Algorithm Design and Analysis, Fall 2011

Midterm Examination

135 points

Time: 9:10am-12:10pm (180 minutes), Friday, November 18, 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. The person who should close the bug report is the one who fixes the bug.
2. $n \log n$ is polynomially larger than n.
3. $n \log n$ is polynomially smaller than $n^{1.001}$.
4. Master theorem sometimes cannot be applied even if the recurrence is in the form of $T(n) = aT(n/b) + f(n)$.
5. In all cases, using a top-down approach when using dynamic programming to solve a problem is slower than using a bottom-up approach since the former would usually involve the overhead of recursively calls (setting up the stack and local variables).

Sol:

1. **False**. The person who discovers the bug should be the one to close the bug report.
2. **False**. $n \log n$ is not larger than n by a factor n^{ϵ}, for any $\epsilon > 0$.
3. **True**. Let $f(n) = n \log n$ and $g(n) = n^{1.001}$. We can see from $\log n = O(n^{\epsilon})$, for any $\epsilon > 0$, that $f(n)$ is polynomially smaller than $g(n)$.
4. **True**. The recurrence of the form $T(n) = 2T(n/2) + n \log n$ 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.)

5. **False**. If some subproblems in the subproblem space need not be solved at all, the top-down approach has the advantage of solving only those subproblems that are definitely required.

Problem 2. "Short answer" questions: (32 points)

1. Describe two benefits of using the paper prototype instead of the regular software prototype. (4 points)
2. What are the 3 things required in every bug report? (6 points)
3. Explain why a binary tree cannot correspond to an optimal prefix code if it is not full (no internal node has only one children). (4 points)
4. Show how you would use an adjacency matrix to represent directed graph G in Figure 5. (3 points)
5. Show how you would use adjacency lists to represent directed graph G in Figure 5. (3 points)
6. Draw a breadth-first tree of directed graph G in Figure 5. Assume that we start the breadth-first search on vertex 2. (4 points)
7. Draw a depth-first forest of directed graph G in Figure 5. Assume that we start the depth-first search on vertex 2. (4 points)
8. Following the previous question, after the depth-first search you performed on G, mark the type of each edge in the original graph G. The types of the edges include tree edges, forward edges, back edges, and cross edges. (4 points)

Sol:

1. 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.
2. The three things that are needed in every bug report are (1) steps to reproduce, (2) what you expected to see, and (3) what you saw instead.

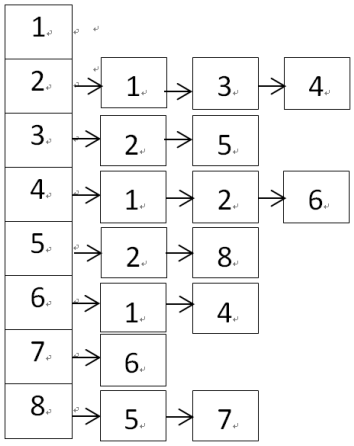

Figure 1: Adjacency lists

3. Let T be a binary tree which is not full. Then there must exist an internal node u who has only one child v. We can merge u and v into a single node. Therefore, this new tree has depth less than T.

4.

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	1	0	1	1	0	0	0	0
3	0	1	0	0	1	0	0	0
4	1	1	0	0	0	1	0	0
5	0	1	0	0	0	0	0	1
6	1	0	0	1	0	0	0	0
7	0	0	0	0	0	1	0	0
8	0	0	0	0	1	0	1	0

5. See Figure 1.

6. See Figure 2.

7. See Figure 3.

8. See Figure 4.

Problem 3. Determine the cost and the structure of an optimal binary search tree for a set of $n = 7$ distinct keys with the probabilities in Table 1.

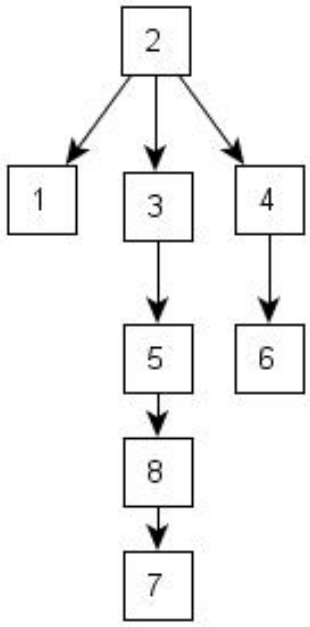

Figure 2: BFS

Table 1: Keys and Their Probabilities

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

If you do not remember what p_i and q_i represents, here are some hints. You are given a sequence $K = \langle k_1, k_2, ..., k_7 \rangle$ of 7 distinct keys in sorted order (so that $k_1 < k_2 < ... < k_7$). For each key k_i, we have a probability p_i that the search will be k_i. Some searches may be for values not in K, and so we also have $7+1$ "dummy keys" $d_0, d_1, ..., d_7$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, ..., 6$ the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search will correspond to d_i. (10 points)

Sol: Use the recursive function told in the class(In the dp3.pdf). The results are shown in the following two table. (Table 2 and Table 3) And the optimal binary search tree is in the Figure 6.

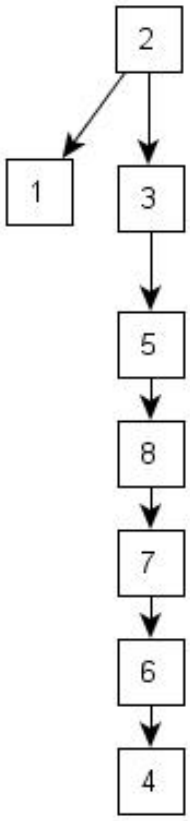

Figure 3: DFS

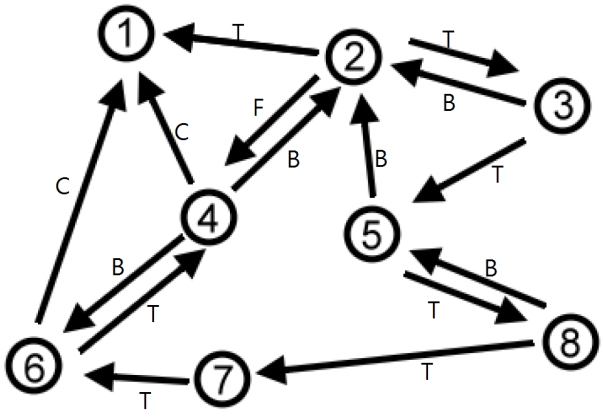

Figure 4: 2-8

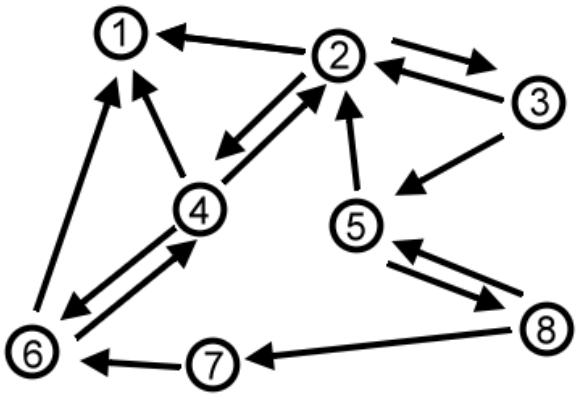

Figure 5: Directed Graph G

	0	1	2	3	4	5	6	7
1	0.06	0.16	0.28	0.42	0.49	0.64	0.81	1.00
2		0.06	0.18	0.32	0.39	0.54	0.71	0.9
3			0.06	0.2	0.27	0.42	0.59	0.78
4				0.06	0.13	0.28	0.45	0.64
5					0.05	0.2	0.37	0.56
6						0.05	0.22	0.41
7							0.05	0.24
8								0.05

Table 2: The table the recursive result of w[i,j]. The column index is the index for dummy keys(q_j), and the row index is for p_i

	0	1	2	3	4	5	6	7
1	0.06	0.28	0.62	1.02	1.34	1.83	1.42	3.12
2		0.06	0.30	0.68	0.93	1.41	1.96	2.61
3			0.06	0.32	0.57	1.04	1.48	2.13
4				0.06	0.24	0.57	0.99	1.55
5					0.05	0.30	0.72	1.20
6						0.05	0.32	0.78
7							0.05	0.34
8								0.05

Table 3: The table the recursive result of e[i,j]. The column index is the index for dummy keys(q_j), and the row index is for p_i

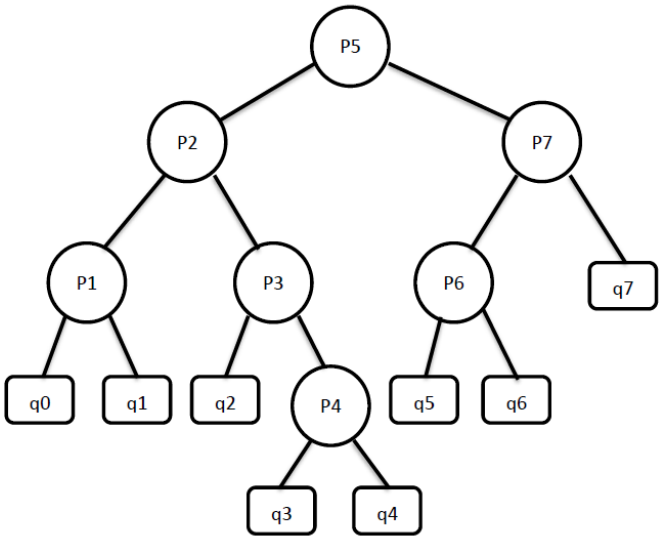

Figure 6: Optimal binary search tree.

Problem 4. Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer. (10 points, 5 points for the recursion tree and 5 points for the substitution method)

Sol: Suppose that $T(i) \leq cn^2$, for all $i < n$. Then

$$\begin{aligned} T(n) &= T(n/2) + n^2 \\ &\leq c(n/2)^2 + n^2 \\ &= (\frac{c}{4} + 1)n^2 \\ &\leq cn^2, \end{aligned}$$

as long as $c \geq 4/3$. Hence, $T(n) = O(n^2)$.

Problem 5. Given an unweighted graph $G = (V, E)$, we need to find the shortest path from u to v, where $u, v \in V$. Show that the problem of finding the shortest path from u to v exhibits the optimal substructure property. (Hint: prove that the shortest path between u and v contains the shortest paths from u to another vertex w and from w to v) (5 points)

Sol: If we find the shortest path of u to v denoted as P. Then we assume one vertex in the path P denoted as w. The path between u and w in the P is P_{uw}. And the path between w to v in the P denoted as P_{wv}. We want to prove that the $P_u w$ in the shortest path P is shortest path between u and w. If the P_{uw} is not the shortest path from u and w. Then we can find another path $P_{uw}{}^*$shorter than P_{uw}. The path $P_{uw}{}^* + P_{wv} < P_{uw} + P_{wv}$, and it is contradiction. So the path from u to w in the P is a shortest path too. Then this problem has the optimal substructure.

Problem 6. Suppose you are given an array A[1..n] of sorted integers that has been circularly shifted k positions to the right. For example, [35, 42, 5, 15, 27, 29] is a sorted array that has been circularly shifted $k = 2$ positions, while [27, 29, 35, 42, 5, 15] has been shifted $k = 4$ positions. We can obviously find the largest element in A in $O(n)$ time. Describe an $O(\log n)$ algorithm based on the divide-and-conquer strategy to find the largest element in A. (You need to show that your algorithm's time complexity is $O(\log n)$) (10 points)

Sol: When the input array contains identical numbers, no algorithm can solve this problem in the $O(\log n)$. So we assume the numbers of input array are all distinct.

1. Divide the input array into two part A_L and A_R.
2. Compare the first number of A_L and A_R. If $A_L[1] > A_R[1]$, then we know the biggest number is in the A_L, and vice versa.
3. Then we divide A_L into two part, recursively find the biggest number until the base case.(only one number in the array)

We divide the array into two part each iteration. So we have $\log n$ iteration, and one comparision per iteration. The time complexity is $O(\log n)$.

Problem 7. There is a river which flows horizontally through a country. There are N cities on the north side of the river and N cities on the south side of the river. The X coordinates of the N cities on the north side of the river are $n_1, n_2, ..., n_N$, and the X coordinates of the N cities on the south side of the river are $s_1, s_2, ..., s_N$. Assume that we can only build bridges between cities with the same number; that is, we can only build bridges between cities with coordinates n_i and s_i, where $1 \leq i \leq N$. In this problem, we ask you to determine the maximum number of bridges we can build without any bridges crossing with each other. Note that n_1 through n_N and s_1 through s_N are both not sorted.

1. Describe your definition of a subproblem. Use that definition, prove that this problem exhibits optimal substructure. (5 points)
2. Describe a dynamic-programming algorithm to solve the problem. (10 points)
3. What is the time complexity of your algorithm? (3 points)

Sol:

1. First we sort the coordinates of the N cities on the north side in non-decreasing order. Assume that $a_1, a_2, \ldots, a_n$ is the sorted sequence and b_j is the coordinate of the city at the south side which corresponds to the city located at a_j in the north, that is, for each $1 \leq i \leq N$, if $a_i = n_j$ then $b_i = s_j$, where $1 \leq j \leq N$ and

$a_1 \leq a_2 \leq \cdots \leq a_n$. In this case, the problem of finding the maximum number of bridges we can build without any bridges crossing with each other is equivalent to finding the longest increasing subsequence of $\{b_1, b_2, \ldots, b_N\}$, since if there are k bridges can be built, there exists an increasing subsequence of $b_1, \ldots, b_N$ of length k, and the sequence formed by the corresponding a_i of that increasing subsequence is also non-decreasing by definition.

Let $L(k)$ denote the length of the longest increasing subsequence of $b_1, \ldots, b_k$. Since the increasing subsequence can be extended from the subsequence that ends with b_i if $b_k > b_i$, for $i = 1, 2, ..., k-1$, the recurrence relation for the optimal solution is

$$L(k) = \max_{1 \leq i \leq k-1} \{L(i) + 1 \mid b_i < b_k\}.$$

2. Algorithm 1 is a dynamic programming algorithm for solving the longest increasing subsequence, which returns the length of the longest increasing subsequence of the input string.

Algorithm 1 LIS($A[1..n]$)

```
1: for i = 1 to n do
2:     L[i] = 1
3: for i = 1 to n do
4:     for j = 1 to i − 1 do
5:         if A[i] > A[j] then
6:             L[i] = max{L[i], L[j] + 1}
7: LIS_length = 0
8: for i = 1 to n do
9:     if L[i] > LIS_length then
10:        LIS_length = L[i]
11: return LIS_length
```

3. It takes $O(N \log N)$ for sorting and $O(N^2)$ for computing the longest increasing subsequence. Therefore, the time complexity of the algorithm is $O(N^2)$.

Problem 8. Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the **cache** - a small but faster memory - overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, ..., r_n \rangle$ of n memory requests, where each request is for a particular data element. For example, a program that accesses 4 distinct elements $\{a, b, c, d\}$ might make the sequence of requests $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Let k be the size of the cache. When the cache contains k elements and the program requests the $(k+1)$st element, the system must decide, for this and each subsequent request, which k elements to keep in the cache. More precisely, for each request r_i, the cache-management algorithm checks whether element r_i is already in the cache. If it is, then we have a **cache hit**; otherwise, we have a **cache miss**. Upon a cache miss, the system retrieves r_i from the main memory, and the cache-management algorithm must decide whether to keep r_i in the cache. If it decides to keep r_i and the cache already holds k elements, then it must evict one element to make room for r_i . The cache-management algorithm evicts data with the goal of minimizing the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of n requests and the cache size k, and we wish to minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called **furthest-in-future**, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

1. Write pseudocode for a cache manager that uses the **furthest-in-future** strategy. The input should be a sequence $\langle r_1, r_2, ..., r_n \rangle$ of requests and a cache size k, and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm? (10 points)

2. Show that the off-line caching problem exhibits optimal substructure. (5 points)

3. Prove that furthest-in-future produces the minimum possible number of cache misses (prove that this greedy choice is correct). (5 points)

Sol:

1. Algorithm 2 is the pseudocode for implementing the furthest-in-future strategy. The running time of the algorithm is $O(n^2k)$.

Algorithm 2 FURTHEST-IN-FUTURE($\langle r_1, ..., r_n \rangle$)

```
1:  for i = 1 to n do
2:      if r_i is in the cache then
3:          "cache hit," do nothing
4:      else
5:          if cache is not full then
6:              put r_i in cache
7:          else
8:              "cache miss"
9:              max_dist = 0
10:             furthest = cache[j]
11:             for j = 1 to k do
12:                 l = i + 1
13:                 while r_i ≠ cache[j] do
14:                     l = l + 1
15:                 dist[j] = l − i
16:                 if dist[j] > max_dist then
17:                     furthest = cache[j]
18:                     max_dist = dist[j]
19:             evict furthest and put r_i in cache
```

2. Suppose that $C(n, k)$ is the minimum number of cache misses of an optimal solution upon request sequence $\langle r_1, ..., r_n \rangle$. Then, $C(n-1, k)$ would be the minimum number if cache misses upon request sequence $\langle r_1, ..., r_{n-1} \rangle$. Otherwise if there exists an

algorithm which involves fewer cache misses upon request sequence $\langle r_1, ..., r_{n-1} \rangle$ than A^*, which is an optimal algorithm, there must exist an algorithm that involes fewer cache misses upon $\langle r_1, ..., r_n \rangle$ than A^*. This contradicts to the fact that $C(n, k)$ is the minimum. Therefore the off-line caching problem exhibits optimal substructure.

3. Let A^* denote the furthest-in-future algorithm. For any other algorithm A, we shall show that the number of cache misses produced by A is less than or equal to the number of cache misses produced by A^*.

   Let request r_i be the request which triggers A and A' to behave differently at the first time, where A' is the algorithm that choose the item in the cache whose next appearance in the request sequence comes furthest in the future upon r_i. More specifically, suppose that A evicts a and A' evicts a' while a cache miss occurs upon r_i, $a \neq a'$. Design A' to behave the same as A afterwards (i.e., a with respect to A' as a' is to A, when evicting), until a is requested. Then, since a is in the cache on the machine executing A' but not on the one executing A, the number of cache misses will increase by 1 only for A. Suppose that b is evicted upon a, by regarding b for A' as a' for A, A' can be designed to behave the same as A until a' is requested. After then, A and A' are identical. By repeating the above substitution, we will have an algorithm $A' = A^*$ in the end. Therefore the number of cache misses of A^* will be less than or equal to that of A.

Problem 9. I have the tradition of letting the students write some feedbacks about the course in the exam and I would like to continue this tradition (though you have just given me some feedbacks in homework 2, but hey, 10 exam points for free!): please write down 3 things you like about this course and 3 things that you would like to see some changes (and your suggestion about how we should change them). (10 points)

Sol: Skipped.