The Hong Kong University of Science & Technology

Department of Computer Science

COMP 271, Section 1: Design and Analysis of Algorithms

mid-term examination

Date: 4:30 - 6:30 p.m., 3 April, 1997.

Venue: LTB

Student name: ______________________________

Student ID: ______________________________

e_mail: ______________________________

Question	*Grade*
1	/12 pts
2	/12 pts
3	/12 pts
4	/12 pts
Total	/48 Pts

Problem 1

Answer the following short questions.

(a) For each part of expressions (A, B) below, indicate whether A is O, Ω, or θ of B. Note that zero, one or more of these relations may hold for a given pair; list all correct ones. No explanation is needed.

(i) $A = (n + a)^b$, $B = n^b$; a and b are real constants > 0.

$A = O(B), A = \Omega(B), A = \theta(B)$.

(ii) $A = 3^n$, $B = 3^{n+c}$; c is a real constant > 0.

$A = O(B), A = \Omega(B), A = \theta(B)$.

(iii) $A = O(f(n))$, $B = O(f(n))$; for some function $f(n)$.

no relation

(iv) $A = \Omega(f(n))$, $B = \Omega(f(n))$; for some function $f(n)$.

no relation

(b) Show how to sort n integers in the range 1 to n^2 in $O(n)$ time. Your need not to argue the correctness and running time of your algorithm.

Observe the fact that an integer in the range $[0 \ldots n^2 - 1]$ can be written as a 2-tuple of base-n digits, each in the range $[0 \ldots n - 1]$. More formally, any integer v in 0 to $n^2 - 1$, can be written as $< c_1, c_0 >$ such that

$$v = c_1 n + c_0.$$

where each c_1 and c_0 is in the range $[0 \ldots n-1]$. The c_1 and c_0 can be found by the following standard base-reduction procedure, which converts a number $v \in [0 \ldots n^2 - 1]$ into 2 digits base n:

```
Digits (v, n, 2) {
        j=0;
        for j=0 to 1 do {
                c[j++]=v%n;
                v=v/n;
        }
        return c[0,1];
}
```

We can apply the radix sort, stably sorting each of the 2 base-n digits in least to most significant order. Each sorting iteration takes $O(n)$ time and the number of digits is k. Therefore $O(2n)$ time is taken altogether. i.e., $O(n)$ running time.

(c) Consider the following recurrence relation,

$$T(n) = \begin{cases} 1 & \text{if } n = 1; \\ 2T(\lfloor \frac{n}{2} \rfloor) + n & \text{if } n > 1. \end{cases}$$

(i) Mr. Careless use the substitution method to argue the $T(n) = O(n)$. He guesses $T(n) \leq cn$ for some constant c. Clearly, $T(1) = 1 \leq c \cdot 1$ for some constant $c \geq 1$. Here is his inductive step,

$$\begin{aligned} T(n) &= 2T(\lfloor \tfrac{n}{2} \rfloor) + n \\ &\leq 2(c\lfloor \tfrac{n}{2} \rfloor) + n \text{ (use the inductive hypothesis)} \\ &\leq cn + n \\ &= \mathrm{O(n)}. \end{aligned}$$

Discuss briefly why his proof is not correct.

He has not proved the exact form of the inductive hypothesis, that is $T(n) \leq cn$ for some constant c.

(ii) Using the substitution method or other methods to solve the recurrence relation.

Problem 2

Given a sequence of numbers $a_1, a_2, \ldots, a_n$, and an integer i, $1 \leq i \leq n$, compute the i-th smallest element. This is called the **selection** problem.

(a) Give a $O(nlogn)$ algorithm for this problem. You need not to prove the correctness and running time of your algorithm.

We can sort the n numbers using merge sort (or any other $O(nlogn)$ sorting algorithm) in ascending order. We then can report the i-th smallest number in O(1) time. Therefore, the total running time is $O(nlogn)$.

(b) Mr. Careless proposes the following **randomzied** algorithm to the problem.

```
procedure Select(A[p...r],i)
{
        if (p == r) return(A[p]);
        q=Randomized_Partition(A[p...r]);
        k=q-p+1;
        if (i == k) return (A[q]);
        if (i < k) return Select(A[p...q],i));
        else return Select(A[q+1...r],i);
}
```

The procedure Randomized_Partition($A[p \ldots r]$) will randomly select an element x from the list $A[p \ldots r]$, called the pivot. It then partitions the elements into two subarrays, $A[p \ldots q-1]$ contains the element less than or equal to x, $A[q] = x$ and $A[q+1 \ldots r]$ contains the element greater than or equal to x.

To select the i-th smallest element from the array $A[1 \ldots n]$, Mr. Careless simply calls `Select(A[1...n], i)`. Mr. Careless's idea is nearly correctly, but he has made a fatal mistake. Briefly discuss why it is incorrect and write down the correction.

The algorithm determines the rank of x (i.e., $A[q]$), $k = q - p + 1$. If $i > k$, then we recurse on the right sublist, but we have eliminated the k elements from comparisons. Therefore, we look for $(i - k)$-th smallest element in the next recursive call.

The last line of the algorithm should be changed to

```
        else return Select(A[q+1...r],i-k);
```

(c) Mr. Careless has read your corrected procedure and understand your idea. But he claims the corrected randomized algorithm runs in $O(nlogn)$ for every case. Give a sequence of partitions that results in a worst-case performance of `Select(`$A[1 \ldots n]$`, i)` and argue briefly the worst case running time of `Select(`$A[1 \ldots n]$`, i)` is not $O(nlogn)$.

(d) What is the upper bound on the **expected time** of the corrected procedure `Select(`$A[1 \ldots n]$`, i)` ? Argue briefly how you deduce the answer.

There are n possible choices for the pivot element, each is equally likely to be selected with probability $\frac{1}{n}$. If x is the k-th smallest element in the list, then we create two sublist of sizes $k-1$ and $n-k$. If we assume that we recurse on the larger of the two list, we get the following recurrence:

$$
\begin{aligned}
T(n) &\leq \frac{1}{n}\left(\sum_{k=1}^{n} T(max(k-1, n-k))\right) + O(n) \\
&\leq \frac{2}{n}\left(\sum_{k=\frac{n}{2}}^{n-1} T(k)\right) + O(n)
\end{aligned}
$$

Problem 2
continue (d)

We solve the recurrence by substitution. Assume $T(n) \leq cn$ for some constant c that satisfies the initial conditions of the recurrence. Using the inductive hypothesis, we have

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\left(\sum_{k=\frac{n}{2}}^{n-1} ck\right) + O(n) \\
&\leq \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\frac{n}{2}-1} k\right) + O(n) \\
&\leq c(n-1) - \frac{c}{n}(\frac{n}{2}-1)(\frac{n}{2}) + O(n) \\
&= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \\
&= cn + O(n) - c(\frac{n}{4}+\frac{1}{2}) \\
&\leq cn,
\end{aligned}
$$

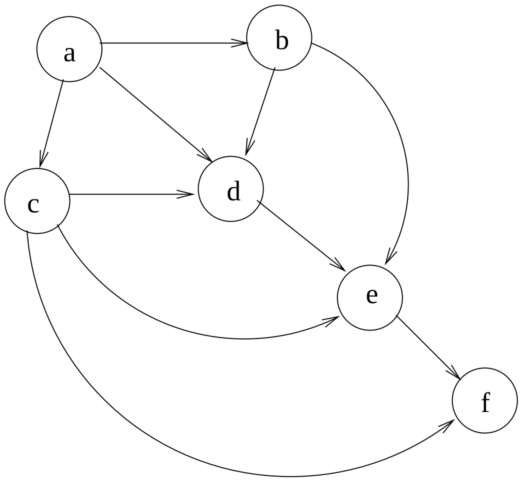
since we can pick c large enough so that $c(\frac{n}{4}+\frac{1}{2})$ dominates the $O(n)$ term.

By the principle of mathematical induction, we can conclude $T(n) \leq cn$. Therefore the `Select(`$A[1\ldots n]$`, i)` has linear worst case expected running time.

Problem 3

(a) Consider the following digraph $G = (V, E)$. Traverse the following digraph using Depth-First search starting from vertex a. You must use the adjacent list representation given below. Show the tree (both tree edges, forward edges, cross edges and time stamps) produced by the search.

$a :\rightarrow b, d, c$
$b :\rightarrow d, e$
$c :\rightarrow d, e, f$
$d :\rightarrow e$
$e :\rightarrow f$
$f :\rightarrow \phi$ (empty)



(b) A **topological sort** of a directed acyclic graph (DAG) is a linear ordering of vertices of the DAG such that for each edge (u, v), u appears before v in the ordering. Consider the following proposed algorithm which takes a directed acyclic graph $G = (V, E)$ as input parameter.

```
proposed_topological_sort(G) {
      run DFS(G), computing finish times f[v] for each vertex v;
      as each vertex is finished, insert it onto the front of a linked list;
      return the linked list;
}
```

Use the graph from (a) as the input for `proposed_topological_sort(G)`, write down the output of the procedure.

(c) Prove `proposed_topological_sort(G)` produces a topological sort of a directed acyclic graph G. You may need to use the following lemma to complete the proof.

Lemma: A digraph is acyclic if and only if DFS on G yields no back edges.

It suffices to show that for any pair of distinct vertices, $u, v \in V$, if there is an edge in G from u to v, then $f[v] < f[u]$. Since the input graph G is acyclic, we only have tree edges, forward edges and cross edges. (i) Let (u, v) be any tree/forward edge, since u is the ancestor of v, $f[v] < f[u]$. (ii) Let (u, v) be any cross edge, we must have finished explored v before we explore u (or else, (u, v) will not be a cross edge). We have $f[v] < f[u]$. In all cases, we also have $f[v] < f[u]$, for any $e = (u, v) \in E$. In conclusion, the `proposed_topological_sort(G)` produces a topological sort of a directed acyclic graph G.

Problem 4

This question is about **minimum spanning tree** problem. Given an connected, undirected graph $G = (V, E)$, a spanning tree is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together. Assuming that each edge (u, v) of G has a numeric weight or cost, $w(u, v)$, we define the cost of spanning tree T to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

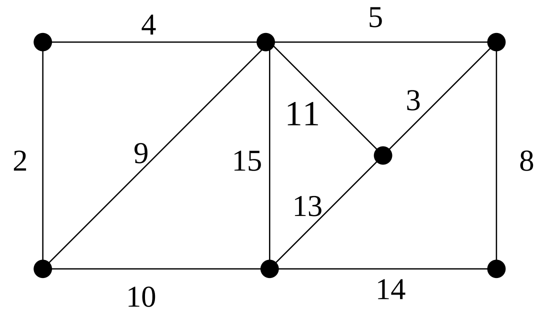
A minimum spanning tree (MST) is a spanning tree of minimum weight.

(a) In class we studied the **Kruskal's algorithm** to solve the MST problem. Write down the pseudo code of Kruskal's algorithm. Discuss briefly the idea of Kruskal's algorithm.

We add the cheapest of edge among all edges in each step. If the addition edge will not create a cycle, we accpet it, or else we discard it. The process continue until we have tested all edges.

```
Kruskal(G, w) {
       A=Empty;
       Sort the edges of E in increasing order by weight w;
       for each edge (u,v) in E (in order) {
              if (add (u,v) will NOT creat a cycle in A) {
                     we add (u,v) to A;
              }
       }
       return A;
}
```

(b) Given the following graph $G = (V, E)$. Trace the actions of Kruskal's algorithm. Show every intermediate step as the algorithm proceeds.



(c) The tricky part of Kruskal's algorithm is how to detect whether the addition of an edge will create a cycle. One way to solve the problem is to use the disjoint set UNION-FIND data structure. This data structure supports three operations:

Create-Set: Create a set containing a single item.
Find-Set: Find the set that contains a given item.
Union: Merge two sets into a common set.

Discuss in detail how UNION-FIND data structure can help us to detect whether the addition of an edge will create a cycle.

We can use the UNION-FIND data structure to detect the cycle, here is the modified code:

```
Kruskal(G, w) {
      A=Empty;
      for each v in V do
            Create-Set(v);
      Sort the edges of E in increasing order by weight w;
      for each edge (u,v) in E (in order) {
            if (Find-Set(u) != Find-Set(v)) {
                  we add (u,v) to A;
                  Union(u, v);
            }
      }
      return A;
}
```

We first create $|V|$ sets that contains a single element (i.e., we create $|V|$ separate trees). If $e = (u, v)$ belongs (i.e., `Find-set(u) == Find-set(v)`) to the same set(tree), the addition of (u, v) will create a cycle, we have to discard (u, v). If $e = (u, v)$ belongs to a different set(tree), (i.e., `Find-set(u) != Find-set(v)`) the addition of (u, v) will join two sets(tree) (`Union(u,v)`). These processes can help us to detect the addition edge will create a cycle or not.

Additional Problem

Following Problem 4, let `Union(u,v)` form the union of the sets containing u and v. To perform `Union(u,v)`, we first find sets containing u and v, set the size of this new set be the sum of two original sets, and then we relabel all the elements of the **smaller** of the sets being merged.

For example, let set $i = \{v_1, v_2, v_3\}$ and set $j = \{v_7, v_8\}$. After `Union` (v_1, v_7), set $i = \{v_1, v_2, v_3, v_7, v_8\}$ and v_7 and v_8 will be put in set i. And the size of set i is 5.

What is the running time of Kruskal's algorithm ? Explain briefly how you deduce the answer.

We first show that every vertex can be relabeled at most $lg|V|$ time. For any element v, the first time we relabel v, it has been unioned into a set whose size is at least 2 (or else we will not relabel v). The second time we relabel v, it must be merged with a set that us at least as large, so the new size must be at least 4. The third time v is relabeled, it must be merged with a set at least as large, so the new size must be at least 8. It is easy to see that if v is relabeled k times total, then its final set is of size 2^k. But size we only have $|V|$ vertices, we have $2^k \leq |V|$ implying that $k \leq lg|V|$.

Since every vertex can be relabeled at most lgV time, we spent at most $O(VlogV)$ time relabeling. And we need to spend $O(ElogE)$ time to sort the edges. Hence, the total running time of Kruskal algorithm is :

$$O(ElogE + VlogV).$$