COMP 271: Final Exam
2003 Spring Semester
27 May 2003
8:30 am – 11:30 am
solution sketches

You have 3 hours to solve the 6 problems for a total of 100 marks. Answer all problems in the space provided. You may also use the back of the test papers. Good luck!

Student name: ____________________ **Student ID:** ____________________

Problem	Your mark	Maximum
1		15
2		15
3		20
4		20
5		10
6		20
Total		100

Problem 1. (15 points) For $n > 0$, let $T(n)$ be defined by

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 5T\left(\frac{n}{4}\right) + n & \text{if } n > 1. \end{cases}$$

Give an asymptotic upper bound for $T(n)$. Make your bound as tight as possible. For the sake of this problem you may assume that n is always a power of 4.

The solution is $O\left(n^{\log_4 5}\right)$ *or, equivalently* $O\left(5^{\log_4 n}\right)$.

Problem 2. (15 points) Recall the *selection problem.*

Given an array of numbers $[a_1, \ldots, a_n]$, and an integer i, $1 \le i \le n$, find the ith smallest element. When $i = \lceil n/2 \rceil$, this is called the *median problem.*

Example: given $[1, 8, 23, 10, 19, 33, 100]$, the 3rd smallest element is 10. In this case $n = 7$ so $\lceil n/2 \rceil = 4$ and the median is 19.

Now assume that you are given a black-box worst-case linear-time median-finding algorithm. That is, you are given a procedure $X(\,)$ that takes as input an array $A[\,]$ with n elements and, in $O(n)$ time, returns the value of the median element in A.

(a) Using $X(\,)$ as a subroutine design a a linear time algorithm that solves the selection problem for an arbitrary i. That is given input array $A[\,]$ with n elements and input integer i your algorithm should find the ith smallest item in the array in $O(n)$ time. For the sake of this problem you may, if you like, assume that all elements in $A[\,]$ are unique.

(b) Prove that your algorithm runs in $O(n)$ time.

(c) Argue the correctness of your algorithm.

This was problem 3 from Tutorial 2.

There are at least two very different solutions for this problem. The first one is a straightforward divide-and-conquer, the second a clever transformation. We will see both.

I) Solution 1.
Let $Select(A, n, i)$ be the procedure that we want to construct. A is an array, n is the size of the array and we want to find the ith largest item in A.

$Select(A, n, i)$
{ $med = X(A)$; — $O(n)$ time to find median of A
  if $(i == \lceil \frac{n}{2} \rceil)$
    $return(med)$
  else if $(i < \lceil \frac{n}{2} \rceil)$
    Create a matrix $B[\,]$ containing the $\lceil \frac{n}{2} \rceil - 1$ elements of $A[\,]$ that are $< med$. — $O(n)$ time to create
    $Select(B, \lceil \frac{n}{2} \rceil - 1, i)$
  else — $(i > \lceil \frac{n}{2} \rceil)$
    Create a matrix $B[\,]$ containing the $n - \lceil \frac{n}{2} \rceil$ elements of $A[\,]$ that are $> med$. — $O(n)$ time to create
    $Select(B, n - \lceil \frac{n}{2} \rceil, i - \lceil \frac{n}{2} \rceil)$
}

Note that this algorithm will always terminate since, in each recursive call, the value of n decreases and no recursion call is made with $n < 1$ (verify this.

The correctness of this algorithm follows from the fact that there are only three possibilities

- $i == \lceil \frac{n}{2} \rceil$*: In this case we are looking for the median and get the correct answer via $X()$.*
- $i < \lceil \frac{n}{2} \rceil$*: In this case the ith smallest item in A is also the ith smallest item in $\{x, : x \in A$ and $x < med\}$ which is the set contained in B.*
- $i > \lceil \frac{n}{2} \rceil$*: In this case the ith smallest item in A is also the $i - \lceil \frac{n}{2} \rceil$th smallest item in $\{x, : x \in A$ and $x > med\}$ which is the set contained in B.*

The running time of the algorithm satisfies

$$T(n) = O(n) + \max\left\{T\left(\left\lceil\frac{n}{2}\right\rceil - 1\right), T\left(n - \left\lceil\frac{n}{2}\right\rceil\right)\right\}$$

which can be shown by induction to satisfy $O(n)$. Another way of analyzing the recurrence is to make the assumption that $T(n)$ is non-decreasing (as discussed in class this is valid) and noting that

$$\max\left\{\left\lceil\frac{n}{2}\right\rceil - 1, n - \left\lceil\frac{n}{2}\right\rceil\right\} \le n/2$$

so

$$T(n) \le O(n) + T(n/2)$$

which can be expanded to show that

$$T(n) \le O(n) + O(n/2) + O(n/4) + \ldots = O(n).$$

II) Solution 2.
Let $m = \lceil \frac{n}{2} \rceil$.

The idea is to create a new matrix B of size n' such that the median of B will be the ith element of A. There are three cases:

- $i == \lceil \frac{n}{2} \rceil$*: Then just set $n' = n$ and $B = A$.*
- $i < \lceil \frac{n}{2} \rceil$*:*
  First use $O(n)$ time to find $min = (\min_i A[i]) - 1$
  Set $n' = 2(m - i) + n$. Let the n' elements of B be the elements in A combined with $2(m - i)$ new elements all having the same value min. Since min is less than all of the elements in A the ith element in A is exactly the $i + 2(m - i) = (2m - i)$th element in B.

Now simply note that the median element of of n' has rank

$$\left\lceil \frac{2(m-i)+n}{2} \right\rceil = m - i + \left\lceil \frac{n}{2} \right\rceil = 2m - i$$

which is exactly the position of the ith element of A.

So, by calling $X(B)$ we can in $O(n') = O(n)$ time find the median of B which is the ith element in A.

- $i > \lceil \frac{n}{2} \rceil$*:*
  First use $O(n)$ time to find max $= (\max_i A[i]) + 1)$. Set $n' = 2(i - m) + n$. Let the n' elements of B be the elements in A combined with $2(i - m)$ new elements all having the same value max. Since max is greater than all of the elements in A the ith element in A is exactly the ith element in B.

  Now simply note that the median element of of n' has rank

  $$\left\lceil \frac{2(i-m)+n}{2} \right\rceil = i - m + \left\lceil \frac{n}{2} \right\rceil = i.$$

  So, by calling $X(B)$ we can in $O(n') = O(n)$ time find the median of B which is the ith element in A.

Problem 3. (20 points) For the following three statements circle either **True** or **False** depending upon whether the statement is either true or false. If the statement is true, prove the correctness from first principles, i.e., without relying on any lemmas and theorems proven in class. If the statement is false, give a counterexample.

In all of the statements $G = (V, E)$ is an undirected weighted graph with edge weights $w(e)$. You should also assume that $|V| > 2$.

e is a *minimum weight edge* in G if $w(e) = \min_{e' \in E} w(e')$.

e is a *maximum weight edge* in G if $w(e) = \max_{e' \in E} w(e')$.

A *complete* graph on $n = |V|$ vertices contains all $n(n-1)/2$ possible edges.

The notation $e \in T$ denotes that edge e is one of the edges in tree T.

(a) **True** / **False**

Let e be a minimum weight edge in G.
Then there exists some minimum spanning tree T for G such that $e \in T$.

The answer is **True**.

Proof: Suppose that there is some MST T such that $e \notin T$. Adding e to T forms a cycle. Let $e' \in T$ be any edge from the cycle.
$T' = T \cup \{e\} - \{e'\}$ is connected with $|V| - 1$ edges so T' is a tree. Now, since e is a minumum-weight edge $w(e) \le w(e')$ so

$$w(T') = w(T) + w(e) - w(e') \le w(T).$$

Since T is a **Minimum** *Spanning Tree it is impossible to have $w(T') < w(T)$ so we must have $w(T') = w(T)$ and T' is also a MST. Since $e \in T'$ we have just proven the statement.*

Note: When answering this question you could not assume that the minimum-weight edge was unique. You had to assume that it is possible for there to be more than one minimum-weight edge. Thus, you can not claim that $w(e) < w(e')$. You could only claim $w(e) \le w(e')$)

(b) **True** / **False**
Let e be a minimum weight edge in G.
Let T be any arbitrary minimum spanning tree for G.
Then it is always correct that $e \in T$.

The answer is **False**.

Counterexample: Let $G = (V, E)$ be a graph with three vertices $V = \{u, v, w\}$ and three edges $E = \{(u, v), (v, w), (w, u)\}$ such that all of the edges have the same weight

$$w((u, v)) = w((v, w)) = w((w, u)) = 1.$$

G has the three MSTs

$$T_1 = \{(u, v), (v, w)\}, \quad T_2 = \{(v, w), (w, u)\}, \quad T_2 = \{(u, v), (w, u)\}.$$

Now (u, v) is a minimum-weight edge (actually, all of the edges are minimum weight edges) but $(u, v) \notin T_2$.

(c) **True** / **False**

Let G be a complete graph.

Let e be a maximum weight edge in G.

Then there exists some minimum spanning tree T for G such that $e \notin T$.

The answer is **True**.

Proof: let T be a MST. If $e \notin T$ we are done. Otherwise $e \in T$.

Let $u, v \in V$ be such that $e = (u, v)$. There must be some edge $e' \in T$ such that either u or v is an endpoint of e' (since otherwise the vertices u and v would be disconnected from the rest of the vertices in spanning tree T, which is impossible). Without loss of generality suppose u is an endpoint of e', i.e., $e' = (u, w)$ for some $w \neq v$.

So far we have that $e, e' \in T$. Notice that $e'' = (w, v)$ is in graph G (since G is complete) but can not be in T since otherwise all of $e, e', e'' \in T$ and those three edges form a cycle. Therefore, adding e'' to T forms a cycle containing e so $T' = T \cup \{e''\} - \{e\}$ is connected and is therefore a tree. Since e is a maximum weight edge we have that $w(e) \geq w(e'')$ so

$$w(T') = w(T) - w(e) + w(e'') \leq w(T).$$

But T is a **minimum** *spanning tree so $w(T') = w(T)$ and T' is a MST as well.*

We have therefore shown that if $e \in T$ then there is some MST T' such that $e \notin T'$.

Note that in this problem you may not assume that the min-weight edge is unique, i.e., there may be many edges with the same max-weight cost.

Problem 4. (20 points) A *subsequence* of sequence $\langle a_1, a_2, a_3, \ldots, a_n \rangle$ is a sequence $\langle a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_t} \rangle$ where $i_1 < i_2 < \cdots < i_t$.

A *monotonically increasing subsequence* of a sequence is a subsequence that is increasing, i.e, each number in the subsequence is greater than its predecessor. For example, if the input sequence is $\langle 5, 24, 8, 17, 12, 45 \rangle$, then $\langle 5, 8, 45 \rangle$, $\langle 5, 8, 12, 45 \rangle$ and $\langle 5, 8, 17, 45 \rangle$ are all monotonically increasing subsequences.

A *longest* monotonically increasing subsequence of a sequence is a longest possible monotonically increasing subsequence. For the previous example the longest monotonically increasing subsequences are $\langle 5, 8, 12, 45 \rangle$ or $\langle 5, 8, 17, 45 \rangle$.

(a) Give an $O(n^2)$ time dynamic programming algorithm to find a longest monotonically increasing subsequence of a sequence of n numbers.

(b) Prove that your algorithm runs in $O(n^2)$ time.

(c) Argue the correctness of your algorithm.

This was question 5 in question-bank 3. Please see the solution given in question-bank 3 for this.

Problem 5. (10 points) Recall the following definitions:

A *vertex cover* of undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that every edge in G is incident to at least one vertex in S. That is, $\forall (u, v) \in E$ at least one of u, v is in S.

An *independent set* of of undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that no pair of vertices in S is joined by an edge in E. That is, if $u \in S$ and $v \in S$ then $(u, v) \notin E$.

The *size* of set $S \subseteq V$ is $|S|$, the number of vertices in S.

The *Decision Vertex Cover (DVC)* problem has input (G, k), k an integer, and answers yes if and only if G has a vertex cover S with $|S| = k$.

The *Decision Independent Set (DIS)* problem has input (G, k), k an integer, and answers yes if and only if G has an independent set S with $|S| = k$.

You are given the fact that DIS is NP-Complete. Prove that DVC is NP-Complete. More specifically prove the following two facts:

(a) $DVC \in \mathcal{NP}$.

(b) $DIS \leq_P DVC$.

As stated you may asume that DIS is NP-Complete. Any other lemmas you use must be proven from first principles. That is, you must prove any lemmas or theorems that you use.

(a) A **certificate** *for input G, k to DVC would be a set of vertices $S \subseteq V$.*

Verifying *the certificate S requires*

- *checking that $|S| = k$. This can obviously be done in $O(|V|)$ time.*
- *checking that S is a vertex cover. This can be done in polynomial time by loking at every edge $(u, v) \in E$ and checking that at least one of u, v is in S.*

(b) We first need to prove the following lemma:

Lemma: S is a vertex cover of $G = (V, E)$ if and only if $V - S$ is an independent set of G.

Proof: In the class notes you saw the following two facts.

1) S is an independent set of G if and only if S is a clique in $\overline{G}$ (the complement of G.
2) S is a clique in G if and only if $V - S$ is a vertex cover in $\overline{G}$.

The lemma follows directly from those two facts.

The lemma can also be proven directly as follows.

Suppose S is a vertex cover of G. If both $u, v \notin V - S$ then, from the definition of vertex cover, $(u, v) \notin E$. Thus $V - S$ is an independent set.

Now suppse $V - S$ is an independent set. From the definition of independent set if $e = (u, v) \in E$ then at least one of u, v is not in $V - S$. Therefore S is a vertex cover.

We now define the transformation

$$f(G, k) = (G, |V| - k)$$

that transforms inputs of DIS into inputs of DVC. Note that this transformation can obviously be calculated in polynomial time in the size of G.

Furthermore, from the lemma, we have that G has an independent set of size k if and only if G has a vertex cover of size $|V| - k$. This means that (G, k) is a yes-input to DIS if and only if $f(G, k)$ is a yes input to DVC. We have therefore proven $DIS \leq_P DVC$.

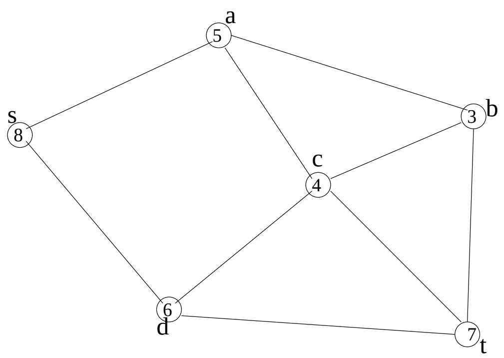
Problem 6. (20 points) In this problem the input is an undirected graph $G = (V, E)$ represented by adjacency lists. You are also given *vertex weights*, $w(v)$, for each $v \in V$.

The *vertex-bottleneck cost* of a path $P = v_1, v_2, v_3, \ldots, v_k$ is

$$cost(P) = \min\{w(v_1), w(v_2), \ldots, w(v_k)\}.$$

Intuitively, you can think of $w(v)$ as being the maximum number of units that can be shipped through v. When shipping items from v_1 to v_k along path $P = v_1, v_2, v_3, \ldots, v_k$ the maximum number of units that can be shipped is the cost of the path, since the vertex with the smallest weight acts as a bottleneck.

In the following graph vetex weights are given inside the vertex. Path s, a, c, b, t has cost $3 = w(b)$, path s, d, c, t has cost $4 = w(c)$ and path s, d, t has cost $6 = w(d)$.



A *max-vertex-bottleneck* $s-t$ path is a path in the graph from s to t with maximum cost. For example, in the given graph, s, d, t is a max-vertex-bottleneck $s-t$ path.

The *max-vertex-bottleneck problem* is, given (i) graph G, (ii) weights $w(v)$, and (iii) specified nodes s and t, to find a max-vertex-bottleneck $s-t$ path. In this question you need to design an efficient algorithm for solving the max-vertex-bottleneck problem. Full credit will be given for an $O((|V| + |E|) \log |V|)$ algorithm. Partial credit will be given for correct but slower algorithms.

Note: you are permitted to use data structures discussed in class as subroutines for this problem. If you do so, please carefully state in advance what operations the data structures support and what the running times for those operations are.

(a) Give an algorithm for solving the max-vertex-bottleneck problem.

(b) Analyze the running time of your algorithm.

(c) Prove the correctness of your algorithm.

We start by running an $O(|V|+|E|)$ BFS algorithm to find all of the vertices in the connected component containing s. If t is not in that component then we report there is no such path. Otherwise, since we are only concerned with the connected component containing s and t, we will assume that our graph is connected.

Note that this problem is very similar to the 3rd problem of the second homework set which asked for an algorithm to the max-edge-bottleneck problem. The same type of solution given there, a modified Dijkstra's shortest path algorithm, works here as well. Although what is described below is a formal solution it was not necessary to write somethnig so formal on the exam. Any solution that contained the basic ideas received credit.

The algorithm (given below) uses a **Max-heap** *(priority queue) instead of a* **Min-heap** *(priority queue). A max-heap permits, in $O(\log n)$ time extracting the item with largest key (***Extract-Max***) and an* **Increase-Key** *operation, in contrast to the* **Extract-Min** *and* **Decrease-Key** *operations of Min-heaps.*

Note that in the following black *vertices have already been processed (extracted from the queue) and* white *vertices are unprocessed. We will see that the output of the algorithm will be an array $d[u]$ that will store the value of some max-bottleneck $s-u$ path.* pred$[u]$ *will store the predecessor of u on some max-bottleneck $s-u$ path. The best $s-t$ path can be found by working backwards from t using this array.*

```
Bottleneck(G,w,s)
{                                             % Initialize
    for (each u ∈ V)
    {   d[u] = 0;
        color[u] =white;
    }
    d[s] = w(s);
    pred[s] = NIL;
    Q = (priority queue with all vertices);

    while (Non-Empty(Q))                      % Process all vertices
    {   u = Extract-Max(Q);                   % Find new vertex

        for (each v ∈ Adj[u] with color[v] == white)
            if (min (d[u], w(v)) > d[v])      % If estimate improves
            {
                d[v] = min (d[u], w(v));          relax
                Increase-Key(Q, v, d[v]);
                pred[v] = u;
            }
```

```
        color[u] =black;
    }
}
```

Note that this algorithm runs in $O(|E|\log|V|)$ time; the analysis is exactly the same as that of Dijkstra's algorithm.

Note that, at any given time, the set of edges

$$\{(\text{pred}(u), u) \,:\, u \textit{ a black node}\}$$

form a tree since, immediately before u is made black, the edge $(\text{pred}(u), u)$ connects a black vertex to a white one. Thus no edge $(\text{pred}(u), u)$ can create a cycle when it is added to the set.

Furthermore notice that, as long as all of the vertices have not yet been extracted, there will always be a vertex in the queue with non-zero key value. This follows from the fact that

$$\{(\text{pred}(u), u) \,:\, u \textit{ a black node}\}$$

always forms a tree. So, at any given time all neighbors of nodes in the tree have non-zero key cost. Since the graph is is connected, some neighbor of the tree must be in the queue.

Combining these two facts we find that at the end of the algorithm, the edges

$$\{(\text{pred}(u), u) \,:\, u \in V\}$$

form a tree. In particular, for every t, there is a unique path from s to t in the tree.

Let $D[u]$ be the actual cost of a max-vertex-bottleneck path from s to u. (Note that we will set $D[s] = w(s)$). We now claim that the unique $s-t$ path in this tree is a max-bottleneck $s-t$ path. This will follow from the lemma:

Theorem: *When vertex u is extracted from the queue then $d[u] = D[u]$.*

Proof: *The proof will be by contradiction. Suppose the theorem is incorrect. Let u be the first vertex extracted from the queue for which the theorem is wrong, i.e, $d[u] \neq D[u]$. Since $d[u]$ always corresponds to the bottleneck value of some $s-u$ path this implies that $d[u] < D[u]$. Note that, by the choice of u, all nodes v previously extracted from the queue satisfy $d[v] = D[v]$. Furthermore, $d[u]$ is a maximum value key currently in the queue.*

Let $s = v_0, v_1, \ldots, v_{m-1}, v_m = u$ be a max-bottleneck $s-u$ path.
Then $D[u] = \min_{1\leq i\leq m} w(v_i)$.

Now, $\forall i$, the path $s = v_0, v_1, \ldots, v_i$ has bottleneck value $\min_{1 \le j \le i} w(v_j)$ *so*

$$D[u] = \min_{1 \le j \le m} w(v_j) \le \min_{1 \le j \le i} w(v_j) \le D[v_i].$$

Set

$$i = \min\{k \,:\, v_k \textit{ not extracted from queue before } u\}$$

to be the index of the first item on the given $s - u$ path that is still in the queue when u is extracted. We point out that $i < m - 1$ since, otherwise, when v_{m-1} was processed, we would have found

$$d[u] \ge \min\left(D[v_{m-1}],\, w(u)\right) \ge D[u]$$

contradicting the fact that $d[u] < D[u]$.

This means that, at the time that u is extracted from the queue, v_i was still on the queue. But, since v_{i-1} was already extracted from the queue and its edges processed, we have that, at the moment that u is extracted from the queue

$$\begin{aligned} d[v_i] &\ge \min\left(D[v_{i-1}],\, w(v_i)\right) \\ &\ge \min\left(D[v_{i-1}],\, D[u]\right) \\ &\ge D[u] \\ &> d[u]. \end{aligned}$$

But, the fact that $d[v_i] > d[u]$ *contradicts the fact that u was extracted from the queue because it had maximum key value on the queue. We have just proven the theorem.*