COMP 3711 Design and Analysis of Algorithms
Fall 2012 Final Exam

1. **Quick-Answer Questions** (15 pts)

   1.1 (6 pts) Please arrange the following functions in asymptotic ascending order (e.g., n, n^2, n^3): (a) n; (b) $\log n$; (c) $\log\log n$; (d) $\log^* n$; (e) $2^{2\log n}$.

   Solution: d, c, b, a, e

   1.2 (5 pts) Given n integers in the range 0 to n^3, what is the fastest sorting algorithm (asymptotically) to sort them?

   Solution: Use radix sort with base n. The time complexity is for radix sort is $\Theta(d(n+k))$. Here $k = n$ and $d = 4$, and the time complexity is $\Theta(n)$.

   1.3 (4 pts) Put "$\forall$" and "$\exists$" into the following blankets.

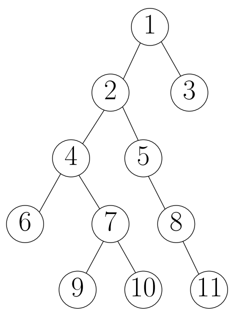
   The precise meaning of "the problem can be solved in $O(n)$ time" is:

   _______ a correct algorithm for the problem, _______ $c > 0$, _______ $n > 0$, such that _______ input of size n, the running time of the algorithm on the input is at most cn.

   Solution: $\exists$ a correct algorithm for the problem, $\exists$ $c > 0$, $\forall$ $n > 0$, such that $\forall$ input of size n, the running time of the algorithm on the input is at most cn.

2. **Longest Path in a Tree** (10 pts)

Design an algorithm that, given a binary tree, finds the longest path in the tree. For example, in the tree below, the longest path is 9-7-4-2-5-8-11. Your algorithm just needs to output the two endpoints of the path, i.e., 9 and 11 in this example. For full credits, your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree.



Solution:

(1) Let $d(u)$ be the length of the longest path from u to one of the leaves below u. This does not directly solve the original problem, but we can find the length of the longest path with u as the highest node as follows:

$$path\_length(u) = \begin{cases} d(u.left) + d(u.right) + 2 & \text{if u.left != null and u.right != null,} \\ d(u.left) + 1 & \text{if u.left != null and u.right = null,} \\ d(u.right) + 1 & \text{if u.left = null and u.right != null,} \\ 0 & \text{if u is a leaf.} \end{cases}$$

Then we just find the u with the maximum $path\_length(u)$.

(2) Recurrence for $d(u)$:

$$d(u) = \begin{cases} \max\{d(u.left), d(u.right)\} + 1 & \text{if u.left != null and u.right != null,} \\ d(u.left) + 1 & \text{if u.left != null and u.right = null,} \\ d(u.right) + 1 & \text{if u.left = null and u.right != null,} \\ 0 & \text{if u is a leaf.} \end{cases}$$

(3) We can compute all the $d(u)$'s from leaves to the root. The order can be obtained by a post-order traversal of the binary tree. The running time is $O(n)$.

(4) For each u, let $c(u)$ be the child of u that the DP has chosen during the bottom-up computation of $d(u)$. After we have found the u with the maximum $path\_length(u)$, we trace down the two paths starting from $u.left$ and $u.right$ (if they exist) following the $c()$ array.

3. **Shortest Path on an Chessboard** (10 pts)

   In the game of chess, the Knight can move two squares horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter "L". You are given two squares on an $n \times n$ chessboard. Design an algorithm to determine the shortest sequence of knight moves from one square to the other. For full credits, your algorithm should run in $O(n^2)$ time.

   Solution: We build an undirected graph $G = \{V, E\}$, where V consists of n^2 nodes corresponding to the n^2 squares in the chessboard. For two nodes $v_{i,j}$ and $v_{p,q}$, they are connected by an edge if and only if the Knight can directly move from $[i, j]$ to $[p, q]$. Then we just need to find the shortest path from the source square to the destination square. Since the graph is unweighted, we can use BFS which takes time $O(V + E)$. In this graph, we have n^2 vertices. Each vertex has 8 neighbors so there are at most $8n^2 = O(n^2)$ edges. So the running time is $O(n^2)$.

4. **Bottleneck Spanning Tree** (10 pts)

   A *bottleneck spanning tree* T of an undirected, weighted graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T. Give a linear-time algorithm that, given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.

   Solution: Obtain a new graph G' from G by linear search on the edge set of G and remove all the edges with weight larger than b. Apply BFS or DFS on G'. If the result is one connected component, return YES. Otherwise return NO. The running time is obviously $O(|V| + |E|)$.

   One more question: How do you find a bottleneck spanning tree with the minimum value in $O(E \log V)$ time (assuming the graph is connected)?

   Solution: Sort all edges in increasing order of their weights ($O(E \log E)$ time). Then delete all the edges, and we have V singleton vertices. Add the edges back one by one, and we stop as soon as all vertices are connected. This can be checked by using the disjoint set data structure. Note this is the same as in Kruskal's algorithm, except that we keep all edges, even if an edge creates a cycle. This step takes $O(E \log V)$ time. Finally, we find any spanning tree of these edges. We can just run BFS and return the BFS-tree (or run DFS and return the DFS-tree).

5. **Non-adjacent Elements in an Array** (15 pts)

Given an array of n positive numbers $A[1], A[2], \ldots, A[n]$. Our goal is to pick out a subset of non-adjacent elements whose sum is maximized. For example, if the array is $(1, 8, 6, 3, 6)$, then the elements chosen should be $A[2]$ and $A[5]$, whose sum is 14.

(a) (5 pts) Give an example to show that the following algorithm does not always find a subset of non-adjacent elements whose sum is maximized.

```
Start with S equal to the empty set
While some elements remain in A
  Pick the largest A[i]
  Add A[i] to S
  Mark A[i] and its neighbors as deleted
Endwhile
Return S
```

Solution: $(1, 4, 6, 4)$. The above algorithm gives the result is $(1, 6)$ with the sum is 7. The optimal solution should be $(4, 4)$ with the sum is 8.

(b) (10 pts) Give an algorithm based on dynamic programming that finds a subset of non-adjacent elements in A whose sum is maximized. For full credits, the running time should be $O(n)$. All elements in A are positive.

Solution:
Step1: Space of subproblem
For $0 \leq i \leq n$, define $d[i]$ be the maximum sum of the subset of non-adjacent elements from the first i elements in A. $d[n]$ is the optimal value for our original problem.

Step2: Recursive formulation
Base cases ($i = 0, i = 1$): $d[0] = 0$, $d[1] = A[1]$.
Recursive cases ($i > 1$): $d[i] = \max(d[i-1], d[i-2] + A[i])$

Step3: Bottom-up computation
For $0 \leq i \leq n$, we solve the subproblems $d[i]$ in increasing order of i, i.e. from $i = 0$ to $i = n$.

Step4: Construction optimal solution
For $1 \leq i \leq n$, define $s[i] = 0$ if $d[i-1] > d[i-2]+A[i]$, and $s[i] = 1$ if $d[i-1] \leq d[i-2]+A[i]$. Then we use the following procedure `PrintSubset` to output the subset after $d[n]$ is computed.

```
PrintSubset(s,i)
if (i > 0)
  if (s[i] == 1)
    output A[i];
    PrintSubset(s,i-2);
  else
    PrintSubset(s,i-1);
  Endif
Endif
```

The initial call is `PrintSubset(s,n)`. The running time of this algorithm is obviously runs in $O(n)$.

6. **Longest Balanced Subsequence** (15 pts)

A string of parentheses is said to be balanced if the left- and right-parentheses in the string can be paired off properly. For example, the strings (()) and ()() are both balanced, while the string (()))(is not. Given a string S of length n consisting of parentheses, design an algorithm to find the longest subsequence of S that is balanced. For this problem, you will gain full marks as long as your algorithm is correct and runs in polynomial time.

Solution by Dynamic Programming

Step1: Space of subproblem
For $1 \leq i \leq n, 1 \leq j \leq n$, define $D[i,j]$ be the longest balanced subsequence of the substring $S[i..j]$.

Step2: Recursive formulation
Base cases ($i \geq j$): $D[i,j] = 0$.

Recursive cases ($1 \leq i < j \leq n$):

$$D[i,j] = \max \begin{cases} [1]\, D[i+1, j-1] + 2 & S[i] =' (' \text{ and } S[j] =')' \\ [2]\, \max\limits_{i<k<j} \{D[i,k] + D[k+1,j]\} & \\ [3]\, D[i+1, j] & \\ [4]\, D[i, j-1] & \end{cases}$$

Step3: Bottom-up computation
Just like Matrix Multiplication Problem, we compute the $D[i,j]$ in increasing order of $|j-i|$.

Step4: Construction optimal solution
For $1 \leq i \leq n, 1 \leq j \leq n$, define $c[i,j]$ to record which case we choose to get the solution for the subproblem. Then we can start from $c[1,n]$ and recursively construct the optimal solution.

7. **Greedy Algorithm** (15 + 10 pts)

A teacher assigns a homework with n questions at the beginning of the first day of class. Each homework question carries a certain number of marks, if submitted on or before a specified deadline; otherwise it earns no marks. Each homework question takes exactly one day to complete. Your task is to find a schedule to finish the homework questions so as to maximize marks earned.

For instance, suppose there are seven questions with deadlines and marks as follows:

Problem	1	2	3	4	5	6	7
Deadline	1	1	3	3	2	2	6
Marks	6	7	2	1	4	5	1

Then the maximum number of marks one can obtain is 15, which can be achieved by completing the problems in the order of 2, 6, 3, 1, 7, 5, 4. Note that there are also other schedules that obtain the same marks.

For this problem, you will gain full marks as long as your algorithm is correct and runs in polynomial time. You get 10 bonus marks if your algorithm runs in time $O(n \log n)$.

Solution: Let $S[i]$ be the question we do on day i. Initially, it's all empty.

step 1 Sort all questions by the descending order of marks. Denote these questions (after sorting) by $p_1, \dots, p_n$, and their corresponding deadlines and marks by $d_1, \dots, d_n$ and $m_1, \dots, m_n$. We have $m_1 \geq m_2 \geq \cdots \geq m_n$.

step 2 For each $i = 1$ to n:
We try to do p_i as late as possible but on or before its deadline d_i. More precisely, we scan the array $S[.]$ from $S[d_i]$ to $d[1]$ to find the first empty position, say e_i, and then set $S[e_i] = i$. If all positions are occupied, we put it in the last available day (so we will not earn marks for this question).

Correctness proof: We denote our schedule by $S[.]$ and the optimal schedule by $OPT[.]$. We look at $p_1, p_2, \dots$ in order, and find the first p_i that S and OPT do on different days. Suppose $S[a] = i$, $OPT[b] = i$. Note that $OPT[a]$ must store some j such that $m_j \leq m_i$. Now, we swap $OPT[a]$ and $OPT[b]$ in the optimal solution. There are a few cases:

(a) If $a > d_i$, that means greedy does not earn its marks. Then OPT cannot earn its marks, either. So we have $d_i < b < a$. Thus this swap does not make OPT worse.

(b) If $a < d_i$, that means greedy has earned p_i. If OPT has also earned p_i, then we must have $b < a$. Then this swap will not make OPT worse. If OPT has not earned p_i, then this swap cannot make OPT worse, either, since OPT will earn p_i in exchange for p_j which as lower (or the same) marks.

We can repeatedly apply the above transformation until OPT and S are the same.

Running time: Step 1 takes $O(n \log n)$ time. Step 2 has n iterations, each takes $O(n)$ time to find an empty slot. So the total running time is $O(n^2)$.

Bonus marks: The bottleneck of the above algorithm is to find the latest empty slot before the deadline. To speed up this step, we insert all the empty slots in an AVL-tree. Given a deadline, finding the latest empty slot before the deadline in the AVL-tree takes $O(\log n)$ time. Once a slot is taken, we delete it from the AVL-tree. In total we do n insertions, n finds and n deletions. The total time is thus $O(n \log n)$.

8. **Huffman Coding** (10 pts)

What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a	b	c	d	e	f	g	h
1	1	2	3	5	8	13	21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers? Recall that the Fibonacci numbers are defined inductively as $f_1 = f_2 = 1, f_n = f_{n-2} + f_{n-1}$ for $n \geq 3$. You need to justify your answer.

Solution: An optimal Huffman code for the given alphabet is:
a:0000000
b:0000001
c:000001
d:00001
e:0001
f:001
g:01
h:1

When the frequencies are the first n Fibonacci numbers (l_i denotes the letter with frequency f_i):
l_n : 0^{n-1}
l_i : $0^{i-1}1$ for $i \in \{1, ..., n-1\}$

In building the Huffman tree, two trees with the smallest frequencies will always be merged first. Since $f_{k+2} > \sum_{i=1}^{k} f_i$ (to be proved by induction) and $f_{k+2} > f_{k+1}$ (more obvious), we can see that the two trees with letters $\{l_i | i \in \{1, ..., k\}\}$ and $\{l_{k+1}\}$ will always be merged before $\{l_{k+2}\}$, and the result is a "linear" tree structure that gives the above codes.

Proof of $f_{k+2} > \sum_{i=1}^{k} f_i$ by induction:
Base case: $f_3 > f_1$
Assume $f_{k+1} > \sum_{i=1}^{k-1} f_i$ holds. Then $f_{k+2} = f_{k+1} + f_k > \sum_{i=1}^{k-1} f_i + f_k = \sum_{i=1}^{k} f_i$. $\square$