CSE 5211
ANALYSIS
OF
ALGORITHMS

(VERSION 1/1)

WILLIAM DAVID SHOAFF

(OKTOBER 31, 2016)
## Contents

*Course Syllabus* 5

*Proposed Course Calendar* 7

1. *Preparation* 9

2. *Introduction* 25

3. *Algorithmic Design* 33

4. *Euclid, Fibonacci, Lamé & Lucas* 39

5. *Numerics* 49

6. *Pattern Matching* 53

7. *All Keywords in Text* 87

8. *Dictionaries and Hash Tables* 93

9. *Sorting* 105

10. *Medians and Order Statistics* 135

11. *Dynamic Programming* 141

12. *Greedy Algorithms* 163
# Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sketch of Turing machine by Sebastian Sardina</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Morris–Pratt: Extend border</td>
<td>65</td>
</tr>
<tr>
<td>3</td>
<td>Morris–Pratt: Shrink border</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>Alignment with match using right-to-left scan</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>Boyer-Moore shift between 1 and j positions</td>
<td>72</td>
</tr>
<tr>
<td>6</td>
<td>Boyer-Moore shift between j+1 and m positions</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>The last occurrence or bad character heuristic</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>The Aho-Corasick goto function</td>
<td>88</td>
</tr>
<tr>
<td>9</td>
<td>The Aho-Corasick failure function</td>
<td>88</td>
</tr>
<tr>
<td>10</td>
<td>Triangular fan</td>
<td>142</td>
</tr>
<tr>
<td>11</td>
<td>Triangular strip</td>
<td>143</td>
</tr>
<tr>
<td>12</td>
<td>Triangulations with perimeter weights</td>
<td>144</td>
</tr>
<tr>
<td>13</td>
<td>Sketch of a Turing Machine</td>
<td>193</td>
</tr>
</tbody>
</table>
Consult the syllabus for details about the course: Assignments, content, contacts, material, policy, grades, and so on.

But, I give in, you are most interested in what you have to do:

- Teams formed and submit projects for Algorithmics 2016.
- A project to show you can program, conduct experiments, and report your results.
- A midterm examination to show mastery of topics so far.
- Teams report on their progress for Algorithmics 2016.
- An second project to improve upon your first.
• A team presentation to show you can play with and explain to others.

• A final examination to show you remember, at least for now.
Proposed Course Calendar

The course calendar predicts when class events are expected to happen. It is not written in stone. Nothing is certain. Things may change. Pay attention. Colors are used in the following calendar to indicate an exam or assignment due date, a holiday, or a link to additional information,

Week 1: Course structure (Syllabus);
Preparation

Week 2: Introduction to algorithms;
Algorithm design;
Teams formed (Friday 09/02);

Week 3: Labor Day 09/05;
Euclid, Fibonacci & Lamé

Week 4: Proposals submitted (Monday 09/12);
Numerical algorithms
Text algorithms & pattern matching

Week 5: More text algorithms
First project due (09/19);

Week 6: Hash Tables (Dictionaries and Maps) Minimal coverage
Start sorting

Week 7: Midterm exam (Monday 10/3)
Finish sorting

Week 8: Fall Break 10/10 – 10/11;
Dynamic programming
Progress report (Friday 10/14);

Week 9: Greedy algorithms

Week 10:

Week 11: Second project due (10/31);
Week 12: Computational complexity
    Veterans Day 11/11

Week 13:

Week 14: Algorithmics 2016 Thanksgiving 11/23 – 11/25

Week 15: Algorithmics 2016

Week 16: Review;
    Study Days 12/08 – 12/09

Week 17: Final examination: Friday, December 16, in FTC 244, from 1:00 p.m. to 3:00 p.m.
1. **Preparation**

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

**Common Functions**

- **Polynomials:**
  \[ f(n) = 1, \quad f(n) = n, \quad f(n) = n^2, \quad f(n) = n^3, \ldots \]
  \[ f(n) = 1, \quad f(n) = n, \quad f(n) = n(n-1), \quad f(n) = n(n-1)(n-2), \ldots \]

- **Poly-Logarithmic:**
  \[ f(n) = \lg(n), \quad f(n) = n\lg(n), \quad f(n) = n^2\lg n, \ldots \]

- **Exponential:**
  \[ f(n) = 2^n, \quad f(n) = n!, \quad f(n) = n^n, \ldots \]

**Notes:**

1. \( \lg(n) \) is the logarithm of \( n \) base 2.
   \[ \lg(n) = \frac{\log(n)}{\log(2)} \approx 3.3219 \ldots \log(n) \]

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

<table>
<thead>
<tr>
<th>Function</th>
<th>n = 1</th>
<th>n = 10</th>
<th>n = 100</th>
<th>n = 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>0</td>
<td>3.32</td>
<td>6.64</td>
<td>9.96</td>
</tr>
<tr>
<td>( n )</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>0</td>
<td>33.2</td>
<td>664</td>
<td>9966</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>1</td>
<td>100</td>
<td>10000</td>
<td>10^6</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>1</td>
<td>1000</td>
<td>10^6</td>
<td>10^9</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>2</td>
<td>1024</td>
<td>10^{30}</td>
<td>10^{300}</td>
</tr>
<tr>
<td>( n^n )</td>
<td>1</td>
<td>10^{10}</td>
<td>10^{200}</td>
<td>10^{3000}</td>
</tr>
</tbody>
</table>

**Big-O Notation**

**Definition 1: Big-O notation**

\[ f : \mathbb{N} \rightarrow \mathbb{R} \text{ is } \text{big-O of } g : \mathbb{N} \rightarrow \mathbb{R}, \text{ denoted } f = O(g) \]

\[ f \text{ is big-O of } g \]

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 \( f = O(g) \), then \( cf(n) = O(g) \) for any constant \( c > 0 \).
4. If \( f_1 = O(g_1) \) and \( f_2 = O(g_2) \), then
   \[ f_1 + f_2 = O(\max(g_1, g_2)) \]
   \[ f_1 f_2 = O(g_1 g_2) \]

\[ \text{Omega Notation} \]

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 \( n > m \).

\[ \text{Definition 2: Big-\( \Omega \) notation} \]

\( f : \mathbb{N} \rightarrow \mathbb{R} \) is big-Omega of \( g : \mathbb{N} \rightarrow \mathbb{R} \), 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 \( \Omega(\sqrt{n}) \)

\[ \text{Theta Notation} \]

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-$\Theta$ notation

$f: \mathbb{N} \to \mathbb{R}$ is big-theta of $g: \mathbb{N} \to \mathbb{R}$, 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 $\Theta(n \log n)$

Theorem 2: Big-$\Theta$ 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

$f: \mathbb{R} \to \mathbb{R}$ is little-o of $g: \mathbb{R} \to \mathbb{R}$, denoted $f(x) = o(g(x))$

means for all $c > 0$ there exists some $m > 0$ such that $0 \leq$
\[ f(x) < cg(x) \text{ for all } x \geq m. \]

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

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

- Given a positive integer \( n \), define
  \[
  n! = 1 \cdot 2 \cdots n
  \]

- Gauss's trick to show \( n! \) is pretty big
  \[
  (n!)^2 = (1 \cdot 2 \cdots n)(n \cdots 2 \cdot 1) = \prod_{k=1}^{n} k(n-k+1)
  \]

  For all integer \( k \) between 1 and \( n \), we have
  \[
  n \leq k(n+1-k) \leq \frac{1}{4}(n+1)^2,
  \]
  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
  \[
  \prod_{k=1}^{n} n \leq (n!)^2 \leq \prod_{k=1}^{n} \frac{(n+1)^2}{4}
  \]
  that is,
  \[
  n^{n/2} \leq n! \leq \left( \frac{n+1}{2} \right)^n
  \]

- 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
  \[
  n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{5184n^3} + O\left( \frac{1}{n^4} \right) \right)
  \]

- Stirling's formula is called an asymptotic formula for \( n! \)

Binomial Coefficients

- 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
  \[
  \binom{n}{k} = \frac{n(n-1) \cdots (n-k+1)}{k!}
  = \frac{n!}{k!(n-k)!}
  \]
• Pascal’s Triangle is an array of binomial coefficients

<table>
<thead>
<tr>
<th>n ( \backslash ) k</th>
<th>0 1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
<tr>
<td>2</td>
<td>1 2 1</td>
</tr>
<tr>
<td>3</td>
<td>1 3 3 1</td>
</tr>
<tr>
<td>4</td>
<td>1 4 6 4 1</td>
</tr>
<tr>
<td>5</td>
<td>1 5 10 10 5 1</td>
</tr>
<tr>
<td>6</td>
<td>1 6 15 20 15 6 1</td>
</tr>
</tbody>
</table>

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:
\[
\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}
\]

• Summation on Upper Index
\[
\sum_{m \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}
\]

• \( 2^n = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} \).

• \( 0^n = \binom{n}{0} - \binom{n}{1} + \cdots + (-1)^n \binom{n}{n} \).

• \( (1+z)^r = \sum \binom{r}{k} z^k, \quad |z| < 1. \)

• \( 1 + 2 + \cdots + n = \binom{1}{1} + \binom{2}{1} + \cdots + \binom{n}{1} = \binom{n+1}{2} \)
Harmonic Numbers

- Define the $n$th 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 $k$th 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

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b(x^n) = n \log_b x$
- $\log_b x = \frac{1}{\log_a x}$
- $\log_b x = \frac{\log_a x}{\log_a b}$
- $x^{\log_b y} = y^{\log_b x}$

- 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 \textit{basis for induction}

• Proving $P(n) \Rightarrow P(n+1)$ is called the \textit{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 $P(n+1)$

\[
\begin{align*}
1^2 + 2^2 + \cdots + n^2 + (n+1)^2
&= \frac{1}{3}n \left( n + \frac{1}{2} \right) (n+1) + (n+1)^2 \\
&= \frac{1}{3}(n+1) \left( n + \frac{3}{2} \right) (n+2)
\end{align*}
\]

• 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 $P(2n)$

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

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

- Arithmetic sum

\[ \sum_{k=0}^{n-1} (a + kd) = na + d \left( \frac{n}{2} \right) \]

- 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

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

- Derivative of geometric series

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

- 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

\[ T(n) = T(n/2) + 1, \quad n \in \{m | m \mod 2 = 0, m \geq 2\} \]

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

\[ T(n) = 2T(n/2) + n, \quad n \in \{m | m \mod 2 = 0, m \geq 2\} \]

What function \(T(n)\) satisfies the recursion

\[ T(n) = 2T(n/2) + 1, \quad n \in \{m | m \mod 2 = 0, m \geq 2\} \]
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 $T(n) = \Theta(n^{\log_b a \log n})$.
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 $\bar{S} = \langle s_0, s_1, s_2, \ldots \rangle$ be a sequence. The generating function for $\bar{S}$ is

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

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

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

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

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

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

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

\[ \langle \text{Stack operations 20a} \rangle \equiv \]

isEmptyp :: Stack a -> Bool
pushp :: a -> Stack a -> Stack a
popp :: Stack a -> (a, Stack a)
topp :: Stack a -> a

Queues

\[ \langle \text{Queue operations 20b} \rangle \equiv \]

tenqueue :: a -> Queue a -> Queue a
tendequeue :: Queue a -> (a, Queue a)

Lists

\[ \langle \text{List operations 20c} \rangle \equiv \]

(++) :: [a] -> [a] -> [a]
headp :: [a] -> a
tailp :: [a] -> a
last :: [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

Definition 5: Directed Graph

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

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 \( ^n k \) a maximum, when \( n \) is a given positive integer?

10. How many \( n \)-bit binary numbers have \( k \) bits set 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, ...
   (b) 1, 2/3, 3/9, 4/27, ..., (n + 1)/3^n, ...
   (c) 1, 1, 2, 2, 3, 3, 4, 4, ...
   (d) 0, 5, 50, ..., n5^n, ...

18. Determine a discrete numeric sequence for the following generating functions.
   (a) \( \frac{1}{1+ze} \)
   (b) \( \frac{1}{5-6z+z^2} \)
   (c) \( \frac{2z^3}{(1-2z)(1+3z)} \)
   (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 \( f(n) = \Theta(g(n)) \).

\[
\begin{array}{cccccccc}
\lg(\lg(n)) & 2\lg(n) & (\sqrt{2})\lg(n) & n^2 & n! \\
(3/2)^n & n^3 & \lg^2 n & \lg(n!) & \log(\log(n)) \\
n^* 2^n & \lg n & 1 & \ln n & 2\log n \\
e^n & 5n^3 + 4n^2 & \sqrt{n^2} & \lg(2^n) & 2^n n \\
4^n & (n + 1)! & n^2 + n + 3 & n & 2n
\end{array}
\]

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
\[ \sum_{i=0}^{\infty} \frac{a^i}{i+1} \]

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_n )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

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 (·) 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 (· 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 20?
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)

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.

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 function 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 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.

Church’s \( \lambda \) calculus is another system equivalent in computing power to Turing machines. The basic ideas in \( \lambda \) 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

\[(\lambda x.f)s \xrightarrow{\beta} f(x := s)\]

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:

\[(\lambda x.f)s \xrightarrow{\eta} f\] when \(f\) does not contain \(x\) 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.

**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.
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Op Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD</td>
<td>operand</td>
</tr>
<tr>
<td>2</td>
<td>STORE</td>
<td>operand</td>
</tr>
<tr>
<td>3</td>
<td>ADD</td>
<td>operand</td>
</tr>
<tr>
<td>4</td>
<td>SUB</td>
<td>operand</td>
</tr>
<tr>
<td>5</td>
<td>MULT</td>
<td>operand</td>
</tr>
<tr>
<td>6</td>
<td>DIV</td>
<td>operand</td>
</tr>
<tr>
<td>7</td>
<td>READ</td>
<td>operand</td>
</tr>
<tr>
<td>8</td>
<td>WRITE</td>
<td>operand</td>
</tr>
<tr>
<td>9</td>
<td>JUMP</td>
<td>address</td>
</tr>
<tr>
<td>10</td>
<td>JGTZ</td>
<td>address</td>
</tr>
<tr>
<td>11</td>
<td>WRITE</td>
<td>address</td>
</tr>
<tr>
<td>12</td>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>

**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.
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

```haskell
{- | 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 learned.
3. Algorithmic Design

This section is about algorithmic design. To gain a deeper understanding, read Jon Bentley’s Programming Pearl “Algorithm Design Techniques,” (Bentley, 1984a) 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

\[ X = [-1, -2, 3, 5, 6, -2, -1, 4, -4, 2, -1] \]

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.

Brute Force

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

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, \ldots, and 1 sum with 11 terms.

Computing these sums can take up to

\[ 11 \cdot 1 + 10 \cdot 2 + 9 \cdot 3 + \ldots + 1 \cdot 11 = 286 \]

additions.

\[
\sum_{k=0}^{n} (n-k)k = \frac{(n+2)(n+1)n}{6} = O(n^3)
\]
Listing 2: Cubic algorithm

(Cubic time maximum subsequence sum 34a)≡
int maxSubseqSum(int X[], int n) {
    int MaxSoFar = 0; // local state
    ⟨For each start of a subsequence 34b⟩ {
        ⟨For each end of the subsequence 34c⟩ {
            int Sum = 0; // More local state
            ⟨For each subsequence 34d⟩ {
                ⟨Compute partial sum; Check MaxSoFar 34e⟩
            }
            return MaxSoFar;
        }
    }
}

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

(For each start of a subsequence 34b)≡
for (int start = 0; start < n; start++)

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

(For each end of the subsequence 34c)≡
for (int end = start; end < n; end++)

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

(For each subsequence 34d)≡
for (int k = start; k <= end; k++)

(Compute partial sum; Check MaxSoFar 34e)≡
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 \( \langle \text{Compute partial sum; Check MaxSoFar} \rangle \) 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\[ \sum_{e=s}^{n-1} (e - s + 1)c = \sum_{k=1}^{n-s} kc = c \frac{(n - s + 1)(n - s)}{2} \]Finally, the time complexity of the outer for loop on \( s \) can be computed by\[ \sum_{s=0}^{n-1} c \frac{(n - s + 1)(n - s)}{2} = c \sum_{s=0}^{n-1} \left( \frac{n - s + 1}{2} \right) = c \left( \frac{n + 2}{3} \right) = c \frac{(n + 2)(n + 1)n}{6} \]

**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**

\[
\langle \text{Imperative linear time algorithm} \rangle \equiv \\
\text{int maxSubseqSum(int x[], int n) \{} \\
\quad \langle \text{Local state} \rangle \\
\quad \text{for } (\langle \text{Every end position} \rangle) \{ \\
\} \\
\}
\]
We need to keep track of the maximum so far and the maximum that ends at some position. Both can be initialized to 0.

```plaintext
36a Local state 36a
int MaxSoFar = 0;
int MaxEndingHere = 0;
```

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

```plaintext
36b Every end position 36b
(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.

```plaintext
36c Compute the maximum that ends here 36c
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.

```plaintext
36d Compute the maximum so far 36d
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. A helper function foldl 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.

```plaintext
36e The second of a pair function 36e
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.
Listing 5: The foldl functions

(Folding a from the left 37a)≡
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

(Linear time algorithm 37b)≡
maxsubseq :: (Ord a, Num a) => [a] -> (a, [a])
maxsubseq = snd . foldl f ((0,[0]),(0,[0])) where
  f ((maxToHere,witnessToHere),sofar) x = (a,b) where
    a = max (0,[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

\( T(n+1) = T(n) + n \) with initial condition \( T(1) = 1 \)

4. What is the solution to the recurrence equation

\( T(n+1) = T(9n/10) + n \) 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.

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.

9 liters = 7 flagons
1 liter = 0.7777... flagons
1.285714... liters = 1 flagon

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

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.

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!
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.

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.

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;

\[
\begin{align*}
1 \text{ flagon} &= 1 \text{ liter} + \text{ flagon-residue} \\
1 \text{ liter} &= 2 \text{ flagon-residue} + \text{ liter-residue} \\
1 \text{ flagon-residue} &= 2 \text{ liter-residue}
\end{align*}
\]

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 liter-residue = \(\frac{1}{2}\) flagon-residues

Therefore,

1 liter = 3 flagon-residue plus 1 liter-residue = \(\frac{7}{2}\) flagon-residues

and

1 flagon-residue = \(\frac{2}{7}\) liters

Finally,

1 flagon = 1 liter plus 1 flagon-residue = \(\frac{9}{7}\) 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.

\[
\begin{array}{c|cccc}
\text{Dividend} & \text{Divisor} & \text{Quotient} & \text{Remainder} \\
\hline
34 & 21 & 1 & 13 \\
\end{array}
\]

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.

\[ \gcd(a, m) = \max \{k \mid k \mid a \text{ and } k \mid m\} \]

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

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

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.

<table>
<thead>
<tr>
<th>Theorem 5: Greatest Common Divisor Recurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ( 0 \leq m &lt; a ). Then,</td>
</tr>
<tr>
<td>( \gcd(a, 0) = a )</td>
</tr>
<tr>
<td>( \gcd(a, m) = \gcd(m, a \mod m), ) for ( m &gt; 0 )</td>
</tr>
</tbody>
</table>

In C, the code might look something like this:

```c
42a ⟨Imperative GCD algorithm 42a⟩≡
int gcd(int a, int m) {
  ⟨GCD local state 42b⟩
  while (⟨The divisor m is not 0 43a⟩) {
    ⟨Move m to a and a mod m to m 42c⟩
  }
  ⟨Return the absolute value of a 43b⟩
}
```

To change the values: \( m \) goes to \( a \), and \( a \mod m \) goes to \( m \), a local temporary value \( t \) is used.

```
42b ⟨GCD local state 42b⟩≡
int t;
```

```
42c ⟨Move m to a and a mod m to m 42c⟩≡
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.

\[43a\]
\[\langle The\ divisor\ m\ is\ not\ 0\ \rangle \equiv\]
\[m\]

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.

\[43b\]
\[\langle Return\ the\ absolute\ value\ of\ a\ \rangle \equiv\]
\[return\ a < 0 ? -a : a;\]
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, $\text{gcd}(0, 0)$ is undefined and raises an error.

```
Listing 8: Functional GCD algorithm

44 (Functional GCD algorithm 44)≡
  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 $\text{gcd}(34, 21)$, see in 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:

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

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.

\[ 10 = 2 \cdot 5 + 0 \]
\[ 12 = 10 + 2 \]
\[ 22 = 12 + 10 \]
\[ 34 = 22 + 12 \]
\[ \vdots \]
\[ 10F_{n} + 2F_{n-1} \text{ for } n \geq 3. \]

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.

\[ gq = g \cdot q + 0 \]
\[ gq + g = gq + g \]
\[ 2gq + g = (gq + g) + gq \]
\[ 3gq + 2g = (2gq + g) + (gq + g) \]
\[ \vdots \]
\[ 5gq + 3g = (3gq + 2g) + (2gq + q) \]
\[ gqF_{n} + gF_{n-1} \text{ for } n \geq 3. \]

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

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

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

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

Let $F_i$ denote the $i$th Fibonacci number. Then

\[
\begin{align*}
    r_n &\geq 1 = F_2 \\
    r_{n-1} &= r_n q_n \geq 2r_n \geq 2 = F_3 \\
    r_{n-2} &\geq r_{n-1} + r_n \geq F_3 + F_2 = F_4 \\
    &\vdots \\
    r_2 &\geq r_3 + r_4 \geq F_{n-1} + F_{n-2} = F_n \\
    r_1 &\geq r_2 + r_3 \geq F_n + F_{n-1} = F_{n+1}
\end{align*}
\]

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

\[
m = r_1 \geq F_{n+1} > \phi^{n-1}
\]

Take the logarithm base $\phi$ of both sides and use the change of base formula for logarithms to derive the inequality

\[
\log_\phi m = \frac{\lg m}{\lg \phi} > n - 1
\]

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

\[
3\lg m > \frac{\lg m}{\lg \phi} > n - 1
\]

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 $\frac{5}{3}$ and $\frac{8}{5}$ use a common denominator. In this case $3 \cdot 5 = 15$.

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

Coding the extended Euclidean algorithm

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

**Theorem 7: Bézout’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: Bézout’s identity**

Let $L = \{ax + my > 0 : x, y \in \mathbb{Z}\}$ 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 $L$. 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

$$r = a - dq$$

$$= a(1 - tq) + m(sq)$$

$$\in \{ax + my \geq 0 : x, y \in \mathbb{Z}\} \text{ and } 0 \leq r < d.$$ 

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](https://example.com). The original is by by Trevor Dixon.
Listing 9: Haskell Extended Euclidean Algorithm

```haskell
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

2. What is the time complexity of the algorithm in listing 9.
3. Show that \( \text{lcm}(a, m) \text{gcd}(a, m) = am \).
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.

---

Much of early computing revolved around numerical algorithms that approximate values really sought. Ancient records show algorithms to approximate the value of $\sqrt{2}$, $\pi$ and other useful numbers. Algorithms that compute numbers like $\sqrt{2}$ or $\pi$ cannot not 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**

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

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

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

$$(x_0, x_1, x_2, \ldots)$$

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

$$x = \frac{x + m}{2}$$

$$2x = x + m/x$$

$$x = m/x$$

$$x^2 = m$$

$$x = \sqrt{m}$$

---

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 more or less arbitrary 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.

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

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

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

```haskell
⟨Newton’s square root recurrence 50a⟩≡
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 repeatedly applies a function $f :: \text{Double} \rightarrow \text{Double}$ and an initial value $a$ to generate an infinite list.

### Listing 11: Repeatedly Apply a Function

```haskell
⟨Repeatedly apply a function 50b⟩≡
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 is between successive iterates is:

$$|x_k - x_{k-1}|$$

Computer arithmetic on floating point numbers is not exact. The absolute difference may 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 approaches 1. This is the relative difference

$$\left|\frac{x_k}{x_{k-1}} - 1\right|.$$
Now we can express Newton's method to compute the square root of \( m \) to within a tolerance \( \epsilon \) starting with an initial guess \( x_0 \) as the function:

\[
\text{my} \text{sqrt} \; x_0 \; \text{eps} \; m = \text{relative} \; \text{eps} \; (\text{repeatedly} \; (\text{next} \; m) \; x_0)
\]

**Convergence of Newton's Method**

The number of times the \((\text{next} \; m)\) function is evaluated measures the time complexity of the \( \text{my} \text{sqrt} \) 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

\[
x^2 - m = (x_{k-1}^2 - m) + 2x_{k-1}(x - x_{k-1}) + (x - x_{k-1})^2
\]

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,

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

That is, the 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.

**Exercises**

1. **Simpson’s rule** is a simple quadrature algorithm. It is defined by the equation

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

Prerequisite concepts

Definition 7: Alphabets, Strings, ...

- An alphabet $\Sigma$ is a finite set of symbols, often called characters or letters.
- A string $s$ over $\Sigma$ is a finite sequence of symbols from $\Sigma$.
- $\Sigma^*$ (the Kleene closure) is the set of all strings over $\Sigma$.
- $\Sigma^+$ is the set of all non-empty strings over $\Sigma$.
- A language $L$ is a subset of $\Sigma^*$.
- 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.

Imagine applications where pattern matching could be useful.

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

(Pattern match 54)≡

(Brute-force pattern matching with left-to-right scan 56a)
(Morris–Pratt pattern matching 58)
(Knuth–Morris–Pratt pattern matching 67)
(Boyer-Moore pattern matching with right-to-left scan 71a)
(Boyer-Moore pattern matching 72)
(Auxiliary functions 66a)
(Test the pattern matching algorithms 81c)
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:

```
text  b a b c b a b c a b c a b c a b c a b c a b 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 b c a b c a b c a b 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 text. 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 b c a b c a b c a b 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.

```
⟨ 1, 4, 1, 1, 8, 1, 1, 5, 1, 1, 2, 8, 1, 1, 10 ⟩
```

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

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

bool bruteForce (const char *txt, const char *pat) {
    // Initialize local state
    int n = strlen(txt);
    int m = strlen(pat);
    int i = 0, j = 0;

    while ((i <= n-m)) {
        // Scan left-to-right
        if (pat[j] == txt[i+j]) return true;
        // Shift the pattern one place in the text
        i++;
    }
    return false;
}
```

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.

```c
Initialize local state
```

```c
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.

```c
text index is legal
```

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

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

```c
pattern index is legal
```

```c
(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.

```c
pat[j] matches txt[i+j]
```

```c
(pat[j] == txt[i+j])
```
All these conditions for matching are neatly handled in a while statement, whose body simply increments the pat index.

\[\text{scan left-to-right} \equiv\]
\[
\text{while } (\langle \text{pattern index is legal} \rangle \&\& \langle \text{pat[j] matches txt[i+j]} \rangle) \{
    j++;
\}
\]

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.

\[\text{pattern found in text} \equiv\]
\[(j == m)\]

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.

\[\text{shift the pattern one place in the text} \equiv\]
\[
i++;
    j = 0;
\]

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

\[\text{pattern not found in text} \equiv\]
\[\text{return false;}\]

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.

\[\text{Function List 15: Functional Brute Force Algorithm}\]

\[\text{Functional brute force pattern matching} \equiv\]
\[
\text{bf} :: \text{String} \rightarrow \text{String} \rightarrow \text{Bool}
\text{bf} [] \text{txt} = \text{True}
\text{bf} \text{pat} [] = \text{False}
\text{bf} (p:ps) (t:ts) = (p == t) \&\& (\text{bf} ps ts)
\]
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.

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

bool morrisPratt (const char *txt, const char *pat) {
    // Initialize local state
    while (text index is legal) {
        // scan left-to-right
        if (pattern found in text) return true;
        // Morris–Pratt shift
    }
    // pattern not found in text
}
```

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 \( \text{pat}[0..(j-1)] = \text{txt}[i..(i+j-1)] \) and
\( \text{pat}[j] \neq \text{txt}[i+j] \) for some \( j \). It simply restarts comparing \( \text{pat}[0] \)
with \( \text{txt}[i+1] \). Recall the configurations from the brute force exam-
ple from page 68.

Consider the initial configuration for some text and a pattern:

\[
\begin{align*}
\text{text} & \quad b \ a \ b \ c \ b \ a \ b \ c \ a \ b \ c \ a \ b \ c \\
\text{pattern} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b
\end{align*}
\]
\[
↑ \quad 1 \text{ compare}
\]
Since there a mismatch at the up-arrow, the pattern is shifted one
position right in the text to yield the next configuration:

\[
\begin{align*}
\text{text} & \quad b \ a \ b \ c \ b \ a \ b \ c \ a \ b \ c \ a \ b \ c \\
\text{pattern} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b
\end{align*}
\]
\[
↑ \quad 4 \text{ compares}
\]
Now, the up-arrow shows a mismatch in this configuration at pat-
tern 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:

\[
\begin{align*}
\text{text} & \quad b \ a \ b \ c \ b \ a \ b \ c \ a \ b \ c \ a \ b \ c \\
\text{pattern} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b
\end{align*}
\]
\[
↑ \quad 1 \text{ compare}
\]
After the immediate mismatch between \( \text{pat}[0] = a \) and \( \text{txt}[4] = b \),
the pattern is shifted one place right:

\[
\begin{align*}
\text{text} & \quad b \ a \ b \ c \ b \ a \ b \ c \ a \ b \ c \ a \ b \ c \\
\text{pattern} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b
\end{align*}
\]
\[
↑ \quad 8 \text{ compares}
\]
The mismatch occurs with the partial match abcabca and and mis-
match between \( c \) and a pattern index \( j = 7 \) and text index \( i = 12 \).
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!

\[
\begin{align*}
\text{text} & \quad b \ a \ b \ c \ b \ a \ b \ c \ a \ b \ c \ a \ b \ c \\
\text{pattern} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b
\end{align*}
\]
\[
↑ \quad 1 \text{ compares}
\]
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.

\[
\begin{align*}
\text{text} & \quad b \quad a \quad b \quad c \quad b \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad b \quad c \\
\text{pattern} & \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad c \quad a \quad b \\
& \uparrow \quad 1 \text{ compares}
\end{align*}
\]

Shift the pattern one place to find:

\[
\begin{align*}
\text{text} & \quad b \quad a \quad b \quad c \quad b \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad b \quad c \\
\text{pattern} & \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad c \quad b \\
& \uparrow \quad 8 \text{ compares}
\end{align*}
\]

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

\[
\begin{align*}
\text{text} & \quad b \quad a \quad b \quad c \quad b \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad b \quad c \\
\text{pattern} & \quad a \quad b \quad c \quad a \quad b \quad c \quad a \quad c \quad a \quad b \quad c \\
& \uparrow \quad 6 \text{ compares}
\end{align*}
\]

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

\[(1, 4, 1, 8, 1, 1, 8, 6)\]

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

\[
\text{pat}[0..(j-1)] = \text{txt}[i..(i+(j-1))]
\]

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 $\text{pat}$ cannot occur at any position in between. That is, $\text{pat}$ cannot occur in the $\text{txt}$ 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 a b c a b c a b c a b c a b c
pattern a b c a b c a c a b
```

$\text{pat}[0..2] = \text{txt}[1..3]$. There is a mismatch between $\text{pat}[3]=a$ and $\text{txt}[4]=b$. 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. $\lambda$, 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 $w = \text{abracadabra}$, has non-empty proper prefixes:

$a, ab, abr, abra, abrac, abraca, abracad, abracad, abracadab, abracadabr$

and, non-empty proper suffixes:

$a, ra, bra, abra, dabra, adabra, cadabra, acadabra, racadabra, bracadabra$

For the our sample $\text{pat}=abcabcacab$, the non-empty proper prefixes are:

$a, ab, abc, abca, abcab, abcabc, abcabca, abcabcac, abcabcaca$
and the non-empty proper suffixes are:

- b, ab, cab, acab, cab, bcab, abcab, cabcab, bcabcab

**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 \( \text{pat}=abcabca \) has proper borders \( a \) and \( abca \).

Let’s explore how to define a function \( \text{border} \) that will map a string \( w \) to the longest proper border. Interestingly, upon exploring we find \( \text{border} \) is recursive. Consider the sample \( \text{pat}=abcabca \).

\[
\begin{align*}
\text{pat} = abcabca \\
\text{border} abcabca &= abca \\
\text{border} abca &= a \\
\text{border} a &= \lambda
\end{align*}
\]

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

For instance, the periods of \( \text{pat}=abcabcacab \) are 8 and 10 since

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{pat} & b & a & c & a & b & c & a & c & a & b \\
p & 8 & a & b & a & b & a & c & a & b & a \\
p & 10 & a & b & a & b & a & c & a & b & a
\end{array}
\]

where the last period \( p = 10 \) corresponds to the empty border.

As another example, the periods of \( \text{pat}=abaabaabaaba \) are 7, 10, 12, and 13 since

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\text{pattern} & a & b & a & a & b & a & a & b & a & b & a & b & a \\
p=7 & a & b & a & a & b & a & a & b & a & b & a & b & a \\
p=10 & a & b & a & a & b & a & a & b & a & b & a & b & a \\
p=12 & a & & a & & a & & a & & a & & a & & a & & a \\
\end{array}
\]

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

\[ p + \text{length} (\text{border } w) = \text{length } w \]

The key idea is:

*Shifting a pattern by its periods aligns borders.*

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

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Borders</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>-1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>0</td>
</tr>
<tr>
<td>aba</td>
<td>1</td>
</tr>
<tr>
<td>abaa</td>
<td>1</td>
</tr>
<tr>
<td>ababa</td>
<td>2</td>
</tr>
<tr>
<td>ababa</td>
<td>3</td>
</tr>
</tbody>
</table>
length border \( \text{pat}[0..(j-1)] \) for each index \( j=1..m \) and store this length in an array \( \text{border} \). We'll see below that \( \text{border}[0] \) should be set to \(-1\). For the pattern \( \text{abaaba} \), the border array is:

\[
\text{border} = \langle -1, 0, 0, 1, 1, 2, 3 \rangle
\]

When a mismatch occurs at pat index \( j \), the pat is shifted by its period \( p=\text{border}[j] \). Here's an example:

<table>
<thead>
<tr>
<th>pat index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>txt</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>txt index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>pat index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>txt</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>txt index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

We can restart matching at pattern index \( j = \text{border}[3] = 1 \). We don't need to recheck the match at \( \text{pat}[0] \) with \( \text{txt}[3] \). 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

\[
p = j - \text{border}[j]
\]

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

\[
p = 1 = 0 - \text{border}[0]
\]

which implies \( \text{border}[0] = -1 \).

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 \( i<k<p \),

\[
\text{pat}[0..(m-1)] = \text{txt}[k..(k+m-1)]
\]

We know \( \text{pat}[0..(j-1)] = \text{txt}[i..i+(j-1)] \) and \( \text{pat}[j] \neq \text{txt}[i+j] \). But, \( \text{pat}[j] = \text{txt}[k+j] \).

For instance, if \( j=0 \), \( \text{pat}[0]=a \) and \( \text{txt}[1]=b \), then shifting the pattern one place right is safe.

Pretend \( j = 6 \), pattern[0..5] = abaaba and \( \text{txt}[1..(1+5)]=abaabc \). Shifting pat \( p=6-3 \) places right is safe.
Morris-Pratt Shift

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

\[
\text{64a} \quad \langle \text{Morris-Pratt shift 64a} \rangle \equiv \\
i = i + j - \text{pattern.border}[j]; \\
j = (0 < \text{pattern.border}[j]) \ ? \ \text{pattern.border}[j] \ : \ 0;
\]

Computation of borders

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

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

We know \( \text{border}[0] = -1 \) and so we can set this initial value. Also, we’ll store the length of the current border in a local variable \( b \).

\[
\text{64b} \quad \langle \text{Define border}[0] 64b \rangle \equiv \\
\text{border}[0] = -1;
\]

\[
\text{int } b = -1;
\]

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

\[
\text{64c} \quad \langle \text{each non-empty prefix 64c} \rangle \equiv \\
(\text{int } j = 1; \ j <= m; \ j++)
\]
Let's pretend we've computed \( b = \text{border}[j-1] \) of \( \text{pat}[0..(j-2)] \) for some \( j \geq 1 \). We want to compute \( \text{border}[j] \). Figure 2 shows that if \( \text{pat}[0..b] = \text{pat}[(j-b-1)..(j-1)] \) and \( \text{pat}[b+1] = \text{pat}[j] \), then \( \text{border}[j] = b+1 \). That is, when the character after the prefix matches the character after the right border, then the border can be extended by 1.

\[
\begin{array}{c}
\text{pattern} \\
\hline
x \ y \ \cdots \ z \ a \\
\hline
j-1
\end{array}
\]

\[ \langle \text{Extend the border by one character when next characters match} \rangle \equiv \]
\[
++b; \\
\text{border}[j] = b;
\]

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

\[
\begin{array}{c}
\text{pattern} \\
\hline
x \ y \ \cdots \ y \ a \\
\hline
b \\
\hline
j-1 \\
\hline
b
\end{array}
\]

\[
\langle \text{Reduce the border until a match is found or no border exists} \rangle \equiv \\
\text{while } ((b \geq 0) \&\& \text{pat.charAt}(b) \neq \text{pat.charAt}(j-1)) \\
\{ \\
\quad b = \text{border}[b]; \\
\}
\]
Example: Computing Borders

Consider the length 8 pattern

\[ p = \text{abaababa} \]

We want to fill out the array \( \text{border}[0..8] \) of border lengths.

<table>
<thead>
<tr>
<th>( j )</th>
<th>Prefix</th>
<th>Border</th>
<th>Border Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \lambda )</td>
<td>-</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>( \lambda )</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ab</td>
<td>( \lambda )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>abaa</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>abaab</td>
<td>ab</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>abaaba</td>
<td>aba</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>abaabab</td>
<td>ab</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>abaababa</td>
<td>aba</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

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

```java
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
\[
(n-m+1) - 1 = 2n-m
\]
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:

\[
\text{pat}_{[0..(j-1)]} = \text{txt}_{[i..i+(j-1)]}
\]
and
\[
\text{pat}_{[j]} \neq \text{txt}_{[i+j]}.
\]

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

```
⟨Knuth–Morris–Pratt pattern matching 67⟩≡
public boolean KMP (Text pattern)
{
    ⟨Initialize Knuth–Morris–Pratt local state 69c⟩
    while (⟨text index is legal 56c⟩)
    {
        ⟨scan left-to-right 57a⟩
        if (⟨pattern found in text 57b⟩) { return true;}
        ⟨Knuth–Morris–Pratt shift 69a⟩
    }
    ⟨pattern not found in text 57d⟩
}
```

Consider the initial configuration for some text and a pattern:

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

```
```
Since there is 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 b c a b c a b c a b c a b c a b 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:

```
text b a b c b a b c a b c a b c a b c a b c a b c a b c a b 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 b c a b c a b c a b c a b c a b 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 $4 - 0 = 4$ is safe and feasible, where the 0 is the length $\lambda$, the border of a.

```
text b a b c b a b c a b c a b c a b c a b c a b c a b c a b 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 b c a b c a b c a b c a b c a b 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.

69a ⟨Knuth–Morris–Pratt shift 69a⟩≡
\[
i += j - pat.strictBorder[j];
\]
\[
j = (0 < pat.strictBorder[j]) ? pat.strictBorder[j] : 0;
\]
Declare strictBorder[] and initialize its elements to -1.

69b ⟨Pattern matching instance properties 66b⟩≡
public int[] strictBorder;

69c ⟨Initialize Knuth–Morris–Pratt local state 69c⟩≡
\[
int n = strlen(txt);
int m = strlen(pat);
int i = 0, j = 0;
for (int j = 0; j < pat.length(); j++) { strictBorder[j] = -1; }
\]
To fill out the strictBorder[] array, consider what we know:
\[
\text{pat}[0..j-1]=\text{txt}[i..i+j-1] \text{ and pat}[j] /= \text{txt}[i+j]
\]
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
\[
\text{abaababa}.
\]
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.
Here's an example showing the shift that occurs on a mismatch.

<table>
<thead>
<tr>
<th>j</th>
<th>Prefix</th>
<th>Border String</th>
<th>Border Length</th>
<th>Char after Prefix</th>
<th>Char after Border</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>λ</td>
<td>-</td>
<td>-1</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>λ</td>
<td>0</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>ab</td>
<td>λ</td>
<td>0</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>a</td>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>abaa</td>
<td>a</td>
<td>1</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>abaab</td>
<td>ab</td>
<td>2</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>abaaba</td>
<td>a</td>
<td>3</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>abaabab</td>
<td>ab</td>
<td>2</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>abaababa</td>
<td>aba</td>
<td>3</td>
<td>λ</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MP Shift</th>
<th>Strict Border</th>
<th>KMP Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

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 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.

⟨Brute-force pattern matching with right-to-left scan 71a⟩≡
public boolean rightLeftBruteForce (Text pattern) {
    ⟨Initialize right-to-left local state 71b⟩
    int j = m-1;
    And decrements pat’s index j so long as the txt and pat match.

    ⟨right-to-left scan 71c⟩≡
    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.

    ⟨right-to-left brute-force shift of pattern 71d⟩≡
    ++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.
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

\[ \text{pat}[j] \neq \text{txt}[i+j] \]

and

\[ \text{pat}[(j+1)\ldots(m-1)] = \text{txt}[(i+j+1)\ldots(i+m-1)] \]

This invariant is shown in figure 4, where \( j = 6, i = 0 \) and \( m-1 = 9 \), and \( \text{pat}[7\ldots9] = \text{txt}[7\ldots9] \) and \( \text{pat}[6] \neq \text{txt}[6] \). Let's pretend, after a mismatch, pat is shifted \( s \) positions to the right where \( 1 \leq s \leq j \). This aligns \( \text{pat}[0\ldots m-1-s] \) and \( \text{pat}[s\ldots m-1] \) as shown in Figure 5. In particular, \( \text{pat}[j-s] \) aligns with \( \text{pat}[j] \) and \( \text{txt}[i+j] \).

Thus, we require that the shift \( s \) satisfy three conditions:

1. \( \text{pat}[j-s] \neq \text{pat}[j] \). If they were equal a mismatch between \( \text{pat}[j-s] \) and \( \text{txt}[i+j] \) would occur; such a shift is not feasible.

2. A border of the reversed prefix \( \text{pat}[(m-1)\ldots(j-s+1)] \) has length \( m-1-j \), that is, \( \text{pat}[(m-1)\ldots(j+1)] = \text{pat}[(m-1-s)\ldots(j+1-s)] \).

Notice condition 2 is a statement about some border of a prefix of the reversed string; this border may not be the border.
Now let’s pretend that we shift \( j+1 \leq s < m \) characters. Such a shift aligns \( \text{pat}[0..m-1-s] \) and \( \text{pat}[s..m-1] \) with \( \text{txt}[i+s..i+m-1] \), see figure 6. Such a shift satisfies the third condition:

3. \( \text{pat} \) has a border \( \text{pat}[0..m-1-s] \) of length \( m-j-1 \) or less.

Place drawing to showing shift here.

When either the first two conditions or the third condition fail to hold we can safely shift \( \text{pat} \) the maximal amount \( m \). Given the \( \text{pat} \) 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 \( \text{goodSuffix} \). Then when a mismatch occurs on pattern index \( j \), the shift \( \text{goodSuffix}[j] \) is added to \( i \) and pat index \( j \) is reset to the end of \( \text{pat} \).

\[
\begin{align*}
73a \quad \langle \text{Boyer-Moore shift of pattern } 73a \rangle &\equiv \\
& i += \text{pat}.\text{goodSuffix}[j]; \\
& j = m-1;
73b \quad \langle \text{Initialize Boyer-Moore local state } 73b \rangle &\equiv \\
& i = m - 1; \\
& j = m - 1;
\end{align*}
\]

Figure 6: Boyer-Moore shift between \( j+1 \) and \( m \) positions.
Example: Good Suffix Shift

<table>
<thead>
<tr>
<th>j</th>
<th>Suffix</th>
<th>Border Length</th>
<th>Char Before Suffix</th>
<th>Char Before Border</th>
<th>Good Suffix Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>λ</td>
<td>-1</td>
<td>a</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>0</td>
<td>b</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>ba</td>
<td>0</td>
<td>a</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>1</td>
<td>a</td>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>baba</td>
<td>1</td>
<td>b</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>ababa</td>
<td>2</td>
<td>a</td>
<td>a</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>aababa</td>
<td>3</td>
<td>b</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>baababa</td>
<td>2</td>
<td>a</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>abaababa</td>
<td>3</td>
<td>λ</td>
<td>a</td>
<td>2</td>
</tr>
</tbody>
</table>

**Computing the goodSuffix array**

We start by declaring an instance of the goodSuffix[] array. Its length will be set to m when the [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.)

```java
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.

```java
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.

\[ \text{Declarations and initializations} \]
\[
\begin{align*}
\text{int n = txt.length();} \\
\text{Text reverse = reverse();} \\
\text{computeBorders();} \\
\text{reverse.computeBorders();} \\
\text{int s;}
\end{align*}
\]

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

\[ \text{Set each shift to the maximal value} \]
\[
\begin{align*}
\text{for (int j = 0; j < n; j++)} \\
\text{goodSuffix[j] = n;}
\end{align*}
\]

**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 \( \text{pat}[m-1] \) and \( \text{txt}[i+m-1] \), a shift of 1 is appropriate, but so is a shift by the smallest value \( s \) such that \( \text{pat}[m-1-s] \neq \text{pat}[m-1] \). This is condition 1 for the case \( j=m-1 \). The requirement is that \( s \) be the smallest value satisfying \( \text{reverse}[0..s+1] = 0 \).

**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 \( \text{pat}[0] \neq \text{txt}[i] \). 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 \( m - \text{border}[m] \), is both safe and feasible. The border is a \text{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 \( \text{pat}[j] \) 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 $1 \leq s \leq j$  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 $\text{pat}[j]$. Figure 5 illustrates that two conditions must hold:

$$\text{pat}[j-s] \neq \text{pat}[j]$$  \hspace{1cm} (2)

and

$$\text{pat}[j+1-s..m-1-s] = \text{pat}[j+1..m-1].$$  \hspace{1cm} (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 $\text{pat}$, condition 3 says that prefix $\text{reverse}[0..m-j+s-2]$ 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.

```
76a ⟨Search for a safe shift between 1 and j 76a⟩
    s = 1;
    while (s <= j) {
        if ⟨Strict border condition 76b⟩ {
            ⟨Initialize border of reverse[0..(n-j-1+s)] 76c⟩
            while ⟨Border greater than tail to match 77a⟩ {
                ⟨Reset to smaller border 77b⟩
            }
            if (k == n-j-1) { // border condition satisfied
                ⟨Set goodSuffix[j] and exit while loop 77c⟩
            }
        }
        ++s;
    }
```

The strict border condition is:

```
76b ⟨Strict border condition 76b⟩≡
    (\text{txt}.\text{charAt}(j-s) \neq \text{txt}.\text{charAt}(j))
```

The prefix of reverse, whose borders we want to test, is $\text{reverse}[0..(n-j-1+s)]$. 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 $\text{pat}$ we want to match, that is, $n-1-j$.

```
76c  ⟨Initialize border of reverse[0..(n-j-1+s)] 76c⟩≡
    \text{int} \ k = \text{reverse}.\text{border}[n-j-1+s];
```
The length of the tail that has been matched when a mismatch occurs at \( j \) is \( n-1-(j+1)-1 = n-1-j \).

\[
\langle \text{Border greater than tail to match 77a}\rangle \equiv \\
(k > n-j-1)
\]

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

\[
\langle \text{Reset to smaller border 77b}\rangle \equiv \\
k = \text{reverse.border}[k];
\]

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

\[
\langle \text{Set goodSuffix}[j] \text{ and exit while loop 77c}\rangle \equiv \\
goodSuffix[j] = s; \\
s = j;
\]

Putting all of these pieces together gives the code below.

\[
\langle \text{Reset shifts when 1 \leq goodSuffix}[j] \leq j 77d\rangle \equiv \\
\text{for (int } j=1; j < n; j++) \\
\langle \text{Search for a safe shift between 1 and } j 76a\rangle
\]
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

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

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

78a \( \langle \text{Reset shifts when goodSuffix}[j] > j \rangle \equiv \)

78b \( \langle \text{Set the border length} \rangle \)

78c \( \langle \text{There is a non-empty border} \rangle \equiv \)

78d \( \langle \text{Set shift to the period of the current border} \rangle \equiv \)

78e \( \langle \text{Initialize the index where the search starts} \rangle \equiv \)

while \( \langle \text{There is a non-empty border} \rangle \) {

\( \langle \text{Set shift to the period of the current border} \rangle \equiv \)

for (int \( j = \text{start}; j < s; j++ \) ) {

\( \text{goodSuffix}[j] = (s < \text{goodSuffix}[j]) \ ? \ s : \ \text{goodSuffix}[j]; \)

System.out.println("**gs["+j+"]="+goodSuffix[j]);
}

\( \langle \text{Reset the start index} \rangle \)

\( \langle \text{Reset to smaller border} \rangle \)

}

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

78b \( \langle \text{Set the border length} \rangle \equiv \)

\( \text{int } k = \text{border}[n]; \)

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.

78c \( \langle \text{There is a non-empty border} \rangle \equiv \)

\( (k > 0) \)

With a border of length \( k \) the period to shift aligning \( \text{pat} \) borders is \( n-k \).

78d \( \langle \text{Set shift to the period of the current border} \rangle \equiv \)

\( s = n - k; \)

A placeholder \( \text{start} \) will be used to control the search over \( j \). The first time through \( \text{pat} \) index \( j \) starts at 0.

78e \( \langle \text{Initialize the index where the search starts} \rangle \equiv \)

\( \text{int start} = 0; \)
Once we've searched over a range \( \text{start} \leq j < s \), the next search can be over a range that begins with \( \text{start} = s \).

\[
\langle \text{Reset the start index} \rangle \equiv \\
\text{start} = s;
\]
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.

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.

public Text(String t)
6. Pattern Matching

```java
{  
    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.

```81a```  
```java
public int length()
{
    return txt.length();
}

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

```81b```  
```java
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.

```81c```  
```java
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() 82a)
    (Test computeBorders() 82b)
    (Test MorrisPratt() 82c)
    (Test computeStrictBorders() 82d)
    (Test KnuthMorrisPratt() 82e)
    (Test the right-to-left scan brute-force patternMatcher2() 82f)
    (Test computeGoodSuffix() 82g)
    (Test BoyerMoore() 83)
}
```
The first test will be of the brute force left-to-right scan pattern matcher.

```java
82a ⟨Test the left-to-right scan brute-force patternMatcher()⟩ ≡
   System.out.println(txt.patternMatcher(pat));
```

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

```java
82b ⟨Test computeBorders()⟩ ≡
   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.

```java
82c ⟨Test MorrisPratt()⟩ ≡
   System.out.println(txt.MorrisPratt(pat));
```

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

```java
82d ⟨Test computeStrictBorders()⟩ ≡
   // pat.computeStrictBorders();
   // for (int j = 0; j <= pat.length(); j++) {
   //   System.out.println("border[" + j + "] = " + pat.border[j]);
   // }
```

```java
82e ⟨Test KnuthMorrisPratt()⟩ ≡
   // System.out.println(txt.KnuthMorrisPratt(pat));
```

```java
82f ⟨Test the right-to-left scan brute-force patternMatcher2()⟩ ≡
   System.out.println(txt.patternMatcher2(pat));
```

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

```java
82g ⟨Test computeGoodSuffix()⟩ ≡
   pat.computeGoodSuffix();
   for (int j = 0; j < pat.length(); j++) {
     System.out.println("goodSuffix[" + j + "] = " + pat.goodSuffix[j]);
   }
```
And now our test of BoyerMoore().

\[
\text{Test BoyerMoore()} \equiv \text{System.out.println(txt.BoyerMoore(pat))};
\]
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 \texttt{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 \texttt{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

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

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 \( \text{strictBorder}[j] = \text{border}[j] \) if \( \text{pat[border[j-1]+1]} = \text{pat}[j] \), while when this inequality does not hold we set \( j = \text{border}[j] \) 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 \texttt{computeGoodSuffix()}

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

8. Write a program \texttt{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

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).

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:

“the time for this ushers ashes”

The keywords occur at positions

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>1 20 27</td>
</tr>
<tr>
<td>she</td>
<td>19 26</td>
</tr>
<tr>
<td>his</td>
<td>14</td>
</tr>
<tr>
<td>hers</td>
<td>20</td>
</tr>
</tbody>
</table>

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 \textit{goto}, \textit{failure}, and \textit{output}. The \textit{goto} function for the keywords, he, she, his, and hers is shown below. When a match is found, the \textit{goto} function maps a (state, character) pair forward to a next state. When the next character in the text does not move the search forward, the \textit{failure} function is called.

For instance, if in state 1 and the next character is neither e nor
i, fail 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.

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

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>he</td>
</tr>
<tr>
<td>5</td>
<td>{she, he}</td>
</tr>
<tr>
<td>7</td>
<td>his</td>
</tr>
<tr>
<td>9</td>
<td>hers</td>
</tr>
</tbody>
</table>

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 goto, failure, and output functions.
Listing 19: Aho-Corasick All Keywords Search

Aho-Corasick Search
```java
public void ahoCorasick(String text) {
    State state = start;
    for (int i = 0; i < text.length; i++) {
        // While goto fails follow the failure function
        if (state.goto(text.charAt(i)) == fail) {
            state = state.failure();
        } else {
            state = state.goto(text.charAt(i));
        }
        // If in an output state, print information
        if (state.hasOutput()) {
            state.output();
        }
    }
```
```
While goto fails follow the failure function
```
while (state.goto(text.charAt(i)) == fail) {
    state = state.failure();
    // If in an output state, print information
    if (state.hasOutput()) {
        state.output();
    }
}
```

Otherwise, follow the goto function
```java
state = state.goto(text.charAt(i));
```

Pretend the class State has a Boolean function that decides the if a state has output to be displayed.
```
If in an output state, print information
```
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 algorithm makes fewer than $2n$ 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

An array of keywords is passed into the goto function. The goto function is constructed and a partially computing output function. Assume output(state) is empty when state is first created. Also, assume goto(state, c)=fail if c is undefined or if state.goto(c) has not yet been defined.

\[\text{Construct goto transitions}\]
\[
\text{public void buildGoToFunction(String[] keyword) {}
\text{State start = new State();}
\text{(Enter each keyword into the goto table)}
\text{(Add self-loops to start for characters not starting keywords)}
\text{}}
\]

\[\text{Enter each keyword into the goto table}\]
\[
\text{for (int i = 0; i < keyword.length; i++) {}
\text{(Enter keyword[i])}
\text{}}
\]

\[\text{Add self-loops to start for characters not starting keywords}\]
\[
\text{(For each character c in the alphabet)}
\text{if (start.goto(c) == fail) start.goto(c) = start;}
\text{The alphabet and its implementation are not fleshed out here.}
\]

\[\text{For each character c in the alphabet}\]
\[
\text{To enter a new keyword, first follow its already matched prefix.}
\text{Then add states for its remaining suffix.}
\]

\[\text{Enter keyword[i]}\]
\[
\text{State state = start;}
\text{int j = 0}
\text{(Follow existing path prefix of keyword[i])}
\text{(Construct new path for suffix of keyword[i])}
\text{(Save keyword[i] as output)}
\]

\[\text{Follow existing path prefix of keyword[i]}\]
\[
\text{while (state.goto(keyword[i].charAt(j)) != fail) {}
\text{state = state.goto(keyword[i].charAt(j));}
\text{++j;}
\text{}}
\]
Construct new path for suffix of keyword[i] 91a\equiv
for (int k = j; k < keyword[i].length; k++) {
    State newState = new State();
    state.setGoto(keyword[i].charAt(k)) = newState;
    state = newState;
}

Save keyword[i] as output 91b\equiv
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.

Construct the failure transitions 91c\equiv
public void buildFailureTranstions() {
    Add each state of depth one to a queue 91d\equiv
    Compute failure for states of depth d from those of depth (d − 1) 92a\equiv
}

Add each state of depth one to a queue 91d\equiv
Queue queue = new Queue;
For each character c in the alphabet 90d\equiv
    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 \( \text{nextState} \). Then, follow failures from state until they end. Set \( \text{nextState} \) to fail to where the \( \text{goto} \) function moves \( c \).

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

\[
\text{Compute failure for states of depth } d \text{ from those of depth } (d - 1)
\]

\[
\text{while (queue.notEmpty()) } \{
\text{State state = queue.dequeue(); }
\text{(For each character } c \text{ in the alphabet) } \{
\text{if (state.goto(c) != fail) } \{
\text{State nextState = state.goto(c); }
\text{queue.enqueue(nextState);} 
\text{(Follow failures from state until they end) } \{
\text{nextState.setFailure(failState.goto(c));}
\text{state.saveOutput(nextState.output());}
\}\}
\}\}
\]

\[
\text{Follow failures from state until they end}
\]

\[
\text{failState = state.failure(); }
\text{while (failState.goto(c) == fail) } \{
\text{failState = failState.failure();}
\}
\]
8. Dictionaries and Hash Tables

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

**Definition 11: Dynamic Sets and Dictionaries**

- A dynamic set \( X \) 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 identity function $\text{key}[x]=x$ is the most simple hash.

Listing 20: Direct Address Operations

```java
94 ⟨Direct Address Operations 94⟩≡
95     directAddressSearch (int T[], int k) {
96         return T[k];
97     }
98
99     void directAddressInsert(int T[], int x) {
100         T[key[x]] = x;
101     }
102
103     directAddressDelete(int T[], int x) {
104         T[key[x]] = null;
105     }
```

Hash Tables

Direct addressing can waste space when the size $|U|$ is large and the set of used keys $K \subset U$ 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 $O(|K|)$. 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 $U$ of keys into slots of a hash table $T[0..m-1]$.

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$
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 |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.

---

### Listing 21: Hashing with Chaining

```plaintext
⟨Hash Table with Chaining⟩ ≡
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*
The 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 probability $P$ 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 independent 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 $\mathbb{N} = \{0, 1, 2, \ldots\}$ 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

$$h(k) = k \mod m$$

For example, if $m = 15$ and $k = 123$, then $h(123) = 123 \mod 15 = 3$ ($15 \cdot 8 + 3 = 123$).

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 \( 39 \mod 3 = 0 \)
  - String “cabd” is represented as \( 2 \cdot 4^3 + 0 \cdot 4^2 + 1 \cdot 4 + 3 = 135 \);
    which hashes \( 135 \mod 3 = 0 \)

• 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
    \[
    h(k) = k \mod 1381
    \]

*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
  \[
  A = \frac{1}{\phi} = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887 \ldots
  \]

• For example, if \( k = 123456 \), \( m = 10000 \) and \( A = 0.6180339887 \), then
  \[
  k \cdot A = 123456 \cdot 0.6180339887 = 76300.0041089472
  \]
\[ h(k) = \left\lfloor 10000 \cdot (76300.0041089472 - \left\lfloor 76300.0041089472 \right\rfloor) \right\rfloor \\
= \left\lfloor 10000 \cdot (0.0041089472) \right\rfloor \\
= \left\lfloor (41.1089472) \right\rfloor \\
= 41 \]

**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 \( \mathcal{U} \) into \( \{0, 1, \ldots, m - 1\} \).
- The collection \( \mathcal{H} \) is said to be universal if for each pair of keys \( x, y \in \mathcal{U}, x \neq y \), 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.

\[ x = (x_0, x_1, \ldots, x_r) \]

Let \( \langle A \rangle = (a_0, a_1, \ldots, a_r) \) 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 \(\alpha\) 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:
  \[
  h : \mathbb{U} \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}
  \]
- 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:

```
(Hash Search 99) ≡
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, ... , 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, ... , m – 1}

- None guarantees uniform hashing: linear and quadratic probing only produce m probe sequences and double hashing produces m²; uniform hashing requires m! probe sequences

**Linear Probing**

given an ordinary hash function h’ : U → {0, 1, ... , m – 1}, linear probing uses the hash function

\[ h(k, i) = (h'(k) + i) \mod m \] for i = 0, 1, ..., 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 initial 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

$$h(k, i) = (h_1(k) + ih_2(k)) \pmod{m}$$

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

$$h_1(k) = k \mod m$$
$$h_2(k) = 1 + (k \mod m')$$

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

$$(h(k, 0), h(k, 1), \ldots, h(k, m-1))$$

is equally likely to be any permutation of

$$(0, 1, \ldots, m-1)$$

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 a 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

\[
\frac{1}{1 - \frac{i}{m}} = \frac{m}{m - i}
\]

- 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 \( \ln i \leq H_i \leq \ln i + 1 \) we obtain

\[
\frac{1}{\alpha} (H_m - H_{m-n}) \leq \frac{1}{\alpha} (\ln m + 1 - \ln (m - n))
\]

\[
= \frac{1}{\alpha} (\ln \frac{m}{m-n} + \frac{1}{\alpha})
\]

\[
= \frac{1}{\alpha} (\ln \frac{1}{1 - \alpha} + \frac{1}{\alpha})
\]
9. Sorting

Please read Part II Sorting and Order Statistics in the textbook (Corman et al., 2009).

Ineffective Sorts

**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

\(<\), \(\leq\), \(=\), \(\geq\), \(>\), \(\neq\), \text{min}, \text{max}

are defined on the keys.

The function \text{sorted} witnesses that a list is sorted in ascending order or not.

\begin{verbatim}
Listing 23: The Sorted Decision Problem

\text{sorted} :: (Ord a) => [a] -> Bool
\text{sorted} [] = True
\text{sorted} [x] = True
\text{sorted} (x:y:ys) = (x \leq y) \&\& \text{sorted} (y:ys)
\end{verbatim}

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.

\textbf{Sentinels:} To terminate a sort, some algorithms benefit from \textit{sentinels} at an ends of records (\(A[0]\) or \(A[n+1]\)). Sentinels are typically below or above every valid value.

\textbf{Comparison sorts:} The most common sorts require comparing keys. The lower bound time complexity for comparison sorts is \(\Omega(n \lg n)\).

\textbf{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.

\textbf{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 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
- Quicksort
- Insertion Sort
- Heap Sort
- Selection Sort
- Counting Sort
- Shell Sort
- Radix Sort
- Merge 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(n^2)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$O(n)$</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$O(n)$</td>
<td>$O((n\log(n))^2)$</td>
<td>$O((n\log(n))^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(nk)$</td>
<td>$O(nk)$</td>
<td>$O(nk)$</td>
<td>$O(n + k)$</td>
</tr>
</tbody>
</table>

Bubble Sort

The heuristic is: Repeatedly pass through the records exchanging adjacent elements that are out of order. When no exchanges are needed 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. As base cases, the empty list and a singleton are sorted and need no bubbling or changes to the list. Otherwise, on a longer list \((x:xs)\), bubble the smallest value to the head.

Listing 24: Functional Bubbling

1.Every element \(x\) is \(xs\) is in \(zs\), and vice versa.
\[
(\forall x)((x \in xs) \iff (x \in zs))
\]

2. The head of \(zs\) is the smallest element in \(xs\).
\[
\text{head } zs = \text{min } xs
\]

3. If \(\text{changed} = \text{False}\) then \(xs\) is sorted. Otherwise, the last check swapped some value and cannot guarantee the list was sorted.

The time complexity of \(\text{bubble}\) can be described by the recurrence
\[
T(n) = T(n - 1) + c, \quad T(0) = c
\]

To bubble a list of size \(n\) requires bubbling a list of size \(n - 1\) plus a

Inductive and equational reasoning is useful here. Trace the executions
• \(\text{bubble } [7, 5] =\)
  – let \([5], \text{False}\) = \(\text{bubble } [5]\)
  – in if \(7 > 5\)
  – then \([5, 7], \text{True}\)
• \(\text{bubble } [2, 7, 5] =\)
  – let \([5, 7], \text{True}\) = \(\text{bubble } [7, 5]\)
  – in if \(2 > 7\)
  – else \([2, 5, 7], \text{True}\)
• \(\text{bubble } [5, 7] =\)
  – let \([7], \text{False}\) = \(\text{bubble } [7]\)
  – in if \(5 > 7\)
  – else \([5, 7], \text{False}\)

By induction, \(T(0) = c = (0 + 1)c\). And, if \(T(n - 1) = cn\), then \(T(n) = T(n - 1) + c = cn + c = c(n + 1)\).
few other constant time instructions. The solution to the equation is

\[ T(n) = c(n + 1) = O(n) \]

bubble_sort uses bubble and calls itself repeatedly until the list is unchanged.

**Listing 25: Functional Bubble Sort**

```haskell
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 \( xs \) returns \((zs, \text{False})\) where \( zs \) is the sorted list. The time complexity is \( T(n) = c(n + 1) \).

- In the worst case, bubble \( xs \) returns \((zs, \text{True})\) each of \( n - 1 \) times and \((zs, \text{False})\) on the last pass. The time cost is \( T(n) = cn(n + 1) = O(n^2) \).

Here is an imperative implementation of bubble sort.

**Listing 26: Imperative Bubble Sort**

```c
void bubblesort(int A[], int n) 
{ 
  for (int i = n; i >= 2; i--) 
  { 
    for (int j = 2; j <= i; j++) 
    { 
      if (A[j-1] > A[j]) 
        swap(A[j-1], A[j]); 
    } 
  } 
}
```

**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 compare can be calculated as

\[ \sum_{i=2}^{n} \sum_{j=2}^{i} 1 = \sum_{i=2}^{n} (i - 1) = \frac{n(n - 1)}{2} \]

- 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**

*Pretend each permutation of the keys is equally likely.*

<table>
<thead>
<tr>
<th>order</th>
<th>swaps</th>
<th>reorder</th>
<th>swaps</th>
<th>reorder</th>
<th>swaps</th>
<th>reorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 7</td>
<td>0</td>
<td>0</td>
<td>2 5 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 7 5</td>
<td>0</td>
<td>1</td>
<td>2 5 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5 2 7</td>
<td>1</td>
<td>2 5 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5 7 2</td>
<td>0</td>
<td>1</td>
<td>5 2 7</td>
<td>1</td>
<td>2 5 7</td>
<td>0</td>
</tr>
<tr>
<td>7 2 5</td>
<td>1</td>
<td>2 5 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7 5 2</td>
<td>1</td>
<td>5 7 2</td>
<td>1</td>
<td>5 2 7</td>
<td>1</td>
<td>2 5 7</td>
</tr>
</tbody>
</table>

\[ \text{P}(\text{swap}) = \frac{3}{6}, \quad \text{P}(\text{swap}) = \frac{4}{6}, \quad \text{P}(\text{swap}) = \frac{2}{6} \]

There are \((3+4+2) = 9\) swaps for the 6 cases. The average number of swaps is

\[ \frac{9}{6} = \frac{3}{2} = \frac{(3)(3 - 1)}{4} \]

- On the first pass of the outer loop \( i = n \), the if test will be True at any given \( 2 \leq j \leq i \) if and only if

\[ \max\{A[1], A[2], \ldots, A[j-1]\} > A[j] \]

- This will be True if the largest element from

\[ \{A[1], A[2], \ldots, A[j]\} \]

in any of the first \( j-1 \) of \( j \) positions

- The probability that the if test evaluates to True, during the first pass on \( i \) is

\[ \frac{j - 1}{j} = \frac{1 \ 2 \ 3 \ 4 \ \cdots \ n - 1}{n} \]
• On the next pass of the loop on $i = n-1$, the if test will be True if and only if

$$\max\{A[1], A[2], \ldots, A[j-1]\} > A[j]$$

and

$$\max\{A'[1], A'[2], \ldots, A'[j]\} > A'[j+1]$$

where the primed values refer to the original array values.

• The probability that the if test evaluates to True on this second pass is

$$\frac{j-1}{j} \cdot \frac{j}{j+1} = \frac{j-1}{j+1}$$

or

$$\frac{j-1}{j+1} = \frac{1}{3} \cdot \frac{2}{4} \cdot \frac{3}{5} \cdots \frac{n-2}{n}$$

• In general, the probability that the if test evaluates to True on the $(n - i + 1)$th pass is

$$\frac{j-1}{j+n-i}, \ i = n, \ldots, 2$$

• 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

$$\sum_{i=2}^{n} \sum_{j=2}^{n} \frac{j-1}{j+n-i}(1) = \frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \cdots + \frac{n-1}{n}$$

or

$$= \frac{1}{2} + \frac{1}{3} (1 + 2) + \frac{1}{4} (1 + 2 + 3) + \cdots + \frac{1}{n} (1 + 2 + 3 + \cdots + (n - 1))$$

$$= \frac{1}{2} + \frac{1}{3} \left( \frac{2 \cdot 3}{2} \right) + \frac{1}{4} \left( \frac{3 \cdot 4}{2} \right) + \cdots + \frac{1}{n} \left( \frac{n(n-1)}{2} \right)$$

$$= \frac{1}{2} (1 + 2 + 3 + \cdots + (n - 1))$$

$$= \frac{n(n-1)}{4}$$

**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.
Function Insertion Sort

\[ \text{insertion\_sort} :: \text{Ord} \ a \Rightarrow [a] \rightarrow [a] \]
\[ \text{insertion\_sort} \ [] = [] \]
\[ \text{insertion\_sort} \ [x] = [x] \]
\[ \text{insertion\_sort} \ (x:xs) = \text{insert} (\text{insertion\_sort} \ xs) \]
where
\[ \text{insert} \ [[] = [x] \]
\[ \text{insert} \ (y:ys) | x \leq y = x : y : ys \]
\[ \text{otherwise} = y : \text{insert} \ ys \]

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 \( i \)th step, the array \( A[0..i] \) is sorted.

Listing 28: Imperative Insertion sort

\[ \text{void InsertionSort(int A[])} \]
\[ \{ \]
\[ \text{int i, j, v;} \]
\[ \text{for (i = 2; i <= n; i++)} \}
\[ \text{v = A[i];} \]
\[ j = i; \]
\[ \text{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 \( \text{for} \) loop on \( i \) is executed \( n - 1 \) times
- There are two initial assignment in this outer loop, a \( \text{while} \) loop, and a final assignment
- The comparison in the \( \text{while} \) 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 \( \text{while} \) 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

\[
\]

and

\[
A[i - k] \leq A[i]
\]

• That is, \( A[i] \) is the \( k \)th 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 \( k \)th largest element

• Thus the average number of comparisons is

\[
\sum_{i=2}^{n} \sum_{k=1}^{i} \frac{1}{i} = \sum_{i=2}^{n} \frac{i + 1}{2} = \frac{(n + 1)(n + 2)}{4} - \frac{3}{2}
\]

• The average case complexity is \( O(n^2) \)
Example: Insertion sort operations

<table>
<thead>
<tr>
<th>i = 2</th>
<th>order</th>
<th>compares</th>
<th>reordered</th>
<th>i = 3</th>
<th>compares</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 5 7</td>
<td>1</td>
<td></td>
<td></td>
<td>0 2 5 7</td>
<td>1</td>
</tr>
<tr>
<td>0 2 7 5</td>
<td>1</td>
<td></td>
<td></td>
<td>0 2 7 5</td>
<td>2</td>
</tr>
<tr>
<td>0 5 2 7</td>
<td>2</td>
<td>0 2 5 7</td>
<td>1</td>
<td>0 5 7 2</td>
<td>3</td>
</tr>
<tr>
<td>0 5 7 2</td>
<td>1</td>
<td></td>
<td></td>
<td>0 5 7 2</td>
<td>3</td>
</tr>
<tr>
<td>0 7 2 5</td>
<td>2</td>
<td>0 2 7 5</td>
<td>2</td>
<td>0 7 5 2</td>
<td>3</td>
</tr>
<tr>
<td>0 7 5 2</td>
<td>2</td>
<td>0 5 7 2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For i = 2, there is \( \frac{3}{3} \) comparisons, on average
For i = 3, there are \( \frac{12}{6} \) comparisons, on average
The average number of comparisons, over all i, is

\[
\frac{21}{6} = \frac{7}{2} = \frac{(3 + 1)(3 + 2)}{4} - \frac{3}{2} = 5 - \frac{3}{2}
\]

Selection Sort

The idea behind selection sort is to let x be the largest value in a list xs. Place it at the tail of the list obtained by selection sorting xs with x removed.

Here is a functional implementation, the original is from Rosetta code

Listing 29: Functional Selection Sort

\[
\langle \text{Functional Selection sort } 114a \rangle \equiv
\begin{align*}
\text{select_sort} &: (\text{Ord } a) \to [a] \to [a] \\
\text{select_sort} \ [\] &= [\] \\
\text{select_sort} \ xs &= \text{let } x = \text{maximum } xs \\
&\quad \text{in } \text{select_sort} \ (\text{remove } x \ xs) ++ [x] \\
\text{where} \ \text{remove }_ - \ [\] &= [\] \\
&\quad \text{remove } a \ (x:xs) \\
&\quad \quad | \ x == a = xs \\
&\quad \quad | \ otherwise = x : \text{remove } a \ xs
\end{align*}
\]

Below is an imperative version of selection sort. It puts the smallest first rather than the largest last. For i = 1, 2, ..., n - 1, select the smallest value from A[i..n]. Place it in position i. The invariant of the algorithm is that after the ith step, the array A[1..i] is sorted.

Listing 30: Imperative selection sort

\[
\langle \text{Imperative selection sort } 114b \rangle \equiv
\begin{align*}
\text{void selectionsort}(\text{int } A[], \text{int } n) \\
\{
\end{align*}
\]
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]);
        }
    }
}

Selection Sort — Analysis of Complexity

- In all cases, selection sort makes
\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1) = \sum_{i=1}^{n-1} (n-i) = \frac{(n-1)n}{2}
\]

- Selection sort makes \(n - 1\) swaps
- Selection sort may be the best method when the records are large

Shell Sort

- An extension of insertion sort that allows exchanges of items that are far apart
- Rearrange the file so that every \(h\)th element (starting anywhere) yields a sorted file
- Use a decreasing sequence of increments \(h\) that ends at 1
- The decreasing sequence
\[
\ldots, 1093, 364, 121, 40, 13, 4, 1
\]
where \(h_k = 3h_{k-1} + 1, h_0 = 1\), works well in practice
- Shell sort never does more than \(n^{3/2}\) comparisons for the increments 1, 4, 13, 40, \ldots
Listing 31: Shell sort

```c
void ShellSort(int A[], int n)
{
    int i, j, h, v;
    for (h = 1; h < n; h = 3h + 1) {
        do {
            h = h/3;
            for (i = h + 1; i <= n; i += h) {
                v = A[i]
                j = i;
                while (j > h && A[j-h] > v) {
                    A[j] = A[j-h];
                    j = j - h;
                }
                A[j] = v;
            }
        } while (h>1);
    }
}
```

**Merge Sort**

Merge sort is straightforward:

- Split the array into two equal halves
- Merge sort each half
- Merge the sorted halves

The cost of merging is clear from the code below. If \(n + m\) is the lengths of the two lists \(xs\) and \(ys\), then

\[
T(n + m) = T(n + m - 1) + c
\]

which has solution

\[
T(n + m) = O(n + m)
\]

In **Haskell** patterns of the form \(xs@(x:xt)\) are called as-patterns. You can use \(xs\) as a name for the value being matched by \((x:xt)\).
Listing 32: Merging two sorted list

\[
\begin{align*}
\text{(Merging two list, functionally 117a)} & \equiv \\
\text{merge } [] \ ys & = ys \\
\text{merge } xs \ [] & = xs \\
\text{merge } xs@(x:xt) \ ys@(y:yt) & \\
& \mid x \leq y \quad = x : \text{merge } xt \ ys \\
& \mid \text{otherwise} \quad = y : \text{merge } xs \ yt
\end{align*}
\]

Split maps a list to a pair of lists. An empty list splits into two empty lists. A singleton splits into itself and an empty list. A list with 2 or more elements \(x:y:zs\) splits into two list: One with \(x\) as its head and another headed by \(y\). The tails of these two lists are the splits of the tail \(zs\).

Listing 33: Splitting a list

\[
\begin{align*}
\text{(Split a list 117b)} & \equiv \\
\text{split} : : [a] \to ([a], [a]) \\
\text{split } [] & = ([], []) \\
\text{split } [x] & = ([x], []) \\
\text{split } (x:y:zs) & = \text{let } (xs,ys) = \text{split } zs \text{ in } (x:xs,y:ys)
\end{align*}
\]

The cost of splitting a list, based on the cost above, is modeled by

\[ T(n) = T(n - 2) + c \]

The solution to this recurrence has big-O behavior.

\[ T(n) = O(n) \]

Listing 34: Merge sorting

\[
\text{(Split a list 117b) + } \equiv \\
\text{mergeSort } [] & = [] \\
\text{mergeSort } [x] & = [x] \\
\text{mergeSort } xs & = \text{let } (as,bs) = \text{split } xs \\
& \quad \text{in } \text{merge } (\text{mergeSort } as) \ (\text{mergeSort } bs)
\]

Here is an imperative way to implement merge sort. The Merge algorithm needs to be refactored and better explained.
Merge Sort – Analysis of Complexity

- Let \( n = \text{high} - \text{low} + 1 \) be the number of elements to be sorted

- With little loss of generality, assume \( n \) is a power of 2, \( n = 2^p \)

- Each time \text{mergesort} is called with an array of length \( n \) it makes two calls to itself with arrays of length \( n/2 \)

- The call to \text{merge} costs
- n compares of array elements (the for loop on k is executed n times)
- exactly 2n data moves.

- A formula for the time complexity is

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn + d \]

where c and d are unspecified constants

- Together with an initial condition \( T(1) = 0 \), one can verify that

\[ T(n) = cn \cdot \lg n + d(n - 1) \]

satisfies the recurrence and initial condition

**Quicksort**

- Attributed to Tony Hoare (only one of his contributions to computer science; find out who he is if you don’t already know)

- Pick an element \( p \) of the array; partition array so all values smaller than \( p \) are to \( p \)'s left and all values greater than \( p \) are to the right; recursively quicksort sub-array on either side of \( p \)

- Average and best case time complexity is \( O(n \cdot \lg n) \)

- Worst case time complexity is \( O(n^2) \)

- Usually, the fastest when compared with all know (sequential) sorting algorithms

- Does not require extra (explicit) space

- Ways to improve Quicksort
  - Remove recursion — it can consume unacceptable amounts of space for the implicit stack needed for recursion
  - Avoid small sub-arrays — switch to insertion sort when array size is small (e.g., somewhere in the range 5 to 15)
  - Avoid the worst case behavior by using a random element \( p \)

**Quicksort Algorithm**

The quicksort algorithm is attributed to Tony Hoare (Hoare, 1961). Sedgewick’s analysis of quicksort (Sedgewick, 1977) and (Sedgewick, 1978) provide an in-depth analysis and details of its implementation.

The basic quicksort idea is to place the first element \( s \) of a list in its correct position. That is, smaller elements are placed before \( s \) and larger elements after \( s \).
Here is a summary of Bentley’s code for quick sorting (Bentley, 1984b). The essence is partition an array about a pivot that gets placed in its correct position. Then quick sort the lower and upper arrays.

### Listing 36: Functional Quicksort

```hs
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (p:xs) = qsort [x|x<-xs,x<p] ++ [p] ++ qsort [x|x<-xs,x>=p]
```

### Listing 37: Imperative Quicksort

```c
void quickSort(int A[], int lo}, int hi)
{
    int pivot;
    if (hi > lo) {
        pivot = Partition(A, lo, hi);
        quickSort(A, lo, pivot-1);
        quickSort(A, pivot+1, hi);
    }
}
```

### Example: Quicksort Example

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>indexes</td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>8</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>indexes</td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>indexes</td>
<td>j</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

### Listing 38: Imperative partitioning about a pivot

```c
int partition(int A[], int lo, int hi)
{
```

```
```c
int v, i, j;
v = A[hi]; i = lo - 1; j = hi;
for (;;) {
    while (A[++i] < v) ;
    while (A[--j] > v) ;
    if (i >= j) break;
    swap (A[i], A[j]);
}
swap (A[i], A[hi]);
return i;
}
```

Let's analyze the worst, best, and average case of Quicksort.

**Quicksort: Worst Case**
- Given an array of length $n$, quicksort makes two calls to itself, once with an array of length $p$ and once with an array of length $n - p - 1$
- Here $p$ is the size of the array from `low` to `pivot-1`
- The cost of the call to `partition` is $n + 1 = O(n)$
- In the worst case $p = 0$ and

\[
T(n) = (n + 1) + T(n - 1)
\]

with initial condition $T(1) = 1$
- By mathematical induction, or unrolling the recurrence,

\[
T(n) = (n + 1) + (n) + (n - 1) + \cdots + 3 + T(1)
\]

\[
= (n + 1) + (n) + (n - 1) + \cdots + 3 + (2 + 1 - 2)
\]

\[
= (n + 1)(n + 2)/2 - 2
\]

\[
= O(n^2)
\]

**Quicksort: Best Case**
- In the best case $p = n/2$ and

\[
T(n) = (n + 1) + 2T(n/2)
\]

with initial condition $T(1) = 1$
- Note we are cheating a little here since $n - p - 1 = n/2 - 1 \neq n/2$, but this fudge will not alter the timing analysis
Unrolling the formula

\[ T(n) = (n + 1) + 2(n/2 + 1) + \cdots + 2^q T(n/2^q) \]
\[ = n \lg n + 2n - 1 \]
\[ = O(n \lg n) \]

where \( n = 2^q \) is a power of 2 and \( q = \lg n \).

**Quicksort: Average Case**

- In the average case

\[ T(n) = n + 1 + \sum_{p=0}^{n-1} (T(p) + T(n - p - 1))/n \]

- Initial condition: \( T(0) = 0 \)

- Massaging \( T(n) \) into shape:
  
  First we can rewrite

\[ T(n) = n + 1 + 2/n \sum_{p=0}^{n-1} T(p) \]

Multiply by \( n \)

\[ nT(n) = n^2 + n + 2 \sum_{p=0}^{n-1} T(p) \]

Replacing \( n \) by \( n + 1 \)

\[ (n + 1)T(n + 1) = n^2 + 3n + 2 + 2 \sum_{p=0}^{n} T(p) \]

Subtract the previous line from this

\[ (n + 1)T(n + 1) - nT(n) = 2(n + 1) + 2T(n) \]

- Now suppose we knew a function \( G(z) \) such that

\[ G(z) = T_0 + T_1 z + T_2 z^2 + \cdots \]
\[ = \sum_{n=0}^{\infty} T_n z^n \]

where we write \( T_n \) for \( T(n) \)
• Notice that

\[
G'(z) - zG'(z) = \sum_{n=0}^{\infty} nT_n z^{n-1} - z \sum_{n=0}^{\infty} nT_n z^{n-1} \\
= \sum_{n=0}^{\infty} (n+1)T_{n+1} z^{n} - \sum_{n=0}^{\infty} nT_n z^{n} \\
= \sum_{n=0}^{\infty} [(n+1)T_{n+1} - nT_n] z^{n} \\
= \sum_{n=0}^{\infty} [2(n+1) + 2T(n)] z^{n} \\
= \frac{2}{(1-z)^2} + 2G(z)
\]

• Thus

\[
G'(z) = \frac{2}{(1-z)^2} + \frac{2}{1-z}G(z)
\]

• Or, multiplying by \((1-z)^2\) and rearranging terms

\[
(1-z)^2 G'(z) - 2(1-z)G(z) = \frac{2}{1-z}
\]

• But the left-hand side above is the derivative of

\[
(1-z)^2 G(z)
\]

so integrating both sides

\[
(1-z)^2 G(z) = -2\ln(1-z) + C
\]

where \(C = 0\) since \(G(0) = T_0 = 0\)

• It follows that

\[
G(z) = \frac{-2}{(1-z)^2} \ln(1-z) \\
= 2 \sum_{i=1}^{\infty} iz^{i-1} \sum_{j=1}^{\infty} \frac{z^j}{j} \\
= 2 \sum_{n=1}^{\infty} \left[ \sum_{k=1}^{n} \frac{\binom{n}{k}}{(n-k+1)k} \right] z^n \\
= \sum_{n=1}^{\infty} [2(n+1)H_n - 2n] z^n
\]

• Therefore

\[
T(n) = 2(n+1)H_n - 2n = O(n\lg n)
\]
Heap Sort

- **Priority Queues** — a data structure where the largest, i.e., highest priority item is always first
- Priority queues need not be completely sorted
- The following operations should be efficient
  - *Construct* a priority queue from a list
  - *Find* the highest priority item
  - *Remove* the highest priority item
  - *Insert* a new item
  - *Delete* an item
- The *Heap* data structure for implementing priority queues
  - A *left-complete binary tree* is a binary tree which is completely filled at all levels except possibly the last, which is filled from left-to-right.
  - A heap is a complete binary tree that satisfies the *heap condition* that the key in each node is larger than (or equal to) the keys in its children
- A heap can be stored in an array where node $j$ has left child in position $2j$ and and right child in position $2j + 1$. The parent of $j$ is in position $\lfloor j/2 \rfloor$
- For example, the following array is a heap.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>15</td>
<td>12</td>
<td>14</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose we are given an array to *heapify*, say

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>6</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>
• We can start in the middle of the array (at position \( \lfloor n/2 \rfloor = 4 \) in this case) and exchange the larger of the two children with the parent if necessary

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 3 9 7 15 11 12 6 8 14 |

• Moving left one position, we heapify the tree at this position

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 3 9 12 15 11 7 6 8 14 |

• And continue

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 3 15 12 9 11 7 6 8 14 |

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 3 15 12 14 11 7 6 8 9 |

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 15 3 12 14 11 7 6 8 9 |

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 15 14 12 3 11 7 6 8 9 |

| position | 1 2 3 4 5 6 7 8 9 |
| value    | 15 14 12 9 11 7 6 8 3 |

• Given an array \( A \) and an index \( i \), where the binary tree rooted at \( 2i \) and \( 2i + 1 \) are heaps, we make the tree rooted at \( i \) a heap with the heapify algorithm

• Idea is to exchange the element \( A[i] \) with the largest of \( A[2i] \) and \( A[2i+1] \) (if necessary), and then heapify the changed sub-tree

• Given a sub-tree rooted at \( i \) of size \( n \), the time to fix the relationship among \( A[i], A[2i], A[2i+1] \) is \( \Omega(1) \)

• The child’s sub-tree can have size at most \( 2n/3 \), which occurs when the last row of the tree is half full

• The running time of heapify is given by \( T(n) = T(2n/3) + \Omega(1) \) so \( T(n) = O(\lg n) \)
Listing 39: Heapify

```c
(Heapify 126a) ≡
void heapify(int A[], int k, int n)
{
    int j, v = A[k];
    while (k <= eq n/2) {
        j := 2*k;
        if (j < n && A[j] < A[j+1]) {
            j = j+1;
        }
        if (v >= A[j]) break;
        a[k] = a[j]; k = j;
    }
    A[k] = v;
}
```

**Constructing a Heap**

- Since the elements $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ have no children, they are each one element heaps
- We can build a heap by running heapify on the remaining nodes
- Each call to heapify costs at most $O(\log n)$ and there are $O(n)$ calls
- Constructing a heap is at most $O(n \log n)$
- A more careful analysis shows we can build a heap in linear time ($O(n)$)

Listing 40: Build a heap

```c
(Build a heap 126b) ≡
void buildheap(int A[], int n)
{
    int i;
    for (i = floor(n/2); i >=1; i--) {
        heapify(A, i, n);
    }
}
```

**The Heap Sort Algorithm**

- The steps are
  - Build a heap
– Exchange the root of the tree with the last element of the tree
– Decrement the heap size by one
– Heapify from the root of the tree

• Building a heap is $O(n)$
• Swapping the root and the last element and decrementing heap size are $\Omega(1)$
• Heapifying from the root of the tree takes time

$$O(lg i), \quad i = n - 1, \ldots, 2$$

• The time complexity of heap sort is

$$O(n) + \sum_{i=2}^{n} [c + O(lg i)] = O(n lg n)$$

### Listing 41: Heap sort

```c
void heapSort(int A[], int n) {
    int i;
    buildHeap(A, \, n);
    for (i = n; i >= 2; i–) {
        swap(A[1], A[i]);
        n = n-1;
        heapify(A, \, 1, \, n);
    }
}
```

### Time Complexity Lower Bound for Comparison Sorts

• Comparison sorts determine the order of elements based only on comparisons between the input elements.

• Examples of comparison sorts are insertion sort, Merge sort and quicksort.

• Comparison sorts can be viewed in terms of a decision tree.

• Theorem: Any decision tree that sort $n$ elements has height $\Omega(n lg n)$.

• There are $n!$ leaves in a decision tree, so the height of the tree is at least $lg n! = \Omega(n lg n)$

• If operations other than comparisons are used, we may be able to sort faster than $O(n lg n)$

• Some sorting techniques use special properties of the input data to sort faster than $\Omega(n lg n)$.
Counting Sort

Counting sort assumes that the \( n \) values to be sorted are integer in the range \( 1 \) to \( k \) for some \( k \). When \( k = O(n) \), the counting sort runs in \( O(n) \) (linear) time. Three arrays are used:

1. Array \( \bar{A} \) is the input array
2. Array \( B \) holds the sorted output
3. Array \( C \) is temporary work space.

Here is an example: Let

\[
\bar{A} = \langle 3, 6, 4, 1, 3, 4, 1, 4 \rangle
\]

Count the values equal to \( i \) and store them in \( C \).

\[
\bar{C} = \langle 0, 2, 0, 2, 3, 0, 1 \rangle
\]

Next compute the partial sums of \( \bar{C} \), reusing its space, to find the numbers of values less than or equal to the upper index.

\[
\bar{C} = \langle 0, 2, 2, 4, 7, 7, 8 \rangle
\]

Now move the input data, starting from the last of \( A \), into \( B \).

There are no zeros, two ones, no twos, two threes, three fours, and one six.

There are no values less than or equal to zero, two values less than or equal to one and two, four values less than or equal three, seven values less than or equal four and five, and eight values less than or equal six.

### Listing 42: Counting sort

128\( a \) \( \langle \text{Counting sort 128a} \rangle \equiv \)

```c
void countingsort(int A[], int B[], int k, int n)
{
    \langle Zero out C, the temporary work space 128b \rangle
    \langle Set C[i] = count of numbers equal to i 128c \rangle
    \langle Set C[i] = count of numbers less than or equal to i 129a \rangle
    for (j = n; j >= 1; j–)
    {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
}
```

First, make certain the array \( C \) has all entries initialized to 0.

128\( b \) \( \langle \text{Zero out C, the temporary work space 128b} \rangle \equiv \)

```c
for (i = 0; i < k; i++) {
    C[i] = 0;
}
```

128\( c \) \( \langle \text{Set C[i] = count of numbers equal to i 128c} \rangle \equiv \)

```c
for (j = 0; j < n; j++) {
    C[A[j]] = C[A[j]] + 1;
}
```
\( \langle Set \ C[i] = \text{count of numbers less than or equal to } i \rangle \equiv \\
\text{for } (i = 1; i < k; i++) \{ \\
\quad C[i] = C[i] + C[i-1]; \\
\} \\
\)

The time complexity of counting sort \(O(n + k)\) where \(n\) is the length of \(\vec{A}\). Counting sort is \textit{stable}: numbers with the same value appear in the output array in the same order as they do in the input array.

\textbf{Radix Sort}

Radix sort is used by card-sorting machines, which if you may never have seen unless you are getting old, have been to computer museums, or have studied the history of technology.

The idea is to sort the input elements a numeral at a time starting with the \textit{least significant} numeral. For example suppose the list is

\[ \vec{A} = \langle 329, 457, 657, 839, 436, 720, 355 \rangle \]

Array holds the input elements each of which has \(d\) or less digits Time complexity is \(O(d n)\), which is linear for constant \(d\).

\textbf{Listing 43: Radix Sort}

\( \langle \text{Radix sort } \rangle \equiv \\
\text{void radixSort(int A[], int d)} \\
\{ \\
\quad \text{// Use counting sort on the i-th digit of each element in array A; */} \\
\} \\
\)

\textbf{Bucket (Bin) Sort}

\begin{itemize}
  \item Assume input elements uniformly distributed over some interval.
  \item Divide the interval into \(n\) equal-sized sub-intervals.
  \item For convenience assume data is uniformly distributed over \([0, 1)\).
  \item Define sub-intervals \([0, 1/n), [1/n, 2/n), \ldots, [(n - 1)/n, n)\)
  \item Array \(A\) holds input elements, array \(B\) is a list of pointers (buckets).
  \item If we can guarantee that each bucket \(B[i]\) has at most a few items on average, then bucket sort has time complexity \(O(n)\).
\end{itemize}
Listing 44: Bucket Sort

Listing 44: Bucket Sort
\[
\text{(Bucket sort)} \equiv
\]
\[
\text{void bucketsort(int A[], int n)}
\]
\[
\{ \text{int i, B[n];} \]
\[
\text{for (i = 0; i < n; i++ \{} \text{insert(A[i], B[floor(n*A[i])]);} \]
\[
\text{for (i = 0; i < n; i++) \{ } \text{insertationsort(B[], ,);} \]
\[
\text{concatenate ist B[0], B[1],..,B[n-1];} \]
\[
\}
\]

Sorting Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>(n^2)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Insertion</td>
<td>(n)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Selection</td>
<td>(n^2)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Merge</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
</tr>
<tr>
<td>Quick</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Heap</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
</tr>
<tr>
<td>Counting</td>
<td>(n + k)</td>
<td>(n + k)</td>
<td>(n + k^*)</td>
</tr>
<tr>
<td>Radix</td>
<td>(d(n + k))</td>
<td>(d(n + k))</td>
<td>(d(n + k)^{**})</td>
</tr>
<tr>
<td>Bucket</td>
<td>(n)</td>
<td>(n)</td>
<td>(n)</td>
</tr>
</tbody>
</table>

* the integers begin sorted lie in the range 1, \ldots, k
** each digit, of the \(d\) digits numbers, lies in the range 1, \ldots, k

Exercises

1. Given

\[31, 41, 59, 26, 53, 58, 97, 28, 16, 37, 42, 18, 85, 63\]

show how each of the following algorithms would sort the data: Bubble sort; Insertion sort; Selection sort; Merge sort; Quick sort; Heap sort

2. The number of comparison steps the odd-even merge sort algorithm takes to sort \(n\) numbers satisfies the recurrence

\[T(n) = 2T(n - 1) + (n - 1)2^{n-1} + 1.\]
What is the time complexity of the odd-even merge sort algorithm? (Even though there are many comparisons, the algorithm is of interest because many of the compares can be accomplished in parallel.)

3. If stability is not required in counting sort, show how to simplify the code.

4. If there is only one record for each key 1, 2, ..., n, Show that the following code correctly sorts n records

5. Given

031, 741, 259, 526, 053, 058, 697, 028, 516, 237, 442, 718, 285, 563

show how each of the following algorithms would sort the data: Counting sort; Bin sort; Radix sort.

6. How many times is the comparison in the if test made in the following nested loops? What is the average number of swaps?

```c
for (i = 0; i < n; i++) {
    for (j = n; j > i; j--) {
        if (A[j] < A[j-1]) {
            temp = A[j];
            A[j] = A[j-1];
            A[j-1] = temp;
        }
    }
}
```

7. How many additions are made in the following algorithm?

```c
for (i = 0; i <= n; i++) {
    A[i] = 1;
}
for (i = 1; i <= n-r; i++) {
    for (j = 1; j <= r; j++) {
    }
}
```

8. Consider the problem of determining whether two arrays a and b have an element in common. One possible algorithm is

```c
found = FALSE;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = TRUE;
```

- Give an analysis of the worst-case running time of the algorithm.
- Give an analysis of the best-case running time of the algorithm.
9. Consider the following graphic algorithm that produces a mind-boggling variety of pleasing displays on a computer screen.

```c
wallpaper(a, b, s, n)
int a, b, s, n;
{
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            x = a + i*s/n;
            y = b + j*s/n;
            c = floor(x*x + y*y);
            if (c is even) then plot (i, j);
        }
    }
}
```

What is the (worst case) big-O complexity of the algorithm?

10. Professor’s Howard, Fine, and Howard have proposed the following “elegant” sorting algorithm:

```c
StoogeSort(A, i, j)
char A[];
int i, j;
{
    if (A[i] > A[j])
        swap (A[i], A[j]);
    if (i + 1 >= j)
        return;
    k = floor((j - i + 1)/3);
    StoogeSort(A, i, j - k); /* First two-thirds */
    StoogeSort(A, i + k, j); /* Last two-thirds */
    StoogeSort(A, i, j - k); /* First two-thirds again */
}
```

- Give a recurrence relation for the worst-case running time of StoogeSort.
- Solve the recurrence relation to find the $\Theta$ bound on the worst-case running time.
- Do the professors deserve tenure?

11. The algorithm below is called radix exchange sort. It exchanges $m+1$-bit integers stored in array $A[0..n-1]$ so that all numbers with $b^{th}$ bit 0 come before those with $b^{th}$ bit 1 for $b = m$, $m-1$, ..., 0. The initial call is `exchange(A, 0, n-1, m)`.
a) What is the time complexity of the while \( j 
eq i \) loop?

b) What recurrence equation describes the time complexity of this algorithm in the best case?

c) What recurrence equation describes the time complexity of this algorithm in the worst case?

d) What recurrence equation describes the time complexity of this algorithm in the average case?
10. Medians and Order Statistics

Definition 12: Orders and Medians

Let \( A \) 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 \( A \). The maximum of \( A \) 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 \( 1 \leq i \leq n \).

Problem 5: The Selection Problem

Let \( n \) be a positive integer and let \( A \) be a set of \( n \) values from a totally ordered set.

Decision Problem: Is the value \( x \in A \) larger than exactly \( i - 1 \) other elements in \( A \)?

Function Problem: Let \( 1 \leq i \leq n \). Find the element \( x \in A \) that is larger than exactly \( i - 1 \) other elements in \( A \).

As an example, the elements in the set \( A \) have orders indicated below.

\[
A = \{7, 12, 5, 17, 9, 1, 14, 8, 18\}
\]

Order = \{3, 6, 2, 8, 5, 1, 7, 4, 9\}

A simple linear time algorithm that solves the minimum problem is shown below. It assumes \( A \) is represented as a 0-indexed array of \( n \) integers.

Listing 46: Imperative Minimum

```c
135 ⟨Minimum algorithm 136⟩≡
    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:

### Listing 47: Functional Maximum

```haskell
136a (Maximum algorithm 136a)≡
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
```

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.

### Randomized Selection

Using a randomized implementation of the partition function, described in the notes on Quicksort, a worst case linear time algorithm can be developed for the i-th order problem. The randomizing heuristic is to swap the head of the array with a randomly selected element.

### Listing 48: Imperative Randomized Partition

```c
136b (Randomized Partition 136b)≡
#include <stdlib.h>

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;
(Quicksort partition (never defined))
}

This 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.

A functional implementation of randomized partition might look like this.

Two functions are needed: one to partition about the head, and a second to make the head random.
**138a**  \(\text{Make the head random, then partition} \) 138a  

\[
\text{randomPartition :: Ord a => [a] \rightarrow ([a], Int)}
\]

\[
\text{randomPartition [] = ([], 0)}
\]

\[
\text{randomPartition [x] = ([x], 1)}
\]

\[
\text{randomPartition xs =}
\]

\[
\begin{align*}
\text{let (k, _)} & = \text{randomR (0, length xs)} \ (\text{mkStdGen 359353)} : : (\text{Int, StdGen}) \\
\text{in let (first, second)} & = \text{splitAt k xs} \\
\text{in partition (second ++ first)}
\end{align*}
\]

The code uses a Haskell system package `System.Random`. I find using the magic call `mkStdGen 359353` very unsatisfactory.

**138b**  \(\text{import Haskell Package for Randomness} \) 138b  

import System.Random

---

**Example: Randomize Partition**

*Given an array and index*

\[
A = \langle 7, 12, 5, 17, 9, 1, 14, 8, 18 \rangle \quad \text{and index} \quad i = 6
\]

randomized selection might work something like this:

- Pretend we randomly partition about the 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.

- Therefore, call random partition of the tail \(\langle 12, 17, 14, 18 \rangle\) and pretend 17 is randomly choosen as the pivot. This results in the sequence \(\langle 12, 17, 14, 18 \rangle\) where 17 is of order 3 in the sub-list and order \(5 + 3 = 8\) in the original list.

---

**Listing 51: Randomized Selection**

\(\text{Randomized Selection} \) 138c  

```c
#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 {
```
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:

\[
\begin{align*}
T(n) &= T(n/2) + n \\
    &= T(n/4) + n/2 + n \\
    &= T(n/8) + n/4 + n/2 + n \\
    &\vdots \\
    &= T(1) + n/2^{p-1} + \cdots + n/4 + n/2 + n \quad \text{for some } p = \log_2 n \\
    &= 2n \left(1 - \frac{1}{n}\right) \\
    &= O(n)
\end{align*}
\]

The textbook contains a detailed analysis concluding that the average case time complexity is \( O(n) \).
11. Dynamic Programming

Precomute, don’t recompute

I wish I could remember the source of this quote

Traveling Salesman Problem

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 $\varphi$ 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 $v_0 = v_n$ to close the last side. Triangles, squares, pentagons are common polygons.

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 $v_i v_j$ 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 forth.
forth across the polygon, see Figure 11.

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 6: Minimal Triangulation**

*Assume a weight function* $w(p_i, p_j, p_k)$ *defined on triangles* $\Delta = (p_i, p_j, p_k)$. *Let* $T$ *be a triangulation of a polygon.*

**Decision Problem:** Does triangulation $T$ minimize the sum of weights

$$\sum_{\Delta \in T} w(\Delta)?$$

**Function Problem:** Find optimal triangulations $T$, those that minimize the sum of weights

$$\sum_{\Delta \in T} w(\Delta)$$

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,

$$t(i, j) = \min \left\{ \sum_{\Delta \in T} w(\Delta) \right\}$$
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 $(p_{i−1}, ..., p_j)$ there must be one triangle $(p_{i−1}, p_k, p_j)$ where $i ≤ k ≤ j − 1$. 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,

$$t(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min \{t(i, k) + t(k + 1, j) + w(p_i, p_k, p_j) : i ≤ k ≤ j − 1\} & \text{otherwise} \end{cases}$$

**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](image)

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.

<table>
<thead>
<tr>
<th></th>
<th>$j = 1$</th>
<th>$j = 2$</th>
<th>$j = 3$</th>
<th></th>
<th>$j = 1$</th>
<th>$j = 2$</th>
<th>$j = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>$t(1, 1)$</td>
<td>$t(1, 2)$</td>
<td>$t(1, 3)$</td>
<td>$i = 1$</td>
<td>$0$</td>
<td>$8$</td>
<td>$10$</td>
</tr>
<tr>
<td>$i = 2$</td>
<td>$t(2, 2)$</td>
<td>$t(2, 3)$</td>
<td></td>
<td>$i = 2$</td>
<td>$0$</td>
<td>$5$</td>
<td></td>
</tr>
<tr>
<td>$i = 3$</td>
<td>$t(3, 3)$</td>
<td></td>
<td></td>
<td>$i = 3$</td>
<td></td>
<td></td>
<td>$0$</td>
</tr>
</tbody>
</table>

The values along the main diagonal $t(k, k)$ can be initialized to 0. Along the next upper diagonal the values are:

- $t(1, 2) = \min \{t(1, 1) + t(k + 1, 2) + w(p_0, p_k, p_2) : 1 ≤ k ≤ 1\}$
  - $= \min \{0 + 0 + 8\}$
  - $= 8$
- $t(2, 3) = \min \{t(2, 2) + t(k + 1, 3) + w(p_1, p_k, p_3) : 2 ≤ k ≤ 2\}$
  - $= \min \{0 + 0 + 5\}$
  - $= 5$
Lastly, the value of \( t(1, 3) \) is

\[
\begin{align*}
t(1, 3) &= \min \{ t(1, k) + t(k + 1, 3) + w(p_0, p_k, p_3) : 1 \leq k \leq 2 \} \\
&= \min \{ [t(1, 1) + t(2, 3) + w(p_0, p_1, p_3)], [t(1, 2) + t(3, 3) + w(p_0, p_2, p_3)] \} \\
&= \min \{|0 + 5 + 5|, |8 + 0 + 8|\} \\
&= 10
\end{align*}
\]

The Trace back

Not only do we wish 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 52: Optimally Triangulate a Polygon**

```java
145a ⟨Triangulate a polygon 145a⟩≡
public void triangulate(Polygon poly) {
 ⟨Initialize triangulate local state 145b⟩
 ⟨For each diagonal 145c⟩ {
  ⟨For each row 145d⟩ {
   ⟨Initialize the column index and table entry 145e⟩
   ⟨Test every vertex k between i and j 146⟩
  }
 }
}

145b ⟨Initialize triangulate local state 145b⟩≡
int n = poly.countOfVertices();
double t[n][n];
for (int i = 0; i < n; i++) { t[i][i] = 0; }

145c ⟨For each diagonal 145c⟩≡
for (int d = 2; d < n; d++)

145d ⟨For each row 145d⟩≡
for (int i = 1; i < n-d+1; i++)

145e ⟨Initialize the column index and table entry 145e⟩≡
Initialize the column index and >=
int j = i+d-1;
t[i, j] = INFINITY;
```
Test every vertex $k$ between $i$ and $j$} \equiv 
{\text{for (int } k = i; i < j; k++) { 
    \text{int } q = t[i,k] + t[k+1,j] + \text{poly.weight}(i-1, k, j); 
    \text{if (} q < t[i,j] \text{) } \{ 
        t[i, j] = q; 
        s[i, j] = k; 
    \} 
}} \\

This algorithm would be useful if its only application were 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**

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

$$\text{DNA} = \{A, C, G, T\}$$

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.

\[
\begin{array}{cccccccc}
A & L & G & O & - & R & I & T \ H \ M \\
A & L & G & E & B & R & A & - & - \\
\hline
s & i & s & d & d & d \\
\end{array}
\]

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 7: 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.

**Definition 13: Insertions, Deletions, and Substitutions**

Let \( s, t, u \) and \( v \) be strings over alphabet \( \Sigma \), and let \( a \) and \( b \) be a character in \( \Sigma \).

- **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.

\[
\begin{align*}
w_{\text{ins}}(a) &= \text{cost of inserting } a. \\
w_{\text{del}}(a) &= \text{cost of deleting } a. \\
w_{\text{sub}}(a, b) &= \text{cost of substituting } b \text{ for } a.
\end{align*}
\]

Let \( d(i, j) \) be the minimum edit distance between \( s[0, \ldots, i-1] \) and \( t[0, \ldots, j-1] \). 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

\[
d(i, j) = \begin{cases} 
  d(i-1, j-1) & \text{if } s_i = t_j \\
  \min \left\{ 
  d(i-1, j) + w_{\text{del}}(s_i) 
  , 
  d(i, j-1) + w_{\text{ins}}(t_j) 
  , 
  d(i-1, j-1) + w_{\text{sub}}(s_i, t_j) 
  \right\} & \text{otherwise}
\end{cases}
\]

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 \( t_i \).

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.
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:

$$\begin{array}{cccccccccc}
T & A & G & - & C & T & A & T & C & A \\
- & A & G & G & C & T & A & T & T & A \\
d & i & s
\end{array}$$

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 $\lambda$ 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 $\lambda$ row shows the costs for inserting AGGCTATTA into an empty string. These costs are $d(0, j) = j$ for $j = 0, \ldots, 9$.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values can be computed along diagonals. The first computed value
comes from comparing T and A.

\[
d(1, 1) = \begin{cases} 
  d(0, 0) (= 0) & \text{if } T = A \\
  \min \begin{cases} 
    d(0, 1) + 1 (= 2) \\
    d(1, 0) + 1 (= 2) \\
    d(0, 0) + 2 (= 2)
  \end{cases} & \text{otherwise}
\end{cases}
\]

\[
\begin{array}{cccccccccc}
\lambda & A & G & G & C & T & A & T & T & A \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{array}
\]

Possible edits to compute \(d(1, 1)\):

- Substitute
- Delete–Insert
- Insert–Delete

Now, compute values in the next diagonal:
\[
d(2, 1) = \begin{cases} 
d(1, 0)(=1) & \text{if } A = A \\
\min \left\{ \begin{array}{c}
d(1, 1) + 1(=3) \\
d(2, 0) + 1(=3) \\
d(1, 0) + 2(=3)
\end{array} \right. & \text{otherwise}
\end{cases}
\]

Next, compute \(d(3, 1)\), \(d(2, 2)\), and \(d(1, 3)\)
The complete edit distance table is:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

The optimal alignment has cost 4 for 1 deletion, 1 insertion, and 1 substitution.

```
TAGCATCA
-AGGCTATTA
```

This can be seen from tracing back in the array.
With these rules, edit distance defines a metric space on strings.

### Definition 14: 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 $\Sigma$. 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 and 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 53: Iterative String Edit Distance

```c
int editDist(char *s, int ls, char *t, int lt) {
    int distances[ls][lt];
    return editDist(s, ls, t, lt);
}
```
If either string 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."

153a  ⟨If either string is empty return 153a⟩≡
      if (!ls) return lt;
      if (!lt) return ls;

      Initializing the first row and column has time complexity $\Theta(n + m)$.

153b  ⟨Initialize the first row and first column 153b⟩≡
      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 $lt = m$.

153c  ⟨For every pair of characters 153c⟩≡
      for (int i = 1; i < ls; i++)
        for (int j = 1; j < lt; j++)

      Testing for a match has complexity $O(1)$.

153d  ⟨If characters match, use the previous distance 153d⟩≡
      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.

\[
\text{Otherwise, use the minimum distance}
\]

\[
\text{else}
\]

\[
\text{\{}
\text{min} = \text{distance}[i-1][j-1];
\text{if (min > distance}[i][j-1])}
\text{\{}
\text{min} = \text{distance}[i][j-1];
\text{\}}
\text{if (min > distance}[i-1][j])}
\text{\{}
\text{min} = \text{distance}[i-1][j];
\text{\}}
\text{distance}[i][j] = 1 + \text{min};
\text{\}}
\]

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 8: Matrix Chain Multiplication**

**Function Problem:** Find the way to parenthesis a matrix product

\[M_1M_2\cdots M_n\]

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

\[\vec{X} = (-1, 2, -1)\] and \(\vec{Y} = (2, 2, 2)\)

then

\[
\langle \vec{X} \cdot \vec{Y} \rangle = (-1 \cdot 2) + (2 \cdot 2) + (-1 \cdot 2) = 0
\]

In this case \((-1, 2, -1)\) and \((2, 2, 2)\) are orthogonal.
11. Dynamic Programming

Listing 54: Functional Inner Product

155a

\[ \text{Functional Inner Product} \]
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 55: Imperative Inner Product

155b

\[ \text{Imperative Inner Product} \]
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 56: Functional Inner Product

155c

\[ \text{Functional Matrix Multiplication} \]
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.

\[
M_1 \times M_2 \times M_3 \times M_4 = (M_1 \times (M_2 \times (M_3 \times M_4)))
\]
\[
= (M_1 \times ((M_2 \times M_3) \times M_4))
\]
\[
= ((M_1 \times M_2) \times (M_3 \times M_4))
\]
\[
= ((M_1 \times (M_2 \times M_3)) \times M_4)
\]
\[
= (((M_1 \times M_2) \times M_3) \times M_4)
\]

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

\[
C(3) = \frac{1}{4} \binom{6}{3} = \frac{6!}{4 \cdot 3! \cdot 3!} = 5
\]

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:

- \(M_1\) is \(10 \times 100\), \(M_2\) is \(100 \times 5\), \(M_3\) is \(5 \times 50\)

There are \(C(2) = \frac{1}{4} \binom{4}{2} = 2\) 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

\[
\vec{p} = (p_0, p_1, \ldots, p_n)
\]

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 $p_0p_kp_n$, the cost to compute the product $((M_1 \cdots M_k) \times (M_{k+1} \cdots M_n))$.

That is,

$$d(1, n) = d(1, k) + d(k + 1, n) + p_0p_kp_n$$

But, you don’t know a priori which $k$ to use, so compute them all and take a minimum

$$d(1, n) = \begin{cases} 0 & \text{if } 1 = n \\ \min_{1 \leq k < n} (d(1, k) + d(k + 1, n) + p_0p_kp_n) & \text{if } 1 < n \end{cases}$$

For intermediate products, the optimal cost is defined by the recursion

$$d(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} (d(i, k) + d(k + 1, j) + p_{i-1}p_kp_j) & \text{if } i < j \end{cases}$$

**Example: Matrix Chain Example**

*Consider the product* $M_1M_2M_3$

*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:

\[
\begin{align*}
d(1, 2) &= \min \{d(1, 1) + d(2, 2) + p[0]p[1]p[2]\} \\
&= 0 + 0 + 10 \cdot 100 \cdot 5 \\
&= 5000 \\
d(2, 3) &= \min \{d(2, 2) + d(3, 3) + p[1]p[2]p[3]\} \\
&= 0 + 0 + 100 \cdot 5 \cdot 50 \\
&= 25000
\end{align*}
\]

\[
\begin{bmatrix}
0 & 5000 & - \\
0 & 25000 \\
0 & & 
\end{bmatrix}
\]

Next:

\[
\begin{align*}
d(1, 3) &= \min \{d(1, 1) + d(2, 3) + p[0]p[1]p[3], \\
& \quad d(1, 2) + d(3, 3) + p[0]p[2]p[3]\} \\
&= \min \{25000 + 10 \cdot 100 \cdot 50, 5000 + 10 \cdot 5 \cdot 50\} \\
&= \min \{75000, 7500\}
\end{align*}
\]

\[
\begin{bmatrix}
0 & 5000 & 7500 \\
0 & 25000 \\
0 & & 
\end{bmatrix}
\]

We know that is the matrix sizes are give by the sequence

\[\mathbf{p} = \langle p_0, \ldots, p_n \rangle\]

then the final product has size \(p_0 \times p_n\).

**Listing 57: Matrix Chain Multiplication Order**

158 ⟨Matrix Chain Multiplication Order 158⟩≡
158 ⟨Initialize the main diagonal 159a⟩

158 ⟨For each sub-diagonal 159b⟩
158 { 
158 ⟨For each sub-diagonal row 159c⟩
158 { 
158 ⟨Set the column and impossible value 159d⟩
158 ⟨For each way to partition a product 159e⟩
158 { 
158 ⟨Compute the cost of this partition 160a⟩
158 } 
158 } 
158 }
First, initialize the main diagonal at a cost that is $O(n)$.

159a \(\text{Initialize the main diagonal } 159a\)\(\equiv\)
\[
\text{for(int } i = 1; i <= n; i++) \{ \\
\quad d[i, i] = 0 \\
\}
\]

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 $O(n - 1)$.

159b \(\text{For each sub-diagonal } 159b\)\(\equiv\)
\[
\text{for(int } s = 2; s <= n; s++) \\
\quad \text{When you start at subdiagonal } s, \text{ the rows go from } i = 1 \text{ to } i = (n - s + 1). \text{ Visualize it: The upper sub-diagonal starting at column 2 goes from row } 1 \text{ to } n - 1 = n - 2 + 1.
\]

159c \(\text{For each sub-diagonal row } 159c\)\(\equiv\)
\[
\text{for(int } i = 1; i <= n - s + 1; i++) \\
\quad \text{Now we want to compute the value in subdiagonal } s \text{ and row } i. \text{ This value is at column index } j = i + s - 1. \text{ Think about it: On subdiagonal 2 at row 5 the column index will be } j = 5 + 2 - 1 = 6.
\]

Set the value here to INFINITY so any computed value will be smaller. This has constant cost, but happens
\[
\sum_{s=2}^{n} \sum_{i=1}^{n-s+1} 1 = \sum_{s=2}^{n} (n - s + 1) \\
= \sum_{p=1}^{n-1} p \\
= \frac{n(n-1)}{2} \text{ times}
\]

159d \(\text{Set the column and impossible value } 159d\)\(\equiv\)
\[
\text{\quad } j = i + s - 1; \\
\text{\quad } d[i, j] = \text{INFINITY;}
\]

Now partition the matrix product $M_i \cdot M_j$ at $k$ for each $k = i$ to $j - 1$.

159e \(\text{For each way to partition a product } 159e\)\(\equiv\)
\[
\text{\quad } \text{for(int } k = i; k < j; k++) \\
\]

\(\text{If smaller cost: update minimum, save partition } 160b\)
Everything from now on occurs within three nested \texttt{for} loops. Each operation within the inner loop on $k$ has constant cost. Therefore, the overall complexity can be computed from the sum:

\[
\sum_{s=2}^{n} \sum_{i=1}^{n-s+1} k = \sum_{s=2}^{n} \sum_{i=1}^{n-s+1} (i^{s-2}) = \sum_{s=2}^{n} (s-1) = \left( n - 1 \right) \left( \frac{n(n - 1)}{2} \right)
\]

\textit{Compute the cost of this partition 160a)}

\[ q = d[i, k] + d[k + 1, j] + p[i-1]p[k]p[j]; \]

\textit{If smaller cost: update minimum, save partition 160b)}

\begin{verbatim}
if q < d[i, j]
{
    d[i, j] = q;
    s[i, j] = k;
}
\end{verbatim}
Exercises

1. Complete the edit distance table for the following pair of strings.

<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>G</th>
<th>R</th>
<th>A</th>
<th>P</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
M_0 \times M_1 \times M_2 \times M_3 = M_0 \times (M_1 \times (M_2 \times M_3)) \\
= M_0 \times ((M_1 \times M_2) \times M_3) \\
= (M_0 \times M_1) \times (M_2 \times M_3) \\
= (M_0 \times (M_1 \times M_2)) \times M_3 \\
= ((M_0 \times M_1) \times M_2) \times M_3
\]

Suppose \( M_0 \) is \( 10 \times 100 \), \( M_1 \) is \( 100 \times 5 \), \( M_2 \) is \( 5 \times 50 \), and \( M_3 \) is \( 50 \times 10 \). 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).

\[
C(i, j) = 0 \quad \forall i \geq j \\
C(i, j) = \max\{C(i, k) + C(\ell, j) + 2\} \quad (\forall i < k \leq n) \land (\forall 1 \leq \ell < j) \land (\forall 1 \leq i < j \leq n)
\]

4. What is the best way to parenthesize the product \( M_1 M_2 M_3 M_4 \) when their sizes are described by the sequence \( \vec{p} = (5, 10, 10, 100, 5) \)?
12. Greedy Algorithms

Greedy Algorithm Concept

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 9: Combinatorial Optimization

Given an instance $I$ of a function problem $P$, 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 objective function.

Here is the outline for these notes.

\[ Greedy \text{ algorithms 165} \equiv \]
\[ (\text{Tape Storage of Files 165}) \]
\[ (\text{Rational Knapsack 167a}) \]
\[ (\text{Activity Selection 170}) \]
\[ (\text{Kruskal’s Minimum Spanning Tree 172}) \]
\[ (\text{Prim’s Minimum Spanning Tree 181a}) \]
\[ (\text{Dijkstra’s Algorithm 186a}) \]

Please read Chapter 16 Greedy Algorithms in the textbook (Corman et al., 2009).
Tape Storage of Files

Okay, it is old-school, but the problem of how to store 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

$$T_{\text{avg}}(n) = \frac{1}{n} \sum_{k=1}^{n} T(k) = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[i]$$


If the files are stored in order $\langle 3, 1, 2, 4 \rangle$ then the average access time is

$$T_{\text{avg}}(n) = \frac{1}{4}(50 + (50 + 100) + (50 + 100 + 150) + (50 + 100 + 150 + 200))$$

$$= \frac{1}{4}(4 \cdot 50 + 3 \cdot 100 + 2 \cdot 150 + 200) = 250$$

You can compute that other order increase the average access time.

For example, the order $\langle 1, 2, 3, 4 \rangle$ has average cost

$$T_{\text{avg}}(n) = \frac{1}{4}(100 + (100 + 150) + (100 + 150 + 50) + (100 + 150 + 50 + 200))$$

$$= \frac{1}{4}(4 \cdot 100 + 3 \cdot 150 + 2 \cdot 50 + 200) = 287.5$$

**Problem 10**

Tape Storage Problem: Find permutation $\pi(i)$ of $i = 1, \ldots, n$ that minimizes the average cost.

$$T_{\text{min-avg}}(n) = \min_{\pi} \left\{ \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[\pi(i)]}{n} \right\}$$

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 \( \pi \) is the optimal file storage permutation, but two consecutive files are out of length order in this minimal 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 negative amount:

\[
- \frac{L[j] - L[k]}{n} < 0.
\]

But this contradict that the given order \( \pi \) was the one giving minimal average cost for file access.

\[ \langle \text{Tape Storage of Files 165} \rangle \equiv \]

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 11: Rational (Fractional) Knapsack Problem**

An instance \( I \) consist of a knapsack with capacity \( C \) and a list of \( n \) (divisible) items with associated weights

\[ w_0, w_1, \ldots, w_{n-1} \]

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

\[ \sum r_j w_j \leq C \quad \text{the weight does not exceed the capacity} \]

The objective function is to maximize the sum over the fractional values

\[ \max \sum r_k v_k \quad \text{the value is as large as possible} \]

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.

   \[ \text{Constraint: } 1 + 5 + \frac{4}{10} 10 \leq 10 \quad \text{Objective: } 2 + 20 + \frac{4}{10} 30 = 34 \]

2. **Greedy value approach:** Place all of item 0.

   \[ \text{Constraint: } 10 \leq 10 \quad \text{Objective: } 30 \]

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.

   \[ \text{Constraint: } 5 + \frac{5}{10} 10 \leq 10 \quad \text{Objective: } 20 + \frac{5}{10} 30 = 35 \]

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 been previously computed and sorted. If the ratios need to be sorted its time complexity is \(O(n \log n)\). Building a heap and using a priority queue may, in some cases, be less expensive than sorting all the ratios. The precondition is

\[ \frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq \cdots \geq \frac{v_{n-1}}{w_{n-1}} \]

<table>
<thead>
<tr>
<th>Listing 58: Rational Knapsack</th>
</tr>
</thead>
<tbody>
<tr>
<td>167a</td>
</tr>
<tr>
<td>167b</td>
</tr>
<tr>
<td>167c</td>
</tr>
<tr>
<td>167d</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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.

167b  \(\langle\text{Initialize rational knapsack local state 167b}\rangle\) ≡

\(\text{int } k = 0, V = 0, W = 0;\)
\(\text{int } r[n];\)
\(\text{for (int } j = 0; \ j < n; \ j++) \{ r[j] = 0; \}\)

The while condition is this:  

167c  \(\langle\text{While accumulated weight } \leq C \text{ and more items 167c}\rangle\) ≡

\(\text{while ((W } < C) && (k < n))\)

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 \(C\).

167d  \(\langle\text{If all of the next item can be added 167d}\rangle\) ≡

\(\text{if (W } + w[k] \leq C)\)
When all of the next item fits, update every state value.

\[ \langle \text{Update fraction, weight, value and next item} \rangle = \]
\[
\begin{align*}
    r[k] &= 1; \\
    W &= W + w[k]; \\
    V &= V + v[k]; \\
    k &= k + 1;
\end{align*}
\]

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
\[
W + r_k w_k = C
\]

\[ \langle \text{Otherwise add a fraction of the next item} \rangle = \]
\[
\text{else} \{
\begin{align*}
    r[k] &= (C-W)/w[k]; \\
    W &= C; \\
    V &= V + r[k]*v[k]; \\
    k &= k + 1;
\end{align*}
\}
\]
Activity Selection

**Problem 12: Activity Selection**

Pretend there is a set $S = \{a_0, a_1, \ldots, a_{n-1}\}$ 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, $s_i \geq f_j$ 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:

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_k$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_k$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

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 $S_{ij}$ be the set of activities $a_k$ that start after $a_i$ finishes and finish before $a_j$ starts.

$$S_{ij} = \{a_k : (f_i \leq s_k) \land (f_k \leq s_j)\}$$

$S_{01} = \emptyset$
$S_{02} = \emptyset$
$S_{03} = \{a_3\}$
$S_{04} = \emptyset$
$S_{05} = \emptyset$
$S_{06} = \emptyset$
$S_{07} = \{a_3\}$
$S_{08} = \{a_3\}$
$S_{09} = \emptyset$
$S_{0,10} = \{a_3, a_5, a_6, a_7, a_8\}$

Suppose $A_{ij}$ is maximal set of compatible activities.

Suppose $a_k \in A_{ij}$ and let

$$A_{ik} = A_{ij} \cap S_{ik} \quad \text{and} \quad A_{kj} = A_{ij} \cap S_{kj}$$
so that
\[ A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \]

That is, computing the size of \( A_{ij} \) reduces to the sub-problem computation:
\[ |A_{ij}| = |A_{ik}| + |A_{kj}| + 1 \]

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.

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 \( S_k = \{a_i : s_i \geq f_k\} \) be non-empty and let \( a_m \in S_k \) 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 \( A_k \) be a maximum size subset of compatible activities from \( S_k \). Let \( a_j \in A_k \) 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
\[ A'_k = (A_k - \{a_j\}) \cup \{a_m\} \]

Then \( A'_k \) is a set of compatible activities, \( a_m \in A'_k \) and
\[ |A'_k| = |A_k| \]

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 59: Activity Selection

170
\[
\langle Activity Selection 170\rangle \equiv
\]
set activitySelector(int *s, int *f, int n)
{
    set A = set(0);
    j = 0;
    for (i = 1; i < n; i++)
    {  

12. Greedy Algorithms

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.

![Graph with edge weights](image)

**Definition 15: 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 16: Spanning Tree**

Given a graph \( G = (V, E) \), a spanning tree for \( G \) is a tree that contains each node.

A minimal cost spanning tree \( T \) for the graph is shown below. It has cost 37.
Problem 13: Minimal Spanning Trees

*Function Problem:* Let $G = (V, E)$ be a weighted graph. Assume that $G$ 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.

**Listing 60: Kruskal Minimum Spanning Tree**

```
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)
```
12. Greedy Algorithms

```c
{
    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 $\text{findSet}(5)=11$.

- **union** function returns the union of two sets.

In Kruskal’s algorithm, the initial calls to makeSet have time complexity $O(|V|) = O(n)$. The time to sort the edges has time complexity $O(|E| \lg |E|)$. This has worst case cost $O(2n^2 \lg n)$. Recall a complete graph on $n$ vertices has $|E| = n(n - 1)/2 = \binom{n}{2}$ edges (a triangular number).

\[
|E| \lg |E| = \frac{n(n-1)}{2} \lg \left(\frac{n(n-1)}{2}\right) \\
\leq \frac{n^2}{2} \lg \left(\frac{n^2}{2}\right) \\
= n^2 \lg(n^2) - \frac{n^2}{2} \\
= 2n^2 \lg(n) - \frac{n^2}{2}
\]

**findSet and union Operations on Sets**  
One implementation of findSet and union uses an array data structure

\[S = \langle S[0], \ldots, S[n-1] \rangle\]
where $S[i] = k$ if $s_i \in S_k$. For instance, if

$$\tilde{S} = \langle S[0], \ldots, S[8] \rangle = \langle 0, 0, 2, 0, 2, 0, 5, 5 \rangle$$

then

$$s_0, s_1, s_3, s_6 \in 0$$
$$s_2, s_4, s_5 \in 2$$
$$s_7, s_8 \in 5$$

Using this representation the `findSet` operation can be implemented as:

```
Listing 61: A simple findSet function

174a  (Simple findSet 174a) ≡
    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 62: A simple union function

174b  (Simple union 174b) ≡
    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 $O(\alpha(|E| |V|))$ time where $\alpha$ is the inverse of the Ackermann function. Ackermann's function is defined by:
12. Greedy algorithms

Listing 63: Ackermann Function

\[
\langle \text{Ackermann function } 175 \rangle \equiv \\
\text{ack :: Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{ack } 0 \ n = n + 1 \\
\text{ack } m \ 0 = \text{ack } (m-1) \ 1 \\
\text{ack } m \ n = \text{ack } (m-1) \ (\text{ack } m \ (n-1))
\]

Here are the first few values. I killed the computations in row 4 when they did not complete quickly.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>125</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value of element \(A(4, 4)\) is

\[A(4, 4) = 2^{2^{2^{2^2}} - 3}\]

The inverse of Ackermann's function \(\alpha\), 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

\(\vec{S} = \langle 0, 0, 2, 0, 2, 0, 5, 5 \rangle\)

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.

```c
Better findSet 176a≡
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:

- \( i = 7; \) \( S[7] = 5; \)
- \( i = 5; \) \( S[5] = 2; \)
- \( i = 2; \) \( S[2] = 2; \)
- return 2;

The best case of this `findSet` algorithm is \( O(1) \) while the worst case is \( O(n) \).

Now, the `union` operation can be implemented as

```c
Better union 176b≡
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*

\( (0, 0, 2, 0, 2, 2, 0, 5, 5) \)

*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*

\( (0, 0, 2, 0, 2, 0, 0, 5, 5) \)

*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 64: union by Rank (A Best union?)

```c
(Best union 177)≡
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 65: findSet with Path Compression

178 ⟨Best findSet 178⟩≡

    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 \(m = |E|\) and \(n = |V|\). 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 \(O(m \log n)\) 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 \texttt{extractMin} operation removes the highest priority (closest) vertex and re-builds the heap. For each vertex \(v\), \(\text{key}[v]\) is the minimum weight of any edge connecting \(v\) to a vertex in the tree.

\(\text{key}[v] = \text{maxInt}\) 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

![Graph Image]

The (weighted) adjacency list for the graph is:

\[
\begin{array}{llll}
\text{a} & (b, 4) & (h, 8) \\
\text{b} & (a, 4) & (c, 8) & (h, 11) \\
\text{c} & (b, 8) & (d, 7) & (f, 4) & (i, 2) \\
\text{d} & (c, 7) & (e, 9) & (f, 14) \\
\text{e} & (d, 9) & (f, 10) \\
\text{f} & (c, 4) & (d, 14) & (e, 10) & (g, 2) \\
\text{g} & (f, 2) & (h, 1) & (i, 6) \\
\text{h} & (a, 8) & (b, 11) & (g, 1) & (i, 7) \\
\text{i} & (c, 2) & (g, 6) & (h, 7)
\end{array}
\]

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**

\[
\begin{array}{cccccccccc}
Q & a & b & c & d & e & f & g & h & i \\
key & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}\]

---

Extract a from Q; process (a, b) and (a, h)

\[
\begin{array}{cccccccccc}
Q & b & h & c & d & e & f & g & i \\
key & 4 & 8 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[h] = a\]

---

Extract b from Q; process (b, c) and (b, h)

\[
\begin{array}{cccccccccc}
Q & h & c & d & e & f & g & i \\
key & 8 & 8 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = b, \pi[h] = a\]

---

Extract h from Q; process (h, g) and (h, i)

\[
\begin{array}{cccccccccc}
Q & g & i & c & d & e & f \\
key & 1 & 7 & 8 & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = b, \pi[h] = a, \pi[g] = h, \pi[i] = h\]

---

Extract g from Q; process (g, i) and (g, f)

\[
\begin{array}{cccccccccc}
Q & f & i & c & d & e \\
key & 2 & 6 & 8 & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = b, \pi[h] = a, \pi[g] = h, \pi[i] = g, \pi[f] = g\]

---

Extract f from Q; process (f, c), (f, d) and (f, e)

\[
\begin{array}{cccccccccc}
Q & c & i & d & e \\
key & 4 & 6 & 10 & 14 & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = f, \pi[h] = a, \pi[g] = h, \pi[i] = g, \pi[f] = g, \pi[d] = f, \pi[e] = f\]

---

Extract c from Q; process (c, d) and (c, e)

\[
\begin{array}{cccccccccc}
Q & i & d & e \\
key & 2 & 7 & 10 & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = f, \pi[h] = a, \pi[g] = h, \pi[i] = c, \pi[f] = g, \pi[d] = d, \pi[e] = f\]

---

Extract i from Q; process no edges

\[
\begin{array}{cccccccccc}
Q & d & e \\
key & 7 & 10 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = f, \pi[h] = a, \pi[g] = h, \pi[i] = c, \pi[f] = g, \pi[d] = d, \pi[e] = f\]

---

Extract d from Q; process (d, e)

\[
\begin{array}{cccccccccc}
Q & e \\
key & 9 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[\pi[a] = \text{NIL}, \pi[b] = a, \pi[c] = f, \pi[h] = a, \pi[g] = h, \pi[i] = c, \pi[f] = g, \pi[d] = c, \pi[e] = d\]

---

Extract e from Q; process no edges
A minimal cost spanning tree is shown below and has a cost of 37

![Diagram of a minimal cost spanning tree]

**Listing 66: Prim's Algorithm**

```plaintext
(Prim’s Minimum Spanning Tree 181a) ≡
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 181b⟩
        u = extractMin(Q);
        for (each v adjacent to minimum)
        {
            if (v in Q & W(minimum, v) < key[v])
            {
                pi[v] = minimum;
                ⟨Decrease the key 182⟩
            }
        }
    }
}
```

**181b** (Extract the Minimum 181b) ≡
vertex extractMin(heap A)
{
    if (heapSize(A) < 1) { error; }
    min = A[1];
}
heapSize(A) = heapSize(A) - 1;
heapify(A, 1, heapSize(A));
return min;
}

\[\text{Decrease the key}\]
\[
\text{decreaseKey}(A, x, k) = \\
\{ \\
\text{if } (k > key[x]) \text{ } \{ \text{error; } \} \\
key[x] = k; \\
j = (\text{where } A[j] = x); \\
\text{Heapify}(A, j, \text{heapSize}(A)); \\
\}
\]

\text{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 \( \text{key}[v] \) 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?

\text{Dijkstra's Algorithm (Single source shortest path)}

\textbf{Problem 14: Single Source Shortest Path}

Let \( G = (V, E) \) be a weighted directed graph. and let \( s = 0 \) be a source or start vertex.
Decision Problem: Let $1 \leq t \leq (n - 1)$ be a target or sink vertex. Let $d$ be a positive integer. 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 $\mathcal{V}$.

Assume vertices are named $0, \ldots, (n - 1)$. Let $W[u, v]$ be a matrix of edge weights between vertices $u$ and $v$.

$$W[u, v] = \begin{cases} \text{if } v \text{ is adjacent to } u \\ \infty \text{ otherwise} \end{cases}$$

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 $\mathcal{V} - S$ 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 $\mathcal{V} - S$. 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:

<table>
<thead>
<tr>
<th>vertex</th>
<th>(b, 4)</th>
<th>(h, 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(b, 4)</td>
<td>(h, 5)</td>
</tr>
<tr>
<td>b</td>
<td>(a, 5)</td>
<td>(c, 8)</td>
</tr>
<tr>
<td>c</td>
<td>(d, 7)</td>
<td>(f, 4)</td>
</tr>
<tr>
<td>d</td>
<td>(c, 5)</td>
<td>(e, 9)</td>
</tr>
<tr>
<td>e</td>
<td>(f, 8)</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>(c, 6)</td>
<td>(e, 10)</td>
</tr>
<tr>
<td>g</td>
<td>(f, 2)</td>
<td>(i, 4)</td>
</tr>
<tr>
<td>h</td>
<td>(g, 1)</td>
<td>(i, 2)</td>
</tr>
<tr>
<td>i</td>
<td>(c, 2)</td>
<td>(g, 6)</td>
</tr>
</tbody>
</table>

Assume the source vertex is node a
Initialization Step

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Q</td>
<td>(a, b, c, d, e, f, g, h, i)</td>
<td>S = ∅</td>
<td>π[a] = NIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract a from Q; process (a, b) and (a, h)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>Q</td>
<td>(b, h, c, d, e, f, g, i)</td>
<td>S = {a}</td>
<td>π[a] = NIL, π[b] = a, π[h] = a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract b from Q; process (b, c), (b, h) and (b, i)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>(h, i, c, d, e, f, g)</td>
<td>S = {a, b}</td>
<td>π[a] = NIL, π[b] = a, π[c] = b, π[h] = a, π[i] = b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract h from Q; process (h, g) and (h, i)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Q</td>
<td>(g, i, c, d, e, f)</td>
<td>S = {a, b, h}</td>
<td>π[a] = NIL, π[b] = a, π[c] = b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract g from Q; process (g, i) and (g, f)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Q</td>
<td>(i, c, d, e)</td>
<td>S = {a, b, h, g}</td>
<td>π[a] = NIL, π[b] = a, π[c] = b, π[h] = a, π[g] = h, π[i] = h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract i from Q; process (i, g) and (i, c)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Q</td>
<td>(f, c, d, e)</td>
<td>S = {a, b, h, g, i}</td>
<td>π[a] = NIL, π[b] = a, π[c] = i, π[h] = a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extract f from Q; process (f, e), and (f, c)

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>∞</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Q</td>
<td>(c, d, e)</td>
<td>S = {a, b, h, g, i, f}</td>
<td>π[a] = NIL, π[b] = a, π[c] = i, π[h] = a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 67: Dijkstra's Single Source Shortest Path**

```plaintext
Dijkstra's Algorithm
Dijkstra (G, W, s)
{
  Initialize Dijkstra's local data
  while (Q != ())
  {
    u = extractMin(Q);
    S = union(S, u);
    for (each v adjacent to u)
    {
      Relax
    }
  }
}
```

To initialize the local data, make the following assignments.

```plaintext
Initialize Dijkstra's local data
for (each vertex v) {
  distance[v] = infinity;
  pi[v] = NULL;
  q[s] = 0;
  S = {};
  Q = buildHeap(distance, numVertices);
}
```

Relax means:

```plaintext
Relax
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 $\pi$ 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 $n$ times where $m = |E|$ with each iteration taking $O(\lg n)$ time.

Therefore, the time complexity Dijkstra's algorithm is

$$O((n + m)\lg n)$$

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

\[
\begin{align*}
\Psi(1, m) &= 2^m \\
\Psi(n, 1) &= \Psi(n - 1, 2) \\
\Psi(n, m) &= \Psi(n - 1, \Psi(n, m - 1))
\end{align*}
\]

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.
A randomized algorithm uses a probability model in its implementation.
14. 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

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 simple decision problems:

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 vertexes \( u \) 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: Give a knapsack that can hold \( C \) pounds and a list of provisions each of which has a weight and a value, is it possible to fill the knapsack this provisions weighing no more than \( C \) and having a total value greater than \( V \)?

Presburger arithmetic is an example of class of decidable problems. Presburger arithmetic is the collection of propositions \( P \) about the natural numbers \( \mathbb{N} \) that only involve addition, equality, and Boolean operations among sub-expressions. If \( P \) is such a proposition, 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 proposition \( P \), either \( P \) or \( \neg P \), but not both, can be derived from the axioms.

- Presburger arithmetic is decidable: There is an algorithm that decides whether proposition \( P \) is True or False.
On the other hand, Gödel demonstrated how to construct propositions from the Peano axioms for general arithmetic that can not be proven True or False. That is, some decision problems are undecidable!

**Turing Machines**

An algorithm can be thought of as a Turing machine for some decision problem. Informally, a Turing machine uses a transition function \( \delta \) to map the state of the machine and the character read to a state, a character, and a direction to move the read write head.

The next state is a machine state in \( k \in K \), or one of three special states: answers y “yes” and n, “no,” or the “halt” state h. The read/write head can move ← “left”, → “right,” or — “stay.”

There are many ways to define a Turing machine. Here is Papadimitriou’s (Papadimitriou, 1994) definition.

**Definition 17: Turing Machine**

A Turing machine is a 4-tuple \( M = (K, \Sigma, \delta, s) \) where:

1. \( K \) is a finite set of states
2. \( s \in K \) is the initial (start) state
3. \( \Sigma \) is an alphabet (a finite set of symbols (characters). \( \Sigma \) contains two special symbols: ⊥ and ⊥, called blank and first, respectively.
4. \( \delta \) is a transition function. It maps a (state, character) pair to a triple (next state, character, direction).

\[
\delta : (K, \Sigma) \rightarrow (K \cup \{h, y, n\}, \Sigma, \{←, →, −\})
\]

**The Universal Turing Machine**

One important aspect of Turing machines is that there is a universal Turing machine, called \( U \). Turing showed (Turing, 1936) is possible to invent a single machine that can simulate any other Turing machine. The input to the universal machine \( U \) is a pair \( (M, x) \). The universal machine \( U \) computes \( M(x) \), that is \( U(M, x) = M(x) \).

The existence of a universal machine leads to undecidable problems, the most famous of which is the halting problem.

**Problem 15: 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. The halting language \( H \) is the set of all (machine, input) pairs such that \( M \)
halts on $x$.

$\mathcal{H} = \{(M, x) : M(x) \neq \uparrow\}$

There is no Turing machine that decides whether or not $(M, x) \in \mathcal{H}$ for all pairs $(M, x)$. The proof is by contradiction.

Consider the thought experiment of executing the pseudo-code below.

```pseudo
diagonalMachine(machine M) {
  if (M(M) halts) then { Loop forever }
  else halt;
}
diagonalMachine(diagonalMachine);
```

The traditional contradiction proof goes like this:

**Proof: The Halting Problem is Undecidable**

_Pretend there is a Turing machine $M_{\mathcal{H}}$ that decides the halting problem._

```
(M, x)  M_{\mathcal{H}}
```

Use $M_{\mathcal{H}}$ to construct a Turing machine $D$ that accepts the encoding of a Turing machine $M$ and runs $M_{\mathcal{H}}$ on $(M, M)$. The behavior of $D$ is this:

1. $D$ does not halt if $M$ halts on $M$. 

The symbol $\uparrow$ stands for “does not halt.”
If \((M(M) \neq \lambda)\), then \(D(M) = \lambda\).

2. \(D\) halts if \(M\) does not halt on \(M\) \((M(M) = \lambda)\).

If \((M(M) = \lambda)\), then \(D(M) \neq \lambda\)

Consider \(D(D)\)

1. \(D\) does not halt (on input \(D\)) if \(D\) halts on \(D\)

If \((D(D) \neq \lambda)\), then \(D(D) = \lambda\).

2. \(D\) halts (on input \(D\)), if \(D\) does not halt on \(D\)

If \((D(D) = \lambda)\), then \(D(D) \neq \lambda\)

This contradiction implies that the halting machine \(M_H\) cannot exist.

**Determinism versus Non-Determinism**

By default, Turing machines are deterministic: Their transition functions \(\delta\) are functions. When transitions are relaxed to be relations, the machine is said to be non-deterministic.

**Definition 18: The \(P\) and \(NP\) Complexity Classes**

The complexity class \(P\) is the set of all deterministic Turing machines that decide their problem instances in polynomial time \(O(n^k)\), where \(n\) is the size of the instance and \(k\) is a fixed natural number.

The complexity class \(NP\) is the set of all non-deterministic Turing machines that decide their problem instances in polynomial time \(O(n^k)\), where \(n\) is the size of the instance and \(k\) is a fixed natural number.
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 $\pi$ billion seconds per century.

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), can check the answer in polynomial time. Cook in his seminal paper (Cook, 1971) clearly described these ideas and their implications.

**Problem 16: Satisfiability**

*Decision Problem: Given a Boolean expression of $n$ literals in conjunctive normal form, does it have a truth assignment?*

Cook describes the *satisfiability* problem, which is clearly in $NP$ but is not known to be in $P$. The non-deterministic algorithm guesses the satisfying truth assignment for $\phi$ and checks that it satisfies each clause in $\phi$. 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 17: P versus NP**

*Decision Problem: Does $P = NP$?*

Most people who understand the $P$ versus $NP$ problem believe the answer is “no,” but, at this time, no one knows for certain.

**Reductions**

A classic problem solving technique is to reduce a new problem to an already solved problem. Consider *validity*: Is a Boolean expression always True.

**Definition 19: Validity of a Boolean Expression**

*A Boolean expression is valid if it is True for every assignment of True or False to its variables.*

Validity can be reduced to satisfiability. To show that Boolean expression $\phi$ valid, show that $\neg \phi$ is not satisfiable. Pretend $\neg \phi$ has no satisfying truth assignment: $\neg \phi$ is always False. Therefore, $\phi$ is always True.

**NP-Completeness**

The book by Garey and Johnson (Garey and Johnson, 1979) is the classic textbook on NP-completeness. A surprising number of prob-
lems have been shown to be NP-complete.

<table>
<thead>
<tr>
<th>Definition 20: NP-Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>A decision problem ( C ) is NP-complete if:</td>
</tr>
<tr>
<td>1. ( C \in \text{NP} ), and</td>
</tr>
<tr>
<td>2. Every problem in NP is reducible to ( C ) in polynomial time.</td>
</tr>
</tbody>
</table>

1. Satisfiability: Boolean expression \( \phi \) has a truth assignment.

**Exercises**

1. Consider the virus decision problem: Given a program \( P \), does \( P \) contain a virus? Show that the virus problem is undecidable.
Projects

Project 1: Algorithms for Maximum Subsequence Sum

The maximum subsequence sum problem was described in class. Below are algorithms that solve it; they were discussed in class and presented in the handouts.

Listing 1: Max Subsequence Sum Algorithm – Cubic $O(N^3)$

```c
int maxSubseqSum(int X[], int N) {
    int MaxSoFar = 0; /* Initialize default maximum sum */
    for (int L = 0; L < N; L++) { /* for every lower limit */
        for (int U = L; U < N; U++) { /* for every upper limit */
            int Sum = 0; /* Initialize current sum */
            for (int K = L; K <= U; K++) { /* Compute X[L]+...+X[U] */
                Sum = Sum + X[K];
                MaxSoFar = (Sum > MaxSoFar) ? Sum : MaxSoFar;
            }
        }
    }
    return MaxSoFar;
}
```

Listing 2: Max Subsequence Sum Algorithm – Linear time $O(N)$, Constant space $O(1)$

```c
int maxSubseqSum(int X[], int N) {
    int MaxSoFar = 0; /* Initialize default maximum sum */
    double MaxEndingHere = 0.0
    for (int I = 1; i < N; i++) {
        MaxEndingHere = max(0.0, MaxEndingHere + X[I])
        MaxSoFar = max(MaxSoFar, MaxEndingHere)
    }
    return MaxSoFar;
}
```

Listing 3: Functional Max Subsequence Sum Algorithm

```haskell
maxsubseq = snd . foldl f ((0,[0]),(0,[])) where
  f ((h1,h2),sofar) x = (a,b) where
    a = max (0,[0]) (h1 + x, h2 ++ [x])
    b = maxsofar a
```

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.

To Do List

These tasks test your ability to: (1) construct software, (2) analyze algorithms experimentally, and (3) communicate what you have done.

1. Make a simple change to the algorithm in Listing 1 turning it into an algorithm with quadratic time complexity.

2. Compile the algorithms in Listings 1–3 and your quadratic algorithm into executable programs.

3. Test your programs on randomly generated sequences of size $n = 100, 1000, \ldots, 10^m$ for some appropriate $m$ that you choose.

4. Collect statistics on your runs: number of additions and comparisons; running time in seconds, memory usage, etc.

5. What do your statistics show about the time and space complexity of the algorithms?

6. Write a report on what you have done. Document any and all sources used.

7. Deliver your code and report using the department's submit server.

Project 2: Student’s Choice

By Monday, October 3, 2016 inform 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 of Boyer-Moore algorithms
- MapReduce
- 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:**

<table>
<thead>
<tr>
<th>Category</th>
<th>Beginning 1</th>
<th>Developing 2</th>
<th>Accomplished 3</th>
<th>Exemplary 4</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation &amp; Tests</td>
<td>The submitted code does not compile.</td>
<td>The code compiles but passes too few test cases.</td>
<td>The code compiles but passes most test cases.</td>
<td>The code compiles but passes all test cases.</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td>The code has no or very little documentation.</td>
<td>The code has some documentation as an apparent afterthought.</td>
<td>The code contains useful documentation.</td>
<td>The code is well documented, perhaps even in a literate style.</td>
<td></td>
</tr>
<tr>
<td>Report Writing</td>
<td>Too few group members speak and can be understood.</td>
<td>Some group members speak clearly and are easy to understand.</td>
<td>Most group members speak clearly and are easy to understand.</td>
<td>The report describes the problem, algorithms that solve the problem, and analyzes results of experiments execution the program on various data sets.</td>
<td></td>
</tr>
<tr>
<td>Tool Usage</td>
<td>Little evidence that software tools were used.</td>
<td>Some evidence of minimal tool usage.</td>
<td>Evidence that several software tools were used.</td>
<td>Documented use of a wide array of tools: Software configuration management tools, build tools, testing tools, debuggers, profilers, etc.</td>
<td></td>
</tr>
</tbody>
</table>

**Individual Project Average of Scores**
This is a call for participation in Algorithmics 2016, a workshop on algorithms that runs from November 14 to 28, 2016. 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.

**Deadlines**

Keep track of the course calendar. I am absent-minded and may not remind you.

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.


• 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 2016 workshop 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: ____________________________
<table>
<thead>
<tr>
<th>Category</th>
<th>Beginning 1</th>
<th>Developing 2</th>
<th>Accomplished 3</th>
<th>Exemplary 4</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict</td>
<td>Participated in regular conflict that interfered with group progress. The conflict was discussed outside of the group.</td>
<td>Was the source of conflict within the group. The group sought assistance in resolution from the instructor.</td>
<td>Was minimally involved in either starting or solving conflicts.</td>
<td>Worked to minimize conflict and was effective at solving personal issues within the group.</td>
<td></td>
</tr>
<tr>
<td>Assistance</td>
<td>Contributions were insignificant or nonexistent.</td>
<td>Contributed some toward the project.</td>
<td>Contributed significantly but other members clearly contributed more.</td>
<td>Completed an equal share of work and strives to maintain equity throughout the project.</td>
<td></td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Work performed was ineffective and mostly useless toward the final project.</td>
<td>Work performed was incomplete and contributions were less than expected.</td>
<td>Work performed was useful and contributed to the final project.</td>
<td>Work performed was very useful and contributed significantly to the final project.</td>
<td></td>
</tr>
<tr>
<td>Attitude</td>
<td>Rarely had a positive attitude toward the group and project.</td>
<td>Usually had a positive attitude toward the group and project.</td>
<td>Often had a positive attitude toward the group and the project.</td>
<td>Always had a positive attitude toward the group and the project.</td>
<td></td>
</tr>
<tr>
<td>Attendance &amp; Readiness</td>
<td>Rarely attended group meetings, rarely brought needed materials, and was rarely ready to work.</td>
<td>Sometimes attended group meetings, sometimes brought needed materials, and was sometimes ready to work.</td>
<td>Almost always attended group meetings, brought needed materials, was ready to work.</td>
<td>Always attended group meetings, always brought needed materials, and was always ready to work.</td>
<td></td>
</tr>
<tr>
<td>Task Focus</td>
<td>Rarely focused on the task and what needed to be done. Let others do the work.</td>
<td>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.</td>
<td>Focused on the task and what needed to be done most of the time. Other group members could count on this person.</td>
<td>Consistently stayed focused on the task and what needed to be done. Other group members could count on this person all of the time.</td>
<td></td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>Category</th>
<th>Beginning 1</th>
<th>Developing 2</th>
<th>Accomplished 3</th>
<th>Exemplary 4</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participation</td>
<td>Too few group members participate.</td>
<td>Some group members participate.</td>
<td>Most group members participate.</td>
<td>All group members participate equally.</td>
<td></td>
</tr>
<tr>
<td>Presence</td>
<td>Most group members do not make eye contact and have poor body language.</td>
<td>Some group members do not make eye contact or have poor body language.</td>
<td>Most group members make eye contact and have good body language.</td>
<td>All group members do make eye contact and have good body language.</td>
<td></td>
</tr>
<tr>
<td>Delivery</td>
<td>Too few group members speak and can be understood.</td>
<td>Some group members speak clearly and are easy to understand.</td>
<td>Most group members speak clearly and are easy to understand.</td>
<td>All group members speak clearly and are easy to understand.</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>Information is disorganized.</td>
<td>Information may be only partially organized.</td>
<td>Most information is presented in an organized way.</td>
<td>All information is presented in an organized way.</td>
<td></td>
</tr>
<tr>
<td>Visual Aids</td>
<td>Presentation is incomplete and disorganized.</td>
<td>Presentation is complete but disorganized.</td>
<td>Presentation is organized but incomplete.</td>
<td>Presentation is visually organized and complete.</td>
<td></td>
</tr>
</tbody>
</table>

**Group Presentation Average of Scores**

Modified from: Read, Write, Think


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 186], [page 191], [page 198], [page 201]


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
  Ackermann, 174
  Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstadter, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabriel, 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 Convergence Test, 50
  Sample Code Header, 31
  Second of a pair, 36
  Selection Sort, 114
  Sorted Decision Problem, 106