CSC 373 Midterm, Winter 2009

Name: _____Model Student_________

Student Id: _____123456789________

Advice: The midterm has three questions. We expect you to spend as many minutes on a problem as it is worth points. You should read through the whole midterm, and start with the problem you find easiest. You will get more credit for doing one problem well then for doing both poorly. If you cannot answer a question, you will receive 10% of the marks for that question if you leave it blank. Good luck.

Problem 1 (15 points): A server has n customers waiting to be served. The service time required by each customer is known in advance: it is t(i) minutes for customer i. The server must serve each customer one at a time, and once he starts serving a customer he must completely finish serving him before starting to serve the next one.

The server wishes to serve the customers in an order that minimizes the total waiting time:

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)$$

His idea is to serve the customers in increasing order of t(i). We can assume that the customers are pre-sorted in this order.

(a, 5 points) Explain why the total waiting time can be written as $T = \sum_{i=1}^{n} (n-i)t(i)$

We can view the total waiting time as the sum, over every interval when someone is being served, of the time everyone else spends waiting for them.

(b, 10 points) Give a proof that the server's algorithm is always optimal. You may assume the above formula is true, even if you weren't able to explain it.

Be very brief but concise (6 sentences can be enough). You may omit mathematical details without loosing marks, as long you give intuition as to why they are true.

Suppose that the server's algorithm is not optimal. Then there must exist an optimal schedule where some customer i is served before customer j (i<j) and t(j) < t(i). Exchange the customers with each other in the ordering. The total waiting time, according to the above formula, changes by

$$\begin{aligned} T_{new} - T_{old} &= (n-j)t(i) + (n-i)t(j) - (n-i)t(i) - (n-j)t(j) \\ &= -jt(i) - it(j) + it(i) + jt(j) \\ &= (t(i) - t(j))(i-j) \\ &\leq 0 \end{aligned}$$

The total waiting time of the new schedule is lower – a contradiction.

(In an alternate, and easier, solution, we pick the inversion such that j = i + 1. Then the waiting time changes

$$T_{new} - T_{old} = (n-i)t(j) + (n-i-1)t(i) - (n-i)t(i) - (n-i-1)t(j) = -t(i) + t(j) \leq 0$$
)

Problem 2 (15 points): You are given an array of n objects, A[1]…A[n]. You are allowed to compare whether two objects are equal (A[i] == A[j]) but there is no ordering on the objects, i.e. you cannot sort them. An element of A is called the majority if it occurs at least (n/2 + 1) times.

(a, 10 points) Design a divide & conquer algorithm that return either returns the majority element and the number of times it appears, or returns null if there is no majority element. To receive full credit your algorithm should run in time O(n* log n). You may describe your algorithm in plain English, or write pseudocode. In pseudocode, your function can return a pair of values instead of just one, if this helps you. For simplicity, you may assume that n is a power of two.

```
Majority (A, i, j )
If (i = j) return (A[i], 1)
(L_ele, L_count) = majority(A, i, (i+j-1)/2 )
(R_ele, R_count) = majority(A, (i+j+1)/2, j)
If (L_ele != null)
     For k =  ((i+j+1) / 2) to j
          If A[k] == L_ele then L_count++
     If L_count > (j-i+1)/2 return (L_ele, L_count)
If (R_ele != null)
     For k = i to (i+j-1) /2
          If A[k] == R_ele then R_count++
     If R_count > (j-i+1)/2 return (R_ele, R_count)
Return null
```

Since the overall majority must be the majority of either the left or right halves, we get these, count their occurrences in the whole array, and return if either is a majority. This requires O(N) time, plus the time for the two recursive calls.

(b, 5 points) Write the recurrence describing the running time of your algorithm.

T(n) = 2 T(n/2) + O(n), so it is O(n log n)

Problem 3 (20 points): You are given a schedule of video game tournaments, sorted by the date on which they will occur. Each tournament i has a difficulty level d[i] and a prize amount p[i]. You feel confident you can win any tournament, however the organizers do not allow anyone who has won a more difficult tournament to participate (so if you won a tournament with difficulty 2, from this point on, you can only participate in tournaments with difficulty level 2 or greater. You are asked to design a Dynamic Programming algorithm to select which tournaments you should participate in if you want to maximize your profit (you plan to win all the tournaments in which you participate).

(a, 10 points) Design a DP recurrence that will describe your expected income at the end of each tournament (whether you participate in it or not). You should explain the recurrence briefly (in english, 2-3 sentences is enough).

$$T(N) = \max \begin{cases} \forall_{k<N \text{ where } d[k] \le d[N]} \; T[k] + p[i] \\ T[k-1] \end{cases}$$

If we don't participate in the tournament, our income is the same as after the previous tournament. If we do, we pick the previous tournament k that gave us the largest profit and had a difficulty level no higher than the current one.

(b, 6 points) Using the recurrence you got above, write down (in pseudocode) an algorithm for the problem.

```
T[0] = 0;
d[0] = -inf

for (i = 1; i < n; i++)
   for (j=i; j >=0; j++) {
      if (d[i] >= d[j]) {
         T[i] = max (T[i], T[j] + p[i]);
   }
   T[i] = max(T[i], T[i-1]);

return T[n];
```

This will give the total profit; to get the schedule, just add backpointers to store which tournament followed which.

(c, 4 points) What is the running time of your algorithm?

$O(N^2)$