Fundamental Algorithm: Final Exam: Solutions

## Problem 1: 5 points

Put the following functions in increasing order of order-of-magnitude growth. If two functions have the same order of magnitude growth, indicate that.

$n^2$; $n \log n$; $n \log^2 n$; $n^2 \log n$; $n(n+1)/2$; $2^n$; $2^{\log n}$

**Answer:** $2^{\log n}(= n) < n \log n < n \log^2 n < n^2 \equiv n(n+1)/2 < n^2 \log n < 2^n$.

## Problem 2: 15 points

A. Suppose that you have a set $W$, implemented as a ordered linked list, and you want to create a 2-3 tree for $W$. Show that this can be done in linear time; that is $O(|W|)$.

   **Answer:** Work from the bottom up. Go through the items in $W$ in groups of 3 and create a parent node for those 3. At the end, if you get to the point where there are 4 items left group them in two groups of 2; if you get to the point where this are 2 items left, group them together; otherwise, you can group all the items in 3. Keep iterating at higher levels until you reach the root. The total time spent is proportional to the total number of nodes in the tree, and the total number of nodes in the tree is less than twice the number of leaves (actually about 3/2 the number of leaves.)

B. Suppose that you have two sets $S$ and $T$, each represented in a separate 2-3 tree. Assume that both trees are tagged with the size of the set. You wish to generate a new 2-3 tree for $S \cap T$; this should be non-destructive for both $S$ and $T$. Show how this can be done in time $O(\min |S| \log |T|, |T| \log |S|)$.

   **Answer:** If $|S| < |T|$, then loop through the items in $S$, search for it in $T$. If it is in $T$, then add it to a linked list. At the end, convert the linked list into a 2-3 tree as in part (A). If $|S| \geq |T|$ do the reverse.

C. Show how $S \cap T$ can be constructed in time $O(|S| + |T|)$.

   **Answer:** Loop through the elements of $S$ and $T$ in parallel, using a merge procedure. Put the common elements into a new 2-3 tree, using the method of part (A).

## Problem 3: 20 points

Let $G$ be a DAG. A vertex in $G$ is a *sink* if it has no outarcs. A *forward path* from vertex $U$ is a path that ends in a sink. Vertex $V$ is a *terminus* of vertex $U$ if $V$ is a sink and there is a path from $U$ to $V$.

A. Construct an algorithm `NumForPaths(G)` that computes the number of forward paths from every node in DAG $G$ in linear time. If $U$ is itself a sink, then `NumForPath[U]` should be 1.

   **Answer:** Do a DFS of $G$. In post-order (that is, just before DFSVisit(U) returns), insert the following step:

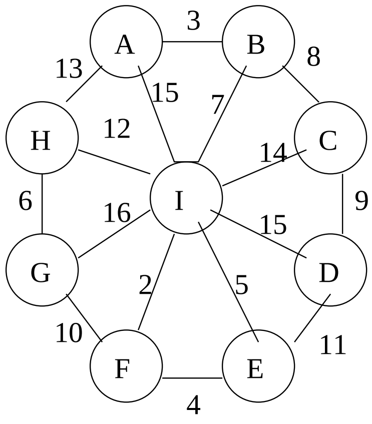
```
if (U has no outarcs)
  then NumForPaths[U]=1
  else NumForPath[U] = sum of NumForPath[V] over all V such that U-->V
```

B. Construct an algorithm `NumTerminus(G,U)` that computes the number of terminuses for vertex $U$ in DAG $G$ in linear time.

**Answer:** Keep a global counter. Call DFSVisit(U). Increment the counter each time the procedure reaches a white node with no outarcs.

## Problem 4: 15 points

Consider the problem of finding the minimum spanning tree in the graph below.



A. Show the sequence in which Prim's algorithm, starting from vertex A, adds edges to the tree.

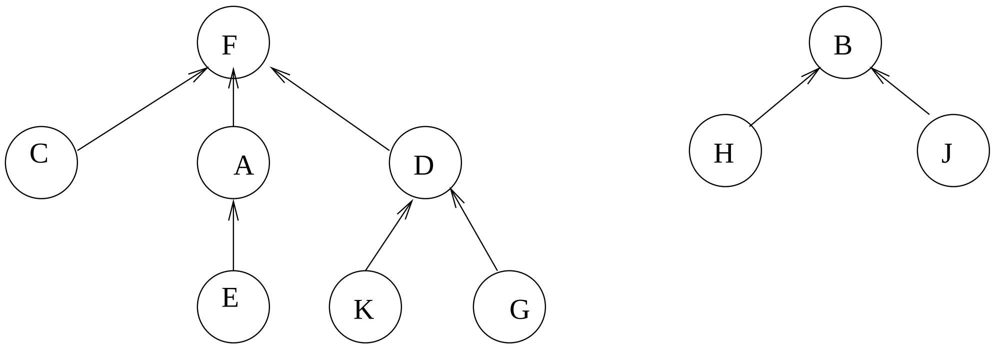
**Answer:** A-B, B-I, I-F, F-E, B-C, C-D, F-G, G-H

B. Show the sequence in which Kruskal's algorithm adds edges to the tree.

**Answer:** I-F, A-B, F-E, H-G, B-I, B-C, C-D, F-G

## Problem 5: 10 points

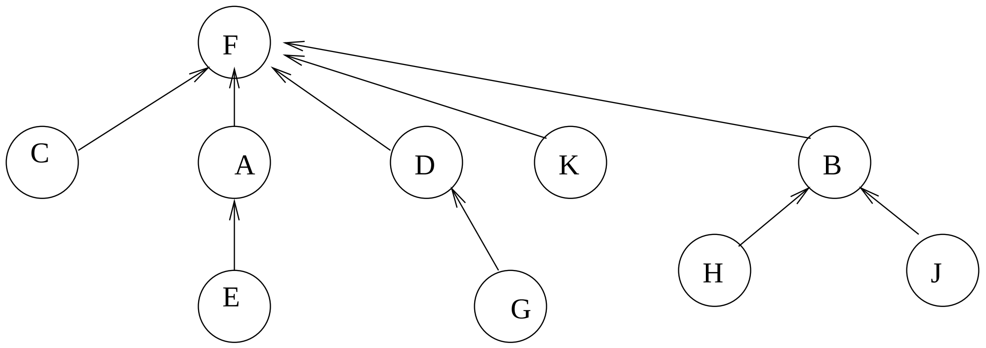
Assume that a collections of Union-Find disjoint sets is implemented using the tree-base implementation, with rank-based merging and path-compression. After a certain number of operations, the forest of trees looks like this:



A. Give a sequence of operations that could have created this forest. Each operation should have the form "Union(Find(x),Find(y))" ; you may abbreviate this as "UF(x,y)".

   **Answer:** UF(C,F). UF(E,A). UF(A,F), UF(K,D). UF(G,D), UF(D,F). UF(H,B), UF(J,B).

B. What is the outcome of carrying out the operation "Union(Find(K),Find(H))"?



## Problem 6: 20 points

Let $G$ be a directed graph where each edge $U \rightarrow V$ has a cost $C[U, V]$. Modify the Floyd-Warshall algorithm so that it returns two matrices: $A[U, V]$ is the cost of the shortest path from $U$ to $V$ and $M[U, V]$ is the number of different paths from $U$ to $V$ that have the minimal cost. For instance, in the graph below, there are seven different paths of cost 8 from A to G:

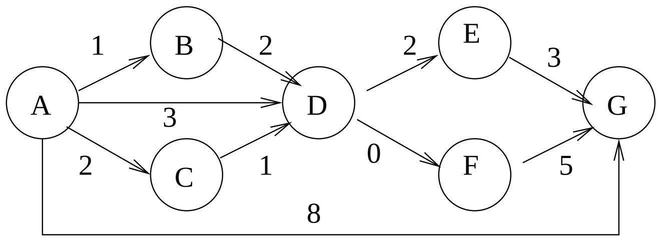
```
A--->B--->D--->E--->G
A--->B--->D--->F--->G
A--->C--->D--->E--->G
A--->C--->D--->F--->G
A--->D--->E--->G
A--->D--->F--->G
A--->G
```

Hint: When you are comparing the shortest paths from $I$ to $J$ through $K$ with the shortest paths already found from $I$ to $J$, you have to consider three cases:

A. The old paths are shorter.

B. The new paths are shorter.

C. The paths are the same length.



**Answer:** In case (A), $M[I,J]$ is unchanged. In case (B) we switch to the paths $I \rightarrow K \rightarrow J$. The number of these is $M[I,K] \cdot M[K,J]$. In case (C), we combine both sets of paths. In that case we have a total of $M[I,J] + M[I,K] \cdot M[K,J]$ shortest paths. So the algorithm is as follows

```
for (I=1 to N)
   for (J=1 to N) {
     A[I,J]=Cost[I,J];
     M[I,J]= 1;
   }
for (K=1 to N)
   for (I=1 to N)
     for (J=1 to N) {
        NewShort = A[I,K}+ A[K,J];
        if (NewShort < A[I,J]) {
          A[I,J] = NewShort;
          M[I,J]=M[I,K] * M[K,J];
          }
        elseif (NewShort == A[I,J])
          M[I,J] = M[I,J] + (M[I,K] * M[K,J]);
      }
```

## Problem 7: 20 points

You are running for Governor and you have a budget of $L$ dollars to spend on advertising. A research company has assembled a table `Votes[C,Q]` which says, for each county $C$, how many votes you will get if you spend $Q$ dollars on advertising. Assume the counties are numbered $1..K$. Assume also that spending more money never causes you to get fewer votes; that is, for each county $C$ `Votes[C,Q]` is a monotonically non-decreasing function of $Q$.

A. (15 points) Construct an efficient dynamic programming algorithm that computes how many votes you can get for the optimal spending strategy. You may write the algorithm in terms of a recursive routine with memoization. The algorithm should run in time polynomial in $L$ and $K$.

   **Answer:** If you decide to spend $I$ dollars in the $N$th county, then you have $L - I$ dollars to spend in counties $1 \ldots N-1$, and you should obviously spend those as efficiently as possible. So if you try every possible value of $I$ and take the maximum. Therefore the following algorithm works. Let Total[C,I] be an array holding the total number of votes obtained if you spend $I$ dollars in counties $1 \ldots C$. Then

```
function TotalVotes(C,I)
  if (Total[C,I] != null)
    then return(Total[C,I])
    else {
       Best=0;
         for (J=0 \ldots I)
           Best=max(Best,TotalVotes(C-1,I-J) + Votes(C,J))
       Total[C,I]=Best;
       return Best;
     }
```

   This can be written iteratively as follows:

```
for (C=1 to N)
  for (I = 0 to L)  {
     Total[C,I]=0;
     for (J=0 to I)
        Total[C,I] = max(Total[C,I],Total[C-1,I-J]+Votes(C,J))
    }
```

B. (5 points) What is the running time of your algorithm?

**Answere:** Either way, the total time is $O(N \cdot L^2)$