CSE 5211 Analysis of Algorithm Notes
William David Shoaff
Monday 1st January, 2018

Preliminaries

• Read and understand the syllabus

• Edward Tufte recommends this structure for presentations:
  – Provide audience 1 to 2 pages of notes
  – Allow audience 10 to 20 minutes to read notes
  – Glare at those who are not reading (his wry sense of humor?)
  – While fleshing-out notes ask and answer questions

• I’m going to change Tufte’s formula a bit.
  – I’ll provide a few pages of notes before class
  – Audience will read the notes before class
  – Audience will develop a list of questions
  – Audience will bring notes and questions to class

Contents

Preliminaries 1
Personal Assignment 2
Team Assignment 8
Questions and Problems 10
Algorithms 17
Algorithm Design 20
Mathematical Tools 25
More Mathematical Tools 28
Representative Problems 30
Divide and Conquer 35
The Quicksort Algorithm 38
Numerical Analysis 54
Greedy Algorithms 59
Dynamic Programming 67
Graph Algorithms 73
Backtracking Algorithms 76
Complexity 90
Computability 98
**Personal Assignment**

One quarter of a student’s course grade is based on personal performance. The basic chores are:

- Write code for a well-known, simple problem. Typically, the code at most a few dozen lines.
- Test the code on a range of input sizes, collecting and plotting the running times.
- Use the test data to find a function that well approximates the data.
- Analyze the code’s syntax to develop a mathematical model of the algorithm’s time complexity: A big-$O$ analysis.

This term’s personal software development and analysis project is on Gaussian elimination.

**Gaussian Elimination**

Gaussian elimination is a method for computing the solution vector $\vec{x}$ of a given system of linear equations.

$$A\vec{x} = \vec{b}$$

First, a reduction step transforms the augmented matrix $[A|\vec{b}]$ into upper-triangular form. Then, a second solving step produces the solution $\vec{x}$ by back-substitution.

Let’s get on with the code. Some types and helper functions will be needed. The reduction and solver steps must be constructed. And, a main routine is needed as an entry point. Here is the structure of the code. It is written in noweb style.

```
2a ⟨Gauss 2a⟩≡
  ⟨Row, Column, and Matrix Types 2b⟩
  ⟨IO conversion functions 4d⟩
  ⟨The Gaussian reduction step 3a⟩
  ⟨The Gaussian solver step 6⟩
  ⟨Main module 4c⟩

Define the Row and Column types to be lists of double precision floating point numbers. Define the RMatrix type to be a list indexed by rows. Define the CMatrix type to be a list indexed by columns.
```

```
2b ⟨Row, Column, and Matrix Types 2b⟩≡
type Row = [Double]
type Column = [Double]
type RMatrix = [Row]
type CMatrix = [Column]
```

1 Thanks to Lucky’s Notes for the structure of the code.
The Reduction Step

The reduction process used in Gaussian elimination transforms a matrix into upper-triangular form. Pictorially, for a small example.

\[
\begin{bmatrix}
\text{P} & \cdot & \cdot & \cdot & | & \cdot \\
\cdot & \cdot & \cdot & | & \cdot \\
\cdot & \cdot & \cdot & | & \cdot \\
\cdot & \cdot & \cdot & | & \cdot \\
\end{bmatrix} \mapsto \begin{bmatrix}
1 & \cdot & \cdot & \cdot & | & \cdot \\
0 & \cdot & \cdot & | & \cdot \\
0 & \cdot & \cdot & | & \cdot \\
0 & \cdot & \cdot & | & \cdot \\
\end{bmatrix} \mapsto \begin{bmatrix}
1 & \cdot & \cdot & \cdot & | & \cdot \\
0 & 1 & \cdot & | & \cdot \\
0 & 0 & \cdot & | & \cdot \\
0 & 0 & 0 & 1 & | \\
\end{bmatrix}
\]

To reduce a matrix to upper-triangular form repeat these steps iteratively across all rows of the matrix. For the first row:

1. Assume \( p \), the pivot, is not 0 and normalize the row by scaling it by \( 1/p \).
2. Repeatedly, multiply the normalized row by \( a \), \( b \), \( c \) and subtract the result row from second, third, and fourth rows, respectively.

\[\text{(The Gaussian reduction step)\quad 3a}\equiv\]

\[
\text{gaussianReduce} :: \text{RMatrix} \rightarrow \text{RMatrix}
\]

\[
\text{gaussianReduce} \text{ matrix} = \text{foldl reducedrow matrix [0..length matrix-2]} \text{ where}
\]

\[
\text{reducedrow} :: \text{RMatrix} \rightarrow \text{Int} \rightarrow \text{RMatrix}
\]

\[
\text{reducedrow} \text{ m} \text{ r} = \text{let}
\]

\[\text{(Pick the row to reduce by, its pivot, and normalize this pivot row)\quad 3b}\equiv\]

\[
\text{row} = \text{m} \text{ !} \text{!} \text{r}
\]

\[
\text{p} = \text{row} \text{ !} \text{!} \text{r}
\]

\[
\text{row'} = \text{map} (\lambda x \rightarrow x/\text{p}) \text{ row}
\]

\[\text{(Construct a function that reduces other rows)\quad 3c}\equiv\]

\[
\text{reduceonerow} \text{ nrow} = \text{let} \text{ nr} = \text{nrow} \text{ !} \text{!} \text{r} \text{ in} \text{ zipWith} (\lambda a b \rightarrow \text{nr} \cdot a - b) \text{ row'} \text{ nrow}
\]

3a (The Gaussian reduction step 3a)≡

\[
\text{gaussianReduce} :: \text{RMatrix} \rightarrow \text{RMatrix}
\]

\[
\text{gaussianReduce} \text{ matrix} = \text{foldl reducedrow matrix [0..length matrix-2]} \text{ where}
\]

\[
\text{reducedrow} :: \text{RMatrix} \rightarrow \text{Int} \rightarrow \text{RMatrix}
\]

\[
\text{reducedrow} \text{ m} \text{ r} = \text{let}
\]

\[\text{(Pick the row to reduce by, its pivot, and normalize this pivot row)\quad 3b}\equiv\]

\[
\text{row} = \text{m} \text{ !} \text{!} \text{r}
\]

\[
\text{p} = \text{row} \text{ !} \text{!} \text{r}
\]

\[
\text{row'} = \text{map} (\lambda x \rightarrow x/\text{p}) \text{ row}
\]

\[\text{(Construct a function that reduces other rows)\quad 3c}\equiv\]

\[
\text{reduceonerow} \text{ nrow} = \text{let} \text{ nr} = \text{nrow} \text{ !} \text{!} \text{r} \text{ in} \text{ zipWith} (\lambda a b \rightarrow \text{nr} \cdot a - b) \text{ row'} \text{ nrow}
\]

• Pick out the \( r \)-th row of matrix \( m \), using \(!\), Haskell’s indexing operator.
• Pick out the \( r \)-th value in the \( r \)-th row and call it \( p \), the pivot. To keep things simple, assume these pivot elements along the main diagonal never equal 0.
• Normalize the row by dividing each element in it by the pivot \( p \). The Haskell idiom that does this is to map the anonymous function \( x \mapsto x/p \) across the row. Call the normalized row \( \text{row’} \).

3b (Pick the row to reduce by, its pivot, and normalize this pivot row)\quad 3b)≡

\[
\text{row} = \text{m} \text{ !} \text{!} \text{r}
\]

\[
\text{p} = \text{row} \text{ !} \text{!} \text{r}
\]

\[
\text{row'} = \text{map} (\lambda x \rightarrow x/\text{p}) \text{ row}
\]

Now, to reduce another row, say \( \text{nr} \), by \( \text{row’} \) apply the function \( \text{nr} \cdot a - b \) (where \( \text{nr} \) is the \( r \)-th element in \( \text{nr} \))

to each entry \( a \) in \( \text{row} \) and \( b \) in \( \text{nr} \). The Haskell idiom for this is to \text{zip} the function with the values in \( \text{row’} \) and \( \text{nr} \).

3c (Construct a function that reduces other rows)\quad 3c)≡

\[
\text{reduceonerow} \text{ nrow} = \text{let} \text{ nr} = \text{nrow} \text{ !} \text{!} \text{r} \text{ in} \text{ zipWith} (\lambda a b \rightarrow \text{nr} \cdot a - b) \text{ row'} \text{ nrow}
\]
Next, map \texttt{reduceRow} across all rows in matrix below the pivot row.

\begin{itemize}
\item \textbf{Apply the reduction function to rows below the pivot 4a}\end{itemize}
\begin{itemize}
\item \texttt{nextrows = map reduceOneRow (drop (r+1) m)}
\end{itemize}

And finally, piece the results back together: Concatenate the first \(r\) rows from \(m\), row \(row'\) and the reduces \texttt{nextrows}.

\begin{itemize}
\item \textbf{Piece the matrix back together 4b}\end{itemize}
\begin{itemize}
\item \texttt{in take r m ++ [row'] ++ nextrows}
\end{itemize}

To turn \texttt{gaussianReduce} into a standalone program, a main module, an \textit{entry} point, must be established. IO can be tricky. The structure of the input must be known, and that must be translated into the structure of the Haskell function the implements the program.

Let’s agree, for the purpose of these notes, that the program is executed with input redirected from \textit{standard input}, like so
\begin{itemize}
\item \texttt{./Gauss < gauss.dat}
\end{itemize}
Assume the contents of \texttt{gauss.dat} is a string of lines separated by newline characters like so:
\begin{verbatim}
a_{00} a_{01} a_{02} \ldots a_{0(n-1)} b_{0} \backslash n
a_{10} a_{11} a_{12} \ldots a_{1(n-1)} b_{1} \backslash n
\vdots \vdots \vdots \vdots \vdots \vdots
a_{n-1,0} a_{n-1,1} a_{n-1,2} \ldots a_{n-1,n-1} b_{n-1} \backslash n
\end{verbatim}

Let’s agree to use the Haskell \textit{idiom} \texttt{raw <- getContents} to input all of \texttt{gauss.dat} into one \texttt{String} called \texttt{raw}.
\begin{itemize}
\item The \texttt{lines} function breaks \texttt{raw} up into a list of strings \texttt{[String]}, separated at the newline. Call the result \texttt{rows}.
\item Mapping the \texttt{words} function over \texttt{rows} breaks each string in \texttt{rows} list of \texttt{Strings}. These values need to be converted from \texttt{String} to \texttt{Double}.
\end{itemize}

\begin{itemize}
\item \textbf{Main module 4c}\end{itemize}
\begin{itemize}
\item \texttt{main :: IO ()}
\item \texttt{main = do}
\item \texttt{raw <- getContents}
\item \texttt{let rows = lines raw}
\item \texttt{let rows' = map words rows}
\item \texttt{let matrix = map stringsToDoubles rows'}
\item \texttt{print $ matrix ++ gaussianReduce matrix}
\end{itemize}

\begin{itemize}
\item \textbf{IO conversion functions 4d}\end{itemize}
\begin{itemize}
\item \texttt{stringToDouble = read :: String -> Double}
\item \texttt{stringsToDoubles :: [String] -> [Double]}
\item \texttt{stringsToDoubles xs = map stringToDouble xs}
\end{itemize}
**Test the Reduction Step**

The noweb source file Gauss.nw is here.

- Running notangle Gauss.nw > Gauss.hs generates the Haskell code.
- Running noweave -index -delay Gauss.nw > Gauss.tex generates a \LaTeX file that is included in a wrapper \LaTeX file, that produces this document.

You may not have the noweb tools. You can retrieve the Haskell code from a link in the first step of this part of the assignment.

1. Download these files at the links: Gauss.hs and Gauss.tex.
2. Install the Glasgow Haskell Compiler and use its interpreter ghci to load Gauss.hs. Check the reduction code is correct on some simple cases, for instance,
   - `Gauss> gaussianReduce [[]]` – the empty matrix
   - `Gauss> gaussianReduce [[1]]` – a $1 \times 1$ matrix that needs no normalization
   - `Gauss> gaussianReduce [[2]]` – a $1 \times 1$ matrix that needs normalization
   - `Gauss> gaussianReduce [[1,2]]` – the equation $2x = 1$
   - `Gauss> gaussianReduce [[1,2],[2,3]]` – an inconsistent system
   - `Gauss> gaussianReduce [[1,-2,1,4],[2,3,-1,5],[3,1,4,7]]` – a picked out of the air example
3. Write code to generate $n$ lists of random Doubles of length $n + 1$ that represents a linear system $Ax = b$, where $A$ is an $n \times n$ matrix and $b$ is an $n \times 1$ vector.
4. Run your data generation code to generate several data files of varying sizes $n \times n + 1$.
5. Compile the Haskell source Gauss.hs with ghc profiling options, see the ghc User’s Guide on Profiling for instructions on this.
6. Execute Gauss on your data files, and collect running times.
7. Plot the running times. Find a curve that approximates the data. The $R^2$ value for the approximation should be close to 1.

**Analyze the reduction step**

A big-$O$ time complexity can be computed for each function in the code. For instance, the anonymous function \((\lambda a \ b \rightarrow \text{nr} \star a - b)\) has time complexity $O(1)$. The time cost to map it over a list of length $n + 1$ is $O(n)$.

What are the big-$O$ time complexities for the functions below. Explain your reasoning for each function. In particular, identify the size and type for the input and output of each function.

- `stringsToDoubles`
- `map stringsToDoubles`
• reduceonerow
• reduceRow
• take r m ++ [row'] ++ nextrows
• gaussianReduce

Does the empirical run time data from experiments in you performed when testing gaussianReduce agree with the analytic analysis?

**The Gaussian Solver**

The solving step in Gaussian elimination uses back substitution to solve for the values in $\vec{x}$ in order $x_{n-1}, x_{n-2}, \ldots, x_0$. Write Haskell code that implements the solver step.

6 \[ \text{(The Gaussian solver step 6)} \equiv \]
- gaussianSolve :: RMatrix -> [Double]
- Your code goes here

**Test the Solver Step**

Test your gaussianSolve code following steps similar to those outlined in testing gaussianReduce.

**Analyze the solver step**

1. What is the big-O time complexity of your gaussianSolve code?
2. Write a function that tests the accuracy of the computed solution $\vec{x}'$.
   (a) Use the $L_\infty$ norm to measure and report solution accuracy.
   \[ \|b' - b\|_\infty = \max \{|b_0' - b_0|, |b_1' - b_1|, \ldots, |b_{n-1}' - b_{n-1}|\} \]
   where $b' = A\vec{x}'$.
   (b) Hilbert matrices $H_n$ are famous examples of ill-conditioned matrices. Informally, this means solving $H_n\vec{x} = b$ accurately is difficult. The Hilbert $H_5$ is

   \[
   H_5 = \begin{bmatrix}
   1 & 1 & 1 & 1 & 1 \\
   1 & 1 & 1 & 1 & 1 \\
   1 & 1 & 1 & 1 & 1 \\
   1 & 1 & 1 & 1 & 1 \\
   1 & 1 & 1 & 1 & 1 \\
   \end{bmatrix}
   \]

   In general, the value in row $i$, column $j$ is
   \[ (H_n)_{ij} = \frac{1}{i+j+1}, \quad i, j = 0, \ldots, n-1 \]
   Test your code on Hilbert matrices.
References


Team Assignment

One quarter of a student’s course grade is based on team performance.

- Canvas will randomly generate three person teams. If enrollment is \(3k + 1\) there will be 2 two-person teams. If enrollment is \(3k + 2\) there will be 1 two-person team.
- Each team must schedule regular meetings of its members, bi-weekly at least.
  - The team will choose a topic to research from the list below. Notify the instructor of the team’s choice by email (mailto:wds@cs.fit.edu) on Monday of week 3 (Jan, 22). Requests will be granted first-come, first-serve. If a collision occurs the latter team must choose another available topic.
  - On Mondays of weeks 5 (Feb, 5), 8 (Feb, 26), 10 (Mar, 12), and 12 (Mar, 26) the team must email a status report to (mailto:wds@cs.fit.edu). The report must outline who has researched, read, written, . . . , what. Software development, mathematical analysis, and presentation outlines should be reported.
  - On Wednesdays and Fridays of weeks: 5, 8, and 12, teams schedule review meeting with the instructor during office hours: 9:30 – 10:50 or by appointment.

Potential Team Projects

Some of these topics are algorithms, others are data structures that support algorithms, and some are problems where the algorithm is unspecified.

- Aho – Corasick algorithm
- B-trees
- Boyer – Moore algorithm
- CYK algorithm
- Euclidean GCD algorithm
- Fibonacci heaps
- Ford – Fulkerson maximum flow
- Gale – Shapley stable matching
- Huffman codes
- k-nearest neighbors
- Knuth – Morris – Pratt algorithm
- Longest common subsequence
- Miller – Rabin primality test
- Multiprocessor scheduling
- QR matrix decomposition
- Rabin – Karp string search
- Red-Black trees
- Secure hash functions
- Set covers
- Simplex algorithm
- Singular value decomposition
- Strassen’s matrix multiplication algorithm
- Transitive closure
- Traveling salesman
- Union-Find for disjoint sets
- Vertex cover
- Skip list search

Team and Presentation Advice

Guidelines from Teamwork in the Classroom:
• Have clear goals
• Be results-driven
• Be a competent member
• Be committed to the goal

• Collaborate
• Have high standards
• Follow principled leadership
• Seek support, advice, and encouragement

A summary of Tufte’s tips for a successful presentation:

• Show up early
• Lay out the problem
• Present complicated material in order particular, general, particular, …
• Avoid an obvious reliance on notes
• Give everyone in your audience a piece of paper
• Match the information density to the allotted time

• Avoid overhead projectors, keep the lights on
• Never apologize
• Use (relevant, never irritating) humor
• Use gender-neutral speech
• Practice intensely beforehand
• Take questions without condescending
• Express (real) enthusiasm
• Finish early

Presentation schedule

The team must provide classmates with a one page, double sided handout highlighting the main results of their research. Every team member must participate in discussing the team’s work.

• Monday of week 15 (Apr, 16): Team 1 & 2
• Wednesday of week 15 (Apr, 18): Team 3 & 4
• Friday of week 15 (Apr, 18): Team 5 & 6
• Monday of week 16 (Apr, 23): Team 7 & 8

Final submission

On or before Wednesday of week 16 (Apr, 25) the team must submit on Canvas:

• The handout prepared for classmates
• Slides prepared for the presentation
• Code written, scripts (instructions) for compiling and executing your program, and test data
• A final report summarizing the work: The problem, the algorithm, the analysis, the experiments, the summary, and the references

On or before Wednesday of week 15 (Apr, 25) each student complete:

• A performance evaluation of their teammates
• An evaluation of presentations by other teams
Questions and Problems

Here’s collection of exercises are serve to outline topics and what students are expected to know by the end of the course.

Theory

1. What is a (deterministic) Turing machine?
2. What is a non-deterministic Turing machine?
3. Are there decision problems whose answers (“yes” or “no”) are unknown? Be able to explain your answer.
4. Are there decision problems that have no algorithm? Be able to explain your answer.
5. Are there decision problems where it is unknown whether or not an algorithm exists? Be able to explain your answer.
6. What is the key difference between decision problems and function problems?
7. Other than time and space, what other resources are consumed when an algorithm executes?

Asymptotic Notation

1. A running time (complexity) function \( f : X \rightarrow Y \) that defines a class \( O(f(x)) \) should have certain properties: What would you choose for:
   
   (a) \( X \), the domain of \( f \)?
   (b) \( Y \), the range of \( f \)?
   (c) The monotonicity of \( f \)?
   (d) The computability of \( f \)?
   (e) The time complexity of computing \( f(x) \)?
   (f) The space complexity of computing \( f(x) \)?

2. Define the class of running time functions \( O(n^2) \).

3. Find witnesses that show the following:
   
   (a) \( 5n + 6 = O(n^2) \)
   (b) \( 3n^2 + n + 6 = O(n^2) \)

4. Find witnesses that show the following:
   
   (a) \( 3n^2 + n + 6 = \Theta(n^2) \)
   (b) \( \log_{b_1} x = \Theta(\log_{b_2} x) \) for bases \( b_1, b_2 > 1 \).

5. True or False: All exponentials grow at the same rate: Is \( 2^n = \Theta(b^n) \) for every base \( b > 1 \)?

6. Assume \( f(n) \in O(g(n)) \). Prove that:
   
   (a) \( f(n)^2 = O(g(n)^2) \).
   (b) \( \lg f(n) = O(\lg g(n)) \).
   (c) \( 2^{f(n)} = O(2^{g(n)}) \)?

7. For each of the following pairs of functions \( f \) and \( g \), indicate whether \( f(n) = O(g(n)) \), \( f(n) = \Omega(g(n)) \), or both (i.e., \( f(n) = \Theta(g(n)) \)).
\[ f(n) = 100n + \lg n; \quad g(n) = n + (\lg n)^2 \]  
(1)  
\[ f(n) = \lg n \quad g(n) = \lg(n^2) \]  
(2)  
\[ f(n) = \lg n \quad g(n) = (\lg n)^2 \]  
(3)  
\[ f(n) = n^2/\lg n \quad g(n) = n(\lg n)^2 \]  
(4)  
\[ f(n) = (\lg n)^{\lg n} \quad g(n) = n/\lg n \]  
(5)  
\[ f(n) = \sqrt{n} \quad g(n) = (\lg n)^5 \]  
(6)  
\[ f(n) = n2^n \quad g(n) = 3^n \]  
(7)  
\[ f(n) = 100n + \lg n; \quad g(n) = n + (\lg n)^2 \]  
(8)  
\[ f(n) = \lg n \quad g(n) = \lg(n^2) \]  
(9)  
\[ f(n) = \lg n \quad g(n) = (\lg n)^2 \]  
(10)  
\[ f(n) = n^2/\lg n \quad g(n) = n(\lg n)^2 \]  
(11)  
\[ f(n) = (\lg n)^{\lg n} \quad g(n) = n/\lg n \]  
(12)  
\[ f(n) = \sqrt{n} \quad g(n) = (\lg n)^5 \]  
(13)  
\[ f(n) = n2^n \quad g(n) = 3^n \]  
(14)  

8. Define the slowdown due to an increased \( \triangle \) in input size to be

\[
\frac{f(n + \triangle)}{f(n)}
\]

Compute slowdowns for the following functions and input size increases of \( \triangle = 1 \) and \( \triangle = n \). Interpret the results.

(a) \( f(n) = \lg n \);  
(b) \( f(n) = n\lg n \)  
(c) \( f(n) = n^2 \)  
(d) \( f(n) = 2^n \)

**Summations**

The cost of executing a for loop or applying a function over a list depends on the cost of executing each step of the loop or evaluating the function. Please learn to use the identity

\[
\sum_{0 \leq i < n} \frac{1}{2} \left( \frac{i}{n} \right) = 2 \sum_{0 \leq i < n} i(i - 1) = 2 \binom{n}{3}
\]  
instead of the cumbersome

\[
\sum_{0 \leq i < n} i(i - i) = \sum_{0 \leq i < n} i^2 - \sum_{0 \leq i < n} i
\]

1. Find simple expressions for the following summations.

(a) \( \sum_{0 \leq k < n} 1 \)  
(b) \( \sum_{0 \leq k < n} k \)  
(c) \( \sum_{k \leq m < n} \binom{m}{k} \)  
(d) \( \sum_{0 \leq k < n} 2^k \)  
(e) \( \sum_{0 \leq i, j < n} 1 \)  
(f) \( \sum_{0 \leq i, j < n} 1 \)
2. Use summation notation to express the time complexity of the following loops. Test your answers by compiling and executing the code printing `sum`.

(a) A single `for` loop.
```c
int sum = 0;
for (int k = 0; k < n; k--) { sum = sum + 1; }
```

(b) A doubly-nested `for` loop.
```c
int sum = 0;
for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) { sum = sum + 1; }
}
```

(c) A triply-nested `for` loop.
```c
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) { sum = sum + 1; }
    }
}
```

(d) A triply-nested `for` loop with variable bounds.
```c
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        for (int k = j; k < n; k++) { sum = sum + 1; }
    }
}
```

Recurrence Relations

1. Verify the absorption identity for binomial coefficients

\[
\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}
\]

and explain why it is a good way to compute them?

2. Verify Pascal’s identity for binomial coefficients

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]

and explain why it is a poor way to compute them?

3. Consider the recurrence \( T(n) = T(n-1) + 2^{n-1} \) for \( n > 0 \), with initial value \( T(0) = c \).

(a) Show that \( T(n) = 2^n - 1 \) solves the recurrence.
(b) Show how to solve the recurrence $T(n) = T(n - 1) + 2^{n - 1}$, with initial value $T(0) = c$. by summing
the telescoping sum $T(k) - T(k - 1)$ for $k = 0$ to $k = n$.

4. Consider the recurrence $T(n) = 3T(n - 1) + 4T(n - 2)$.

(a) Solve the characteristic polynomial $x^2 = 3x - 4$
(b) Show that $T(n) = 4^n + (-1)^n$ solves the recurrence

5. Analyze the complexity of the algorithm

```
1. schedule :: [(Int, Int)] -> [(Int, Int)]
2. schedule [] = []
3. schedule [(s, f)] = [(s, f)]
4. schedule [(so, fo), (s1, f1):rest] =
   | s1 >= fo = [(so, fo), (s1, f1)] ++ schedule [(s1, f1):rest]
6. otherwise schedule [(so, fo):rest]
```

Generating Functions

1. Two fundamental generating functions are:

$$\frac{1}{1 - z} = \sum_{k \geq 0} z^k = \sum_{k \geq 0} \binom{k + 0}{0} z^k$$
and

$$\frac{1}{(1 - z)^2} = \sum_{k \geq 0} k z^{k-1} = \sum_{k \geq 0} \binom{k + 1}{1} z^k$$

Use induction to show

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

2. Use generating functions to show $M_n = 2^n - 1$ solves the Mersenne recurrence $M_n = 2M_{n-1} + 1$, $M_0 = 0$.

3. Show how to solve the recurrence $T(n) = T(n - 1) + 2^{n - 1}$ by generating functions.

4. Use generating functions to find the solution to the recurrence $T(n) = 3T(n - 1) + 4T(n - 2)$.

5. What function generates the sequence $\langle 1, 0, 1, 0, \ldots \rangle$?

Divide-and-Conquer Algorithms

Greedy Algorithms

1. The minimal spanning tree problem is: Given a graph $G = (V, E)$ with $n$ vertices and edge weights $w(i, j) > 0$ for $(i, j) \in E$. Find a spanning tree $T$ that minimized the weight of the tree

$$w(T) = \sum_{(i, j) \in E} w(i, j)$$

(a) What are the characteristics of a spanning tree $T$?

(b) One greedy heuristic is to process the edges in order of increasing cost: Start with an empty tree $T = \emptyset$. If edge $e = (i, j)$ does not contain a cycle when added to $T$, let $T = T \cup \{e\}$ until $T$ contains $n - 1$ edges.

2. Consider the traveling salesman problem.
(a) Describe a greedy heuristic that could be used to find an Hamiltonian cycle that approximates an optimal salesman’s cycle.

(b) Construct example weighted graphs where the greedy heuristic does and does not find an optimal Hamiltonian cycle.

**Dynamic Programming**

1. The dynamic programming paradigm requires computing values in an $n \times m$ matrix with values $A_{n,m}$. To order of computation depends on the recurrence that describes $A_{n,m}$ in terms of previously computed values. This dependence is typically by column $A_{i,j} \quad i = 0, \ldots, (n-1), j = 0, \ldots, (m-1)$ by row $A_{i,j} \quad j = 0, \ldots, (m-1), i = 0, \ldots, (n-1)$ or, by diagonal $A_{i,j} \quad i+j = 0, \ldots, n+m-2$

(a) What is the time complexity of a dynamic programming algorithm where each value in $A$ can be computed in $O(1)$ time?

(b) Sometimes only values in the upper (or lower) triangle of $A$ need to be computed. How does this alter the answer to the previous question?

(c) What nested for loops fill $A$ by columns order?

(d) What nested for loops fill $A$ by rows order?

(e) Filling $A$ by diagonals appears harder. From a position $(i, j)$ move to $(i + \Delta i, j + \Delta j)$ along a diagonal until a border is reached. Here’s diagram for a $3 \times 3$ matrix.

- On a right or left border move down: $(\Delta i, \Delta j) = (0, -1)$.
- On a top or bottom border move right: $(\Delta i, \Delta j) = (1, 0)$.
- On a downward diagonal move southwest: $(\Delta i, \Delta j) = (1, -1)$.
- On an upward diagonal move northeast: $(\Delta i, \Delta j) = (-1, 1)$.

To get things started move from entry $(0, 0)$ using displacements $(1, -1)$

1. walk $(n, m) = \text{step} (1, -1) (0, 0)$
2. where step $(\text{di}, \text{dj}) (i, j)$
3. \quad | (i, j) \equiv (n-1, m-1) | \{(i, j)\}
4. \quad | (i' == -1 && j' < m) \| i' == n = (i, j) : \text{step} (-\text{di}, -\text{dj}) (i, j + 1)
5. \quad | (j' == -1 && i' < n) \| j' == m = (i, j) : \text{step} (-\text{di}, -\text{dj}) (i + 1, j)
6. \quad | \text{otherwise} = (i, j) : \text{step} (\text{di}, \text{dj}) (i', j')
7. where \quad i' = i + \text{di}
8. \quad j' = j + \text{dj}
Load the code in the ghci interpreter and verify it correctly walks a matrix in diagonal order.

(f) In some problems, the number of columns, \(m\), in \(A\) is a bound on a constraint that any solution must satisfy. In these cases, the input size is taken to be \(\lceil \log m \rceil + 1\), the minimal number of bits needed to store \(m\). How does this alter the interpretation of the answer to problem 1a? question?

2. The 0–1 knapsack problem is: Given weights \(w_i\) and profits \(p_i\), \(0 \leq i < n\), and a capacity \(M\), find optimal packing of the knapsack. Let \(P[j, m]\) be the profit of an optimal solution using provisions \(0, \ldots, j\) and knapsack capacity \(m\).

(a) Write a dynamic programming algorithm for computing \(P[n - 1, M]\), the maximum value of the optimal solution. The recursion to use has initial conditions

\[
P[0, m] = \begin{cases} 
0 & \text{if } w_0 > m \\
p_0 & \text{if } w_0 \leq m 
\end{cases}
\]

and recurrence

\[
P[j, m] = \begin{cases} 
P[j - 1, m] & \text{if } w_j > m \\
\max \{P[j - 1, m], p_j + P[j - 1, m - w_j]\} & \text{if } w_j \leq m
\end{cases} \quad (1 \leq j < n \text{ and } 0 \leq m \leq M)
\]

(b) What rows and columns does the computation of \(P[j, m]\) depend upon?

(c) What is the time complexity of your algorithm?

(d) Is the time complexity polynomial or exponential in the number of provisions and the capacity \((n\) and \(M)\)?

(e) What would the time complexity be if, instead of storing values \(P[i, j]\) in a look-up (memoized) matrix, these values were computed by recursive functions calls?

3. The matrix chain product problem is: Given matrices \(M_0, M_1, \ldots, M_{n-1}\). Find an optimal way to parenthesize the product \(M_0 \cdot M_1 \cdots M_{n-1}\) to minimize the number of scalar multiplications.

(a) Show that the Catalan number

\[
C_{n-1} = \frac{1}{n} \binom{2n}{n-1}
\]

counts the number of different ways to parenthesize \(n\) matrices.

(b) Write a dynamic programming algorithm for computing \(C[0, n]\), the minimal value of the optimal solution. The recursion to use has initial conditions

\[
C[j, j] = 0 \quad \text{for } 0 \leq j \leq n - 1
\]

and recurrence

\[
C[j, k] = \min \{C[j, i] + C[i + 1, k] + r_j \cdot c_i \cdot c_k : j \leq i \leq k - 1\} \quad (1 \leq j < n, 0 \leq k \leq M)
\]

(c) What rows and columns does the computation of \(C[j, k]\) depend upon? A matrix layout diagram would be a good way to explain an answer.

(d) Show that the sum

\[
\sum_{j \leq i < k} 1
\]

(models the time cost of computing \(C[j, k]\))

(e) What is the time complexity of your algorithm?
(f) What would the time complexity be if, instead of storing values \( C[j, k] \) in a look-up (memoized) matrix, these values were computed by recursive functions calls?

4. The traveling salesman problem is: Given an undirected graph \( G = (V, E) \) on \( n \) vertices and weights \( w(i, j) > 0 \) for each edge \((i, j) \in E \). Find a minimal cost Hamiltonian cycle \( H \), a closed loop, starts at vertex 0 and visits every other node exactly once returning to 0.

For notation,

- Let \( S \) be a subset of \( \{1, \ldots, n - 1\} \)
- Let \( P[S, k] \) be the cost of an optimal path from vertex 0 to vertex \( k \), where all intermediate vertices lie in \( S \).

(a) Explain why \( \min \{ P[V - \{0\}, k] : 1 \leq k < n \} \) is the cost of an optimal salesman’s cycle.

(b) Explain why the recursion with initial conditions

\[
P[\{k\}, k] = w(0, k), \quad 1 \leq k < n
\]

and recurrence

\[
P[S, k] = \min \{ P[S - \{k\}, m] + w(m, k) : m \in S - \{k\} \} \quad \text{(if } |S| > 1 \text{)}
\]

describes the construction of optimal solutions.

(c) Write a dynamic programming algorithm for the traveling salesman problem. Your algorithm will calculate the values \( P[S, k] \) in order of increasing cardinality: \( |S| = 1, \ |S| = 2, \ldots, \ |S| = n - 1 \)

(d) What is the time complexity of your algorithm?

**Numerics**

1. Let \( \omega = e^{-2\pi i/4} \) be the principle 4-th root of unity.

   (a) Show that \( \omega^2 = e^{-\pi i} = -1 \).

   (b) Let \( m = 2k + r \) and show \( \omega^m = \omega^{2k+r} = (-1)^k \omega^r \)

**Backtracking**

**Complexity and Computability**
Algorithms

The word algorithm has several interpretations. A formal definition might be: An algorithm is a Turing machine $M$ that always halts when executed with $w$ written on $M$’s input tape. The output $u$, is what is written on the tape when $M$ halts. Every Turing machine that always halts on its input specifies some algorithm. (Knuth, 1997) says, an algorithm has properties:

- **Finiteness**: An algorithm must always terminate after a finite number of steps
- **Definiteness**: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case
- **Input**: (Some) quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects
- **Output**: (Some) quantities which have a specified relation to the inputs
- **Effectiveness**: All of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil

A prime consideration in the analysis of algorithms is the number of steps taken in computing the output from the input. We’d like to bound this number of steps above and below by functions of input size. We’ve also like to know how the algorithm behaves on average given a distribution of input values.

**Decision and Function Problems**

Algorithms are used to solve problems. **Decision problems** are basic: A question that has a “yes” (**True**) or “no” (**False**) answer is a decision problem. Consider the decision problem: Is the list of $n$ numbers $a_0, a_1, \ldots, a_{n-1}$ sorted?

**Function problems** require output that is more complex than “yes” or “no”. Function problems can often be reduced to a sequence of decision problems. Consider the function problem: $a_0, a_1, \ldots, a_{n-1}$. The output is the sorted list $a_{\pi(k)} \leq a_{\pi(k+1)}$ for $k = 0, \ldots, (n-1)$ where $\pi : \mathbb{Z}_n \mapsto \mathbb{Z}_n$ is a permutation, a one-to-one and onto function.

There are known problems for which there is no algorithm (assuming the Church–Turing hypothesis is True). The halting problem: Does Turing machine $M$ halt on input $w$? is undecidable.

There are known problems for which it is not known if an algorithm exists or not: Is there an infinite number of twin-primes?

For the problems covered here, almost always there will be more than one algorithm. Since this is the case, a method of comparing algorithms is sought.

- The number of steps (instructions) taken is one measure for comparison. The number of steps is called the time complexity of the algorithm. This begs the question: How can a step be quantified?
- The amount of memory used is another measure for comparison. The memory usage is called the space complexity of the algorithm. It is traditional to not include the input or output space in this measure. Only the extra space needed to implement the algorithm is counted. This may be a mistaken assumption in the era of big data. Memory usage can be measured by the number of addresses: A uniform model. Or, it can be measured logarithmically by the amount of memory at an address. (Think of a postman and mail delivery)
• What other factors could be measured.

Asymptotic (Order) Notation

The time complexity of an algorithm is a function of \( n \), its input size. Space complexity can also depend on input size, but sometimes it does not. Let \( T(n) \) and \( S(n) \) be used to stand for the time and space complexity of some algorithm.

For a fixed input size \( n \), an algorithm may be faster or slower on different instances: It may be faster to sort a list that is already sorted than a same sized list that is not sorted.

• Big-\( O \) notation is used to discuss the worst case running time of an algorithm.
• Big-\( \Omega \) (Omega) notation is used to discuss the best case running time of an algorithm.
• Big-\( \Theta \) (Theta) notation is used to discuss the running time of an algorithm that has identical best and worst case running times.

Informally, to say:
• \( T(n) = O(f(n)) \) means the graph of \( T(n) \) as \( n \) increases will be bounded above by the graph \( cf(n) \), for some constant \( c > 0 \). The standard definition (Knuth) is
  \[
  O(f(n)) = \{ g(n) : (\exists c > 0, m > 0)(\forall n \geq m)(g(n) \leq cf(n)) \}
  \]
• \( T(n) = \Omega(f(n)) \) means the graph of \( T(n) \) as \( n \) increases will be bounded below by the graph of \( cf(n) \), for some constant \( c > 0 \). The standard definition (Knuth) is
  \[
  \Omega(f(n)) = \{ g(n) : (\exists c > 0, m > 0)(\forall n \geq m)(g(n) \geq cf(n)) \}
  \]
• \( T(n) = \Theta(f(n)) \) means the graph of \( T(n) \) as \( n \) increases will bounded below by the graph of \( cf(n) \) and bounded above by the graph of \( df(n) \), for some constants \( c > 0 \) and \( d > 0 \).
  \[
  \Theta(f(n)) = \{ g(n) : (\exists c, d > 0, m > 0)(\forall n \geq m)(cf(n) \leq g(n) \leq df(n)) \}
  \]

Here is a list of common complexity functions \( f(n) \) in increasing order of growth.

\[
1 < \log \log n < \log n < n^\epsilon < n^c < n\log n < c^n < n^n < c^{cn}
\]

Here \( \epsilon \) and \( c \) are arbitrary constants with \( 0 < \epsilon < 1 < c \), and \( \prec \) is a relation defined by

\[
f(n) \prec g(n) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]

Courses on data structure teach methods to show how function \( g(n) \) is \( O(f(n)) \). Some methods are simple. For instance,

\[
n^2 \leq 3n^2 + 5n + 2 \leq (3 + 5 + 2)n^2 = 10n^2 \quad \text{showing} \quad 3n^2 + 5n + 2 = \Theta(n^2)
\]

Calculus concepts, limits and l’Hopital’s rule can be useful for more difficult problems: When used to analyze algorithms, the functions \( f(n) \) and \( g(n) \) grow without bound and l’Hopital’s rule states that when

\[
\lim_{n \to \infty} \frac{f'(n)}{g'(n)} = L \quad \text{then} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = L
\]
References

Algorithm Design

Maximum Subsequence Sum Problem

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

Motivation for the problem comes from instances.

• 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?

• Consider the bitcoin market. It too has runs up and down.

A particular example is a 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

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

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

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

additions.

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

The maximum subsequence sum problem is:

The maximum subsequence sum problem is: Given a list of integers \( X[k], k = 0, \ldots, (n-1), n \geq 0 \), find the maximal value of

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

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

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

Brute Force

The brute-force approach computes the sum of every possible subsequence and remembers the largest. It has \( O(n^3) \) running time.

\[
\text{int maxSubseqSum(int X[], int n)} \{
\text{int MaxSoFar = 0; // local state}
\text{for (int \( s = 0 \) to (n-1)) for (int \( e = s \) to (n-1)) let \( \text{MaxSoFar = max(MaxSoFar, \sum_{k=s}^{e} X[k])} \text{ compute \( \sum_{k=s}^{e} X[k] \) by brute force};}
\text{return MaxSoFar;}\}
\]
(For each start of a subsequence 21a) {  
(For each end of the subsequence 21b) {  
int Sum = 0; // More local state  
(For each subsequence 21c) {  
(Compute partial sum; Check MaxSoFar 21d)  
}  
}  
return MaxSoFar;
}  

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

(For each start of a subsequence 21a)≡  
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 21b)≡  
for (int end = start; end < n; end++)

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

(For each subsequence 21c)≡  
for (int k = start; k <= end; k++)

(Compute partial sum; Check MaxSoFar 21d)≡  
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, the cost of \langle Compute partial sum; Check MaxSoFar 21d \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

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

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

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

$$\sum_{s=0}^{n-1} c \frac{(n - s + 1)(n - s)}{2} = c \sum_{s=0}^{n-1} \frac{(n - s + 1)}{2}$$

$$= c \binom{n + 2}{3}$$

$$= c \frac{(n + 2)(n + 1)n}{6}$$

**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 $k$.

This ideas leads to a linear time/constant space algorithm.

\langle Imperative linear time algorithm 22 \rangle ≡

```cpp
int maxSubseqSum(int x[], int n) {
    ⟨Local state 23a⟩
    for ⟨(Every end position 23b)⟩ {
        ⟨Compute the maximum that ends here 23c⟩
        ⟨Compute the maximum so far 23d⟩
    }
    return MaxSoFar;
}
```
We need to keep track of the maximum so far and the maximum that ends at some position. Both can be initialized to 0.

\[ \text{Local state 23a} \equiv \]
\[
\begin{align*}
\text{int MaxSoFar} & = 0; \\
\text{int MaxEndingHere} & = 0;
\end{align*}
\]

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

\[ \text{Every end position 23b} \equiv \]
\[
\begin{align*}
\text{(int k = 0; k < n; k++)}
\end{align*}
\]

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 \( \text{MaxEndingHere} \) to 0.

\[ \text{Compute the maximum that ends here 23c} \equiv \]
\[
\begin{align*}
\text{MaxEndingHere} & = \max(0, \text{MaxEndingHere} + x[k]);
\end{align*}
\]

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

\[ \text{Compute the maximum so far 23d} \equiv \]
\[
\begin{align*}
\text{MaxSoFar} & = \max(\text{MaxSoFar}, \text{MaxEndingHere});
\end{align*}
\]

**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 \( \text{snd} \), returns the second element in a pair.

The Haskell function \( \text{snd} \) returns the second value in a pair.

\[ \text{The second of a pair function 23e} \equiv \]
\[
\begin{align*}
\text{snd} & :: (a,b) \to b \\
\text{snd (x,y)} & = y
\end{align*}
\]

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

\[ \text{Folding a from the left 23f} \equiv \]
\[
\begin{align*}
\text{foldl} & :: (a -> b -> a) -> a -> [b] -> a \\
\text{foldl f z []} & = z \\
\text{foldl f z (x:xs)} & = \text{foldl f (f z x) xs}
\end{align*}
\]

\[
\begin{align*}
\text{sum [a]} & = \text{foldl (+) 0 [a]} \\
\text{product [a]} & = \text{foldl (*) 1 [a]}
\end{align*}
\]
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. It did not execute in linear time as claimed. There is was a simple fix.

A Haskell functional algorithm for maximum subsequence sum is given below.

A Haskell functional algorithm for maximum subsequence sum is given below.

\[
\text{maxsubseq} :: (\text{Ord a}, \text{Num a}) \Rightarrow [a] \rightarrow (a, [a])
\]

\[
\text{maxsubseq} = \text{snd} \ . \ \text{foldl} \ f ((0,[]),(0,[])) \text{ where}
\]

\[
f ((\text{maxToHere},\text{witnessToHere}),\text{sofar}) x = (a,b) \text{ where}
\]

\[
a = \max (0,[]) (x:\text{maxToHere}, x:\text{witnessToHere})
\]

\[
b = \max \text{sofar} a
\]

**Questions and Problems**

1. 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?

2. What is the solution to the recurrence equation

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

3. What is the solution to the recurrence equation

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

**References**


**Mathematical Tools**

**Summations**

**To analyze algorithms requires skills.** Some of these are mathematical. Mastery of summations is a fundamental skill. Summations are discrete analogs of integrals from continuous mathematics. There are many useful summations. Here are a few. Mathematical induction can be used to prove they are correct.

- **Tally sums:**
  $$\sum_{0 \leq k < j} m = mj$$  
  sums along a flat line increase linearly.

  Code template:
  ```cpp
  for (int k = 0; k < j; k++) { sum = sum + k; }
  ```

- **Arithmetic sums:**
  $$\sum_{0 \leq j < n} (mj + b) = mn\binom{n}{2} + nb$$  
  sums along a sloped line increase quadratically.

  Code template:
  ```cpp
  for (int j = n-1; j > 0; j--) {
    for (int k = 1; j <= j; k++) {
      if (A[k-1] > A[k]) { swap(A[k-1], A[k]); }
    }
  }
  ```

- **Binomial coefficients column sums** generalize the above results to higher dimensions. Let the column be fixed at $m$, and sum binomial coefficients
  $$\sum_{0 \leq i \leq n} \binom{i}{m} = \binom{n+1}{m+1}; \quad \text{When } m = 2, \sum_{0 \leq i \leq n} \binom{i}{2} = \binom{n+1}{3}$$

  Code template:
  ```cpp
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      for (int k = 0; k < j; k++) { O(1) operation; }
    }
  }
  ```

- **Geometric sums**
  $$\sum_{0 \leq k < n} (1 + r)^k = \frac{(1 + r)^n - 1}{r}$$  
  sum along an exponential

  Good for earning money, bad for time complexity.

(Graham et al., 1989) is a great source for summations.
Recursion

Recursion is a common problem solving strategy. Express a solution in terms of solutions to reduced problems. Recurrence equations are discrete analogs of differential equations from continuous mathematics.

There are many useful recurrences and simple functions that satisfy them. There is a master theorem that summarizes many cases. You can always look it up. I prefer to know the basic rules that lead to it.

- Pretend \( n \) is a power of 2. The function \( T(n) = \lg(n) \) satisfies the recursion
  \[
  T(n) = T(n/2) + 1
  \]
  \[
  \tag{15}
  \]

Code template:

```haskell
binarySearch :: Integral a => (a -> Ordering) -> (a, a) -> Maybe a
binarySearch p (low, high)
  | high < low = Nothing
  | otherwise =
    let mid = (low + high) 'div' 2
    in case p mid of
        LT => binarySearch p (low, mid - 1)
        GT => binarySearch p (mid + 1, high)
        EQ => Just mid
```

Well, this demands some explanation doesn’t it. Like Fermat, I have discovered a truly marvelous description of this, which this margin is too narrow to contain.

- Pretend \( n \) is a power of 2. The function \( T(n) = n \lg(n) \) satisfies the recursion
  \[
  T(n) = 2T(n/2) + n
  \]
  \[
  \tag{16}
  \]

Code template:

```haskell
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

Well isn’t that neat: To quicksort a list starting with \( x \), quicksort the values less than \( x \) and those greater that (or maybe equal to) \( x \) and put them in order.

- Some recursions grow exponentially. The function \( T(n) = 2^n - 1 \) satisfies the recursion
  \[
  T(n) = 2T(n - 1) + 1
  \]
  \[
  \tag{17}
  \]

Code template:

```haskell
hanoi :: Integer -> a -> a -> a -> [(a, a)]
hanoi 0 _ _ _ = []
hanoi n a b c = hanoi (n-1) a c b ++ [(a,b)] ++ hanoi (n-1) c b a
```

Questions and Problems

1. Use mathematical induction to prove the summation formulas for tally sums, arithmetic sums, binomial coefficient column sums, and geometric sums.
2. Use direct substitution to prove the given functions satisfy the recurrences 15, 16, 17.
3. Find simple expressions for the following summations.
4. Interpret the above sums as for loops.

References

More Mathematical Tools

Generating Functions

To analyze algorithms requires skill. Some of these are mathematical. The best math book for computer science ever written is 4.

Generating functions are a powerful tool for skillfully solving recurrence relations. I recommend the book (Wilf, 2006) for a deep understanding of generating functions. The idea is that a time complexity function \( T(n) \) generates a sequence of values \( \vec{T} = \langle t_0, t_1, \ldots \rangle \) as \( n \) ranges over the natural numbers. These values become coefficients in a formal power series

\[
G(\vec{T}, z) = \sum_{k=0}^{\infty} t_k z^k
\]

Generating functions are discrete analogs of integral transforms from continuous mathematics. There are many functions that generate useful sequences. Here are a few interesting functions.

\[
\begin{align*}
\frac{1}{1-z} &= \sum_{n=0}^{\infty} z^n \\
(1-z)^{-2} &= \sum_{n=0}^{\infty} n z^{n-1} \\
\frac{z}{(1-z-z^2)} &= \sum_{n=0}^{\infty} F_n z^n \\
\frac{1}{1-2z} &= \sum_{n=0}^{\infty} 2^n z^n \\
-\ln(1-z) &= \sum_{n=0}^{\infty} \frac{z^{n+1}}{n+1} \\
(1+z)^m &= \sum_{n=0}^{\infty} \binom{m}{n} z^n
\end{align*}
\]

A simple example shows how generating functions can be used to compute the time complexity of an algorithm. Pretend you are given the recurrence relation

\[
T(n) = T(n-1) + (n-1), \quad n > 0
\]

corresponding to a code template

\[
\begin{array}{l}
\text{function } (n) \text{ = function } (n-1) \text{ + (n-1) other operations}
\end{array}
\]

Now consider the calculation to find the sequence for the function \( G(z) = \sum_{n=0}^{\infty} t_n z^n \).

\[
G(z) = t_0 + \sum_{n=1}^{\infty} t_n z^n = t_0 + \sum_{n=1}^{\infty} [t_{n-1} + (n-1)] z^n = t_0 + \sum_{n=0}^{\infty} t_n z^n + z \sum_{n=0}^{\infty} n z^n
\]

\[
= t_0 + zG(z) + \frac{z}{(1-z)^2}
\]

\[
G(z) = \frac{t_0}{1-z} + \frac{z}{(1-z)^2} = \sum_{n=0}^{\infty} t_0 z^n + \sum_{n=0}^{\infty} \frac{n(n-1)}{2} z^n
\]

\[
= \sum_{n=0}^{\infty} \left[ t_0 + \binom{n}{2} \right] z^n
\]

\[
T(n) = t_0 + \binom{n}{2}
\]
Complexity Functions

A function \( f : \mathbb{N} \mapsto \mathbb{N} \) is a proper complexity function (Papadimitriou, 1994) if

1. \( f \) is non-decreasing: \( f(n+1) \geq f(n), \forall n \)
2. There exists a Turing machine \( M_f \) such that on input \( x \):
   - \( M_f(x) = \square^{f(|x|)} \) \( M \) computes a string of blanks of length \( f(|x|) \).
   - \( M_f(x) \) halts after \( O(|x| + f(|x|)) \) steps
   - \( M_f(x) \) uses \( O(|x| + f(|x|)) \) space

Harmonic Numbers

The harmonic number \( H_n \) is defined by

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

The name derives from music (the diagram is from Wikipedia)

An asymptotic formula for Harmonic numbers

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

where \( \gamma \approx 0.577215664901 \cdots \) is another one of Euler’s constants, not \( e = 2.71828 \cdots \). You might recognize the Bernoulli numbers in this expansion of \( H_n \)

References

Representative Problems

The idea here is to consider a sequence of problems that resemble one another, but have increasing levels of difficulty. The outline is based on chapter 1 of (Kleinberg and Tardos, 2006).

Interval scheduling

Interval scheduling: A set of overlapping intervals (jobs) over a period of time. Each interval $k$ has start and finish times $s_k < f_k$ and duration $f_k - s_k$. Two jobs $i$ and $j$ are compatible if they do not overlap: $f_i \leq s_j$ or $f_j \leq s_i$ (job $i$ finish at or before the start of job $j$, and vice versa.

A function problem is to find a subset of mutually compatible intervals that has the largest duration: sum of lengths.

$$\max \left\{ \sum_{k \in A} (f_k - s_k) : k \in A, \text{where } A \text{ is a set of mutually compatible intervals} \right\}$$

Pretend the jobs have been sorted by earliest finishing time

$$(\forall k)(f_k \leq f_{k+1})$$

A greedy approach that can be viewed as matching problem. Visualize the problem as two parallel timelines: One for starts and one below for later finishes. Plot some start times on the top line and some later finishing times on the bottom line; connect them with dotted lines.

Schedule the first job and thicken its line from start to finish. Move along the start line until the first start time $s_k$ that is at or after the last finishing time. Include this job in the mix. Continue until all intervals have been considered. A function that does this takes a list of (start, finish) time of jobs and returns a list of (start, finish) times of scheduled jobs. Pretend the jobs have been presorted by finishing times.

1. $\text{schedule} : \ [(\text{Int, Int})] \rightarrow [(\text{Int, Int})]$
2. $\text{schedule} \ [\ ] = [\ ]$
3. $\text{schedule} \ [(s, f)] = [(s, f)]$
4. $\text{schedule} \ ((s_0, f_0) : (s_1, f_1) : \text{rest})$
   | $s_1 \geq f_0 = (s_0, f_0) : (\text{schedule} \ ((s_1, f_1) : \text{rest}))$
   | otherwise $= (s_0, f_0) : (\text{schedule} \ \text{rest})$

Assuming jobs are presorted by finishing times, the time complexity of this greedy approach is $O(n)$.

The function schedule does not compute the maximum duration. To do that, a function that take the output of schedule and sums the duration of each interval is needed.

Weighted interval scheduling

Now pretend each job $k$ has an associated value $v_k$. Imagine the edges in the start-finish graph as having weights representing the job’s value.

$$\max \left\{ \sum_{k \in A} v_k : k \in A, \text{where } A \text{ is a set of mutually compatible intervals} \right\}$$

Greedy approaches do not succeed for this variation. (An early to finish job may have little value, while high value non-compatibles job are not chosen.) A dynamic programming approach where optimal solutions
are memoized for each increasing subset of jobs: 0, 1, . . . , k can produce an optimal solution. As each new job is considered, the new optimal value is the largest of the previous set of jobs or the value of the new job plus the value optimal subset of jobs compatible with the new job.

Define \( p(k) \), for an job \( k \), to be the largest index \( i < k \) such that intervals \( i \) and \( k \) are disjoint. Define \( p(k) = 0 \) if no request \( i < k \) is disjoint from \( k \). A top-down recursive algorithms in pseudo-code is:

```pseudocode
Compute−Opt (k) {
    | (k == 0) = 0
    | (k == 1) = v[1]
    | otherwise = max(Compute−Opt (k−1), v[k] + Compute−Opt (p(k)))
}
```

A bad instance occurs when \( p(k)=k-2 \) for all \( k \). In this case, Compute−Opt makes a Fibonacci-like sequence of calls and the algorithm executes in exponential time in the worst case.

Computing the optimal values bottom-up. Store the optimal values in an array and look them up when needed.

```pseudocode
Memoized−Compute−Opt {
    opt[0] = 0;
    for (j = 1; j < n; j++) {
        opt[j] = max(v[j] + opt[p[j]], opt[j-1]);
    }
    return opt[n];
}
```

**Bipartite matching**

Given a graph \( G = (V, E) \) can its vertices \( V \) be partitions into sets \( X \) and \( Y \) such that each edges starts in \( X \) and ends in \( Y \).

A *matching* is a subset of edge \( M \subseteq E \) such that no two edges in \( M \) share a common vertex. Each vertex \( v \in V \) is in at most one edge in \( M \). Some vertices may not be on any edge in \( M \).

A *perfect matching* is a matching \( M \subseteq E \) such each vertex \( v \in V \) is in exactly one edge in \( M \). That is, every vertex of the graph is incident to exactly one edge of the matching.

**Independent Set**

In a graph \( G = (V, E) \) a set \( I \) of vertices is *independent* if no two vertices in \( I \) are adjacent (there is no edge connecting any pair of vertices in \( I \)).

A function problem is to find a maximal independent set. The graph below shows an independent set of size 6, the black nodes.
Finding the maximal independent set is an NP-hard optimization problem. A non-deterministic polynomial-time hard problem $P$ is one such that every NP problem $Q$ can be reduced to $P$ in polynomial time. As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered hard.

References

Binomial Coefficients

Pascal’s Triangle: Divide-and-Conquer

Consider computing binomial coefficients by recursively applying Pascal’s identity:

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]

with boundary conditions

\[
\binom{n}{0} = \binom{n}{n} = 1
\]

The recursion tree shows multiple overlapping sub-problems, which is a clear sign that direct recursion will be more costly than necessary. At each level the values of the left and right nodes are added: There is one add at level 1, two adds at level 2, four adds at level 3, six adds at level 4, and six adds at level 5. A total of 19 additions.

The factorial expression

\[
\frac{n!}{k!(n-k)!}
\]

solves the equation and boundary conditions, so it must be big-O time complexity of an algorithm that implements this recurrence. Indeed \(\binom{6}{3} = 20\) and 19 additions were counted (there are 19 additions among 20 terms). A recursion stack depth of size \(n\) must also be maintained.

Binomial coefficients attain their maximum at the median of each row. In even rows this is

\[
\frac{(2n+1)!}{n!(n+1)!} = \frac{(2n+1)!}{(n+1)!n!}, \quad n = 1, 2, 3, \ldots
\]

In odd rows there are two maxima.

\[
\frac{(2n+1)!}{n!(n+1)!} = \frac{(2n+1)!}{n!(n+1)!}, \quad n = 1, 2, 3, \ldots
\]

Stirling’s asymptotic approximation

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

can be used to approximate the factorial expression for binomial coefficients.

\[
\frac{2n!}{n!n!} \sim \frac{\sqrt{2\pi 2n} \left(\frac{2n}{e}\right)^{2n}}{(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n)^2} = \frac{2^n}{\sqrt{n\pi}}
\]
Pascal’s Triangle: Dynamic Programming

Precompute, don’t recompute! The dictum of dynamic programming. Each binomial coefficient is computed once, from the bottom up and stored in a table. Given $n$ and $k$, do this:

1. For each row $i = 1$ to $n - 1$ and for each column $j$ compute $bc[i][j] = bc[i-1][j] + bc[i-1][j-1]$.

Using simple translations of for loops into summations, the time complexity is

$$
\sum_{i=1}^{k} \sum_{j=1}^{n} c + \sum_{i=k+1}^{n} \sum_{j=1}^{k} ic + \sum_{i=k+1}^{n} kc = c \left( \binom{k+1}{2} + ck(n-k) \right) \leq \frac{n(n+2)}{4} + \frac{n^2}{4}
$$

This captures a main concept of dynamic programming, use of a table to store values, but it misses an important aspect: Neither a minimum or maximum is required over a set of previously computed values.

The Factorial Algorithm

The factorial expression could be evaluated directly. Computing $n!/k!(n-k)!$ takes $n-1$ multiplies for the numerator and $(k-1) + (n-k-1) + 1 = n-1$ multiplies for the denominator plus one division. However, integer overflow is a concern. Assuming a 64 bit register for naturals, the largest value that can be stored is $2^{64} - 1 = 18,446,744,073,709,551,616$ – 1. But at $21! = 51,090,942,171,709,440,000$ overflow will occur.

The Absorption Identity

It is easy to verify the absorption identity for binomial coefficients.

$$
\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}, \quad \binom{n-k}{0} = 1, \quad \binom{n-k+1}{1} = n-k+1
$$

This leads to a linear recursion tree. For instance,

$$
\binom{6}{3} = \binom{5}{2} \cdot \frac{6}{3} = 6 \cdot \frac{5}{2} \cdot \frac{4}{1} = \frac{6 \cdot 5 \cdot 4}{3 \cdot 2 \cdot 1}
$$

It may be best to refactor the computation to avoid any complications between integer and floating point arithmetic.

$$
\binom{6}{3} = \binom{5}{2} / 3 = \frac{6 \cdot 5}{2 \cdot 1} / 3 = \frac{6 \cdot 5 \cdot 4}{2 \cdot 1 \cdot 3}
$$

This approach executes $k-1$ multiplies and $k-1$ divides. And, when unrolled like this, the algorithm becomes a product method for computing $\binom{n}{k}$. Here’s Haskell code for this computation.

```haskell
choose :: (Integral a) => a -> a -> a
choose n k = product [k+1..n] 'div' product [2..n-k]
```
**Divide and Conquer**

Divide-and-conquer is attributed to Phillip of Macedonia, father of Alexander the Great. In politics and warfare the phrase describes a strategy to separate adversaries into smaller and smaller groups until each can be readily defeated. In computer science, divide-and-conquer summarizes a problem solving paradigm. Unlike warfare, where the objective is destruction, divide-and-conquer problem solving requires rebuilding an overall solution from solutions to subproblems. There are several classic divide-and-conquer algorithms.

**Euclidean Algorithm**

The Euclidean algorithm is literally divide-and-conquer. It computes the largest common divisor of two integers, not both 0.

\[
\begin{align*}
&\text{gcd}(a, 0) = a \\
&\text{gcd}(a, m) = \text{gcd}(m, a \mod m), \quad \text{for } m > 0
\end{align*}
\]

In C, the code might look something like this:

```c
int gcd(int a, int m) {
    int t;
    while (m) {
        t = a; a = m; m = t % a;
    }
    return a < 0 ? -a : a;
}
```

Here's a functional implementation from Haskell's standard Prelude.

```haskell
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)
```

Analysis of the Euclidean algorithm is the subject of a student presentation. The upshot is that Fibonacci-like sequences produce the worst case time behavior. The time complexity grows like the log base \( \varphi \approx 1.618 \) of max \( \{|x|, |y|\} \). The \texttt{gcd} algorithm is very efficient.

**Merge Sort**

Consider the well-known \texttt{mergeSort} algorithm. The cost depends on splitting a list into two parts.

```haskell
split (x:y:zs) = let (xs,ys) = split zs in (x:xs,y:ys)
split [x] = ([x],[])
split [] = ([],[])
```
This splitting is not stable, but is reasonable efficient on lists. It places every other item on one of two lists. The recurrence that describes \texttt{split}'s time complexity is

\[ T_{\text{split}}(n) = T_{\text{split}}(n - 2) + c, \quad T_{\text{split}}(n) = \frac{nc}{2} = O(n) \]

The split lists will have the same size or the first will be one more than the second. Once split, the two lists need to be merged into one.

\begin{verbatim}
merge [] ys = ys
merge xs [] = xs
merge xs@(x : x t) ys@(y : yt) | x <= y = x : merge x t ys
otherwise = y : merge xs yt
\end{verbatim}

The recurrence that describes \texttt{merge}'s time complexity is

\[ T_{\text{merge}}(n) = T_{\text{merge}}(n - 1) + b, \quad T_{\text{merge}}(n) = nb = O(n) \]

To complete the sort, split the lists, \texttt{mergeSort} both halves, and merge the results.

\begin{verbatim}
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = let (as, bs) = \texttt{split} xs
         in merge (mergeSort as) (mergeSort bs)
\end{verbatim}

The recurrence that describes \texttt{mergeSort}'s time complexity is

\[ T_{\text{mergeSort}}(n) = T_{\text{split}}(n) + 2T_{\text{mergeSort}}(n/2) + T_{\text{merge}}(n) \]

Or, more simply

\[ T_{\text{mergeSort}}(n) = 2T_{\text{mergeSort}}(n/2) + O(n) \]

which as solution

\[ T_{\text{mergeSort}}(n) = O(n \log n) \]

\textit{Binary Search}

\textbf{Binary search is a classic algorithm}. Given a ordered list \( \texttt{xs} \), a value to search for, and a range from \texttt{low} to \texttt{high} over which to search, \texttt{binsearch} returns the position (index) where value is found, or \texttt{Nothing} if it is not in the list. The cases are:

\begin{itemize}
  \item \texttt{Nothing} when the range is empty
  \item The result of searching the bottom half of the range when value is smaller than the value in the middle of the range
  \item The result of searching the top half of the range when value is larger than the value in the middle of the range
  \item And, \texttt{Just mid}, the middle value when it equals value.
\end{itemize}
binsearch :: [Int] -> Int -> Int -> Int -> Maybe Int
binsearch xs value low high
  | high < low = Maybe Nothing
  | xs !! mid > value = binsearch xs value low ( mid – 1)
  | xs !! mid < value = binsearch xs value ( mid + 1) high
  | otherwise = Just mid
  where
    mid = low + ( (high - low) ‘div’ 2)

Working with lists instead of arrays increased the running time of the algorithm. There is a cost of order $O(n/2)$ to walk through the list to its middle. The time complexity recurrence for the list-based algorithm is

$$T(n) = T(n/2) + cn, \quad n = 2^k, \quad k = 1, 2, \ldots, \quad T(1) = 1$$

where $n$ is the length of the list $x$.

$$T(n) = T(n/2) + cn$$
$$= [T(n/4) + cn/2] + cn$$
$$= [T(n/8) + cn/4] + cn/2 + cn$$

$$= \cdots$$
$$= T(n/2^k) + cn/2^{k-1} + \cdots + cn/4 + cn/2 + cn$$
$$= T(1) + cn \sum_{0 \leq j < k} (1/2)^j$$
$$= 1 + cn \left( 2 - \frac{2}{n} \right)$$
$$= 2cn + 1 - 2c = O(n)$$

The array-based algorithm has constant time access to values by index. And, in this case, the recurrence is

$$T(n) = T(n/2) + c, \quad n = 2^k, \quad k = 1, 2, \ldots, \quad T(1) = 1$$

Show $T(n) = O(n \log n)$
for the equation $T(n) = T(n/2) + c$. 
The 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 idea is to place the head of a list in its correct position with smaller elements before it and larger elements after it. The head is often called the pivot.

Here imperative code for quick sorting (Bentley, 1984). Assume a sentinel at A[0] that keeps indices within bounds.

```
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);
    }
}

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

Let's analyze quicksort.
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 $\text{lo}$ to $\text{pivot} - 1$. 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, or telescoping sums

$$T(n) = (n + 1) + (n - 1) + \cdots + 3 + T(1)$$

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

$$= (n + 1)(n + 2)/2 - 2$$

$$= O(n^2)$$

In the best case, the pivot is $p = n/2$ and partitions the array into sub-arrays of equal size. The recurrence relation is

$$T(n) = (n + 1) + 2T(n/2)$$

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 $q = \lg n$.

References


Quicksort

Quicksort (Hoare, 1961) is a classic algorithm that has a nice description in Haskell. The outline for the Quicksort module is simple: Import other needed modules, define functions as needed, and include a main function for IO and testing.

```haskell
module Quicksort (quicksort) where

import external modules

auxiliary functions

main function

The quicksort function sorts a list of values into increasing order. The idea is simple. Given a list (p:xs) with pivot p at its head:

1. Walk over the list and place all values x in the tail xs that are less than p in a list.
2. Quicksort this list.
3. Concatenate the sorted list with [p].
4. Walk over the original list again and place all values x in the tail xs that are greater than or equal to p in another list.
5. Quicksort this list.
6. Concatenate this sorted list with the list from step 3.

Here’s the code.

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

QuickSort Analysis

Steps 1 and 4, above, have $O(n)$ running time. The cost of concatenations in these steps depends on the length $k$ of of the first list. In any case, the costs are bounded by $O(n)$.

If there are $n$ values to be sorted and the pivot splits them into lists of size $k$ and $n - k - 1$, then the recurrence for the overall time complexity is

$$T(n) = T(k) + T(n - k - 1) + O(n)$$

- In the worst case, $k = 0$ or $k = n - 1$ and the recurrence has an $O(n^2)$ solution.
- In the best case, consider even $n = 2k$ and odd $n = 2k + 1$ input sizes. The recurrences are

$$T(2k) = T(k) + T(k - 1) + O(n) \quad \text{and} \quad T(2k + 1) = 2T(k) + O(n)$$

In either case the time complexity now is $O(n \log n)$.

- The average case running time is $O(n \log n)$ assuming the pivot lands in all positions of the list with uniform probability. The analysis is relegated to the appendix.
Testing Quicksort

To test the code, generate some random sequences.

\[ \text{Import external modules 41a} \equiv \]
\begin{verbatim}
import System.Random
And, create a list of \( n \) random integers with \( \text{createList} \ n \).
\end{verbatim}

\[ \text{Auxiliary functions 40b} + \equiv \]
\begin{verbatim}
createList :: Int -> [Int]
createList \( n \) = take \( n \) (randoms (mkStdGen 10))
\end{verbatim}

The main routine can then read an integer, create a list, sort it, and show the results.

\[ \text{Main function 41c} \equiv \]
\begin{verbatim}
main :: IO ()
main = interact (show . quicksort . createList . read)
\end{verbatim}
Appendix: An average case analysis of quicksort

The expression for average case running time is

\[ T_{\text{ave}}(n) = \sum_{\text{all cases}} \Pr(\text{case}) T(\text{case}) + (n + 1) \]

Write the Quicksort recurrence as

\[ T(n) = T(k) + T(n - k - 1) + (n + 1), \quad n > 0, \quad \text{and} \quad T(0) = 1 \]

Let’s assume, with uniform probability \( \Pr(\text{case}) = \frac{1}{n} \), that the pivot partitions the length \( n \) list into two lists: One of length \( k \) and another of length \( n - k - 1 \), for \( k = 0, \ldots, n - 1 \). This gives the average case expression for Quicksort at

\[ T(n) = (n + 1) + \frac{1}{n} \sum_{0 \leq k < n} [T(k) + T(n - k - 1)] \]

\[ = (n + 1) + \frac{1}{n} \left[ \sum_{0 \leq k < n} T(k) + \sum_{0 \leq k < n} T(n - k - 1) \right] \]

\[ = (n + 1) + \frac{1}{n} \left[ \sum_{0 \leq k < n} T(k) + \sum_{0 \leq k < n} T(k) \right] \]

\[ = (n + 1) + \frac{2}{n} \sum_{0 \leq k < n} T(k) \]

Now, massage \( T(n) \) into shape.

- Remove the fraction by multiplying by \( n \) giving

\[ nT(n) = n(n + 1) + 2 \sum_{0 \leq k < n} T(k) \]

- Remove the sum by substituting \( n + 1 \) for \( n \), and subtract the two expressions

\[ (n + 1)T(n + 1) - nT(n) = \left[ (n + 1)(n + 2) + 2 \sum_{0 \leq k < n + 1} T(k) \right] - \left[ n(n + 1) + 2 \sum_{0 \leq k < n} T(k) \right] \]

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

Next, define the generating function \( G(z) \) for the sequence \( (T_0, T_1, \ldots) \).

\[ G(z) = T_0 + T_1z + T_2z^2 + T_3z^3 + \cdots \]

\[ = \sum_{0 \leq k} T_kz^k \]
And notice

\[ G'(z) - zG'(z) = \sum_{0 \leq k} kT_k z^{k-1} - z \sum_{0 \leq k} kT_k z^{k-1} \]

\[ (1 - z)G'(z) = \sum_{0 \leq k} (k + 1)T_{k+1} z^k - \sum_{0 \leq k} kT_k z^k \]

\[ = \sum_{0 \leq k} [(k + 1)T_{k+1} - kT_k] z^k \]

\[ = \sum_{0 \leq k} [2(k + 1) + 2T_k] z^k \]

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

Multiply by \((1 - z)\) and rearrange terms to get a perfect differential.

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

that is

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

Integrating both sides gives

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

where \(C = 0\) since \(G(0) = T_0 = 0\) So we have

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

The Cauchy product of two power series is

\[ \left( \sum_{0 \leq i} a_i x^i \right) \cdot \left( \sum_{0 \leq j} b_j x^j \right) = \sum_{0 \leq n} c_n x^n \]

where

\[ c_n = \sum_{0 \leq k \leq n} a_{n-k} b_k \]

It follows that

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

\[ = 2 \sum_{1 \leq i} (i + 1) z^i \cdot \sum_{1 \leq j} z^j \]

\[ = 2 \sum_{1 \leq n} \left[ \sum_{1 \leq k \leq n} \frac{n + 1 - k}{k} \right] z^n \]

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

\[ = \sum_{1 \leq n} [2(n + 1) H_n - 2n] z^n \]
Recall the Harmonic numbers

\[ H_n = \sum_{1 \leq k \leq n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \]

\[ = H_{n-1} + \frac{1}{n} \quad n \geq 1 \]

\[ H_0 = 0 \]

The harmonic number \( H_n \) can be asymptotically approximated by

\[ H_n \sim \ln n + \gamma + O(1/2n) \]

where \( \gamma \approx 0.5772156649 \cdots \) is Euler’s constant.

Therefore, the average case time complexity of Quicksort is

\[ T(n) = 2(n + 1)H_n - 2n = O(n \log n) \]

References


Order Statistics and the Selection Problem

Let \( A \) be a set of \( n \) values from a totally ordered set. The \( i \)th order statistic is the \( i \)th smallest value in \( A \).

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 \( 1 \leq i \leq n \). It can be posed as a decision or a function problem.

**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.

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

\[
A = \{7, 12, 5, 17, 9, 1, 14, 8, 18\} \\
\text{Order} = \{3, 6, 2, 8, 5, 1, 7, 4, 9\}
\]

Simple linear time algorithms solve the minimum and maximum problems. Assume \( A \) is represented as a 0-indexed array of \( n \) integers.

```c
int minimum(int A[], int n) {
    int min = A[0];
    for (int k = 1; k < n; k++) {
        if (min > A[k]) min = A[k];
    }
    return min;
}
```

The time complexity of \( \text{minimum} \) is \( \Theta(n) \). A functional implementation to compute the maximum of a list might look like this:

```c
maximum :: (Ord a) => [a] -> a
maximum [] = error "maximum of empty list"
maximum [x] = x
maximum (x:xs) = maximum (x > maxTail) = x
otherwise = maxTail
where maxTail = maximum xs
```

Write and solve a recurrence equation that describes the time complexity of the \( \text{maximum} \) code.

One approach to compute the \( i \)th order statistic is to sort the list and report the \( i \)th element. What would the time complexity of this approach be? What are its advantages and disadvantages?

Randomized Partition & Selection

Using a randomized implementation of the Quicksort partition function, an average case linear-time algorithm can be developed for the selection problem.

Here’s one implementation of Quicksort’s \( \text{partition} \). Assume a sentinel \( (A[0] \leq \forall A[j]) \) to keep \( j \) within bounds. Check the C code compiles and executes correctly on test cases.

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

Exercise: Explain why this code has $O(n)$ time complexity. Exercise: Show how partition executes on the array below.

<table>
<thead>
<tr>
<th>position</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>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>-1</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>11</td>
<td>17</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>

A randomization heuristic swaps the head of the array with some randomly selected element. The randomizing goal is avoid bad partitioning.

#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;
    return partition(A, lo, hi);
}

This random partition computation can be completed in $O(n)$ time. Let’s use it to solve the selection problem in $O(n)$ in the average case.

Now, a randomized selection algorithm can be developed to compute the $i$-th order statistic.

#include <stdlib.h>

int randomSelect(int A[], int lo, int hi, int i) {
    if (lo == hi) { return A[lo]; } // if one element return it
    int p = randomPartition(A, lo, hi); // else select random partition
    int k = p - lo + 1; // count elements in left sub-array
    if (i == k) { return A[p]; } // if $i$-th found return it
    else if (i < k) { return randomSelect(A, lo, p-1, i); } // $i$-th in lo portion
    else { return randomSelect(A, p+1, hi, i-k); } // $i$-th in hi offset by $k$
}

Assume each pivot position is equally likely. Each recursive call is on a sub-array of size $k-1$ or $n-k$. The worst that can happen is to choose the largest in all cases, leading to the recurrence.

$$T(n) = n + \frac{1}{n} \sum_{k=1}^{n} T(\max\{k-1, n-k\})$$

As $k$ ranges from 1 to $n$ each term $T(n-1)$ down to $T([n/2])$ appears twice, and if $n$ is odd, $T([n/2])$ appears once. Therefore the recurrence can be simplified to

$$T(n) \leq n + \frac{2}{n} \sum_{k=[n/2]}^{n-1} T(k)$$
Apply the algebraic tricks: Use = in place of ≤; clear the fraction; replace \( n \) with \( n + 1 \); and subtract the equations.

\[
T(n) = n^2 + 2 \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k)
\]

\[
(n + 1)T(n + 1) = n^2 + 2n + 1 + 2 \sum_{k=\lfloor (n+1)/2 \rfloor}^{n} T(k)
\]

\[
(n + 1)T(n + 1) - nT(n) = 2n + 1 + 2T(n) \quad \text{(if \( n \) is even)}
\]

\[
(n + 1)T(n + 1) - nT(n) = 2n + 1 + 2(T(n) - T(\lfloor n/2 \rfloor)) \quad \text{(if \( n \) is odd)}
\]

Let’s assume \( n \) is even as the odd case looks less expensive.

Next, apply the analysis tricks: Let \( \langle T_0, T_1, T_2, \ldots \rangle \) be the sequence of time complexities for \texttt{randomSelection}. Let \( G(z) = \sum T_n z^n \) be the generating function for this sequence.

\[
(1 - z)G'(z) = \sum_{n=0}^{\infty} nT_n z^{n-1} - z \sum_{n=0}^{\infty} nT_n z^{n-1}
\]

\[
= \sum_{n=0}^{\infty} (n + 1)T_{n+1} z^n - \sum_{n=0}^{\infty} nT_n z^n
\]

\[
= \sum_{n=0}^{\infty} [(n + 1)T_{n+1} - nT_n] z^n
\]

\[
= \sum_{n=0}^{\infty} [2n + 1 + 2T_n] z^n
\]

\[
= \frac{2}{(1 - z)^2} + 2G(z)
\]
The Fast Discrete Fourier Transform

The motivating problem is spectral analysis: Think of sampling a continuous time signal $a(t)$ at $N$ uniformly spaced times. The discrete Fourier transform (DFT) map these samples to values in the frequency domain, linear combinations of sine and cosine functions.

The discrete Fourier transform (DFT) transforms a sequence $\bar{a} = (a_0, a_1, \ldots, a_{N-1})$ of $N = 2^n$ complex numbers into another sequence $\bar{y} = (y_0, y_0, \ldots, y_{N-1})$. The computation for $y_k$ is:

$$y_k = \sum_{0 \leq j < N} a_j \left( e^{-2\pi i k/N} \right)^j \quad (k = 0, \ldots, N - 1)$$

The discrete Fourier transform can be represented as a matrix equation where $\omega = e^{-2\pi i/N}$ (Use $\omega_N$ when syntactic sugar is needed).

$$\bar{y} = W\bar{a} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{bmatrix}$$

Computing $\bar{y}$ using matrix–vector multiplication requires not only evaluation the expressions $\omega^k$ for $k = 0, \ldots, (N - 1)^2$, but also $N(N-1)$ additions and $N^2$ multiplications.

To see if we can do better, let’s start with the two point Fourier transform, where $\omega = e^{-\pi i} = -1$.

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{(The Fourier transform of } [a_0, a_1] \text{ is their sum and difference.)}$$

Next, consider the four point Fourier transform matrix, where $\omega = e^{-\pi i/2} = -i$. Since $\omega^2 = -\omega^0 = -1$ any power $\omega^m$ can be reduced to $\pm \omega^r$ where $r = m \mod 2$ and the sign depends on the parity mod 4 of $m$. Lastly, permute the columns putting the even columns first. See how this partitions the matrix into four quadrants.

$$W = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & -\omega^0 & -\omega^1 \\ \omega^0 & -\omega^0 & \omega^0 & -\omega^0 \\ \omega^0 & -\omega^1 & -\omega^0 & \omega^1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -i & -i \end{bmatrix}$$

Apply this last matrix operator to even–odd samples $[a_0, a_2, a_1, a_3]$. This computes, in steps, the DFT of $[a_0, a_2]$:

$$y_0' = a_0 + a_2, \quad y_2' = a_0 - a_2$$

and

$$y_1' = a_1 + a_3, \quad y_3' = a_1 - a_3$$
the DFT of \([a_1, a_3]\). Then, combine the results.

\[
[y_0, y_1, y_2, y_3] = [y_0' + y_1', y_2' + \omega y_3', y_0' - y_1', y_2' - \omega y_3']
\]

This is the Fourier transform, check that it is so.

\[
y_0 = y_0' + y_1' = (a_0 + a_2) + (a_1 + a_3)
y_1 = y_2' + \omega y_3' = (a_0 - a_2) - i(a_1 - a_3)
y_2 = y_0' - y_1' = (a_0 + a_2) - (a_1 + a_3)
y_3 = y_2' - \omega y_3' = (a_0 - a_2) + i(a_1 - a_3)
\]

The eight point case further reveals the pattern. Now, \(\omega = e^{-2\pi i/8}\) and \(\omega^4 = -1\). Therefore, any power \(\omega^m\) can be reduced to \(\pm \omega^r\) where \(r = m \mod 4\) and the sign depends on the parity \(\mod 8\) of \(m\). Lastly, permute the columns putting the even columns first. See how this partitions the matrix into four quadrants.

\[
W = \begin{bmatrix}
\omega^0 & \omega^0 & \omega^0 & \omega^0 \\
\omega^0 & \omega^1 & \omega^2 & -\omega^0 \\
\omega^0 & -\omega^0 & \omega^0 & -\omega^2 \\
\omega^0 & -\omega^1 & -\omega^2 & \omega^0
\end{bmatrix}
\mapsto \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & -i & i \\
1 & -1 & i & i \\
1 & i & -1 & i
\end{bmatrix}
\]

The Fourier transform of 8 samples can be computed by constructing the Fourier transform of \([a_0, a_2, a_4, a_6]\) and \([a_1, a_3, a_5, a_7]\) and then combining the results

\[
[y_0, y_2, y_4, y_6] + [y_1, \omega y_3, \omega^2 y_5, \omega^3 y_7], [y_0, y_2, y_4, y_6] - [y_1, \omega y_3, \omega^2 y_5, \omega^3 y_7]
\]

The upshot is: To compute the Fourier transform of \(N = 2^n\) samples, compute the Fourier transform of \(N/2\) samples twice. Then combine the results

\[
\bar{y}_{\text{even}} + \bar{\omega} \oplus \bar{y}_{\text{odd}}, \bar{y}_{\text{even}} - \bar{\omega} \oplus \bar{y}_{\text{odd}}
\]

where

\[
\bar{\omega} \oplus \bar{y}_{\text{odd}} = [y_1 + \omega y_3 + \omega^2 y_5 + \cdots + \omega^{N/2} y_{N-1}]
\]
Cooley and Tukey (Cooley and Tukey, 1965) are credited with discovering how to make the discrete Fourier transform fast by exploiting facts about polynomials and roots of unity. Their fast discrete Fourier transform (FDFT) computes the values \( y_k, k = 0, \ldots, N - 1 \) in \( O(N \lg N) \) steps. A huge improvement.

To better understand the FDFT some basic facts are needed. Consider the problem in terms of polynomials. Let

\[
y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_{N-1} x^{N-1}
\]

be a polynomial of degree \( N - 1 \). The polynomial can be split by the parity (even or odd) of its coefficients.

\[
y_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{N-2} x^{N/2-1}
\]

and

\[
y_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{N-1} x^{N/2-1}
\]

and \( y(x) \) can be reconstructed by

\[
y(x) = y_{\text{even}}(x^2) + xy_{\text{odd}}(x^2)
\]

The DFT calls for evaluating \( y(x) \) at \( x = \omega^k = e^{-2\pi i k / N} \) for \( k = 0, \ldots, N - 1 \). This reduces to evaluating \( y_{\text{even}}(x^2) \) and \( y_{\text{odd}}(x^2) \) at \( x^2 = \omega^k = e^{-4\pi i k / N} \) for \( 0 \leq k < N/2 \).

The values \( \omega^k \) are \( N \)-th complex roots of unity: They lie on the unit circle in the complex plain.

\[
\left( \omega^k \right)^N = e^{-2\pi i k} = \cos 2\pi k - i \sin 2\pi k = 1 \quad (0 \leq k < N - 1)
\]

The principal \( N \)-th root of unity is

\[
\omega = \omega_N = e^{-2\pi i / N}
\]

Further,

\[
\omega^{N/2} = \left( e^{-2\pi i / N} \right)^{N/2} = e^{-\pi i} = -1
\]

\[
\omega^{k+N/2} = e^{-2\pi i (k+N/2)} = e^{-2\pi i k} e^{-\pi i} = -\omega^k
\]

\[
\left( \omega^{k+N/2} \right)^2 = e^{-2\pi i (2k+N)} = e^{-2\pi i (2k)} = (\omega^k)^2
\]

Developing the Cooley–Tukey FFT in Haskell

Here’s Haskell code\(^5\) for the fast discrete Fourier transform. The outline for the `fft` module is simple: Import other needed modules, define other functions as needed, and include a `main` function for IO and testing.

\[^5\] I am indebted to a former student for the Haskell code structure.

\[\text{FFT (50a)}\equiv\]

```haskell
module FFT (fft) where

import external modules 50b

(Auxiliary functions 51a)

(Main function 52)
```

The `Complex` data type is needed for constructing complex numbers and performing arithmetic on values of this type.

\[\text{Import external modules 50b)}\equiv\]

```haskell
import Data.Complex
```
Looking forward, here are descriptions of the functions that will be needed. The value \( n \) is what has been called \( N \) previously, a quirk of Haskell here.

\[\textbf{Auxiliary functions}\]

\( \langle \text{Compute a root of unity } e^{(-2\pi k/n)} \rangle \equiv \)
\( \langle \text{Split the list into even and odd lists} \rangle \equiv \)
\( \langle \text{Combine the lists} \rangle \equiv \)
\( \langle \text{Cooley-Tukey FDFT} \rangle \equiv \)

The \textit{mkPolar} function maps an amplitude \( r \) and phase \( \theta \) to complex number \( re^{i\theta} = r(\cos \theta + i \sin \theta) \). For roots of unity, the amplitude is \( r = 1 \), and the phase is \( \theta = -2\pi jk/n \).

\[\textbf{Compute a root of unity } e^{(-2\pi k/n)} \equiv \]
\( \text{cmpExp } n \ k \ j = \text{mkPolar } 1 \ (-2 \ast \ pi \ast (\text{fromIntegral } k) / (\text{fromIntegral } n)) \ast j \)

To apply the recursion, the list of samples must be split into two pairs: The first list contains sample with even indexes; the second contains odd indexed samples.

\[\textbf{Split the list into even and odd lists} \equiv \]
\( \text{splitEvenOdd } [] = ([], []) \)
\( \text{splitEvenOdd } [x] = ([x], []) \)
\( \text{splitEvenOdd } (x:y:xs) = (x:xe, y:yo) \)
\( \text{where } (xe, yo) = \text{splitEvenOdd } xs \)

Combine the two lists into one. The recursion requires concatenating the sum and difference of the two lists:

\[\textbf{Combine the lists} \equiv \]
\( \text{combine } [] [] = [] \)
\( \text{combine } xs [] = xs \)
\( \text{combine } [] ys = ys \)
\( \text{combine } xs ys = \text{add ++ sub} \)
\( \text{where } \text{add} = \text{zipWith } (+) \ xs \ ys \)
\( \text{sub} = \text{zipWith } (-) \ xs \ ys \)

\[\textbf{Cooley-Tukey FDFT} \equiv \]
\( \text{fft } :: \{\text{Complex Double}\} \rightarrow \{\text{Complex Double}\} \)
\( \text{fft } [] = [] \)
\( \text{fft } [x] = [x] \)
\( \text{fft } xs = \text{combine } \text{even } \text{cmpOdd} \)
\( \text{where } (e, o) = \text{splitEvenOdd } xs \)
\( \text{even} = \text{fft } e \)
\( \text{odd} = \text{fft } o \)
\( \text{size} = \text{length } xs \)
\( \text{cmpOdd} = \text{zipWith } (\text{cmpExp } \text{size}) [0..] \ odd \)

\[\textit{Testing the code}\]

Okay, we’ve got some code, let’s test that it works as expected. Interactive tests are good for unit testing functions.
**Unit Tests with ghci**

The Haskell source *(should be)* here. The small sample script below will get you started. The script shows the results of tests on computing roots of unity, splitting and combining lists, and the computing the Fourier transform.

Recall the values \( \text{cmpExp} n \ k \ j = (\omega^k)^j = e^{-2\pi kj/n} \) are used in evaluating values \( y_k \) in the Fourier transform.

\[
y_k = \sum_{0 \leq j < n} a_j \omega^{kj}
\]

Therefore, \( \text{cmpExp} \ 2 \ 0 \ 1 = e^{-2\pi \cdot 1/2} = 1 \) is computed correctly, but there is round-off error in the imaginary part of \( \text{cmpExp} \ 2 \ 1 \ 1 \).

% ghci

GHCi, version (your version 

Prelude> :l FFT.hs
[1 of 1] Compiling Main ( FFT.hs, interpreted )
Ok, modules loaded: Main.

*FFT> \text{cmpExp} \ 2 \ 0 \ 1
1.0 :+ 0.0
*FFT> \text{cmpExp} \ 2 \ 1 \ 1
(-1.0) :+ (-1.2246467991473532e-16)
*FFT> \text{splitEvenOdd} \ [0,1,2,3]
([0,2],[1,3])
*FFT> \text{combine} \ [1,2] \ [3,4]
[4,6,-2,-2]
*FFT> \text{fft} \ [1,1]
[2.0 :+ 0.0,0.0 :+ 0.0]

**Integration Tests with ghc**

The topic of the class is analysis of algorithms. Theory says the time complexity is modeled by some function in \( \Theta(N \lg N) = \Theta(2^n n) \) where \( N = 2^n \) is the number of samples. Does running the \text{fft} code produce running times that fit this model?

The \text{noweb} source file \text{FFT.nw} is here. Running \text{notangle FFT.nw > FFT.hs} generates the Haskell source \text{FFT.hs} Running \text{noweave -index -delay FFT.nw > FFT.tex} generates the \LaTeX source \text{FFT.tex} The overall idea of the main function is simple: Read an integer \( n \); create a list of \( N = 2^n \) (random) complex numbers; compute their Fourier transform; and show the results.

\[
\langle \text{Main function} \rangle \equiv
\]

\[
\begin{align*}
\text{main} & : \text{IO}() \\
\text{main} & = \text{interact} (\text{show} \ . \ \text{fft} \ . \ \text{createList} \ . \ \text{read})
\end{align*}
\]
A list of \( N = 2^n \) random complex numbers will be generated. To do this, the \texttt{System.Random} module will be used.

\begin{verbatim}
53a  (Import external modules 50b) +≡
   import System.Random

53b  (Auxiliary functions 51a) +≡
   (Create a list of random complex numbers 53c)

53c  (Create a list of random complex numbers 53c) ≡
   createList :: Int -> [Complex Double]
   createList n = take (2^n) $ zipWith (:+) (randoms (mkStdGen 10)) (randoms (mkStdGen 10))
\end{verbatim}

References

**Numerical Analysis**

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

---

John Tukey

Much of early computing revolved around numerical algorithms that produce values that approximate the True value. Approximation 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 an $x$ such that $f(x) = 0$. In particular, we’ll illustrate the method to approximate $\sqrt{m}$.

**Newton’s Method**

A classic numeric problem is: Given a function $f : \mathbb{R} \rightarrow \mathbb{R}$, compute roots (zeros) $x$ such that $f(x) = 0$. In Western society, Newton’s name is attached to one method for computing roots. But the roots of the method can be traced to ancient mathematicians of Mesopotomia, the region of modern day Iraq and Iran. Clay tablets from Mesopotamia, dated between 1800 B.C. to 1600 B.C., showing how to approximate $\sqrt{2}$ and perform other arithmetic operations have been found. Algorithms that compute numbers like $\sqrt{2}$ or $\pi$ do not terminate: There is no finite positional notation for these numbers.

Newton is credited with generalizing the nascent idea to a broad class of functions. An example function is $f(m) = x^2 - m$, which has root $x = \sqrt{m}$. The idea is: Let $x$ be a zero of function $f$. Use Taylor’s theorem. Solve for $x$, and discard the second order error.

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

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

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

For the function $f(x) = x^2 - 2$ the approximation reduces to

$$x \approx x_k - \frac{x_k^2 - 2}{2x_k} = \frac{x_k + 2/x_k}{2}$$

The approximation is the average of $x_k > 0$ and $2/x_k$. If $x_k$ is too small: $x_k^2 - 2 < 0$, then $2/x_k$ is too large:

$$((0 < 2 - x_k^2) \Rightarrow (0 < 2/x_k^2 - 1)) \Rightarrow ((2/x_k)^2 - 2 > 0)$$

The average of $x_k$ and $2/x_k$ is a better approximation.

- Start with $h_0 = 1$ as an initial approximation to $\sqrt{2}$
- Clearly 1 is too small, as the Babylonians could easily measure
- But, if $1 = \sqrt{2}$, then $1 \cdot 1 = \sqrt{2} \cdot \sqrt{2} = 2$ and so $2/1 = \sqrt{2}$
- As it is, $2/1$ is too large
• The average of the under estimate 1 and the over estimate 2/1 provides a better approximation to $\sqrt{2}$, call this
\[ h_1 = \frac{1}{2} \left( h_0 + \frac{2}{h_0} \right) = \frac{1}{2} \left( 1 + \frac{2}{1} \right) = \frac{3}{2} \]
• But $h_1 = 3/2$ is too large, as the Babylonians could measure
• But, if $3/2$ were the exact square root, then $2/(3/2) = 4/3$ would equal $\sqrt{2}$
• As it is, $4/3 \approx 1.333 \cdots$ is too small
• The average of the over estimate 3/2 and the under estimate 4/3 will provide a better approximation
\[ h_2 = \frac{1}{2} \left( h_1 + \frac{2}{h_1} \right) = \frac{1}{2} \left( \frac{3}{2} + \frac{4}{3} \right) = \frac{17}{12} \approx 1.4166 \cdots \]
• But $h_2 = 17/12$ is too small
• The Babylonians carried out this iteration more times computing the $\sqrt{2}$ accurately to at least 9 decimal places
• That is, they next computed the average of $h_2 = 17/12$ and $2/h_2 = 24/17$
\[ h_3 = \frac{1}{2} \left( h_2 + \frac{2}{h_2} \right) \]
\[ = \frac{1}{2} \left( \frac{17}{12} + \frac{24}{17} \right) \]
\[ = \frac{1}{2} \left( \frac{289 + 288}{17 \times 12} \right) \]
\[ \approx 1.41421568628 \cdots \]

To generalize the $\sqrt{2}$ method, pretend you want to compute $\sqrt{m}$. This is equivalent to computing a solution to the equation
\[ x^2 - m = 0 \]

Consider the recurrence equation
\[ x_k = \frac{x_{k-1} + m/x_{k-1}}{2}, \quad k \geq 1 \] (19)

Given an initial value $x_0$, equation 19 can be used to generate a sequence
\[ \langle x_0, x_1, x_2, \ldots \rangle. \]

If you pretend that $x_k$ converges to $x$ as $k$ goes to infinity, that is,
\[ \lim_{k \to \infty} x_k = x \]

Then $x$ satisfies the equations
\[ x = \frac{x + m/x}{2} \]
\[ 2x = x + m/x \]
\[ x = m/x \]
\[ x^2 = m \]
\[ x = \sqrt{m} \]
Implementing Newton’s method in Haskell

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 $x_k$.

56a \( (\text{Newton } 56a) \equiv \)

module Newton (newton) where

56b \( (\text{Auxiliary functions } 56b) \equiv \)

56c \( (\text{Newton’s method } 56c) \equiv \)

56d \( (\text{Apply the recursion } 56d) \equiv \)

56e \( (\text{Repeatedly apply f to x } 56e) \equiv \)

56f \( (\text{Test if successive values meet a relative tolerance } 57a) \equiv \)

56g \( (\text{Apply the tolerance test } 57b) \equiv \)

\[
\text{next} :: \text{Double} \to \text{Double} \to \text{Double}
\]

\[
\text{next } m \ 0 = \text{error} \ "\text{Division by zero}" \\
\text{next } m \ x = (x + m/x)/2
\]

We want to repeatedly apply \texttt{next} to some initial value and generate a list of \texttt{Doubles}. Let’s define repeatedly to be a function that applies a function \( f :: \text{Double} \to \text{Double} \) to itself repeatedly. An initial value (seed) \( a \) for \( f \) starts the iteration, generating an infinite list.

56f \( (\text{Repeatedly apply f to x } 56e) \equiv \)

\[
\text{repeatedly} :: (\text{Double} \to \text{Double}) \to \text{Double} \to [\text{Double}]
\]

\[
\text{repeatedly } f \ a = a : \text{repeatedly } f \ (f \ a)
\]
Although repeatedly does not terminate, it can be terminated once a computed value is \textit{close enough}. A common way to do this is to define a tolerance usually the \textit{machine epsilon} and declare that the last computed approximation is \textit{good enough} once it and the previous approximation are within the tolerance. 

The \textit{absolute difference} \(|x_k - x_{k-1}|\) between successive iterates is a measure of closeness. One way to terminate Newton’s iteration is to stop when the absolute difference is within the tolerance.

\[ |x_k - x_{k-1}| \leq \tau \]

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

Instead of computing until the difference of successive approximations approaches 0, it is often better to compute until the ratio of successive approximations approach 1. This measure of closeness is the \textit{relative difference} \(\left|\frac{x_{k-1}}{x_k} - 1\right|\). Some define the relative error as

\[ re_k = \frac{x_k - x}{x} \]

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

Newton’s method terminates when

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

The \textit{relative function} maps a tolerance \(\tau\) and a sequence to the first value in the sequence where the relative error is within tolerance.

57a \hspace{1cm} \langle \text{Test if successive values meet a relative tolerance 57a}\rangle \equiv

\begin{verbatim}
relative :: Double -> [Double] -> Double
relative tau (a:b:rest)
| abs (a/b-1) <= tau = b
| otherwise = relative tau (b:rest)
\end{verbatim}

Now we can express Newton’s method to compute the square root of \(m\) to within a relative error tolerance \(\tau\) starting with an initial guess \(x_0\) as the function \(\text{mysqrt}\).

57b \hspace{1cm} \langle \text{Newton’s square root 57b}\rangle \equiv

\begin{verbatim}
mysqrt x0 tau m = relative tau (repeatedly (next m) x0)
\end{verbatim}

Show how the code can be changed to compute other values. Test this on other functions, for instance \(f(x) = x^3 - 2\) to compute \(\sqrt[3]{2}\).
Convergence of Newton’s Method

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

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

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,

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

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

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

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

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

\[
x \approx x_k - \frac{f(x_k)}{f'(x_k)}
\]
Greedy Algorithms

Greedy Algorithm: Make a choice that seems best at the moment. This locally optimal choice may lead to a globally optimal solution. However, greedy algorithms don’t always work. Deciding how to make good local choices is key.

Combinatorial Optimization

Greedy algorithms are often used to solve combinatorial optimization problems. 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.

For a greedy algorithm to work, an optimal solution to a problem must have optimal solutions to sub-problems.

One way to learn about greedy algorithms is to studying examples.

Making Change

Working a cash register is a common example of greedy approaches. Raise your hand if you’ve understand the problem and its solution.

Pretend a cashier needs to give a customer $2.73 in change. How can the cashier give the customer the fewest coins and/or bills?

The abstraction is this: A value \( V \) to be represented in denominations \( d_0, d_1, \ldots, d_{n-1} \) as a sum

\[
V = a_0d_0 + a_1d_1 + \cdots + a_{n-1}d_{n-1}
\]

such that the number of “coins”

\[
a_0 + a_1 + \cdots + a_{n-1}
\]

is as small as possible

The greedy approach is very Euclidean algorithm-like:

1. Preconditions: Assume all the \( a \)’s are set to 0, and the denominations are sorted in decreasing order.
2. Find the largest denomination \( d_k \leq V \).
3. Divide \( d_k \) into \( V \)
   - If the remainder \( R = 0 \) stop with the quotient set to \( a_k \).
   - Otherwise, start again from step 2 with \( V \) replaced by the remainder \( R \).

Ignoring the preconditions, the algorithm takes at most \( n \) steps, the number of denominations.

Greedy coin changing computes the optimal (minimal number of coins) for some collections of denominations, but not for all. It is optimal for 1, 5, 10, 25, 50, 100, where the proof is by cases and induction on \( V \). It is not optimal for 1, 10, 21, 34, 70, 100. Consider 140.
Tape Storage of Files

Okay, it is old-school, but the problem: How to store files on tape helps illustrate greedy algorithms. Pretend there are \( n \) files to be 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, call this \textit{sequential access time}. The time 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 assess 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

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

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

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

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

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.

\textbf{Theorem 1.} File Access Minimization The average cost of file access is minimized when the files are stored from smallest to largest.

\textbf{Proof.} File Access Minimization Suppose that \( \pi \) is the optimal file storage permutation, but two consecutive files are out of order by length. Call them file \( k = \pi(i) \) and file \( j = \pi(i+1), k < j \). 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 \( L[k] \). Therefore, the average cost is changed by a negative amount: \((L[j] - L[k])/n < 0\). But, this contradict that the given order \( \pi \) was one giving minimal average cost for file access. \qed
Questions and Problems

1. Another greedy approach for tape storage is to store the longest file first. Compare this with the shortest file first heuristic.
Rational Knapsack Packing

Shortest first is a greedy approach: What can measure shortest? The rational knapsack problem offers several variant of shortest. The scenario is a set of provisions to place in a knapsack without overflowing its capacity $C$. Each resource has a weight and value. The objective is to fill the knapsack achieving maximum value. And, for the rational problem, fractions of provisions can be used.

Here’s an example: Pretend the knapsack has capacity $C = 10$ and there are three items with these weights and values.

$$ w_0 = 10, \ v_0 = 30; \quad w_1 = 5, \ v_1 = 20; \quad w_2 = 1, \ v_2 = 2 $$

And here are some ways to measure shortest:

1. Place lightest first. Place all of item 2, all of 1, and 4/10 of item 0.
   
   Constraint: $1 + 5 + \frac{4}{10} \leq 10$  
   Objective: $2 + 20 + \frac{4}{10} 30 = 34$

2. Place heaviest first. Place all of item 0 and no other items.
   
   Constraint: $10 \leq 10$  
   Objective: $30$

3. Place most valuable first. For this instance this approach is the same as heaviest first.

4. Greedy value:weight approach: The sorted value-to-weight ratios are

   $$ \frac{v_1}{w_1} = 4; \quad \frac{v_0}{w_0} = 3; \quad \frac{v_2}{w_2} = 2 $$

   Place all of item 1 and 5/10 of item 0.

   Constraint: $5 + \frac{5}{10} \leq 10$  
   Objective: $20 + \frac{5}{10} 30 = 35$

The optimal value is $20 + 15 = 35$, given by the greedy value-to-weight ratio approach.

Greedy algorithm for Rational Knapsack Packing

The algorithm below uses this greedy heuristic to solve the rational knapsack problem. Its running time is $O(n)$ if the ratios have be previously computed and sorted. If the ratios need to be sorted its time complexity is $O(n \lg n)$. Building a heap and using a priority queue may, in some cases, be less expensive than sorting all the ratios. The precondition is

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

```c
int knapsack(int *v, int *w, int n, int C)
{
    (Initialize rational knapsack local state)
    (While accumulated weight ≤ C and more items)
      (If all of the next item can be added)
        (Update local state)
      (Otherwise add a fraction of the next item)
    }
}```
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.

\[\text{Initialize rational knapsack local state} \equiv \]
\[
\text{int } k = 0, \quad V = 0, \quad W = 0;
\]
\[
\text{int } r[n];
\]
\[
\text{for } (\text{int } j = 0; \quad j < n; \quad j++) \quad \{ \quad r[j] = 0; \quad \}
\]

The while condition is this:

\[\text{While accumulated weight } \leq C \text{ and more items} \equiv \]
\[
\text{while } ((W < C) \quad \&\& \quad (k < n))
\]

Inside of the \text{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 \( C \).

\[\text{If all of the next item can be added} \equiv \]
\[
\text{if } (W + w[k] \leq C)
\]

When all of the next item fits, update every state value.

\[\text{Update local state} \equiv \]
\[
\text{r[k] = 1; \quad W = W + w[k]; \quad V = V + v[k]; \quad k = k + 1;}
\]

If not all of the next item fits, the fraction
\[
r_k = \frac{C - W}{w_k}
\]
determines how much of item \( k \) can be placed in the knapsack. Notice that
\[
W + r_k w_k = C
\]

\[\text{Otherwise add a fraction of the next item} \equiv \]
\[
\text{else } \{
\text{r[k] = } (C-W)/w[k]; \quad W = C; \quad \text{V = V + r[k]*v[k];} \quad \text{k = k + 1;}
\}
\]

**Questions and Problems**

1. What is the time complexity of the greed value-to-weight rational knapsack algorithm?
Multiprocessor Scheduling

The multiprocessor scheduling problem (MPS) can be described by the ideas below.

A problem instance: A set $T$ of $n$ tasks, where each task $t \in T$ has a positive integer length $l(t)$, and a positive integer $m$ of available processors $p$.

A feasible solution: A schedule $s$ for all tasks such that each processor executes only one task at a time and each task is scheduled on only one processor.

The objective function: The finishing time $f(s)$ of the last task in the schedule.

An optimal solution: The feasible solutions that minimizes $f(s)$.

This function problem can be reduced to a sequence of decision problems: Does schedule $s$ have finishing time less than or equal to a deadline $d$. Is $f(s) \leq d$? This decision problem is NP-complete. Vaguely, this means:

- If a schedule $s$ exists that finishes before or by the deadline $d$, a non-deterministic algorithm for it is:
  - Guess a such a schedule $s$ that meets the deadline
  - Test that the finishing time $f(s) \leq d$. This can be accomplished in polynomial time.
- The complementary problem seems harder: If no schedule exists every possible schedule must be checked to prove the deadline is missed.
  - A schedule can be imagined as a function from $\mathbb{Z}_n$ to $\mathbb{Z}_m$.
  - The number of schedules is then the number of such functions: What is an expression for this count?
  - Since there are an exponential number of schedules for two or more processors, the complementary problem is hard.

Later, let’s make this more formal by reducing a known NP-complete problem (Partition) to MSP. Thus, if MSP could be solve in polynomial time, then Partition could be solve in deterministic polynomial time, which does not appear likely.

Greedy algorithm for Multiprocessor Scheduling

A greedy approach often gives acceptable results: Schedule tasks feasibly by longest processing time. Consider the tasks sorted in decreasing order. Assign each task to the next, in time, free processor. For example, with 3 processors and 7 jobs whose lengths are

$\{6, 6, 5, 5, 3, 3, 3\}$

Scanning the list, assign the first three tasks to processors $p_0$, $p_1$, and $p_2$ in that order. For the next task, with length 5, assign it to processor 3, since it is the next, in time, free processor. Then the last three jobs are assigned to processor $p_0$, $p_1$, and $p_0$. Draw a processor-time task scheduled diagram showing this. Thus, the finishing time is 12.

However, an optimal schedule is: Processor 0 executes tasks 0 and 2; processor 1 executes tasks 1 and 3; and, processor 2 executes tasks 4, 5, and 6. with a finishing time of 11.

Here’s some pseudo-code that describes this greedy approach. The array $T[]$ keeps track of when a processor will be free.
sort the task lengths into decreasing order \((l(t_1) \geq l(t_2) \geq \cdots \geq l(t_n))\);

for (int \(i = 0; i < m; i++\) ) { \(T(i) = 0;\)

for (int \(j = 0; j < n; j++\) ) {

\(p = 0;\)

for (int \(i = 1; i < m; i++\) )

if \((T(i) < T(p))\) \(p = i;\)

assign \(t_j\) to processor \(p\) at time \(T(p)\);

\(T(p) = T(p) + l(t_j) ;\)

}

}

return \(\max\{T(0), \ldots, T(m)\}\)

Line 1 has time complexity \(O(n \lg n)\). Line 2 has time complexity \(O(m)\). The internal loop at line 5 has time complexity \(O(m)\): Each lines 6 to 9 executes in constant time. Therefore, the outer loop at line 3 has time complexity \(O(nm)\). The mathematics is a double sum

\[
\sum_{j=0}^{n-1} \sum_{i=1}^{m-1} c = \sum_{j=0}^{n-1} (m - 1)c = n(m - 1)c
\]

Let \(f(g)\) be the finishing time of the greedy schedule and let \(f(o)\) be the finishing time of an optimal schedule. The relative error is

\[
\frac{f(g) - f(o)}{f(o)} = \frac{f(g)}{f(o)} - 1
\]

By the pigeonhole principle, for any feasible schedule \(s\) has finish time

\[
f(s) \geq \frac{1}{m} \sum_{j=0}^{n-1} l(t_j)
\]

establishing a lower bound of the finishing time of the greedy algorithm.

It can be shown

\[
f(g) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) f(o)
\]

giving an upper bound on \(f(g)\) with relative error.

\[
\frac{1}{3} - \frac{1}{3m}
\]
Graph Coloring

The graph coloring problem (GCP) can be described by the ideas below.

A problem instance: A graph $G = (V, E)$ of $n$ vertices, where each $v \in V$ has a color $c(v)$ assigned to it from a set of $k$ colors.

A feasible solution: A proper vertex coloring $c(G)$ is where each edge $e = (u, v) \in E$ has different colors assigned to $v$ and $u$: $c(v) \neq c(u)$.

The objective function: The number of colors $n(c(G))$ in a feasible coloring of $G$.

An optimal solution: The feasible solutions that minimizes $n(c(G))$.

This function problem can be reduced to a sequence of decision problems: Does the graph have a coloring with $d$ or fewer colors? This decision problem is $NP$-complete.

- Describe a non-deterministic polynomial time algorithm that solves instances of the $k$-coloring problem.
- The complementary problem seems harder. How could the fact that no $k$-coloring exists be demonstrated? What would be the complexity of such a solution?

The $NP$-completeness of GCP can be made more formal by reducing a known $NP$-complete problem ($3$-SAT) to GCP. Thus, if instances of GCP could be solve in polynomial time, then instances of $3$-SAT could also be solve in deterministic polynomial time, which does not appear likely.

Greedy algorithm for Graph Coloring

Devise a greedy algorithm for GCP. Use your greedy idea to color the complete graph $K_5$ on 5 vertices and the complete bipartite graph $K_{4,4}$ on $4 \times 4$ vertices.

1. What is the minimum number of colors for $K_5$ and $K_{4,4}$?
2. Does your algorithm achieve these minimums?
3. Do different vertex processing orders change the number of colors your algorithm returns?

References
Dynamic Programming

Richard Bellman is credited for inventing the dynamic programming problem solving methodology (Bellman, 1957). A nutshell definition of dynamic programming is: Bottom-up computation with memorization. Solve small sub-problems first and store the results. When needed later, by a recurrence, these values are looked up. Problems that lend themselves to a dynamic programming characteristics such as:

- 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 not too large, and results from them can be saved in a memo.

Dynamic programming algorithms have the following features:

- A recurrence is implemented iteratively, from the bottom-up.
- 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: \( O(\phi^n) \), where the golden ratio \( \phi \) is about 1.618. But, the iterative algorithm is linear and computes \( F_n \) in \( O(n) \) steps. The table holds the last two values \( F_{n-2} \) and \( F_{n-1} \), which are initialized to 0 and 1 when \( n = 2 \). Dynamically update these values using the Fibonacci recurrence as the computation proceeds.

Polygon Triangulation

A polygon \( P \) with \( n \) vertices and \( n \) edges (\( n \geq 3 \)) is a finite collection of vertices \( v_0, v_1, \ldots, v_{n-1} \) lying in the Cartesian \((x, y)\) plane with edges \((v_i, v_{i+1})\), \( i = 0, 1, \ldots, (n-1) \) where \( v_n = v_0 \) to close the last side. Triangles, squares, and pentagons are common polygons.

A polygon is simple if no edge crosses another. A polygon is convex if the line between any two points in the polygon lies entirely within the polygon. All polygons here are simple and convex.

A chord \( v_i v_j \) is a line segment between two nonadjacent vertices, that is \( v_j \neq v_{i+1} \) and if \( j = n - 1 \) then \( v_j \neq v_0 \). A triangulation \( T = T(P) \) is a set of chords that partition a simple convex polygon \( P \) into disjoint triangles. Triangle fans and strips are two simple ways to triangulate a polygon. The fan on the left draw successive chords from a single vertex. The strip on the right zigzags back and forth across the polygon.

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; minimize variation in surface normals when the polygon is non-planar.
Minimal Polygon Triangulation

Assume a weight function \( w(\triangle) = w(v_i, v_j, v_k) \) is defined on triangles. Let \( T = T(P) \) be a triangulation of a polygon \( P \). The objective function is the sum of

Decision Problem: Does triangulation \( T(P) \) minimize the sum of weights

\[
\sum_{\triangle \in T(P)} w(\triangle)
\]

Function Problem: Find optimal triangulations \( T \), those that minimize the sum of weights

\[
\sum_{\triangle \in T} w(\triangle)
\]

Let’s develop a recursion that describes an optimal triangulation.

Polygon Triangulation Recursion

Let’s start by noting some useful facts.

1. Every triangulation of a \( P \) is a polygon with \( n \) vertices \( n - 3 \) chords and partitions the polygon into \( n - 2 \) triangles.
2. Each polygon edge belongs to one and only one triangle.
3. Each triangle has one or two polygon edges.

Let \( W(i, j) \) be the weight of the optimal triangulation of polygon \( P_{ij} = (v_{i-1}, \ldots, v_j), 1 \leq i \leq j \leq n - 1 \). That is,

\[
W(i, j) = \min_T \left\{ \sum_{\triangle \in T} w(\triangle) \right\}
\]

where \( T \) is a triangulation of \( P_{ij} \). We want to know the optimal value of \( W(1, n - 1) \) and the triangulation that produces it.

In any triangulation of \( P_{ij} = (v_{i-1}, \ldots, v_j) \) there must be one triangle \( (v_{i-1}, v_k, v_j) \) where \( i \leq k \leq j - 1 \). By considering each choice for \( k \), the problem of computing \( W(i, j) \) can be reduced to finding the minimum of \( W(i, k) + W(k + 1, j) + w(v_i, v_k, v_j) \), that is,

\[
W(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\min \left\{ W(i, k) + W(k + 1, j) + w(v_{i-1}, v_k, v_j) : i \leq k \leq j - 1 \right\} & \text{otherwise}
\end{cases}
\]

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 \( (v_0, v_1, v_2, v_3) \) with two triangulations who weights measure perimeters.
The perimeters of the four triangles are:

\[ w(v_0, v_1, v_2) = 8 \quad w(v_0, v_2, v_3) = 8 \]
\[ w(v_0, v_1, v_3) = 5 \quad w(v_1, v_2, v_3) = 5 \]

We want to compute values \( W(i, j) \) in table below.

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

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

\[ W(1, 2) = \min \{ W(1, k) + W(k + 1, 2) + w(v_0, v_k, v_2) : 1 \leq k \leq 1 \} \]
\[ = \min \{ 0 + 0 + 8 \} = 8 \]
\[ W(2, 3) = \min \{ W(2, k) + W(k + 1, 3) + w(v_1, v_k, v_3) : 2 \leq k \leq 2 \} \]
\[ = \min \{ 0 + 0 + 5 \} = 5 \]

Lastly, the value of \( W(1, 3) \) is

\[ W(1, 3) = \min \{ W(1, k) + W(k + 1, 3) + w(v_0, v_k, v_3) : 1 \leq k \leq 2 \} \]
\[ = \min \{ [W(1, 1) + W(2, 3) + w(v_0, v_1, v_3)], [W(1, 2) + W(3, 3) + w(v_0, v_2, v_3)] \} \]
\[ = \min \{ [0 + 5 + 5], [8 + 0 + 8] \} = 10 \]

The Trace back

Not only is the optimal weight sought, the triangulation should construct too. This can accomplished by recording the path to the optimal solution. Let \( S[i, j] = k \) if the optimal triangulation of \( P_{ij} \) includes the triangle \( v_{i-1}, v_k, v_j \)

```java
public void triangulate(Polygon poly) {
    // initialize local state
    int n = poly.countOfVertices();
    double W[n][n];
    double S[n][n];
    for (int i = 0; i < n; i++) { W[i][i] = 0; }
    for (int d = 2; d < n; d++) { // for every diagonal
        for (int i = 1; i < n-d+1; i++) { // for every row
            int j = i+d-1; // first column in a row
            W[i][j] = INFINITY;
            for (int k = i; i < j; k++) { // for every splitting at k
                int q = W[i][k] + W[k+1][j] + poly.weight(i-1, k, j);
                if (q < W[i][j]) { W[i][j] = q; S[i][j] = k; }
            }
        }
    }
}
```
What is the time complexity of triangulate? What is its space complexity?

This algorithm would be useful if its only application was triangulating polygons. The basic idea can be used to solve many problems. See (Sankoff and Kruskal, 1983) for an overview.

References


Matrix Chain Multiplication

The Matrix Chain Multiplication Problem (MCMP) can be described as a function problem: Find the parenthesization a matrix product 

\[ M_0M_1 \cdots M_{n-1} \]

to minimize the number of multiply and addition operations. Using a bound \( B > 0 \), the MCM decision problem asks: Does a given way to parenthesize \( M_0M_1 \cdots M_{n-1} \) exist to that uses \( B \) or fewer operations?

The number of ways to insert balanced parentheses into a matrix product \( M_0M_1 \cdots M_{n-1} \) is the Catalan number \( C_{n-1} \). For instance, parentheses can be place in \( M_0M_1M_2M_3 \) in \( C_3 = 5 \) ways:

\[
(M_0M_1)(M_2M_3), \quad ((M_0M_1)M_2)M_3, \quad (M_0(M_1M_2))M_3, \quad M_0(M_1M_2M_3), \quad M_0(M_1(M_2M_3))
\]

Recall, \( C_n = \binom{2n}{n}/(n+1) \), grows quickly, so a brute-force search is not feasible for large \( n \).

Inner Products

The matrix product \( AB = C \) is the inner product of each row in \( A \) with each column in \( B \).

```haskell
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

matrixMult :: Num a => [[[a]]] -> [[[a]]] -> [[[a]]]
matrixMult xs ys = [[innerProduct x y | y <- ys] | x <- xs]
```

When \( A \) is \( n \times m \) and \( B \) is \( m \times p \) there are \( np \) inner products and each inner product requires \( m \) multiplications and \( m-1 \) additions.

\[
C[i, j] = \sum_{0 \leq k < m} A[i, k]B[k, j], \quad 0 \leq i < n, 0 \leq j < p
\]

Therefore the time complexity of the basic matrix multiply algorithm is \( O(nmp) \) multiplications and additions.

To see that multiplication order really does matter, consider this example:

\( M_0 \) has dimensions 10 \( \times \) 100, \( M_1 \) has dimensions 100 \( \times \) 5, \( M_2 \) has dimensions 5 \( \times \) 50

There 2 ways to compute the product \( M_0M_1M_2 \):

\[
\begin{align*}
((M_0 \times M_1) \times M_2) & \quad \text{Cost: } (10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = 7500 \\
(M_0 \times (M_1 \times M_2)) & \quad \text{Cost: } (100 \cdot 5 \cdot 50) + (10 \cdot 100 \cdot 50) = 75000
\end{align*}
\]

Structure of the Optimal Parenthesization

Pretend you want to optimally compute \( M_0M_1 \cdots M_{n-1} \) where \( M_k \) is a \( p_k \times p_{k+1} \) matrix. Let

\[
d(i, j) = \text{optimal cost to compute the product } M_i \cdots M_j
\]
The optimal value \( d(0, n-1) \) and parenthesization are sought.

If the optimal parenthesization is \((M_0 \cdots M_k)(M_{k+1} \cdots M_{n-1})\), then the optimal cost is the optimal cost to compute \((M_0 \cdots M_k)\) plus the optimal cost to compute \((M_{k+1} \cdots M_{n-1})\) plus \(p_0 p_{k+1} p_n\), the cost to compute the product \((M_0 \cdots M_k) \times (M_{k+1} \cdots M_{n-1})\) with dimensions \((p_0 \times p_{k+1}) \times (p_{k+1} \times p_n)\). That is,

\[
d(0, n-1) = d(0, k) + d(k+1, n-1) + p_0 p_{k+1} p_n
\]

A priori which \( k \) to use is unknown, so each case \( 0 \leq k \leq n-2 \) is computed and the minimum taken. In general, the optimal cost \( d(i, j) \) to compute \( M_i \cdots M_j \) is defined by the recursion

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

To compute \( d(i, j) \) requires precomputing the values in row \( i \), columns \( k = i, \ldots, (j-1) \) and column \( j \), rows \( k = i + 1, \ldots, j \). This dependency can be fulfilled different orders: By column up from the diagonal or by subdiagonals in the upper triangle.

For the example matrices above, the dimensions are

\[
\vec{p} = \langle 10, 100, 5, 50 \rangle
\]

The main diagonal values in \( d(\cdot, \cdot) \) are all 0. Compute: \( d(0, 1) = 10 \cdot 100 \cdot 5 \), then \( d(2, 3) = 10 \cdot 100 \cdot 50 \), and finally, \( d(0, 2) = \min \{ 5000 + (10 \cdot 5 \cdot 50), 25000 + (10 \cdot 100 \cdot 50) \} \)

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

```c
for (int i = 0; i < n; i++) { d[i][i] = 0; } // initialize diagonal
for (int s = 1; s < n-1; s++) { // for each sub-diagonal
    for (int i = 0; i < n-1-s; i++) { // for each row in sub-diagonal
        j = i + s; // fix the column
        d[i][j] = INFINITY;
        for (int k = i; k < j; k++) { // apply the recurrence
            q = d[i][k] + d[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < d[i][j]) { d[i][j] = q; s[i][j] = k; }
        }
    }
}
return d[0][n-1] and s[1][];
```

The loop on \( k \) executes \( j - i \) times, but \( j - i = s \). Therefore, the loop on \( i \) has complexity \( s(n - 1 - s) \). Summing over \( s \) gives

\[
\sum_{1 \leq s \leq n-2} s(n - 1 - s) = \frac{(n-2)(n-1)^2}{2} - \frac{(n-2)(n-1)(2n-3)}{6} = \binom{n}{3} = O(n^3)
\]
Graph Algorithms

It is useful and interesting to explore graphs. A graph $G = (V, E)$ can represent many problems. I once had the epiphany that every problem had a graph representation. Now I think that may be a reach.

Trees

Trees are special! There are many species: Binary in particular. A binary Tree holding data of type $a$ can be defined as:

```haskell
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

There are three basic traversals of a binary tree.

1. **Preorder traversal:** Visit where you are, then go left, then right.
   ```haskell
   preorder :: Tree a -> [a]
   preorder EmptyTree = []
   preorder (Node root left right) = root : (preorder left) ++ (preorder right)
   ```

2. **Inorder traversal:** Go left, then home to the root, then right.
   ```haskell
   inorder :: Tree a -> [a]
   inorder EmptyTree = []
   inorder (Node root left right) = inorder left ++ (root : inorder right)
   ```

3. **Postorder traversal:** Go left, go right, then visit the root.
   ```haskell
   postorder :: Tree a -> [a]
   postorder EmptyTree = []
   postorder (Node root left right) = postorder left ++ postorder right ++ [root]
   ```

How to analyze these tree traversal? Assume there are $n$ nodes in a tree. Then each of these traversals have time complexity $O(n)$.

Depth-first search

Go deep. Depth-first search starts at some vertex $v$ in a directed graph $G = (V, E)$ and follows some path as far as possible. When no further progress is possible, the algorithm backtracks until a new path is found or all vertices have been visited. This generalizes the concept of a preorder traversal of a tree.

For the example directed graph, visit A, then B, then C (or D). From C visit A (detect cycle), backtrack to B and visit D, then backtrack to A. To complete depth-first search of the graph, another unvisited vertex must be explored.
Assume the vertices w adjacent to v are stored in a list L[v]. The pseudo-code might look like this.

```c
1. dfs(vertex v) {
   2.   v.mark = visited;
   3.   for (each vertex w on L[v]) if (w.mark == unvisited) { dfs(w); }
}
```

This looks like a complex recursion: It depends on the condition in line 4. But, once visited, the condition will never be True, so dfs is called only once on each vertex. The time spend is in traversing the list L[v] of edges from v to adjacent nodes w.

Cormen et al (Corman et al., 2009) give a more complete algorithm. They keep explicit track of predecessors of a node: \( v.\pi = u \) if \( v \) is on the adjacency list of a previously discovered vertex \( u \). They also keep track of the times when a vertex is discovered and finished.

```c
1. dfs(graph G) {
   2.   for (each vertex u ∈ G.V) { u.color = white; u.π = nil; }
   3.   time = 0;
   4.   for (each vertex u ∈ G.V) {
   5.     if (u.color == white) { dfsVisit(G, u); }
   6.   }
   7.}
8. dfsVisit(graph G, vertex u) {
   9.   time = time + 1;
  10.  u.d = time;
  11.  u.color = gray;
  12.  for (each vertex v ∈ G.Adj[u]) {
  13.    if (v.color == white) {
  14.      v.π = u;
  15.      dfsVisit(G, v);
  16.    }
  17.  }
  18.  u.color = black;
  19.  time = time + 1;
  20.  u.f = time;
}
```

The time cost of this implementation is \( \Theta(|V| + |E|) \). dfsVisit is called once for each vertex. In dfsVisit the for loop executes \( |Adj[u]| \) times, and the sum over all vertices is \( \Theta(|E|) \).

**Breath-first search**

**Topological Sort**

The vertices in a directed acyclic graph can be placed in linear order. To do this, modify dfs so that as each vertex is finished, it is place at the head of a list.

```c
1. topologicalSort(graph G) {
   2.   dfs(G);
   3.   for (each vertex v ∈ G.V) {
   4.     output vertices in order of decreasing finishing time
   5.   }
```
References

Backtracking Algorithms

Conceptually, backtracking algorithms perform depth-first searches. Backtracking algorithms are often used to solve constraint satisfaction problems. Constraints are a set of equations or inequalities that must be met. Usually, the set of feasible solutions is large. The goal is to find a feasible solution that minimizes or maximizes an objective function.

The Traveling Salesman & Hamiltonian Circuit Problem

A Hamiltonian tour or circuit in an undirected graph $G = (E, V)$ is a cycle that passes through each node exactly once. When the edges have non-negative weights, a traveling salesman wants to find the shortest tour. Name the nodes 0 through $n - 1$. Since the tour is a cycle, the first node is arbitrary, and might as well be 0. A tour is a permutation of $1, 2, \ldots, (n - 1)$, so there are $(n - 1)!$ possible tours. But, the graph is undirected, so a permutation $[0, 1, 2, 3, 4, 5]$ is equivalent to $[0, 5, 4, 3, 2, 1]$.

A dynamic programming algorithm for TSP was presented previously. Backtracking algorithms usually have a bounding function that allows the search space to be pruned.

An adjacency matrix for the above graph is:

\[
\begin{pmatrix}
0 & 1 & 2 & 3 & 4 \\
1 & \infty & 2 & 2 & 1 & 4 \\
2 & \infty & 2 & \infty & 3 & 3 \\
3 & \infty & 1 & \infty & 4 \\
4 & \infty & \infty & \infty &
\end{pmatrix}
\]

You can compute the cost of a tour by walking through the graph, for instance $0 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 0$ has cost $2 + 2 + 4 + 3 + 2 = 13$. A tour covers all rows and columns: $(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)$.

The adjacency matrix $M$ can be reduced in two steps to create a lower bound on the cost of a tour.

1. First subtract the minimum in each row from itself and other values in the row. Sum the amount subtracted.
2. Second, using this intermediate matrix, subtract the minimum in each column from itself and the other values in the column. Sum the amount subtracted and add it to the previous sum. Call this value $V(M)$. 

A dynamic programming algorithm for TSP was presented previously.
With $V(M) = (1 + 2 + 1 + 4) + (1 + 0 + 0 + 0) = 9$. The resulting matrix has two key properties:

- All entries are non-negative.
- Every row and every column contains at least one 0.

$V(M)$ is a lower bound on the cost of any tour. That is, $V(M) \leq C(H)$ where $C(H)$ is the cost of any tour $H$. To see this, let

$$H = [0x_1x_2 \cdots x_{n-1}]$$

be any Hamiltonian circuit (tour): $0 \mapsto x_1 \mapsto x_2 \mapsto \cdots \mapsto x_{n-1} \mapsto 0$. Essentially a cyclic permutation on the set $\{0, 1, 2, \ldots, (n-1)\}$ that begins with 0.

On this tour through the adjacency matrix, every entry

$$M[0, x_1], M[x_1, x_2], \ldots, M[x_{n-1}, 0]$$

will be visited. The sum of these entries is the cost of the tour $H$.

$$C(H) = \sum_{i=1}^{n} M[x_{i-1}, x_i] \quad x_0 = x_n = 0$$

For each row, subtract $M[x_{i-1}, x_i]$ from each entry in row $(i - 1)$. The total amount subtracted is the cost of the tour $C(H)$. Consider the tour $H = [0, 2, 4, 3, 1]$ of the graph, with cost $2 + 2 + 4 + 3 + 2 = 13$.

$\begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & \infty & 2 & 2 & 1 & 4 \\
1 & \infty & 2 & \infty & 3 & 3 \\
2 & \infty & 1 & 2 & \infty & 4 \\
3 & \infty & 4 & 3 & \infty & 4 \\
4 & \infty & 4 & 3 & 2 & 4 & \infty
\end{bmatrix}$

The resulting matrix has a 0 in every row and column. But, some of its entries are negative. This process reduced the matrix by more than the reduction that computes $V(M)$.

$$V(M) \leq C(H)$$

That is, $V(M)$ is a lower bound on any tour.

Suppose the cost of some tour has been computed, yielding a potential minimum value. Say $[0, 2, 4, 1, 3]$ which has cost $2 + 2 + 3 + 3 + 1 = 11$. Backtracking would then compute the cost of $[0, 2, 4, 3, 1]$ which has cost $2 + 2 + 4 + 3 + 2 = 13$. The first, shorter tour would be kept as the optimal so far.

Now suppose some other tour has been started, to see if a shorter tour can be found. For example $X = [0, 3, \ldots]$, with a cost $1 + \cdots$ is a partial solution. We want to extend it to completion. There are six completions, here are a first three:

$$[0, 3, 2, 1, 4], \quad [0, 3, 2, 4, 1], \quad [0, 3, 1, 2, 4], \quad \ldots$$
The adjacency matrix \(M\) can be reduced by eliminating visited rows and columns. Suppose the completion \(X = [0, 3, 2, \ldots]\) is explored.

- Strike the columns of all visited nodes, except the first, column 0, it is needed to get back.
- Strike the rows of all visited nodes, except the last, it is needed to go forward.
- Set \(M'[x, 0] = \infty\) for each visited node. A tour cannot return to an already visited node.

\[
M'(0, 3, 2) = \begin{pmatrix}
0 & 1 & 2 & \beta & 4 \\
0 & 2 & 2 & 1 & 4 \\
1 & 2 & \infty & 2 & 3 \\
\beta & \infty & \beta & 1 & \infty \\
4 & 4 & 3 & 2 & \infty \\
\end{pmatrix}
\]

The value of this reduced matrix is \(V(M') = (2 + 2 + 3) + (0) = 7\). It can be shown a bounding function is

\[
B(X) = V(M'(X)) + C(X)
\]

In this case,

\[
B(0, 3, 2) = 7 + 2 = 9
\]

To say \(B(X)\) is a bounding function means:

- \(B(H) = C(H)\) if \(H\) is a complete tour. For a tour \(H\), all rows and columns will have been stricken, \(M'\) will be an empty matrix, and \(V(M'(H)) = 0\).
- \(B(X) \geq C(X)\) for any feasible partial solution. That’s because \(V(M'(X)) \geq 0\). The upshot is: \(B(X)\) is a upper bound on the cost of any tour that starts with path \(X\).

In the example \(X = [0, 3, 2, \ldots]\), \(V(M'(X)) = (2 + 2 + 3) + (0) = 7\), \(C(X) = 1 + 1 = 2\), and \(B(X) = 7 + 2 = 9\).

**Exercise:** Extend the tour to \(X = [0, 3, 2, 4, \ldots]\). What are the values of \(C(X), V(M'(X))\) and \(B(X)\) for this path?
Backtracking Algorithms, Part 2

Backtracking algorithms can be used to solve constraint satisfaction problems. The $0\text{-}1$ Knapsack Problem is an example problem that can be solved by backtracking. (There are other approaches (greedy, dynamic programming) for this problem.) The Set Partition problem: Can a set of integers be partitioned into two non-empty subsets that have equal sums over their values. It is an $NP$-complete problem. Partition is a special case of the Knapsack problem: Assume each item's weight equals its value and $C = V = 1/2 \sum$ weights. Solving Knapsack in this special case solves Partition.

$0\text{-}1$ Knapsack Problem

Given a knapsack with capacity $C$, and a list of provisions (an inventory) $\langle p_k : k \in \mathbb{N} \rangle$ (the list could be unbounded) A provision $p$ is a 3-tuple

$$p :: (\text{String}, \text{Num}, \text{Num}) = (\text{name}, \text{weight}, \text{value})$$

Decision Problem: Given a value $V$, is there a subset $I$ of provisions such that

$$\sum_{p \in I} p_{\text{weight}} \leq C, \quad \text{and} \quad \sum_{p \in I} p_{\text{value}} \geq V$$

This decision problem is $NP$-complete.

Function Problem: Find the maximum value of $V$ over all feasible subsets of provisions. A subset is feasible if the sum of its weights does not exceed the capacity $C$ of the knapsack. This function problem is $NP$-hard.

A useful abstraction is to consider both summations to be over the entire inventory (list of provisions). This is accomplished using a bit vector

$$\langle b_0, b_1, b_2, b_3, \ldots \rangle$$

where a particular bit $b_k$ is set to 1 or 0 depending on whether or not provision $p_k$ is or is not placed in the knapsack. There are several things this reveals.

- There are $2^n$ bit vectors of length $n$ (or subsets of an $n$ element set $\mathbb{Z}_n = \{0, 1, \ldots, (n-1)\}$). Exhaustive search will take exponential time.

- Not all subsets need to be explored: Once a subset is infeasible so are all of its super-sets.

Here is a functional algorithm for the problem. An example inventory is given. The [[combs]] function that searches over all combinations of provisions from the inventory (subsets of it). The [[main]] function specifies the input and output. The [[combs]] function maps a list of provisions and a capacity to an ordered pair: a value and its feasible list of provisions.

As an initial condition, if the provision list is empty, then for any capacity, the returned value is 0 and the empty list.

Given an non-empty list of provisions, there are two possibilities: (1) If the weight of the provision at the head of the list is less than the capacity, then including the provision is feasible, otherwise (2) the provision cannot be in a feasible solution.
In the second case, return the result from the rest of the list and the given capacity. This is the case where $b_p$, the bit representing the provision, is 0. In the first case, provision may or may not be in the optimal feasible solution. So, compute both cases and return the one that is largest.

Functional 0–1 Knapsack

```plaintext
data Provision = (String, Num, Num)

inventory = [(
    "map", 9, 150), (
    "compass", 13, 35), (water", 153, 200),
    ("sandwich", 50, 160), ("glucose", 15, 60), ("tin", 68, 45),
    ("banana", 27, 60), ("apple", 39, 40), ("cheese", 23, 30),
    ("beer", 52, 10), ("cream", 11, 70), ("camera", 32, 30),
    ("tshirt"", 24, 15), ("trousers", 48, 10), ("socks", 4, 50),
    ("umbrella", 73, 40), ("towel", 18, 12), ("book", 30, 10),
    ("trousers", 42, 70), ("overclothes", 43, 75),
    ("notecase", 22, 80), ("sunglasses", 7, 20)]

combs [] = (0, [])
combs ((n,w,v):rest) cap
  | w <= cap = max (combs rest cap)
  | prepend (n,w,v) (combs rest (cap - w)))
  | otherwise = combs rest cap
where prepend (n,w,v) (value, list) = (value + v, (n,w,v):list)

main = do print (combs inventory 400)
```

Given a list of $n$ provisions [[combs]] always calls itself again with a list of size $n - 1$, and sometimes calls itself twice on the tail of the list with different capacities. Prepending a triple onto the current optimal value and list takes constant time. Therefore, in the worst case, the code’s time complexity can be modeled by the famous Mersenne recurrence

$$T(n) = 2T(n - 1) + 1, T(0) = 0$$

which has solution $T(n) = 2^n - 1$. A more general model would be

$$T(n) = 2T(n - 1) + c, T(0) = a$$

which has solution $T(n) = 2^a + (2^n - 1)c$.

Here is a C (pseudo-code) for an imperative implementation backtracking for the 0–1 knapsack problem.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *name;
    int weight;
    int value;
} provision;

provision items[] = [(
    "map", 9, 150),
    ("compass", 13, 35),
    ("water", 153, 200),
    ("sandwich", 50, 160), ("glucose", 15, 60), ("tin", 68, 45),
    ("banana", 27, 60), ("apple", 39, 40), ("cheese", 23, 30),
    ("beer", 52, 10), ("cream", 11, 70), ("camera", 32, 30),
    ("tshirt"", 24, 15), ("trousers", 48, 10), ("socks", 4, 50),
    ("umbrella", 73, 40), ("towel", 18, 12), ("book", 30, 10),
    ("trousers", 42, 70), ("overclothes", 43, 75),
    ("notecase", 22, 80), ("sunglasses", 7, 20)]
```
"sunglasses", 7, 20];

int cap; // capacity of knapsack
int n; // number of items
int X[n]; // current array of bits
int optBits[n]; // optimal array of bits
int optValue = 0;

bool isFeasible(provision *items) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + (X[i] * items[i].weight);
    }
    if (sum <= cap) { return true; }
    else { return false; }
}

bool betterValue(provisions *items) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + (X[i] * items[i].value);
    }
    if (sum > optValue) {
        optValue = sum;
        return true;
    }
    else { return false; }
}

int knapsack(provisions *items, int level) {
    if (level == n) {
        if (isFeasible(items)) {
            if (betterValue(items)) {
                optBits = X;
            }
        }
    }
    else {
        X[level] = 1;
        knapsack(items, level + 1);
        X[level] = 0;
        knapsack(items, level + 1);
    }
}
The above code has time complexity described by

\[ T(n) = 2T(n-1) + n\]
\[ = 2[2T(n-2) + (n-1)] + n\]
\[ \vdots \]
\[ = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i (n-i)\]
\[ = \sum_{i=0}^{n-1} 2^i (n-i)\]
\[ = n(2^n - 1) - \sum_{i=0}^{n-1} 2^i\]
\[ = n(2^n - 1) - (2 + 2^n(n-2))\]
\[ = 2^{n+1} - n - 2\]

Here is another imperative C algorithm for the 0–1 Knapsack. It uses dynamic programming and comes from Rosetta Code. It makes me not want to be a C programmer. This algorithm is pseudo-polynomial, that is, its time complexity is \(T(n) = O(nw)\) where \(w\) is the capacity of the knapsack.

Imperative Dynamic Programming 0–1 Knapsack Algorithm

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *name;
    int weight;
    int value;
} provision;

provision items[] = {
    "map", 9, 150,
    "compass", 13, 35,
    "water", 153, 200,
    ...
    "sunglasses", 7, 20
};

int *knapsack(provision *items, int n, int w) {
    int i, j, a, b, *mm, **m, *s;
    mm = malloc((n + 1) * (w + 1), sizeof (int));
    m = malloc((n + 1) * sizeof (int *));
    m[0] = mm;
    for (i = 1; i <= n; i++) {
        m[i] = &mm[i*(w+1)];
        for (j = 0; j <= w; j++) {
            if (items[i-1].weight > j) {
                m[i][j] = m[i-1][j];
            }
        }
    }
}```
```c
else {
    a = m[i-1][j];
    b = m[i-1][j-1] + items[i-1].weight + items[i-1].value;
    m[i][j] = a > b ? a : b;
}
}
s = calloc(n, sizeof(int));
for (i = n, j = w; i > 0; i--){
    if (m[i][j] > m[i-1][j]) {
        s[i-1] = 1;
        j -= items[i-1].weight;
    }
}
free(mm);
free(m);
return s;
}

int main () {
    int i, n, tw = 0, tv = 0, *s;
    n = sizeof(items) / sizeof(provision);
    s = knapsack(items, n, 400);
    for (i = 0; i < n; i++) {
        if (s[i]) {
            printf("%-22s %5d %5d\n", items[i].name, items[i].weight, items[i].value);
            tw += items[i].weight;
            tv += items[i].value;
        }
    }
    printf("%-22s %5d %5d\n", "totals:", tw, tv);
    return 0;
}
```

### Pruning and Bounding Functions

The functional backtracking algorithm for Knapsack prunes the search space by only exploring branches where \([w \leq \text{cap}]\). The imperative backtracking algorithm explores the entire search space, but the search can be pruned by a guard before each recursive call: Does the current weight plus the weight of the next provision not exceed the capacity?

Bounding functions provide more general approaches to pruning.

Let \(\overline{X} = (x_0, x_1, \ldots, x_{k-1}, \ldots)\) be a \(k\)-bit string representing a (partial) solution to a constraint satisfaction problem. Let

\[
C(\overline{X}) = \left( \sum_{i=0}^{k-1} p_i x_i \right) + \max \left\{ \sum_{i=k}^{n-1} w_i x_i : \sum_{j=0}^{n-1} w_j x_j \leq C \right\}
\]

That is, \(C(\overline{X})\) is the maximum value of feasible descendants (extensions) of \(\overline{X}\). It is the value of the currently
selected provisions plus the largest value over the set of feasible extensions (descendants) of $\vec{X}$. In particular, if $|\vec{X}| = n$, then $\vec{X}$ is a feasible solution, $C(\vec{X})$ is its value, but $\vec{X}$ may not be optimal. Also, if $|\vec{X}| = 0$, then $C(\vec{X})$ is the optimal value of the problem.

Bounding Function Properties

A bounding function $B$ is any function defined on variable length bit strings such that

- If $\vec{X}$ is a feasible solution, the $C(\vec{X}) = B(\vec{X})$
- For all partial feasible solutions, $C(\vec{X}) \leq B(\vec{X})$

If such a bounding function $B()$ can be found, and if at any point of the computation $B(\vec{X}) \leq \text{[optValue]}$ holds, then no extensions of $\vec{X}$ can lead to an optimal solution. And, searching descendants of $\vec{X}$ can be pruned.

Computing $C(\vec{X})$ is expensive when $\vec{X}$ has many descendants. The bounding function $B()$ should be much easier to compute. And, we want $B()$ be a good approximation of $C()$.

One trick that can lead to discovering a bounding function is to find a simpler, easier to solve, related problems. A natural approximation to the 0–1 Knapsack problem is the Rational Knapsack problem (RK)

Recall, the greedy approach to the RK problem: Given a list of items, sort them in descending order in their value-to-weight ratios. An item with value 10 and weight 3 is worth more than an item with value 10 and weight 4. This sort only needs to be done once. Assume its time complexity is $O(n \lg n)$.

The greedy algorithm places items in the knapsack in order, one at a time as long as they fit. Some fraction of the last item might need to be used to fill, but not overfill, the knapsack. The time complexity is $O(n)$. See the notes on Greedy algorithms.

Let $\vec{X} = \langle x_0, x_1, \ldots, x_{k-1}, \ldots \rangle$ be a string of k-bit strings representing a (partial) solution to a Knapsack problem. Let $R(k, C')$ be the optimal solution to the Rational Knapsack problem with capacity $C'$, over all rational descendants of $\vec{X}$, that is, $\langle x_k, x_{k+1}, \ldots, x_{n-1} \rangle$ where the values of $x_j \in \mathbb{Q}$, the set of rationals.

Define a bounding function by

$$B(\vec{X}) = \sum_{i=0}^{k-1} x_i p_i + R \left( k, C - \sum_{i=0}^{k-1} x_i w_i \right) = CV + R(k, C - CW)$$

where $CV$ is the current value and $CW$ is the current weight.

That is, $B(\vec{X})$ is the value selected provision from 0 to $k - 1$, plus the value that can be gained from the remaining provisions using the remaining capacity and rational $x$'s. When all of the $x$'s are restricted to 0 or 1, then $C(X) = B(X)$. Also, since rational $x$'s yield more freedom (choices), $C(X) \leq B(X)$

Here’s an example from (Stinson, 1987). Assume there are 5 items with weights

$\vec{W} = \langle 11, 12, 8, 7, 9 \rangle$

and values

$\vec{V} = \langle 23, 24, 15, 13, 16 \rangle$

Pretend the knapsack’s capacity of $C = 26$. The weights and values are sorted by value-to-weight ratio:

$\langle 23/11, 24/12, 15/8, 13/7, 16/9 \rangle \approx \langle 2.09, 2, 1.875, 1.857, 1.77 \rangle$
The search space tree shown below and explained after the diagram. A node is a triple \((xs, B, CW)\), a list \([xs]\) of previously set bits, the value of the bounding function \(B\), and the current weight of the included items.

\[
\begin{array}{c}
[1, 52.6, 0] \\
[0, 50.1 < 51, 0] \\
[11, 52.5, 23] \quad [10, 51, 11] \\
[111, 31 > 26] \quad [100, 52.57, 23] \quad [101, 51, 19] \\
[1101, 30 > 26] \quad [1100, 52.33, 23] \quad [1011, 51, 26] \\
[11100, 52.57] \quad [10100, 47, 23] \quad [11010, 51, 26] \\
\end{array}
\]

Assume that the positive \(x = 1\) branch is explored first. The greedy algorithm first computes \(x's\): 1, 1, 3/8 to fill the knapsack. The bounding value is

\[
B([1]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625
\]

Now explore the 1 branch:

\[
B([1]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625, \quad CW = 11
\]

- \(B([1]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625, \ CW = 23\)
  - \(B([11])\) is infeasible: \(cw = 31 > 26 = C\), prune this branch
  - \(B([111]) = 23 + 24 + \frac{3}{8} \cdot 13 \approx 52.57, \ CW = 23\) (The left (down) branch \([1101]\) is infeasible: its weight is \(CW = 30\))
  - The search follows \([110] \rightarrow [1100] \rightarrow [11000]\): a feasible solution. Along this branch \(B\) is updated: \(52.57 \rightarrow 52.33 \rightarrow 47\) and a potential optimal value \([[optValue=47]]\) is set.

- Now explore the \([10]\) branch. \(B([10]) = 23 + 15 + 13 \approx 51, \ CW = 26\)
  - \(B([101]) = 51, \ CW = 26\)
    - \(B([1011]) = 51, \ CW = 26\)
      - \([10111]\) is infeasible: \(CW = 11 + 8 + 7 + 9 = 35 > 26 = C\)
· [10110] is feasible: \( B([10110]) = 51, CW = C \), and a new, better, potential optimal value \([opt-Value=51]\) is set.

* \( B([100]) = (23) + 13 + \frac{2}{3} \cdot 16 = 46.6 < 51, CW = 11 \). Since this value of \( B \) is less than the previously computed optimal value 51 this branch can be pruned.

- When the [0] branch is explored, we find 

\[
B([0]) = 0 \cdot 23 + 24 + 15 + \frac{6}{7} \cdot 13 \approx 50.14 < 51
\]

Since this value of \( B \) is less than the previously computed potential optimal solution 51 its entire sub-tree can be pruned.

**The Traveling Salesman & Hamiltonian Circuit Problem**

A Hamiltonian tour or circuit in an undirected graph \( G = (E, V) \) is a cycle that passes through each node exactly once. When the edges have non-negative weights, a traveling salesman want to find the shortest tour. Name the nodes 0 through \( n - 1 \). Since the tour is a cycle, the first node is arbitrary, and might as well be 0. A tour is a permutation of 1, 2, \ldots, \((n - 1)\), so there are \((n - 1)!\) possible tours. But, the graph is undirected, so a permutation \([0, 1, 2, 3, 4, 5]\) is equivalent to \([0, 5, 4, 3, 2, 1]\).

\[
M = \begin{pmatrix}
a & b & c & d \\
a & \infty & 3 & 5 & 8 \\
b & 3 & \infty & 2 & 7 \\
c & 5 & 2 & \infty & 6 \\
d & 8 & 7 & 6 & \infty \\
\end{pmatrix}
\]

An adjacency matrix can be reduced by subtracting the minimum in each row from itself and others in the row. Then, using this new matrix, subtract the minimum in each column from itself and the other values in the column. Keep a running tab of the amount subtracted: Call the value \( V(M) \).

The resulting matrix has:

- All non-negative entries
- Every row and every column contains at least one 0.

\[
\begin{array}{cccc}
 a & b & c & d \\
\infty & 0 & 2 & 5 \\
\infty & 0 & 2 & 1 \\
1 & \infty & 0 & 6 \\
2 & \infty & 0 & 0 \\
0 & 1 & \infty & 0 \\
\end{array}
\]

With \( V(M) = (3 + 2 + 2 + 6) + (1 + 0 + 0 + 4) = 18 \).

\( V(M) \) is a lower bound on the cost of any Hamiltonian circuit. To see this, let

\[
H = [0x_1x_2 \cdots x_{n-1}]
\]

be any tour: \( x_1x_2 \cdots x_{n-1} \) is a permutation on the set \( \{1, 2, \ldots, (n - 1)\} \). The adjacency matrix entries \( M[0, x_1], M[x_1, x_2], \ldots, M[x_{n-1}, 0] \) cover all rows and columns. The sum of these entries is the cost of the tour \( H \).

\[
C(H) = \sum_{i=1}^{n} M[x_{i-1}, x_i] \quad x_0 = x_n = 0
\]
For each row $i = 1, \ldots, n$, subtract $M[x_{i-1}, x_i]$ from each entry in the row.

References

The Set Partition Problem

Can a set integers be partitioned into two non-empty subsets that have equal sums over their values? This is an \(NP\)-complete problem. Given the two subsets, a linear algorithm adds their values and tests for equality. A variant would ask for the absolute value of the difference in sums to be as small as possible.

For example, the set \(S = \{3, 1, 5, 7\}\) can be partitioned as \(\{3, 5\}\) and \(\{1, 7\}\). While \(S = \{3, 1, 5, 8\}\) has only an unbalanced partition \(\{3, 5\}\) and \(\{1, 8\}\).

Stirling’s number of the second kind \(\{n\}_2\) counts the number of partitions of an \(n\)-element set into two non-empty subsets. It can be shown that \(\{n\}_2 = 2^{n-1} - 1\). Therefore, except for small examples, it is infeasible to use brute-force and test every partition.

A set of integers can be partitioned into two equal summing subsets only if the sum of all values is even. And, the sum over either subset must be one-half of this total.

A useful abstraction is to consider both summations to be over the entire set of integers. This is accomplished using a a bit vector \(\langle b_0, b_1, b_2, b_3, \ldots \rangle\) where a particular bit \(b_k\) is set to 1 or 0 depending on whether or not integer \(i_k\) is or is not in a subset. Not all subsets need to be explored: Once a subset is infeasible so are all of its super-sets.

I could not get my Haskell implementation of Set Partition to work correctly. Here is an implementation from StackOverflow.

```haskell
import Data.List (minimumBy)

minimumBy takes a comparison function and a list and returns the least element of the list by the comparison function. The list must be finite and non-empty. }
import Data.Ord (comparing)

comparing takes a function f and two values x and y. It compares \((f x) (f y)\)

Here is a sample set of integers to partition into equal sums.

\[
\text{set} = [90, 13, 153, 50, 15, 68, 27, 39, 23, 52, 11, 32, 24, 48, 4, 73, 18, 30, 42, 43, 22, 7]
\]

\[
\text{partitions generates all partitions of a list into an ordered pair of sublists. Given a list } (x:xs) \text{, compute the partitions of the tail } xs \text{ and call the result } ps = (ys, zs). \text{ Then place } x \text{ at the head of both } ys \text{ and } zs. \}
\]

\[
\text{partitions :: [a] } \rightarrow \text{ } \langle\text{[(a], [a])}\rangle
\]

\[
\text{partitions } [] = \langle\text{[[], []]}\rangle
\]

\[
\text{partitions } (x : xs) = \text{ let } ps = \text{ partitions } xs \text{ in } \langle(x : ys, zs) | (ys, zs) <- ps \rangle ++ \langle(ys, x : zs) | (ys, zs) <- ps \rangle
\]

\[
\text{unbalance sums two list and returns the absolute value of the difference}
\]

\[
\text{unbalance :: Num a } \Rightarrow \text{ ([a], [a]) } \rightarrow \text{ a}
\]

\[
\text{unbalance } (ys, zs) = \text{ abs } (\text{sum } ys - \text{sum } zs)
\]

\[
\text{balancePartition generates all two sublist partitions of a list. It composes this function with unbalanced of non-negative values and compares them returning the minimum by doing this. }
\]

\[
\text{balancedPartition :: (Num a, Ord a) } \Rightarrow \text{ [a] } \rightarrow \text{ ([a], [a])}
\]

\[
\text{balancedPartition } = \text{ minimumBy } (\text{comparing unbalance}) \cdot \text{ partitions}
\]

main = do print (balancedPartition set)
```
Wikipedia offers this algorithm for Set Partition. It returns True if there is a set partition and False otherwise.

```plaintext
find_partition(S) {
    n = |S|;
    k = sum(S);
    P = empty Boolean table of size \( \lceil k/2 \rceil \times (n+1) \);
    initialize top row \( P(0,x) \) of P to true;
    initialize leftmost column \( P(x, 0) \) of P, except for \( P(o, o) \) to false;
    for (i from 1 to \( \lfloor k/2 \rfloor \)) {
        for (j from 1 to n) {
            if \( (i-S[j-1]) \geq 0 \) { P(i, j) = P(i, j-1) \| P(i-S[j-1], j-1); }
            else { P(i, j) = P(i, j-1); }
        }
    }
    return P(\( \lfloor k/2 \rfloor \), n);
}
```

The time complexity is \( O(kn) \) which is pseudo-polynomial. The value of \( k \) can be exponentially large. But, it show Set Partition can be solved quickly in some instances. Such problems are often called “easy NP-hard” problems.
Complexity

Problems can be classified by at least two types: Decision and Function Problems.

- A decision problem is a question with a “yes” (True) or “no” (False) answer. For instance, are values in list \( \vec{X} \) ascending?
- A function problem requires a solution to the problem. Sort values in list \( \vec{X} \) into ascending order.

Function problems are typically harder than decision problems.

Good decision problems come from consistent theories. Example axiomatic theories (models) include: Arithmetic, Boolean (propositional) logic, and first-order (predicate) logic.

- A theory or model is consistent if it does not contain a contradiction: Only one of \( p \) or \( \neg p \) can have a proof.
- A model is complete when at least one of \( p \) or \( \neg p \) has a proof.
- A problem within a model is decidable if there is an algorithm that always answers the problem correctly.

There are some unfortunate results.

Gödel’s First Incompleteness Theorem.

“Every consistent formal proof system about a sufficiently rich arithmetic is incomplete.”

The theorem says there are statements about the arithmetic we learn as children that are True but have no proof.

Gödel’s Second Incompleteness Theorem

“If a formal proof system about a sufficiently rich arithmetic is consistent, there is no proof that it is consistent.”

Gödel’s second theorem says you cannot prove a consistent arithmetic is consistent.

Complexity can be defined by a hierarchy of problem classes. Some relationships among these classes are unknown. Here is an image from (Papadimitriou, 1994) (created by Sebastian Sardina) that shows the relationship among several complexity classes under common assumptions that have not been fully proven. Computational complexity is complex.
• The class $P$: Problems that can be correctly solved in polynomial time using a deterministic Turing machine. Exercise: List some problems that belong to $P$.

• The class $NP$: Problems that can be solved in polynomial time using a non-deterministic Turing machine.

Problem in class $NP$ are characterized by the idea that answers can be checked for correctness quickly, that is, in polynomial time on a deterministic Turing machine.

Exercise: List some problems that belong to $NP$, but are not known to be in $P$.

Every problem in $P$ is in $NP$ ($P \subseteq NP$). If a problem can be solved in polynomial time, it can be checked in polynomial time. Deterministic time is a subset of non-deterministic time.

The central question in complexity theory is:

Does $P \neq NP$.

Most people who understand the question believe, I believe, the answer is “no,” but there is no proof one way or another, so shrug your shoulders.

There are problems in $NP$ that are not known to be in $P$:

• Is a Boolean expression satisfiable? There are $2^n$ truth assignments when the expression has $n$ variables. To find satisfying truth assignment may require search over this large space. However, checking that a truth assignment satisfies an expression is straightforward.

The complementary problem: Is a Boolean expression unsatisfiable seem even harder: Every truth assignment must results in a False value. Unsatisfiable is in co-$NP$. 

• Given a set of integers is there partition that has equal sums? There are $2^{n-1} - 1$ ways to partition an $n$-element set into two subsets.
  But, given a proposed two-set partition, it is easy to check that their sums are identical.
• Does a subset of graph vertices cover the graph? That is, each edge is incident on some vertex in the subset. Again, lots of subsets. Easy to check that one is a vertex cover.
• Does a graph have a Hamiltonian circuit (tour)? Again, lots of paths. Easy to check that one is a Hamiltonian tour.

Reduction is a problem solving strategy. Suppose there is an algorithm $A$ that solves instances of problem $p$. Given a new problem $q$ suppose there is an algorithm $R$ that reduces instances of $q$ into instances of problem $p$. Then an instance of problem $q$ can be solved by applying $R$ to to get an instance of $p$ and apply $A$ to it to solve the original instance.

The reduction $R$ is always assumed to be deterministic and polynomial in time.

$$Q \xrightarrow{R} p \xrightarrow{A} \{\text{True, False}\}$$

There are several conclusions that can be drawn when a reduction $R$ exists.
• If $p \in P$, then $q \in P$.
• If $p$ is decidable, then $q$ is decidable.
• If $q \in NP$, then $p \in NP$.
• If $q$ is undecidable, then $p$ is undecidable (algorithm $A$ does not exist).

The concept of reductions leads to complete and hard complexity classes.

Complete and Hard Problems

Complete problems are the hardest in a problem class; hard problems may not even belong to the class.

Let $K$ be a complexity class.
• A problem $p \in K$ is $K$-complete if every problem $q$ in $K$ can be reduced to $p$ using some deterministic, polynomial time algorithm.
• A problem $p$ is $K$-hard if every problem $q$ in $K$ can be reduced $p$.

For the For the class $NP$ of non-deterministic polynomial time problems Euler’s diagram of the relationships between $NP$, $NP$-complete, and $NP$-hard depends on the answer to the question

$$P \overset{?}{=} NP$$
References

Complexity, Part 2

There are many \( NP \)-Complete problems, see (Garey and Johnson, 1979). A few are described here.

The Set Partition Problem

Can a set integers be partitioned into two non-empty subsets that have equal sums? An \( NP \) algorithm is: If a set partition exists, ask the oracle for one; check that the subsets partition the set and that they have equal sums.

This is an \( NP \)-complete problem. Given the two subsets, a linear algorithm adds their values and tests for equality. A variant would ask for the absolute value of the difference in sums to be as small as possible.

For example, the set \( S = \{3, 1, 5, 7\} \) can be partitioned as \( \{3, 5\} \) and \( \{1, 7\} \). While \( S = \{3, 1, 5, 8\} \) has only an unbalanced partition \( \{3, 5\} \) and \( \{1, 8\} \).

Stirling’s number of the second kind \( \binom{n}{2} \) counts the number of partitions of an \( n \)-element set into two non-empty subsets. It can be shown that \( \binom{n}{2} = 2^{n-1} - 1 \). Therefore, except for small examples, it is infeasible to use brute-force and test every partition.

A set of integers can be partitioned into two equal summing subsets only if the sum of all values is even. And, the sum over either subset must be one-half of this total.

A useful abstraction is to consider both summations to be over the entire set of integers. This is accomplished using a a bit vector

\[
\langle b_0, b_1, b_2, b_3, \ldots \rangle
\]

where a particular bit \( b_k \) is set to 1 or 0 depending on whether or not integer \( i_k \) is or is not in a subset. Not all subsets need to be explored: Once a subset is infeasible so are all of its super-sets.

I could not get my Haskell implementation of Set Partition to work correctly. Here is an implementation from StackOverflow.

```haskell
import Data.List (minimumBy)
minimumBy takes a comparison function and a list and
returns the least element of the list by the comparison function. The
list must be finite and non-empty. }
import Data.Ord (comparing)
comparing takes a function f and two values x and y. It compares (f x) (f y)
Here is a sample set of integers to partition into equal sums. ]
set = [90, 13, 153, 50, 15, 68, 27, 39, 23, 52, 11, 32, 24, 48, 4, 73, 18, 30, 42, 43, 22, 7]

| partitions generates all partitions of a list into an ordered pair
| of sub-lists. Given a list (x:xs), compute the partitions of the tail xs
| and call the result ps = (ys, zs). Then place x at the head of both ys and zs. }
partitions :: [a] -> [[a], [a]]
partitions [] = [[]]
partitions (x : xs) = let ps = partitions xs
                  in [(x : ys, zs) | (ys, zs) <- ps] ++ [(ys, x : zs) | (ys, zs) <- ps]
unbalance :: Num a => ([a], [a]) -> a
unbalance (ys, zs) = abs (sum ys - sum zs)
balancePartition generates all two sublist partitions of a list. It
composes this function with unbalanced of non-negative
Wikipedia offers this algorithm for Set Partition. It returns True if there is a set partition and False otherwise.

```java
find_partition(S) {
    n = |S|;
    k = sum(S);
    P = empty Boolean table of size \( \lfloor \frac{k}{2} \rfloor \times (n+1) \);
    initialize top row (P(0,x)) of P to true;
    initialize leftmost column (P(x, 0)) of P, except for P(0, 0) to false;
    for (i from 1 to \( \lfloor \frac{k}{2} \rfloor \)) {
        for (j from 1 to n) {
            if ((i-S[j-1]) >= 0) { P(i, j) = P(i, j-1) || P(i-S[j-1], j-1); }
            else { P(i, j) = P(i, j-1); }
        }
    }
    return P(\( \lfloor \frac{k}{2} \rfloor \), n);
}
```

The time complexity is \( O(kn) \) which is pseudo-polynomial. The value of \( k \) can be exponentially large. Set Partition can be solved quickly in some instances, when \( k \) is not too large. Problems of this type are called “easy NP-hard” problems.

The Circuit-Value Problem

Circuit-Value is one of the hardest problems in class P. Circuit-Value (CV) is \( P \)-complete. A Boolean circuit is a directed graph \( G = (V, E) \) where the nodes are gates of type True, False, \( \neg \), \( \land \), \( \lor \), or a variable name \( p \). There are no cycles in \( G \) so the edges have the form \((i, j)\) where \( i < j \) (A directed acyclic graph can be topologically sorted.) The in-degree of each gate is either 0, 1 or 2. Gates of type True, False, or a variable have in-degree 0. These gates are called inputs. Not gates have in-degree 1, and And and Or gates have in-degree 2. The output of a gate is computed by the standard Boolean operations. The value of the circuit is the output of the largest node, given truth assignments for each of the variables.

The CV problem is to compute the value of a circuit when there are no variable inputs. Here is an instance (numbers provide a naming and order for the gates).

```
\land (6)
\lor (5)
\land (3)
\neg (4)
\begin{align*}
\text{True (0)} & \quad \text{False (1)} & \quad \text{False (2)}
\end{align*}
```

A useful exercise would be to write the Boolean expression for the above graph.
To show CV is $P$-complete two things must be shown:

1. CV $\in P$. Describe a polynomial time, deterministic algorithm that computes the value of a circuit.
   
   For the example circuit, evaluate it by traversing it in topological order:
   
   $$(0, 3), (1, 3), (1, 4), (3, 5), (4, 5), (5, 6), (2, 6)$$
   
2. If $L \in P$, there is a polynomial time deterministic reduction $R$ of instances in $L$ to instances in CV.
   
   Given any input $x$, $R(x)$ must be a variable-free circuit such that $x \in L$ if and only if $R(x)$ is True. The construction of $R$ is complex and beyond my ability to summarize. See (Papadimitriou, 1994) for details. The fact that such a reduction $R$ exist is amazing.

Cook (Cook, 1971) and Levin (Levin, 1973) demonstrated the existence of $NP$-Complete problems. Karp (Karp, 1972) showed how to reduce many problems to known $NP$-complete problems. The book by Garey and Johnson (Garey and Johnson, 1979) documents innumerable hard problems. Before you undertake the task of finding a problem solution, know complexity of the problem.

Recall, colloquially a problem $L$ is in $NP$ if each instance $x$ has a short certificate that can be checked and returns True if $x \in L$ and False otherwise.

Satisfiability (SAT) is the classic $NP$-complete example: Given a Boolean expression $\phi$ (in conjunctive normal form) is $\phi$ satisfiable? That is, is there a truth assignment for the variables in $\phi$ that makes $\phi$ True.

Conjunctive normal form (CNF) means: $\phi$ is the And of Or phrases, called clauses. All expressions can be represented in CNF. The basics of the reduction is to replace conditionals $p \Rightarrow q$ with $\neg p \lor q$. Distribute Nots using De Morgan’s laws and eliminate double negations. And, finally, distribute Ors across Ands. Unfortunately the conversion (reduction) to CNF can produce expressions of size $2^n$ where $n$ is the number of propositions in the original expression. But, CNF captures the difficulty of the problem. And, the test for satisfiability reasonable: For the given truth assignment, every clause must be True.

SAT an example of a $P$-hard problem. Any (simple) problem in $P$ can be reduces to an instance of SAT. But SAT is not known to be in $P$, most likely $SAT \neq P$, but know one knows for sure (at this point in time).

Just as a Circuit-Value represents Boolean expressions with fixed truth assignments, Circuit-SAT represents Boolean expressions with variable input. That a reduction from SAT to Circuit-SAT exists may be clear. Explaining and understanding the details is best done off-line.

It is interesting that 2-SAT has a polynomial time algorithm. In an instance of 2-SAT each clause has two literals. On the other hand, 3-SAT is $NP$-complete.

The complement of $NP$ problems is an interesting class. The class co-$NP$ consist of problems whose complement are in $NP$.

Validity (VAL) is stronger than satisfiable: All truth assignments make $\phi$ True. A Boolean expression is valid if and only if its negation is unsatisfiable (always False). Unsatisfiable (UNSAT) is the complement of SAT.

Definition. A decision problem $L$ is in co-$C$ if its negation is in $C$.

For instance, the problem “are the values in list $\vec{X}$ in ascending order?” is in $P$. It’s complement, “are the values in list $\vec{X}$ not in ascending order?” is also in $P$.

It is not too hard to understand that co-$P = P$. If a decision problem can be decided “yes” or “no” in polynomial time, then so can its complement Think of feeding an problem instance into a machine (program). If the machine says “yes” the complement machine says “no” and vice versa.
On the other hand, it is not known where or not co-NP = NP, and most suspect, I suspect, would say no, they are not equal. Consider: Does no truth assignment satisfy an expression? Is there no Hamiltonian tour in a graph? Is there no non-empty subset of a given set integers that sum to 0? In each case, it seems every possible solution must be tested.

Problems that are NP-hard are interesting too. An NP-hard problem H is one where every L ∈ NP problem can be reduced to H. The halting problem is NP-hard. Halting is undecidable. There is no known certificate the demonstrates a program P will halt on input x. But, every problem in P or NP can be reduced to an instance of halting, not that that should be done.

References


Computability

The theory of computation focuses on several topics. One is the complexity of problems: If a problem can be solved, at what costs? Another topic is computational models: What are the mechanisms for expressing and implementing solutions?

Turing machine, $\lambda$-calculus, recursive functions, and other computational models have been proposed. All sufficiently strong models have been shown to be equivalent. They solve the same class of problems. There are many ways to define a Turing machine. Here is Papadimitriou’s (Papadimitriou, 1994) definition.

A deterministic 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). $\Sigma$ contains two special symbols: $\square$ and $\triangleright$, called blank and first, respectively.
4. $\delta$ is a transition function. It maps a (state, character) pair to a triple (next state, character, direction).

$$\delta : (K, \Sigma) \rightarrow (K \cup \{h, y, n\}, \Sigma, \{←, →, −\})$$

A non-deterministic Turing machine is the same, except $\delta$ is a choice relation mapping a (state, character) to subsets of (state, character, direction) triples.

The Universal Turing Machine

Turing showed (Turing, 1936) an important aspect of his machine: There is a single universal Turing machine $U$ that can simulate any other Turing machine. The input to the $U$ is a (machine description, input) 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.

Decision Problem: Given a Turing machine $M$ and its input $x$, does $M$ halt on $x$?

There is no algorithm that decides the halting problem. It may be possible to decide if a particular machine $M$ halts on a particular input $x$, but there is no algorithm that answers the halting problem for every instance of $M$ and $x$.

Define the halting language $\mathcal{H}$ is the set of all (machine, input) pairs such that $M$ halts on $x$.

$$\mathcal{H} = \{(M, x) : M(x) \neq \triangleright\}$$  \hspace{1cm} (\triangleright \text{ means does not halt})

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: Execute the diagonalMachine pseudo-code below. It assumes halting can be determined somehow. The program accepts the encoding of a machine $M$ as input. It runs $M$ on $M$, and loops forever if $M(M)$ halts. If $M(M)$ does not halt, then the diagonal machine halts. The main routine runs diagonalMachine on itself. The term diagonalization comes from Cantor’s proof that the real numbers are uncountable.

```
diagonalMachine(m: machine M)  
```
if (M(M) halts) then  { ↗; }  
else halt;  
}

main  diagonalMachine ( diagonalMachine );

Now consider the logic:

• If diagonalMachine(diagonalMachine) halts, then diagonalMachine(diagonalMachine) loops forever, that is, diagonalMachine(diagonalMachine) does not halt.

• On the other hand, if diagonalMachine(diagonalMachine) does not halt, then diagonalMachine(diagonalMachine) halts.

There is a contradiction in both cases. Therefore, the assumption that halting can be determined is wrong. There is no algorithm that correctly solves the halting decision problem:

“For all Turing machines M and for all inputs x, does M halt on x?”

The traditional proof that the halting problem is undecidable goes something like this:

Proof that the Halting Problem is undecidable. Pretend there is a Turing machine $M_H$ that decides the halting problem.

Use $M_H$ to construct a Turing machine $D$ that accepts the encoding of a Turing machine $M$ and runs $M_H$ on $(M, M)$. The behavior of $D$ is this:

1. $D$ does not halt if $M$ halts on $M$.
   
   If $(M(M) \neq \triangleright)$, then $D(M) = \triangleright$ ($D(M)$ produces no output).

2. $D$ halts if $M$ does not halt on $M$.

   If $(M(M) = \triangleright)$, then $D(M) \neq \triangleright$ ($D(M)$ reports $M$ does not halt on $M$).

Now consider $D(D)$.

1. $D$ does not halt (on input $D$) if $D$ halts on $D$

   If $(D(D) \neq \triangleright)$, then $D(D) = \triangleright$.

2. $D$ halts (on input $D$), if $D$ does not halt on $D$
If \((D(D) = \not\wedge)\), then \(D(D) \neq \not\wedge\)

This contradiction implies that the halting machine \(M_{\text{Hal}}\) cannot exist.

The Virus Problem

Evans presents a nice description of the virus problem (VP), see On the Impossibility of Virus Detection 6. A virus is a computer program that when executed will copy its own code into another program. A virus detection algorithm would act like this:

- **Input:** A description of a program \(P\) and its input \(x\).
- **Output:** If \(P(x)\) behaves like a virus (running it infects other files) output True. Otherwise, output False.

Assume there a program, virusDetect, that decides the Virus Problem (VP).

```plaintext
1. Boolean virusDetect(program P, input x) {
2.   if (P(x) behaves like a virus) then true;
3.   else false;
4. }
```

Here is the diagonalization argument that virus detection is undecidable. Assume there is a virus \(V\). Mimic the argument above about the halting problem. That is, write a program, call it \(D(P)\) if you like, that uses virusDetect, to create a contradiction.

The idea behind \(D\) is to use virusDetect to check if the input \(P\) is a virus, and then do the opposite. That is,

- If \(P\) is a virus, \(D(P)\) simply halts without executing a virus.
- If \(P\) is not a virus, \(D(P)\) executes virus \(V\).

```plaintext
1. D(program P) {
2.   if (virusDetect(P, P)) then halt;
3.   else V;
4. }
```

Now consider the execution of \(D\) on itself.

- If \(D(D)\) acts like a virus, (That is, if VirusDetect\((D, D)\)=true), then \(D(D)\) halts, That is, \(D(D)\) never executes a virus \(V\).
  (If \(D(D)\) is a virus, then \(D(D)\) does not act like a virus.)
- On, the other hand, if \(D(D)\) is a not a virus, then \(D(D)\) executes virus \(V\) acting like a virus.
  (If \(D(D)\) is not a virus, then \(D(D)\) acts like a virus.)

There are contradictions in both cases.

Reduction of HP to VP

Spend a few minutes going over Evans’ note On the Impossibility of Virus Detection.

Another way to show the virus problem is undecidable is to show HP reduces to VP. That is, if VP were decidable, then HP would be decidable. And, since HP is undecidable, VP must be also.

The main idea is to construct a program makeVirus that first executes any input program \(P\), and then serially, if \(P\) halts, executes virus \(V\).
makeVirus(program P) { P; V; }

Argue that program halt below decides the halting problem. Use this to conclude there cannot be a virus detection decider.

halt(program P) {
  if (virusDetect(makeVirus, P)) then true;
  else false;
}

• If $P$ halts, makeVirus will execute virus $V$ and virusDetect(makeVirus, P) will return True.
• On the other hand, if $P$ does not halt, then makeVirus will never execute $V$ and virusDetect(makeVirus, P) will return False.

In both cases the halting problem is correctly decided. Since this is not possible, there cannot be a virus detection program. There are a few wrinkles to iron out to fully nail down the argument. See 7.

References

