CS6212-10, Computing Algorithms I
Lecture Note

1 Introduction

- Algorithm
  - What is algorithm?
  - Validation or correctness
  - Analysis of complexity (time and space)
- How good is the approximation algorithm?
- Offline or online algorithms
- Centralized or distributed algorithms
- Parallel algorithms

1.1 Max-Min Problem

Given n numbers, find the maximum and the minimum.

Algorithm 1: Find the maximum from the n numbers using $(n-1)$ comparisons; then find the minimum from the remaining $(n-1)$ numbers using $(n-2)$ comparisons. The total number of comparisons used this algorithm is $(2n-3)$.

Algorithm 2: Let $S = \{a_1, \cdots, a_n\}$. Assume for simplicity that $n = 2^k$ for an integer k.

```
MAXMIN(S)
    if |S| = 2
        then { let S = {a, b};
               return(MAX{a, b}, MIN{a, b})}
        else {  divide S into S1 and S2 such that |S1| = |S2|;
               (max1, min1) ← MAXMIN(S1);
               (max2, min2) ← MAXMIN(S2);
               return(MAX{max1, max2}, MIN{min1, min2}) }
```

- Time Complexity of Algorithm 2: Let $T(n)$ denote the number of comparisons used in Algorithm 2 for input of size n. We then have

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ 2T(n/2) + 2 & \text{otherwise} \end{cases}$$

Solving this recursive form, $T(n) = \frac{3n}{2} - 2$.

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 = 2^2T(n/2^2) + 2^2 + 2 \\ &= 2^2(2T(n/2^3) + 2) + 2^2 + 2 = 2^3T(n/2^3) + 2^3 + 2^2 + 2 \\ &\cdots \\ &= 2^{k-1}T(2) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 \\ &= 2^{k-1}T(2) + 2(2^{k-1} - 1) \\ &= n/2 + 2(n/2 - 1) = \frac{3n}{2} - 2 \end{aligned}$$

1.2 Fibonacci Sequence

$0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$;
$f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

1.2.1 Closed Form

$$f_n = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

1.2.2 Implementation

- Recursive implementation: $T(n) > 2^{\frac{n}{2}}$
- Iterative implementation: $T(n) = n$

1.3 Asymptotic Notation

- Ignore all the factors which are dependent on machine and programming languages, and concentrate on the frequency of executing statements.
- Limiting behavior of complexity when the input size increases.

Definition: $f(n) = O(g(n))$ if and only if there exist two constants c (> 0) and n_0 (≥ 1) such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Theorem 1 *Let* $A(n) = a_m n^m + a_{m-1}n^{m-1} + \cdots + a_1 n + a_0$. *Then,* $A(n) = O(n^m)$.

Proof. We will show that there exist constants c and n_0 such that $|A(n)| \leq c|n^m|$ for all $n \geq n_0$. Note that

$$\begin{aligned} A(n) &= n^m(a_m n^0 + a_{m-1}n^{-1} + \cdots + a_1 n^{-(m-1)} + a_0 n^{-m}) \\ |A(n)| &\leq n^m(|a_m|n^0 + |a_{m-1}|n^{-1} + \cdots + |a_1|n^{-(m-1)} + |a_0|n^{-m}) \text{ for all } n \geq 1 \\ &\leq n^m(|a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|) \end{aligned}$$

This, choosing $c = |a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|$ and $n_0 = 1$ gives $A(n) = O(n^m)$. ■

Note: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

Definition: $f(n) = \Omega(g(n))$ if and only if there exist two constants $c(> 0)$ and $n_0(\geq 1)$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Definition: $f(n) = \theta(g(n))$ if and only if there exist three constants c_1 (> 0), c_2 (> 0), and n_0 (≥ 1) such that $c_1 g(n) \leq f(n) \leq cg(n)$ for all $n \geq n_0$, i.e., $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Note: Let $f(n) = O(s(n))$ and $g(n) = O(t(n))$. Then,

$$\begin{aligned} f(n) + g(n) &= O(s(n) + t(n)) \text{ and} \\ f(n) \cdot g(n) &= O(s(n) \cdot t(n)) \end{aligned}$$

It is not necessarily true that

$$\begin{aligned} f(n) - g(n) &= O(s(n) - t(n)) \text{ or} \\ f(n)/g(n) &= O(s(n)/t(n)) \end{aligned}$$

counter example:

1. $f(n) = 2n^2$, $s(n) = n^2 + 2n$, $g(n) = n^2$, and $t(n) = n^2 + n$. Then, $f(n) - g(n) = n^2$ but $s(n) - t(n) = n$
2. $f(n) = 3n^3$, $s(n) = n^3$, $g(n) = n$, and $t(n) = n^2$. Then, $\frac{f(n)}{g(n)} = 3n^2$, but $\frac{s(n)}{t(n)} = n$.

Note:

1. if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
2. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

* Answer "true" or "false".

(a) $\log_2 n = O(\log_5 n)$. *true.*

(b) $\frac{n^3}{\log_e n} = O(n^{2.81})$. *false.*

2 Data Structure

2.1 Binary Search Trees

We assume that the root of a binary tree is on level 1.

Lemma 1 *The maximum number of nodes on level i of a binary tree is 2^{k-1}. Also, the maximum number of nodes in a binary tree of depth k is 2^{k-1} for $k \geq 1$.*

2.1.1 Definition

A *binary search tree* is a binary tree satisfying the following.

1. Every element has a key.
2. The keys (if any) in the left subtree are smaller than or equal to the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Many interesting problems from this definition can be formulated using recursive formula. Following are some examples.

- Given a list A of n distinct keys, how many binary search trees can be formed from A?

  Note: The number $f(n)$ of binary search trees can be formulated as
  $f(n) = \sum_{k=1}^{n} f(k-1)f(n-k)$ where $f(0) = f(1) = 1$.

- Given a list A of n distinct keys, how many binary search trees can be formed such that in any subtree, the number of nodes in its right subtree is equal to or one less than the number of nodes in its left subtree?

  Note: The number $g(n)$ of such binary search trees can be formulated as
  $g(n) = g(\lfloor \frac{n}{2} \rfloor)g(\lfloor \frac{n-1}{2} \rfloor)$,
  where $g(0) = g(1) = 1$. Note that for any $n \geq 0$, we will then have $g(n) = 1$, i.e., such a binary tree is uniquely defined.

- Given a list A of n distinct keys, how many binary search trees can be formed such that in any subtree, the difference between the numbers of nodes in its left and right subtrees is at most by one?

  Note: The number $h(n)$ of such binary search trees can be formulated as
  $h(n) = 2h(\lfloor \frac{n}{2} \rfloor)h(\lfloor \frac{n-1}{2} \rfloor)$ if n is even, and $h(n) = h(\frac{n-1}{2})h(\frac{n-1}{2})$ otherwise.

- Given a list A of n distinct keys, how many binary search trees can be formed such that in any subtree, the depths of the left and the right subtrees differ by at most one?

- Given a list A of n distinct keys, how many binary search trees can be formed such that in any subtree, the numbers of nodes in the left and the right subtrees differ by at most k, where k is a given integer?
- Given a list A of n distinct keys, how many binary search trees can be formed such that in any subtree, the depths of the left and the right subtrees differ by at most k, where k is a given integer?
- Other variations may be formulated when the given n keys are not necessarily distinct from each other.

2.1.2 Operations

- Searching a binary search tree
- Insertion into a binary search tree
- Deletion from a binary search tree

2.2 Sets and Disjoint Set Union

We consider disjoint sets and wish to perform two operations on these sets.

1. **Union:** If S_i and S_j are two disjoint sets, then their union is $S_i \cup S_j = \{x \mid x\}$ is either in S_i or S_j, and the sets S_i and S_j do not exist independently.
2. **Find(i):** Given an element i, find the set containing i.

We assume that each set is represented using a directed tree such that nodes are linked from children to parents. Three algorithms are considered to perform the union operation $UNION(S_i, S_j)$.

Algorithm1: Make the root of S_j be a son of the root of S_i.

$UNION$: $O(1)$; $FIND$: $O(n)$

Algorithm2: Make every node in S_j be a son of the root of S_i.

$UNION$: $O(n)$; $FIND$: $O(1)$

Algorithm3: Make the root of the smaller tree (i.e., a tree with less number of nodes) be a son of the root of the larger tree. (If two trees have the same number of nodes, choose arbitrarily.)

$UNION : O(1)$; $FIND : O(\log n)$

Theorem 2 *Let T be a tree with n nodes created using Algorithm3. The depth of T is then $\leq \lfloor \log n \rfloor + 1$, where the depth of the root is defined to be 1.*

Proof: Using induction on n. If $n = 1$, it is trivial. Assume that the claim is true for all trees with i nodes where $i \leq n-1$. Let T be a tree with n nodes, and let $UNION(S_k, S_j)$ be the last union operation performed. Let m and $n-m$ denote the number of nodes in S_k and S_j, respectively. Without loss of any generality, assume that $1 \leq m \leq n/2$. We now consider two cases:

Case1: $depth(T) = depth(k)$**:** Note that $depth(k) \leq \lfloor \log(n-m) \rfloor + 1 \leq \lfloor \log n \rfloor + 1$.

Case2: $depth(T) > depth(k)$**:** Note that the depth of T must be then $depth(j)+1$, and $depth(j) \leq \lfloor \log m \rfloor + 1 \leq \lfloor \log \frac{n}{2} \rfloor + 1 \leq \lfloor \log n \rfloor$. Hence, $depth(T) \leq \lfloor \log n \rfloor + 1$. ■

3 Divide and Conquer

A general method of the divide-and-conquer technique is described as:

1. Partition the problem into smaller parts of the same type of the original problem.
2. Find solutions for the parts.
3. Combine the solutions for the parts into a solution for the whole.

   Note that such an algorithm is described using recursion.

3.1 Min-Max Problem

3.2 Ordered Search Problem

Given a list of n numbers in non-decreasing order $A = \{a_1, a_2, \cdots, a_n\}$ such that $a_1 \leq a_2 \leq \cdots \leq a_n$ and a number x, the objective is to determine if x is present in the list A.

- Linear Search Algorithm.
- Binary Search Algorithm.

	worst-case	best-case	average-case
successful	$\theta(\log n)$	$\theta(1)$	$\theta(\log n)$
unsuccessful	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$

Implementation: (i) array, or (ii) binary decision tree

Lemma 2 *The number of external nodes in a binary decision tree with n internal nodes is $n + 1$.*

Proof 1:

Proof 2: Note that a binary decision tree is a *complete* binary tree, i.e., level i ($1 \leq i \leq k-1$) is full, and level k may/may not be full, where k is the depth of the tree.

Lemma 3 *Let T be a binary decision tree with n internal nodes with depth k. Then there are 2^{i-1} nodes in each level i, $1 \leq i \leq k-1$, and the number of nodes at level k is at least 1 and at most 2^{k-1}.*

Lemma 4 *The depth of a binary decision tree with n internal nodes is equal to $\lceil \log(n+1) \rceil$.*

Proof: Let k denote the depth of a binary decision tree with n internal nodes. Then, $2^{k-1} - 1 < n \leq 2^k - 1$, implying that $2^{k-1} < n+1 \leq 2^k$. Hence, $k-1 < \log_2(n+1) \leq k$. Therefore, $\lceil \log(n+1) \rfloor = k$. ■

Remarks: (i) The number of comparisons for an unsuccessful search is either $k-1$ or k. (ii) The number of comparisons for a successful search for a node at level i ($1 \leq i \leq k$) is i.

Average case time complexity analysis:

Let I denote the *internal path length* defined to be the sum of the *distance* of all internal nodes from the root, and E denote the *external path length*, defined to the sum of the *distance* of all external nodes from the root. We then have
(1) $E = I + 2n$ (see Lemma(*) below for a proof).

Let $s(n)$ and $u(n)$, resp., denote the average number of comparisons in a successful and unsuccessful search. Then,

(2) $s(n) = I/n + 1$ and
(3) $u(n) = E/(n+1)$.

Thus, $I = n(s(n) - 1)$ and $E = (n+1)u(n)$. Hence, $(n+1)u(n) = n(s(n) - 1) + 2n$, implying that $s(n) = (1 + \frac{1}{n})u(n) - 1$.

Therefore, E is proportional to $n \log n$; hence, $u(n)$ is proportional to $\log n$, and $s(n)$ is also proportional to $\log n$.

Lemma(*) $E = I + 2n$.
Proof: Let k be the depth of a binary search tree with n nodes. Note that the number of internal nodes at level i is 2^{i-1}, for $1 \leq i \leq k-1$, and the number of nodes at level k is $n - (2^{k-1} - 1)$. Define $l = n - (2^{k-1} - 1)$. We then have

$$(1) \quad I = \sum_{1 \leq i \leq k-1} (i-1)2^{i-1} + (k-1)l.$$

The numbers of external nodes at levels k and $k+1$ are $(2^{k-1}-l)$ and $2l$, respectively. Thus we have

$$(2) \quad E = (k-1)(2^{k-1}-l) + 2lk.$$

Note that $\sum_{1\leq i\leq k-1}(i-1)2^{i-1} = (k-3)2^{k-1}+2$. (See **(*)**.)

(*) Let $A = \sum_{1\leq i\leq k-1}(i-1)2^{i-1}$. Then, $A = 1\cdot 2 + 2\cdot 2^2 + 3\cdot 2^3 + 4\cdot 2^4 + \cdots + (k-2)2^{k-2}$. By multiplying this by two, we have $2A = 1\cdot 2^2 + 2\cdot 2^3 + 3\cdot 2^4 + \cdots + (k-2)2^{k-1}$. Hence, $A-2A = 2+2^2+\cdots+2^{k-2}-(k-2)2^{k-1}$ which is equal to $2(2^{k-2}-1)-(k-2)2^{k-1} = -2^{k-1}(k-3)-2$. Therefore, we have $A = (k-3)2^{k-1}+2$.

By putting A into (1), we have

$$(3) \quad I = (k-1)2^{k-1} + 2 + (k-1)l.$$

By substituting (2) and (3) into $E-I-2n$, we have

$$\begin{aligned} E-I-2n &= (k-1)(2^{k-1}-l) + 2kl - (k-3)2^{k-1} - 2 - (k-1)l - 2 \\ &= 2\cdot 2^{k-1} + 2kl - 2(k-1)l - 2 - 2n \\ &= 2(l-(n-2^{k-1}+1)) \end{aligned}$$

which is equal to 0. Hence, we have $E = I + 2n$. ■

Binary Search Variations

1. You have n coins (n may be even or odd) such that $n-1$ coins are of the same weight and one has different weight. You have a balance scale: you can put any number of coins on each side of the scale at one time, and it will tell you if the two sides weigh the same, or which side is lighter if they do not weigh the same. Outline an algorithm for finding the coin with different weight for each of the following conditions. The number of weighings using your algorithm in each case should be $O(\log n)$.

(a) You know that one coin weighs more than the others.

(b) You know the weight of one coin is different from others, do not know whether it weighs more or less.

2. You are given two arrays $L1$ and $L2$, each with n keys sorted in increasing order. For simplicity, you may assume that the keys are all distinct from each other.

(a) Describe an $O(\log n)$ time algorithm to find the nth smallest of the $2n$ keys.

(b) Describe an $O(\log n)$ time algorithm to find the $\frac{n}{2}$th smallest (i.e., the median) of the $2n$ keys assuming that n is even.

3. The first n cells of array L contains integers sorted in increasing order. The remaining cells all contain very large integer that we may think of as infinity. The array may be arbitrarily large (you may think of it as infinite), and you do **not** know the value of n. Give an $O(\log n)$ time algorithm to find the position of a given integer x in the array if the array contains x.

4. Let L be a list of numbers in non-decreasing order, and x be a given real numbers. Describe an algorithm that counts the number of elements in L whose values are x. For example, if $L = \{1.3, 2.1, 2.1, 2.1, 2.1, 6.7, 7.5, 7.5, 8.6, 9.0\}$ and $x = 2.1$ then the output of your algorithm should be 4. Your algorithm should run in $O(\log n)$ time.

5. Consider a sequence $x_1, x_2, \cdots, x_n$ of n distinct numbers such that the smallest number is x_i for some unknown i, and sequence $x_i < x_{i+1} < \cdots < x_n < x_1 < \cdots x_{i-1}$. Describe an $O(logn)$ time algorithm to find the kth smallest element for an arbitrary integer k.

6. The input is a set S containing n real numbers, and a real number x.

(a) Design an algorithm to determine whether there are two elements in S whose sum is exactly x. The algorithm should run in $O(n \log n)$ time.

(b) Suppose now that the set S is given in a sorted order. Design an algorithm to solve this problem in $O(n)$ time.

7. Given two sets S_1 and S_2, and a real number x, find whether there exists an element from S_1 and an element from S_2 whose sum is exactly x. The algorithm should run in $O(n \log n)$ time, where n is the total number of elements in $S_1 \cup S_2$.

3.3 Mergesort

- **Time Complexity:** Let $T(n)$ denote the number of comparisons required by the Mergesort algorithm to sort n elements. We then have $T(n) = 1$ for $n = 2$, and $T(n) \leq 2T(\frac{n}{2}) + n$ for $n > 2$. Two cases are considered.
- *Case 1:* $n = 2^k$ for some integer k (i.e., n is a power of 2.)

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + n \\ \leq 2\{2T(\frac{n}{2^2}) + \frac{n}{2}\} + n &= 2^2T(\frac{n}{2^2}) + 2n \\ \leq 2^2\{2T(\frac{n}{2^3}) + \frac{n}{2^2}\} + 2n &= 2^3T(\frac{n}{2^3}) + 3n \\ &\cdots \\ \leq 2^{k-1}T(\frac{n}{2^{k-1}}) + (k-1)n \\ = \frac{n}{2}T(2) + (k-1)n \\ = \frac{n}{2} + (k-1)n \\ = O(n\log n). \end{aligned}$$

- *Case 2:* $2^{k-1} < n < n^k$ for some integer k (i.e., n is not a power of 2.)
  We then note that $T(n) \leq T(2^k)$. This implies that $T(n) = O(n\log n)$.

3.4 Quicksort

The Idea: Given a list A, choose an element t randomly from A (usually, the first element in the list), and return $[A_1, t, A_2]$, where all the elements $\in A_1 \leq t$ & those $\in A_2 > t$.

This procedure is called **Partition(m,p).** The aim of this procedure is to divide the list A into two sub-lists A_1 and A_2 such that all the elements of A_1 are smaller than any element of A_2, and this function returns the partitioning index of the list A, i.e., the index of the last element of A_1.

Partition(m, p) : It is assumed that $A(p) = \infty$, and the elements $A(m), A(m+1), \cdots, A(p-1)$ are rearranged such that initially $t = A(m)$; and after completion $A(q) = t$ for some q, where $m \leq q \leq p-1$; $A(k) \leq t$ for $m \leq k \leq q-1$ and $A(k) > t$ for $q+1 \leq k \leq p-1$.

Subsequently, A_1 an A_2 are sorted by recursive calls to Quicksort.

3.4.1 Algorithm

Partition(m,p):
$t \leftarrow A[m]$
$i \leftarrow m$
$j \leftarrow p$
while $(i < j)$ do
    repeat $i \leftarrow i + 1$ until $A[i] > t$

repeat $j \leftarrow j-1$ until $A[j] \leq t$
if $i < j$ then exchange $A[i], A[j]$
endwhile
exchange $A[m], A[j]$
return j

Quicksort(m,p)
if $m < p$ then {
$j \leftarrow p+1$
Partition(m,j)
Quicksort(m,j-1) (Note that j is the value returned from the Partition.)
Quicksort(j+1,p)
}

Note that initially, $Quicksort(1, n)$ is called to sort n elements.

3.4.2 Time complexity

The running time of quicksort depends on the partitioning of the list. If the partioning is balanced, which means that the two sublists generated are of equal size, then the subsequent recursive sorting on those sublists will take less time. On the other hand, if the partioning is unbalanced, then the quicksort time will be more, the worst case being when a list of n elements is partitioned into $n-1$ and 1 elements.

The time complexity of the partitioning function is the no. of comparisons done, which is $p - m + 2$. This implies $n + 1$ comparisons are required in the worst case if n is the number of elements in the list.

- **Worst Case:** Worst case occurs when at every stage of recursion the partitioning produces partitions of 1 and $n-1$ elements.

$$\begin{aligned} T(n) &= T(n-1) + T(1) + n \\ &= T(n-2) + T(1) + (n-1) + n \\ &= \sum n \\ &= \theta(n^2) \end{aligned}$$

- **Average Case:**

  Let $T_{ave}(n)$ denote the average-case time complexity of Quicksort algorithm for n numbers. Note that $T_{ave}(0) = T_{ave}(1) = 0$.

  After calling $Partition(1, n+1)$, suppose $A(1)$ gets placed at position j. We then have two subfiles of sizes $j-1$ and $n-j$. The average case time complexity of sorting these two files are then $T_{ave}(j-1)$ and $T_{ave}(n-j)$. Since the number of comparisons required by

Partition$(1, n+1)$ is $n+1$, we have

$$(1)\quad T_{ave}(n) = (n+1) + \frac{1}{n}\sum_{j=1}^{n}(T_{ave}(j-1) + T_{ave}(n-j)).$$

Multiplying both sides of (1) by n we have

$$(2)\quad nT_{ave}(n) = n(n+1) + \sum_{j=1}^{n}(T_{ave}(j-1) + T_{ave}(n-j)).$$

By replacing n by $n-1$ in (2), we have

$$(3)\quad (n-1)T_{ave}(n-1) = n(n-1) + \sum_{j=1}^{n-1}(T_{ave}(j-1) + T_{ave}(n-j-1)),$$

i.e.,

$$(4)\quad (n-1)T_{ave}(n-1) = n(n-1) + 2(T_{ave}(0) + T_{ave}(1) + \cdots + T_{ave}(n-2)).$$

By subtracting (4) from (2), we have

$$nT_{ave}(n) - (n-1)T_{ave}(n-1) = 2n + 2T_{ave}(n-1).$$

This implies that

$$nT_{ave}(n) = nT_{ave}(n-1) + T_{ave}(n-1) + 2n,$$

i.e.,

$$\begin{aligned}\frac{T_{ave}(n)}{n+1} &= \frac{T_{ave}(n-1)}{n} + \frac{2}{n+1}\\ &= (\frac{T_{ave}(n-2)}{n-1} + \frac{2}{n}) + \frac{2}{n+1}\\ &\cdots\\ &= \frac{T_{ave}(1)}{2} + 2\sum_{k=3}^{n+1}\frac{1}{k}\\ &= 2\sum_{k=3}^{n+1}\frac{1}{k}.\end{aligned}$$

Note that $\sum_{k=3}^{n+1}\frac{1}{k} = 1/3 + 1/4 + \cdots + 1/(n+1)$.

Therefore, $T_{ave}(n) = O(n\log n)$.

3.4.3 Randomized Quicksort

The algorithm works as follows. When $n > 2$, one of the values is randomly chosen, say it is a_i, and then all of the other values are compared with a_i. Those smaller than a_i are put in a bracket to the left of a_i, and those larger than a_i are put in a bracket to the right of a_i. The algorithm then repeats itself on these brackets, continuing until all values have been sorted.

Let X denote the number of comparisons needed. To compute the expected value $E[X]$ of X, we will first express X as the sum of other random variables in the following manner. To begin, give the following names to the values that are to be sorted: Let 1 denote the smallest, let 2 denote the second smallest, and so on. Then, for $1 \leq i < j \leq n$, let $I(i, j)$ equal 1 if i and j are directly compared, and let it equal 0 otherwise. (Note that any i and j may be directly compared but only once.) Summing these variables over all$i < j$ gives the total number of comparisons. That is,

$$X = \sum_{j=2}^{n} \sum_{i=1}^{j-1} I(i, j)$$

which implies that

$$\begin{aligned} E[X] &= E[\sum_{j=2}^{n} \sum_{i=1}^{j-1} I(i, j)] \\ &= \sum_{j=2}^{n} \sum_{i=1}^{j-1} E[I(i, j)] \\ &= \sum_{j=2}^{n} \sum_{i=1}^{j-1} P\{i \text{ and } j \text{ are ever compared}\} \end{aligned}$$

To determine the probability that i and j are ever compared, note that the values $i, i = 1, \cdots, j-1, j$ will initially be in the same bracket (because all values are initially in the same bracket) and will remain in the same bracket if the number chosen for the first comparison is not between i and j. For instance, if the comparison number is larger than j, then all the values $i, i + 1, \cdots, j - 1, j$ will go in a bracket to the left of the comparison number, and if it is smaller than i then they will go to the right. Thus all values will remain in the same bracket until the first time that one of them is chosen as a comparison value. If this comparison value is neither i nor j, then upon comparison with it, i will go to the left bracket and j will go to the right bracket; consequently, i and j will never be compared. On the other hand, if the comparison value of the set $i, i + 1, \cdots, j - 1, j$ is either i or j, then there will be a direct comparison between i and j. Given that the comparison value is one of the values between i and j, it follows that it is likely to be any of these $j - i + 1$ values; thus, the probability that it is either i or j is $2/(j - i + 1)$. Therefore, we may conclude that

$$P\{i \text{ and } j \text{ are ever compared}\} = \frac{2}{j - i + 1}$$

Consequently, we see that

$$E[X] = \sum_{j=2}^{n} \sum_{i=1}^{j-1} \frac{2}{j - i + 1}$$

$$
\begin{aligned}
&= \sum_{j=2}^{n} \sum_{k=2}^{j} \frac{1}{k} \text{ by letting } k = j - i + 1 \\
&= \sum_{j=2}^{n} \sum_{k=2}^{j} \frac{1}{k} \text{ by interchanging the order of summation} \\
&= 2(n+1) \sum_{k=2}^{n} \frac{1}{k} - 2(n-1)
\end{aligned}
$$

Using the approximation that for large n

$$\sum_{k=2}^{n} \frac{1}{k} \sim \log(n),$$

we see that the quicksort algorithm requires, on average, approximately $2n \log(n)$ comparisons to sort n values.

3.5 Select Problem

Given: an array L containing n keys (keys are not necessarily distinct), and an arbitrary integer k such that $1 \leq k \leq n$.

Objective: to find the k_{th} smallest element in L.

3.5.1 Algorithm

Procedure **Select(L,k)**

1. Divide n elements into $\lfloor n/5 \rfloor$ groups with 5 elements in each group. (The first $5\lfloor n/5 \rfloor$ elements from L will be used in this grouping.) Let m_i $(1 \leq i \leq \lfloor n/5 \rfloor)$ denote the median of group i.

2. Find the median from the list $M = \{m_1, m_2, \cdots, m_{\lfloor n/5 \rfloor}\}$ (denoted by m^*, called *median of medians*) recursively using **Select**. (Note that m^* is the $(\lceil \lfloor n/5 \rfloor / 2 \rceil)_{th}$ smallest element in M.)

3. Compute $L_1 = \{$ keys from L that are smaller than $m^*\}$, $R = \{$ keys that are equal to m^*, and $L_2 = \{$ keys that are larger than M^* $\}$.

4. If $|L_1| < k \leq |L_1| + |R|$
   then return m^*
   else if $k \leq |L_1|$
   then call **Select**(L_1, k)
   else call **Select**$(L_2, k - (|L_1| + |R|))$.

3.5.2 Analysis of Time Complexity

Let $T(n)$ denote the time complexity of the Select algorithm for n elements. Note that Steps 1 and 3 can be done in $O(n)$ time, and Step 2 requires $T(n/5)$ time since there are $\lfloor n/5 \rfloor$ elements in M.

We next show that Step 4 requires at most $T(3n/4)$ time. To do this, we will show that $|L_1| \leq 3n/4$ and $|L_2| \leq 3n/4$. Suppose we arrange elements $L(1), L(2), \cdots, L(5\lfloor n/5 \rfloor)$ as shown below.

We then make the following observations. All elements in area A are no larger than m^*; thus, at least $3\lceil \lfloor n/5 \rfloor /2 \rceil$ elements are less than or equal to m^*. Since $3\lceil \lfloor n/5 \rfloor /2 \rceil \geq 3\lfloor n/5 \rfloor /2 = 1.5\lfloor n/5 \rfloor$. This implies that at most $(n - 1.5\lfloor n/5 \rfloor)$ elements are strictly larger than m^*. Since $n - 1.5\lfloor n/5 \rfloor) \leq n - \frac{1.5(n-4)}{5}$ (because $\lfloor n/5 \rfloor \geq \frac{n-4}{5}$), we have at most $n - \frac{1.5(n-4)}{5} (= 0.7n + 1.2)$ elements that are strictly larger than m^*. Similarly, we can show that there are at most $n - \frac{1.5(n-4)}{5} (= 0.7n + 1.2)$ elements that are strictly less than m^*. Note that $0.7n + 1.2 \leq 3n/4$ for any $n \geq 24$. Thus, Step 4 requires at most $T(3n/4)$ time.

The above discussion implies that $T(n) \leq cn + T(n/5) + T(3n/4)$ for any $n \geq 24$. Using induction on n, it can be shown that $T(n) \leq 20cn$, implying that $T(n) = O(n)$.

3.5.3 Additional Discussion

We are now looking into some details on how two constants 3/4 and 20 were selected in the $O(n)$ analysis when the group size is 5. This approach can be used to obtain $O(n)$ time analysis when group size is $7, 9, 11$, etc. (i.e., an odd number larger than 5.) You can also check that this approach is not going to work when the group size is 3.

We have shown that at most $n - \frac{1.5(n-4)}{5} (= 0.7n + 1.2)$ elements are strictly larger than m^*. Let $0.7n + 1.2 \leq \beta n$ for some β. We will then have

$$T(n) \leq cn + T(n/5) + T(\beta n).$$

In order to show $T(n) \leq \alpha cn$, we must have

$$cn + \alpha c\frac{n}{5} + \alpha c\beta n \leq \alpha cn,$$

implying that

$$1 + \frac{\alpha}{5} + \alpha\beta \leq \alpha,$$

that is,

$$1 + \alpha\beta \leq \frac{4}{5}\alpha.$$

Thus, it must be that $\beta < \frac{4}{5}$. So we have two equations:

$$0 < \beta < \frac{4}{5}$$

and

$$1 + \alpha\beta \leq \frac{4}{5}\alpha$$

There are many possible values satisfying these two equations, e.g., $\beta = 0.75$ and $\alpha = 20$, $\beta = 0.71$ and $\alpha = 12$, etc.

3.5.4 Randomized Select

We are given a set of n values $x_1, x_2, \cdots, x_n$, and our objective is to find the kth smallest of them. It starts by randomly choosing one of the items, compares each of the others to this item, and puts those smaller ones in a bracket to the left, those larger in a bracket to the right. Suppose $r-1$ items are put in the bracket to the left. There are now three possibilities:

(1) $r = k$

(2) $r < k$

(3) $r > k$

In case (1), the kth smallest value is the comparison value, and the algorithm ends. In case (2), because the kth smallest is the $(k-r)$th smallest of the $n-r$ values in the right bracket, the process begins anew with the values in this bracket. In case (3), the process begins anew with a search for the kth smallest if the $r-1$ values in the left bracket.

Let X denote the number if comparisons made by this algorithm. Let 1 denote the smallest value, 2 the second smallest, and so on, and let $I(i,j)$ equal if i and j are ever directly compared, and 0 otherwise. Then,

$$X = \Sigma_{j=2}^{n}\Sigma_{i=1}^{j-1} I(i,j)$$

and

$$E[X] = \Sigma_{j=2}^{n}\Sigma_{i=1}^{j-1} P\{i \text{ and } j \text{ are ever compared}\}$$

To determine the probability that i and j are ever compared, we consider cases:

Case 1: $i < j \leq k$

In this case i, j, k will remain together until one of the values $i, i+1, \cdots, k$ is chosen as the comparison value. If the value chose is either i or jm the pair will be compared; if not, they will not be compared. Since the comparison value is equally likely to be any of these $k-i+1$ values, we see that in this case

$$P\{i \; rmand \; j \text{ are ever compared}\} = \frac{2}{k-i+1}$$

Case 2: $i \leq k < j$

In this case, i, j, k will remain together until one of the $j-i+1$ values $i, i+1, \cdots, j$ is chosen as the comparison value. If the value chosen is either i or j, the pair will be compared.; if not, they will not. Consequently,

$$P\{i \; rmand \; j \text{ are ever compared}\} = \frac{2}{j-i+1}$$

Case 2: $k < i < j$

In this case,

$$P\{i \; rmand \; j \text{ are ever compared}\} = \frac{2}{j-k+1}$$

It follows from the proceeding that

$$\frac{1}{2}E[X] = \Sigma_{j=2}^{k}\Sigma_{i=1}^{j-1}\frac{1}{k-i+1} + \Sigma_{j=k+1}^{n}\Sigma_{i=1}^{k}\frac{1}{j-i+1} + \Sigma_{j=k+2}^{n}\Sigma_{i=k+1}^{j-1}\frac{1}{j-k+1}$$

To approximate the proceeding when n and k are large, let $k = \alpha n$, for $0 < \alpha < 1$. Now,

$$\begin{aligned}
\Sigma_{j=2}^{k}\Sigma_{i=1}^{j-1}\frac{1}{k-i+1} &= \Sigma_{i=1}^{k-1}\Sigma_{j=i+1}^{k}\frac{1}{k-i+1} \\
&= \Sigma_{i=1}^{k-1}\frac{k-i}{k-i+1} \\
&= \Sigma_{j=2}^{k}\frac{j-1}{j} \\
&\sim k - \log(k) \\
&\sim k = \alpha n
\end{aligned}$$

Similarly,

$$\begin{aligned}
\Sigma_{j=k+1}^{n}\Sigma_{i=1}^{k}\frac{1}{j-i+1} &= \Sigma_{j=k+1}^{n}(\frac{1}{j-k+1} + \cdots \frac{1}{j}) \\
&\sim \Sigma_{j=k+1}^{n}(\log(j) - \log(j-k)) \\
&\sim \int_{k}^{n}\log(x)dx - \int_{1}^{n-k}\log(x)dx \\
&\sim n\log(n) - n - (\alpha n\log(\alpha n) - \alpha n) \\
&\quad -(n-\alpha n)\log(n-\alpha n) + (n-\alpha n) \\
&\sim n[\alpha\log(\alpha) - (1-\alpha)\log(1-\alpha)]
\end{aligned}$$

As it similarly follows that

$$\Sigma_{j=k+2}^{n}\Sigma_{i=k+1}^{j-1}\frac{1}{j-k+1} \quad \sim \quad n-k=n(1\alpha)$$

we see that

$$E[X] \sim 2n[1-\alpha\log(\alpha)-(1-\alpha)\log(1-\alpha)]$$

Thus, the mean number of comparisons needed by the Select algorithm is a linear function of the number of values.

3.6 Strassen's Matrix Multiplication Algorithm

- Product of two 2×2 matrices:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

  Note: 8 multiplications & 4 additions

- Strassen's Matrix Multiplication: Let

$$\begin{aligned} m_1 &= (a_{11}+a_{22})(b_{11}+b_{22}) \\ m_2 &= (a_{21}+a_{22})b_{11} \\ m_3 &= a_{11}(b_{12}-b_{22}) \\ m_4 &= a_{22}(b_{21}-b_{11}) \\ m_5 &= (a_{11}+a_{12})b_{22} \\ m_6 &= (a_{21}-a_{11})(b_{11}+b_{12}) \\ m_7 &= (a_{12}-a_{22})(b_{21}+b_{22}). \end{aligned}$$

  Then, the product C is given by

$$C = \begin{bmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{bmatrix}$$

  Note: 7 multiplications & 18 additions/subtractions (Only one saving of multiplication at the expense of doing 14 additional multiplications/subtractions!)

- Product of two $n \times n$ matrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  where each of A, B, and C are partitioned into $n/2 \times n/2$ submatrix. (Assume that n is even.) Using Strassen's method, first we compute

$$M_1 = (A_{11}+A_{22})(B_{11}+B_{22}),$$

where operations are now matrix addition/subtraction and matrix multiplication. Similarly, we compute M_2 through M_7. Next, we compute

$$C_{11} = M_1 + M_4 - M_5 + M_7,$$

and C_{12}, C_{21}, and C_{22}.

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \end{aligned} \tag{1}$$

Now, the matrices are sufficiently small, (i.e., 2×2) we multiply in the standard way. Therefore,

$$\begin{aligned} M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\ &= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} \\ &= \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} \end{aligned} \tag{2}$$

After this, M_2 through M_7 are similarly computed, and then values of C_{11}, C_{12}, C_{21}, and C_{22} are computed. They are combined to yield C.

3.6.1 Time Complexity Analysis of Number of Multiplications (Strassens's Algorithm)

Let n denote the number of rows (and columns). Assume that n is a power of two. We then have the following recurrence relation:

$$\begin{aligned} T(n) &= 7T(\frac{n}{2}) \text{ for } n > 1 \\ T(1) &= 1. \end{aligned}$$

Solving this recurrence relation, we have

$$T(n) = n^{\log 7} \approx n^{2.81} = O(n^{2.81}).$$

3.6.2 Time Complexity Analysis of Number of Additions/subtractions (Strassens's Algorithm)

Let n denote the number of rows (and columns). Assume that n is a power of two. We then have the following recurrence relation:

$$\begin{aligned} T(n) &= 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 \text{ for } n > 1 \\ T(1) &= 0. \end{aligned}$$

Solving this recurrence relation, we have

$$T(n) = 6n^{\log 7} - 6n^2 \approx 6n^{2.81} - 6n^2 = O(n^{2.81}).$$

Divide-and-Conquer algorithms. (These are not homework problems.)

1. Suppose that you are given a *black box* - you cannot see how it is designed - that has the following properties. If you input any sequence of real numbers, and an integer k, the box will answer "yes" or "no", indicating whether there is a subset of the numbers whose sum is exactly k. For example, if you input $\{2, 5, 4, 7\}$ and $k = 8$, it will answer "no"; and if you input $\{2, 5, 4, 7, 1\}$ and $k = 8$, it will answer "yes". Show how to use this black box to find a subset of $A = \{a_1, a_2, \cdots, a_n\}$ whose sum is k, if it exists. You should use the box $O(n)$ times.

2. We are given a set S containing n real numbers and a real number x.

(a) Design an $O(n \log n)$ time algorithm to determine whether there exist two elements in S whose sum is exactly x.

(b) Suppose the set S is given in a sorted order. Design an $O(n)$ time algorithm to solve this problem.

3. Describe an (n) time algorithm that, given a set S of n distinct numbers and a positive integer k ($k \leq n$), find k numbers in S that are closest to the median of S.

4. Given a list A of n numbers (some are identical), determine in $O(n)$ time whether there exists a number in A that appears more than $n/2$ times.

4 Greedy Method

The greedy method is perhaps the most straightforward design technique to solve a variety of problems. Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints. We need to find a *feasible* solution that either maximizes or minimizes a given *objective function*. A feasible solution that does this is called an *optimal solution*. Greedy algorithms are usually simple, but verifying the correctness (i.e., proving the optimality) is often non-trivial.

4.1 Optimal Storage on Tapes

Given n programs that are to be stored on a computer tape of length l. A length l_i, $1 \leq i \leq n$, is associated with each program i such that $l \geq \sum_{i=1}^{n} l_i$. The objective is to find a permutation of n programs so that when programs are stored in this order, the mean retrieval time (MRT) is minimized assuming that all programs are retrieved equally often.

Suppose the programs are stored in the order $I = (i_1, i_2, \cdots, i_n)$. Then the time needed to retrieve program i_j is $\sum_{k=1}^{j} l_{i_k}$.

4.1.1 Algorithm

Let $I = (1, 2, ..., n)$ be an order of programs such that $l_1 \leq l_2 \leq \cdots, \leq l_n$. We then show that this is an optimal solution. Note that the time complexity of this algorithm is $\theta(n \lg n)$.

4.1.2 Optimality Proof

Theorem 3 *If $l_1 \leq l_2 \leq ... \leq l_n$, then the ordering $I_0 = (1, 2, 3, \cdots, n)$ minimizes the MRT.*

Proof: Let $I = i_1, i_2, ..., i_n$ be any permutation of (1,2,....,n). Then, the time to retrieve each program, stored in this order, once from the tape is

$$\begin{aligned} {}^{M}D(I) &= (l_{i_1}) + (l_{i_1} + l_{i_2}) + (l_{i_1} + l_{i_2} + l_{i_3}) + \cdots + (l_{i_1} + l_{i_2} + \cdots + l_{i_n}) \\ &= nl_{i_1} + (n-1)l_{i_2} + (n-2)l_{i_3} + \cdots + 2l_{i_{n-1}} + l_{i_n} \\ &= \sum_{i=1}^{n} (n-k+1)l_{i_k} \end{aligned}$$

Suppose there exist two indices a and b such that $a < b$ and $l_{i_a} > l_{i_b}$. Then interchanging i_a and i_b results in a permutation I' with
$D(I') = \sum_{\substack{k=1 \\ k \neq a \\ k \neq b}}^{n} (n-k+1)l_{i_k} + (n-a+1)l_{i_b} + (n-b+1)l_{i_a}$ Subtracting $D(I')$ from $D(I)$, we have

$$\begin{aligned} D(I) - D(I') &= (n-a+1)l_{i_a} + (n-b+1)l_{i_b} - (n-a+1)l_{i_b} - (n-b+1)l_{i_a} \\ &= (n-a+1)(l_{i_a} - l_{i_b}) + (n-b+1)(l_{i_b} - l_{i_a}) \\ &= (b-a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, it is easy to see that no permutation, which is not in the non-decreasing order ($\leq$ .... $\leq$), can have the minimum.

4.1.3 A Generalization

Suppose there are m tapes and n programs, and the objective is to store each program on one of the tapes such that the MRT is minimized. We then have the following greedy algorithm.

Algorithm: Let $l_1 \leq l_2 \leq ..... \leq l_n$. Assign program i to the tape $T_{i \bmod m}$.

Theorem 4 *This algorithm generates an optimal solution.*

Proof: In any storage pattern I, let r_i be the number of programs following program i on its tape. Then the total retrieval time is $TD(I) = \sum_{i=1}^{m}(r_i)l_i$. It is easy to see from the previous theorem that for $m = 1$, the $TD(I)$ is minimized, if m longest programs have $r_i = 0$, then the next m longest programs have $r_i = 1$, etc.

4.1.4 A variation of the MRT problem

Given n programs and m tapes, find an assignment of n programs to m tapes such that $\max_{i \leq j \leq m} L(j)$ is minimized, where $L(j) \cong \sum\{l_i \mid \text{program } i \text{ is in tape } j\}$.

4.2 (Fractional) Knapsack Problem

Given: a list of n items with profit function $P = (p_1, p_2, \cdots, p_n)$] and weight function $W = (w_1, w_2, \cdots, w_n)$, and a knapsack with capacity M.

Objective: to find a solution $X = (x_1, x_2, \cdots, x_n)$ such that (i) x_i is a real number in $0 \leq x_i \leq 1$ for each i, (ii) $\sum_{i=1}^{n} x_i w_i \leq M$, and (iii) satisfying (i-ii), $\sum_{i=1}^{n} x_i p_i$ is maximized.

4.2.1 Three greedy algorithms

1. Start with the largest profit.
2. Start with the smallest weight.
3. Let $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq .... \geq \frac{p_n}{w_n}$, and consider items in this order.

4.2.2 Optimality

Theorem: Algorithm 3 generates an optimal solution.

Proof: Let $X = (x_1, x_2, ..., x_n)$ be a solution generated by Algorithm 3. If $x_i = 1$, for all i, then X is optimal. So, let j be the least index such that $x_j < 1$. Note that

(i) $x_i = 1$ for $i \leq i \leq j - 1$,

(ii) $x_i = 0$ for $j < i \leq n$.

Let $Y = (y_1, y_2, ..., y_n)$ be an optimal solution such that $x \neq Y$. Without loss of generality, we can assume that $\sum_{i=1}^{n} w_i y_i = M$. Let k be the least index such that $x_k \neq y_k$. We next prove that $y_k < x_k$. Consider three cases.

1. $k < j$: $x_k = 1$, thus $y_k < x_k$.
2. $k = j$: Note that by the definition of k, $y_i = x_i$ for all i, $1 \leq i \leq j = k$, and by the definition of j, $y_j < x_j$ since otherwise (i.e., $y_j > x_j$), $W(Y) > M$. Hence, $y_k < x_k$.
3. $k > j$: Then, $W(Y) > M$, which is impossible.

Now define a new solution $Z = (z_1, z_2, \cdots, z_n)$ such that

(a) $z_i = y_i$ for $1 \leq i < k$;

(b) $z_k = x_k$, and

(c) $z_i \leq y_i$ for $k < i \leq n$ such that $w_k(x_k - y_k) = \sum_{i=k+1}^{n} w_i(y_i - z_i)$.

We then have

$$\begin{aligned}
\sum_{i=1}^{n} p_i z_i &= \sum_{i=1}^{n} p_i y_i + (z_k - y_k)p_k - \sum_{i=k+1}^{n} (y_i - z_i)p_i \\
&= \sum_{i=1}^{n} p_i y_i + (z_k - y_k)w_k \frac{p_k}{w_k} - \sum_{i=k+1}^{n} (y_i - z_i)w_i \frac{p_i}{w_i} \\
&\geq \sum_{i=1}^{n} p_i y_i + (z_k - y_k)w_k \frac{p_k}{w_k} - \sum_{i=k+1}^{n} (y_i - z_i)w_i \frac{p_k}{w_k}, \\
&\quad \text{since} \frac{p_i}{w_i} \leq \frac{p_k}{w_k} \text{ for } i \geq k+1. \\
&= \sum_{i=1}^{n} p_i y_i + \frac{p_k}{w_k}(w_k(z_k - y_k) - \sum_{i=k+1}^{n} (y_i - z_i)w_i) \\
&= \sum_{i=1}^{n} p_i y_i
\end{aligned}$$

Since Y is assumed to be optimal, $\sum_{i=1}^{n} p_i z_i = \sum_{i=1}^{n} p_i y_i$. By repeating this process, we can transform Y into X. Hence, X must be optimal. ■

4.3 Optimal Merge Patterns (Two way)

Recall: $\theta(n+m)$ time to merge two sorted files containing n and m records.
Problem: Given more than two sorted files, merge them into one sorted file such that total # of necessary comparisons is minimized.

4.4 Huffman Code

This is another application of Optimal Binary Tree with minimum weighted external path length.

Def: A set of binary numbers are called a prefix code, if any symbol in the set is not a prefix of any other symbol in the set. For example, $\{00, 01, 100, 1010, 1011, 11\}$ is a prefix code, but $\{10, 01, 0110\}$ is not.

Example: Suppose there are 6 letters a_1, a_2, a_3, a_4, a_5, a_6, such that a_1 appears 25 % of the time, and similarly for, a_2 20 %, a_3 20 %, a_4 10 %, a_5 10 % and a_6 15 % . We want to design a prefix code such that a symbol in the code is assigned to each letter and the average length of a code message is minimized.

4.5 Prim's MST Algorithm

4.5.1 Algorithm

begin
    Initially, let $V_T \leftarrow \emptyset$;
    Select an arbitrary vertex, say s, to start a tree;
    while $V_T \neq V$ do
        Select an edge $e = (x, y)$ such that $x \in V_T$, $y \in V - V_T$, and $w(e)$ is
            minimum among all such edges;
        add e and y to the tree;
    endwhile
end.

4.5.2 Time Complexity

$O(n^2)$ time in the worst case.

4.5.3 Correctness

Theorem 5 *Let T_i be the tree constructed after the i_{th} iteration of the while loop. We then claim that there exits a minimum spanning tree that has T_i as a subgraph.*

Proof: Proof by induction on i. Consider an input graph G. A a basis, $T_0 = (\{s\}, \emptyset)$ is clearly a subgraph of any MST. Assume that T_{i-1} is a subgraph of some MST T^*. Now, consider the i_{th} iteration.

Case 1. $(x, y) \in T^*$:

Then, $T_i \in T^*$.

Case 2. $(x, y) \notin T^*$:

In this case, there must be a path in T^* from x to y that does not include the edge (x, y). Let (v, w) be the first edge of that path such that $v \in T_{i-1}$ and $w \notin T_{i-1}$. (Note that edge (v, w) has been considered in the i_{th} iteration.) Define $T' = T^* + (x, y) - (v, w)$. Note that T' is also a spanning tree of G. Further, $w(T') \leq w(T^*)$, since $w(x, y) \leq w(v, w)$. Since T^* is a minimum spanning tree, the equality must hold, i.e., $w(T') = w(T^*)$. Hence, T_i is a subgraph of a MST T'. This completes the proof of the theorem. ■

4.6 Kruskal's MST Algorithm

4.6.1 Algorithm

begin
    Initially, let $V_T = \{v_1, v_2, \cdots, v_n\}$ and $E_T = \emptyset$;
    Sort edges of G such that $e_1 \leq e_2 \leq \cdots \leq e_m$ where $m = |E(G)|$;
    for $i = 1$ to m
        if $E_T \cup \{e_i\}$ does not create a cycle
            then let $E_T \leftarrow E_T \cup \{e_i\}$
            else let $E_T \leftarrow E_T$
    endfor;
end.

4.6.2 Time Complexity

$O(|E| \log |V|)$ time.

4.6.3 Correctness

Let $|V| = n$. Let T^* be a tree generated by the algorithm. Relabel the edges as $e_1, e_2, \cdots, e_{n-1}$ such that these are the edges of T^* added in this order. For any tree T other than T^*, define $f(T)$ to be the smallest index i $(1 \leq i \leq n-1)$ such that $e_i \notin E(T)$. This means that $\{e_1, \cdots, e_{i-1}\} \subseteq E(T)$. Suppose there exists a tree T' such that $w(T') < w(T^*)$. Among such trees (i.e., whose weights are less than $w(T^*)$), we choose T' as a tree which has the largest value of $f(T')$. Let $f(T') = k$, i.e., $e_k \notin T'$. Consider the graph $T' + \{e_k\}$, which contains a unique cycle C such that $e_k \in C$. Let $e^* \in C$ such that $e^* \in T'$ but $e^* \notin T^*$. (There must be such an edge since otherwise $C \subseteq T^*$, impossible.) Therefore, $e^* \notin \{e_1, \cdots, e_{n-1}\}$ $(= E(T^*))$. Now, let $T_0 = T' + \{e_k\} - \{e^*\}$. Note that T_0 is also a spanning tree of G.

Suppose $w(e^*) < w(e_k)$. Note that since $\{e_1, \cdots, e_{k-1}, e^*\} \subseteq E(T')$, the subgraph induced on $\{e_1, \cdots, e_{k-1}, e^*\} \subseteq E(T')$ must be acyclic. This implies that e^* would have been included in T^* since e^* must have been considered before e_k. Therefore, $w(e^*) \geq w(e_k)$, implying that $w(T_0) \leq w(T')$ and $f(T_0) > f(T') = k$. this is a contradiction to the choice of T'. Therefore, there exists no tree T' such that $w(T') < W(T^*)$ which concludes that T^* is a MST. ■

4.6.4 Dijkstra's Shortest Path Algorithm

4.6.5 Algorithm

begin
Let $V' \leftarrow \emptyset$, $l(s) \leftarrow 0$, and for all $v \neq s$, let $l(v) \leftarrow \infty$.
for $i = 1$ to n *do*
Let u be a vertex in $V - V'$ for which $l(u)$ is minimum;
Let $V' \leftarrow V' \cup \{u\}$;
For every edge $e = (u, v)$ such that $v \in V - V'$ and $l(v) > l(u) + w(e)$, let $l(v) \leftarrow l(u) + w(e)$.
endfor
end.

4.6.6 Time Complexity

$O(n^2)$ time.

4.6.7 Correctness

Theorem 6 *Let G be an edge-weighted directed graph. Let $V'(i)$ denote the vertex set V' constructed after the i_{th} iteration of the for loop. Let $u_i \in V - V'(i-1)$ be the vertex selected at the i_{th} iteration. Then, $l(u_i)$ is the length of the shortest path (i.e., the distance) in G from s to u_i.*

Proof: Proof by induction on i. Assume that the claim is true for up to $(i-1)$ iterations, and now consider the i_{th} iteration. When u_i is selected, there exists a path of length $l(u_i)$ from s to u_i such that the intermediate vertices in that path are all in $V'(i-1)$. Let $P = (s, w_1, w_2, \cdots, w_k, u_i)$ be such a path, i.e., $s, w_1, w_2, \cdots, w_k \in V'(i-1)$. Suppose P is not a shortest path in G from s to u_i, and let $P' = (s, z_1, z_2, \cdots, z_l, u_i)$ be a shortest path in G from s to u_i. Let z_t $(1 \leq t \leq l)$ be the first vertex in the path P' which is in $V - V'(i-1)$. Since vertex u_i, not z_t, was selected in the i_{th} iteration, it implies that $l(z_t) \geq l(u_i)$. Let $P'_1 = (s, z_1, \cdots, z_t)$ and $P'_2 = (z_t, \cdots, u_i)$ denote two subpaths of P', where $l(P') = l(P'_1) + l(P'_2)$. Note that $l(z_t)$ must be equal to $l(P'_1)$ during the i_{th} iteration. Since $l(z_t) \geq l(u_i)$ and $l(P'_2) > 0$, it implies that $l(P') > l(P) = l(u_i)$. Therefore, P' cannot be a shortest path in G from s to u_i. This completes the proof of the theorem. ■

Greedy Algorithms (This is not homework problems.)

1. You are given n activity schedules $[s_i, f_i]$ for $1 \leq i \leq n$ for one day, where s_i and f_i denote the start and the finishing time of activity i. You are to select the maximum number of activities that can be scheduled in one room such that no two activities can be selected unless they do not have an overlapping period. Design an $O(n)$ time algorithm.

2. Given n segments of line (on the X axis) with coordinates $[l_i, r_i]$. You are to choose the minimum number of segments that cover the segment $[0, M]$. Design a greedy algorithm to solve this problem.

3. Consider the following problem. The input consists of n skiers with heights $p_1, \cdots, p_n$, and n n skies with heights $s_1, \cdots, s_n$. The problem is to assign each skier a ski to minimize the average difference between the height of a skier and his/her assigned ski. That is, if the ith skier is given the $\alpha(i)$th ski, then you want to minimize:

$$\frac{1}{n} \sum_{i=1}^{n} |p_i - s_{\alpha(i)}|.$$

(a) Consider the following greedy algorithm. Find the skier and ski whose height difference is minimum. Assign this skier this ski. Repeat the process until every skier has a ski. Prove of disprove that this algorithm is correct.

(b) Consider the following greedy algorithm. Give the shortest skier the shortest ski, give the second shortest skier the second shortest ski, give the third shortest skier the third shortest ski, etc. Prove of disprove that this algorithm is correct.

HINT: One of the above greedy algorithms is correct and the other is not.

4. The input to this problem consists of an ordered list of n words. The length of the ith word is w_i, that is the ith word takes up w_i spaces. (For simplicity assume that there are no spaces between words.) The goal is to break this ordered list of words into lines, this is called a layout. Note that you can not reorder the words. The length of a line is the sum of the lengths of the words on that line. The ideal line length is L. Assume that $w_i \leq L$ for all i. No line may be longer than L, although it may be shorter. The penalty for having a line of length K is $L - K$. Consider the following greedy algorithm.

```
For i = 1 to n
    Place the ith word on the current line if it fits
        else place the ith word on a new line
```

(a) The overall penalty is defined to be the sum of the line penalties. The problem is to find a layout that minimizes the overall penalty. Prove of disprove that the above greedy algorithm correctly solves this problem.

(b) The overall penalty is now defined to be the maximum of the line penalties. The problem is to find a layout that minimizes the overall penalty. Prove of disprove that the above greedy algorithm correctly solves this problem.

5. A source node of a data communication network has n communication lines connected to its destination node. Each line i has a transmission rate r_i representing the number of bits that can be transmitted per second. A data needs to be transmitted with transmission rate at least M bits per second from the source node to its destination node. If a fraction x_i ($0 \leq x_i \leq 1$) of line i is used (for example, a fraction x_i of the full bandwidths of line i is used), the transmission rate through line i becomes $x_i \cdot r_i$ and a cost $c_i \cdot x_i$ is incurred. Assume that the cost function c_i ($1 \leq i \leq n$) is given. The objective of the problem is to compute x_i, for $1 \leq i \leq n$, such that $\sum_{1 \leq i \leq n} r_i x_i \geq M$ and $\sum_{1 \leq i \leq n} c_i x_i$ is minimized.

(a) Describe an outline of a greedy algorithm to solve the problem.

(b) Prove that your algorithm in part (a) always produces an optimum solution. You should give all the details of your proof.

6. Suppose we have a sequence of n objects $a_1, a_2, \cdots, a_n$ where each a_i has size s_i. We wish to store these objects in buckets such that the objects in each bucket are consecutive members of the sequence. Assume that we have buckets of k different sizes $l_1, \cdots, l_k$ such that n buckets are available for each size. The cost of any bucket is proportional to its size. The objects $a_{j+1}, a_{j+2}, \cdots, a_{j+s}$ fit into a bucket of size l_r if and only if

$$\sum_{i=j+1}^{j+s} s_i \leq l_r.$$

The problem is to store the objects in buckets with the minimum total cost. Describe a polynomial time algorithm for solving this problem. (**Hint:** You may formulate the problem as the shortest problem.)

7. Modify the Dijkstra's algorithm so that it checks if a directed graph has a cycle. Analyze your algorithm, and show the results using order notation.

8. Can Dijkstra's algorithm be used to find shortest paths in a graph with some negative weights ? Justify your answer.

9. Suppose we assign n persons to n jobs, Let c_{ij} be the cost of assigning the ith person to the jth job. Use a greedy approach to write an algorithm that finds an assignment that minimizes the total cost of assigning all n persons to all n jobs. Is your algorithm optimal?

10. Given an edge-weighted directed acyclic graph (DAG) G, give an $O(|E|)$ time algorithm to find a longest path in G. (Note that finding a longest path in an arbitrary (directed or undirected) graph is NP-complete.)

5 Dynamic Programming

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of stepwise decisions. In dynamic programming, an optimal sequence of decisions is obtained by making explicit appeal to the *principle of optimality*. The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

EX: shortest path problem vs longest path problem.

Note that only problems that satisfy the principle of optimality may be solved using dynamic programming. This implies that if a problem does not satisfy the principle of optimality such as the longest path problem, it cannot be solved using a dynamic programming algorithm. On the other hand, even if a problem satisfies the principle of optimality, it may or may not be solved using a dynamic programming algorithm.

5.1 All Pairs Shortest Paths

Given an edge-weighted directed graph G with the vertex set $V = \{1, 2, \cdots, n\}$, the problem is to find a shortest path between every pair of nodes in G. This can be solved by applying the Dijkstra's algorithm n times resulting in its time complexity $O(n^3)$.

5.1.1 Dynamic Programming Algorithm

Let $A^k(i, j)$ denote the length of a shortest path from i to j going through no vertex of index greater than k. We then have

$$A^k(i, j) = \min\{A^{k-1}(i, j),\ A^{k-1}(i, k) + A^{k-1}(k, j)\}$$

for $k \geq 1$, and $A^0(i, j) = w(i, j)$ for $1 \leq i, j \leq n$.

5.1.2 Time Complexity

$O(n^3)$

5.1.3 Other Related Problems

1. Find a shortest cycles in G.
2. Find a *bottleneck* path for each pair of nodes in G.
3. Find a bottleneck spanning tree.
4. Find a bottleneck cycle.

5.2 Optimal Binary Search Tree

A *binary search tree* T for a set S is a labeled binary tree in which each vertex v is labeled by an element $l(v) \in S$ such that

1. for each vertex u in the left subtree of v, $l(u) < l(v)$;
2. for each vertex u in the right subtree of v, $l(u) \geq l(v)$;
3. for each element $a \in S$, there exists exactly one vertex v such that $l(v) = a$.

Let $S = \{a_1, \cdots, a_n\}$ such that $a_1 < a_2, < \cdots < a_n$. Let $P(i)$ denote the probability that element a_i is searched for. Let $Q(i)$ denote the probability that an element X, for $a_i < X < a_{i+1}$, is searched for. Assume that $a_0 = -\infty$ and $a_{n+1} = +\infty$. Note then

$$\sum_{1 \leq i \leq n} P(i) + \sum_{0 \leq i \leq n} Q(i) = 1.$$

Consider a binary search tree T with n internal nodes $a_1, \cdots, a_n$ and $n+1$ external nodes $E_0, \cdots, E_n$ such that $E_i = \{X \mid a_i < X < a_{i+1}\}$. The *cost* of T is then defined to be the average number of comparisons for a successful or an unsuccessful search of a node in T. We then have

$$C(T) = \sum_{1 \leq i \leq n} P(i) \cdot level(a_i) + \sum_{0 \leq i \leq n} Q(i) \cdot (level(E_i) - 1).$$

The *optimal binary search tree* is a binary search tree with the minimum cost.

Recall: In the binary decision tree discussed w.r.t. the binary search algorithm, $P(i)$'s and $Q(i)$'s are all equal.

5.2.1 Dynamic Programming Algorithm

Suppose a_k is chosen to be the root of T. Then, $L = \{E_0, a_1, E_1, \cdots, a_{k-1}, E_{k-1}\}$ and $R = \{E_k, a_{k+1}, E_{k+1}, \cdots, a_n, E_n\}$ where L and R, respectively, denote the left and right subtrees of a_k. Let $T(i,j)$ denote the optimal binary search tree containing $E_i, a_{i+1}, E_{i+1}, \cdots, a_j, E_j$. Let $C(i,j)$ and $R(i,j)$ denote the cost and the root of $T(i,j)$. Define $W(i,j) = Q(i) + (P(i+1) + Q(i+1)) + \cdots (P(j) + Q(j))$. The cost of tree T when a_k is the root is then

$$C(T) = P(k) + C'(L) + C'(R),$$

where

$$C'(L) = \sum_{1 \leq i \leq k-1} P(i) \cdot (level_L(a_i) + 1) + \sum_{0 \leq i \leq k-1} Q(i) \cdot (level_L(E_i) - 1 + 1)$$

and

$$C'(R) = \sum_{k+1 \leq i \leq n} P(i) \cdot (level_R(a_i) + 1) + \sum_{k+1 \leq i \leq n} Q(i) \cdot (level_R(E_i) - 1 + 1).$$

Thus,

$$C(T) = P(k) + C(L) + \{Q(0) + P(1) + Q(1) + \cdots + P(k-1) + Q(k-1)\}$$

$$
\begin{aligned}
&+C(R) + \{Q(k) + P(k+1) + Q(k+1) + \cdots + P(n) + Q(n)\} \\
&= P(k) + C(L) + C(R) + W(0, k-1) + W(k, n) \\
&= C(L) + C(R) + 1.
\end{aligned}
$$

Therefore, k must be chosen to minimize the above equation. Hence, we have

$$C(0, n) = \min_{1 \le k \le n} \{C(0, k-1) + C(k, n) + 1\}.$$

In general,

$$C(i, j) = \min_{i+1 \le k \le j} \{C(i, k-1) + C(k, j) + W(i, j)\},$$

for $i < j$, and $C(i, i) = 0$ for each $0 \le i \le n$.

Example: Consider $a_1 < a_2 < a_3 < a_4$ with $Q(0) = 2/16$, $P(1) = 4/16$, $Q(1) = 3/16$, $P(2) = 2/16$, $Q(2) = 1/16$, $P(3) = 1/16$, $Q(3) = 1/16$, $P(4) = 1/16$, $Q(4) = 1/16$.

5.2.2 Time Complexity

An naive analysis is $O(n^3)$, but it can be shown as $O(n^2)$. (See #2(a), pp. 282, TEXT.)

5.3 Traveling Salesperson Problem

Let G be a directed edge-weighted graph with edge cost $c_{ij} > 0$. A *tour* of G is a directed simple cycle that includes every vertex of G, and the cost of a tour is the sum of the cost of edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

5.3.1 Dynamic Programming Algorithm

Let $V = \{1, 2, \cdots, n\}$, and assume that a tour starts at node 1. For a subset $S \subset V$, let $g(i, S)$ denote the length of a shortest path that starts at node i, goes through all vertices in S, and terminates at vertex 1. The cost of an optimal tout is then $g(1, V - \{1\})$.

From the principle of optimality, it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

which is generalized as

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}.$$

Consider a graph G with $V = \{1, 2, 3, 4\}$, and the edge cost defined as:

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 12 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

5.3.2 Time Complexity

Note that for each value of $|S|$, there are $n-1$ choices for i in computing $g(i,S)$. The number of distinct subsets S of size k not including 1 or i is

$$\left(\begin{array}{c} n-2 \\ k \end{array} \right)$$

Let N be the number of $g(i,S)$'s that have to be computed before $g(i,S)$ is computed. We then have

$$N = \sum_{k=0}^{n-2} (n-1) \left(\begin{array}{c} n-2 \\ k \end{array} \right)$$

From the Binomial theorem,

$$(x+y)^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) x^k y^{n-k}.$$

By setting $x = y = 1$, this implies that $2^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right)$. Therefore,

$$\sum_{k=0}^{n-2} \left(\begin{array}{c} n-2 \\ k \end{array} \right) = 2^{n-2}.$$

Hence, $N = (n-1)2^{n-2}$.

In each computation of $g(i,S)$, we need $|S|$ comparisons. Therefore,

$$\begin{aligned} T &= \sum_{k=0}^{n-2} (n-1) \left(\begin{array}{c} n-2 \\ k \end{array} \right) k \\ &\leq \sum_{k=0}^{n-2} (n-1) \left(\begin{array}{c} n-2 \\ k \end{array} \right) (n-2) \\ &= (n-1)(n-2) \sum_{k=0}^{n-2} \left(\begin{array}{c} n-2 \\ k \end{array} \right) \\ &= (n-1)(n-2)2^{n-2} \\ &= O(n^2 2^n) \end{aligned}$$

In fact, $T = \theta(n^2 2^n)$.

5.4 Matrix Product Chains

Let $M_1 \times M_2 \times \cdots \times M_r$ be a chain of matrix products. This chain may be evaluated in several different ways. Two possibilities are $(\cdots((M_1 \times M_2) \times M_3) \times M_4) \times \cdots) \times M_r$ and $(M_1 \times (M_2 \times (\cdots \times (M_{r-1} \times M_r) \cdots)$. The cost of any computation pattern is the number of multiplications used.

5.4.1 Dynamic Programming Algorithm

Let $M(i,j)$ denote the matrix product $M_i \times M_{i+1} \times \cdots \times M_j$, and $C(i,j)$ denote the cost of computing $M(i,j)$ using an optimal product sequence for M_{ij}. Let $D(i)$, $0 \leq i \leq r$, represent the dimensions of the matrices such that M_i has $D(i-1)$ rows and $D(i)$ columns. We then have $C(i,i) = 0$, $1 \leq i \leq r$, and $C(i,i+1) = D(i-1)D(i)D(i+1)$, $1 \leq i \leq r$. For $j > i$,

$$C(i,j) = \min_{k=i}^{j-1}\{C(i,k) + C(k+1,j) + D(i-1)D(k)D(j+1)\}.$$

This algorithm leads to $O(r^3)$ time complexity for computing $C(1,r)$.

Dynamic Programming Algorithms (This is not homework problems.)

1. Give an efficient algorithm for the following problem. The input is an n sided convex polygon. Assume that the polygon is specified by the Cartesian coordinates of its vertices. The output should be the triangulation of the polygon into $n-2$ triangles that minimizes the sums of the cuts required to create the triangles.

   HINT: This is very similar to the matrix multiplication problem and the problem of finding the optimal binary search tree.

2. The input to this problem is a sequence of n points $p_1, \cdots, p_n$ in the Euclidean plane. You are to find the shortest routes for two taxis to service these requests in order. The two taxis start at the origin p_1. If a taxi visits a point p_i before p_j then it must be the case that $i < j$. Each point must be visited by one of the two taxis. The cost of a routing is the total distance traveled by the first taxi plus the total distance traveled by the second taxi. Design an efficient dynamic programming algorithm to find the minimum cost routing such that each point is covered by one of the two taxis.

3. Given an matrix of integers, you are to write a program that computes a path of minimal weight. A path starts anywhere in column 1 (the first column) and consists of a sequence of steps terminating in column n (the last column). A step consists of traveling from column i to column i+1 in an adjacent (horizontal or diagonal) row. The first and last rows (rows 1 and m) of a matrix are considered adjacent, i.e., the matrix "wraps" so that it represents a horizontal cylinder. For example, a move from cell $(1,1)$ to cell $(m,2)$ is legal. Legal steps are illustrated in Figure 1.

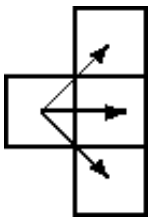

Figure 1:

   The weight of a path is the sum of the integers in each of the n cells of the matrix that are visited. For example, two slightly different 5×6 matrices are shown in Figure 2 (the only difference is the numbers in the bottom row). The minimal path is illustrated for each matrix. Note that the path for the matrix on the right takes advantage of the adjacency property of the first and last rows. Design an dynamic programming algorithm to solve this problem.

4. Suppose we have a sequence of n objects $a_1, a_2, \cdots, a_n$ where each a_i has size s_i. We wish to store these objects in buckets such that the objects in each bucket are consecutive members of the sequence. Assume that we have buckets of k different sizes $l_1, \cdots, l_k$ such that n buckets are available for each size. The cost of any bucket is proportional to its size. The objects $a_{j+1}, a_{j+2}, \cdots, a_{j+s}$ fit into a bucket of size l_r if and only if

$$\sum_{i=j+1}^{j+s} s_i \leq l_r.$$

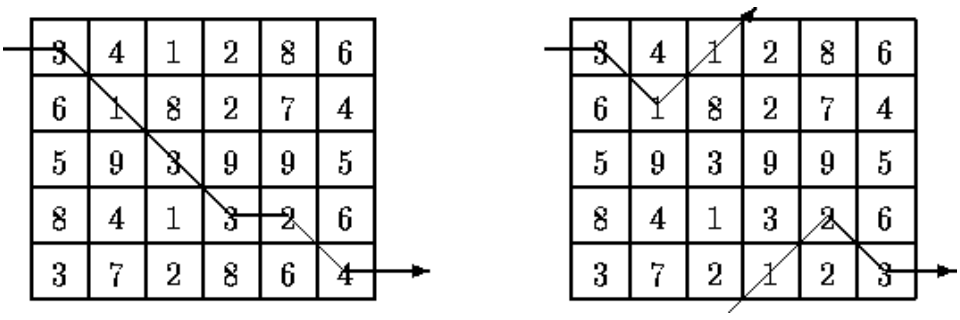

Figure 2:

The problem is to store the objects in buckets with the minimum total cost. Describe a dynamic programming algorithm for solving this problem, and analyze its time complexity.

5. Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates $a(1), \cdots, a(n)$ and n cities on the northern bank with x-coordinates $b(1), \cdots, b(n)$. You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank. Give a dynamic programming algorithm to solve this problem and analyze the time complexity of your algorithm. Note that those x-coordinate values are not sorted, i.e., $a(i)$'s and $b(i)$'s are in an arbitrary order.

6. Give a polynomial time algorithm for the following problem. The input consists of a sequence $R = R_1, \cdots, R_n$ of non-negative integers, and an integer k. The number R_i represents the number of users requesting some particular piece of information at time i (say from a www server). If the server broadcasts this information at some time t, the requests from all the users who requested the information strictly before time t have already been satisfied, and requests arrived at time t will receive service at the next broadcast time. The server can broadcast this information at most k times. The goal is to pick the k times to broadcast in order to minimize the total time (over all requests) that requests/users have to wait in order to have their requests satisfied. As an example, assume that the input was $R = 3, 4, 0, 5, 2, 7$ (so $n = 6$) and $k = 3$. Then one possible solution (there is no claim that this is the optimal solution) would be to broadcast at times $2, 4$, and 7 (note that it is obvious that in every optimal schedule that there is a broadcast at time $n + 1$ if $R_n \neq 0$). The 3 requests at time 1 would then have to wait 1 time unit. The 4 requests at time 2 would then have to wait 2 time units. The 5 requests at time 4 would then have to wait 3 time units. The 2 requests at time 5 would then have to wait 2 time units. The 7 requests at time 6 would then have to wait 1 time units. Thus the total waiting time for this solution would be

$$3 * 1 + 4 * 2 + 5 * 3 + 2 * 2 + 7 * 1.$$

7. Let G be a directed weighted graph, where $V(G) = \{v_1, v_2, \cdots, v_n\}$. Let B be an $n \times n$ matrix such that entry b_{ij} denotes the distance in G from v_i to v_j (using a directed path). Now we are going to insert a new vertex v_{n+1} into G. Let w_i denote the weight of the edge (v_i, v_{n+1}) and w'_i denote the weight of the edge (v_{n+1}, v_i). (If there is no edge from v_i to v_{n+1} or from v_{n+1} to v_i, then w_i or w'_i is inf, respectively.) Describe an algorithm to construct an $(n + 1) \times (n + 1)$ distance matrix B' from B and values of w_i and w'_i for $1 \leq i \leq n$. (Note

that the graph G itself is not given.) Your algorithm should work in $O(n^2)$ time. (Hint: Use the dynamic programming algorithm for finding all pairs shortest paths.)

8. You are given n types of coin denominations of values $v(1) < v(2) < \cdots < v(n)$ (all integers). Assume $v(1) = 1$, so you can always make change for any amount of money C. Give a dynamic programming algorithm which makes change for a given amount of money C with as few coins as possible. Analyze the time complexity of your algorithm.

9. The input to this problem is a pair of strings $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$. The goal is to convert A into B as cheaply as possible. The rules and cost are defined as follow. For a cost of 3 you can delete any letter. For a cost of 4 you can insert a letter in any position. For a cost of 5 you can replace any letter by any other letter. For example, you can convert $A = abcabc$ to $B = abacab$ via the following sequence: *abcabc* at a cost of 5 can be converted to *abaabc*, which at cost of 3 can be converted to *ababc*, which at cost of 3 can be converted to *abac*, which at cost of 4 can be converted to *abacb*, which at cost of 4 can be converted to *abacab*. Thus the total cost for this conversion would be 19. This is probably not the cheapest possible conversion. Give a dynamic programming algorithm to solve this problem. (Note that the Unix *diff* command essentially solves this problem.)

10. Consider the problem of word-wrapping a paragraph. A paragraph is an ordered list of n words, where word w_i is l_i letters long. You want to divide the paragraph into a sequence of lines (no word can be broken into two line), each containing at most L letters. (No word is more than L letters long.) Suppose a line contains words $w_i, \cdots, w_j$. The total length $W(i, j)$ of this line is defined by $W(i, j) = j - i + \sum_{k+i}^{j} l_k$. This length accounts for a single space between successive pairs of words on the line. The slop $S(i, j)$ of this line is defined to be $L - W(i, j)$, the total number of unused spaces at the end of the line. Note that in any feasible solution, the slop of each line must be non-negative. (The cubed slop criterion is a simplified version of what is actually used in, e.g., TeX for paragraph wrapping.) Just to make things concrete, consider the example paragraph Now is the time for all good men., and suppose $L = 10$. One feasible solution is

Now is the
time for
all good
men

This solution has four lines of lengths $10, 8, 8$, and 4; the corresponding slops are $0, 2, 2$, and 6. Your goal is to find a division of the input paragraph into lines that minimizes the sum, over all lines except the last, of the cubed slop of each line. (We omit the last line because it can in general be much shorter than the others.) For example, the total cost of the above solution is $0^2 + 2^3 + 2^3 = 16$. Give an efficient algorithm for this problem.

6 Search and Traversal Techniques

6.1 Binary Tree Traversal

- Inorder
- Preorder
- Postorder

6.1.1 Analysis of time complexity

Let $t(n)$ denote the time needed by the inorder traveral algorithm. (Note that the analysis of preorder or postorder can be similarly done.) Asusme that visiting each node takes a constant amount of time, say c_1. If the no. of nodes in the left tree of T is n_1, we then have

$$t(n) \leq \max_{0 \leq n_1 \leq n-1} \{t(n_1) + t(n - n_1 - 1) + c_1\}.$$

Using induction on n, we next prove that $t(n) \leq c_2 n + c_1$ for some constant c_2 ($c_2 \geq 2c_1$).

Assume that $t(n) \leq c_2 n + 1$ for all $n < m$, and consider $n = m$. Then,

$$\begin{aligned} t(m) &\leq \max_{0 \leq n_1 < m} \{t(n_1) + t(m - n_1 - 1) + c_1\} \\ &\leq \max_{0 \leq n_1 < m} \{c_2 n_1 + c_1 + c_2(m - n_1 - 1) + c_1 + c_1\} \\ &= \max_{0 \leq n_1 < m} \{c_2 m + 3c_1 - c_2\} \\ &\leq c_2 m + c_1 + 2c_1 - c_2 \leq c_2 m + c_1 \end{aligned}$$

Since $t(n) = \Omega(n)$, $t(n) = \theta(n)$.

6.2 Breadth First Search

$O(|E|)$ time.

6.3 Depth First Search

$O(|E|)$ time.

6.4 Biconnected Components and DFS

Definition: articulation point, cut-vertex, cut-edge, biconnected, maximal biconnected subgraph, biconnected component, vertex cut, edge cut, vertex connectivity, edge connectivity

6.4.1 Computing biconnected cmponents

$O(|E|)$ time to report all biconnected components.

Define: $L(v) = \min\{dfn(v), \min\{dfn(w) \mid W$ can be reached from v by following k (for any $k \geq 0$) forward edges and following exactly one backward edge. $\}\}$

1. Let $u \in V$ be such that $u \neq s$. Then, u is an articulation point if and only if for any $(u, v) \in E, \cdots$
2. Suppose $u = s$. Then, u is an articulation point if and only if $\cdots$

Example: computing biconnected components

Consider the following graph G, where the bold-faced number shows the depth-first-number of

the node and other numbers show the low points as revised.

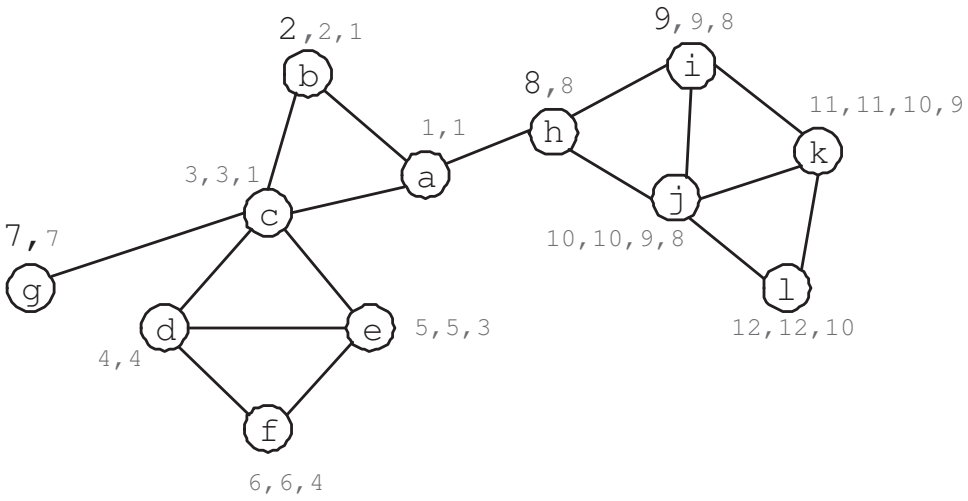

Figure 3: G

We note the following from the above example.

(i) Edge (e, c) was checked before edge (e, f).

(ii) Edge (k, i) was checked after edge (k, l).

(iii) Low points (updated as shown) of each node may be obtained when edges are examined by the following sequence: $(a, b), (b, c), (c, d),\ (d, e), (e, c), (e, f),\ (f, d), (c, a), (c, g),\ (a, h), (h, i), (i, j),$ $(j, k), (k, l), (l, j),\ (k, i), (j, h)$.

7 Backtracking

Many problems dealing with searching for a set of solutions satisfying some constraints can be solved using the *backtracking* formulation. In many applications of the backtracking method, the desired solution is expressed as an n-tuple $(x_1, \cdots, x_n)$ where x_i is chosen from a given finite set S_i satisfying a certain criterion function $P(x_1, \cdots, x_n)$.

For example, consider a sorting problem where we are given an input $a[1:n]$. Then, a solution is expressed as $X = (x_1, \cdots, x_n)$ where $x_i \in \{1, \cdots, n\}$ and $P : a[x_i] \leq a[x_{i+1}]$. The number of possible candidates for X is $n!$.

In general, there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for being X, where $m_i = |S_i|$. A *brute force approach* evaluates all of m n-tuples with P. A *backtracking* approach builds up the solution vector one component at a time and to use modified criterion function $P_i(x_1, \cdots, x_n)$ (sometimes, it is called *bounding function*).

Example 1: 4-Queens problem (n Queens) Let $S = \{1, 2, 3, 4\}$ and $X = (x_1, x_2, x_3, x_4)$ for $x_i \in S$, where $x_i = j$ means that Queen i is placed at column j. Assume $P_{i-1}(x_1, \cdots, x_{i-1})$ is done. Then, $P_i(x_1, \cdots, x_i)$ is defined such that (i) $x_i \neq x_j$ for all $1 \leq j \leq i-1$ and (ii) $|i-j| \neq |x_i - x_j|$ for all $1 \leq j \leq i-1$.

Example 2: Sum-of-Subset Given a set of weights $W = \{w_1, \cdots, w_n\}$ and a number M, the problem is to find a subset of W such that the sum of the elements in the subset is equal to M. For example, for $W = \{11, 13, 24, 7\}$ and $M = 31$, two solutions $\{11, 13, 7\}$ and $\{24, 7\}$ exist. that can be also expressed as $X_1 = (1, 1, 0, 1)$ and $X_2 = (0, 0, 1, 1)$ with a bounding function $P : \sum_{j=1}^{i} x_j w_j \leq M$.

Note that the following bounding function may be more efficient for solving this problem: $P : \sum_{j=1}^{i} x_j w_j \leq M$ and $\sum_{j=i+1}^{n} w_j + \sum_{j=1}^{i} x_j w_j \geq M$.

Graph Coloring:

Hamiltonian Cycle:

0/1 Knapsack:

8 Branch-and-Bound

Definition: state space tree, live node, dead node, E-node, answer node (leaf), solution node, least cost search.

Branch-and-Bound approach is similar to the backtracking approach, better suited for optimization problem.

Traveling Salesman Problem:

Assume: (1) the tour starts at node 1, (2) we define the cost of each node in the state space tree to be the cost of reaching it from the root.

9 Lower Bounds

9.1 Ordered Search Problem

Given a list of n numbers in non-decreasing order, $A : a_1 \leq a_2 \leq \cdots \leq a_n$, and a numver x, the problem is to find j such that $a_j = x$ if such j exists.

Note that the binary search algorithm can solve this problem in $O(\log n)$ time in the worst case. In the following, we discuss that any algorithm solving this problem requires $\Omega(\log n)$ time in the worst case.

Consider all possible comparison trees which model algorithms to solve this searching problem. Let $FIND(n)$ denote the distance of the longest path from the root to a leaf node in the tree with n nodes, i.e., $FIND(n)$ denotes the worst case number of comparisons. Let k denote the depth of the tree. We then have $n \leq 2^k - 1$ which implies that $k \geq \lceil log(n+1) \rceil$. Hence, $FIND(n) \geq \lceil \log(n+1) \rceil$.

9.2 Comparison-based Sorting

Consider n numbers to be sorted using element-wise comparisons. Note that there are $n!$ different orders of n numbers. Let $T(n)$ denote the minimum number of comparisons required to sort n numbers in the worst case. The number of nodes in any binary comparison tree of depth k is at most 2^k. Since all $n!$ possibilities must be covered, we have $n! \leq 2^k = 2^{T(n)}$. This implies that $T(n) \geq \lceil \log n! \rceil$ which is approximated as $n \log n - n/\log_e 2 + \log_2 n/2 + O(1)$. Therefore, $T(n) = \Omega(n \log n)$.

10 NP-Completeness

Satisfiability Problem: Let $U = \{u_1, u_2, \cdots, u_n\}$ be a set of boolean *variables*. A *truth* assignment for U is a function $f : U \to \{T, F\}$. If $f(u_i) = T$, we say u_i is *true* under f; and if $f(u_i) = F$, we say u_i is false under f. For each $u_i \in U$, u_i and $\overline{u}_i$ are *literals* over U. The literal $\overline{u}_i$ is true under f if and only if the variable u_i is false under f. A *clause* over U is a set of literals over U such as $\{u_1, \overline{u}_3, u_8, u_9\}$. Each clause represents the disjunction of its literals, and we say it is

satisfied by a truth assignment function if and only if at least one of its members is true under that assignment. A collection C over U is *satisfiable* if and only if there exists a truth assignment for U that simultaneously satisfies all the clauses in C.

Satisfiability (SAT) Problem

Given: a set U of variable and a collection C of clauses over U

Question: is there a satisfying truth assignment for C?

Example:
$U = \{x_1, x_2, x_3, x_4\}$
$C = \{\{x_1, x_2, x_3\}, \{\overline{x}_1, \overline{x}_3, \overline{x}_4\}, \{\overline{x}_2, \overline{x}_3, x_4\}, \{\overline{x}_1, x_2, x_4\}\}$.

Let $x_1 = T$, $x_2 = F$, $x_3 = F$, $x_4 = T$.

Ans: yes

Reduction from SAT to 3SAT:

(1) $(x_1) \rightarrow (x_1 + a + b)(x_1 + a + \overline{b})(x_1 + \overline{a} + b)(x_1 + \overline{a} + \overline{b})$

(2) $(x_1 + x_2) \rightarrow (x_1 + x_2 + a)(x_1 + x_2 + \overline{a})$

(3) $(x_1 + x_2 + x_3 + x_4 + x_5) \rightarrow (x_1 + x_2 + a_1)(\overline{a_1} + x_3 + a_2)(\overline{a_2} + x_4 + x_5)$

- 3SAT
- Not-All-Equal 3SAT
- One-In-Three 3SAT

Definition:

P: a set of problems that can be solved deterministically in polynomial time.

NP: a set of problems that can be solved nondeterministically in polynomial time.

NPC: a problem B is called NP-complete or a NP-complete problem if (i) $B \in NP$, i.e., B can be solved nondeterministically in polynomial time, and (ii) for all $B' \in NP$, $B' \leq_P B$, i.e., any problem in NP can be transformed to B deterministically in polynomial time.

Cook's Theorem: Every problem in NP can be transformed to the Satisfiability problem deterministicall in polynomial time.

Note:

(i) The SAT is the first problem belonging to NPC.

(ii) To prove a new problem, say B, being NPC, we need to show (1) B is in NP and (2) any known NPC problem, say B', can be transformed to B deterministically in polynomial time. (By definition of $B' \in NPC$, every problem in NP can be transformed to B in polynomial time. As polynomial time transformation is transitive, it implies that every problem in NP can be transformed to B in polynomial time.)

Theorem: $P = NP$ if and only if there exists a problem $B \in NPC \cap P$.

Proof: If $P = NP$, it is clear that every problem in NPC belongs to P. Now assume that there is a problem $B \in NPC$ that can be solved in polynomial time deterministically. Then by definition of $B \in NPC$, any problem in NP can be transformed to B in polynomial time derterministically, which can then be solved in polynomial time deterministically using the algorithm for B. Hence, $NP \subseteq P$. Since $P \subseteq NP$, we conclude that $P = NP$, which completes the proof of the theorem.

Problem Transformations:

Note: The following discussion will not be included in the final exam.

Node Cover Problem:

Given: a graph G and an integer k,

Objective: to find a subset $S \subseteq V$ such that (i) for each $(u, v) \in E$, either u or v (or both) is in S, and (ii) $|S| \leq k$.

Hamiltonian Cycle Problem:

Given: a graph G

Objective: to find a simple cycle of G that goes through every vertex exactly once.

Hamiltonian Path Problem:

Given: a graph G

Objective: to find a simple path of G that goes through every vertex exactly once.

Vertex Coloring Problem:

Given: a graph G and an integer k

Objective: to decide if there exists a *proper* coloring of V (i.e., a coloring of vertices in V such that no two adjacent vertices receive the same color) using k colors.

- $3SAT \leq_P Node - Cover$

Let W be an arbitrary well-formed formula in conjunctive normal form, i.e., in sum-of-product form, where W has n variables and m clauses. We then construct a graph G from W as follows.

The vertex set $V(G)$ is defined as $V(G) = X \cup Y$, where $X = \{x_i, \overline{x}_i \mid 1 \leq i \leq n\}$ and $Y = \{p_j, q_j, r_j \mid 1 \leq j \leq m\}$. The edge set of G is defined to be $E(G) = E_1 \cup E_2 \cup, E_3$, where $E_1 = \{(x_i, \overline{x}_i) \mid 1 \leq i \leq n\}$, $E_2 = \{(p_j, q_j), (q_j, r_j), (r_j, p_j) \mid 1 \leq j \leq m\}$, and E_3 is defined to be a set of edges such that p_j, q_j, and r_j are respectively connected to c_j^1, c_j^2, and c_j^3, where c_j^1, c_j^2, and c_j^3 denote the first, second and the third literals in clause C_j.

For example, let $W = (x_1 + x_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$. Then G is defined such that $V(G) = \{x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, p_1, q_1, r_1, p_2, q_2, r_2, p_3, q_3, r_3\}$ and $E(G) = \{(x_1, \overline{x}_1)$, $(x_2, \overline{x}_2)$, $(x_3, \overline{x}_3)$, (p_1, q_1), (q_1, r_1), (r_1, q_1), (p_2, q_2), (q_2, r_2), (r_2, p_2), (p_3, q_3), (q_3, r_3), (r_3, p_3), (p_1, x_1), (q_1, x_2), (r_1, x_3), $(p_2, \overline{x}_1)$, (q_2, x_2), $(r_3, \overline{x}_3)$, $(p_3, \overline{x}_1)$, $(q_3, \overline{x}_2)$, $(r_3, \overline{x}_3)\}$.

We now claim that there exists a truth assignment to make $W = T$ if and only if G has a node cover of size $k = n + 2m$.

To prove this claim, suppose there exists a truth assignment. We then construct a node cover S such that $x_i \in S$ if $x_i = T$ and $\overline{x}_i \in S$ if $x_i = F$. Since at least one literal in each clause C_j must be true, we include the other two nodes in each triangle (i.e., p_j, q_j, r_j) in S. Conversely, assume that there exists a node cover of size $n + 2m$. We then note that exactly one of $x_i, \overline{x}_i$ for each $1 \leq i \leq n$ must be in S, and exactly two nodes in p_j, q_j, r_j for each $1 \leq j \leq m$ must be in S. It is then easy to see the S must be such that at least one node in each p_j, q_j, r_j for $1 \leq j \leq m$ must be connected to a node x_i or $\overline{x}_i$ for $1 \leq i \leq n$. Hence we can find a truth assignment to W by assigning x_i true if $x_i \in S$ and false $\overline{x}_i \in S$.