- *The* `key` *locates the* `value`*.*
- *The* `value` *can hold* satellite data. *(Indirect addressing, the* `value` *locates additional data)*

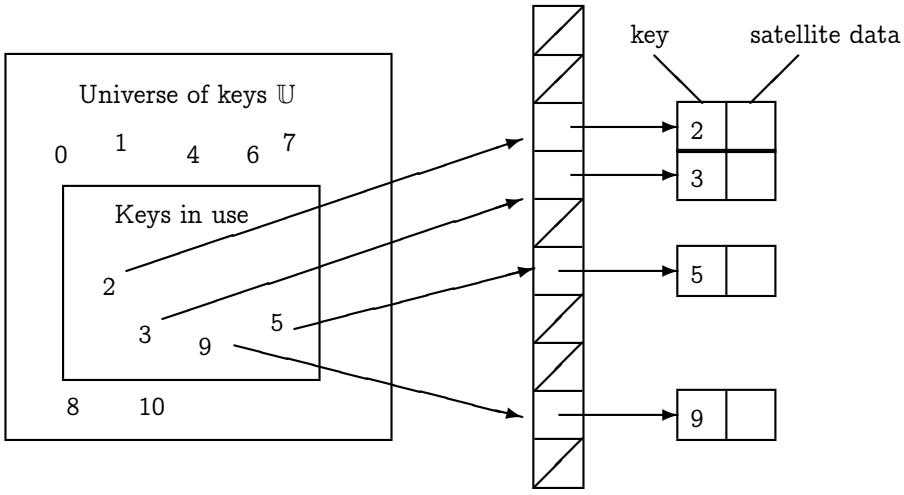
A *hash table* is an effective data structure for implementing dictionaries. A hash table is a generalization of an ordinary array, but a hash table index is computed from a key stored in the hash table. Under reasonable assumptions `insert`, `delete` and `search` can be performed in $O(1)$ (constant) time. First, we'll consider direct-address tables based on arrays. Then, we'll look at hash tables and what makes a good hash function. And, finally, we'll consider open hashing.

Direct Address Tables

Suppose a dynamic set draws keys from the universal set

$$\mathbb{U} = \{0, 1, 2, \ldots, (m-1)\}$$

where m is not too large.



Direct Addressing

A direct-address table is an array $T[0..(m-1)]$ of pointers with three operations: search, insert, and delete. The indentity function key[x]=x is the most simple *hash*.

Direct addressing has $O(1)$ worst-case time for these three operations.

Listing 20: Direct Address Operations

90 ⟨*Direct Address Operations* 90⟩≡

```
directAddressSearch (int T[], int k) {
  return T[k];
}

void directAddressInsert(int T[], int x) {
  T[key[x]] = x;
}

directAddressDelete(int T[], int x) {
  T[key[x]] = null;
}
```

Hash Tables

Direct addressing can waste space when the size $|\mathbb{U}|$ is large and the set of used keys $\mathbb{K} \subset \mathbb{U}$ is small. Also, if the keys are not integers, but say text strings, then some additional processing is necessary.

A *hash table* can require less space than direct address tables. Storage requirements can be reduced to $O(|\mathbb{K}|)$. And, $O(1)$ average time search can still be attained.

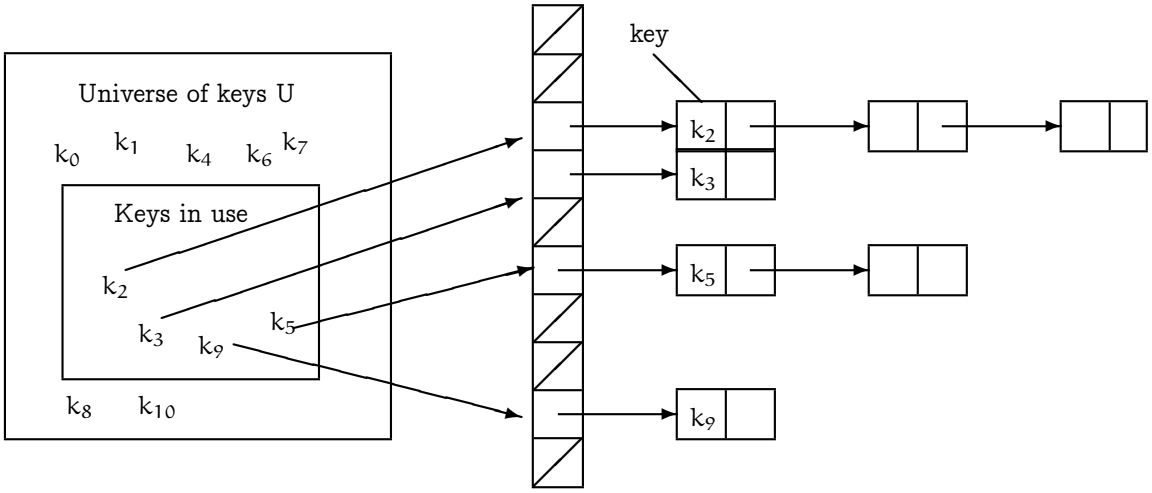
An element x with key k is stored in slot $h(k)$ where h is a *hash function*. The hash function maps the universe $\mathbb{U}$ of keys into slots of a hash table $T[0..m-1]$.

$$h : \mathbb{U} \to \{0, 1, \ldots, m-1\}$$

Key k "hashes" to slot $h(k)$, $h(k)$ is "hash value" of k. Since the size m of the hash table T is much smaller than the size $|\mathbb{U}|$ of the universe of keys, there may be *collisions* when two keys hash to the same slot.

Collision Resolution by Chaining

Chaining puts all elements that hash to the same slot in a linked list.



Collision Resolution by Chaining

Listing 21: Hashing with Chaining

91 ⟨*Hash Table with Chaining* 91⟩≡

```
value chainHashSearch(Table T, Key k) {
  search for element with key k in list T[h(k)];
}

void chainHashInsert(Table T, key x) {
  insert x at the head T[h(key[x])
}

void chainHashDelete(Table T, key x) {
  delete x from list T[key[x]]
}
```

Load Factors

Define the *load factor* α to be n/m for a hash table T with m slots storing n elements. The load factor α represents the average number of elements stored in a chain. The worst case behavior of hashing with chaining is $O(n)$: All keys hash to the same slot. The *simple uniform*

hashing assumption is that each element is equally likely (probability $1/m$) to hash into any of the m slots.

Theorem 9: Uniform Hashing

An unsuccessful search takes time $O(1 + \alpha)$ on average when simple uniform hashing is used and collisions are resolved by chaining.
A successful search takes time $O(1 + \alpha)$ on average when simple uniform hashing is used and collisions are resolved by chaining.

Hash Functions

A good hash function satisfies approximately the assumption of simple uniform hashing: Each key is equally likely to hash to any one of the m slots.

Formally, we want

$$\sum_{\text{all } k \text{ such that } h(k) = j} P(k) = \frac{1}{m}$$

for each $j = 0, 1, \ldots, (m-1)$.

In practice the probabilty P is not known, but several heuristic techniques can create good hash functions. Symbols that are close should not hash to the same slots in the hash table. For example, in a program two variable may be similar, but they should map to different slots in the compiler's symbol table. The hash value should be indepedent of any patterns in the data, that is, it should the value should be uniform and random

Most hash functions assume the keys come from the set of natural numbers $\mathbf{N} = \{0, 1, 2, \ldots\}$ When the keys are not natural numbers they must be converted For example, character strings can be represented via there ASCII code in radix 128 notation

$$\text{aghef} = 97 \cdot 128^4 + 103 \cdot 128^3 + 104 \cdot 128^2 + 101 \cdot 128 + 102$$

The Division Method

The division method maps a key k into one of m slots using the hash function

$$h(k) = k \bmod m$$

For example, if $m = 15$ and $k = 123$, then $h(123) = 123 \bmod 15 = 3$ $(15 \cdot 8 + 3 = 123)$.

The division method is very fast since only a single division is needed. Certain values of m should *not* be used

- If m is a power of 2, $m = 2^p$, then $h(k)$ is the p lowest-order bits of k; not all the bits of k are used in computing $h(k)$

- If keys are decimal numbers, then powers of 10 should be avoided from m.
- If $m = 2^p - 1$ and k is a character string interpreted in radix 2^p, then two strings that are identical except for a transposed character hash to the same value
  - For example let $m = 2^2 - 1 = 3$
  - String "acbd" is represented as $k = 0 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4 + 3 = 39$ which hashes $39 \pmod 3 = 0$
  - String "cabd" is represented as $2 \cdot 4^3 + 0 \cdot 4^2 + 1 \cdot 4 + 3 = 135$; which hashes $135 \pmod 3 = 0$
- Good values of m are primes not too close to exact powers of 2
  - Suppose there are about $n = 4000$ character strings to be held in a hash table
  - Collision is resolved by chaining and we don't mind up to 3 elements in a chain
  - Since $4000/3 \approx 1333$ we could set $m = 1381$, a prime not too close to 1024
  - The hash function would be

$$h(k) = k \pmod{1}381$$

The Multiplication Method

- There are two steps in the multiplication method
  - The key k is multiplied by a constant A in the range $0 < A < 1$ and the fractional part of kA extracted
  - This fractional part is multiplied by m and the floor taken
- Thus the hash function is

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- In the multiplication method the value of m is not critical and $m = 2^p$ for some p is typically chosen
- Knuth suggests that one over the golden mean is generally a good value for A

$$A = \frac{1}{\frac{1+\sqrt{5}}{2}} = \frac{\sqrt{5}-1}{2} \approx 0.6180339887\ldots$$

- For example, if $k = 123456$, $m = 10000$ and $A = 0.6180339887$, then

$$k \cdot A = 123456 \cdot 0.6180339887 = 76300.0041089472$$

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (76300.0041089472 - \lfloor 76300.0041089472 \rfloor) \rfloor \\ &= \lfloor 10000 \cdot (.0041089472 \rfloor) \rfloor \\ &= \lfloor (41.1089472 \rfloor) \rfloor \\ &= 41 \end{aligned}$$

Universal Hashing

- A malicious adversary could choose the keys to be hashed so that they all hash to the same slot yielding an $\Theta(n)$ retrieval time
- Universal hashing uses randomization to avoid this and yield good average case performance, no matter how the keys are chosen by an adversary
- The hash function is chosen randomly and indepedently of how the keys are chosen
- This is similar to the idea of choosing the pivot randomly in quicksort, so no single input gives worst case behavior
- Let $\mathcal{H}$ be a finite collection of hash functions that map universe U into $\{0, 1, \ldots, m-1\}$
- The collection $\mathcal{H}$ is said to be *universal* if for each pair of keys $x, y \in U$, $x \neq y$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$
- That is, with a hash function chosen randomly from $\mathcal{H}$, the chance of collision between x and y when $x \neq y$ is $1/m$

Theorem 10: Universal Hashing

If $h()$ is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m, where $n \leq m$, then the expected number of collisions involving a particular key k is less than 1.
The class $\mathcal{H}$ described below is a universal class of hash functions.

It is easy to design a universal class of hash functions. Let the table size m be prime larger than 255. Let keys be decomposed into $r+1$ bytes.

$$x = \langle x_0, x_1, \ldots, x_r \rangle$$

Let $\langle A \rangle = \langle a_0, a_1, \ldots, a_r \rangle$ denote a sequence of random values chosen from $\{0, 1, \ldots, m-1\}$. Define a corresponding hash function $h_a \in \mathcal{H}$

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m}$$

Define

$$\mathcal{H} = \bigcup_{a} \{h_a\}$$

where the union is over all m^{r+1} possible sequences $a = \langle a_0, a_1, \ldots, a_r \rangle$

Open Addressing

- In open hashing all elements are stored in the hash table itself; there are no chains of nodes attached to a hash slot
- In this case the hash table can "fill up" so that no further insertions can be made; the load factor α can never exceed 1
- If the element in the dynamic set is not at the first computed hash location, then another slot is computed, then another and so on, until the element or **NULL** is found
- The order of the *probes* into the hash table depends on the key and the number of previous probes
- That is, the hash function maps two inputs, the key and probe number to a hash slot

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

- The probe sequence

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

for a key k is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$

Listing 22:

95 ⟨*Hash Search* 95⟩≡

```
hashSearch(T, k)
{
  int i = 0;
  repeat {
    j = h(k, i);
    if (T[j] = k) { return j; }
    i = i+1;
  }
  until ((T[j] == NULL) or i = m);
  return NULL;
}
```

```
hashInsert(T, x)
{
  int i = 0;;
  repeat {
    j = h(k,\,i);
    if (T[j] == NULL) {
       T[j] = k;
       return j;
     }
     else i = i+1;
  until (i = m)
  error ‘‘hash table overflow}’’;
}
```

- Deletion is not difficult, but we can't just mark a slot as **NULL**; doing so will make it impossible to find a key that occurs after that slot in a probe sequence
- A special value such as *Deleted* could be used, with appropriate modifications of `hashSearch` and `hashInsert`
  item We assume *uniform hashing*, that is, each key considered is equally likely to have any of the $m!$ permutations of $\{0, 1, \ldots, m-1\}$ in it probe sequence
- Uniform hashing is a generalization of simple uniform hashing, now the hash function produces an entire sequence of numbers not just a single number
- There are three common techniques for open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
- Each guarantees that the probe sequence of a key is permutation of $\{0, 1, \ldots, m-1\}$
- None guarantees uniform hashing: linear and quadratic probing only produce m probe sequences and double hashing produces m^2; uniform hashing requires $m!$ probe sequences

Linear Probing

- Given an ordinary hash function $h' : U \to \{0, 1, \ldots, m-1\}$, linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \pmod m \quad \text{for } i = 0, 1 \ldots, m-1$$

- The slots are probed as a circular list starting at $T[h'(k)]$

$$T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], \ldots, T[h'(k)-1]$$

- There are only m distinct probe sequences
- Linear probing suffers from *primary clustering*; long runs of occupied slots which increases the average search time
- Runs of occupied slots tend to build up
  - If an empty slot is preceded by i full one, then the probability that it is the next one filled is the probability that the empty slot or any of i preceding slots is hashed next and this is

$$\frac{i+1}{m}$$

  - However, if an empty slot is preceded by an empty one, then the probability that it is the next one filled is just $1/m$

Quadratic Probing

- Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \pmod{m}$$

  for $i = 0, 1 \ldots, m-1$, where h' is an ordinary hash function, and c_1 and $c_2 \neq 0$ are auxiliary constants
- Again only m distinct probe sequences are possible, but quadratic probing works better than linear probing
- Secondary clustering can result since the probe sequence is determined by the intial position

Double Hashing

- Double hashing is one of the best methods for open addressing because it produces hash slots that are close to random
- Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \pmod{m}$$

  for $i = 0, 1 \ldots, m-1$, where h_1 and h_2 are auxiliary hash functions
- The value of $h_2(k)$ must be relatively prime to m for the entire hash table to be searched
- One way to ensure this is to let $m = 2^p$ for some positive integer p, and design h_2 so it always produces an odd number

- Another way is to let m be prime and have h_2 return a values less than m
- For example

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m') \end{aligned}$$

  where m is prime and m' is slightly less than m (say $m-1$ or $m-2$)
- Double hashing produces $\Theta(m^2)$ probe sequences

Analysis of Open Addressing

- Let $\alpha = n/m$ be the *load factor* where m is the size of the hash table and n is the number of entries in the table
- Since with open addressing there is at most one entry per slot $n \leq m$ and $\alpha \leq 1$
- *Uniform hashing* is assumed to be valid, the probe sequence

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

  is equally likely to be any permutation of

$$\langle 0, 1, \ldots, m-1 \rangle$$

  for each key k

Theorem 11: Load Factor for Open-Address Hashing

Given an open-address hash table with load factor $\alpha = n/m \leq 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$ (assuming uniform hashing)

- **Corollary** Inserting an element into an open-address hash table with load factor $\alpha < 1$ requires at most $1/(1-\alpha)$ probes on average
- **Theorem:** Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in an successful search is at most

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right) + \frac{1}{\alpha}$$

  (assuming uniform hashing and each key is equally likely to be searched)

Proof

A search for a key follows the same probe sequence as that followed by an insert.
By the corollary, if k *was the* $(i+1)$*st key inserted, the expected number of probes is at most*

$$\frac{1}{1-i/m} = \frac{m}{m-i}$$

- *Averaging over all* n *keys in the hash table gives*

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

- *Using the bound that* $\ln i \le H_i \le \ln i + 1$ *we obtain*

$$\begin{aligned}\frac{1}{\alpha}(H_m - H_{m-n}) &\le \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) \\ &= \frac{1}{\alpha}(\ln\frac{m}{m-n} + \frac{1}{\alpha} \\ &= \frac{1}{\alpha}(\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}\end{aligned}$$

9. Sorting

Please read Part II Sorting and Order Statistics in (Corman et al., 2009).

Ineffective Sorts

Sorting — Basics

Problem 4: Sorting

Decision Problem: *Given a list of keys, is it sorted?*
Function Problem: *Given a list of keys, sort it into ascending (or descending) order.*

Pretend you are given a *file of records* containing *keys* that can be ordered. That is, the keys are from a totally ordered set where the

common relations and functions

$$(<), (\leq), (=), (\geq), (>), (\neq), \min, \max$$

are defined on the keys.

The function `sorted` *witnesses* that a list is sorted in ascending order or not.

What is the time complexity of `sorted`?

Listing 23: The Sorted Decision Problem

102 ⟨*Sorted list or not?* 102⟩≡

```
sorted :: (Ord a) => [a] -> Bool
sorted []     = True
sorted [x]    = True
sorted (x:y:ys) = (x <= y) && sorted (y:ys)
```

The `sorted` function has time complexity $O(n)$ and this shows the `Sorted` can be decided in polynomial time. That is, `Sorted` is in NP, the class of decision problems that can be solved in polynomial time using a non-deterministic Turing machine. Of course, the function problem: Sort this list, can be solved in polynomial time and is in P,

Two factors dominate the time spent sorting by comparing keys.

1. The number of comparisons made, and
2. The amount of data moved

When an algorithm requires duplicating the records, space complexity can become an issue too.

- Some comparison-based sorting algorithms have $O(n^2)$ worst case time complexity. For example: bubble, insertion, and selection sorts.
- Other comparison-based sorting algorithms have $O(n \lg n)$ worst case time complexity. For example: merge and heap sorts.
- Quicksort is famous for almost always being fastest, but in some races it does not win.

Sentinels: To terminate a sort, some algorithms benefit from *sentinels* at an ends of records ($A[0]$ or $A[n+1]$). Sentinels are typically below or above every valid value.

Comparison sorts: The most common sorts require comparing keys. The lower bound time complexity for comparison sorts is $\Omega(n \lg n)$.

Sorts without Compares: Time complexity can be reduced when data properties are known. There are several algorithms that sort in $O(n)$ time without comparing keys. For example, counting, radix, and bin sorts.

Internal and External Sorts: Sorting algorithms can also be classified by the size of the file to be sorted. Internal sorting processes files that fit into main memory. External sorting sorting processes files too large to fit in main memory. The files are stored in external memory: magnetic tapes, disk, or on a network (in the cloud).

Stability: A sorting algorithm is *stable* if it preserves relative order of equal keys. For example, if an alphabetized file of names is sorted by salary, those names with the same salary will remain in alphabetical order.

Sorting Algorithms

The following algorithms are presented and analyzed in these notes.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Merge Sort
- Quicksort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket (Bin) Sort

Some nice demonstrations of several sorting algorithms can be found at this external site. Know thy complexities! Below is a list of sorting algorithms and their complexities. I found it here. The value n is the length of the list (number of keys in the file) to be sorted.

I don't know Timsort, perhaps you can educate me. See Algorithmics on 219.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Timsort	$O(n)$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Heapsort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$O(n)$	$O((n\log(n))^2)$	$O((n\log(n))^2)$	$O(1)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

A main takeaway from the following notes should be: Small case analysis can lead to general expressions and understanding.

Bubble Sort

The heuristic is: Repeatedly pass through the records exchanging adjacent elements that are out of order. When no exchanges are needed

Bubble sort is infamous for being clear but slow. Although bubble sort is maligned, it is a useful example to learn about algorithm analysis.

the list is sorted. The insight for bubblesort is the `bubble` function that maps a list into a tuple containing a *bubbled* list and a Boolean flag indicating whether an exchange was made or not.

Bubble

Before learning to sort, learn to bubble. The `bubble` function will move the smallest value in a list to the head.

Listing 24: Functional Bubbling

A pattern for defining a function is shown in the code outline.

104a ⟨*Bubble smallest to head* 104a⟩≡
```
⟨Define the function's type 104b⟩
⟨Define base computations 104c⟩
⟨Define the recursion 104d⟩
```

The `bubble` function maps a list `[a]` of values from a totally ordered set to an ordered pair: A permutation of the input list and a Boolean flag that signals if the input was changed or not. Not explicit in the type declaration is the post condition: The smallest value in the input list is the head of the output list.

104b ⟨*Define the function's type* 104b⟩≡
```
bubble :: Ord a => [a] -> ([a], Bool)
```

As base cases, the empty list and a singleton are sorted and need no bubbling or changes to the list.

104c ⟨*Define base computations* 104c⟩≡
```
bubble [] = ([], False)
bubble [x] = ([x], False)
```

Otherwise, on a longer list `(x:xs)`, there are two objectives: Permute the input, if necessary, so its smallest value becomes the head of the list. And, record whether or not the input list was changed.
To bubble the smallest value to the head in a list `(x:xs)`, bubble the tail `xs` returning a permuted list `(y:ys)` where `y` is the smallest value in `xs`. If the permutation is the identity, the `changed` flag is `False`, otherwise `changed = True`.
Now that `x` and `y` are the two smallest values in `(x:xs)`, they can be ordered, setting and returning the `changed` flag appropriately.

104d ⟨*Define the recursion* 104d⟩≡
```
bubble (x:xs) =
  let (y:ys, changed) = bubble xs
  in if x > y
  then (y:x:ys, True)
  else (x:y:ys, changed)
```

It may or may not be clear that the time complexity of `bubble` is

Inductive and equational reasoning is useful here. Trace the executions

- bubble [2, 5] =
  - let ([5], False) = bubble [5]
  - in if 2 > 5
  - then ([2, 5], False)
- bubble [7, 2, 5] =
  - let ([2, 5], False) = bubble [2, 5]
  - in if 7 > 2
  - else ([2, 7, 5], True)
- bubble [4, 7, 2, 5]
  - let ([2, 7, 5], True) = bubble [7, 2, 5]
  - in if 4 > 2
  - then ([2, 4, 7, 5], True)

Convince yourself: One call to `bubble zs` places the smallest value in `zs` at the head of the list.
A second call to `bubble zs` places the next smallest value in `zs` in the second position of the list.
Each subsequence call places another value it it correct position.

$O(n)$ where n is the length of the input list.

The algorithm shows when `bubble` is called on a list of size n, it recursively calls itself on a list of size $n-1$. The additional work takes constant time: Comparing `x` and `y`; and inserting values into the head head of `ys`. The time complexity of `bubble` can be described by the recurrence

$$T(n) = T(n-1) + c, \quad T(0) = c$$

Which has solution

$$T(n) = c(n+1) = O(n)$$

By induction: If $T(n) = c(n+1)$, then $T(0) = c = (0+1)c$. And, by substitution, $T(n-1) = cn$ for $n \geq 1$. Therefore, the equation $T(n) = c(n+1) = cn + c = T(n-1) + c$ is satisfied.

Now `bubble` can be used repeated to sort a list.

Listing 25: Functional Bubble Sort

105a ⟨*Functional bubble sort* 105a⟩≡

```
bubble_sort :: Ord a => [a] -> [a]
bubble_sort xs =
   let (zs, changed) = bubble xs
   in if changed
      then bubble_sort zs
      else zs
```

Each call to `bubble` places at least one element where it belongs. Therefore, `bubble_sort` will be called at most n times on a list of length n.

- In the best case, `bubble_sort xs` calls `bubble xs` which returns `(zs, False)`, and `bubble_sort` immediately returns `zs`. In this case the time complexity is $T(n) = c(n+1)$.
- In the worst case, `bubble xs` returns `(zs, True)` each of $n-1$ times and `(zs, False)` on the last pass. In this case, the time cost is $T(n) = cn(n+1) = O(n^2)$

Here is an imperative implementation of bubble sort. It is naive in that it does not halt once it determines the array is sorted

Listing 26: Imperative Bubble Sort

105b ⟨*Bubble sort: C version* 105b⟩≡

```
#include <stdio.h>

void bubblesort(int A[], int n)
{
  int tmp; // for swapping
  for (int i = n-1; i > 0; i-) {
```

```
      for (int j = 1; j <= i; j++) {
        if (A[j-1] > A[j]) {
           tmp = A[j-1];
           A[j-1] = A[j];
           A[j] = tmp;
        }
      }
    }
}

int main () {
   int A[10] = {21, 8, 13, 55, 34, 5, 3, 2, 0, 1};
   bubblesort(A, 10);
   for (int i = 0; i < 10; i++) {
      printf("A[%d] = %d \n", i, A[i]);
   }
}
```

Bubble Sort – Analysis of Complexity

- Bubble sort uses about $n^2/2$ compares and $n^2/2$ data exchanges in the worst and average cases.
- The comparison `A[j-1] > A[j]` is always executed inside the `for` loops on `i` and `j`. The cost of the compares can be calculated using summation notation

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}$$

  Bubble sort always makes $O(n^2)$ compares.
- In the worst case for swaps, the file is in reverse order, and $O(n^2)$ swaps are required.
- In the best case for swaps, the file is in sorted order, and no swaps are required.
- In the average case for swaps, we need the probability that the `if` test evaluates to `True`.

Example: Bubble sort operations

Consider the six permutations of $\{2, 4, 7\}$. *Pretend each permutation of the keys occurs with equal likelihood* $1/6$.

$i = 2$ $j = 1$		$i = 2$ $j = 2$		$i = 1$ $j = 1$		
order	*swaps*	*reorder*	*swaps*	*reorder*	*swaps*	*reorder*
2 5 7	*0*		*0*		*0*	
2 7 5	*0*		*1*	*2 5 7*	*0*	
5 2 7	*1*	*2 5 7*	*0*		*0*	
5 7 2	*0*		*1*	*5 2 7*	*1*	*2 5 7*
7 2 5	*1*	*2 7 5*	*1*	*2 5 7*	*0*	
7 5 2	*1*	*5 7 2*	*1*	*5 2 7*	*1*	*2 5 7*
$P(swap) = \frac{1}{2}$		$P(swap) = \frac{2}{3}$		$P(swap) = \frac{1}{3}$		

There are (3+4+2) = 9 swaps for the 6 cases. The average number of swaps is

$$\frac{9}{6} = \frac{3}{2} = \frac{3(3-1)}{4} = \frac{n(n-1)}{4}$$

Let's see if this example provides the general rule.

- On the first pass of the outer loop $i = (n-1)$ and $j = 1, \ldots, (n-1)$. the `if` test will be `True` at any given $1 \leq j \leq i$ if and only if

$$\max\{\{A[0], A[1], \ldots, A[j-1]\} > A[j]$$

- This will be `True` if the largest value from the set

$$\{\{A[0], A[1], \ldots, A[j]\}$$

  is in any of the first `j` of `j+1` positions

- Therefore, the probability that the `if` test evaluates to `True`, during the first pass on `i=n-1` for $j = 1, 2, \ldots, (n-1)$ is

$$\frac{j}{j+1} = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \cdots, \frac{n-1}{n}$$

- On the next pass of the outer loop $i = (n-2)$ and the `if` test will be `True` if and only if

$$\max\{\{A[0], A[1], \ldots, A[j-1]\} > A[j]$$

  and

$$\max\{\{A'[0], A'[1], \ldots, A'[j]\} > A'[j+1]$$

  where the primed values refer to the original array values

- The probability that the `if` test evaluates to `True` on this second pass is

$$\frac{j+1}{j+2} \cdot \frac{j}{j+1} = \frac{j}{j+2}$$
$$= \frac{1}{3}, \frac{2}{4}, \frac{3}{5}, \ldots, \frac{n-2}{n} \quad \text{for } j = 1, \ldots, n-2$$

- In general, the probability that the `if` test evaluates to `True` on the k^{th} pass $k = 1, 2, \ldots, (n-1)$ where $i = n - k$ is

$$\frac{j}{j+k}, \quad \text{for } j = 1, \ldots, n-k$$

- Average case cost is computed by the formula

$$\sum_{\text{all cases}} \text{Prob(case)} \cdot \text{Work(case)}$$

- The average number of swaps is

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i} \frac{j}{j+(n-i)}(1) = \begin{array}{lllllll|l} \frac{1}{2} & + & \frac{2}{3} & + & \frac{3}{4} & + & \cdots + & \frac{n-1}{n} & i = n-1,\ j = 1, \ldots (n-1) \\ + & \frac{1}{3} & + & \frac{2}{4} & + & \cdots + & & \frac{n-2}{n} & i = n-2,\ j = 1, \ldots (n-2) \\ & & + & \frac{1}{4} & + & \cdots + & & \frac{n-3}{n} & i = n-3,\ j = 1, \ldots (n-3) \end{array}$$
$$\vdots$$
$$\begin{array}{lll|l} + \frac{1}{n-1} + & & \frac{2}{n} & i = 2,\ j = 1, 2 \\ + & & \frac{1}{n} & i = 1,\ j = 1 \end{array}$$

$$= \frac{1}{2} + \frac{1}{3}(1+2) + \frac{1}{4}(1+2+3) + \cdots \frac{1}{n}(1+2+3+\cdots+(n-1))$$
$$= \frac{1}{2} + \frac{1}{3}\left(\frac{2 \cdot 3}{2}\right) + \frac{1}{4}\left(\frac{3 \cdot 4}{2}\right) + \cdots \frac{1}{n}\left(\frac{n(n-1)}{2}\right)$$
$$= \frac{1}{2}(1+2+3+\cdots+(n-1))$$
$$= \frac{n(n-1)}{4}$$

Exercises

1. Consider the insertion sort algorithm below. Assume $A[0]$ is a *sentinel*, smaller every other element in the array.

Listing 27: Imperative Insertion sort

109 ⟨*Insertion sort* 109⟩≡

```
void InsertionSort(int A[], int n)
{
  int i, j, v;
  for (i = 2; i < n; i++) {
    v = A[i];
    j = i;
    while (A[j-1] > v) {
      A[j] = A[j-1];
      j = j - 1;
    }
    A[j] = v;
  }
}
```

(a) Compute the average number of comparisons (executions of the `while` loop) on the 6 permutations of {0, 2, 5, 7}, where 0 is held fixed as the first element, the sentinel. Construct a table, similar to the table in example to show your computations.

(b) Find a general formula for the average number of comparisons in the inserttion sort algorithm.

10. Insertion Sort

Insertion Sort

It is said that insertion sort is how many sort a hand of cards. An empty hand is sorted. A single, first card, is sorted. As every other card is dealt, insert it into the previously sorted hand.

The functional `insertion_sort` code below uses a helper function to perform the insertion.

To insertion sort a list `x:xs`,

- Insertion sort the tail `xs`.
- Insert `x` at the head if it is smaller than the head of the insertion sorted list `(y:ys)`.
- Otherwise, leave `y` as the head of the sorted list and insert `x` in the tail `ys`.

Listing 28: Functional Insertion Sort

111a ⟨*Functional insertion sort* 111a⟩≡

```
insertion_sort          :: Ord a => [a] -> [a]
insertion_sort []       = []
insertion_sort [x]      = [x]
insertion_sort (x:xs) = insert (insertion_sort xs)
    where insert [] = [x]
          insert (y:ys) | x <= y    = x : y : ys
                        | otherwise = y : insert ys
```

An imperative implementation is given below. It assumes a *sentinel*, $A[0]$, that is smaller every other element in the array. The invariant of the algorithm is that after the ith step, the array $A[0..i]$ is sorted.

Listing 29: Imperative Insertion sort

111b ⟨*Insertion sort* 111b⟩≡

```
void InsertionSort(int A[], int n) {
  int i, j, v;
  for (i = 2; i < n; i++) {
    v = A[i];
    j = i;
    while (A[j-1] > v) {
      A[j] = A[j-1];
      j = j - 1;
    }
    A[j] = v;
```

```
    }
  }
```

Insertion Sort – Analysis of Complexity

The time complexity of the imperative algorithm can be computed using these observations:

- The outer `for` loop on `i` is executed $n-1$ times
- There are two initial assignment in this outer loop, a `while` loop, and a final assignment
- The comparison in the `while` loop Boolean expression may execute as many as `i` times and as few as 0 times
- In the *best case* (the data is sorted in ascending order), insertion sort will execute $3n-3$ assignments and $n-1$ tests of the Boolean expression in the `while` loop. Thus, the time complexity is $O(n)$
- In the *worst case* (the data is in descending order), the `while` loop executes `i` times making `i` evaluations of the Boolean expression and 2 assignments on each pass of the loop. This occurs for every value of `i` from 2 to `n`, thus the total complexity is

  $$3n-3+\sum_{i=2}^{n} 3i = 3n-3+3\left[\frac{n(n+1)}{2}-1\right]$$

  The time complexity is $O(n^2)$
- For the *average case*, we need the probability that `k` compares are made in the `while` test for $k=1$ to `i`
- For given $k=1,\ldots,i$ there will be `k` compares if and only if

  $$A[i-1] > A[i],\ A[i-2] > A[i], \ldots, A[i-k+1] > A[i]$$

  and

  $$A[i-k] \leq A[i]$$

- That is, $A[i]$ is the kth largest element in the array

  $$A[1],\ A[2],\ \ldots,\ A[i-1],\ A[i]$$

- The probability of this is $1/i$: There is one out of `i` positions to place the kth largest element

- Thus the average number of comparisons is

$$\begin{aligned}\sum_{i=2}^{n}\sum_{k=1}^{i}\frac{1}{i}(k) &= \sum_{i=2}^{n}\frac{i+1}{2}\\ &= \frac{(n+1)(n+2)}{4}-\frac{3}{2}\end{aligned}$$

- The average case complexity is $O(n^2)$

Example: Insertion sort operations

$i=2$	*order*	*compares*	*reordered*	$i=3$	*compares*
	0 2 5 7	*1*			*1*
	0 2 7 5	*1*			*2*
	0 5 2 7	*2*	*0 2 5 7*		*1*
	0 5 7 2	*1*			*3*
	0 7 2 5	*2*	*0 2 7 5*		*2*
	0 7 5 2	*2*	*0 5 7 2*		*3*

For $i=2$, there is $\frac{9}{6}$ comparisons, on average
For $i=3$, there are $\frac{12}{6}$ comparisons, on average
The average number of comparisons, over all i, is

$$21/6 = 7/2 = \frac{(3+1)(3+2)}{4}-\frac{3}{2} = 5-\frac{3}{2}$$

Exercises

1. Consider the selection sort algorithm below. Explain the code and analyze its time complexity.

Listing 30: Functional Selection Sort

113 ⟨*Functional Selection sort* 113⟩≡

```
select_sort :: (Ord a) => [a] -> [a]
select_sort [] = []
select_sort xs = let x = maximum xs
  in select_sort (remove x xs) ++ [x]
  where remove _ [] = []
        remove a (x:xs)
          | x == a    = xs
          | otherwise = x : remove a xs
```

2. Consider the selection sort algorithm below.

Listing 31: Imperative selection sort

114 ⟨*Imperative selection sort* 114⟩≡

```
void selectionsort(int  A[], int n)
{
  int i, j, min;
  for (i = 1; i < n; i++) {
    min = i;
    for (j = i+1; j < n; j++) {
      if (A[j] < A[min]) {
        min = j;
        swap(A[min], A[i]);
      }
    }
  }
}
```

(a) Compute the average number of swaps on the 6 permutations of $\{2, 5, 7\}$. Construct a table, similar to the table in example to show your computations.

(b) Find a general formula for the average number of comparisons in the insertion sort algorithm.

11. Backtracking Algorithms

Introduction to backtracking

Backtracking algorithms are often used to solve *constraint satisfaction* problems. The 0—1 Knapsack Problem is an example problem that can be solved by backtracking. (There are other approaches (greedy, dynamic programming) for this problem.)

Conceptually, backtracking performs a depth-first search of a tree.

Problem 5: 0—1 Knapsack Problem

Given a knapsack with capacity C*, and a list of provisions (an inventory)* $\langle p_k : k \in \mathbb{N} \rangle$ *(the list could be unbounded) A provision* p *is a 3–tuple*

$$p :: (\mathrm{String}, \mathrm{Num}, \mathrm{Num}) = (\textit{name}, \textit{weight}, \textit{value})$$

Decision Problem: Given a value V*, is there a subset* $\mathbb{I}$ *of provisions such that*

$$\sum_{p \in \mathbb{I}} p_{\textit{weight}} \leq C, \quad \textit{and}$$
$$\sum_{p \in \mathbb{I}} p_{\textit{value}} \geq V$$

This decision problem is NP*-complete.*
Function Problem: Find the maximum value of V *over all* feasible *subsets of provisions. A subset is feasible if the sum of its weights does not exceed the capacity* C *of the knapsack. This function problem is* NP*-hard.*

The Partition problem: Can a set integers be partitioned into two non-empty subsets that have equal sums over their values. It is an NP-complete problem. Partition is a special case of the Knapsack problem: Assume each items weight equals its value and $C = V = 1/2 \sum$ weights. Solving Knapsack in this special case solves Partition.

A useful abstraction is to consider both summations to be over the entire inventory (list of provisions). This is accomplished using a a bit vector

$$\langle b_0, b_1, b_2, b_3, \ldots \rangle$$

where a particular bit b_k is set to 1 or 0 depending on whether or not provision p_k is or is not placed in the knapsack. There are several things this reveals.

- There are 2^n bit vectors of length n (or subsets of an n element set

$\mathbb{Z}_n = \{0, 1, \ldots, (n-1)\}$. Exhaustive search will take exponential time.

- Not all subsets need to be explored: Once a subset is *infeasible* so are all of its supersets.

Here is a functional algorithm for the problem. An example inventory is given. The combs function that searches over all combinations of provisions from the inventory (subsets of it). The main function specifies the input and output.

From The Haskell code is from Rosetta Code

The combs function maps a list of provisions and a capacity to and ordered pair: a value and its feasible list of provisions.

As an initial condition, if the provision list is empty, then for any capacity, the returned value is 0 and the empty list.

Given an non-empty list of provisions, there are two possibilities: (1) If the weight of the provision at the head of the list is less than the capacity, then including the provision is feasible, otherwise (2) the provision cannot be in a feasible solution.

In the second case, return the result from the rest of the list and the given capacity. This is the case where b_p, the bit representing the provision, is 0. In the first case, provision may or may not be in the optimal feasible solution. So, compute both cases and return the one that is largest.

Listing 32: Functional 0 –1 Knapsack

```
-- A provision is its name, its weight, and its value
data Provision = (String, Num, Num)

-- Here is a sample inventory of provisions
inventory = [("map",9,150), ("compass",13,35), ("water",153,200),
             ("sandwich",50,160), ("glucose",15,60), ("tin",68,45),
             ("banana",27,60), ("apple",39,40), ("cheese",23,30),
             ("beer",52,10), ("cream",11,70), ("camera",32,30),
             ("tshirt",24,15), ("trousers",48,10), ("socks",4,50),
             ("umbrella",73,40), ("towel",18,12), ("book",30,10),
             ("trousers",42,70), ("overclothes",43,75),
             ("notecase",22,80), ("sunglasses",7,20)]

 -- The combs function searches over feasible solutions to find one
 -- that maximizes the value of provisions
 combs [] _ = (0, [])
 combs ((n,w,v):rest) cap
    | w <= cap  = max (combs rest cap)
                      (prepend (n,w,v) (combs rest (cap - w)))
    | otherwise = combs rest cap
    where prepend (n,w,v) (value, list) = (value + v, (n,w,v):list)

main = do print (combs inventory 400)
```

Given a list of n provisions combs always calls itself again with a list of size $n-1$, and sometimes calls itself twice on the tail of the list with

different capacities. Prepending a triple onto the current optimal value and list takes constant time. Therefore, in the worst case, the code's time complexity can be modeled by the famous Mersenne recurrence

$$T(n) = 2T(n-1) + 1,\ T(0) = 0$$

which has solution $T(n) = 2^n - 1$.

Here is a C (pseudo-code) implementation backtracking for the problem.

A more general model would be

$$T(n) = 2T(n-1) + c,\ T(0) = a$$

which has solution

$$T(n) = 2^n a + (2^n - 1)c$$

I believe comparing the functional and imperative codes clearly shows the difference between understanding the problem versus understanding the problem and the machine.

Listing 33: Imperative 0 –1 Knapsack Backtracking Algorithm

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  char *name;
  int weight;
  int value;
} provision;

provision items[] = {
  {"map",          9, 150},
  {"compass",     13,  35},
  {"water",      153, 200},
  ...,
  {"sunglasses",  7,  20}
};

int cap;            // capacity of knapsack
int n;              // number of items
int X[n];           // current array of bits
int optBits[n];     // optimal array of bits
int optValue = 0;

bool isFeasible(provision *items) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum = sum + (X[i] * items[i].weight);
  }
  if (sum <= cap) { return true; }
  else {return false;}
}

bool betterValue(provisions *items) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum = sum + (X[i] * items[i].value);
  }
  if (sum > optValue) {
    optValue = sum;
    return true;
```

```
  }
  else {return false;}
}

int knapsack(provisions *items, int level) {
  if (level == n) {
    if (isFeasible(items)) {
          if (betterValue(items)) {
            optBits = X;
          }
        }
      }
      else {
        X[level] = 1;
        knapsack(items, level + 1);
        X[level] = 0;
        knapsack(items, level + 1,);
      }
    }
```

The above code has time complexity described by

$$\begin{aligned}
T(n) &= 2T(n-1) + n \\
&= 2[2T(n-2) + (n-1)] + n \\
&\vdots \\
&= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i (n-i) \\
&= \sum_{i=0}^{n-1} 2^i (n-i) \\
&= n(2^n - 1) - \sum_{i=0}^{n-1} 2^i i \\
&= n(2^n - 1) - (2 + 2^n(n-2)) \\
&= 2^{n+1} - n - 2
\end{aligned}$$

Here is another imperative C algorithm for the $0-1$ Knapsack. It uses dynamic programming and comes from Rosetta Code. It makes me not want to be a C programmer. This algorithm is pseudo-polynomial, that is, its time complexity is $T(n) = O(wn)$ where w is the capacity of the knapsack.

Pseudo-polynomial means the time complexity depends as a polynomial in value of a number, the capacity w in this case, but the input size depends on the $\lfloor w \rfloor + 1$, and w is exponential in this size.

Listing 34: Imperative Dynamic Programming 0–1 Knapsack Algorithm

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *name;
    int weight;
    int value;
} provision;

provision items[] = {
    {"map",                      9,   150},
    {"compass",                 13,    35},
    {"water",                  153,   200},
    ...,
    {"sunglasses",               7,    20}
};

int *knapsack(provision *items, int n, int w) {
    int i, j, a, b, *mm, **m, *s;
    mm = calloc((n + 1) * (w + 1), sizeof (int));
    m = malloc((n + 1) * sizeof (int *));
    m[0] = mm;
    for (i = 1; i <= n; i++) {
        m[i] = &mm[i*(w+1)];
        for (j = 0; j <= w; j++) {
            if (items[i-1].weight > j) {
                m[i][j] = m[i-1][j];
            }
            else {
                a = m[i-1][j];
                b = m[i-1][j-items[i-1].weight]+items[i-1].value;
                m[i][j] = a > b ? a : b;
            }
        }
    }
    s = calloc(n, sizeof (int));
    for (i = n, j = w; i > 0; i--) {
        if (m[i][j] > m[i-1][j]) {
            s[i-1] = 1;
            j -= items[i-1].weight;
        }
    }
    free(mm);
    free(m);
    return s;
}

int main () {
    int i, n, tw = 0, tv = 0, *s;
    n = sizeof (items) / sizeof (provision);
    s = knapsack(items, n, 400);
    for (i = 0; i < n; i++) {
```

```
      if (s[i]) {
        printf("%-22s %5d %5d\n", items[i].name,
                                  items[i].weight,
                                  items[i].value);
        tw += items[i].weight;
        tv += items[i].value;
      }
    }
    printf("%-22s %5d %5d\n", "totals:", tw, tv);
    return 0;
  }
```

Pruning and Bounding Functions

The functional backtracking algorithm for Knapsack 32 *prunes* the search space by only exploring branches where `w <= cap`. The imperative backtracking algorithm ?? explores the entire search space, but the search can be *pruned* by a guard before each recursive call: Does the current weight plus the weight of the next provision not exceed the capacity?

Bounding functions provide more general approaches to pruning.

Let Let $\vec{X} = \langle x_0, x_1, \ldots, x_{k-1}, \ldots \rangle$ be a k-bit string representing a (partial) solution to a constraint satisfaction problem. Let

$$C(\vec{X}) = \left(\sum_{i=0}^{k-1} v_i x_i \right) + \max \left\{ \left(\sum_{i=k}^{n} v_i x_i : \sum_{i=0}^{n-1} w_i x_i \right) \leq C \right\}$$

That is, $C(\vec{X})$ is the maximum value of *feasible descendants (extensions)* of $\vec{X}$. It is the value of the currently selected provisions plus the largest value over the set of feasible extensions (descendants) of $\vec{X}$.

In particular, if $\left|\vec{X}\right| = n$, then $\vec{X}$ is a feasible solution, $C(\vec{X})$ is its value, but $\vec{X}$ may not be optimal. Also, if $\left|\vec{X}\right| = 0$, then $C(\vec{X})$ is the optimal value of the problem.

Definition 12: Bounding Function Properties

A bounding function B *is any function defined on variable length bit strings such that*

- *If* $\vec{X}$ *is a* feasible *solution, the* $C(\vec{X}) = B(\vec{X})$
- *For all* partial feasible *solutions,* $C(\vec{X}) \leq B(\vec{X})$

If such a bounding function $B()$ can be found, and if at any point of the computation $B(\vec{X}) \leq$ `optValue` holds, then no extensions of $\vec{X}$ can lead to an optimal solution. And, searching descendants of $\vec{X}$ can be pruned.

Computing $C(\vec{X})$ is expensive when $\vec{X}$ has many descendants. The bounding function $B()$ should be much easier to compute. And, we want $B()$ be be a good approximation of $C()$.

One trick that can lead to discovering a bounding function is to find a simpler, easier to solve, related problems. A natural approximation to the 0–1 Knapsack problem is the Rational Knapsack problem (RK)

Recall, the greedy approach to the RK problem: Given a list of items, sort them in descending order in their value-to-weight ratios. An item with value 10 and weight 3 is worth more than an item with value 10 and weight 4. This sort only needs to be done once. Assume its time complexity is $O(n \lg n)$.

The greedy algorithm places items in the knapsack in order, one at a time as long as they fit. Some fraction of the last item might need to be used to fill, but not overfill, the knapsack. The time complexity is $O(n)$. See the notes on Greedy algorithms.

Let $\vec{X} = \langle x_0, x_1, \ldots, x_{k-1}, \ldots \rangle$ be a string of k-bit strings representing a (partial) solution to a Knapsack problem. Let $R(k, C')$ be the optimal solution to the Rational Knapsack problem with capacity C', over all rational descendants of $\vec{X}$, that is, $\langle x_k, x_{k+1}, \ldots, x_{n-1} \rangle$ where the values of $x_j \in \mathbb{Q}$, the set of rationals.

Define a bounding function by

$$B(\vec{X}) = \sum_{i=0}^{k-1} x_i p_i + R\left(k, C - \sum_{i=0}^{k-1} x_i w_i\right) = CV + R(k, C - CW)$$

where CV is the current value and CW is the current weight.

That is, $B(\vec{X})$ is the value selected provision from 0 to $k-1$, plus the value that can be gained from the remaining provisions using the remaining capacity and rational x's. When all of the x's are restricted to 0 or 1, then $C(X) = B(X)$. Also, since rational x's yield more freedom (choices), $C(X) \leq B(X)$

Here is an example from (Stinson, 1987).

Example: Use of bounding function

Assume there are 5 items with weights

$$\vec{W} = \langle 11, 12, 8, 7, 9 \rangle$$

and values

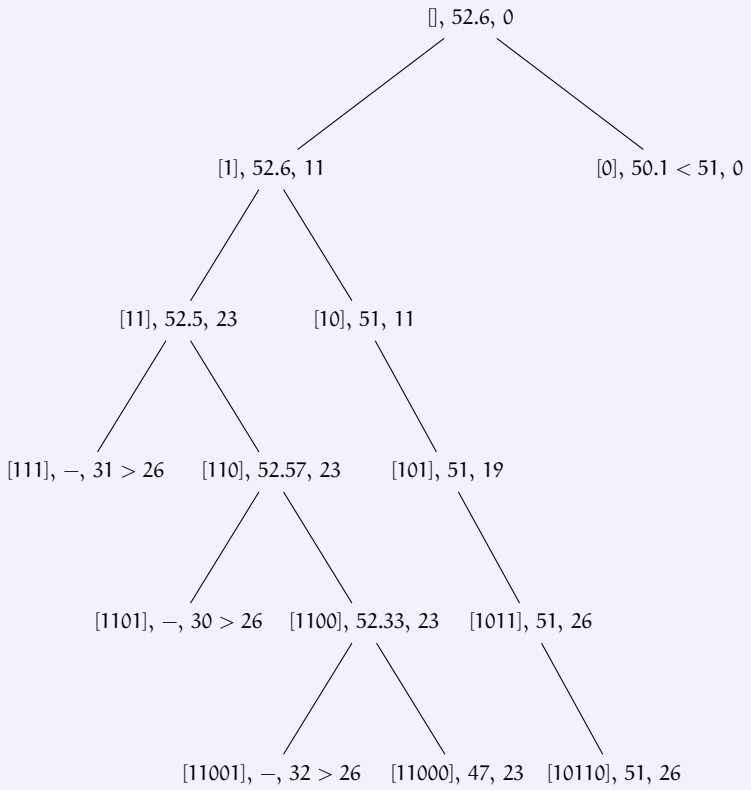
$$\vec{V} = \langle 23, 24, 15, 13, 16 \rangle$$

Pretend the knapsack's capacity of $C = 26$. *The weights and values are sorted by value-to-weight ratio:*

$$\langle 23/11, 24/12, 15/8, 13/7, 16/9 \rangle \approx \langle 2.09, 2, 1.875, 1.857, 1.77 \rangle$$

The search space tree shown below and explained after the diagram. A node is a triple $([xs], B, CW)$, *a list* $[xs]$ *of previously*

set bits, the value of the bounding function B, *and the current weight of the included items.*



Assume that the positive $x = 1$ branch is explored first. The greedy algorithm first computes x's: 1, 1, 3/8 to fill the knapsack. The bounding value is

$$B([]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625$$

Now explore the 1 branch:

$$B([1]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625, \quad CW = 11$$

- $B([11]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625$, $cw = 23$
  - $B([111])$ is infeasible: $cw = 31 > 26 = C$, prune this branch
  - $B([110]) = 23 + 24 + \frac{3}{7} \cdot 13 \approx 52.57$, $cw = 23$ (The left (down) branch [1101] is infeasible: its weight is $CW = 30$)

    The search follows $[110] \mapsto [1100] \mapsto [11000]$: a feasible solution. Along this branch B is updated: $52.57 \mapsto 52.33 \mapsto 47$ and a potential optimal value `optValue=47` is set.
- Now explore the [10] branch. $B([10]) = 23 + 15 + 13 \approx 51$, $cw = 26$

- $B([101]) = 51$, $CW = 26$
  - $B([1011]) = 51$, $CW = 26$
    - $[10111]$ is infeasible: $CW = 11 + 8 + 7 + 9 = 35 > 26 = C$
    - $[10110]$ is feasible: $B([10110]) = 51$, $CW = C$, and a new, better, potential optimal value `optValue=51` is set.
  - $B([100]) = (23) + 13 + \frac{2}{3}16 = 46.6 < 51$, $CW = 11$. Since this value of B is less than the previously computed optimal value 51 this branch can be pruned.

- When the $[0]$ branch is explored, we find

$$B([0]) = 0 \cdot 23 + 24 + 15 + \frac{6}{7} \cdot 13 \approx 50.14 < 51$$

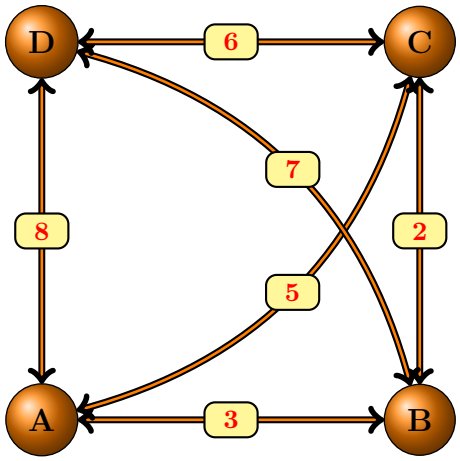
  Since this value of B is less than the previously computed potential optimal solution 51 its entire sub-tree can be pruned

The Traveling Salesman & Hamilton Circuit Problem

12. The Traveling Salesman Problem

The Traveling Salesman & Hamiltonian Circuit Problem

A Hamiltonian tour or circuit in an undirected graph $G = (E, V)$ is a cycle that passes through each node exactly once. When the edges have non-negative weights, a traveling salesman want to find the shortest tour. Name the nodes 0 through $n - 1$. Since the tour is a cycle, the first node is arbitrary, and might as well be 0. A tour is a permutation of $1, 2, \ldots, (n - 1)$, so there are $(n - 1)!$ possible tours. But, the graph is undirected, so a permutation $[0, 1, 2, 3, 4, 5]$ is equivalent to $[0, 5, 4, 3, 2, 1]$.



Searching through all permutations is only reasonable for small problems. If the graph is not complete (fully connected), then some tours can be pruned. Let's develop a *bounding function* that is more efficient at pruning the search space.

An adjacency matrix is one representation of a graph. For example,

$$M = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 3 & 5 & 8 \\ 3 & \infty & 2 & 7 \\ 5 & 2 & \infty & 6 \\ 8 & 7 & 6 & \infty \end{pmatrix} \end{array}$$

An adjacency matrix can be *reduced* in two steps: First subtract the

minimum in each row from itself and others in the row. Next, using this intermediate matrix, subtract the minimum in each column from itself and the other values in the column. Keep a running tab of the amount subtracted: Call the value $V(M)$.

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 3 & 5 & 8 \\ 3 & \infty & 2 & 7 \\ 5 & 2 & \infty & 6 \\ 8 & 7 & 6 & \infty \end{pmatrix} \end{array} \mapsto \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 0 & 2 & 5 \\ 1 & \infty & 0 & 5 \\ 3 & 0 & \infty & 4 \\ 2 & 1 & 0 & \infty \end{pmatrix} \end{array} \mapsto \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 0 & 2 & 1 \\ 0 & \infty & 0 & 2 \\ 2 & 0 & \infty & 0 \\ 1 & 1 & 0 & \infty \end{pmatrix} \end{array}$$

With $V(M) = (3 + 2 + 2 + 6) + (1 + 0 + 0 + 4) = 18$. The resulting matrix has:

- All non-negative entries
- Every row and every column contains at least one 0.

$V(M)$ is a lower bound on the cost of any Hamiltonian circuit H. To see this, let

$$H = [0x_1x_2 \cdots x_{n-1}] = [ax_1x_2 \cdots x_{n-1}]$$

be any tour: $0 \mapsto x_1 \mapsto x_2 \mapsto \cdots \mapsto x_{n-1} \mapsto 0$. That is, a permutation on the set $\{1, 2, \ldots, (n-1)\}$. Therefore, the adjacency matrix entries $M[0, x_1]$, $M[x_1, x_2]$, $\ldots$, $M[x_{n-1}, 0]$ cover all rows and all columns. The sum of these entries is the cost of the tour H.

$$C(H) = \sum_{i=1}^{n} M[x_{i-1}, x_i] \qquad x_0 = x_n = 0$$

For each row, subtract $M[x_{i-1}, x_i]$ from each entry in row $(i-1)$. The amount subtracted is the cost of the tour $C(H)$. Consider the tour $H = [0, 2, 1, 3] = [a, c, b, d]$ of the graph. It's cost is $5 + 2 + 7 + 8 = 22$

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 3 & 5 & 8 \\ 3 & \infty & 2 & 7 \\ 5 & 2 & \infty & 6 \\ 8 & 7 & 6 & \infty \end{pmatrix} \end{array} \mapsto \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & -2 & 0 & 3 \\ -4 & \infty & -5 & 0 \\ 3 & 0 & \infty & 4 \\ 0 & -1 & -2 & \infty \end{pmatrix} \end{array}$$

The resulting matrix has a 0 in every row and column. But, some of its entries are negative. This process reduced the matrix by more than the reduction that computes $V(M)$.

$$V(M) \leq C(H)$$

That is, $V(M)$ is a lower bound on any tour.

Suppose the cost of some tour has been computed, yielding a potential minimum value. Say $[a, b, c, d]$ which has cost $3+2+6+8 = 19$. Backtracking would then compute the cost of $[a, b, d, c]$ which has cost $3 + 7 + 6 + 5 = 21$. The value 19 is kept as the optimal so far.

Now suppose some other tour has been started, to see if a shorter tour can be found. For example $[a, c, \ldots]$, with a cost $5 + \cdots$. We want to extend it to completion. There are two of completions:

$$[a, c, b, d] \quad \text{and} \quad [a, c, d, b]$$

The problem (matrix) can be reduced by eliminating visited rows and columns.

- strike the columns of all visited nodes, except the first (we need to get back to it)
- strike the rows of all visited nodes, except the last (we need to go forward from it)
- set $M[x_i, 0] = \infty$ for each visited node (we don't want to return to the tour start from an already visited node)

$$M = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \infty & 3 & 5 & 8 \\ 3 & \infty & 2 & 7 \\ 5 & 2 & \infty & 6 \\ 8 & 7 & 6 & \infty \end{pmatrix} \end{array} \mapsto M' = \begin{array}{c} \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccc} a & b & d \end{array} \\ \begin{pmatrix} 3 & \infty & 7 \\ \infty & 2 & 6 \\ 8 & 7 & \infty \end{pmatrix} \end{array}$$

It can be shown a bounding function is

$$B(X) = V(M') + C(X)$$

That is,

- $B(X) = C(H)$ if H is a tour. In this case, all rows and columns have been stricken and M' is an empty matrix.
- $B(X) \leq C(X)$ for any feasible partial solution ($B(X)$ is a lower bound on the cost of a tour that starts with path X.)

In the example $[a, c, \ldots]$, $V(M') = (3 + 2 + 7) + 4 = 16$ and $C(X) = 5$. Since $B(X) = 21 > 19$ the branch can be pruned.

Exercises

1. How the algorithm runs on the tour that starts $[a, d, \ldots]$.

13. Medians and Order Statistics

Please read chapter 8 Medians and Order Statistics in the textbook (Corman et al., 2009).

Definition 13: Orders and Medians

Let $\mathbb{A}$ be a set of n values from a totally ordered set. The i^{th} order statistic *is the i^{th} smallest value. For instance, the first order statistic is the* minimum *of $\mathbb{A}$. The* maximum *of $\mathbb{A}$ is the n^{th} order statistic. The* median *is the "halfway point." If n is odd, then the median occurs at $m = (n+1)/2$. If n is even, then there are two medians one at $m_0 = n/2$ and one at $m_1 = n/2 + 1$.*

The *Selection Problem* is to find the i^{th} order statistic for a given i, where $1 \leq i \leq n$.

Describe an $\Theta(n \lg n)$ algorithm that solves the selection problem.

Problem 6: The Selection Problem

Let n be a positive integer and let $\mathbb{A}$ be a set of n values from a totally ordered set.
Decision Problem: Is the value $x \in \mathbb{A}$ larger than exactly $i-1$ other elements in $\mathbb{A}$?
Function Problem: Let $1 \leq i \leq n$. Find the element $x \in \mathbb{A}$ that is larger than exactly $i-1$ other elements in $\mathbb{A}$.

As an example, the elements in the set $\mathbb{A}$ have orders indicated below.

$$\begin{aligned} \mathbb{A} &= \{7, 12, 5, 17, 9, 1, 14, 8, 18\} \\ \text{Order} &= \{3, 6, 2, 8, 5, 1, 7, 4, 9\} \end{aligned}$$

A simple linear time algorithm that solves the *minimum* problem is shown below. It assumes $\mathbb{A}$ is represented as a 0-indexed array of n integers.

Describe an $\Theta(n)$ algorithm that simultaneously solve both the minimum and maximum problems.

Listing 35: Imperative Minimum

129 ⟨*Minimum algorithm* 129⟩≡

```
int minimum(int A[], int n) {
  int min = A[0];
```

```
    for (k = 1; k < n; k++) {
      if (min > A[k]) { min = A[k]; }
    }
    return min;
}
```

The time complexity of the above algorithm is $\Theta(n)$. The `for` loops $n - 1$ times taking a few cycles each time to decide whether or not to update the minimum. Since every element must be examined to determine the minimum, there can be no faster, deterministic algorithm.

A functional implementation to compute the `maximum` of a list might look like this:

The first line declares that the type of `maximum` to be a function that maps a list `[a]` of orderable values to a value of type `a`. The two base cases are: an empty list has no maximum and a singleton list has the value of the single element. Then, recursively, the maximum of a longer list is the head of the list if the head is larger maximum of the tail, otherwise it is the maximum of the list's tail.

Listing 36: Functional Maximum

130 ⟨*Maximum algorithm* 130⟩≡

```
maximum :: (Ord a) => [a] -> a
maximum [] = error "maximum of empty list"
maximum [x] = x
maximum (x:xs)
   | x > maxTail   = x
   | otherwise     = maxTail
     where maxTail = maximum xs

- max and min are in class Ord of the Haskell Prelude
- in
max x y
   | x <= y = y
   | otherwise = x
min x y
   | x <= y = x
   | otherwise = y

{- And these functions in the PreludeList module:
   Here the foldl (``left reduce'') function
   maps a list to its maximum value.  That is,
   foldl max [a, b, c, d] = max ((max (max a b) c) d)
-}
maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
```

```
maximum xs = foldl1 max xs
minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs
```

Randomized Selection

Using a randomized implementation of the partition function, described in the notes on Quicksort, an average case linear time algorithm can be developed for the i-th order problem. The *randomizing heuristic* is to swap the head of an array with a randomly selected element.

Listing 37: Imperative Randomized Partition

131 ⟨*Randomized Partition* 131⟩≡

```
#include <stdlib.h>
#include <stdio.h>

// partition is O(n) when lo = 0 and hi = n-1
// test need: A[j] > v for all j < hi
int partition(int A[], int lo, int hi) {
  int v, i, j, tmp;
  v = A[hi]; i = lo - 1; j = hi;
  for (;;) {
     while (A[++i] < v) ;
     while (A[- -j] > v) ;
     if (i >= j) break;
     // swap A[i] and A[j]
     tmp = A[i];
     A[i] = A[j];
     A[j] = tmp;
  }
  // swap A[i] and A[hi]
  tmp = A[i];
  A[i] = A[hi];
  A[hi] = tmp;
  return i;
}

// randomPartition is O(1)
int randomPartition(int A[], int lo, int hi) {
  int k = rand() % (hi - lo + 1) + lo;
  int tmp = A[lo];
  A[lo] = A[k];
  A[k] = tmp;
```

```
  return partition(A, lo, hi);
}

int main () {
   int A[10] = {21, 8, 13, 55, 34, 5, 3, 17, 12, 0};
   int p = randomPartition(A, 0, 9);
   printf("Partition about A[%d] = %d\n", p, A[p]);
   for (int i = 0; i < 10; i++) {
      printf("A[%d] = %d \n", i, A[i]);
   }
   return 0;
}
```

This partition computation can be completed in $O(n)$ time in a language that support direct access to array elements.

For a list-based language such as HASKELL, swapping elements can take take linear time. The `splitAt k xs` splits xs at index `k` returning two list

Why is $O(k)$ the time complexity of `splitAt k xs`?

(xs[0..(k-1)], xs[k..n])

The time complexity of `splitAt k xs` is $O(k)$. Forming `(head list2):list1` and appending it `[x]` cost $O(1+k)$, and then another $O(k+1)$ steps to appending `tail list2`.

Listing 38: Swap head with the `xs[k]`

132a ⟨*Swapping head with k-th element* 132a⟩≡

```
import Data.List

swapElem :: Int -> [a] -> [a]
swapElem _ [] = []
swapElem _ [x] = [x]
swapElem k (x:xs) = (head list2):list1 ++ [x] ++ tail list2
   where (list1, list2) = splitAt (k-1) xs
```

A functional implementation of randomized partition might look like this.

Listing 39: Functional Randomized Partition

132b ⟨*Functional Randomized Partition* 132b⟩≡

```
import System.Random

⟨Define partition about the head 133a⟩
⟨Make the head random, then partition 133b⟩
```

Two functions are needed: one to partition about the head, and a second to make the head random.

133a ⟨*Define partition about the head* 133a⟩≡

```
partition :: Ord a => [a] -> ([a], Int)
partition [] = ([], 0)
partition [p] = ([p], 1)
partition (p:xs) = (before ++ [p] ++ after, length before)
            where before = [x | x <- xs, x <= p]
                  after  = [x | x <- xs, x > p]
```

133b ⟨*Make the head random, then partition* 133b⟩≡

```
randomPartition :: Ord a => [a] -> ([a], Int)
randomPartition []  = ([], 0)
randomPartition [x] = ([x], 1)
randomPartition xs  =
   let (k, _) = randomR (0, length xs) (mkStdGen 10) :: (Int, StdGen)
   in let (first, second) = splitAt k xs
      in partition (second ++ first)
```

Example: Randomize Partition

Given an array and index

$$\mathbb{A} = \langle 7, 12, 5, 17, 9, 1, 14, 8, 18\rangle \quad \textit{and index} \quad \mathfrak{i} = 6$$

The sought value, the 6^{th} *small element in the list has value* 12.
Randomized selection might work something like this:

- *Pretend the random partition was about index* 4, *value* 9 *creating the array*

$$\langle 7, 5, 1, 8, 9, 12, 17, 14, 18\rangle$$

- *The value* 9 *occurs at the fifth order statistic, which is less than* 6.
- *Therefore, call random partition of the tail* $\langle 12, 17, 14, 18\rangle$ *and pretend* 17 *is randomly chosen as the pivot.*

  This results in the list $\langle 12, 14, 17, 18\rangle$ *where* 17 *is of order* 3 *in the sub-list and order* $5 + 3 = 8$ *in the original list.*
- *Next, because* $6 < 8$, *call random partition on the list* $\langle 12, 14\rangle$. *Here* 14 *will be the pivot. It has order* 2 *and* $8 - 2 = 6$. *Therefore,* 12 *is the* 6^{th} *order statistic.*

Listing 40: Randomized Selection

134 ⟨*Randomized Selection* 134⟩≡

```
#include <stdlib.h>

int randomSelect(int A[], int lo, int hi, int i) {
  if (lo == hi) { return A[lo]; }
  int q = randomPartition(A, lo, hi)
  int k = q - lo + 1;
  if (i == k) { return A[q]; }
  else {
    if (i < k) {
      return randomSelect(A, lo, q-1, i);
    }
    else {
      return randomSelect(A, q+1, hi, i-k);
    }
  }
}
```

In the best case, the array is partitioned at the halfway point each time. This leads to the recurrence relation

$$T(n) = T(n/2) + n, \quad T(1) = 0$$

which unrolls as:

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= T(n/4) + n/2 + n \\ &= T(n/8) + n/4 + n/2 + n \\ &= \vdots \\ &= T(1) + n/2^{p-1} + \cdots + n/4 + n/2 + n \quad \text{for some } p = \lg n \\ &= 2n\left(1 - \frac{1}{n}\right) \\ &= O(n) \end{aligned}$$

In the worst case, the array is always partitioned into a singleton and the rest of the array. This leads to the recurrence relation

$$T(n) = T(n-1) + (n-1), \quad T(1) = 0$$

which unrolls to the sum of the first n natural numbers, that is

$$T(n) = \sum_{i=1}^{n-1} i = \binom{n-1}{2} = O(n^2)$$

The textbook (Corman et al., 2009) contains a detailed analysis concluding that the average case time complexity is $O(n)$. It goes something like this.

Assume `randomSelection` selection returns any of the values $1 \leq k \leq n$ with equal liklihood, $1/n$. It calls itself sub-array of size q or `n-q-1`. In the worst case, assume the call is always to the largest sub-array. Then,

$$T(n) \leq \frac{1}{n}\sum_{q=1}^{n-2} T(\max(q, n-q-1)) + O(n)$$

Note $q > n-q-1$ implies $q > (n-1)/2$ and $q \leq n-q-1$ implies $q \leq (n-1)/2$

$$\max(q, n-q-1) = \begin{cases} q & \text{if } q > \lceil (n-1)/2 \rceil \\ n-q-1 & \text{if } q \leq \lfloor (n-1)/2 \rfloor \end{cases}$$

For instance, if n is even, say $n = 6$, the the terms $T(n-2)$, $T(n-3), \ldots, T(\lceil (n-1)/2 \rceil)$ occur twice in the sum.

And, when n is odd, say $n = 7$, the the terms $T(n-2)$, $T(n-3), \ldots, T(\lceil (n-1)/2 \rceil + 1)$ occur twice and $T(\lceil (n-1)/2 \rceil)$ occurs once in the sum.

In all cases

$$T(n) \leq \frac{2}{n}\sum_{q=\lfloor n-1 \rfloor/2}^{n-2} T(q) + O(n)$$

Assume $T(n) \leq cn$ for some constant c, and the $O(n)$ term is an for some a. Then

$$\begin{aligned} T(n) &\leq \frac{2c}{n}\sum_{q=1}^{n-2} q - \sum_{q=1}^{\lfloor n-1 \rfloor/2-1} q + an \\ &= \frac{2c}{n}\left(\frac{(n-2)(n-1)}{2} - \frac{(\lfloor (n-1)/2 \rfloor - 1)(\lfloor (n-1)/2 \rfloor)}{2}\right) + an \\ &\leq \frac{2c}{n}\left(\frac{(n-2)(n-1)}{2} - \frac{((n-1)/2-2)((n-1)/2-1)}{2}\right) + an \\ &= cn - \left(\frac{cn}{2} - c + \frac{9c}{2n} - an\right) \\ &\leq cn - \left(\frac{cn}{2} - c - an\right) \end{aligned}$$

Which is less than or equal to cn if

$$\left(\frac{cn}{2} - c - an\right) \geq 0$$

or

$$n \geq \frac{2c}{c-2a}$$

Thus, if $T(n) = O(1)$ for $n < 2c/(c-2a)$ then the average case time complexity of random select is $O(n)$.

Assume $n = 6$

$\max 1\ (n-2) = 4$
$\max 2\ (n-3) = 3$
$\max 3\ (n-4) = 3$
$\max 4\ (n-5) = 4$

Now assume $n = 7$

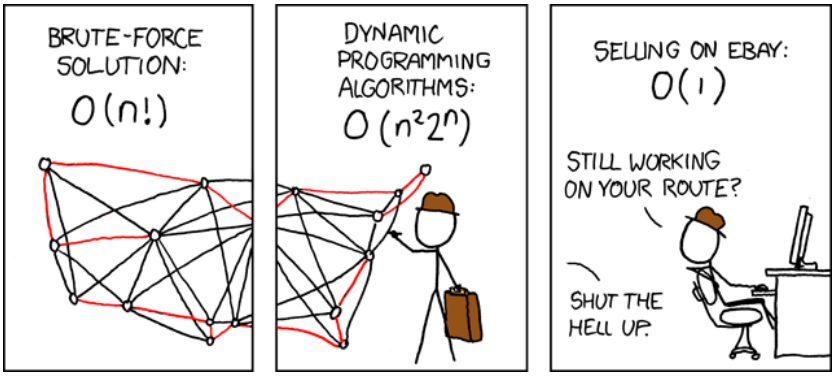
$\max 1\ (n-2) = 5$
$\max 2\ (n-3) = 4$
$\max 3\ (n-4) = 3$
$\max 4\ (n-5) = 4$
$\max 5\ (n-6) = 5$

The textbook (Corman et al., 2009) gives a more complex selection algorithm with worst case time complexity that is $O(n)$.

14. Dynamic Programming

Precomute, don't recompute

I wish I could remember the source of this quote



Traveling Salesman Problem

Please read Chapter 15 Dynamic Programming in in the textbook (Corman et al., 2009).

Dynamic programming is a problem solving methodology, credited to Richard Bellman (Bellman, 1957). A nutshell definition of dynamic programming is: Bottom-up computation with memorization. Problems that lend themselves to a dynamic programming attack have the following characteristics:

- A search over a large space for an optimal solution
- The optimal solution can be expressed in terms of optimal solutions to sub-problems.
- The number of sub-problems is small, saved in a memo (memorization).

Dynamic programming algorithms has the following features:

- A *recurrence* is implemented iteratively.
- A *table* is built to support the iteration by memorizing previously computed values.
- The optimal solution can be found by *tracing* through the table.

The most simple problem I know which where dynamic programming is useful is computing Fibonacci numbers. The recursive, top-down algorithm to compute F_n has exponential cost: $O(\varphi^n)$, where the golden ratio φ is about 1.618.

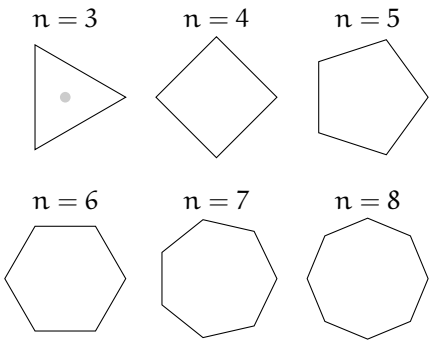
But, the iterative algorithm is linear, computing F_n in only $O(n)$ steps. The *table* is simply two values F_{n-2} and F_{n-1}, which are initialized to 0 and 1 when $n = 2$, and dynamically updated as the computation proceeds. Computing Fibonacci numbers is not a real optimization problem: There is no large search space, so the third point does not apply.

Polygon Triangulation

Here's an sample problem from computer graphics where dynamic programming is useful.

A polygon P with n vertices and n edges ($n \geq 3$) is a finite collection of vertices $v_0, v_1, \ldots, v_{n-1}$ lying in the Cartesian (x, y) plane with edges (v_i, v_{i+1}), $i = 0, 1, \ldots, (n-1)$ where $v_n = v_0$ to close the last side. Triangles, squares, pentagons are common polygons.

While writing this, I'm uncertain how to define the *empty polygon*. Is it a polygon with $n = 0$, 1 or 2 sides?



A polygon is simple if no edges cross one another. A polygon is convex if, given any two points on its boundary or interior, the line segment between them lies entirely within the polygon or its boundary. All of our polygons are simple and convex.

A chord $v_i\bar{v}_j$ is a line segment (not one of the sides) between two nonadjacent vertices, that is $v_j \neq v_{i+1}$ and if $j = n - 1$ then $v_i \neq v_0$. A *triangulation* T is a set of chords that partition a simple polygon into disjoint triangles. Triangle *fans* and *strips* are two simple ways to triangulate a polygon. Fans, as shown in Figure 10. Fans draw successive chords from a single vertex. Strips zigzag back and

Figure 10: Triangular fan

forth across the polygon, see Figure 11.

Figure 11: Triangular strip

In practice, some triangulations are better than others. Some optimization goals could be:

- Minimize the sum of triangle perimeters
- Minimize the largest area over all the triangles
- Relaxing the requirement that the polygons lie in a plane, you may want to minimize variation in surface normals

Problem 7: Minimal Triangulation

Assume a weight function $w(p_i, p_j, p_k)$ *defined on triangles* $\triangle = (p_i, p_j, p_k)$. *Let* T *be a triangulation of a polygon.*
Decision Problem: Does triangulation T *minimize the sum of weights*

$$\sum_{\triangle \in T} w(\triangle)?$$

Function Problem: Find optimal triangulations T, *those that minimize the sum of weights*

$$\sum_{\triangle \in T} w(\triangle)$$

Let's develop a recursion that describes an optimal triangulation.

Polygon Triangulation Recursion

Let's start by noting some useful facts.

1. If P is a polygon with n vertices, then every triangulation of P has $n-3$ chords and divides the polygon into $n-2$ triangles.
2. Each polygon edge belongs to some triangle.
3. Each triangle has one or two polygon edges.

Let $t(i, j)$ be the weight of the optimal triangulation of polygon $P_{ij} = (p_{i-1}, \ldots p_j)$, $1 \leq i \leq j \leq n-1$. That is,

Degenerate polygons have zero weight: $t(i, i) = 0$.

$$t(i, j) = \min \left\{ \sum_{\triangle \in T} w(\triangle) \right\}$$

where T is a triangulation of P_{ij}. We want to know the value of $t(1, n)$ and the triangulation that produces it.

In any triangulation of $(p_{i-1}, \ldots p_j)$ there must be one triangle (p_{i-1}, p_k, p_j) where $i \leq k \leq j-1$. By considering all of these we can reduce the current problem to find the minimum of $t(i, k) + t(k+1, j) + w(p_i, p_k, p_j)$, that is,

$$t(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min\{t(i, k) + t(k+1, j) + w(p_{i-1}, p_k, p_j) : i \leq k \leq j-1\} & \text{otherwise} \end{cases}$$

The Memoized Table

Using the recurrence we can fill out a table of weights $t(i, j)$ Here is a simple example: Let P a quadrilateral (p_0, p_1, p_2, p_3) with two triangulations who weights measure perimeters. See figure 12.

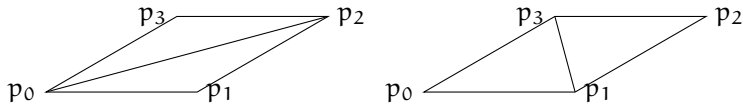


Figure 12: Triangulations with perimeter weights

The perimeters of the four triangles are:

$$w(p_0, p_1, p_2) = 8$$
$$w(p_0, p_2, p_3) = 8$$
$$w(p_1, p_2, p_3) = 5$$
$$w(p_0, p_1, p_3) = 5$$

We want to compute values $t(i, j)$ in table below.

	$j=1$	$j=2$	$j=3$
$i=1$	$t(1,1)$	$t(1,2)$	$t(1,3)$
$i=2$		$t(2,2)$	$t(2,3)$
$i=3$			$t(3,3)$

$=$

	$j=1$	$j=2$	$j=3$
$i=1$	0	8	10
$i=2$		0	5
$i=3$			0

The values along the main diagonal $t(k, k)$ can be initialized to 0. Along the next upper diagonal the values are:

$$\begin{aligned} t(1, 2) &= \min\{t(1, k) + t(k+1, 2) + w(p_0, p_k, p_2) : 1 \leq k \leq 1\} \\ &= \min\{0 + 0 + 8\} \\ &= 8 \\ t(2, 3) &= \min\{t(2, k) + t(k+1, 3) + w(p_1, p_k, p_3) : 2 \leq k \leq 2\} \\ &= \min\{0 + 0 + 5\} \\ &= 5 \end{aligned}$$

Lastly, the value of $t(1, 3)$ is

$$\begin{aligned} t(1, 3) &= \min\{t(1, k) + t(k+1, 3) + w(p_0, p_k, p_3) : 1 \leq k \leq 2\} \\ &= \min\{[t(1, 1) + t(2, 3) + w(p_0, p_1, p_3)], [t(1, 2) + t(3, 3) + w(p_0, p_2, p_3)]\} \\ &= \min\{[0 + 5 + 5], [8 + 0 + 8]\} \\ &= 10 \end{aligned}$$

The Trace back

Not only do we wish to to find the weight of the optimal triangulation, we want to be able to construct it as well. We can do this by recording the path to the optimal solution or dynamically reconstructing it.

Listing 41: Optimally Triangulate a Polygon

141a ⟨*Triangulate a polygon* 141a⟩≡

```
public void triangulate(Polygon poly) {
  ⟨Initialize triangulate local state 141b⟩
  ⟨For each diagonal 141c⟩ {
    ⟨For each row 141d⟩ {
      ⟨Initialize the column index and table entry 141e⟩
      ⟨Test every vertex k between i and j 142⟩
    }
  }
}
```

141b ⟨*Initialize triangulate local state* 141b⟩≡

```
int n = poly.countOfvertices();
double t[n][n];
for (int i = 0; i < n; i++) { t[i,i] = 0; }
```

141c ⟨*For each diagonal* 141c⟩≡

```
for (int d = 2;  d < n; d++)
```

141d ⟨*For each row* 141d⟩≡

```
for (int i = 1; i < n-d+1; i++)
```

141e ⟨*Initialize the column index and table entry* 141e⟩≡

```
Initialize the column index and »=
int j = i+d-1;
t[i, j] = INFINITY;
```

142 ⟨*Test every vertex* k *between* i *and* j 142⟩≡

```
for (int k = i; i < j; k++) {
  int q = t[i,k] + t[k+1,j] + poly.weight(i-1, k, j);
  if (q < t[i,j]) {
    t[i, j] = q;
    s[i, j] = k;
  }
}
```

This algorithm would be useful if its only application where in triangulating polygons. This basic algorithm solves many problems. See (Sankoff and Kruskal, 1983) for an overview of many problems that can be attacked by this method. Below the *edit distance* problem is described.

Edit Distance

Imagine applications where the distance between strings could be useful.

Spell checkers provide a list of *nearby* words when a string is not found in the dictionary. DNA, the molecule of life, can be abstracted as strings over the alphabet

$$\mathbb{DNA} = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$$

Geneticists study the similarities and differences in the DNA of among members of a species and between different species.

The similarity of two strings can be measured by an edit distance. Many different measures can be used. A simple edit distance is the number of

- Insertions: $\alpha\beta \mapsto \alpha\gamma\beta$
- Deletions: $\alpha\gamma\beta \mapsto \alpha\beta$
- Substitutions: $\alpha\gamma\beta \mapsto \alpha\delta\beta$

need to transform one string into another.

For instance, to map ALGORITHM to ALGEBRA might result in this editing sequence, where s, i and d stand for *substitution*, *insertion*, and *deletion*.

Is 6 the smallest unit cost distance between ALGORITHM and ALGEBRA?

A	L	G	O	-	R	I	T	H	M
A	L	G	E	B	R	A	-	-	-
			s	i		s	d	d	d

If each operation has a cost of 1, then the distance of this editing sequence is 6. Another metric charges 2 for a substitution.

Problem 8: String Edit Distance Problem

Decision Problem: *Given strings s and t, is m the minimum number of edits to transform s into t?*
Function Problem: *Given strings s and t, find one or more edit sequences that minimize the distance between the strings.*

Basic string editing operations are insertions, deletions, and substitutions.

Knuth, in (Knuth, 1993), shows how to transform **words** into **graph** as a ladder of seven substitutions.

words, wolds, golds, goads, grads, grade, grape, graph.

Definition 14: Insertions, Deletions, and Substitutions

Let s, t, u and v be strings over alphabet Σ, and let a and b be a character in Σ.

- *Insertion:* *If $s = uv$, then $t = uav$.*
- *Deletion:* *If $s = uav$, then $t = uv$.*
- *Substitution:* *If $s = uav$, then $t = ubv$.*

Positive weights are assigned to each edit operations.

$$\begin{aligned} w_{ins}(a) &= \text{cost of inserting } a. \\ w_{del}(a) &= \text{cost of deleting } a. \\ w_{sub}(a, b) &= \text{cost of substituting } b \text{ for } a. \end{aligned}$$

Let $d(i, j)$ be the minimum edit distance between $s[0, \ldots, i-1]$ and $t[0, \ldots, j-]$. That is, $d(i, j)$ is the minimum edit distance the first i characters of s and first j characters of t.

If $|s| = n$ and $|t| = m$, the minimum edit distance is $d(n, m)$. The edit distance can be defined by the recurrence

$$d(i, j) = \begin{cases} d(i-1, j-1) & \text{if } s_i = t_j \\ \min \begin{cases} d(i-1, j) + w_{del}(s_i) \\ d(i, j-1) + w_{ins}(t_j) \\ d(i-1, j-1) + w_{sub}(s_i, t_j) \end{cases} & \text{otherwise} \end{cases}$$

The recurrence reads as follows:

- If the next characters (s_i and t_j) match, there is no increase in the edit distance.
- Otherwise, take the smallest of three choices:
  - The cost of matching the first $i-1$ and j characters, then deleting s_i.
  - The cost of matching the first i and $j-1$ characters, then inserting t_i.

– The cost of matching the first $i-1$ and $j-1$ characters, then substituting s_i for t_j

Assume that each insertion or deletion has a cost of 1, but a substitution costs 2 (A substitution can be thought of as a deletion followed by an insertion.)

One alignment of

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

might look like this:

T	A	G	-	C	T	A	T	C	A
-	A	G	G	C	T	A	T	T	A
d			i					s	

The table below shows the initial configuration when computing the minimal edit distance between

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

The rows and columns are labeled by the characters in the strings.

The λ column shows the costs for inserting of TAGCTATCA into an empty string. These costs are $d(i, 0) = i$ for $i = 0, \ldots, 9$.

The λ row shows the costs for inserting AGGCTATTA into an empty string. These costs are $d(0, j) = j$ for $j = 0, \ldots, 9$.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1									
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Values can be computed along diagonals. The first computed value

comes from comparing T and A.

$$d(1,1) = \begin{cases} d(0,0)(=0) & \text{if } T = A \\ \min \begin{cases} d(0,1)+1(=2) \\ d(1,0)+1(=2) \\ d(0,0)+2(=2) \end{cases} & \text{otherwise} \end{cases}$$

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	2								
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Possible edits to compute $d(1,1)$:

Substitute
T
A

Delete–Insert
T A
- A

Insert–Delete
- T
A -

$$d(1,2) = \begin{cases} d(0,1)(=1) & \text{if } T = G \\ \min \begin{cases} d(0,2)+1(=3) \\ d(1,1)+1(=3) \\ d(0,1)+2(=3) \end{cases} & \text{otherwise} \end{cases}$$

Now, compute values in the next diagonal:

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	2	3							
A	2	1								
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

$$d(2,1) = \begin{cases} d(1,0)(=1) & \text{if } A = A \\ \min \begin{cases} d(1,1)+1(=3) \\ d(2,0)+1(=3) \\ d(1,0)+2(=3) \end{cases} & \text{otherwise} \end{cases}$$

Next, compute $d(3,1)$ $d(2,2)$, and $d(1,3)$

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	2	3	4						
A	2	1	2							
G	3	2								
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

$$d(3,1) = \begin{cases} d(2,0)(=2) & \text{if } G = A \\ \min \begin{cases} d(2,1)+1(=3) \\ d(3,0)+1(=3) \\ d(2,0)+2(=3) \end{cases} & \text{otherwise} \end{cases}$$

$$d(2,2) = \begin{cases} d(1,1)(=2) & \text{if } A = G \\ \min \begin{cases} d(2,1)+1(=3) \\ d(3,0)+1(=3) \\ d(1,1)+2(=3) \end{cases} & \text{otherwise} \end{cases}$$

$$d(1,3) = \begin{cases} d(0,2)(=2) & \text{if } G = A \\ \min \begin{cases} d(0,3)+1(=3) \\ d(1,2)+1(=3) \\ d(0,2)+2(=3) \end{cases} & \text{otherwise} \end{cases}$$

The complete edit distance table is:

Fill in the next diagonal: $d(4,1)$, $d(3,2)$, $d(2,3)$, and $d(1,4)$.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	2	3	4	5	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	6	7	8
C	4	3	2	3	2	3	4	5	6	7
T	5	4	3	4	3	2	3	4	5	6
A	6	5	4	5	4	3	2	3	4	5
T	7	6	5	6	5	4	3	2	3	4
C	8	7	6	7	6	5	4	3	4	5
A	9	8	7	8	7	6	5	4	5	4

The optimal alignment has cost 4 for 1 deletion, 1 insertion, and 1 substitution.

T	A	G	-	C	T	A	T	C	A
-	A	G	G	C	T	A	T	T	A
d			i					s	

This can be seen from tracing back in the array.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	2	3	4	5	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	6	7	8
C	4	3	2	3	2	3	4	5	6	7
T	5	4	3	4	3	2	3	4	5	6
A	6	5	4	5	4	3	2	3	4	5
T	7	6	5	6	5	4	3	2	3	4
C	8	7	6	7	6	5	4	3	4	5
A	9	8	7	8	7	6	5	4	5	4

With these rules, edit distance defines a metric space on strings.

Definition 15: Metric Space

Let $s = a_0 a_1 \cdots a_{n-1}$ *and* $t = b_0 b_1 \cdots b_{m-1}$ *be strings over* Σ. *Define* $d(s, s')$ *to be the minimum of over all sequences of edits that transform* s *into* s'. *Then,*

1. $d(s, s') = 0$ *if an only if* $s = s'$: *It costs nothing to change a string into itself.*
2. $d(s, s') > 0$ *when* $s \neq s'$: *It costs something to change a string into another string.*
3. $d(s, s') = d(s', s) > 0$ *when* $s \neq s'$: *Edit distance is symmetric.*
4. *The triangle inequality* $d(s, s') \leq d(s, s'') + d(s'', s')$ *holds: It costs no more change* s *into* s' *than to go through any intermediary string* s''. *The length between triangle vertices is no more than the sum of the other legs.*

Here is a C implementation of edit distance.

Listing 42: Iterative String Edit Distance

148 ⟨*Iterative String Edit Distance* 148⟩≡

```
int editDist(char *s, int ls, char *t, int lt)
{
  int distances[ls][lt];
```

```
    ⟨If either string is empty return 149a⟩
    ⟨Initialize the first row and first column 149b⟩
    ⟨For every pair of characters 149c⟩
    {
      ⟨If characters match, use the previous distance 149d⟩
      ⟨Otherwise, use the minimum distance 150⟩
    }
    return distance[ls-1][lt-1];
  }
```

If either string `s` or `t` is empty, return the length of the other, which translated to inserting its characters. The C-idiom is "if `ls=0`, then `!ls` is `True`."

149a ⟨*If either string is empty return* 149a⟩≡
```
if (!ls) return lt;
if (!lt) return ls;
```

Initializing the first row and column has time complexity $\Theta(n + m)$.

149b ⟨*Initialize the first row and first column* 149b⟩≡
```
for (int i = 0, int j = 0; i < m, j < n; i++, j++)
{
  distances[i][0] = i;
  distances[0][j] = j;
}
```

There are nm pairs of characters, assuming the source string `s` has length `ls = n` and target string `t` has length `lt = m`.

149c ⟨*For every pair of characters* 149c⟩≡
```
for (int i = 1; i < ls; i++)
  for (int j = 1; j < lt; j++)
```

Testing for a match has complexity $O(1)$.

149d ⟨*If characters match, use the previous distance* 149d⟩≡
```
if (s[i-1] == t[j-1]) {
  distance[i][j] = distance[i-1][j-1];
}
```

And when a mismatch occurs, only a few table look-ups, comparisons, and assignments are necessary.

150 ⟨*Otherwise, use the minimum distance* 150⟩≡

```
else
{
  min = distance[i-1][j-1];
  if (min > distance[i][j-1])
  {
    min = distance[i][j-1];
  }
  if (min > distance[i-1][j])
  {
    min = distance[i-1][j];
  }
  distance[i][j] = 1 + min;
}
```

The performance of the edit distance algorithm is characterized by

- Time complexity: $O(nm)$ to account for the nested `for` loops.
- Space complexity: $O(nm)$ to account for storing the table.
- Trace-back: $O(n+m)$ to construct the optimal alignment.

Matrix Chain Multiplication

Problem 9: Matrix Chain Multiplication

Function Problem: Find the way to parenthesis a matrix product

$$M_1 M_2 \cdots M_n$$

to minimize the number of operations.

Inner Products

Let $\vec{X}$ and $\vec{Y}$ be vectors of length m. The *inner product* $\langle \vec{X} \cdot \vec{Y} \rangle$ of $\vec{X}$ and $\vec{Y}$ is the value

$$\langle \vec{X} \cdot \vec{Y} \rangle = \sum_{i=0}^{m-1} X[i] * Y[i]$$

For instance, if

$$\vec{X} = \langle -1, 2, -1 \rangle \quad \text{and} \quad \vec{Y} = \langle 2, 2, 2 \rangle$$

then

$$\langle \vec{X} \cdot \vec{Y} \rangle = (-1 \cdot 2) + (2 \cdot 2) + (-1 \cdot 2) = 0$$

In this case $\langle -1, 2, -1 \rangle$ and $\langle 2, 2, 2 \rangle$ are *orthogonal*.

Inner products define cosines.

$$\cos\theta = \frac{\langle \vec{X} \cdot \vec{Y} \rangle}{\left|\vec{X}\right| \left|\vec{Y}\right|}$$

where $\left|\vec{X}\right| = \sqrt{\langle \vec{X} \cdot \vec{X} \rangle}$.

Listing 43: Functional Inner Product

151a ⟨*Functional Inner Product* 151a⟩≡

```
innerProduct :: Num a => [a] -> [a] -> a
innerProduct [] ys = error "first vector too short"
innerProduct xs [] = error "second vector too short"
innerProduct (x:xs) (y:ys) = x*y + innerProduct xs ys

- - Using Haskell idioms:
innerProduct' :: Num a => [a] -> [a] -> a
innerProduct' x y = foldr (+) 0 (zipWith (*) x y)
```

The time complexity of `innerProduct` is $O(m)$.

Listing 44: Imperative Inner Product

151b ⟨*Imperative Inner product* 151b⟩≡

```
double innerProduct(double X[m], double Y[m])
{
  double ip = 0;
  for (int i = 0; i < m; i++) {ip = ip + X[i] * Y[i];}
  return ip;
}
```

Matrix Multiplication

Let A and B be $n \times m$ and $m \times \ell$ matrices. Their product AB is an $n \times \ell$ matrix. The standard algorithm computes the *inner product* of each row of A with each column of B. Therefore, the time complexity of the standard algorithm is $O(nm\ell)$.

Listing 45: Functional Inner Product

151c ⟨*Functional Matrix Multiplication* 151c⟩≡

```
matrixMult :: Num a => [[a]] -> [[a]] -> [[a]]
matrixMult xs ys = [[(innerProduct x y) | y <- ys] | x <-xs]
```

Matrix multiplication is associative. When given a chain of matrices to multiply, say

$$M_1 \times M_2 \times M_3 \times M_4$$

there are several orders in which the computation can proceed.

$$\begin{aligned} M_1 \times M_2 \times M_3 \times M_4 &= (M_1 \times (M_2 \times (M_3 \times M_4))) \\ &= (M_1 \times ((M_2 \times M_3) \times M_4)) \\ &= ((M_1 \times M_2) \times (M_3 \times M_4)) \\ &= ((M_1 \times (M_2 \times M_3)) \times M_4) \\ &= (((M_1 \times M_2) \times M_3) \times M_4) \end{aligned}$$

In this case, where there are 5 different ways to form the product. The numbers of ways to parenthesize a chain of expressions is a Catalan number

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

At $n = 3$, the Catalan number is

$$C(3) = \frac{1}{4}\binom{6}{3} = \frac{6!}{4 \cdot 3!3!} = 5$$

An asymptotic formula for Catalan numbers shows their exponential growth rate.

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Therefore, it is not feasible to search over all possible ways to compute a chain of matrix products.

To see that multiplication order really does matter, consider this example:

$$M_1 \text{ is } 10 \times 100, \quad M_2 \text{ is } 100 \times 5, \quad M_3 \text{ is } 5 \times 50$$

There are $C(2) = \frac{1}{3}\binom{4}{2} = 2$ ways to compute the product

$((M_1 \times M_2) \times M_3)$	Cost: $(10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) =$	7500
$(M_1 \times (M_2 \times M_3))$	Cost: $(100 \cdot 5 \cdot 50) + (10 \cdot 100 \cdot 50) =$	75000

Structure of the optimal solution

Pretend you want to optimally compute

$$M_1 M_2 \cdots M_n$$

where M_k is a $p_{k-1} \times p_k$ matrix. That is, sequence

$$\vec{p} = \langle p_0, p_1, \ldots, p_n \rangle$$

defines valid matrix (row, column) sizes for multiplication.

Let

$$d(i, j) = \text{optimal cost to compute the product } M_i \cdots M_j$$

The ultimate value to compute is $d(1, n)$.

If the optimal parenthesizing is

$$((M_1 \cdots M_k)(M_{k+1} \cdots M_n))$$

then optimal cost is the optimal cost to compute $(M_1 \cdots M_k)$ plus the optimal cost to compute $(M_{k+1} \cdots M_n)$ plus $p_0 p_k p_n$, the cost to compute the product $((M_1 \cdots M_k) \times (M_{k+1} \cdots M_n))$.

That is,

$$d(1, n) = d(1, k) + d(k+1, n) + p_0 p_k p_n$$

But, you don't know *a priori* which k to use, so compute them all and take a minimum

$$d(1, n) = \begin{cases} 0 & \text{if } 1 = n \\ \min\limits_{1 \le k < n} \{d(1, k) + d(k+1, n) + p_0 p_k p_n\} & \text{if } 1 < n \end{cases}$$

For intermediate products, the optimal cost is defined by the recursion

$$d(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < n} \{d(i, k) + d(k+1, j) + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Example: Matrix Chain Example

Consider the product

$$M_1 M_2 M_3$$

where the sizes of the matrices are

$$\vec{p} = \langle 10, 100, 5, 50 \rangle$$

Initialize:

$$d[1, 1) = 0$$
$$d(2, 2) = 0$$
$$d(3, 3) = 0$$

$$\begin{bmatrix} 0 & - & - \\ & 0 & - \\ & & 0 \end{bmatrix}$$

Then:

$$\begin{aligned} d(1,2) &= \min\{d(1,1) + d(2,2) + p[0]p[1]p[2]\} \\ &= 0 + 0 + 10 \cdot 100 \cdot 5 \\ &= 5000 \\ d(2,3) &= \min\{d(2,2) + d(3,3) + p[1]p[2]p[3]\} \\ &= 0 + 0 + 100 \cdot 5 \cdot 50 \\ &= 25000 \end{aligned}$$

$$\begin{bmatrix} 0 & 5000 & - \\ & 0 & 25000 \\ & & 0 \end{bmatrix}$$

Next:

$$\begin{aligned} d(1,3) &= \min\{d(1,1) + d(2,3) + p[0]p[1]p[3], \\ &\qquad d(1,2) + d(3,3) + p[0]p[2]p[3]\} \\ &= \min\{25000 + 10 \cdot 100 \cdot 50, 5000 + 10 \cdot 5 \cdot 50\} \\ &= \min\{75000, 7500\} \end{aligned}$$

$$\begin{bmatrix} 0 & 5000 & 7500 \\ & 0 & 25000 \\ & & 0 \end{bmatrix}$$

We know that is the matrix sizes are give by the sequence

$$\vec{p} = \langle p_0, \ldots, p_n \rangle$$

then the final product has size $p_0 \times p_n$.

Listing 46: Matrix Chain Multiplication Order

```
154 ⟨Matrix Chain Multiplication Order 154⟩≡
      ⟨Initialize the main diagonal 155a⟩

      ⟨For each sub-diagonal 155b⟩
      {
        ⟨For each sub-diagonal row 155c⟩
        {
          ⟨Set the column and impossible value 155d⟩
          ⟨For each way to partition a product 155e⟩
          {
            ⟨Compute the cost of this partition 156a⟩
```

```
        ⟨If smaller cost: update minimum, save partition 156b⟩
      }
    }
    return d and s;
  }
```

First, initialize the main diagonal at a cost that is $O(n)$.

Initialize: $O(n)$.

155a ⟨*Initialize the main diagonal* 155a⟩≡

```
for(int i = 1; i <= n; i++) {
  d[i, i] = 0
}
```

Now loop over each sub-diagonal. Think of them as starting at column 2 and going through n, which is a single value in the northeast corner of a matrix.. This loop has time complexity $O(n-1)$.

155b ⟨*For each sub-diagonal* 155b⟩≡

```
for (int s = 2; s <= n; s++)
```

When you start at subdiagonal s, the rows go from $i = 1$ to $i = (n - s + 1)$. Visualize it: The upper sub-diagonal starting at column 2 goes from row 1 to $n - 1 = n - 2 + 1$.

155c ⟨*For each sub-diagonal row* 155c⟩≡

```
for(int i = 1; i <= n - s + 1; i++)
```

Now we want to compute the value in subdiagonal s and row i. This value is at column index $j = i + s - 1$. Think about it: On subdiagonal 2 at row 5 the column index will be $j = 5 + 2 - 1 = 6$.

Set the value here to `INFINITY` so any computed value will be smaller. This has constant cost, but happens

$$\begin{aligned}\sum_{s=2}^{n}\sum_{i=1}^{n-s+1} 1 &= \sum_{s=2}^{n}(n-s+1)\\ &= \sum_{p=1}^{n-1} p\\ &= \frac{n(n-1)}{2} \quad \text{times}\end{aligned}$$

155d ⟨*Set the column and impossible value* 155d⟩≡

```
j = i + s âĹŠ 1;
d[i, j] = INFINITY;
```

Now partition the matrix product $M_i \cdot M_j$ at k for each $k = i$ to $j - 1$.

155e ⟨*For each way to partition a product* 155e⟩≡

```
for(int k = i; k < j; k++)
```

Everything from now on occurs within three nested `for` loops. Each operation within the inner loop on `k` has constant cost. Therefore, the overall complexity can be computed from the sum:

$$\begin{aligned}\sum_{s=2}^{n}\sum_{i=1}^{n-s+1}\sum k = i^{i+s-2}1 &= \sum_{s=2}^{n}\sum_{i=1}^{n-s+1}(s-1)\\ &= \sum_{p=1}^{n-1}(j+(j-1)+\cdots\\ &= \frac{n(n-1)}{2} \quad \text{times}\end{aligned}$$

156a ⟨*Compute the cost of this partition* 156a⟩≡

```
q = d[i, k] + d[k + 1, j] + p[i-1]p[k]p[j];
```

156b ⟨*If smaller cost: update minimum, save partition* 156b⟩≡

```
if q < d[i, j]
{
  d[i, j] = q;
  s[i, j] = k;
}
```

Exercises

1. Complete the edit distance table for the following pair of strings.

	λ	G	R	A	P	H
λ						
W						
O						
R						
D						
S						

An interesting question is: In how many ways can you parenthesize an associative operation involving n expressions. These are called Catalan numbers. They occur in many applications.

2. Consider multiplying matrices, say $M_0 \times M_1 \times M_2 \times M_3$ Matrix multiplication is associative:

$$\begin{aligned} M_0 \times M_1 \times M_2 \times M_3 &= M_0 \times (M_1 \times (M_2 \times M_3)) \\ &= M_0 \times ((M_1 \times M_2) \times M_3) \\ &= (M_0 \times M_1) \times (M_2 \times M_3) \\ &= (M_0 \times (M_1 \times M_2)) \times M_3 \\ &= ((M_0 \times M_1) \times M_2) \times M_3 \end{aligned}$$

   Suppose M_0 is 10×100, M_1 is 100×5, M_2 is 5×50, and M_3 is 50×10. What are the various costs? What is the best way to parenthesize a sequence of matrix multiplies?

3. Write a dynamic programming algorithm that computes $C(1, n)$ from the following formula. Before setting up the iteration loops carefully observe that all the needed values should be available. Analyze the space and time complexities of your algorithm. Draw a blank table for C indicating the order of your computation (loops).

$$\begin{aligned} C(i,j) &= 0 && (\forall i \geq j) \\ C(i,j) &= \max\{C(i,k) + C(\ell,j) + 2\} && (\forall i < k \leq n) \wedge (\forall 1 \leq \ell < j) \wedge (\forall 1 \leq i < j \leq n) \end{aligned}$$

4. What is the best way to parenthesize the product $M_1M_2M_3M_4$ wnen their sizes are described by the sequence $\vec{p} = \langle 5, 10, 10, 100, 5\rangle$?

15. Edit Distance

String Alignment

Imagine applications where the distance between strings could be useful.

Knuth, in (Knuth, 1993), shows how to transform **words** into **graph** using a seven step *ladder* of one letter substitutions.

$$\text{words} \mapsto \text{wolds} \mapsto \text{golds} \mapsto \text{goads} \mapsto \text{grads} \mapsto \text{grade} \mapsto \text{grape} \mapsto \text{graph}.$$

Spell checkers provide a list of *nearby* words when a string is not found in the dictionary. DNA, the molecule of life, can be abstracted as strings over the alphabet

$$\mathbb{DNA} = \{A, C, G, T\}$$

Geneticists study the similarities and differences in DNA among members of a species and between species.

The similarity of two strings can be measured by an *edit distance*: The cost of a sequence of edit operations that change one string into another. Many different edit operations and cost measures have been proposed. Most measures define a *metric space*.

Definition 16: Metric on Strings

Let Σ^* be the set of all strings over an alphabet Σ. A metric d *is a function*

$$d : \Sigma^* \times \Sigma^* \mapsto [0, \infty)$$

with the properties:

1. ***Non-negativity:*** $d(s, t) \geq 0$ *for all strings* $s, t \in \Sigma^*$
2. ***Zero-distance:*** $d(s, t) = 0$ *if and only if* $s = t$
3. ***Symmetry:*** $d(s, t) = d(t, s)$
4. ***Triangle Inequality:*** $d(s, u) \leq d(s, t) + d(t, u)$

The common edit operations are *substitute*, *insert*, and *delete*.

Definition 17: String Operations

Let α and β be strings over Σ. Let x and y be a character in alphabet Σ. Define three operations:

- *Insertion:* $\alpha\beta \mapsto \alpha x\beta$
- *Deletion:* $\alpha x\beta \mapsto \alpha\beta$
- *Substitution:* $\alpha x\beta \mapsto \alpha y\beta$

For instance, to align $\alpha =$ ALGORITHM with $\beta =$ ALGEBRAIC this sequence of editing operations might be used, where s, i and d stand for *substitution*, *insertion*, and *deletion*.

A	L	G	O	-	R	-	I	T	H	M
A	L	G	E	B	R	A	I	C	-	-
			s	i		i		s	d	d

Is 6 the smallest unit cost distance between ALGORITHM and ALGEBRAIC?
Another metric charges 2 for a substitution.

If each operation has a cost of 1, then the distance between these two string is 6.

Levenshtein distance is a commonly used metric on strings. It is defined iteratively based on string length. Let λ denote the empty string. Then, the distance from any string α to λ is the length of α.

$$d(\alpha, \lambda) = d(\lambda, \alpha) = |\alpha|$$

Otherwise, let $d_{\alpha,\beta}(i, j)$ be the distance between $\alpha[1..i]$, the first i characters of α, and $\beta[1..j]$, the first j characters of β.

- If $\alpha[i]$ is deleted in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..(i-1)]$ and $\beta[1..j]$ plus 1.
- If $\beta[j]$ is deleted in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..i]$ and $\beta[1..(j-1)]$ plus 1.
- If $\alpha[i]$ is substituted $\beta[j]$ in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..(i-1)]$ and $\beta[1..(j-1)]$ plus 1 if $\alpha[i] \neq \beta[j]$. And, otherwise, it is just $d((i-1), (j-1))$.

The value of $d(i, j)$ is the minimum over all of these values.

Definition 18: Levenshtein distance

Let α and β be strings over Σ. For $0 \leq i \leq |\alpha|$ and $0 \leq j \leq |\beta|$,

define the edit distance between $\alpha[1..i]$ *and* $\beta[1..j]$ *to be*

$$d(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} d(i-1,j)+1 \\ d(i,j-1)+1 \\ d(i-1,j-1)+[\alpha_i \neq \beta_j] \end{cases} & \text{otherwise} \end{cases}$$

where $[\texttt{False}] = 0$ *and* $[\texttt{True}] = 1$ *is the* characteristic *function.*

Problem 10: String Edit Distance Problem

Decision Problem: Given strings s *and* t*, is* m *the minimum number of edits to transform* s *into* t*?*
Function Problem: Given strings s *and* t*, find one or more edit sequences that minimize the distance between these strings.*

One alignment of

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

might look like this:

```
T A G - C T A T C A
- A G G C T A T T A
-------------------
d       i       s
```

The table below shows the initial configuration when computing the minimal edit distance between

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

The rows and columns are labeled by the characters in the strings.

The λ column shows the costs for inserting of TAGCTATCA into an empty string. These costs are $d(i,0) = i$ for $i = 0,\ldots,9$.

The λ row shows the costs for inserting AGGCTATTA into an empty string. These costs are $d(0,j) = j$ for $j = 0,\ldots,9$.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1									
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Values can be computed along diagonals. The first computed value comes from comparing T and A.

$$d(1,1) = \begin{cases} d(0,0)\ (=0) & \text{if } T = A \\ \min \begin{cases} d(0,1)+1)\ (=2) \\ d(1,0)+1)\ (=2) \\ d(0,0)+1)\ (=1) \end{cases} & \text{otherwise} \end{cases}$$

Downward moves ↓ and horizontal moves → always cost 1. Diagonal moves ↘ cost 0 or 1 depending on whether the next characters match or not. The minimum of these costs is the new value.

Possible edits to compute $d(1,1)$:

Substitute
T
A

Delete–Insert
T A
- A

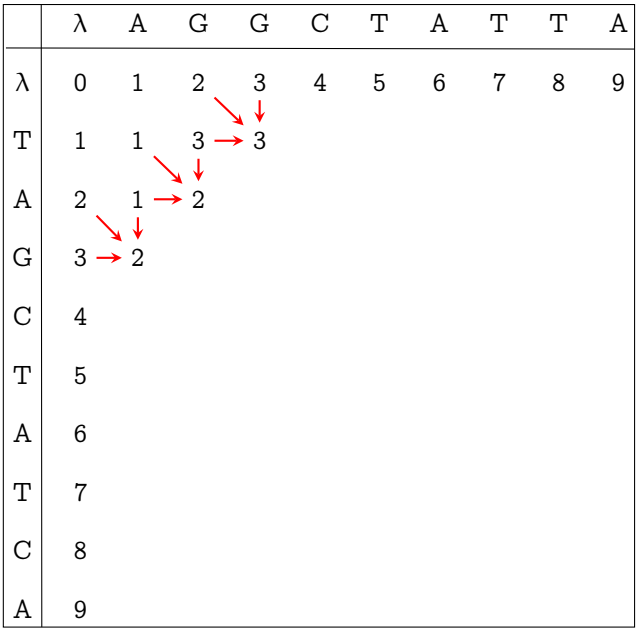
Insert–Delete
- T
A -

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1								
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Now, compute values in the next diagonal:

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2							
A	2	1								
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Next, compute d(3, 1) d(2, 2), and d(1, 3)



	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	3	3						
A	2	1	2							
G	3	2								
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

The complete edit distance table is:

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2	3	4	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	5	6	7
C	4	3	2	2	2	3	4	5	6	7
T	5	4	3	3	3	2	3	4	5	6
A	6	5	4	4	4	3	2	3	4	5
T	7	6	5	5	5	4	3	2	3	4
C	8	7	6	6	5	5	4	3	3	4
A	9	8	7	7	6	6	5	4	4	3

Tracing back in the array reveals optimal alignments.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2	3	4	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	5	6	7
C	4	3	2	2	2	3	4	5	6	7
T	5	4	3	3	3	2	3	4	5	6
A	6	5	4	4	4	3	2	3	4	5
T	7	6	5	5	5	4	3	2	3	4
C	8	7	6	6	5	5	4	3	3	4
A	9	8	7	7	6	6	5	4	4	3

The optimal alignment has cost 3

T	A	G	-	C	T	A	T	C	A
-	A	G	G	C	T	A	T	T	A
d			i					s	

Here is a C implementation of edit distance.

Listing 47: Iterative String Edit Distance

165a ⟨*Iterative String Edit Distance* 165a⟩≡

```
int editDist(char *s, int ls, char *t, int lt)
{
  int distances[ls][lt];

  ⟨If either string is empty return 165b⟩
  ⟨Initialize the first row and first column 166a⟩
  ⟨For every pair of characters 166b⟩
  {
    ⟨If characters match, use the previous distance 166c⟩
    ⟨Otherwise, use the minimum distance 166d⟩
  }
  return distance[ls-1][lt-1];
}
```

If either string `s` or `t` is empty, return the length of the other, which translated to inserting its characters. The C-idiom is "if `ls=0`, then `!ls` is `True`."

165b ⟨*If either string is empty return* 165b⟩≡

```
if (!ls) return lt;
if (!lt) return ls;
```

Initializing the first row and column has time complexity $\Theta(n + m)$.

166a ⟨*Initialize the first row and first column* 166a⟩≡
```
for (int i = 0, int j = 0; i < m, j < n; i++, j++)
{
  distances[i][0] = i;
  distances[0][j] = j;
}
```

There are nm pairs of characters, assuming the source string `s` has length `ls = n` and target string `t` has length `lt = m`.

166b ⟨*For every pair of characters* 166b⟩≡
```
for (int i = 1; i < ls; i++)
  for (int j = 1; j < lt; j++)
```

Testing for a match has complexity $O(1)$.

166c ⟨*If characters match, use the previous distance* 166c⟩≡
```
if (s[i-1] == t[j-1]) {
  distance[i][j] = distance[i-1][j-1];
}
```

And when a mismatch occurs, only a few table look-ups, comparisons, and assignments are necessary.

166d ⟨*Otherwise, use the minimum distance* 166d⟩≡
```
else
{
  min = distance[i-1][j-1];
  if (min > distance[i][j-1])
  {
    min = distance[i][j-1];
  }
  if (min > distance[i-1][j])
  {
    min = distance[i-1][j];
  }
  distance[i][j] = 1 + min;
}
```

The performance of the edit distance algorithm is characterized by

- Time complexity: $O(nm)$ to account for the nested `for` loops.
- Space complexity: $O(nm)$ to account for storing the table.
- Trace-back: $O(n + m)$ to construct the optimal alignment.

Exercises

1. Fill in the the optimal alignment table for *seat* and *belt*.

2. Fill in the the optimal alignment table for *park* and *spake*.

16. Greedy Algorithms

Greedy Algorithm Concept

Please read Chapter 16 Greedy Algorithms in the textbook (Corman et al., 2009).

Greedy algorithms always make a choice that seems best at the moment. This locally optimal choice is made with the hope it will lead to a globally optimal solution. Greedy algorithms don't always work, but sometimes they do. Deciding how to make the right local selection is key. Greedy algorithms are often applied to *combinatorial optimization problems*.

Problem 11: Combinatorial Optimization

Given an instance I *of a function problem* $\mathcal{P}$*, assume there is a set of* candidate *or* feasible solutions *that satisfy the* constraints *of the problem. For each feasible solution there is a value determined by an* objective function*. Find one (or more)* optimal solution(s) *that minimize (or maximize) the value the the objective function.*

Here is the outline for these notes.

169 ⟨*Greedy algorithms* 169⟩≡
- ⟨*Tape Storage of Files* 171⟩
- ⟨*Rational Knapsack* 173a⟩
- ⟨*Activity Selection* 176⟩
- ⟨*Kruskal's Minimum Spanning Tree* 178⟩
- ⟨*Prim's Minimum Spanning Tree* 188a⟩
- ⟨*Dijkstra's Algorithm* 193a⟩

Tape Storage of Files

Okay, it is old-school, but the problem of how to store of files on tape helps illustrate *greedy* algorithms. Pretend there are n files stored on a tape. Let $L[i]$ be the length of file i for $i = 1, 2, \ldots, n$. Assume the cost of accessing a file depends on its length plus the lengths of prior files on the tape, called *sequential access*. That is, the time cost to access file k is

$$T(k) = \sum_{i=1}^{k} L[i]$$

If each file is equally likely to be accessed, then the average (expected) time cost is to access a file is

$$T_{avg}(n) = \sum_{k=1}^{n} \frac{T(k)}{n} = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[i]}{n}$$

Different file storage orders result in different expected costs. For instance, suppose the files have lengths

$$L[1] = 100,\ L[2] = 150,\ L[3] = 50,\ L[4] = 200$$

If the files are stored in order $\langle 3, 1, 2, 4 \rangle$ then the average assess time is

$$\begin{aligned} T_{avg}(n) &= \frac{1}{4}(50 + (50 + 100) + (50 + 100 + 150) + (50 + 100 + 150 + 200)) \\ &= \frac{1}{4}(4 \cdot 50 + 3 \cdot 100 + 2 \cdot 150 + 200) = 250 \end{aligned}$$

You can compute that other order increase the average access time. For example, the order $\langle 1, 2, 3, 4 \rangle$ has average cost

$$\begin{aligned} T_{avg}(n) &= \frac{1}{4}(100 + (100 + 150) + (100 + 150 + 50) + (100 + 150 + 50 + 200)) \\ &= \frac{1}{4}(4 \cdot 100 + 3 \cdot 150 + 2 \cdot 50 + 200) = 287.5 \end{aligned}$$

Problem 12

ape Storagetape-store Tape Storage Problem: Find permutation $\pi(i)$ of $i = 1, \ldots, n$ that minimizes the average cost.

$$T_{min\text{-}avg}(n) = \min_{\pi} \left\{ \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[\pi(i)]}{n} \right\}$$

There are $n!$ permutations of n distinct values. Therefore, it is infeasible to find the minimum by computing the cost of every permutation, except perhaps for small values of n.

The greedy approach sorts the files by their lengths and stores the shortest length files first. That is,

$$L[\pi(i)] \leq L[\pi(i+1)] \quad \text{for all } i$$

And, this greedy approach does produce the smallest average file access cost.

Theorem 12: File Access Minimization

The average cost of file access is minimized when the files are stored from smallest to largest.

Proof: File Access Minimization

Suppose that π is the optimal file storage permutation, but two consecutive files are out of length order in this minimial order. Call them file $k = \pi(i)$ and file $j = \pi(i+1)$. The assumption is that $L[k] > L[j]$.
If files k and j are swapped, then the cost to access file k is increased by $L[j]$. And, the cost to access file j is decreased by $L[k]$. The average cost is changed by a negatrive amount: $(L[j] - L[k])/n < 0$.
But this contradict that the given order π was the one giving minimal average cost for file access.

171 ⟨*Tape Storage of Files* 171⟩≡

```
No code here: Sort the files by their lengths and write them to tape.
```

The time complexity to store the files in optimal access time order is $O(n \lg(n))$ plus $O(\sum L[i])$ where n is the number of files and the sum is over the lengths of all files.

The Rational Knapsack Problem

The *rational knapsack* problem leads to a quintessential greedy algorithm.

Problem 13: Rational (Fractional) Knapsack Problem

An instance I consist of a knapsack with capacity C and a list of n (divisible) items with associated weights

$$w_0, w_2, \ldots, w_{n-1}$$

and values

$$v_0, v_1, \ldots, v_{n-1}$$

A feasible solution is a set of fractions

$$0 \leq r_k \leq 1,\ k = 0, \ldots, (n-1)$$

representing how much of the k^{th} *item is placed in the knapsack, subject to the constraint*

$$\sum r_j w_j \leq C \quad \textit{the weight does not exceed the capacity}$$

The objective function is to maximize the sum over the fractional values

$$\max \sum r_k v_k \textit{the value is as large as possible}$$

Here are three greedy approaches.

1. Sort the items by increasing weight, placing lighter weight items in first.
2. Sort the items by decreasing value, placing more valuable items in first.
3. Sort the items by increasing value to weight ratios, placing more value/weight items in first.

Example: Rational Knapsack Example

Pretend

$$w_0 = 10,\ v_0 = 30;\ w_1 = 5,\ v_1 = 20;\ w_2 = 1,\ v_2 = 2;\ C = 10$$

1. *Greedy weight approach: Place all of item 1, all of 1 and* $4/10$ *of item 0.*

   Constraint: $1 + 5 + \frac{4}{10}10 \leq 10$ *Objective:* $2 + 20 + \frac{4}{10}30 = 34$

2. *Greedy value approach: Place all of item 0.*

   Constraint: $10 \leq 10$ *Objective:* 30

3. *Greedy value:weight approach: The sorted value-to-weight ratios are*

   $$\frac{v_1}{w_1} = 4;\ \frac{v_0}{w_0} = 3;\ \frac{v_2}{w_2} = 2$$

   Place all of item 1 and $5/10$ *of item 0.*

   Constraint: $5 + \frac{5}{10}10 \leq 10$ *Objective:* $20 + \frac{5}{10}30 = 35$

The optimal value is $20 + 15 = 35$*, given by the greedy value-to-weight ratio approach.*

The algorithm below uses this greedy heuristic to solve the rational knapsack problem. Its running time is $O(n)$ if the ratios have be previously computed and sorted. If the ratios need to be sorted its time complexity is $O(n \lg n)$. Building a heap and using a priority queue may, in some cases, be less expensive than sorting all the ratios. The precondition is

$$\frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq \cdots \geq \frac{v_{n-1}}{w_{n-1}}$$

Supply an explanation that the time complexity of this rational knapsack algorithm has time complexity $O(n)$. Assume value-to-weight ratios have been previously sorted.

Listing 48: Rational Knapsack

173a ⟨*Rational Knapsack* 173a⟩≡

```
int knapsack(int *v, int *w, int n, int C)
{
  ⟨Initialize rational knapsack local state 173b⟩
  ⟨While accumulated weight ≤ C and more items 173c⟩
  {
    ⟨If all of the next item can be added 173d⟩
    {
      ⟨Update fraction, weight, value and next item 174a⟩
    }
    ⟨Otherwise add a fraction of the next item 174b⟩
  }
}
```

The local state includes an index `k` into the value, weight, and fraction arrays. The index `k` identifies the next item to be considered. Accumulators for the value and weight are needed, and the fractions can be initialized to zero.

173b ⟨*Initialize rational knapsack local state* 173b⟩≡

```
int k = 0, V = 0, W = 0;
int r[n];
for (int j = 0; j < n; j++) { r[j] = 0; }
```

The `while` condition is this:

173c ⟨*While accumulated weight* ≤ C *and more items* 173c⟩≡

```
while ((W < C) && (k < n))
```

Inside of the `while`, test if all of the next item (`k`)can be placed in the knapsack. This occurs if the current weight `W` and the next item's weight `w[k]` do not exceed the capacity `C`.

173d ⟨*If all of the next item can be added* 173d⟩≡

```
if (W + w[k] <= C)
```

When all of the next item fits, update every state value.

174a ⟨*Update fraction, weight, value and next item* 174a⟩≡

```
r[k] = 1;
W = W + w[k];
V = V + v[k];
k = k + 1;
```

If not all of the next item fits, the fraction

$$r_k = \frac{C - W}{w_k}$$

determines how much of item k can be placed in the knapsack. Notice that

$$W + r_k w_k = C$$

174b ⟨*Otherwise add a fraction of the next item* 174b⟩≡

```
else {
  r[k] = (C-W)/w[k];
  W = C;
  V = V + r[k]*v[k];
  k = k + 1;
}
```

Activity Selection

Describe of some activity selection scenarios.

Problem 14: Activity Selection

Pretend there is a set $\mathbb{S} = \{a_0, a_1, \ldots, a_{(n-1)}\}$ *of* n activities *that want to use a common resource. Each activity* k *has a* start time s_k *and a* finish time f_k *where* $s_k \leq f_k$. *Activities* i *and* j *are (mutually)* compatible *if the intervals* $[s_i, f_i)$ *and* $[s_j, f_j)$ *do not overlap. That is,* $s_i \geq fj$ *or* $s_j \geq f_i$. *The activity selection problem is to select a maximally-sized set of mutually compatible activities.*

Assume the activities have been sorted by their finishing times.

$$f_0 \leq f_1 \leq \cdots \leq f_{(n-1)}$$

An example from (Corman et al., 2009) is:

k	0	1	2	3	4	5	6	7	8	9	10
s_k	1	3	0	5	3	5	6	8	8	2	12
f_k	4	5	6	7	9	9	10	11	12	14	16

The three activities in $\{a_3, a_8, a_{10}\}$ are compatible. However, $\{a_0, a_3, a_7, a_{10}\}$ is a larger set of four compatible activities. And, $\{a_1, a_3, a_7, a_{10}\}$ another set of four compatible activities.

The activity selection problem can be expressed in terms of optimal sub-problems. Let $\mathbb{S}_{ij}$ be the set of activities a_k that start after a_i finishes and finish before a_j starts.

$$\mathbb{S}_{ij} = \{a_k : (f_i \leq s_k) \wedge (f_k \leq s_j)\}$$

$$\begin{aligned}
S_{01} &= \emptyset \\
S_{02} &= \emptyset \\
S_{03} &= \{a_3\} \\
S_{04} &= \emptyset \\
S_{05} &= \emptyset \\
S_{06} &= \emptyset \\
S_{07} &= \{a_3\} \\
S_{08} &= \{a_3\} \\
S_{09} &= \emptyset \\
S_{0,10} &= \{a_3, a5, a_6, a_7, a_8\}
\end{aligned}$$

Suppose $\mathbb{A}_{ij}$ is maximal set of compatible activities.

Suppose $a_k \in \mathbb{A}_{ij}$ and let

$$\mathbb{A}_{ik} = \mathbb{A}_{ij} \cap \mathbb{S}_{ik} \quad \text{and} \quad \mathbb{A}_{kj} = \mathbb{A}_{ij} \cap \mathbb{S}_{kj}$$

so that

$$\mathbb{A}_{ij} = \mathbb{A}_{ik} \cup \{a_k\} \cup \mathbb{A}_{kj}$$

That is, computing the size of $\mathbb{A}_{ij}$ reduces to the sub-problem computation:

$$|\mathbb{A}_{ij}| = |\mathbb{A}_{ik}| + |\mathbb{A}_{kj}| + 1$$

The greedy *heuristic* is to add an activity when it leaves the resource available for as many other activities as possible. That is, choose the activity with the earliest finishing time.

Suggest other greedy heuristics.

Activity a_0 has the earliest finishing time, so it will be in the set of activities formed by this greedy approach: There are no activities that finish before a_0 starts because

$$s_0 < f_0 \leq f_k \quad (\forall k)$$

Theorem 13: Including the earliest finisher is safe

Let $\mathbb{S}_k = \{a_i : s_i \geq f_k\}$ be non-empty and let $a_m \in \mathbb{S}_k$ be the activity with the earliest finishing time. Then a_m is in some maximum size set of compatible activities.

Proof: Earliest finisher is safe

Let $\mathbb{A}_k$ be a maximum size subset of compatible activities from $\mathbb{S}_k$. Let $a_j \in \mathbb{A}_k$ be the activity with the earliest finishing time. If $a_j = a_m$ the proof is complete, so pretend $a_j \neq a_m$. Let

$$\mathbb{A}'_k = (\mathbb{A}_k - \{a_j\}) \cup \{a_m\}$$

Then $\mathbb{A}'_k$ is a set of compatible activities, $a_m \in \mathbb{A}'_k$ and

$$|\mathbb{A}'_k| = |\mathbb{A}_k|$$

In the pseudocode algorithm below. The set `A` collects the selected activities The variable j specifies the most recent addition to A The activity selected is always the one with the earliest finish time that is compatible with already selected activities

Listing 49: Activity Selection

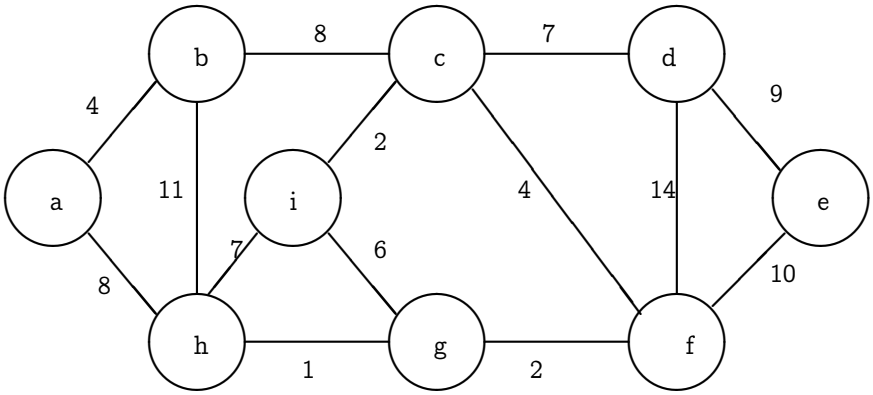
176 ⟨*Activity Selection* 176⟩≡

```
set activitySelector(int *s, int *f, int n)
{
   set A = set(0);
   j = 0;
   for (i = 1; i < n; i++)
   {
```

```
        if (s[i] > f[j]) {
          A = union(A, i);
          j := i;
        }
      }
      return A;
  }
```

Minimal Spanning Trees

Consider the graph below. It might represent the placement of workstations where edge weights would be the distances between them. A goal might be to connect each workstation while minimizing the total distance.



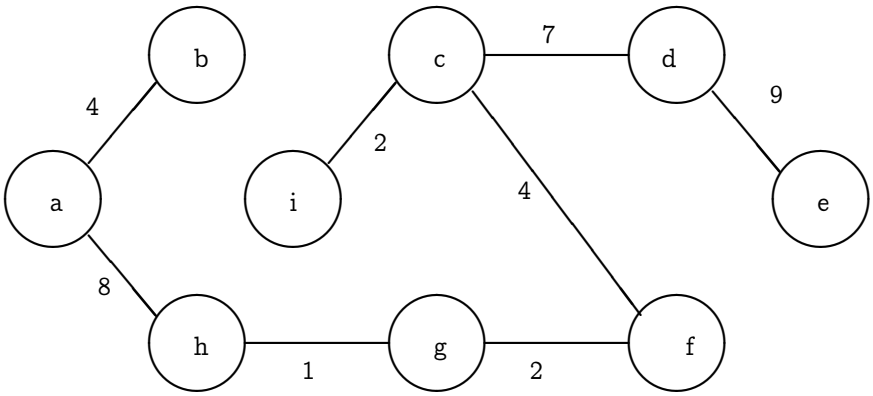
Definition 19: Trees

A tree *is an undirected graph in which any two vertices are connected by exactly one path. In other words, any acyclic connected graph is a tree.*

Definition 20: Spanning Tree

Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$*, a spanning tree for* $\mathbb{G}$ *is a tree that contains each node.*

A minimal cost spanning tree T for the graph is shown below. It has cost 37.



Problem 15: Minimal Spanning Trees

Function Problem: *Let* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *be a weighted graph. Assume that* $\mathbb{G}$ *has* n *vertices and* m *edges. Let edge weights be stored in array* `W`.
Find a spanning tree T *that minimizes*

$$c(T) = \sum_{e \in T} c(e)$$

There are several algorithms for computing the minimal cost spanning tree of a graph. We will look at Kruskal's and Prim's ideas.

Kruskal's Algorithm

Kruskal's solution starts by:

1. Making each vertex a separate tree (set)
2. In order of increasing cost, add vertices connected by edges to these sets, provided the connecting vertices don't belong to the set. (ensuring no cycles)

The cost of Kruskal's algorithm is determined by the cost to find the name of a set containing an element u and the cost to compute the union of two sets.

There are many ways to represent sets: bit strings, lists, arrays, trees. Here the question is: what representation facilitates union of sets and naming a set given an element.

Listing 50: Kruskal Minimum Spanning Tree

178 ⟨*Kruskal's Minimum Spanning Tree* 178⟩≡

```
Tree Kruskal(Graph G, double *W)
{
  set T  = NULL;
  for (each vertex v in G.V) { makeSet(v); }
  sort the edges of G.E by nondecreasing weight
  for (each sorted edge (u, v) in  G.E)
```

```
    {
       U = findSet(u);
       V = findSet(v);
       if (U != V) {
         T = union(T, (u, v));
         Union(U, V);
       }
       return T;
     }
  }
```

For the example graph above, the sorted edges are

$(1,(h,g)),\ (2,(g,f)),\ (2,(i,c)),\ (4,(a,b)),\ (4,(c,f)),\ (6,(i,g)),\ (7,(c,d)),$

$(7,(h,i)),\ (8,(a,h)),\ (8,(b,c)),\ (9,(d,e)),\ (10,(f,e)),\ (11,(b,h)),\ (14,(d,f))$

Analysis of Kruskal's Algorithm The analysis of the time complexity for Kruskal's algorithm depends on the data structure used to represent disjoint sets.

- `makeSet` simply makes a 1-element set and is $O(1)$ in almost any imaginable set implementation.
- `findSet` determines the *name* of the set to which an element belongs. For example, if value 5 belongs to a set named 11, then `findSet(5)=11`.
- `union` function returns the union of two sets.

In Kruskal's algorithm, the initial calls to `makeSet` have time complexity $O(|\mathbb{V}|) = O(n)$. The time to sort the edges has time complexity $O(|\mathbb{E}| \lg |\mathbb{E}|)$. This has worst case cost $O(2n^2 \lg n)$. Recall a complete graph on n vertices has $|\mathbb{E}| = n(n-1)/2 = \binom{n}{2}$ edges (a triangular number).

$$\begin{aligned} |\mathbb{E}| \lg |\mathbb{E}| &= \frac{n(n-1)}{2} \lg \left(\frac{n(n-1)}{2} \right) \\ &\leq \frac{n^2}{2} \lg \left(\frac{n^2}{2} \right) \\ &= n^2 \lg(n^2) - \frac{n^2}{2} \\ &= 2n^2 \lg(n) - \frac{n^2}{2} \end{aligned}$$

`findSet` *and* `union` *Operations on Sets* One implementation of `findSet` and `union` uses an array data structure

$$\vec{S} = \langle S[0], \ldots, S[n-1] \rangle$$

where $S[i] = k$ if $s_i \in S_k$ For instance, if

$$\begin{aligned}\vec{S} &= \langle S[0], \ldots, S[8] \rangle \\ &= \langle 0, 0, 2, 0, 2, 2, 0, 5, 5 \rangle\end{aligned}$$

then

$$\begin{aligned} s_0, s_1, s_3, s_6 &\in 0 \\ s_2, s_4, s_5 &\in 2 \\ s_7, s_8 &\in 5 \end{aligned}$$

Using this representation the `findSet` operation can be implemented as:

Listing 51: A simple `findSet` function

180a ⟨*Simple* `findSet` 180a⟩≡

```
int findSet(int  A)
{
  return S[A];
}
```

where `S` is an array within the scope of `findSet`. This is an $O(1)$ solution for `findSet`.

Using this implementation, `union` can be implemented as follows. Sets `A` and `B` are named by integers. To union the two sets: Find the set named `B` and rename it `A`. That is, for every element in set `B`, make it's name `A`.

The time complexity for this simple implementation of `union` is $O(n)$.

Listing 52: A simple `union` function

180b ⟨*Simple* `union` 180b⟩≡

```
int union(int A, int B)
{
  for (int i = 0; i < n; i++)
  {
    if (S[i] == B) { S[i] = A; }
  }
}
```

Union-by-rank and Path-compression The best know data structure use "union-by-rank" and "path-compression" for the `findSet` and `union` operations. They can be performed in $O(\alpha(|\mathbb{E}|\,|\mathbb{V}|))$ time where α is the inverse of the Ackermann function. Ackermann's function is defined by:

Listing 53: Ackermann Function

181a ⟨*Ackermann function* 181a⟩≡

```
ack :: Int -> Int -> Int
ack 0 n = n + 1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Here are the first few values. I killed the computations in row 4 when they did not complete quickly.

$$A = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 5 & 7 & 9 & 11 \\ 3 & 5 & 13 & 29 & 61 & 125 \\ 4 & 13 & & & & \end{array}$$

The value of element $A(4, 4)$ is

$$A(4, 4) = 2^{2^{2^{2^{2^{2}}}}} - 3$$

The inverse of Ackermann's function α, is the inverse of $A(n, n)$. In conceivable problems, this value will not be greater than 5.

Listing 54: Log Star Function

181b ⟨*Log Star function* 181b⟩≡

```
logstar :: (Ord a, Floating a) => a -> a
logstar n
  | n <= 1    = 1
  | otherwise = 1 + logstar (log n)
```

Here are the first few values. I killed the computations in row 4 when they did not complete quickly.

$$A = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 5 & 7 & 9 & 11 \\ 3 & 5 & 13 & 29 & 61 & 125 \\ 4 & 13 & & & & \end{array}$$

The value of element $A(4, 4)$ is

$$A(4, 4) = 2^{2^{2^{2^{2^{2}}}}} - 3$$

The inverse of Ackermann's function α, is the inverse of $A(n, n)$. In conceivable problems, this value will not be greater than 5.

Data Structure for Union-by-rank and findSet-with-pathcompression This implementation of `findSet` and `union` again uses an array data structure $\langle S[0], \ldots, S[n-1] \rangle$ but now

- if $S[i] = i$ then i is the name (identifier) of a set and visualized as the root of a tree of the set's elements.
- if $S[i] = j \neq i$ then i is the child of some parent j in some tree (to be determined).

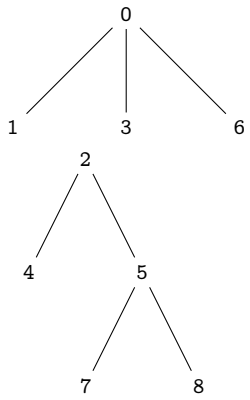
Example: Union-by-rank, FindSet-with-pathcompression

For example, let

$$\vec{S} = \langle 0, 0, 2, 0, 2, 2, 0, 5, 5 \rangle$$

This says:

- 0 *is the root of a tree containing* 1, 3, 6
- 2 *is the root of a tree containing* 4, 5
- 5 *is the root of a tree containing* 7, 8



The `findSet` operation can be implemented as follows. The idea is that given an element named `A`, follow its parent back until the root of the tree is found.

182a ⟨*Better* `findSet` 182a⟩≡

```
int findset(int A)
{
  int i = A;
  while (S[i] != i) { i = S[i]; }
  return i;
}
```

For instance, on the example above, `findSet(7)` produces

$$\begin{aligned} i &= 7; & S[7] &= 5; \\ i &= 5; & S[5] &= 2; \\ i &= 2; & S[2] &= 2; \\ \text{return} & 2; \end{aligned}$$

The best case of this `findSet` algorithm is $O(1)$ while the worst case is $O(n)$.

Why must the algorithm terminate?

What is its average case time complexity?

Now, the `union` operation can be implemented as

182b ⟨*Better* `union` 182b⟩≡

```
int union(int A, int B)
{
  if (A < B) { S[B] = A; }
  else { S[A] = B; }
}
```

The time complexity of this implementation of `union` is $O(1)$.

Example: A Better `union`

Given the data structure

$$\langle 0,\ 0,\ 2,\ 0,\ 2,\ 2,\ 0,\ 5,\ 5\rangle$$

the union of the subset with root 0 *(*$\{0, 1, 3, 6\}$*) with the subset with root* 5 *(*$\{7, 8\}$*) produces the updated data structure*

$$\langle 0,\ 0,\ 2,\ 0,\ 2,\ 0,\ 0,\ 5,\ 5\rangle$$

which says

- 0 *is the root of a tree containing* 1, 3, 5, 6
- 5 *is the root of a tree containing* 7, 8
- *Thus,* 0 *is the root of a tree containing* 1, 3, 5, 6, 7, 8
- 2 *is the root of a tree containing* 2 *and* 4.

`union` *By Rank*

- Make sure the height of the tree stays as small as possible
- Keep a record of the height (rank) of the trees
- Initially `rank(x) = 0` when `makeSet` is called
- Hang the smaller tree off the root of the larger tree
- If the height remains small, `findSet` will remain fast
- Merging two tree of height h_1 and h_2 produces a new tree with height at most $\max\{h_1, h_2\} + 1$

Listing 55: `union` by Rank (A Best `union`?)

183 ⟨*Best* `union` 183⟩≡

```
int union(int A, int B)
{
  if (rank(A) > rank(B))
  {
    S[B] = A;
  }
  else
  {
```

```
      S[A] = B;
      if rank(A) == rank(B)
      {
        rank(B) = rank(B)+1;
      }
    }
  }
```

`findSet` *with Path Compression* To improve the algorithm further, use *path compression* in the `findSet` operation. Path compression goes like this: When `findSet` is called on a non-root node, trace the edges to the root. Then retrace the edges back through the calling nodes reset each of their names to root node along the way. This recursive call `findSet(A)` returns a pointer to the root.

Listing 56: `findSet` with Path Compression

184 ⟨*Best* `findSet` 184⟩≡

```
int findSet(int  A)
{
  if (A != S[A])
  {
    S[A] = findSet(S[A]);
  }
  return S[A];
}
```

In the analysis, which is beyond scope, there are m calls to `findSet` and `union` operations. These operations are performed on a collection of n elements, each initialized as a singleton set. It can be shown, see the references in (Corman et al., 2009), that operations can be performed in

$$O(m\alpha(m, n)) \quad \text{step}$$

where

$$\alpha(m, n) = \min\{k \geq 1 \,:\, A(k, \lfloor m/n \rfloor) > \lg n\}$$

is the "inverse" of the Ackermann function. The Ackermann function grows *extremely* fast, its inverse is therefore an *very slowly* growing function.

Let $m = |\mathbb{E}|$ and $n = |\mathbb{V}|$. With `findSet` with path compression and `union` by rank, Kruskal's algorithm has time complexity

$$\text{Kruskal time complexity} = O(m \lg m)$$

This can be reasoned since

- Initialization calls to `makeSet` has cost $O(n)$

- Sorting the edges has cost $O(m \lg m)$.
- $2m$ calls to `findSet` and m calls to `union` costs

$$O(m\alpha(m, n))$$

where $\alpha(m, n)$ is almost always no more than 4.

Prim's Algorithm

Prim's algorithm has these properties:

- The constructed set always form a single tree `T`.
- The tree grows from an arbitrary vertex `r`, the root.
- The tree grows until it spans all of the vertices in G.

The *greedy heuristic* for Prim's algorithm is to always choose the next vertex to be the minimum cost edge that does not form a cycle.

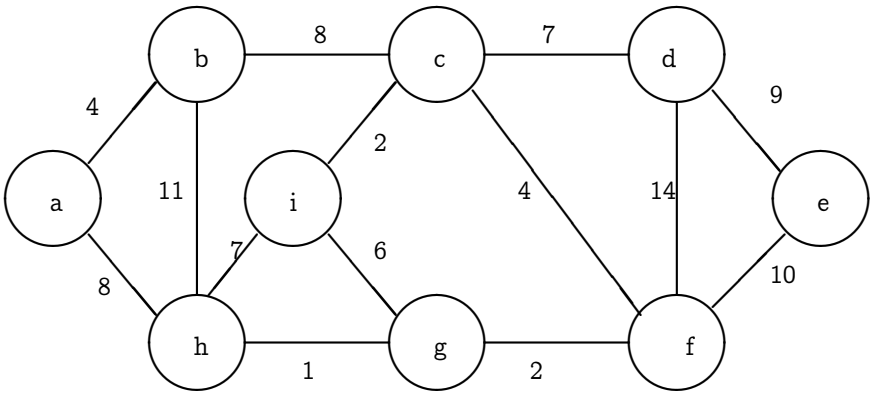
Prim's algorithm run in $O(m \lg n)$ time on *dense* graphs where m is greater than n. For a complete or nearly complete graph $m = O(n^2)$.

All vertices *not* yet in the tree reside in a priority queue `Q` based on a *key*, the minimal distance from nodes in the tree to adjacent vertices.

An `extractMin` operation removes the highest priority (closest) vertex and re-builds the heap. For each vertex v, `key[v]` is the minimum weight of any edge connecting v to a vertex in the tree.

`key[v]=maxInt` if there is no such edge. The array $\pi[v]$ names the "parent" of v in the tree An *adjacency list* is used to represent the graph (ie, for each vertex there is a list of neighboring vertices)

Consider the graph below



The (weighted) adjacency list for the graph is:

a	(b, 4)	(h, 8)		
b	(a, 4)	(c, 8)	(h, 11)	
c	(b, 8)	(d, 7)	(f, 4)	(i, 2)
d	(c, 7)	(e, 9)	(f, 14)	
e	(d, 9)	(f, 10)		
f	(c, 4)	(d, 14)	(e, 10)	(g, 2)
g	(f, 2)	(h, 1)	(i, 6)	
h	(a, 8)	(b, 11)	(g, 1)	(i, 7)
i	(c, 2)	(g, 6)	(h, 7)	

Prim's algorithm puts all nodes in a priority queue based on a key value A parent array $\pi[]$ points back to the parent of each node as they are taken off the queue

Initialization Step

Q	a	b	c	d	e	f	g	h	i
key	0	∞	∞	∞	∞	∞	∞	∞	∞

$\pi[a] = \text{NIL}$

Extract a from Q; process (a, b) and (a, h)

Q	b	h	c	d	e	f	g	i
key	4	8	∞	∞	∞	∞	∞	∞

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[h] = a$

Extract b from Q; process (b, c) and (b, h)

Q	h	c	d	e	f	g	i
key	8	8	∞	∞	∞	∞	∞

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$, $\pi[h] = a$

Extract h from Q; process (h, g) and (h, i)

Q	g	i	c	d	e	f
key	1	7	8	∞	∞	∞

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$,
$\pi[h] = a$, $\pi[g] = h$, $\pi[i] = h$

Extract g from Q; process (g, i) and (g, , f)

Q	f	i	c	d	e
key	2	6	8	∞	∞

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$,
$\pi[h] = a$, $\pi[g] = h$, $\pi[i] = g$, $\pi[f] = g$

Extract f from Q; process (f, c), (f, d) and (f, *e*)

Q	c	i	e	d
key	4	6	10	14

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = f$, $\pi[h] = a$,
$\pi[g] = h$, $\pi[i] = g$, $\pi[f] = g$, $\pi[d] = f$, $\pi[e] = f$

Extract c from Q; process (c, d) and (c, *e*)

Q	i	d	e
key	2	7	10

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = f$, $\pi[h] = a$,
$\pi[g] = h$, $\pi[i] = c$, $\pi[f] = g$, $\pi[d] = c$, $\pi[e] = f$

Extract i from Q; process no edges

Q	d	e
key	7	10

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = f$, $\pi[h] = a$,
$\pi[g] = h$, $\pi[i] = c$, $\pi[f] = g$, $\pi[d] = c$, $\pi[e] = f$

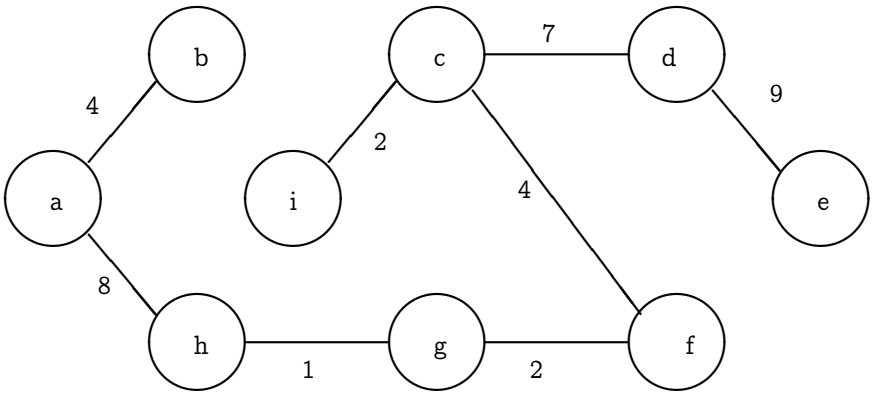
Extract d from Q; process (d, *e*)

Q	e
key	9

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = f$, $\pi[h] = a$,
$\pi[g] = h$, $\pi[i] = c$, $\pi[f] = g$, $\pi[d] = c$, $\pi[e] = d$

Extract *e* from Q; process no edges

A minimal cost spanning tree is shown below and has a cost of 37



Listing 57: Prim's Algorithm

188a ⟨*Prim's Minimum Spanning Tree* 188a⟩≡

```
Tree Prim(G, W, r)
{
  for (each vertex)
  {
    key[v] = INFINITY;
  }
  key[r] = 0;
  pi[r] = NULL;
  Q = buildHeap(key, length(G.V));
  while (Q != {})
  {
    ⟨Extract the Minimum 188b⟩
    u = extractMin(Q);
    for (each v adjacent to minimum)
    {
      if (v in Q && W(minimum, v) < key[v])
      {
        pi[v] = minimum;
        ⟨Decrease the key 189⟩
      }
    }
  }
}
```

188b ⟨*Extract the Minimum* 188b⟩≡

```
vertex extractMin(heap A)
{
  if (heapSize(A) < 1) { error; }
  min = A[1];
```

```
  A[1] = A[heapSize(A)];
  heapSize(A) = heapSize(A) - 1;
  heapify(A, 1, heapSize(A);
  return min;
}
```

189 ⟨*Decrease the key* 189⟩≡

```
decreaseKey(A, x, k)
{
  if (k > key[x]) { error; }
  key[x] = k;
  j = (where A[j] = x);
  Heapify(A, j, heapSize(A));
}
```

Analysis of Prim's Algorithm

- Let $m =| E |$ and $n =| V |$
- Initialization (setting `key` and `buildHeap` takes $O(n)$ time
- The **while** loop executes n times
- The `extractMin` executes in $O(\lg n)$ time
- The inner `for` loop executes $O(m)$ time to search the *adjacency list*
- Within the `for` loop, the test for membership is $O(1)$ if a membership bit is kept
- The call to `decreaseKey` replaces the value of `key`$[v]$ with $W(u, v)$ in the priority queue Q and fixes the heap structure afterwards, this can be done in $O(\lg n)$ time
- Prim's algorithm has time complexity $O(n \lg n + m \lg n) = O(m \lg n)$ on a dense graph where m is greater than n
- Is there data structure that allows us to obtain a faster algorithm?

Dijkstra's Algorithm (Single source shortest path)

Problem 16: Single Source Shortest Path

Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a weighted directed graph. and let $s = 0$ be a source *or* start *vertex.*

A great slide deck titled Getting From A to B can be downloaded here. Design choices that lead to optimizing Dijkstra's algorithm are presented.

Decision Problem: *Let* $1 \leq t \leq (n-1)$ *be a* target *or* sink *vertex. Let* d *be a positive integers. Is the shortest from* s *to* t *equal to* d*?*
Function Problem: *Find the shortest path between* s *and another (or all other) vertices in* $\mathbb{V}$.

Assume vertices are named $0, \ldots, (n-1)$. Let $W[u, v]$ be a matrix of edge weights between vertices u and v.

$$W[u, v] = \begin{cases} w_{uv} > 0 & \text{if } v \text{ is adjacent to } u \\ \infty & \text{otherwise} \end{cases}$$

We want to find the shortest path from the source s to each of the other nodes.

Dijkstra's algorithm works like this:

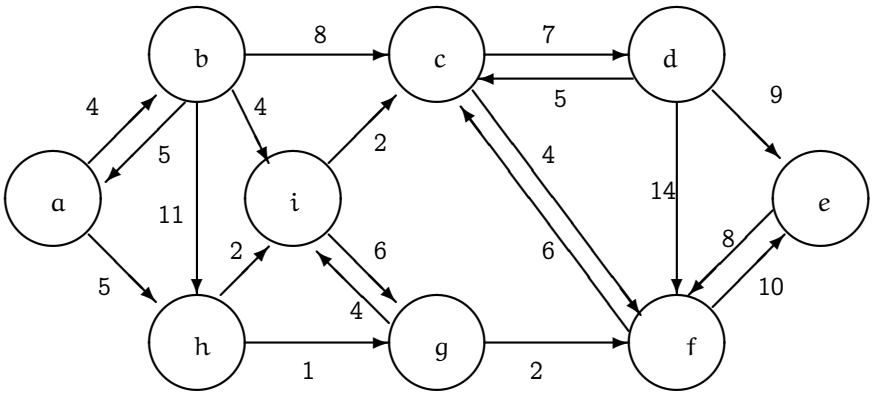
- Initialize `d[0]=0` and the remaining distances `d[1..(n-1)]` to infinity.
- Maintain a set `S` of vertices whose final shortest path from $s = 0$ has been determined. Initially `S = {0}`.
- Repeatedly choose u in `V - S` with minimum shortest path from s and update (*relax* path weights for all edges leaving u.)

  The relaxation step reduces the value of $d[v]$ if the distance from s to v is smaller by going through u.

$$d[v] = \begin{cases} d[u] + W[u, v] & \text{if } d[v] > d[u] + W[u, v] \\ d[v] & \text{otherwise} \end{cases}$$

- The algorithm maintains a priority queue Q of vertices in `V - S`. They are keyed by the values of $d[1..n]$
- An array $\pi[1..n]$ contains *predecessor* vertices along the shortest path from $s = 0$ to $u = k$.

Consider the directed graph below



The (weighted) adjacency list for the graph is:

a	(b, 4)	(h, 5)		
b	(a, 5)	(c, 8)	(h, 11)	(i, 4)
c	(d, 7)	(f, 4)		
d	(c, 5)	(e, 9)	(f, 14)	
e	(f, 8)			
f	(c, 6)	(e, 10)		
g	(f, 2)	(i, 4)		
h	(g, 1)	(i, 2)		
i	(c, 2)	(g, 6)		

Assume the source vertex is node a

Initialization Step

node	a	b	c	d	e	f	g	h	i
d	0	∞	∞	∞	∞	∞	∞	∞	∞

$Q = (a, b, c, d, e, f, g, h, i)$, $S = \emptyset$, $\pi[a] = \text{NIL}$

Extract a from Q; process (a, b) and (a, h)

node	a	b	c	d	e	f	g	h	i
d	0	4	∞	∞	∞	∞	∞	5	∞

$Q = (b, h, c, d, e, f, g, i)$, $S = \{a\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[h] = a$

Extract b from Q; process (b, c), (b, h) and (b, i)

node	a	b	c	d	e	f	g	h	i
d	0	4	12	∞	∞	∞	∞	5	8

$Q = (h, i, c, d, e, f, g)$, $S = \{a, b\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$, $\pi[h] = a$, $\pi[i] = b$

Extract h from Q; process (h, g) and (h, i)

node	a	b	c	d	e	f	g	h	i
d	0	4	12	∞	∞	∞	6	5	7

$Q = (g, i, c, d, e, f)$, $S = \{a, b, h\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$

$\pi[h] = a$, $\pi[g] = h$, $\pi[i] = h$

Extract g from Q; process (g, i) and (g, f)

node	a	b	c	d	e	f	g	h	i
d	0	4	12	∞	∞	8	6	5	7

$Q = (i, f, c, d, e)$, $S = \{a, b, h, g\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = b$, $\pi[h] = a$,

$\pi[g] = h$, $\pi[i] = h$, $\pi[f] = g$

Extract i from Q; process (i, g) and (i, c)

node	a	b	c	d	e	f	g	h	i
d	0	4	9	∞	∞	8	6	5	7

$Q = (f, c, d, e)$, $S = \{a, b, h, g, i\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = i$, $\pi[h] = a$,

$\pi[g] = h$, $\pi[i] = h$, $\pi[f] = g$

Extract f from Q; process (f, e), and (f, c)

node	a	b	c	d	e	f	g	h	i
d	0	4	9	∞	10	8	6	5	7

$Q = (c, d, e)$, $S = \{a, b, h, g, i, f\}$

$\pi[a] = \text{NIL}$, $\pi[b] = a$, $\pi[c] = i$, $\pi[h] = a$,

$\pi[g] = h$, $\pi[i] = h$, $\pi[f] = g$, $\pi[e] = f$

And so on.

Here's pseudocode for Dijkstra's algorithm: Taken from (Corman et al., 2009). It maintains a set S of vertices whose final shortest path from the source s has been determined. A priority queue Q, keyed by minimum distance to s, is used to repeatedly select a vertex u not in S with the smallest shortest path estimate.

Listing 58: Dijkstra's Single Source Shortest Path

193a ⟨*Dijkstra's Algorithm* 193a⟩≡

```
Dijkstra (G, W, s)
{
  ⟨Initialize Dijkstra's local data 193b⟩
  while (Q != ())
  {
    u = extractMin(Q);
    S = union(S, u);
    for (each v adjacent to u)
    {
      ⟨Relax 193c⟩
    }
  }
}
```

To initialize the local data, make the following assignments.

193b ⟨*Initialize Dijkstra's local data* 193b⟩≡

```
for (each vertex v) {
  distance[v] = infinity;
  pi[v] =  NULL;
  q[s] = 0;
  S = {};
  Q = buildHeap(distance, numVertices);
}
```

Relax means:

193c ⟨*Relax* 193c⟩≡

```
if (distance[v] > distance[u] + W[u, v])
{
  decreaseKey(Q, distance[v], distance[u] + W[u,v]);
  pi[v] = u;
}
```

Analysis of Dijkstra's Algorithm

Initializing d and π takes time $O(n)$ ($n = |V|$). The `buildHeap` operation requires time $O(n)$. There are n iterations of the `while` loop Inside the `while`:

- `extractMin` is $O(\lg n)$
- Insert u in the set S can be done in constant time.
- The `for` loop on v executes a total of m times where $m = |\mathbb{E}|$ with each iteration taking $O(\lg n)$ time.

Therefore, the time complexity Dijkstra's algorithm is

$$O((n + m) \lg n)$$

Is there data structure that allows us to obtain a faster algorithm?

Exercises

1. Show how the greedy algorithm for the rational knapsack problem works for 5 items with weights 7, 8, 9, 11, 12 and corresponding values 13, 15, 16, 23, 24 when the knapsack has capacity 26
2. Show how the greedy algorithm for the activity selection problem works for 6 activities with starting times 3, 0, 5, 6, 9, 2 and corresponding finishing times 5, 2, 8, 10, 12, 15
3. In this problem you are to design a greedy algorithm for the multiprocessor task selection problem. Given a set T of n task, we have for each $t \in T$ a length $l(t)$. We are also given m processors. The optimal solution to the problem selects the tasks so that each processor executes only one task at a time (to completion without interruption) with the time when last task finishes being as small as possible. Clearly describe a greedy algorithm for this problem and show how your algorithm would select 6 tasks having lengths 5, 4, 3, 4, 5, 3, 3 on $m = 3$ processors. Do you think your algorithm will always produce an optimal solution.
4. Consider the problem of making change for n cents using the least number of coins.
   a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies.
   b. Suppose that the available coins are in denominations of
   $$c^0, c^1, c^2, \ldots, c^k$$
   for some $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields the correct solution

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution.

5. Let (u, v) be the minimum-weight edge in a graph G. Show that (u, v) belongs to some minimum spanning tree of G.

6. Define the Ackermann function by

$$\begin{aligned} \Psi(1, m) &= 2^m \\ \Psi(n, 1) &= \Psi(n-1, 2) \\ \Psi(n, m) &= \Psi(n-1, \Psi(n, m-1)) \end{aligned}$$

Evaluate $\Psi(2, 1)$, $\Psi(2, 2)$, $\Psi(3, 1)$

7. Draw 2 or 3 weighted directed graphs and apply Kruskal's and Prim's algorithm to find a minimal cost spanning tree in the graphs.

8. Draw 2 or 3 weighted directed graphs and apply Dijkstra's algorithm to find the shortest path between pairs of vertices.

17. Randomized Algorithms

A *randomized* algorithm uses a probability model in its implementation.

18. Computational Complexity

The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest.

Jack Edmonds, 1966

Please read Chapter 34 NP-Completeness in in the textbook (Corman et al., 2009). I also recommend (Papadimitriou, 1994) and (Garey and Johnson, 1979) as references for computational complexity.

Decision Problems

A *decision problem* is a question (in some formal system) that has a `True` or `False` answer. A decision problem is *decidable* if there is an algorithm that correctly answers all of its instances. Here are some classic decision problems:

Imagine other decision problems. Convince yourself of the time complexity given for Sorted and Reachability

1. Sorted: Is the list $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ of integers sorted? Sorted can be solved in $O(n)$ time.
2. Reachability: Given a two vertices u and v in a graph G, is there a path from u to v? Reachability can be solved in $O(n^2)$ time, where n is the number of nodes in G.
3. 0—1 Knapsack: Given a knapsack that can hold weight C and a list of provisions $\langle p_k : k \in \mathbb{N} \rangle$ each of which has a weight w_k and value v_k. Is it possible to fill the knapsack with provisions weighing no more than C and having a total value of V or greater?

$$\sum w_k \leq C$$
$$\sum v_k \geq V$$

Presburger arithmetic is an example of class of decidable problems. Presburger arithmetic is the collection of statements P about the natural numbers $\mathbb{N}$ that only involve addition, equality, and Boolean operations among sub-expressions. The Presburger axioms are:

1. $\neg(0 = x + 1)$
2. $x + 1 = y + 1 \Rightarrow x = y$

Propositional logic studies the truth of Boolean expressions (`True` or `False` values combined using AND, OR, and NOT, and operations that can be defined from these three basic operations.)

First-order logic introduces quantification of formula that involve variables which determine the truth of a expression.

3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. Let $P(n)$ be a first-order formula in the language of Presburger arithmetic about a natural number n. The induction axiom is:

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

If P is a statement about Presburger arithmetic, then P is decidable, that is there is an algorithm that decides if P is `True` or `False`. Moreover,

- Presburger arithmetic is *consistent*: If P is derivable from (Presburger) axioms, then $\neg$P cannot be deduced from these axioms.
- Presburger arithmetic is *complete*: For each expression P, only one of P or $\neg$P is `True`, and the one that is `True` can be derived from the axioms.
- Presburger arithmetic is *decidable*: There is an algorithm that decides whether proposition P is `True` or `False`.

See (Stansifer, 1984) for additional details on on the history and significance of Presburger's discoveries.

Likewise, Gödel, and others, proved completeness for logical expressions in the first-order logic (Gödel, 1930).

Theorem 14: Gödel's Completeness Theorem

Every valid logical expression is provable. Equivalently, every logical expression is either satisfiable or refutable.

On the other hand, Gödel (Gödel, 1992) demonstrated how to construct propositions, from the Peano axioms for general arithmetic, that can not be proven `True` or `False`. Gödel realized that natural numbers could be used to name basic symbols, expressions over these symbols, and proofs. Let $G(s)$ be the Gödel number of symbol s. For instance, if

$$G(0) = 1 \qquad G(+) = 3 \qquad G(=) = 5 \qquad G(x) = 7$$

Then the axiom $x + 0 = x$ has Gödel number

$$G(x + 0 = x) = 2^7 3^3 5^1 7^5 11^7 = 1,131,912,171,637,632$$

If expression `Q` can be derived from `P` by some rule of inference, then there is a function f such that

$$f(G(P)) = G((q))$$

Let $P(n)$ be a predicate and let $G = G(P(n))$ be its Gödel number. Consider $P(G)$. This expression has a Gödel number, call it G'. And, the development goes on from here, beyond the scope of these notes.

The upshot of Gödel's coding of mathematics is that he was able to code a self-referential statement such as: `P` = "there is no proof for this statement."

If `P` is `True`, then general arithmetic is incomplete: The statement `P` is `True`, but it has no proof.

If `P` is `False`, then general arithmetic is inconsistent: There is a proof of `P`, yet `P` is `False`.

Theorem 15: Gödel's First Incompleteness Theorem

Every *consistent* formal proof system $\mathcal{F}$ about a sufficiently rich arithmetic is incomplete.

Theorem 15 says there are statements about the arithmetic we learned as children that are `True` but have no proof. Gödel's second theorem says you cannot prove a consistent arithmetic is consistent.

Theorem 16: Gödel's Second Incompleteness Theorem

If $\mathcal{F}$ is a consistent formal proof system about a sufficiently rich arithmetic, then there is no proof that $\mathcal{F}$ is consistent.

Turing Machines

An algorithm can be thought of as a Turing machine for some decision problem. Informally, a Turing machine uses a transition function δ to map the *current state* of the machine and the *character read* to a *next state*, a *character printed*, and a *direction* to move the read write head.

The next state k' either in $\mathbb{K}$, the set of states, or one of three special states: answers y "yes" and n, "no," or the "halt" state h. The read/write head can move $\leftarrow$ "left", $\rightarrow$ "right," or — "stay."

There are many ways to define a Turing machine. Here is Papadimitriou's (Papadimitriou, 1994) definition.

Definition 21: Turing Machine

A Turing machine is a 4-tuple $M = (\mathbb{K}, \Sigma, \delta, s)$ *where:*

1. $\mathbb{K}$ *is a finite set of states*
2. $s \in \mathbb{K}$ *is the initial (start) state*
3. Σ *is an alphabet (a finite set of symbols (characters).* Σ *contains two special symbols:* $\sqcup$ *and* $\triangleright$*, called* blank *and* first*, respectively.*
4. δ *is a transition function. It maps a (state, character) pair to a triple (next state, character, direction).*

$$\delta : (\mathbb{K}, \Sigma) \rightarrow (\mathbb{K} \cup \{h, y, n\}, \Sigma, \{\leftarrow, \rightarrow, —\})$$

Example: Turing machine to add 1

The transition function for a Turing machine can be defined by a state transition table. Consider adding one to a natural

number written in binary, for instance $n = (101010)_2 = 42$. *Assume after the first symbol* $\triangleright$, *each bit is written on a cell of a tape and the read/write head is positioned on the leading, leftmost, most significant bit,* 1 *in this case. A blank cell,* $\sqcup$, *lies after the rightmost, least significant bit.*
To add one to the n, *the Turing machine*

1. *Copies the bits from left-to-right until the blank cell is scanned.*
2. *When a blank is scanned, it backs up (to the left) and turns* 1*'s into* 0*'s until the first* 0 *is found.*
3. *When the first* 0 *is found, the machine changes the* 0 *into a* 1 *and halts.*

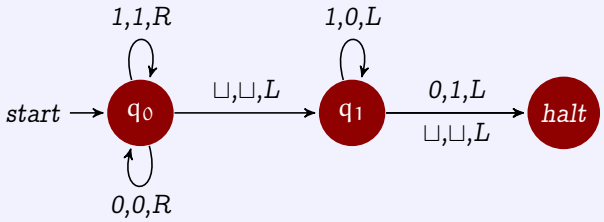
This can be described by the state transition table below. It reads: When in state q_0

- *If* 0 *or* 1 *is scanned, stay in state* q_0, *leave the bit unchanged, and move the read/write head right.*
- *If* $\sqcup$ *is scanned, move to state* q_1, *leave the blank unchanged, and move the head left.*

Similar transitions can be read for state q_1.

	0	1	$\sqcup$
q_0	$(q_0, 0, \rightarrow)$	$(q_0, 1, \rightarrow)$	$(q_1, \sqcup, \leftarrow)$
q_1	$(halt, 1, \leftarrow)$	$(q_1, 0, \leftarrow)$	$(halt, \sqcup, \leftarrow)$

The machine can also be described by a state transition diagram.



Consider how this machine operated on $(101010B)_2 = 42$. *It copies the bits from left-to-right until the blank* B *is scanned. It then moves back left and seeing the* 0, *changes it to* 1 *and halts.*
In a similar manner the string $(101011B)_2 = 43$, *is changed into* $(101100B)_2 = 44$.

The Universal Turing Machine

Turing showed (Turing, 1936) and important aspect of his machine: *It is possible to invent a single machine that can simulate any other Turing machine.* That is, there is a *universal* Turing machine, called U. The input to the universal machine U is a pair (M, x). The universal machine U computes $\mathsf{M}(x)$, that is $\mathsf{U}(\mathsf{M}, x) = \mathsf{M}(x)$.

203a ⟨*Universal Machine* 203a⟩≡

```
U(machine M, input x) {
  M(x);
}
```

The existence of a universal machine leads to *undecidable* problems, the most famous of which is the Halting Problem.

Problem 17: The Halting Problem

Decision Problem: Given a Turing machine M *and its input* x*, does* M *halt on* x*?*

There is no algorithm that *decides* the halting problem. It may be possible to decide is a particular machine M halts on a particular input x, but there is no algorithm that answers the halting problem for every instance of M and x.

Define the halting language $\mathbb{H}$ is the set of all (machine, input) pairs such that M halts on x.

The symbol $\nearrow$ stands for "does not halt."

$$\mathbb{H} = \{(\mathsf{M}, x) : \mathsf{M}(x) \neq \nearrow\}$$

There is no Turing machine that decides whether or not $(\mathsf{M}, x) \in \mathbb{H}$ for all pairs (M, x). The proof is by contradiction.

Consider the thought experiment of executing the pseudo-code below: The program accepts the encoding of a machine `M` as input. It runs `M` on `M`, looping forever if `M(M)` halts and halts if `M(M)` does not halt.

The *diagonalization* idea comes from Cantor's proof that the real numbers are uncountable.

Listing 59: The Diagonal Machine

203b ⟨*Diagonal Machine* 203b⟩≡

```
diagMac(machine M) {
  if (M(M) halts) then { Loop forever; }
    else halt;
}
```

The diagonalization name comes from running the program on itself.

203c ⟨*Diagonalization* 203c⟩≡

```
main diagMac(diagMac);
```

Now consider the logic:

- If `diagMac(diagMac)` halts, then `diagMac(diagMac)` loops forever, that is, `diagMac(diagMac)` does not halt.
- On the other hand, if `diagMac(diagMac)` does not halt, then `diagMac(diagMac)` halts.

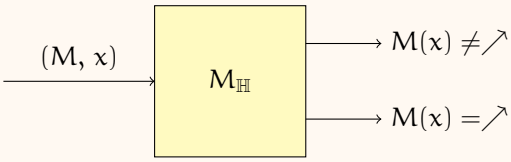
Therefore, there can be no test (algorithm) that correctly answers:

For all Turing machine `M` and for all inputs `x`, does `M` halt on `x`?

The traditional proof that the halting problem is undecidable goes something like this:

Proof: The Halting Problem is Undecidable

Pretend there is a Turing machine $M_{\mathbb{H}}$ *that decides the halting problem.*



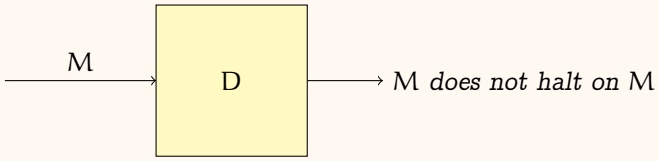
Use $M_{\mathbb{H}}$ *to construct a Turing machine* D *that accepts the encoding of a Turing machine* M *and runs* $M_{\mathbb{H}}$ *on* (M, M)*. The behavior of* D *is this:*

1. D *does not halt if* M *halts on* M*.*

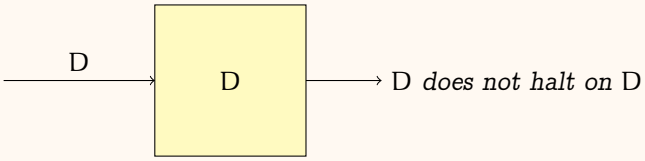
   If $(M(M) \neq \nearrow)$*, then* $D(M) = \nearrow$*.*

2. D *halts if* M *does not halt on* M $(M(M) = \nearrow)$*.*

   If $(M(M) = \nearrow)$*, then* $D(M) \neq \nearrow$



Consider D(D)



1. D *does not halt (on input* D*) if* D *halts on* D

$$\text{If } (D(D) \neq \nearrow), \text{ then } D(D) = \nearrow.$$

2. D *halts (on input* D*), if* D *does not halt on* D

$$\text{If } (D(D) = \nearrow), \text{ then } D(D) \neq \nearrow$$

This contradiction implies that the halting machine $M_{\mathbb{H}}$ *cannot exist.*

Determinism versus Non-Determinism

By default, Turing machines are deterministic: Their transition functions δ are functions. When transitions are relaxed to be *relations*, the machine is said to be *non-deterministic*.

Definition 22: The P and NP Complexity Classes

The complexity class P *is the class of all decision problems where all problem instances can be solved in in polynomial time on a (deterministic) Turing machine.* $O(n^k)$*, where* n *is the size of the instance and* k *is a fixed natural number.*

The complexity class NP *is the class of all decision problems that solve all instances in polynomial time on a non-deterministic Turing machine.*

Intuitively, the class P is the set of all problems that can be *solved* in polynomial time. Such problems are said to be *tractable*, even though they may run for a very long time.

The class NP is the set of all problems that, when given an answer (a *certificate*), the answer can be *checked* to be correct in polynomial time. Cook in his seminal paper (Cook, 1971) clearly described these ideas and their implications.

A cubic algorithm on a problem of size $n = 10^6$ will take about 10^{18} steps, which at 10^{-9} seconds per step will take about 10^9 seconds. That's about 33 years given that there are π billion seconds per century.

Exaflop machines, (10^{18} floating point instructions per second) are being developed.

Let P be a Boolean variable. Then P and $\neg$P are *literals*.

A *clause* is a disjunction of literals. For example $P \vee \neg Q$.

Conjunctive normal form is a conjunction of clauses, that is, an AND of ORs.

A *truth assignment* for a Boolean expression is an assignment of True or False to each variable such that the whole expression is True.

Problem 18: Satisfiability

Decision Problem: Given a Boolean expression B *of* n *literals in conjunctive normal form, does* B *have a* truth *assignment?*

Example: SAT Problems

The expression

$$\phi = (P \vee Q) \wedge \neg P$$

is satisfied by $P = Q =$ `False`.
On the other hand, the expression

$$\phi = (P \vee Q \vee R) \wedge (P \vee \neg Q) \wedge (Q \vee \neg R) \wedge (R \vee \neg P) \wedge (\neg P \vee \neg Q \vee \neg R)$$

is unsatisfiable. Although you can reason about this expression to see it is unsatisfiable. Notice the expression is in conjunctive normal form so a satisfying truth assignment must satisfy all clauses.
The first clause requires at least one of the three variables be `True`. *The next three clauses requires all three values be the same. (If* P *is* `True`, *then* R *must be* `True`, *and then* Q *must be* `True`. *On the other hand, if* P *is* `False`, *then* Q *must be* `False`, *and then* R *must be* `False`.*)*
But, in general, you may need to check all 2^n *truth assignments to confirm an* n *variable Boolean expression is never satisfied.*

Cook describes the *satisfiability* (SAT) problem, which is clearly in NP but is not known to be in P. The non-deterministic algorithm guesses a satisfying truth assignment for ϕ and checks that it satisfies each clause in ϕ. On the other hand, no polynomial-time deterministic algorithm has ever been discovered for satisfiability. This leads to what is said to be the fundamental problem in theoretical computer science.

Problem 19: P versus NP

Decision Problem: Does $P = NP$?

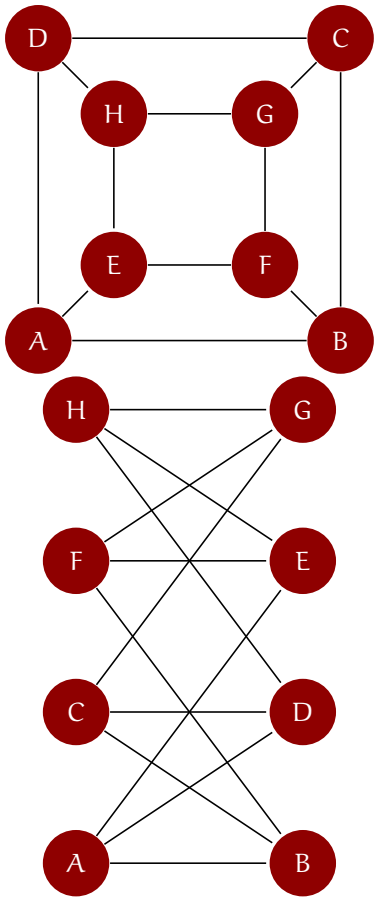
It is clear that P is a subset of NP. Whether the two classes of problems are the same remains unknown. I think the consensus is that $P \neq NP$. "Proofs" that $P \neq NP$ are proffered every so often, but at this time none has stood and no one knows for certain what the answer is.

Here are some sample NP problems. Reason that they belong to NP by convincing yourself that a answer could be checked in polynomial time.

Problem 20: Subgraph Isomorphism

Given two graphs $G_0 = (V_0, E_0)$ *and* $G_1 = (V, E_1)$. *Does* G_0 *contain a subgraph* (V, E) *such that* $|V| = |V_1|$, $|E| = |E_1|$, *and is there a one-to-one function* $f : V \mapsto V_1$ *such that* $\{u, v\} \in E$ *if and only if* $\{f(u), f(v)\} \in E_1$.

Clearly, if given nodes V and edges E, their cardinalities can be checked in polynomial time. Likewise, that f preserves edges and be checked in polynomial time.



Problem 21: Traveling Salesman

Given a finite set of cities $\mathbb{C} = \{c_0, \ldots, c_{n-1}\}$, *distances* $d(c_i, c_j) \in \mathbb{Z}^+$ *for each pair* $(c_i, c_j) \in \mathbb{C}$, *and a bound* $B \in \mathbb{Z}^+$. *Is there a tour of all cities with total length no more than* B. *That is, a permutation* $\langle c_{\pi(0)}, c_{\pi(1)}, \ldots, c_{\pi(n-1)} \rangle$ *of cities such that*

$$\left[\sum_{k=0}^{n-1} d(c_{\pi(k)}, c_{\pi(k+1)})\right] + d(c_{\pi(n)}, c_{\pi(0)}) \leq B$$

Clearly, given the tour, its cost can be computed in polynomial time.

Reductions

A classic problem solving technique is to reduce a new problem to an already solved problem. A *reduction* is an algorithm that solves problem A by transforming any instance of A to an equivalent instance of previously solved problem B. Such a reduction should be executable in

polynomial time. The notation

$$A \leq_P B$$

means if B can be solved in polynomial time, then A can be solved in polynomial time. This establishes potential ways to design algorithms.

On the other hand, if A cannot be solved in polynomial time, then neither can B, establishing *intractability*.

Consider reducing *matching* problem to *max-flow*.

Example: Matching reduced to Reachability

Given an bipartite graph (U, V, E), *where* $|U| = |V| = n$. *Construct a network of nodes* $(U \cup V \cup \{s, t\}$ *where* s *is the source and* t *is the target (sink), and with edges*

$$\{(s, u) : u \in U\} \cup E \cup \{(v, t) : v \in V\}$$

where all capacities equal to 1.
Then the bipartite graph has a matching if and only if the network has a flow of value n.

Consider reducing *validity*: Is a Boolean expression E always `True`. It can be reduced to *satisfiability*.

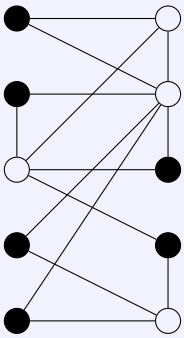
Definition 23: Validity of a Boolean Expression

A Boolean expression ϕ *is* valid *if it is* `True` *for every assignment of* `True` *or* `False` *to its variables.*

To show that Boolean expression ϕ valid, show that $\neg\phi$ is not satisfiable. If $\neg\phi$ has no satisfying truth assignment: $\neg\phi$ is always `False`. Therefore, ϕ is always `True` and valid.

Problem 22: Independent-Set

Given a graph $G = (V, E)$ *and an integer* k, *is there a subset of vertices* $S \subseteq V$ *such that* $|S| \geq k$, *and for each edge at most one of its endpoints is in* S?
The graph below shows an independent set of size 6, *the black nodes.*



Problem 23: Vertex-Cover

Given a graph $G = (V, E)$ *and an integer* k*, is there a subset of vertices* $S \subseteq V$ *such that* $|S| \leq k$*, and for each edge at least one of its endpoints is in* S*?*
The graph in problem show a vertex cover of size 4*, the white nodes.*

Theorem 17: Reducibility: Vertex-cover and Independent-Set

There is a polynomial time reduction of vertex-cover to independent-set. A subset of nodes S is an independent set if and only if $V - S$ is a vertex cover.

Proof: Reducibility: Vertex-cover and Independent-Set

Let S *be an independent set of size* k*. Then* $V - S$ *is of size* $n - k$*. If* (u, v) *is an edge, then either* $u \notin S$ *or* $v \notin S$ *(or both) Therefore, either* $u \in V - S$ *or* $v \notin V - S$ *(or both). That is, for each edge at least one of its nodes is in* $V - S$*.*
On the other hand, Let $V - S$ *be a vertex cover of size* $n - k$*. Then* S *is of size* k*. Let* $u \in S$ *and* $v \in S$*. It must be* $(u, v) \notin E$ *because* $V - S$ *is a vertex cover. (If* $(u, v) \in E$*, then at least one of* u *or* v *is in a vertex cover.) Therefore, no two nodes in* S *are joined by an edge, that is,* S *is an independent set.*

NP-*Complete Problems*

The book (Garey and Johnson, 1979) is the classic textbook on NP-completeness. A surprising number of problems have been shown to be NP-complete.

Definition 24: NP-Complete

A decision problem C is NP-complete if:

1. $C \in NP$, *and*
2. *Every problem in NP is reducible to C in polynomial time.*

Intuitively, NP-complete problems are the hardest in NP. It is not clear that there are any NP-complete problem C. And, showing that *every* problem in NP reduces to C seems to be an insurmountable task.

Cook's theorem addresses the first issue.

Theorem 18: Cook's Theorem

SAT is NP-complete.

The proof is well beyond the scope of this class.

The second issue is addressed by this result.

Lemma 1: Reduction from NP-complete problems

Let A and B be problems in NP.
If A is NP-complete and $A \leq_P B$, then B is NP-complete.

Example: 3SAT is NP-complete

Let ϕ be a Boolean expression in conjunctive normal form where each clause has at most 3 literals. Does ϕ have a truth assignment?

Co-NP *Problems*

PRIMES and COMPOSITE are examples of complementary problems.

What are precise definitions of "is prime" and "is composite"?

- PRIME $= \{n : n \in \mathbb{N}$ and n is prime$\}$.
- COMPOSITE $= \{n : n \in \mathbb{N}$ and n is composite$\}$.

Theorem 19: P is closed under complements

If problem X is in class P, then its complement $\bar{X}$ is in P too.
That is, if $X \in P$. then $\bar{X} \in P$, or more simply, $P =$ co-P.

Proof: P is closed under complements

Let A be a polynomial time deterministic algorithm for decision problem X. An algorithm $\bar{A}$ for $\bar{X}$ runs A on an instance I of X. If A accepts I, then $\bar{A}$ rejects I. Conversely, if A rejects I, then $\bar{A}$ accepts I.

For the class NP, the relationship between NP and co-NP is not as clear.

If $X \in$ NP, then there is a *certificate* (a `True` *solution*) can be *checked* in polynomial time. The *complementary* problem $\bar{X}$ requires a polynomial time *disqualification*. That is, a short proof for *no* instances.

Definition 25: Co-NP

$$co\text{-NP} = \{X : \bar{X} \in \text{NP}\}$$

The COMPOSITE decision problem is: Given a natural number $n > 1$, does it have factors other than 1 and itelf? COMPOSITE is in NP. Given the prime factorization, you can quicky check that its product is n.

Interesting: PRIMES $\in$ P was demonstrated by a 2002 paper (Agrawal et al., 2002).

The PRIMES decision problem is: Given a natural number $n > 1$, does it have no factors other than 1 and itelf? By definition PRIMES is in co-NP.

The *subset sum* problem is in NP.

Problem 24: Subset Sum

Let $\mathbb{A}$ be a finite set of integers. Does $\mathbb{A}$ contain a non-empty subset the sums to 0?

You can check in linear time that values in a non-empty subset sum to 0.

The complementary subset sum problem requires that all non-empty subsets have non-zero sums.

Problem 25: Co-Subset Sum

Let $\mathbb{A}$ be a finite set of integers. Does every non-empty subset of $\mathbb{A}$ sum to a non-zero value?

Problem 26: Unsatisfiability

Decision Problem: Given a Boolean expression B of n literals in conjunctive normal form, does B have no satisfying truth *assignment?*

SAT is in NP: Given a truth assignment that satisfies a Boolean expression B, it can be checked in polynomial time.

UNSAT is in co-NP by definition, it may not be simple to prove there is no satisfying truth assignment.

Problem 27: No Hamiltonian Cycle

Decision Problem: Given a graph $G = (V, E)$*, is there no simple cycle that contains every node of* V*?*

Can give a permutation of nodes to prove there is a Hamiltonian cycle. How can you prove there is no Hamiltonian cycle?

NP-*Hard Problems*

Definition 26: NP-Hard

Decision problem H *is* NP*-hard if every* NP*-complete problem* C *can be reduced to* H *in polynomial time.*

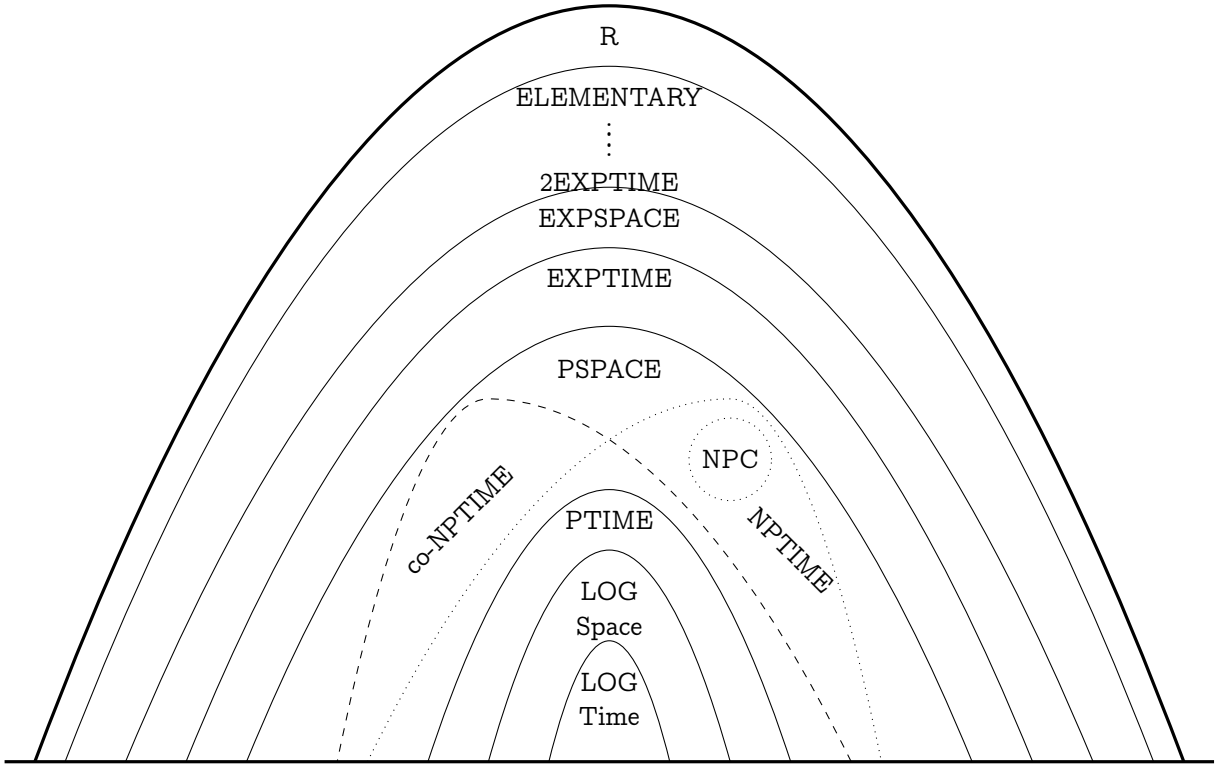
An NP-hard problem is at least as hard as the hardest problems in NP.

The *halting problem* is NP-hard. SAT can be reduced to the halting problem by transforming SAT into a Turing machine that tries all possible truth assignments for an instance I of SAT. When the machine finds a satsifying truth assignment it halts. Otherwise, if there is no satisfying truth assignment, the machine goes into an infinite loop.

PSPACE

Complexity Hierarchy

Computational complexity is complex. Here is an image from (Papadimitriou, 1994) (created by Sebastian Sardina) that shows the relationship among several complexity classes under common assumptions that have not been fully proven.


R
ELEMENTARY
2EXPTIME
EXPSPACE
EXPTIME
PSPACE
NPC
co-NPTIME
PTIME
NPTIME
LOG
Space
LOG
Time

Projects

Please be certain not to plagiarize your assignments. Moss will be used to detect software plagiarism. TurnItIn will be used to detect report plagiarism. If you turn in plagiarized work, you will receive a grade of zero on your project and be reported to the department. On a second offense you will be reported to the university.

The reason for these projects is so you will have the opportunity to:

- Write about your work.
- Explain your work to others.
- Emperically test the performance of your code.
- Compare emperical data and theoretical results.

Each project requires you to deliver a report containing

- A description of the problem.
- A well-commented program that solves the problem.
- Visualization of run-time data over a collection of inputs.
- The algorithm's theoretical run-time complexity.
- Comparison of emperical and theoretical results.

You may use any computing system that you have rights to access and you may write in any programming language you choose. Whichever programming language you choose, you must be able to collect profile data from executing your code.

Project 1

: Algorithms for Maximum Subsequence Sum

- Write a program that implements insertion sort.
- Write a program that implements quicksort.
- Write a program that generates a sequence of random integers.
- Compile your programs so that execution-time profile data is collected.
- Determine a sequence of input sizes that exercise your code.
- For each input size generate sample sequences of random integers and feed them to your insertion sort and quick sort routines.
- Compute averages of execution time and memory space usage for each inpup size and plot the results.

Project 2: Student's Choice

By Monday, October 3, 2018inform your instructor of the project you propose to complete. Rather than assign a project to graduate-level students, let me suggest some advanced areas you can explore. Several of these topics are described in the textbook (Corman et al., 2009).

- B-Trees
- Blockchain Protocol
- Fast Fourier Transform
- Floyd-Warshall Algorithm
- Huffman Codes
- Knuth–Morris-Pratt Algorithm
- Boyer–Moore Algorithm
- Aho–Corasick Algorithm
- RSA Public Key Encryption
- Skiplists

If you have another idea for a project check for approval from your instructor before proceeding.

Individual Projects Rubric

I will use the following rubric to evaluate individual. My evaluation will be honest.

Student Name: _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Compilation & Tests	The submitted code does not compile.	The code compiles but passes too few test cases.	The code compiles but passes most test cases.	The code compiles but passes all test cases.	
Documentation	The code is has no or very little documentation.	The code has some documentation as an apparent afterthought.	The code contains useful documentation.	The code is well documented, perhaps even in a literate style.	
Report Writing	Too few group members speak and can be understood.	Some group are difficult to understand.	Most group members speak clearly and are easy to understand.	The report describes the problem, algorithms that solve the problem, and analyzes results of experiments execution the program on various data sets.	
Tool Usage	Little evidence that software tools were used.	Some evidence of minimal tool usage.	Evidence that several software tools were used.	Documented use of a wide array of tools: Software configuration management tools, build tools, testing tools, debuggers, profilers, etc.	
Individual Project Average of Scores					

Algorithmics 2018

This is a call for participation in Algorithmics 2018, a workshop on algorithms that runs from November 14 to 28, 2018. It is sponsored by the School of Computing at the Florida Institute of Technology.

Teams present their research on algorithms. Research teams select a problem and report on algorithms that solve it. Team size is three. If the enrollment is not a multiple of three, some teams may be of size two or four.

Participation in Algorithmics 2018 is required.

Deadlines

Keep track of the course calendar. I am absent-minded and may not remind you.

Use the submit server for all submissions.

Guidelines from Teamwork in the Classroom

- Have clear goals
- Be results-driven
- Be a competent member
- Be committed to the goal
- Collaborate
- Have high standards
- Follow principled leadership
- Seek support, advice, and encouragement

1. Monday of week two: Research teams assigned. I'll make-up the teams.
2. Monday of week four: Teams submit the algorithm(s) they propose to study. Wikipedia has a list of algorithms from which you can choose. The textbook (Corman et al., 2009) describes many algorithms. Additional suggestions may be mentioned during class.

   - A topic paragraph summarizing the problem to be solved.
   - An brief overview of known algorithms for the problem.
   - An description of deliverables.
   - A task assignment matrix.

   If the proposal is considered insufficient, the workshop organizer will call a team meeting.
3. Friday of week eight: Teams submit a progress report.

   - An expansion on their topic paragraph.
   - A throughout description of the algorithm(s) to be analyzed.
   - Some small worked examples.
   - Illustrative drawings, diagrams, charts and graphs.
   - Pseudo-code or programming language descriptions that implement the algorithms(s).

- Each team member's accomplishments, future tasks, and impediments.

4. Starting Monday of week thirteen: Teams participate in Algorithmics 2018.

   - Every member of a team participates in a ten to fifteen minute presentation.
   - Team provides finished product to all class members.
   - Team leader submits a zip archive of the team's project.

5. Monday of week sixteen

   Each student submits:

   - Evaluations of their team members
   - Evaluations of team presentations

Rubrics

The Algorithmics 2018workshop required team work, presentation, and report writing skills. Rubrics will be used for (1) teammates to evaluate each other; (2) classmates to evaluate team presentations; and (3) the instructor to evaluate submitted reports. These rubrics are listed below so that you can know the characteristics on which you and your team will be evaluated.

Teammate Participation Rubric

Use the following rubric to evaluate each member of your group.
Your evaluation should be honest.

Group Member Name: _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Conflict	Participated in regular conflict that interfered with group progress. The conflict was discussed outside of the group.	Was the source of conflict within the group. The group sought assistance in resolution from the instructor.	Was minimally involved in either starting or solving conflicts.	Worked to minimize conflict and was effective at solving personal issues within the group.	
Assistance	Contributions were insignificant or nonexistent.	Contributed some toward the project.	Contributed significantly but other members clearly contributed more.	Completed an equal share of work and strives to maintain equity throughout the project.	
Effectiveness	Work performed was ineffective and mostly useless toward the final project.	Work performed was incomplete and contributions were less than expected.	Work performed was useful and contributed to the final project.	Work performed was very useful and contributed significantly to the final project.	
Attitude	Rarely had a positive attitude toward the group and project.	Usually had a positive attitude toward the group and project.	Often had a positive attitude toward the group and the project.	Always had a positive attitude toward the group and the project.	
Attendance & Readiness	Rarely attended group meetings, rarely brought needed materials, and was rarely ready to work.	Sometimes attended group meetings, sometimes brought needed materials, and was sometimes ready to work.	Almost always attended group meetings, brought needed materials, was ready to work.	Always attended group meetings, always brought needed materials, and was always ready to work.	
Task Focus	Rarely focused on the task and what needed to be done. Let others do the work.	Focused on the task and what needed to be done some of the time. Other group members had remind to keep this member on task.	Focused on the task and what needed to be done most of the time. Other group members could count on this person.	Consistently stayed focused on the task and what needed to be done. Other group members could count on this person all of the time.	
Group Member Average of Scores					

Modified from: Teammate Participation Rubric – Wikispaces which was taken from: from a University of Southern Mississippi site that appears stale.

Group Presentation Rubric

Use the following rubric to evaluate the presentation by each group. Your evaluation should be honest.

Group Name: _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Participation	Too few group members participate.	Some group members participate.	Most group members participate.	All group members participate equally.	
Presence	Most group members do not make eye contact and have poor body language.	Some group members do not make eye contact or have poor body language.	Most group members do make eye contact and have good body language.	All group members do make eye contact and have good body language.	
Delivery	Too few group members speak and can be understood.	Some group are difficult to understand.	Most group members speak clearly and are easy to understand.	All group members speak clearly and are easy to understand.	
Organization	Information is disorganized.	Information may be only partially organized.	Most information is presented in an organized way.	All information is presented in an organized way.	
Visual Aids	Presentation is incomplete and disorganized.	Presentation is complete but disorganized.	Presentation is organized but incomplete.	Presentation is visually organized and complete.	
Group Presentation Average of Scores					

Modified from: Read, Write, Think

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 211]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 81]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 47]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 137]

Bentley, J. (1984). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 25]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 65]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 205]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 1], [page 6], [page 10], [page 13], [page 23], [page 25], [page 31], [page 47], [page 89], [page 101], [page 129], [page 134], [page 135], [page 137], [page 169], [page 175], [page 184], [page 193], [page 199], [page 216], [page 219]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 199], [page 209]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 200]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 200]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 47]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 17]

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall, second edition. [page 23]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 22]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 143], [page 159]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 49]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 23]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 199], [page 201], [page 212]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 22]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 18]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 23]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 22]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 142]

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 23]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 36]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 200]

Stinson, D. R. (1987). *An Introduction to the Design and Analysis of Algorithms*. The Charles Babbage Research Center, P. O. Box 272, St. Norbert Postal Station, Winnipeg, Manitoba R3V 1L6, Canada, second edition. [page 121]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 203]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 11]

Index

HASKELL programming language, 20, 31, 44, 116, 137
JAVA programming language, 31

Ackermann function, 174
Axioms
  Peano, 192

Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193

Dijkstra's algorithm, 183
DNA, 146

Empty
  String λ, 61

Fibonacci, 44, 46, 142
Floating point
  Machine epsilon, 50
Function
  Ackermamm, 174
Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstader, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabreil, 44
λ calculus, 26, 28
Linear congruence, 47
Listing
  C
    Matrix Chain Multiplication, 158
Listings
  C
    A Minimum Algorithm, 135
    Aho-Corasick, 89
    Boyer-Moore Pattern Matching, 72
    Brute-force Left-to-Right Pattern Matching, 56
    Bubble Sort, 109
    Build a Heap, 126
    Counting Sort, 128
    Direct Address Table Operations, 94
    Edit Distance, 152
    Find Set with Path Compression, 177
    Greatest Common Divisor, 42
    Hash Table with Chaining, 95
    Heap Sort, 127
    Heapify a List, 126
    Inner Product, 155
    Insertion Sort, 112
    Knuth–Morris–Pratt Pattern Matching, 67
    Maximum Subsequence Sum (Cubic), 34
    Maximum Subsequence Sum (Linear), 35
    Merge Sort, 118
    Morris–Pratt Pattern Matching, 58
    Partition about a pivot, 120
    Quick Sort, 120
    Radix Exchange Sort, 132
    Radix Sort, 129
    Randomized Partition, 136
    Randomized Selection, 138
    Rational Knapsack, 167
    Selection Sort, 114
    Shell Sort, 116
    Union by rank, 177
  Haskell
    Ackermann function, 174
    Apply a Function Repeatedly, 50
    Brute-force Left-to-Right Pattern Matching, 57
    Bubble Function, 108
    Bubble Sort, 109
    Extended Euclidean Algorithm, 48
    Fold from the left, 37
    Greatest Common Divisor, 44
    Inner Product, 155
    Insertion Sort, 112
    Matrix Multiplication, 155
    Maximum Subsequence Sum, 37
    Merge Sort, 117
    Merging Sorted Lists, 117
    Newton's Square Root Method, 51
    Newton's Square Root Recurrence, 50
    Quicksort, 120
    Randomized Partition, 137
    Relative Error Convergece Test, 50
    Sample Code Header, 31
    Second of a pair, 36
    Selection Sort, 114
    Sorted Decision Problem, 106

Splitting a List, 117
Swapping head with Another Element, 137
Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra's Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181
Lucas, Édouard, 44

Memorization, 141
Metric sace, 152

Newton's method, 13
Newton, Issac, 49
Noweb, 30, 55

Object-oriented programming, 31

Pattern Matching, 53, 54
Boyer-Moore, 71
Brute Force, 55
Knuth–Morris–Pratt, 67
Morris–Pratt, 58
Right-to-Left Brute Force, 70
φ, 142
Presburger arithmetic, 191

RAM, 26, 29

Sequence, 19
Lucas, 44
Sets
Cartesian product, 142
Simpson's rule, 51
Sorting Algorithms
Bubble Sort, 107
Bucket Sort, 129
Counting Sort, 128
Heap Sort, 124
Insertion Srt, 111
Merge Sort, 116
Quick Sort, 119
Radix Sort, 129
Selection Sort, 114
Shell Sort, 115
Time complexity, 127
String
Border, 61
Period, 61
Prefix, 61
Suffix, 61
Syllabus, 5

Theorems
Bezout's, 47
Euclidean division, 41
Lamé's, 45
Taylor's, 49, 51
Turing
Machine, 9, 26, 28, 192

Wikipedia, 201

xkcd
Algorithms by Complexity, 25
Calendar of meaningful dates, 7
Haskell, 30
Ineffective Sorts, 105
Traveling Salesman Problem, 141