Data Structure and Algorithm II, Spring 2011

Midterm Examination

140 points

Time: 9:10am-12:10pm (180 minutes), Friday, April 22, 2011

Problem 1. In each of the following question, please specify if the statement is **true** or **false**. If the statement is true, explain why it is true. If it is false, explain what the correct answer is and why. (20 points. For each question, 1 point for the true/false answer and 3 points for the explanations.)

1. $n \log n$ is polynomially larger than n.
2. $n^{1+10^{-100}}$ is polynomially larger than $n \log n$.
3. As long as a problem exhitbits optimal substructure, it can be solved by using a greedy algorithm.
4. The solution of $T(n) = 4T(n/2) + n^2$ is $T(n) = \Theta(n \log n)$.
5. As long as a recurrence is of the form $T(n) = aT(\frac{n}{b}) + f(n)$, it can be solved by using the master method.

Sol:

1. **False**. $n \log n$ is not larger than n by a factor n^ϵ, for any $\epsilon > 0$.
2. **True**. Let $f(n) = n^{10^{-100}}$ and $g(n) = \log n$. We can see from $\log n = O(n^\epsilon)$, for any $\epsilon > 0$, that $f(n)$ is polynomially larger than $g(n)$.
3. **False**. Take the 0-1 knapsack problem as a counterexample.
4. **False**. By master theorem, the solution is $\Theta(n^2 \log n)$.
5. **False**. The recurrence of the form $T(n) = 2T(\frac{n}{2}) + n \log n$ can't be solved the master theorem, since $n \log n$ is not polynomially larger than n. (However, this can be solved by using the extended version of the master theorem.)

Problem 2. "Short answer" questions: (25 points)

1. What is the main job of a Quality Assurance (QA) engineer in a software company? (4 points)

2. Give a formal definition of "uniform random permutation". (4 points)

3. Derive an optimal Huffman code for the first n Fibonacci numbers, i.e., $\{c_1 : 1\ c_2 : 1\ c_3 : 2\ c_4 : 3\ c_5 : 5\ c_6 : 8\ c_7 : 13\ c_8 : 21\ ...\ c_n : F(n)\}$. In other words, show the general form of the codeword for the i-th character c_i (with $F(i)$ as its frequency). (5 points)

4. Give two reasons for implementing the paper prototype instead of the "real" prototype in a software project. (4 points)

5. How do we roughly estimate the cost of a software product? (4 points)

6. Give two reasons for writing down the specifications of a software project before we start writing codes. (4 points)

Sol:

1. Quality Assurance (QA) engineers perform tests on the software function to guarantee the stability of the product. They supervise and report the software utility to see if they reflects the quality requirements from the specification.

2. A uniform random permutation is the one in which each of the $n!$ possible permutations is equally likely to occur.

3. The Huffman tree for the first n Fibonacci numbers is a full binary tree of height n with exactly one external node in all levels, except for the first level which contains only the root and the last level which contains two external nodes, since nodes are being merged in the increasing order of their frequency. We claim this by observing that $\sum_{i=1}^{n} F_i = F_{n+2} - 1$, for all $n \geq 1$, which can be verified by induction. Based on this formulation, the internal node with frequency $\sum_{i=1}^{k} F_i$ (which is smaller than F_{k+2}) and the external node representing c_k with frequency F_{k+1} are the two nodes with the least frequency at the k-th merging step. Therefore, the Huffman code for

c_k is 1...10, in which there are $n-k$ 1s precedes 0, for $k=2$ to n, and the code for c_1 is a bit string of length $n-1$ containing no 0 in it.

4. The main idea of using paper prototype is to keep down the cost of time spent on implementing a real prototype, so that the total producing time can be well managed. The productivity gains can be significant once the time is less consumed.

5. The cost of producing a software product consists mostly of human resources, we can roughly estimate on the number of members involved in and the time required to finish the project. The cost of marketing can be estimated based on the scale of market and channels of distribution.

6. Writing down the specifications would save lots of time to design. Communications between divisions would be more efficient, while the product content is not only recognized by software engineers, but also people from other departments.

Problem 3. Given a list of n distinct numbers (not sorted), please derive a divide-and-conquer algorithm to return the *first k smallest numbers in the list*. Your algorithm should have a running time of $O(n)$. Note that you cannot use the number selection algorithm taught in the class for k times, as the running time will be $\Theta(kn) = O(n^2)$. Sorting does not work either as it will take $O(n \log n)$. In addition to the algorithm, please write down the recurrence representing the running time, solve the recurrence, and prove your solution by induction. (15 points, 7 points for the algorithm, 3 points for the recurrence, and 5 points for the proof)

Sol: The algorithm 1 proposes a method to find the first k smallest integers using divide and conquer method. At first, we divide the list into 5 groups and find median of median. This part costs $T(\frac{n}{5})$. The second part is the partition function of quicksort. It runs in $O(n)$. Lastly, we use $newPivot$ to call the function recursively. The running time of this part is the max length between $left$ to $newPivot-1$ and $newPivot+1$ to $right$, $\max(T(\frac{3n}{10}), T(\frac{7n}{10}))$. Thus, the running time of the last part is $T(\frac{7n}{10})$. The equation 1 represents for the total running time of the algorithm 1. The similar proof could be found

Algorithm 1 QuicksortFindFirstK($list$,$left$,$right$,k)

```
1: if right > left then
2:     select median pivot between left and right using median of median
3:     newPivot ← Partition(list,left,right,pivot)
4:     if newPivot > k then
5:         QuicksortFindFirstK(list, left, newPivot-1, k )
6:     else if newPivot < k then
7:         QuicksortFindFirstK(list,newPivot+1, right, k )
```

from the course materials. Therefore, the total running time is $T(n) = \theta(n)$.

$$T(n) = T(\frac{n}{5}) + \theta(n) + T(\frac{7n}{10}) \tag{1}$$

Problem 4. Use the recursion-tree method to derive an asymptotic tight upper bound for $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$. You can assume that $T(n)$ is a constant when n is sufficiently small. Prove that your bound is correct by induction. (10 points, 5 points for the recursion-tree and 5 points for the proof)

Sol: By observing that expanding the i-th level in the recursion tree costs $(\frac{7}{8})^i n$, we claim that the cost of expanding all nodes in the recursion tree is no more than $n + \frac{7}{8}n + (\frac{7}{8})^2 n + ... = \Theta(n)$. We assume in our induction hypothesis that $T(n) \leq cn$, for some positive constant c. Then, $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n \leq c(\frac{n}{2} + \frac{n}{4} + \frac{n}{8}) + n = (\frac{7}{8}c + 1)n \leq cn$, as long as we pick $c \geq 8$. Since it is obvious that n is the minimum requirement for $T(n)$, the tight upper bound for $T(n)$ is $O(n)$.

Problem 5. For bit strings $X = x_1...x_m$, $Y = y_1...y_n$ and $Z = z_1...z_{m+n}$, we say that Z is an interleaving of X and Y if it can be obtained by interleaving the bits in X and Y in a way that maintains the left-to-right order of the bits in X and Y . For example if $X = 101$ and $Y = 01$ then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y , whereas 11010 is not. (15 points)

1. Please come up with the definition of the subproblem. Use your definition and prove that this problem has optimal substructure. (5 points)

2. Give the most efficient algorithm you can to determine if Z is an interleaving of X and Y. (7 points)

3. Analyze the time complexity of your algorithm as a function $m = |X|$ and $n = |Y|$. (3 points)

Sol: Consider $x = x_1...x_m$, $y = y_1...y_n$, and $z = z_1...z_{m+n}$. We claim that if z is the interleaving of x and y, it contains a subproblem $z' = z_1...z_{m+n-1}$ and the last bit z_{m+n} must be x_m or y_n. We will have the following substructure and build an two-dimension array D[m,n] to check whether z is the interleaving of x and y. Since all cases D[i,j] comes from one of the answers of the subproblems, this problem exists the optimal substructure property. At the end of this problem, we present a $T(n) = O(mn)$ dynamic programming algorithm 2 to solve this problem.

$$D[i,j] = \begin{cases} true, \text{ if } i = j = 0 \\ false, \text{ if } x_i \neq z_{i+j} \text{ and } y_j \neq z_{i+j} \\ D[i-1,j], \text{ if } x_i = z_{i+j} \text{ and } y_j \neq z_{i+j} \\ D[i,j-1], \text{ if } x_i \neq z_{i+j} \text{ and } y_j = z_{i+j} \\ D[i-1,j] or D[i,j-1], \text{ if } x_i = z_{i+j} \text{ and } y_j = z_{i+j} \end{cases}$$

Problem 6. Consider the problem of making change for n dollars using the fewest number of coins. Assume that each coin's value is an integer. The same coin can be used for any number of times in the change. (25 points)

1. Prove that the problem exhibits optimal substructure. (5 points)

2. Assume the following set of coins is available to you: $1, 5, 10, 50$. Prove that under this condition, the problem has the greedy property. (5 points)

3. Describe a greedy algorithm to solve the problem with the coin set in 2. (5 points)

4. Give a set of coins for which the greedy algorithm in 3 does not yield an optimal solution. Your set should include a one-dollar coin so that there is a solution to every value of n. Explain the intuition behind your choice of coins in the set. (3 points)

Algorithm 2 $isInterleaving$ = CheckInterleaving(x,y,z)

```
1: D[0, 0] ← true
2: m ← x.length
3: n ← y.length
4: for i from 1 to m do
5:     for j from 1 to n do
6:         if x_i ≠ z_{i+j} and y_i ≠ z_{i+j} then
7:             D[i, j] = false
8:         else if x_i = z_{i+j} and y_i ≠ z_{i+j} then
9:             D[i, j] = D[i − 1, j]
10:        else if x_i ≠ z_{i+j} and y_i = z_{i+j} then
11:            D[i, j] = D[i, j − 1]
12:        else if x_i = z_{i+j} and y_i = z_{i+j} then
13:            D[i, j] = D[i − 1, j]orD[i, j − 1]
14: return D[m, n]
```

5. Describe a dynamic programming algorithm which solve the problem with any coin set with k different coins, assuming that one of them is a one-dollar coin. (7 points)

Sol:

1. The coin changing problem exhibits optimal substructure in the following explanation. Consider we have a coin set $C = \{c_1, ..., c_k\}$. Suppose we want to make change for n dollars, we divide n dollars into left-part $n - m$ dollars and right-part m dollars. Therefore, the solution to $n - m$ dollars is $s_L = \{c_1, c_1, c_2\}$ and the solution to m dollars is $s_R = \{c_1, c_2, c_4\}$. We claim that the left-part of solution must be the optimal way using coin set C and the right-part of the solution must be the optimal way using coin set C so that the optimal solution to n dollars is $s = s_L + s_R$.

   Proof. By contradiction, suppose there exists a solution s'_R better than right-part solution s_R. The right-part solution could be replaced by the better solution s'_R, yielding a solution $s' = s_L + s'_R$ which is better than $s = s_L + s_R$. This contracts to our previous assumption. □

Algorithm 3 $minNum$ = GreedyCoinChange(n)

```
numOfCoins ← 0
for c from c_k to c_1 do
    if c > n then
        numOfCoins ← numOfCoins + ⌊n/c⌋
        n ← n mod c
return numOfCoins
```

Therefore, the optimal solution to coin changing problem is composed of smaller subproblems of optimal solutions.

2. It it clear that there are at most 4,1,4 coins with value 1,5,10, respectively, in an optimal solution. Observing that $4 * 1 + 1 * 5 + 4 * 10 = 49 < 50$, there's no way to not choose 50 to have an optimal solution while $n > 50$. With the same reason, there's no way to not choose 10 to have an optimal solution while $50 > n > 10$, and so on. Therefore, the optimality is guaranteed by the greedy approach, which selects the largest available coin at each step.

3. Suppose the coin set has the greedy property, the algorithm 3 is developed to solve this problem. We assume that for all $c \in C = \{c_1, ..., c_k\}$ if $i < j$ then $c_i < c_j$. The main concept of this algorithm is always select the biggest possible coin in order to make the least number of the coins.

4. Consider the list $\{1, 6, 7\}$. If $n = 12$, greedy algorithm give the solution $\{7, 1, 1, 1, 1, 1\}$. However, the optimal solution for $n = 12$ is $\{6, 6\}$, which is much less than the previous solution. This is because this coin list does not contains greedy property in the coin changing problem.

5. Here we present a DP algorithm 4 to solve the coin changing problem. $D[1..n]$ array stores the optimal solution using coin set C. The $S[1..n]$ array is the solution to the combination of the coins. For certain $D[p]$, it stores the optimal solution to number of coins for p dollars using coin set $C = \{c_1, ..., c_k\}$. The following equation give the concept of the our algorithm.

$$D[p] = \begin{cases} 0, \text{ if } p = 0 \\ \min_{i:c_i \leq p}\{1 + D[p - c_i]\}, \text{ if } p > 0 \end{cases}$$

Problem 7. This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements (x appears in A for k times, $k \geq 0$): (20 points)

- Algorithm α: We pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

- Algorithm β: The algorithm searches A for x in order, considering $A[1], A[2], ..., A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

- Algorithm γ: We uniformly and randomly permute the input array A and then run Algorithm β.

1. For each algorithm, derive the expected running time when (a) $k = 0$ (b) $k \geq 1$. (18 points, 3 points for each answer)

2. Which algorithm would you use? Please explain your answer. (2 points)

Sol:

Algorithm 4 DPCoinChange(n)

```
1: D[0] ← 0
2: S[0] ← 0
3: for p from 1 to n do
4:     for c from c_1 to c_k do
5:         if  c ≥ p and 1 + D[p − c] < D[p] then
6:             D[p] ← 1 + D[p − c]
7:             S[p] ← c
```

1.
   - Algorithm α:

     (a) Let T be the time to collect all n elements in A, and t_i be the time to collect the i-th element after $i-1$ elements have been collected. Since the probability of picking a new element given $i-1$ elements already being selected is $p_i = (n-i+1)/n$, the expected value of the geometrically distributed random variable t_i is $E(t_i) = 1/p_i = n/(n-i+1)$. Therefore, $E(T) = \sum_{i=1}^{n} E(t_i) = n\sum_{i=1}^{n} \frac{1}{n-i+1} = n \cdot H_n$, where H_n is the n-th harmonic number. Thus the expected running time is approximately $n \ln n$.

     (b) Let $x = \frac{n-k}{n}$. The expected running time for $k \geq 1$ is $1(\frac{k}{n}) + 2(\frac{n-k}{n})(\frac{k}{n}) + 3(\frac{n-k}{n})^2(\frac{k}{n}) + ... = \frac{k}{n}\sum_{i=0}^{\infty}(i+1)(\frac{n-k}{n})^i = \frac{k}{n}(\sum_{i=0}^{\infty} ix^i + \sum_{i=0}^{\infty} x^i) = \frac{k}{n}(\frac{x}{(1-x)^2} + \frac{1}{1-x}) = \frac{n}{k}$.

   - Algorithm β:

     (a) The expected running time is n when $k = 0$.

     (b) Since the search will end until the first x is read, the expected time to read x is 1. For every y in A, $y \neq x$, the probability of y has been read before the first x appears is $\frac{1}{k+1}$, and there are $n-k$ elements in A which is not equal to k, thus the expected time of reading these non-x elements is $\frac{n-k}{k+1}$. Therefore, the expected running time is $1 + \frac{n-k}{k+1} = \frac{n+1}{k+1}$.

   - Algorithm γ:

     (a) The expected running time is $n + n = 2n$.

     (b) The expected running time is $n + \frac{n+1}{k+1}$.

2. If we do not have any information about the input data, Algorithm β seems better in overall evaluation, while the expected cost of both the cases are asymptotically linear to n and will never exceed n.

Problem 8. This semester we made lots of changes in the course compared to last semester (homeworks, programming assignments, course content), and we are curious about how you feel about the current form of the course. If you have the power of changing 3 things in the course, what would these 3 things be and how would you change them? Your feedbacks are very important to the teaching team; we thank you for your valuable

opinion. :) (10 points)

Sol: Skipped.