COMP 3711 Design and Analysis of Algorithms
Fall 2011 Final Exam Solutions

1. **Quick-Answer Questions** ($5 \times 4 = 20$ pts)

   1.1 (d), (a), (c), (b), (e)

   1.2 7

   1.3 It's not a contradiction. The $\Omega(n \log n)$ lower bound holds for algorithms that only use comparisons on the elements, but radix sort is not a comparison-based algorithm.

   1.4 No. A simple example consists of 3 vertices: s, a, b. $w(s,a) = 2, w(s,b) = 4, w(a,b) = 3$. s is the source vertex.

2. **Majority** (10 pts)

   (a) Divide A into two parts $A[1..n/2]$ and $A[n/2+1..n]$. Since a majority element in A must be a majority in at least one of the halves, we recursively find a majority in $A[1..n/2]$ and $A[n/2+1..n]$. If $A[1..n/2]$ returns a majority element e, we scan the entire A to count its occurrences. If it's more than $n/2$, we return it. We do the same thing for the majority returned from $A[n/2+1..n]$ if it returns one. If we cannot find a majority after this, we return "no majority exists". The base case is when $n = 1$, we simply return the only element as the majority. The running time of the algorithm satisfies $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) = O(n \log n)$.

   (b) Initially set e =NULL and a counter $c = 0$. Then for $i = 1$ to n we do the following: If $c = 0$, we set $e = A[i]$. If $c > 0$, we check if $e = A[i]$. If so, we increment c by 1; else we decrement c by 1. We claim that in the end, e is the only possible majority if there exists one (why?). Then we scan A again to count the actual number of occurrences of e and decide if it is indeed a majority.

3. **Topological Sort** (8 pts)

Algorithm 1: DFS-visit(u)

```
output u; color[u] = BLACK;
foreach v ∈ Adj(u) do
    in-degree[v] = in-degree[v] −1;
    if in-degree[v]= 0 then DFS-visit(v);
end
```

   Another solution is to use DFS directly. After that, we output the nodes in the reverse order of their finishing times.

4. **Minimum Spanning Tree** (10 pts) Prim's algorithm uses a priority queue to organize all the nodes in the white color and find the lightest edge. As there are V vertices in a graph, the number of nodes in the queue is up to V. Extract-Min and Decrease-Key both take $O(\log V)$ so the total time complexity is $O(E \log V)$.

   In this problem, as the weights are only integers from 1 to k, we can put all vertices with the same weight of the lightest edge into one node (e.g., a linked list) of the priority queue. As there are only k possible weights, so the priority queue will only have up to k nodes. The Extract-Min and Decrease-Key operations will take $O(\log k)$.

   The total cost will be $O((V+E)\log k) = O(E \log k)$

5. **Counting Paths** (12 pts) Let $d[u]$ be the number of distinct paths from vertex u to target t. Set $d[t] = 1$. Do a topological sort of the DAG, then scan the vertices from the target t backwards to the source s. For each vertex u, we compute $d[u] = \sum_{v \in Adj(u)} d[v]$.

   If you process the nodes in the reverse topological order, you are guaranteed that all direct descendants are already computed when you visit any node.

Another (essentially the same) algorithm is to define $d[u]$ = number of distinct paths from source s to u. Initially set $d[s] = 1$, and $d[u] = 0$ for $u \neq s$. Then do a topological sort on G, and scan the vertices from s to t. For each u, we do $d[v] = d[v] + d[u]$ for every $v \in Adj(u)$.

The time complexity is O(E).

6. **Consulting Firm** (20 pts)

   (a) Counter example: $n = 4, M = 10$, and the profits are given by the following table:

	Month 1	Month 2	Month 3	Month 4
HK	80	90	1	10
BJ	20	30	12	1

   (b) Define $H[i]$ as the optimal net profit from month 1 to month i that ends in HK in month i, and $B[i]$ as the optimal net profit from month 1 to month i that ends in BJ in month i. The base case is $H[1] = H_1, B[1] = -\infty$. The recurrence is:

   $$\begin{aligned} H[i] &= \max\{H[i-1] + H_i, B[i-1] - M + H_i\} \\ B[i] &= \max\{B[i-1] + B_i, H[i-1] - M + B_i\} \end{aligned}$$

   The final optimal solution is the larger of $B[n]$ and $H[n]$.

   To construct the actual plan, we record the choices we have made in two other arrays CH and CB. $CH[i] = 0$ means $H[i] = H[i-1] + H_i$ and $CH[i] = 1$ means $H[i] = B[i-1] - M + H_i$ and similarly for CB. Suppose the final optimal solution is $H[n]$. Then we first recursively print the solution for $H[n-1]$ if $CH[n] = 0$, or $B[n-1]$ if $CH[n] = 1$; then we print "HK in month n". If the final optimal solution is $B[n]$, the process is similar.

   (c) We define $H[i]$ and $B[i]$ the same way as before, except that we don't have restrictions on the starting month. Just that the recurrence changes to:

   $$\begin{aligned} H[i] &= \max\{H_i, H[i-1] + H_i, B[i-1] - M + H_i\} \\ B[i] &= \max\{B[i-1] + B_i, H[i-1] - M + B_i\} \end{aligned}$$

   The optimal solution is the largest in $H[i], B[i], i = 1, \ldots, n$.

7. **Greedy Algorithm** (10 pts)

Algorithm 2: Greedy

```
j=1;
for i = 1 to m do
    while X[i] ≠ Y[j] do
        j = j + 1;
        if j = n + 1 then output "no" and halt;
    end
    j = j + 1;
end
output "yes";
```

Correctness: If X is not a subsequence of Y, the algorithm will obviously output "no". Now suppose X is a subsequence of Y. This greedy algorithm will match each $X[i]$ to the first matching $Y[j]$. Suppose there is another matching M between X and Y. We will convert this matching to the greedy matching, so the greedy algorithm will always output "yes". In M, we find the first $A[i]$ that it matches to $Y[j']$ which is different from the $Y[j]$ matched by the greedy algorithm. By the greedy nature of the algorithm, we must have $j' > j$. Then we modify M so that $A[i]$ matches with $Y[j]$. We then do this repeatedly until we have converted M to the greedy matching.