CS 341: Algorithms

Slides by Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Spring 2013

Table of Contents

Course mechanics - Spring 2014

- Instructors:
  - Mark Petrick
  - Email: mdtpetri@uwaterloo.ca
  - Office hours: Tuesday and Thursday 11:45–1:30 or by appointment in DC 3120
  - Section 1: T Th 10:00–11:20, MC 1056
  - Section 2: T Th 8:30–9:50, MC 1056

Course mechanics

- Teaching assistants:
  - Ankit Pat; userid: apat
  - Narges Fallahi; userid: nfallahi
  - Aaron Moss; userid: a3moss
- Come to class! Not all the material will be on the slides or in the text.
- You will need an account in the student.cs environment
  - If you don't have a student.cs account for some reason, get one set up in MC 3017.

Course website

- The course website can be found at

  `https://www.student.cs.uwaterloo.ca/~cs341/`

  - Syllabus, calendar, lecture slides, additional notes, assignments, announcements, policies, etc.
  - The website will be updated regularly
  - It is your responsibility to keep up with the information on the course website.

- Feedback is encouraged!

Piazza

- Discussion related to the course will take place on Piazza (piazza.com)
  - General course questions, announcements
  - Assignment-related questions
- You must keep up with the information posted there as well

Additional communication

- Some communication might be sent to your uWaterloo email address
  - Check uWaterloo email account regularly or have email forwarded to your regular account
- Use discussion forums in Piazza for questions of general interest
- Always use your regular uWaterloo email account for course-related email

Courtesy

- Please silence cell phones and other mobile devices before coming to class.
- Questions are encouraged, but please refrain from talking in class – it is distracting to your classmates who are trying to listen to the lectures and to your professor who is trying to think, talk and write at the same time.
- Carefully consider whether using your laptop in class will help you learn the material and follow the lectures.
- Do not play games, tweet, watch youtube videos, update your facebook page or use your laptop in any other way that will distract your classmates.

Course syllabus

- You are expected to be familiar with the contents of the course syllabus
- Available on the course home page
- If you haven't read it, read it after this lecture

Plagiarism and academic offenses

- We take academic offenses very seriously
- There is a nice explanation of plagiarism online:
  - `http://arts.uwaterloo.ca/arts/ugrad/academic_responsibility.html`
- Read this and understand it
  - Ignorance is no excuse!
  - Questions should be brought to instructor
- Plagiarism applies to both text and code
- You are free (even encouraged) to exchange ideas, but no sharing code or text

Plagiarism (2)

- Common mistakes
  - Excess collaboration with other students
    - Share ideas, but no design or code!
  - Using solutions from other sources (like for previous offerings of this course, maybe written by yourself)
- Possible penalties
  - First offense (for assignments; exams are harsher)
    - 0% for that assignment, -5% on final grade
  - Second offense
    - Expulsion is possible
- More information linked to from course syllabus

Grading scheme for CS 341

- Midterm (25%)
  - Friday, June 20, 2014 at 6:30–8:00 PM in M3 1006
- Assignments (30%)
  - Work alone
  - See syllabus for late and reappraisal policies, academic integrity policy, and other details
- Final (45%)

Assignments

- Assignments will be due at 3:00 PM on the due date
- Late submissions will be accepted up to 24 hours after due date
- There will be a penalty of 15% for accepted late submissions
- Multiple assignments can be submitted late
- No assistance will be given after the due date
- You need to notify your instructor before the due date of a severe, long-lasting problem that prevents you from doing an assignment

Required textbook

- Introduction to Algorithms, Third Edition, by Cormen, Leiserson, Rivest and Stein, MIT Press, 2009.
- You are expected to know
  - entire textbook sections, as listed on course website
  - all the material presented in class

Table of Contents

Analysis of algorithms

In this course, we study the **design** and **analysis** of algorithms. "Analysis" refers to mathematical techniques for establishing both the **correctness** and **efficiency** of algorithms.

Correctness: We often want a formal proof of correctness of an algorithm we design. This might be accomplished through the use of **loop invariants** and mathematical induction.

Analysis of algorithms (cont.)

Efficiency: Given an algorithm A, we want to know how efficient it is. This includes several possible criteria:

- What is the **asymptotic complexity** of algorithm A?
- What is the **exact number** of specified computations done by A?
- How does the **average-case** complexity of A compare to the **worst-case** complexity?
- Is A the most efficient algorithm to solve the given problem? (For example, can we find a **lower bound** on the complexity of **any** algorithm to solve the given problem?)
- Are there problems that cannot be solved efficiently? This topic is addressed in the theory of **NP-completeness**.
- Are there problems that cannot be solved by **any** algorithm? Such problems are termed **undecidable**.

Design of algorithms

"Design" refers to **general strategies** for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms. Also, we want to avoid using **ad hoc** algorithms that are hard to analyze and understand.

Here are some useful design strategies, many of which we will study:

divide-and conquer

greedy

dynamic programming

depth-first and breadth-first search

local search (not studied in this course)

linear programming (not studied in this course)

The "Maximum" problem

Problem

Maximum

Instance: *an array A of n integers,*

$$A = [A[1], \dots, A[n]].$$

Find: *the maximum element in A.*

The **Maximum** problem has an obvious simple solution.

Algorithm: *FindMaximum*$(A = [A[1], \dots, A[n]])$

$max \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do $\begin{cases} \textbf{if } A[i] > max \\ \quad \textbf{then } max \leftarrow A[i] \end{cases}$
return (max)

Correctness of FindMaximum

How can we formally prove that *FindMaximum* is correct?

Claim: At the end of iteration i $(i = 2, \ldots, n)$, the current value of max is the maximum element in $[A[1], \ldots, A[i]]$.

The claim can be proven by induction. The base case, when $i = 2$, is obvious.

Now we make an induction assumption that the claim is true for $i = j$, where $2 \leq j \leq n - 1$, and we prove that the claim is true for $i = j + 1$ (fill in the details!).

When $j = n$ we are done and the correctness of *FindMaximum* is proven.

Analysis of FindMaximum

It is obvious that the complexity of *FindMaximum* is $\Theta(n)$.

More precisely, we can observe that the number of comparisons of array elements done by *FindMaximum* is **exactly** $n - 1$.

It turns out that *FindMaximum* is **optimal** with respect to the number of comparisons of array elements.

That is, any algorithm that correctly solves the **Maximum** problem for an array of n elements requires **at least** $n - 1$ comparisons of array elements.

How can we prove this assertion?

The "Max-Min" problem

Problem

Max-Min

Instance: *an array A of n integers, $A = [A[1], \ldots, A[n]]$.*

Find: *the maximum and the minimum element in A.*

The **Max-Min** problem also has an obvious simple solution.

Algorithm: *FindMaximumAndMinimum*$(A = [A[1], \ldots, A[n]])$

$max \leftarrow A[1]$
$min \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do $\begin{cases} \textbf{if } A[i] > max \\ \quad \textbf{then } max \leftarrow A[i] \\ \textbf{if } A[i] < min \\ \quad \textbf{then } min \leftarrow A[i] \end{cases}$
return (max, min)

Analysis of FindMaximumAndMinimum

Exercise: Give a formal proof by induction that *FindMaximumAndMinimum* is correct.

The complexity of *FindMaximumAndMinimum* is $\Theta(n)$

More precisely, *FindMaximumAndMinimum* requires $2n - 2$ comparisons of array elements given an array of size n.

The **complexity** is optimal (why?), but there are are algorithms to solve the **Max-Min** problem which require fewer comparisons of array elements than *FindMaximumAndMinimum*.

Note: An algorithm requiring fewer comparisons of array elements **may or may not be faster** than *FindMaximumAndMinimum*.

Is there a simple optimization that we can perform to improve *FindMaximumAndMinimum*? (Hint: consider the two **if** statements.)

An improved algorithm

Algorithm: *ImprovedFindMaximumAndMinimum*(A)

$max \leftarrow A[1]$
$min \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do $\begin{cases} \textbf{if } A[i] > max \\ \quad \textbf{then } max \leftarrow A[i] \\ \quad \textbf{else } \begin{cases} \textbf{if } A[i] < min \\ \quad \textbf{then } min \leftarrow A[i] \end{cases} \end{cases}$
return (max, min)

Justification: We only need to execute the second **if** statement when the first **if** statement is false.

Analysis of the improved algorithm

The number of comparisons of array elements required by the improved algorithm varies between $n-1$ and $2n-2$. (When do these extreme cases occur?)

It may be of interest to compute the **average** number of comparisons of array elements required by the improved algorithm. The average will be computed over the $n!$ possible **orderings** (i.e., permutations) of n distinct elements.

In iteration i, we perform only one comparison when $A[i]$ is the **maximum** value in $A[1], \ldots, A[i]$; this happens with probability $1/i$.

So we perform one comparison with probability $1/i$, and two comparisons are done with probability $1-1/i$. The **average** (i.e., expected) number of comparisons done in iteration i is

$$1 \times \frac{1}{i} + 2 \times \left(1 - \frac{1}{i}\right) = 2 - \frac{1}{i}.$$

Analysis of the improved algorithm (cont.)

Therefore, the expected number of comparisons of array elements is

$$\sum_{i=2}^{n} \left(2 - \frac{1}{i}\right) = 2n - 2 - \Theta(\log n)$$

(this will be justified on a later slide).

Asymptotically, this is essentially $2n$, since $\log n$ is **insignificant** compared to n.

So the improvement is not so great!

A more significant improvement

With some ingenuity, we can actually reduce the number of comparisons of array elements by (roughly) 25%.

Suppose n is even and we consider the elements two at a time. Initially, we compare the first two elements and initialize maximum and minimum values. (**One comparison** is required here.)

Then, each time we compare a new pair of elements, we subsequently compare the larger of the two elements to the current maximum and the smaller of the two to the current minimum. (**Three comparisons** are done here to process two array elements.)

This yields an algorithm requiring a total of $3n/2 - 2$ comparisons.

An optimal (?) algorithm

Algorithm: *OptimalFindMaximumAndMinimum*(A)

comment: assume n is even

if $A[1] > A[2]$ **then** $\begin{cases} max \leftarrow A[1] \\ min \leftarrow A[2] \end{cases}$

else $\begin{cases} max \leftarrow A[2] \\ min \leftarrow A[1] \end{cases}$

for $i \leftarrow 2$ **to** $n/2$

do $\begin{cases} \textbf{if } A[2i-1] > A[2i] \\ \quad \textbf{then } \begin{cases} \textbf{if } A[2i-1] > max \textbf{ then } max \leftarrow A[2i-1] \\ \textbf{if } A[2i] < min \textbf{ then } min \leftarrow A[2i] \end{cases} \\ \quad \textbf{else } \begin{cases} \textbf{if } A[2i] > max \textbf{ then } max \leftarrow A[2i] \\ \textbf{if } A[2i-1] < min \textbf{ then } min \leftarrow A[2i-1] \end{cases} \end{cases}$

return (max, min)

Optimality of the previous algorithm

It is possible to **prove** that any algorithm that solves the **Max-Min** problem requires at least $3n/2 - 2$ comparisons of array elements in the worst case. A proof of this is given in a supplementary note valuable on the course web site.

Therefore the algorithm *OptimalFindMaximumAndMinimum* is in fact **optimal** with respect to the number of comparisons of array elements required.

Problems

Problem: Given a problem instance I for a problem **P**, carry out a particular computational task.

Problem Instance: **Input** for the specified problem.

Problem Solution: **Output** (correct answer) for the specified problem.

Size of a problem instance: Size(I) is a positive integer which is a measure of the size of the instance I.

Algorithms and Programs

Algorithm: An algorithm is a step-by-step process (e.g., described in **pseudocode**) for carrying out a series of computations, given some appropriate input.

Algorithm solving a problem: An Algorithm A **solves** a problem **P** if, for every instance I of **P**, A finds a valid solution for the instance I in finite time.

Program: A program is an **implementation** of an algorithm using a specified computer language.

Running Time

Running Time of a Program: $T_{\mathbf{M}}(I)$ denotes the running time (in seconds) of a program $\mathbf{M}$ on a problem instance I.

Worst-case Running Time as a Function of Input Size: $T_{\mathbf{M}}(n)$ denotes the **maximum** running time of program $\mathbf{M}$ on instances of size n:

$$T_{\mathbf{M}}(n) = \max\{T_{\mathbf{M}}(I) : \mathsf{Size}(I) = n\}.$$

Average-case Running Time as a Function of Input Size: $T_{\mathbf{M}}^{avg}(n)$ denotes the **average** running time of program $\mathbf{M}$ over all instances of size n:

$$T_{\mathbf{M}}^{avg}(n) = \frac{1}{|\{I : \mathsf{Size}(I) = n\}|} \sum_{\{I : \mathsf{Size}(I) = n\}} T_{\mathbf{M}}(I).$$

Complexity

Worst-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. An algorithm *A* has **worst-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm *A* such that $T_{\mathbf{M}}(n) \in \Theta(f(n))$.

Average-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. An algorithm *A* has **average-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm *A* such that $T_{\mathbf{M}}^{avg}(n) \in \Theta(f(n))$.

Running Time vs Complexity

Running time can only be determined by implementing a program and running it on a specific computer.

Running time is influenced by many factors, including the programming language, processor, operating system, etc.

Complexity (AKA **growth rate**) can be analyzed by high-level mathematical analysis. It is **independent** of the above-mentioned factors affecting running time.

Complexity is a less precise measure than running time since it is asymptotic and it incorporates unspecified constant factors and unspecified lower order terms.

However, if algorithm A has lower complexity than algorithm B, then a program implementing algorithm A will be faster than a program implementing algorithm B for sufficiently large inputs.

Order Notation

O-notation:

$f(n) \in O(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.

Here the complexity of f is **not higher** than the complexity of g.

Ω-notation:

$f(n) \in \Omega(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$.

Here the complexity of f is **not lower** than the complexity of g.

Θ-notation:

$f(n) \in \Theta(g(n))$ if **there exist** constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.

Here f and g have the **same complexity**.

Order Notation (cont.)

o-notation:

$f(n) \in o(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$.

Here f has **lower complexity** than g.

ω-notation:

$f(n) \in \omega(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le c\, g(n) \le f(n)$ for all $n \ge n_0$.

Here f has **higher complexity** than g.

Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty. \end{cases}$$

Relationships between Order Notations

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$
$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$
$$f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$$

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$
$$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$$
$$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$$

Algebra of Order Notations

"Maximum" rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Then:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$
$$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$$
$$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$$

"Summation" rules:

$$O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O(f(i))$$
$$\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))$$
$$\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))$$

Sequences

Arithmetic sequence:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

Geometric sequence:

$$\sum_{i=0}^{n-1} a\, r^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$

Arithmetic-geometric sequence:

$$\sum_{i=0}^{n-1}(a+di)r^i = \frac{a}{1-r} - \frac{(a+(n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}$$

provided that $r \neq 1$.

Sequences (cont.)

Harmonic sequence:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

More precisely, it is possible to prove that

$$\lim_{n\to\infty} (H_n - \ln n) = \gamma,$$

where $\gamma \approx 0.57721$ is **Euler's constant**.

Miscellaneous Formulae

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b x/y = \log_b x - \log_b y$$

$$\log_b 1/x = -\log_b x$$

$$\log_b x^y = y \log_b x$$

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

$$n! \in \Theta\left(n^{n+1/2} e^{-n}\right)$$

$$\log n! \in \Theta(n \log n)$$

Techniques for Algorithm Analysis

Two general strategies are as follows:

- Use Θ-bounds **throughout the analysis** and thereby obtain a Θ-bound for the complexity of the algorithm.
- Prove a O-bound and a **matching** Ω-bound **separately** to get a Θ-bound. Sometimes this technique is easier because arguments for O-bounds may use simpler upper bounds (and arguments for Ω-bounds may use simpler lower bounds) than arguments for Θ-bounds do.

Techniques for Loop Analysis

Identify **elementary operations** that require constant time (denoted $\Theta(1)$ time).

The complexity of a loop is expressed as the **sum** of the complexities of each iteration of the loop.

Analyze independent loops **separately**, and then **add** the results: use "maximum rules" and simplify whenever possible.

If loops are nested, start with the **innermost loop** and proceed outwards. In general, this kind of analysis requires evaluation of **nested summations**.

Example of Loop Analysis

Algorithm: *LoopAnalysis1*$(n : integer)$

(1) $sum \leftarrow 0$

(2) **for** $i \leftarrow 1$ **to** n

do $\begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } i \\ \quad \textbf{do } \begin{cases} sum \leftarrow sum + (i-j)^2 \\ sum \leftarrow sum/i \end{cases} \end{cases}$

(3) **return** (sum)

Θ-bound analysis

(1)	$\Theta(1)$
(2)	Complexity of inner **for** loop: $\Theta(i)$
	Complexity of outer **for** loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
	Note: $\sum_{i=1}^{n} i = n(n+1)/2$
(3)	$\Theta(1)$
total	$\Theta(n^2)$

Example of Loop Analysis (cont.)

Proving separate O- and Ω-bounds

We focus on the two nested **for** loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per iteration.

Upper bound:

$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds **match**, the complexity is $\Theta(n^2)$.

Another Example of Loop Analysis

Algorithm: *LoopAnalysis2*$(A : array; n : integer)$

$max \leftarrow 0$
for $i \leftarrow 1$ **to** n
do $\left\{\begin{array}{l} \textbf{for } j \leftarrow i \textbf{ to } n \\ \textbf{do } \left\{\begin{array}{l} sum \leftarrow 0 \\ \textbf{for } k \leftarrow i \textbf{ to } j \\ \textbf{do } \left\{\begin{array}{l} sum \leftarrow sum + A[k] \\ \textbf{if } sum > max \\ \quad \textbf{then } max \leftarrow sum \end{array}\right. \end{array}\right. \end{array}\right.$
return (max)

Yet Another Example of Loop Analysis

Algorithm: *LoopAnalysis3*$(n : integer)$

$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n

$$\textbf{do} \begin{cases} j \leftarrow i \\ \textbf{while } j \geq 1 \\ \quad \textbf{do} \begin{cases} sum \leftarrow sum + i/j \\ j \leftarrow \left\lfloor \frac{j}{2} \right\rfloor \end{cases} \end{cases}$$

return (sum)

Table of Contents

The Divide-and-Conquer Design Strategy

divide: Given a problem instance I, construct one or more smaller problem instances, denoted $I_1, \ldots, I_a$ (these are called **subproblems**). Usually, we want the size of these subproblems to be small compared to the size of I, e.g., half the size.

conquer: For $1 \leq j \leq a$, solve instance I_j recursively, obtaining solutions $S_1, \ldots, S_a$.

combine: Given $S_1, \ldots, S_a$, use an appropriate **combining** function to find the solution S to the problem instance I, i.e., $S \leftarrow \mathit{Combine}(S_1, \ldots, S_a)$.

Example: Design of Mergesort

Here, a problem instance consists of an array A of n integers, which we want to sort in increasing order. The size of the problem instance is n.

divide: Split A into two subarrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A.

conquer: Run *Mergesort* on A_L and A_R.

combine: After A_L and A_R have been sorted, use a function *Merge* to merge A_L and A_R into a single sorted array. Recall that this can be done in time $\Theta(n)$ with a single pass through A_L and A_R. We simply keep track of the "current" element of A_L and A_R, always copying the smaller one into the sorted array.

Mergesort

Algorithm: *Mergesort*$(A : array; n : integer)$
if $n = 1$
  then $S \leftarrow A$
  else $\begin{cases} n_L \leftarrow \lceil \frac{n}{2} \rceil \\ n_R \leftarrow \lfloor \frac{n}{2} \rfloor \\ A_L \leftarrow [A[1], \dots, A[n_L]] \\ A_R \leftarrow [A[n_L + 1], \dots, A[n]] \\ S_L \leftarrow \textit{Mergesort}(A_L, n_L) \\ S_R \leftarrow \textit{Mergesort}(A_R, n_R) \\ S \leftarrow \textit{Merge}(S_L, n_L, S_R, n_R) \end{cases}$
return (S, n)

Analysis of Mergesort

Let $T(n)$ denote the time to run *Mergesort* on an array of length n.

divide takes time $\Theta(n)$

conquer takes time $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$

combine takes time $\Theta(n)$

Recurrence relation:

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Recurrence Relations

It is simpler to replace the Θ's by constant factors c and d. The resulting recurrence relation is called the **exact recurence**.

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

If we then remove the floors, an ceilings, we obtain the **sloppy recurrence**:

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

The exact and sloppy recurrences are identical when n is a power of 2.

We will begin by solving the recurrence when $n = 2^j$ using the **recursion tree method**.

Recursion Tree Method

We construct a **recursion tree**, assuming $n = 2^j$, as follows:

step 1 Start with a one-node tree, say N, which receives the value $T(n)$.

step 2 Grow two children of N. These children, say N_1 and N_2, receive the value $T(n/2)$, and the value of N is updated to be cn.

step 3 Repeat this process recursively, terminating when a node receives the value $T(1) = d$.

step 4 Sum the values on each level of the tree, and then compute the sum of all these sums; the result is $T(n)$.

Solving the Exact Recurrence

The recursion tree method finds the **exact** solution of the recurrence when $n = 2^j$ (it is in fact a **proof** for these values of n).

Suppose we express this solution (for powers of 2) as a function of n, using Θ-notation.

The resulting function of n will in fact yield the **complexity** of the solution of the exact recurrence for **all values** of n.

This derivation of the complexity of $T(n)$ not a proof, however. If a rigourous mathematical proof is required, then it is necessary to use induction along with the exact recurrence.

Master Theorem

The Master Theorem provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The following is a simplified version of the Master Theorem:

Theorem

Suppose that $a \geq 1$ *and* $b > 1$*. Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^y). \tag{1}$$

Denote $x = \log_b a$*. Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x. \end{cases}$$

Proof of the Master Theorem (simplified version)

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = a\,T\left(\frac{n}{b}\right) + c\,n^y, \qquad T(1) = d.$$

Let $n = b^j$.

size of subproblem	# nodes	cost/node	total cost
$n = b^j$	1	$c\,n^y$	$c\,n^y$
$n/b = b^{j-1}$	a	$c\,(n/b)^y$	$c\,a\,(n/b)^y$
$n/b^2 = b^{j-2}$	a^2	$c\,(n/b^2)^y$	$c\,a^2\,(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n/b^{j-1} = b$	a^{j-1}	$c\,(n/b^{j-1})^y$	$c\,a^{j-1}\,(n/b^{j-1})^y$
$n/b^j = 1$	a^j	d	$d\,a^j$

Computing $T(n)$

Summing the costs of all levels of the recursion tree, we have that

$$T(n) = d\,a^j + c\,n^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i.$$

Recall that $b^x = a$ and $n = b^j$. Hence $a^j = (b^x)^j = (b^j)^x = n^x$.

The formula for $T(n)$ is a **geometric sequence** with ratio $r = a/b^y = b^{x-y}$:

$$T(n) = d\,n^x + c\,n^y \sum_{i=0}^{j-1} r^i.$$

There are **three cases**, depending on whether $r > 1$, $r = 1$ or $r < 1$.

Complexity of $T(n)$

case	r	y, x	complexity of $T(n)$
heavy leaves	$r > 1$	$y < x$	$T(n) \in \Theta(n^x)$
balanced	$r = 1$	$y = x$	$T(n) \in \Theta(n^x \log n)$
heavy top	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

heavy leaves means that cost of the recursion tree is dominated by the cost of the leaf nodes.

balanced means that costs of the levels of the recursion tree stay constant (except for the last level)

heavy top means that cost of the recursion tree is dominated by the cost of the root node.

Master Theorem (modified general version)

Theorem

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n).$$

Denote $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } f(n) \in O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\ \Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \\ & \text{for some } \epsilon > 0. \end{cases}$$

The Max-Min Problem

Let's design a divide-and-conquer algorithm for the **Max-Min** problem.

Divide: Suppose we split A into two equal-sized subarrays, A_L and A_R.

Conquer: We find the maximum and minimum elements in each subarray recursively, obtaining max_L, min_L, max_R and min_R.

Combine: Then we can easily "combine" the solutions to the two subproblems to solve the original problem instance:

$$max \leftarrow \max\{max_L, max_R\}$$

and

$$min \leftarrow \min\{min_L, min_R\}$$

The Max-Min Problem (cont.)

The recurrence relation describing the complexity of the running time is $T(n) = 2T(n/2) + \Theta(1)$.

The Master Theorem shows that the $T(n) \in \Theta(n)$.

However, we can also count the **exact number** of comparisons done by the algorithm, obtaining the (sloppy) recurrence

$$C(n) = 2C(n/2) + 2, \quad C(2) = 1.$$

For n a power of 2, the solution to this recurrence relation is $C(n) = 3n/2 - 2$, so the divide-and-conquer algorithm is **optimal** for these values of n.

Multiprecision Multiplication

Problem

Multiprecision Multiplication

Instance: *Two k-bit positive integers, X and Y, having binary representations*

$$X = [X[k-1], ..., X[0]]$$

and

$$Y = [Y[k-1], ..., Y[0]].$$

Question: *Compute the $2k$-bit positive integer $Z = XY$, where*

$$Z = (Z[2k-1], ..., Z[0]).$$

We are interested in the **bit complexity** of algorithms that solve **Multiprecision Multiplication**, which means that the complexity is expressed as a function of k (the size of the problem instance is $2k$ bits).

Not-So-Fast D&C Multiprecision Multiplication

Algorithm: *NotSoFastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else $\begin{cases} Z_1 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_R, k/2) \\ Z_4 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_L, k/2) \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 + Z_4, k/2) \end{cases}$

return (Z)

Fast D&C Multiprecision Multiplication

Algorithm: *FastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

$$\textbf{else} \begin{cases} X_T \leftarrow X_L + X_R \\ Y_T \leftarrow Y_L + Y_R \\ Z_1 \leftarrow \textit{FastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{FastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{FastMultiply}(X_T, Y_T, k/2), \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 - Z_1 - Z_2, k/2) \end{cases}$$

return (Z)

Matrix Multiplication

Problem

Matrix Multiplication

Instance: *Two n by n matrices, A and B.*

Question: *Compute the n by n matrix product $C = AB$.*

The naive algorithm for **Matrix Multiplication** has complexity $\Theta(n^3)$.

Matrix Multiplication: Problem Decomposition

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

If A, B are n by n matrices, then $a, b, ..., h, r, s, t, u$ are $\frac{n}{2}$ by $\frac{n}{2}$ matrices, where

$$\begin{aligned} r &= a\,e + b\,g & \qquad s &= a\,f + b\,h \\ t &= c\,e + d\,g & \qquad u &= c\,f + d\,h \end{aligned}$$

We require 8 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Efficient D&C Matrix Multiplication

Define

$$\begin{aligned} P_1 &= a(f-h) & P_2 &= (a+b)h \\ P_3 &= (c+d)e & P_4 &= d(g-e) \\ P_5 &= (a+d)(e+h) & P_6 &= (b-d)(g+h) \\ P_7 &= (a-c)(e+f). \end{aligned}$$

Then, compute

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 & s &= P_1 + P_2 \\ t &= P_3 + P_4 & u &= P_5 + P_1 - P_3 - P_7. \end{aligned}$$

We now require only 7 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Closest Pair

Problem

Closest Pair

Instance: *a set* Q *of* n *distinct points in the Euclidean plane,*

$$Q = \{Q[1], \ldots, Q[n]\}.$$

Find: *Two distinct points* $Q[i] = (x, y), Q[j] = (x', y')$ *such that the Euclidean distance*

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.

Closest Pair: Problem Decomposition

Suppose we presort the points in Q with respect to their x-coordinates (this takes time $\Theta(n \log n)$).

Then we can easily find the vertical line that partitions the set of points Q into two sets of size $n/2$: this line has equation $x = Q[m].x$, where $m = n/2$.

The set Q is global with respect to the recursive procedure *ClosestPair1*.

At any given point in the recursion, we are examining a subarray $(Q[\ell], \ldots, Q[r])$, and $m = \lfloor (\ell + r)/2 \rfloor$.

We call *ClosestPair1*$(1, n)$ to solve the given problem instance.

Closest Pair: Solution 1

Algorithm: *ClosestPair1*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else $\begin{cases} m \leftarrow \lfloor (\ell + r)/2 \rfloor \\ \delta_L \leftarrow \textit{ClosestPair1}(\ell, m) \\ \delta_R \leftarrow \textit{ClosestPair1}(m+1, r) \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q[m].x) \\ R \leftarrow \textit{SortY}(R) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$

return (δ)

Selecting Candidates from the Vertical Strip

Algorithm: *SelectCandidates*$(\ell, r, \delta, xmid)$

$$
\begin{array}{l}
j \leftarrow 0 \\
\textbf{for } i \leftarrow \ell \textbf{ to } r \\
\quad \textbf{do } \left\{ \begin{array}{l} \textbf{if } |Q[i].x - xmid| \leq \delta \\ \quad \textbf{then } \left\{ \begin{array}{l} j \leftarrow j + 1 \\ R[j] \leftarrow Q[i] \end{array} \right. \end{array} \right. \\
\textbf{return } (R)
\end{array}
$$

Checking the Vertical Strip

Algorithm: *CheckStrip*(R, δ)

$t \leftarrow$ *size*(R)
$\delta' \leftarrow \delta$
for $j \leftarrow 1$ **to** $t - 1$
do $\begin{cases} \textbf{for } k \leftarrow j + 1 \textbf{ to } \min\{t, j + 7\} \\ \quad \textbf{do } \begin{cases} x \leftarrow R[j].x \\ x' \leftarrow R[k].x \\ y \leftarrow R[j].y \\ y' \leftarrow R[k].y \\ \delta' \leftarrow \min\left\{\delta', \sqrt{(x' - x)^2 + (y' - y)^2}\right\} \end{cases} \end{cases}$
return (δ')

Closest Pair: Solution 2

Algorithm: *ClosestPair2*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else $\begin{cases} m \leftarrow \lfloor (\ell + r)/2 \rfloor \\ \delta_L \leftarrow \textit{ClosestPair2}(\ell, m) \\ \text{comment: } Q[\ell], \ldots, Q[m] \text{ is sorted WRT } y\text{-coordinates} \\ \delta_R \leftarrow \textit{ClosestPair2}(m+1, r) \\ \text{comment: } Q[m+1], \ldots, Q[r] \text{ is sorted WRT } y\text{-coordinates} \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ \textit{Merge}(\ell, m, r) \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q[m].x) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$

return (δ)

Selection

Problem

Selection

Instance: *An array* $A[1], \ldots, A[n]$ *of distinct integer values, and an integer* k, *where* $1 \leq k \leq n$.

Find: *The* k*th smallest integer in the array* A.

The problem **Median** is the special case of **Selection** where $k = \lceil \frac{n}{2} \rceil$.

QuickSelect

Suppose we choose a **pivot** element y in the array A, and we **restructure** A so that all elements less than y precede y in A, and all elements greater than y occur after y in A. (This is exactly what is done in *Quicksort*, and it takes **linear time**.)

Suppose that $A[posn] = y$ after restructuring. Let A_L be the subarray $A[1], \ldots, A[posn-1]$ and let A_R be the subarray (of size $n - posn$) $A[posn+1], \ldots, A[n]$.

Then the kth smallest element of A is

$$\begin{cases} y & \text{if } k = posn \\ \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\ \text{the } (k-posn)\text{th smallest element of } A_R & \text{if } k > posn. \end{cases}$$

We make (at most) one recursive call at each level of the recursion.

Average-case Analysis of QuickSelect

We say that a pivot is **good** if $posn$ is in the middle half of A.

The probability that a pivot is good is $1/2$.

On average, after **two iterations**, we will encounter a good pivot.

If a pivot is good, then $|A_L| \leq 3n/4$ and $|A_R| \leq 3n/4$.

With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.

It follows that the average-case complexity of the *QuickSelect* is linear.

Achieving $O(n)$ Worst-Case Complexity: A Strategy for Choosing the Pivot

We choose the pivot to be a certain **median-of-medians**:

step 1 Given $n \geq 15$, write $n = 10r + 5 + \theta$, where $r \geq 1$ and $0 \leq \theta \leq 9$.

step 2 Divide A into $2r + 1$ disjoint subarrays of 5 elements. Denote these subarrays by $B_1, \ldots, B_{2r+1}$.

step 3 For $1 \leq i \leq 2r + 1$, find the median of B_i (nonrecursively), and denote it by m_i.

step 4 Define M to be the array consisting of elements $m_1, \ldots, m_{2r+1}$.

step 5 Find the median y of the array M (recursively).

step 6 Use the element y as the pivot for A.

Median-of-medians-QuickSelect

Algorithm: *Mom-QuickSelect*(k, n, A)

if $n \leq 14$ **then** sort A and **return** $(A[k])$
write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
construct $B_1, \dots, B_{2r+1}$ (subarrays of A, each of size 5)
find medians $m_1, \dots, m_{2r+1}$ (non-recursively)
$M \leftarrow [m_1, \dots, m_{2r+1}]$
$y \leftarrow$ *Mom-QuickSelect*$(r + 1, 2r + 1, M)$
$(A_L, A_R, posn) \leftarrow$ *Restructure*(A, y)
if $k = posn$ **then return** (y)
  else if $k < posn$ **then return** (*Mom-QuickSelect*$(k, posn - 1, A_L)$)
    else return (*Mom-QuickSelect*$(k - posn, n - posn, A_R)$)

Worst-case Analysis of Mom-QuickSelect

When the pivot is the median-of-medians, we have that $|A_L| \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$ and $A_R \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$.

The *Mom-QuickSelect* algorithm requires **two recursive calls**.

The worst-case complexity $T(n)$ of this algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{7n+12}{10} \right\rfloor\right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14. \end{cases}$$

It can be shown that $T(n)$ is $O(n)$.

Table of Contents

Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Problem Instance: **Input** for the specified problem.

Problem Constraints: **Requirements** that must be satisfied by any feasible solution.

Feasible Solution: For any problem instance I, *feasible*(I) is the set of all outputs (i.e., solutions) for the instance I that satisfy the given constraints.

Objective Function: A function f : *feasible*$(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of f as being a **profit** or a **cost** function.

Optimal Solution: A feasible solution $X \in$ *feasible*(I) such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).

The Greedy Method

partial solutions

Given a problem instance I, it should be possible to write a feasible solution X as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer n, where $x_i \in \mathcal{X}$ for all i. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a **partial solution** if no constraints are violated. Note: it may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the **choice set**

$$\mathit{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$

The Greedy Method (cont.)

local evaluation criterion

For any $y \in \mathcal{X}$, $g(y)$ is a **local evaluation criterion** that measures the cost or profit of including y in a (partial) solution.

extension

Given a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, choose $y \in$ *choice*(X) so that $g(y)$ is as small (or large) as possible. Update X to be the $(i+1)$-tuple $[x_1, \ldots, x_i, y]$.

greedy algorithm

Starting with the "empty" partial solution, repeatedly extend it until a feasible solution X is constructed. This feasible solution may or may not be optimal.

Features of the Greedy Method

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function g, followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function g).

Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

Interval Selection

Problem

Interval Selection

Instance: *A set* $\mathcal{A} = \{A_1, \ldots, A_n\}$ *of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, *where* s_i *is the* **start time** *of interval* A_i *and* f_i *is the* **finish time** *of* A_i.

Feasible solution: *A subset* $\mathcal{B} \subseteq \mathcal{A}$ *of* **pairwise disjoint intervals**.

Find: *A feasible solution of maximum size (i.e., one that maximizes* $|\mathcal{B}|$*).*

Possible Greedy Strategies for Interval Selection

1. Choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is s_i).
2. Choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).
3. Choose the **earliest finishing** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is f_i).

Does one of these strategies yield a correct greedy algorithm?

A Greedy Algorithm for Interval Selection

Algorithm: *GreedyIntervalSelection*($\mathcal{A}$)

rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$

$\mathcal{B} \leftarrow \{A_1\}$

$prev \leftarrow 1$

comment: $prev$ is the index of the last selected interval

for $i \leftarrow 2$ **to** n

do $\begin{cases} \textbf{if } s_i \geq f_{prev} \\ \quad \textbf{then } \begin{cases} \mathcal{B} \leftarrow \mathcal{B} \cup \{A_i\} \\ prev \leftarrow i \end{cases} \end{cases}$

return $(\mathcal{B})$

Interval Colouring

Problem

Interval Colouring

Instance: *A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where s_i is the **start time** *of interval A_i and f_i is the* **finish time** *of A_i.*

Feasible solution: *A c-***colouring** *is a mapping $col : \mathcal{A} \rightarrow \{1, \ldots, c\}$ that assigns each interval a* **colour** *such that two intervals receiving the same colour are always disjoint.*

Find: *A c-colouring of $\mathcal{A}$ with the minimum number of colours.*

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first $i < n$ intervals using nc colours.

We will colour the $(i + 1)$st interval with the **any permissible colour**. If it cannot be coloured using any of the existing nc colours, then we introduce a **new colour** and nc is increased by 1.

Question: In **what order** should we consider the intervals?

A Greedy Algorithm for Interval Colouring

Algorithm: *GreedyIntervalColouring*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $s_1 \leq \cdots \leq s_n$
$nc \leftarrow 1$
$colour[1] \leftarrow 1$
$finish[1] \leftarrow f_1$
for $i \leftarrow 2$ **to** n

$$\textbf{do} \begin{cases} flag \leftarrow \textbf{ false} \\ c \leftarrow 1 \\ \textbf{while } c \leq nc \textbf{ and } (\textbf{ not } flag) \\ \quad \textbf{do} \begin{cases} \textbf{if } finish[c] \leq s_i \textbf{ then } \begin{cases} colour[i] \leftarrow c \\ finish[c] \leftarrow f_i \\ flag \leftarrow \textbf{ true} \end{cases} \\ \quad \textbf{else } c \leftarrow c+1 \end{cases} \\ \textbf{if not } flag \textbf{ then } \begin{cases} nc \leftarrow nc+1 \\ colour[i] \leftarrow nc \\ finish[nc] \leftarrow f_i \end{cases} \end{cases}$$

return $(nc, colour)$

Comments and Questions

In the algorithm on the previous slide, at any point in time, $\mathit{finish}[c]$ denotes the finishing time of the **last interval** that has received colour c. Therefore, a new interval A_i can be assigned colour c if $s_i \geq \mathit{finish}[c]$.

The complexity of the algorithm is $O(n \times nc)$.

If it turns out that $nc \in \Omega(n)$, then the best we can say is that the complexity is $O(n^2)$.

What **inefficiencies** exist in this algorithm?

What **data structure** would allow a more efficient algorithm to be designed?

What would be the complexity of an algorithm making use of an appropriate data structure?

The Stable Marriage Problem

Problem

Stable Marriage

Instance: *A set of n* **men***, say $M = [m_1, \ldots, m_n]$, and a set of n* **women***, $W = [w_1, \ldots, w_n]$.*

Each man m_i has a **preference ranking** *of the n women, and each woman w_i has a preference ranking of the n men: $pref(m_i, j) = w_k$ if w_k is the j-th favourite woman of man m_i; and $pref(w_i, j) = m_k$ if m_k is the j-th favourite man of woman w_i.*

Find: *A* **matching** *of the n men with the n women such that there does not exist a couple (m_i, w_j) who are* **not** *engaged to each other, but prefer each other to their existing matches. A matching with this this property is called a* **stable matching***.*

Overview of the Gale-Shapley Algorithm

Men propose to women.

If a woman accepts a proposal, then the couple is **engaged**.

An unmatched woman **must accept** a proposal.

If an engaged woman receives a proposal from a man whom she prefers to her current match, the she **cancels** her existing engagement and she becomes engaged to the new proposer; her previous match is no longer engaged.

If an engaged woman receives a proposal from a man, but she prefers her current match, then the proposal is **rejected**.

Engaged women never become unengaged.

A man might make a number of proposals (up to n); the order of the proposals is determined by the man's preference list.

Gale-Shapley Algorithm

Algorithm: *Gale-Shapley*(M, W, pref)

$$
\begin{array}{l}
\mathit{Match} \leftarrow \emptyset \\
\textbf{while } \text{there exists an unengaged man } m_i \\
\quad \textbf{do } \left\{
\begin{array}{l}
\text{let } w_j \text{ be the next woman in } m_i\text{'s preference list} \\
\textbf{if } w_j \text{ is not engaged} \\
\quad \textbf{then } \mathit{Match} \leftarrow \mathit{Match} \cup \{m_i, w_j\} \\
\quad \textbf{else } \left\{
\begin{array}{l}
\text{suppose } \{m_k, w_j\} \in \mathit{Match} \\
\textbf{if } w_j \text{ prefers } m_i \text{ to } m_k \\
\quad \textbf{then } \left\{
\begin{array}{l}
\mathit{Match} \leftarrow \mathit{Match} \backslash \{m_k, w_j\} \cup \{m_i, w_j\} \\
\text{comment: } m_k \text{ is now unengaged}
\end{array}
\right.
\end{array}
\right.
\end{array}
\right. \\
\textbf{return } (\mathit{Match})
\end{array}
$$

Questions

How do we prove that the *Gale-Shapley* algorithm always **terminates**?

How many **iterations** does this algorithm require in the worst case?

How do we prove that this algorithm is **correct**, i.e., that it finds a stable matching?

Is there an efficient way to **identify** an unengaged man at any point in the algorithm? What **data structure** would be helpful in doing this?

What can we say about the **complexity** of the algorithm?

Knapsack Problems

Problem

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An n-tuple $X = [x_1, \ldots, x_n]$ where $\sum_{i=1}^{n} w_i x_i \leq M$.*

In the **0-1 Knapsack** *problem (often denoted just as* **Knapsack***), we require that $x_i \in \{0, 1\}$, $1 \leq i \leq n$.*

In the **Rational Knapsack** *problem, we require that $x_i \in \mathbb{Q}$ and $0 \leq x_i \leq 1$, $1 \leq i \leq n$.*

Find: *A feasible solution X that maximizes $\sum_{i=1}^{n} p_i x_i$.*

Possible Greedy Strategies for Knapsack Problems

1. Consider the items in decreasing order of **profit** (i.e., the local evaluation criterion is p_i).
2. Consider the items in increasing order of **weight** (i.e., the local evaluation criterion is w_i).
3. Consider the items in decreasing order of **profit divided by weight** (i.e., the local evaluation criterion is p_i/w_i).

Does one of these strategies yield a correct greedy algorithm for the **0-1 Knapsack** or **Rational Knapsack** problem?

A Greedy Algorithm for Rational Knapsack

Algorithm: *GreedyRationalKnapsack*$(P, W : array; M : integer)$

rename the items, sorting if necessary, so that $p_1/w_1 \geq \cdots \geq p_n/w_n$

$X \leftarrow [0, \ldots, 0]$

$i \leftarrow 1$

$CurW \leftarrow 0$

while $(CurW < M)$ **and** $(i \leq n)$

$$\textbf{do} \begin{cases} \textbf{if } CurW + w_i \leq M \\ \quad \textbf{then } \begin{cases} x_i \leftarrow 1 \\ CurW \leftarrow CurW + w_i \end{cases} \\ \quad \textbf{else } \begin{cases} x_i \leftarrow (M - CurW)/w_i \\ CurW := M \end{cases} \end{cases}$$

return (X)

Table of Contents

Graphs and Digraphs

A **graph** is a pair $G = (V, E)$. V is a set whose elements are called **vertices** and E is a set whose elements are called **edges**. Each edge joins two distinct vertices. An edge can be represented as a set of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as uv or vu.

We often denote the number of vertices by n and the number of edges by m. Clearly $m \leq \binom{n}{2}$.

A **directed graph** or **digraph** is also a pair $G = (V, E)$. The elements of E are called **directed edges** or **arcs** in a digraph. Each arc joins two vertices, and an arc can be represented as a ordered pair, e.g., (u, v). The arc (u, v) is directed from u (the **tail**) to v (the **head**), and we allow $u = v$.

If we denote the number of vertices by n and the number of arcs by m, then $m \leq n^2$.

Data Structures for Graphs: Adjacency Matrices

There are two main data structures to represent graphs: an **adjacency matrix** and a set of **adjacency lists**.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The **adjacency matrix** of G is an n by n matrix $A = (a_{u,v})$, which is indexed by V, such that

$$a_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

There are exactly $2m$ entries of A equal to 1.

If G is a digraph, then

$$a_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For a digraph, there are exactly m entries of A equal to 1.

Data Structures for Graphs: Adjacency Lists

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An **adjacency list representation** of G consists of n linked lists.

For every $u \in V$, there is a linked list (called an **adjacency list**) which is named $Adj[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled v. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in $Adj[u]$ and a node u in $Adj[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.

Breadth-first Search of an Undirected Graph

A **breadth-first search** of an undirected graph begins at a specified vertex s.

The search "spreads out" from s, proceeding in **layers**.

First, all the neighbours of s are **explored**.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A **queue** is used to keep track of the vertices to be explored.

Breadth-first Search

Algorithm: $\mathit{BFS}(G, s)$
for each $v \in V(G)$
do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$
$colour[s] \leftarrow \text{gray}$
$\mathit{InitializeQueue}(Q)$
$\mathit{Enqueue}(Q, s)$
while $Q \neq \emptyset$

$$\textbf{do} \begin{cases} u \leftarrow \mathit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then} \begin{cases} colour[v] = \text{gray} \\ \pi[v] \leftarrow u \\ \mathit{Enqueue}(Q, v) \end{cases} \end{cases} \\ colour[u] \leftarrow \text{black} \end{cases}$$

Properties of Breadth-first Search

A vertex is white if it is **undiscovered**.

A vertex is gray if it has been **discovered**, but we are still processing its adjacent vertices.

A vertex becomes black when all the adjacent vertices have been processed.

If G is **connected**, then every vertex eventually is coloured black.

When we explore an edge $\{u, v\}$ starting from u:

- if v is white, then uv is a **tree edge** and $\pi[v] = u$ is the **predecessor** of v in the **BFS tree**
- otherwise, uv is a **cross edge**.

The BFS tree consists of all the tree edges.

Every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

Shortest Paths via Breadth-first Search

Algorithm: $\mathit{BFS}(G, s)$

for each $v \in V(G)$ **do** $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \text{gray}$

$\boxed{dist[s] \leftarrow 0}$

$\mathit{InitializeQueue}(Q)$

$\mathit{Enqueue}(Q, s)$

while $Q \neq \emptyset$

$$\textbf{do} \begin{cases} u \leftarrow \mathit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \text{white} \textbf{ then } \begin{cases} colour[v] = \text{gray} \\ \pi[v] \leftarrow u \\ \mathit{Enqueue}(Q, v) \\ \boxed{dist[v] \leftarrow dist[u] + 1} \end{cases} \end{cases} \\ colour[u] \leftarrow \text{black} \end{cases}$$

Distances in Breadth-first Search

If $\{u, v\}$ is **any edge**, then $|dist[u] - dist[v]| \leq 1$.

If uv is a **tree edge**, then $dist[v] = dist[u] + 1$.

$dist[u]$ is the length of the **shortest path** from s to u.

This is also called the **distance** from s to u.

Bipartite Graphs and Breadth-first Search

A graph is **bipartite** if the vertex set can be partitioned as $V = X \cup Y$, in such a way that all edges have one endpoint in X and one endpoint in Y.

A graph is bipartite if and only if it does not contain an **odd cycle**.

BFS can be used to test if a graph is bipartite:

- if we encounter an edge $\{u, v\}$ with $dist[u] = dist[v]$, then G is not bipartite, whereas
- if no such edge is found, then define $X = \{u : dist[u] \text{ is even}\}$ and $Y = \{u : dist[u] \text{ is odd}\}$; then X, Y forms a bipartition.

Depth-first Search of a Directed Graph

A **depth-first search** uses a **stack** (or **recursion**) instead of a queue.

We define predecessors and colour vertices as in BFS.

It is also useful to specify a **discovery time** $d[v]$ and a **finishing time** $f[v]$ for every vertex v.

We increment a **time counter** every time a value $d[v]$ or $f[v]$ is assigned.

We eventually visit all the vertices, and the algorithm constructs a **depth-first forest**.

Depth-first Search

Algorithm: *DFS*(G)

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow \text{gray}$
$time \leftarrow time + 1$
$d[v] \leftarrow time$
comment: $d[v]$ is the discovery time for vertex v
for each $w \in Adj[v]$

$$\textbf{do} \begin{cases} \textbf{if } colour[w] = \text{white} \\ \quad \textbf{then } \begin{cases} \pi[w] \leftarrow v \\ \textit{DFSvisit}(w) \end{cases} \end{cases}$$

$colour[v] \leftarrow \text{black}$
$time \leftarrow time + 1$
$f[v] \leftarrow time$
comment: $f[v]$ is the finishing time for vertex v

Classification of Edges in Depth-first Search

- uv is a **tree edge** if $u = \pi[v]$
- uv is a **forward edge** if it is not a tree edge, and v is a descendant of u in a tree in the depth-first forest
- uv is a **back edge** if u is a descendant of v in a tree in the depth-first forest
- any other edge is a **cross edge**.

Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex v when an edge uv is discovered, and the relation between the start and finishing times of u and v, for each possible type of edge uv.

edge type	colour of v	discovery/finish times
tree	white	$d[u] < d[v] < f[v] < f[u]$
forward	black	$d[u] < d[v] < f[v] < f[u]$
back	gray	$d[v] < d[u] < f[u] < f[v]$
cross	black	$d[v] < f[v] < d[u] < f[u]$

Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never **overlap**. Two intervals are either **disjoint** or **nested**. This is sometimes called the parenthesis theorem.

Topological Orderings and DAGs

A directed graph G is a **directed acyclic graph**, or **DAG**, if G contains no directed cycle.

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in V such that $u < v$ whenever $uv \in E$.

Some interesting/useful facts:

- A DAG contains a vertex of indegree 0.
- A directed graph G has a topological ordering if and only if it is a DAG.
- A directed graph G is a DAG if and only if a DFS of G has no back edges.
- If uv is an edge in a DAG, then a DFS of G has $f[v] < f[u]$.

Topological Ordering via Depth-first Search

Algorithm: *DFS*(G)

$\boxed{\textit{InitializeStack}(S)}$

$\boxed{DAG \leftarrow true}$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

$\boxed{\textbf{if } DAG \ \textbf{ then return } (S) \ \textbf{ else return } (DAG)}$

Topological Ordering via Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow \text{gray}$
$time \leftarrow time + 1$
$d[v] \leftarrow time$
comment: $d[v]$ is the discovery time for vertex v
for each $w \in Adj[v]$

$$\textbf{do} \begin{cases} \textbf{if } colour[w] = \text{white} \\ \quad \textbf{then } \begin{cases} \pi[w] \leftarrow v \\ \textit{DFSvisit}(w) \end{cases} \\ \boxed{\textbf{if } colour[w] = \text{gray} \quad \textbf{then } DAG \leftarrow false} \end{cases}$$

$colour[v] \leftarrow \text{black}$
$\boxed{\textit{Push}(S, v)}$
$time \leftarrow time + 1$
$f[v] \leftarrow time$
comment: $f[v]$ is the finishing time for vertex v

Strongly Connected Components of a Digraph G

For two vertices x and y of G, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from x to y **and** from y to x.

The relation $\sim$ is an **equivalence relation**.

The **strongly connected components** of G are the equivalence classes of vertices defined by the relation $\sim$.

The **component graph** of G is a directed graph whose vertices are the strongly connected components of G. There is an arc from C_i to C_j if and only if there is an arc in G from some vertex of C_i to some vertex of C_j.

For a strongly connected component C, define $f[C] = \max\{f[v] : v \in C\}$ and $d[C] = \min\{d[v] : v \in C\}$.

Some interesting/useful facts:

- The component graph of G is a DAG.
- If C_i, C_j are strongly connected components, and there is an arc from C_i to C_j in the component graph, then $f[C_i] > f[C_j]$.

An Algorithm to Find the Strongly Connected Components

step 1 Perform a depth-first search of G, recording the finishing times $f[v]$ for all vertices v.

step 2 Construct a directed graph H from G by **reversing** the direction of all edges in G.

step 3 Perform a depth-first search of H, considering the vertices in **decreasing** order of the values $f[v]$ computed in step 1.

step 4 The strongly connected components of G are the trees in the depth-first forest constructed in step 3.

Depth-first Search of H

Assume that $f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}]$.

Algorithm: *DFS*(H)

for $j \leftarrow 1$ **to** n
  do $colour[v_{i_j}] \leftarrow \mathrm{white}$
$scc \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $\begin{cases} \textbf{if } colour[v_{i_j}] = \mathrm{white} \\ \quad \textbf{then } \begin{cases} scc \leftarrow scc + 1 \\ \mathit{DFSvisit}(H, v_{i_j}, scc) \end{cases} \end{cases}$
return $(comp)$
comment: $comp[v]$ is the strongly connected component containing v

DFSvisit for H

Algorithm: *DFSvisit*(H, v, scc)

$colour[v] \leftarrow \text{gray}$
$comp[v] \leftarrow scc$
for each $w \in Adj[v]$
  do $\begin{cases} \textbf{if } colour[w] = \text{white} \\ \quad \textbf{then } \textit{DFSvisit}(H, w, scc) \end{cases}$
$colour[v] \leftarrow \text{black}$

Minimum Spanning Trees

A **spanning tree** in a connected, undirected graph $G = (V, E)$ is a subgraph T that is a tree which contains every vertex of V.

T is a spanning tree of G if and only if T is an acyclic subgraph of G that has $n - 1$ edges (where $n = |V|$).

Problem

Minimum Spanning Tree

Instance: *A connected, undirected graph* $G = (V, E)$ *and a* **weight function** $w : E \to \mathbb{R}$.

Find: *A spanning tree* T *of* G *such that*

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

is minimized (this is called a **minimum spanning tree**, *or* **MST***).*

Kruskal's Algorithm

Assume that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, where $m = |E|$).

Algorithm: *Kruskal*(G, w)

$A \leftarrow \emptyset$
for $j \leftarrow 1$ **to** m
  do $\begin{cases} \textbf{if } A \cup \{e_j\} \text{ does not contain a cycle} \\ \quad \textbf{then } A \leftarrow A \cup \{e_j\} \end{cases}$
return (A)

Prim's Algorithm (idea)

We initially choose an arbitrary vertex u_0 and define $A = \{e\}$, where e is the **minimum weight** edge incident with u_0.

A is always a **single tree**, and at each step we select the minimum weight edge that joins a vertex in VA to a vertex not in VA.

Remark: VA denotes the set of vertices in the tree A.

For a vertex $v \notin VA$, define

$$\begin{aligned} N[v] &= \text{a minimum weight edge } \{u, v\} \text{ such that } u \in VA \\ W[v] &= w(N[v], v). \end{aligned}$$

Assume $w(u, v) = \infty$ if $\{u, v\} \notin E$.

Prim's Algorithm

Algorithm: *Prim*(G, w)

$A \leftarrow \emptyset$

$VA \leftarrow \{u_0\}$, where u_0 is arbitrary

for all $v \in V \backslash \{u_0\}$

do $\begin{cases} W[v] \leftarrow w(u_0, v) \\ N[v] \leftarrow u_0 \end{cases}$

while $|A| < n - 1$

do $\begin{cases} \text{choose } v \in V \setminus VA \text{ such that } W[v] \text{ is minimized} \\ VA \leftarrow VA \cup \{v\} \\ u \leftarrow N[v] \\ A \leftarrow A \cup \{uv\} \\ \textbf{for all } v' \in V \setminus VA \\ \quad \textbf{do } \begin{cases} \textbf{if } w(v, v') < W[v'] \\ \quad \textbf{then } \begin{cases} W[v'] \leftarrow w(v, v') \\ N[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (A)

A General Greedy Algorithm to Find an MST

Algorithm: *GreedyMST*(G, w)

$A \leftarrow \emptyset$

while $|A| < n - 1$

do $\begin{cases} \text{let } (S, V \backslash S) \text{ be a cut that respects } A \\ \text{let } e \text{ be a minimum weight crossing edge} \\ A \leftarrow A \cup \{e\} \end{cases}$

return (A)

Some Relevant Definitions for Proof of Correctness

Let $G = (V, E)$ be a graph. A **cut** is a partition of V into two non-empty (disjoint) sets, i.e., a pair $(S, V \backslash S)$, where $S \subseteq V$ and $1 \leq |S| \leq n - 1$.

Let $(S, V \backslash S)$ be a cut in a graph $G = (V, E)$. An edge $e \in E$ is a **crossing edge** with respect to the cut $(S, V \backslash S)$ if e has one endpoint in S and one endpoint in $V \backslash S$.

Let $A \subseteq E$. A cut $(S, V \backslash S)$ **respects** the set of edges A provided that no edge in A is a crossing edge.

Single Source Shortest Paths

Problem

Single Source Shortest Paths

Instance: *A directed graph $G = (V, E)$, a non-negative* **weight function** $w : E \rightarrow \mathbb{R}^+ \cup \{0\}$, *and a* **source vertex** $u_0 \in V$.

Find: *For every vertex $v \in V$, a directed path P from u_0 to v such that*

$$w(P) = \sum_{e \in P} w(e)$$

is minimized.

The term **shortest path** really means **minimum weight path**.

We are asked to find n different shortest paths, one for each vertex $v \in V$.

If all edges have weight 1, we can just use *BFS* to solve this problem.

Dijkstra's Algorithm (Main Ideas)

S is a subset of vertices such that the shortest paths from u_0 to all vertices in S are known; initially, $S = \{u_0\}$.

For all vertices $v \in S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v, and all vertices on P_v are in the set S.

For all vertices $v \notin S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v in which all interior vertices are in S.

For $v \neq u_0$, $\pi[v]$ is the **predecessor** of v on the path P_v.

At each stage of the algorithm, we choose $v \in V \backslash S$ so that $D[v]$ is minimized, and then we add v to S.

Then the arrays D and π are updated appropriately.

Dijkstra's Algorithm

Algorithm: *Dijkstra*(G, w, u_0)

$S \leftarrow \{u_0\}$
$D[u_0] \leftarrow 0$
for all $v \in V \backslash \{u_0\}$
  do $\begin{cases} D[v] \leftarrow w(u_0, v) \\ \pi[v] \leftarrow u_0 \end{cases}$
while $|S| < n$
  do $\begin{cases} \text{choose } v \in V \backslash S \text{ such that } D[v] \text{ is minimized} \\ S \leftarrow S \cup \{v\} \\ \textbf{for all } v' \in V \backslash S \\ \quad \textbf{do } \begin{cases} \textbf{if } D[v] + w(v, v') < D[v'] \\ \quad \textbf{then } \begin{cases} D[v'] \leftarrow D[v] + w(v, v') \\ \pi[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$
return (D, π)

Finding the Shortest Paths

Algorithm: *FindPath*(u_0, π, v)

$path \leftarrow v$

$u \leftarrow v$

while $u \neq u_0$

do $\begin{cases} u \leftarrow \pi[u] \\ path \leftarrow u \parallel path \end{cases}$

return $(path)$

Table of Contents

Computing Fibonacci Numbers Inefficiently

Algorithm: *BadFib*(n)

if $n = 0$ **then** $f \leftarrow 0$

else if $n = 1$ **then** $f \leftarrow 1$

else $\begin{cases} f_1 \leftarrow \textit{BadFib}(n-1) \\ f_2 \leftarrow \textit{BadFib}(n-2) \\ f \leftarrow f_1 + f_2 \end{cases}$

return (f);

Computing Fibonacci Numbers More Efficiently

Algorithm: *BetterFib*(n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do $f[i] \leftarrow f[i-1] + f[i-2]$
return $(f[n])$

Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure

Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain **subproblems** of I.

Define Subproblems

Define a set of subproblems $\mathcal{S}(I)$ of the instance I, the solution of which enables the optimal solution of I to be computed. I will be the last or largest instance in the set $\mathcal{S}(I)$.

Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation

Derive a **recurrence relation** on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. Compute these solutions using the recurrence relation in a **bottom-up** fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to I.

0-1 Knapsack

Problem

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An n-tuple $X = [x_1, \ldots, x_n]$, where $x_i \in \{0, 1\}$ for $1 \leq i \leq n$, and*

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

Find: *A feasible solution X that maximizes*

$$\sum_{i=1}^{n} p_i x_i.$$

A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M)$

for $m \leftarrow 0$ **to** M
  do { **if** $m \geq w_1$
    then $P[1, m] \leftarrow p_1$
    else $P[1, m] \leftarrow 0$ }
for $i \leftarrow 2$ **to** n
  do { **for** $m \leftarrow 0$ **to** M
    do { **if** $m < w_i$
      then $P[i, m] \leftarrow P[i-1, m]$
      else $P[i, m] \leftarrow \max\{P[i-1, m-w_i] + p_i, P[i-1, m]\}$ } }
return $(P[n, M])$;

Computing X

Algorithm: *0-1Knapsack*$(p_1, \dots, p_n, w_1, \dots, w_n, M, P)$

$m \leftarrow M$

$p \leftarrow P[n, M]$

for $i \leftarrow n$ **downto** 2

$$\textbf{do} \begin{cases} \textbf{if } p = P[i-1, m] \\ \quad \textbf{then } x_i \leftarrow 0 \\ \quad \textbf{else } \begin{cases} x_i \leftarrow 1 \\ p \leftarrow p - p_i \\ m \leftarrow m - w_i \end{cases} \end{cases}$$

if $p = 0$

  then $x_1 \leftarrow 0$

  else $x_1 \leftarrow 1$

return (X);

0-1 Knapsack

Problem

Knapsack

Instance: **Values** $[v_1, \ldots, v_n]$; **weights** $[w_1, \ldots, w_n]$; *and a* **capacity**, W. *These are all positive integers.*

Feasible solution: *An n-tuple $X = [x_1, \ldots, x_n]$, where $x_i \in \{0, 1\}$ for $1 \leq i \leq n$, and*

$$\sum_{i=1}^{n} w_i x_i \leq W.$$

Find: *A feasible solution X that maximizes*

$$\sum_{i=1}^{n} v_i x_i.$$

A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: *0-1Knapsack*$(w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

for $w \leftarrow 0$ **to** W
  do $\{M[0,w] \leftarrow 0$
for $i \leftarrow 1$ **to** n
  do $\begin{cases} \textbf{for } w \leftarrow 0 \textbf{ to } W \\ \quad \textbf{do } \begin{cases} \textbf{if } w_i > w \\ \quad \textbf{then } M[i,w] \leftarrow M[i-1,w] \\ \quad \textbf{else } M[i,w] \leftarrow \max\{M[i-1,w], v_i + M[i-1,w-w_i]\} \end{cases} \end{cases}$
return $(M[n,W])$;

Computing X

Algorithm: *0-1Knapsack*$(w_1, \ldots, w_n, v_1, \ldots, v_n, W, M)$

$w \leftarrow W$

$v \leftarrow M[n, W]$

for $i \leftarrow n$ **downto** 2

do $\begin{cases} \textbf{if } v = M[i-1, w] \\ \quad \textbf{then } x_i \leftarrow 0 \\ \quad \textbf{else } \begin{cases} x_i \leftarrow 1 \\ v \leftarrow v - v_i \\ w \leftarrow w - w_i \end{cases} \end{cases}$

if $v = 0$

  then $x_1 \leftarrow 0$

  else $x_1 \leftarrow 1$

return (X);

Longest Common Subsequence

Problem

Longest Common Subsequence

Instance: *Two sequences* $X = (x_1, \ldots, x_m)$ *and* $Y = (y_1, \ldots, y_n)$ *over some finite alphabet* Γ.

Find: *A maximum length sequence* Z *that is a subsequence of both* X *and* Y.

$Z = (z_1, \ldots, z_k)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, Z is a subsequence of Y if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Computing the Length of the LCS of X and Y

Algorithm: *LCS1*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m

$$\textbf{do} \begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do} \begin{cases} \textbf{if } x_i = y_j \quad \textbf{then } c[i, j] \leftarrow c[i-1, j-1] + 1 \\ \quad \textbf{else } c[i, j] \leftarrow \max\{c[i, j-1], c[i-1, j]\} \end{cases} \end{cases}$$

return $(c[m, n])$;

Finding the LCS of X and Y

Algorithm: *LCS2*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m **do** $c[i,0] \leftarrow 0$
for $j \leftarrow 0$ **to** n **do** $c[0,j] \leftarrow 0$
for $i \leftarrow 1$ **to** m

$$\textbf{do} \begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do} \begin{cases} \textbf{if } x_i = y_j \\ \quad \textbf{then} \begin{cases} c[i,j] \leftarrow c[i-1,j-1]+1 \\ \pi[i,j] \leftarrow \mathrm{UL} \end{cases} \\ \textbf{else if } c[i,j-1] > c[i-1,j] \\ \quad \textbf{then} \begin{cases} c[i,j] \leftarrow c[i,j-1] \\ \pi[i,j] \leftarrow \mathrm{L} \end{cases} \\ \textbf{else} \begin{cases} c[i,j] \leftarrow c[i-1,j] \\ \pi[i,j] \leftarrow \mathrm{U} \end{cases} \end{cases} \end{cases}$$

return (c, π);

Finding the LCS

Algorithm: *FindLCS*(c, π, v)

$seq \leftarrow ()$
$i \leftarrow m$
$j \leftarrow n$
while $\min\{i, j\} > 0$
do $\begin{cases} \textbf{if } \pi[i,j] = \mathrm{UL} \\ \quad \textbf{then } \begin{cases} seq \leftarrow x_i \parallel seq \\ i \leftarrow i - 1 \\ j \leftarrow j - 1 \end{cases} \\ \textbf{else if } \pi[i,j] = \mathrm{L} \ \ \textbf{then } j \leftarrow j - 1 \\ \textbf{else } i \leftarrow i - 1 \end{cases}$
return (seq)

All-Pairs Shortest Paths

Problem

All-Pairs Shortest Paths

Instance: *A directed graph* $G = (V, E)$, *and a* **weight matrix** W, *where* $W[i, j]$ *denotes the weight of edge* ij, *for all* $i, j \in V$, $i \neq j$.

Find: *For all pairs of vertices* $u, v \in V$, $u \neq v$, *a directed path* P *from* u *to* v *such that*

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in G.

First Solution

Algorithm: *SlowAllPairsShortestPath*(W)

$L_1 \leftarrow W$
for $m \leftarrow 2$ **to** $n - 1$
  do {
    for $i \leftarrow 1$ **to** n
      do {
        for $j \leftarrow 1$ **to** n
          do {
            $\ell \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
              do $\ell \leftarrow \min\{\ell, L_{m-1}[i,k] + W[k,j]\}$
            $L_m[i,j] \leftarrow \ell$
          }
      }
  }
return (L_{n-1})

Second Solution

Algorithm: *FasterAllPairsShortestPath*(W)

$L_1 \leftarrow W$
$m \leftarrow 2$
while $m < n - 1$
  do {
    for $i \leftarrow 1$ **to** n
      do {
        for $j \leftarrow 1$ **to** n
          do {
            $\ell \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
              do $\ell \leftarrow \min\{\ell, L_{m/2}[i,k] + L_{m/2}[k,j]\}$
            $L_m[i,j] \leftarrow \ell$
          }
      }
    $m \leftarrow 2m$
  }
return (L_m)

Third Solution

Algorithm: *FloydWarshall*(W)

$D_0 \leftarrow W$

for $m \leftarrow 1$ **to** n

do $\begin{cases} \textbf{for } i \leftarrow 1 \textbf{ to } n \\ \textbf{do } \begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } n \textbf{ do} \\ D_m[i,j] \leftarrow \min\{D_{m-1}[i,j], D_{m-1}[i,m] + D_{m-1}[m,j]\} \end{cases} \end{cases}$

return (D_n)

Table of Contents

Decision Problems

Decision Problem: Given a problem instance I, answer a certain question "yes" or "no".

Problem Instance: Input for the specified problem.

Problem Solution: Correct answer ("yes" or "no") for the specified problem instance. I is a **yes-instance** if the correct answer for the instance I is "yes". I is a **no-instance** if the correct answer for the instance I is "no".

Size of a problem instance: *Size*(I) is the number of bits required to specify (or encode) the instance I.

The Complexity Class P

Algorithm Solving a Decision Problem: An algorithm *A* is said to **solve** a decision problem Π provided that *A* finds the correct answer ("yes" or "no") for every instance I of Π in finite time.

Polynomial-time Algorithm: An algorithm *A* for a decision problem Π is said to be a **polynomial-time algorithm** provided that the complexity of *A* is $O(n^k)$, where k is a positive integer and $n = \mathsf{Size}(I)$.

The Complexity Class P denotes the set of all decision problems that have polynomial-time algorithms solving them. We write $\Pi \in P$ if the decision problem Π is in the complexity class *P*.

Cycles in Graphs

Problem

Cycle

Instance: *An undirected graph* $G = (V, E)$.

Question: *Does* G *contain a cycle?*

Problem

Hamiltonian Cycle

Instance: *An undirected graph* $G = (V, E)$.

Question: *Does* G *contain a hamiltonian cycle?*

A **hamiltonian cycle** is a cycle that passes through every vertex in V exactly once.

Knapsack Problems

Problem

0-1 Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in \{0, 1\}^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$?

Problem

Rational Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in [0, 1]^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$?

Certificates

Certificate: Informally, a certificate for a yes-instance I is some "extra information" C which makes it easy to **verify** that I is a yes-instance.

Certificate Verification Algorithm: Suppose that *Ver* is an algorithm that verifies certificates for yes-instances. Then *Ver*(I, C) outputs "yes" if I is a yes-instance and C is a valid certificate for I. If *Ver*(I, C) outputs "no", then either I is a no-instance, or I is a yes-instance and C is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm *Ver* is a polynomial-time certificate verification algorithm if the complexity of *Ver* is $O(n^k)$, where k is a positive integer and $n =$ *Size*(I).

The Complexity Class NP

Certificate Verification Algorithm for a Decision Problem: A certificate verification algorithm *Ver* is said to **solve** a decision problem Π provided that

- **for every** yes-instance I, **there exists** a certificate C such that *Ver*(I, C) outputs "yes".
- **for every** no-instance I and **for every** certificate C, *Ver*(I, C) outputs "no".

The Complexity Class NP denotes the set of all decision problems that have polynomial-time certificate verification algorithms solving them. We write $\Pi \in$ *NP* if the decision problem Π is in the complexity class *NP*.

Certificate Verification Algorithm for Hamiltonian Cycle

A certificate consists of an n-tuple, $X = [x_1, \ldots, x_n]$, that might be a hamiltonian cycle for a given graph $G = (V, E)$ (where $n = |V|$).

Algorithm: *Hamiltonian Cycle Certificate Verification*(G, X)

$flag \leftarrow$ **true**
$Used \leftarrow \{x_1\}$
$j \leftarrow 2$
while $(j \leq n)$ **and** $flag$
do $\begin{cases} flag \leftarrow (x_j \notin Used) \textbf{ and } (\{x_{j-1}, x_j\} \in E) \\ \textbf{if } (j = n) \textbf{ then } flag \leftarrow flag \textbf{ and } (\{x_n, x_1\} \in E) \\ Used \leftarrow Used \cup \{x_j\} \end{cases}$
return $(flag)$

The Complexity Class ExpTime

An algorithm is an **exponential-time** algorithm if its running time is $O(2^{p(n)})$, where $p(n)$ is a polynomial in n and $n = \mathit{Size}(I)$.

The Complexity Class ExpTime denotes the set of all decision problems that have exponential-time algorithms solving them. We write $\Pi \in \mathit{ExpTime}$ if the decision problem Π is in the complexity class *ExpTime*.

Theorem

$P \subseteq NP \subseteq \mathit{ExpTime}$.

Polynomial-time Reductions

For a decision problem Π, let $\mathcal{I}(\Pi)$ denote the set of all instances of Π. Let $\mathcal{I}_{\text{yes}}(\Pi)$ and $\mathcal{I}_{\text{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of Π.

Suppose that Π_1 and Π_2 are decision problems. We say that there is a **polynomial-time reduction** from Π_1 to Π_2 (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \rightarrow \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

- $f(I)$ is computable in polynomial time (as a function of *size*(I), where $I \in \mathcal{I}(\Pi_1)$)
- if $I \in \mathcal{I}_{\text{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{yes}}(\Pi_2)$
- if $I \in \mathcal{I}_{\text{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{no}}(\Pi_2)$

Two Graph Theory Problems

Problem

Clique-Dec

Instance: *An undirected graph* $G = (V, E)$ *and an integer* k*, where* $1 \leq k \leq |V|$.

Question: *Does* G *contain a clique of size* $\geq k$*? (A* **clique** *is a subset of vertices* $W \subseteq V$ *such that* $uv \in E$ *for all* $u, v \in W$, $u \neq v$*.)*

Problem

Vertex Cover-Dec

Instance: *An undirected graph* $G = (V, E)$ *and an integer* k*, where* $1 \leq k \leq |V|$.

Question: *Does* G *contain a vertex cover of size* $\leq k$*? (A* **vertex cover** *is a subset of vertices* $W \subseteq V$ *such that* $\{u, v\} \cap W \neq \emptyset$ *for all edges* $uv \in E$*.)*

Clique-Dec $\leq_P$ Vertex-Cover-Dec

Suppose that $I = (G, k)$ is an instance of **Clique-Dec**, where $G = (V, E)$, $V = \{v_1, \dots, v_n\}$ and $1 \leq k \leq n$.

Construct an instance $f(I) = (H, \ell)$ of **Vertex Cover-Dec**, where $H = (V, F)$, $\ell = n - k$ and

$$v_i v_j \in F \Leftrightarrow v_i v_j \notin E.$$

H is called the **complement** of G, because every edge of G is a non-edge of H and every non-edge of G is an edge of H.

Properties of Polynomial-time Reductions

Suppose that $\Pi_1, \Pi_2, \ldots$ are decision problems.

Theorem

If $\Pi_1 \leq_P \Pi_2$ *and* $\Pi_2 \in$ *P, then* $\Pi_1 \in$ *P.*

Theorem

$\Pi_1 \leq_P \Pi_2$ *and* $\Pi_2 \leq_P \Pi_3$*, then* $\Pi_1 \leq_P \Pi_3$*.*

The Complexity Class **NPC**

The complexity class *NPC* denotes the set of all decision problems Π that satisfy the following two properties:

- $\Pi \in$ *NP*
- For all $\Pi' \in$ *NP*, $\Pi' \leq_P \Pi$.

NPC is an abbreviation for **NP-complete**.

Theorem

If $P \cap NPC \neq \emptyset$, then $P = NP$.

Theorem

If $P = NP$, then $P = NP = NPC$.

Satisfiability Problems

Problem

Satisfiability

Instance: *A boolean circuit $\mathcal{C}$ having n boolean inputs, $x_1, \ldots, x_n$, and one boolean output. $\mathcal{C}$ contains* and, or *and* not *gates.*

Question: *Is there a* **truth assignment** $t : \{x_1, \ldots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ *such that $\mathcal{C}$ outputs* true*?*

Problem

CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables $x_1, \ldots, x_n$, such that F is the* **conjunction** *(logical "and") of m* **clauses***, where each clause is the* **disjunction** *(logical "or") of literals. (A* **literal** *is a boolean variable or its negation.)*

Question: *Is there a truth assignment such that F evaluates to* true*?*

Cook's Theorem

Cook's Theorem proves that at least one NP-complete problem exists:

Theorem

Satisfiability $\in$ *NPC*.

How to reduce an arbitrary problem Π in *NP* to **Satisfiability**:

1. Suppose Π is in *NP*. Then there is a polynomial-time certificate verification algorithm for Π, say *Ver*(I, Cert), where I is an instance of Π and Cert is a certificate.
2. Convert the algorithm *Ver* to a boolean circuit $\mathcal{C}$ that performs the same computation.
3. Given an instance I of Π, fix (i.e., hardwire) the boolean inputs to $\mathcal{C}$ that correspond to the given instance I. The resulting circuit is defined to be $f(I)$.

Proving Problems NP-complete

Now, given any NP-complete problem, say Π_1, other problems in *NP* can be proven to be NP-complete via polynomial reductions **from** Π_1, as stated in the following theorem:

Theorem

Suppose that the following conditions are satisfied:

- $\Pi_1 \in$ *NPC,*
- $\Pi_1 \leq_P \Pi_2$*, and*
- $\Pi_2 \in$ *NP.*

Then $\Pi_2 \in$ *NPC.*

Satisfiability $\leq_P$ CNF-Satisfiability

Suppose that $\mathcal{C}$ is an instance of **SAT**, where $\mathcal{C}$ has inputs $X = \{x_1, \ldots, x_n\}$.

Denote the gates in $\mathcal{C}$ by $G_1, \ldots, G_m$, where G_m is the output gate.

We will construct $f(I)$, which will be an instance of **CNF-SAT**.

The instance $f(I)$ has boolean variables $X' = X \cup \mathcal{G}$, where $\mathcal{G} = \{g_1, \ldots, g_m\}$.

The boolean variable $g_i \in \mathcal{G}$ will "correspond" to the gate G_i in I.

Now, we can construct the clauses in $f(I)$:

- Suppose G_i is an or gate, say "$x \mathrm{~or~} y$". We create three clauses in $f(I)$:

$$(g_i \vee \overline{x}), (g_i \vee \overline{y}), (\overline{g_i} \vee x \vee y).$$

Satisfiability $\leq_P$ CNF-Satisfiability (cont.)

- Suppose G_i is an and gate, say "x and y". We create three clauses in $f(I)$:

$$(\overline{g_i} \vee x), (\overline{g_i} \vee y), (g_i \vee \overline{x} \vee \overline{y}).$$

- Suppose G_i is a not gate, say "not x". We create two clauses in $f(I)$:

$$(g_i \vee x), (\overline{g_i} \vee \overline{x}).$$

- Suppose G_i is a true gate (i.e., it is hard-wired to output the value T). We create one clause in $f(I)$:

$$(g_i).$$

- Suppose G_i is a false gate (i.e., it is hard-wired to output the value F). We create one clause in $f(I)$:

$$(\overline{g_i}).$$

- Finally, we construct one additional clause in $f(I)$, namely the clause (g_m).

More Satisfiability Problems

Problem

3-CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **three** *literals.*

Question: *Is there a truth assignment such that F evaluates to* true*?*

Problem

2-CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **two** *literals.*

Question: *Is there a truth assignment such that F evaluates to* true*?*

CNF-Satisfiability $\leq_P$ 3-CNF-Satisfiability

Suppose that $(X, \mathcal{C})$ is an instance of **CNF-SAT**, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{C} = \{C_1, \ldots, C_m\}$. For each C_j, do the following:

case 1 If $|C_j| = 1$, say $C_j = \{z\}$, construct four clauses

$$\{z, a, b\}, \{z, a, \overline{b}\}, \{z, \overline{a}, b\}, \{z, \overline{a}, \overline{b}\}.$$

case 2 If $|C_j| = 2$, say $C_j = \{z_1, z_2\}$, construct two clauses

$$\{z_1, z_2, c\}, \{z_1, z_2, \overline{c}\}.$$

case 3 If $|C_j| = 3$, then leave C_j unchanged.

case 4 If $|C_j| \geq 4$, say $C_j = \{z_1, z_2, \ldots, z_k\}$, then construct $k-2$ new clauses

$$\{z_1, z_2, d_1\}, \{\overline{d_1}, z_3, d_2\}, \{\overline{d_2}, z_4, d_3\}, \ldots,$$
$$\{\overline{d_{k-4}}, z_{k-2}, d_{k-3}\}, \{\overline{d_{k-3}}, z_{k-1}, z_k\}.$$

3-CNF-Satisfiability $\leq_P$ Clique

Let I be the instance of **3-CNF-SAT** consisting of n variables, $x_1, \ldots, x_n$, and m clauses, $C_1, \ldots, C_m$. Let $C_i = \{z_1^i, z_2^i, z_3^i\}$, $1 \leq i \leq m$.

Define $f(I) = (G, k)$, where $G = (V, E)$ according to the following rules:

- $V = \{v_j^i : 1 \leq i \leq m, 1 \leq j \leq n\}$,
- $v_j^i v_{j'}^{i'} \in E$ if and only if $i \neq i'$ and $z_j^i \neq \overline{z_{j'}^{i'}}$, and
- $k = m$.

Subset Sum and Partition

Problem

Subset Sum

Instance: *A list of* **sizes** $S = [s_1, \ldots, s_n]$*; and a* **target sum**, TS. *These are all positive integers.*

Question: *Does there exist a subset* $J \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in J} s_i = TS$*?*

Problem

Partition

Instance: *A list of* **sizes** $S = [s_1, \ldots, s_n]$. *These are all positive integers.*

Question: *Can* $\{1, \ldots, n\}$ *be* **partitioned** *into two subsets* J_1 *and* J_2 *such that* $\sum_{i \in J_1} s_i = \sum_{i \in J_2} s_i$*?*

3-CNF-Satisfiability $\leq_P$ Subset Sum

Let I be the instance of **3-CNF-SAT** consisting of n variables, $x_1, \ldots, x_n$, and m clauses, $C_1, \ldots, C_m$.

We construct a $2(n+m)$ by $n+m$ integer-valued matrix M.

The first $2n$ rows of M are indexed by the $2n$ literals $x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$ and the last $2m$ rows are named $r_1, r_1', \ldots, r_m, r_m'$.

The columns are named $x_1, \ldots, x_n, C_1, \ldots, C_m$.

Define M as follows:

$$\begin{array}{ll} M[z_i, x_j] = 1 & \text{if } z_i = x_j \text{ or } z_i = \overline{x_j} \\ M[z_i, C_j] = 1 & \text{if } z_i \in C_j \\ M[r_j, C_j] = 1 & \text{for } j = 1, \ldots, m \\ M[r_j', C_j] = 2 & \text{for } j = 1, \ldots, m \\ M[i, j] = 0 & \text{otherwise.} \end{array}$$

3-CNF-Satisfiability $\leq_P$ Subset Sum (cont.)

Each row i of M is interpreted as an $(n+m)$-digit (base-10) integer, s_i.

The target sum is $TS = \underbrace{11\cdots1}_{n}\underbrace{44\cdots4}_{m}$.

Define $f(I)$ to be the instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_{2m+2n}]$ and target sum TS.

Subset Sum $\leq_P$ 0-1 Knapsack

Let I be an instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_n]$ and target sum TS.

Define

$$
\begin{aligned}
p_i &= s_i,\ 1 \leq i \leq n \\
w_i &= s_i,\ 1 \leq i \leq n \\
M &= TS \\
T &= TS.
\end{aligned}
$$

Then define $f(I)$ to be the instance of **0-1 Knapsack** consisting of profits $[p_1, \ldots, p_n]$, weights $[w_1, \ldots, w_n]$, capacity M and target profit T.

Subset Sum $\leq_P$ Partition

Let I be an instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_n]$ and target sum TS.

Denote

$$S = \sum_{i=1}^{n} s_i.$$

Then define $f(I)$ to be the instance of **Partition** consisting of $n + 2$ sizes

$$[s_1, \ldots, s_n, TS + 1, S - TS + 1].$$

Reductions among NP-complete Problems (summary)

$$
\begin{array}{ccccc}
\textbf{SAT} & & & & \\
\Downarrow & & & & \\
\textbf{CNF-SAT} & & & & \\
\Downarrow & & & & \\
\textbf{3-CNF-SAT} & \Longrightarrow & \textbf{Subset Sum} & \Longrightarrow & \textbf{Partition} \\
\Downarrow & & \Downarrow & & \\
\textbf{Clique} & & \textbf{0-1 Knapsack} & & \\
\Downarrow & & & & \\
\textbf{Vertex Cover} & & & &
\end{array}
$$

In the above diagram, arrows denote polynomial reductions.

Polynomial-time Turing Reductions

Suppose Π_1 and Π_2 are problems (not necessarily decision problems). A (hypothetical) algorithm $\mathbf{A_2}$ to solve Π_2 is called an **oracle** for Π_2.

Suppose that $\mathbf{A}$ is an algorithm that solves Π_1, assuming the existence of an oracle $\mathbf{A_2}$ for Π_2. ($\mathbf{A_2}$ is used as a subroutine within the algorithm $\mathbf{A}$.)

Then we say that $\mathbf{A}$ is a **Turing reduction** from Π_1 to Π_2, denoted $\Pi_1 \leq^T \Pi_2$.

A Turing reduction $\mathbf{A}$ is a **polynomial-time Turing reduction** if the running time of $\mathbf{A}$ is polynomial, under the assumption that the oracle $\mathbf{A_2}$ has **unit cost** running time.

If there is a polynomial-time Turing reduction from Π_1 to Π_2, we write $\Pi_1 \leq_P^T \Pi_2$.

Properties of Polynomial-time Turing Reductions

Theorem

If $\Pi_1 \leq_P \Pi_2$ *then* $\Pi_1 \leq_P^T \Pi_2$.

Theorem

If $\Pi_1 \leq_P^T \Pi_2$ *and* $\Pi_2 \leq_P^T \Pi_3$, *then* $\Pi_1 \leq_P^T \Pi_3$.

Theorem

If $\Pi_1 \leq_P^T \Pi_2$ *and* Π_2 *is solvable in polynomial time then* Π_1 *is solvable in polynomial time.*

Corollary

Suppose that Π_1 *and* Π_2 *are decision problems. If* $\Pi_1 \leq_P^T \Pi_2$ *and* $\Pi_2 \in$ *P, then* $\Pi_1 \in$ *P.*

Travelling Salesperson Problems

Problem

TSP-Optimization

Instance: *A graph G and edge weights $w : E \to \mathbb{Z}^+$.*

Find: *A hamiltonian cycle H in G such that $w(H) = \sum_{e \in H} w(e)$ is minimized.*

Problem

TSP-Optimal Value

Instance: *A graph G and edge weights $w : E \to \mathbb{Z}^+$.*

Find: *The minimum T such that there exists a hamiltonian cycle H in G with $w(H) = T$.*

Problem

TSP-Decision

Instance: *A graph G, edge weights $w : E \to \mathbb{Z}^+$, and a target T.*

Question: *Does there exist a hamiltonian cycle H in G with $w(H) \leq T$?*

TSP-Optimal Value $\leq_P^T$ TSP-Dec

Algorithm: *TSP-Reduction1*(G, w)
external *TSP-Dec-Solver*
$hi \leftarrow \sum_{e \in E} w(e)$
$lo \leftarrow 0$
if not *TSP-Dec-Solver*(G, w, hi) **then return** (∞)
while $hi > lo$
do $\begin{cases} mid \leftarrow \left\lfloor \frac{hi+lo}{2} \right\rfloor \\ \textbf{if } \textit{TSP-Dec-Solver}(G, w, mid) \textbf{ then } hi \leftarrow mid \\ \quad \textbf{else } lo \leftarrow mid + 1 \end{cases}$
return (hi)

TSP-Optimization $\leq_P^T$ TSP-Dec

Algorithm: *TSP-Reduction2*$(G = (V, E), w)$
external *TSP-OptimalValue-Solver*, *TSP-Dec-Solver*
$T^* \leftarrow$ *TSP-OptimalValue-Solver*(G, w)
if $T^* = \infty$ **then return** ("no hamiltonian cycle exists")
$w_0 \leftarrow w$
$H \leftarrow \emptyset$
for all $e \in E$
do $\begin{cases} w_0[e] \leftarrow \infty \\ \textbf{if not } \textit{TSP-Dec-Solver}(G, w_0, T^*) \\ \quad \textbf{then } w_0[e] \leftarrow w[e]; \quad H \leftarrow H \cup \{e\} \end{cases}$
return (H)

NP-hard Problems

Suppose Π is a problem (not necessarily a decision problem).

Π is **NP-hard** if there exists a problem $\Pi' \in \mathit{NPC}$ such that $\Pi' \leq_P^T \Pi$.

Note: A more restrictive definition is given in the textbook [CLRS]. The definition given here is the standard definition.

If Π is an optimization or optimal value problem and the related decision problem is NP-complete, then Π is NP-hard.

This is because there are trivial poly-time Turing reductions from the decision problem to the optimal value and optimization versions of the problem.

Theorem

If Π is NP-hard and Π is solvable in polynomial time, then $P = NP$.