100 marks | **CS 301 Midsemester** | **2:30-4:30 pm, 10/9/7**

Problem 1: Consider the following algorithm fragment for finding the closest pair of n points given in the array $P[1..n]$. The algorithm is called CP, it takes an array of points and returns the closest distance.

1. Base case ...
2. Q = Sort P by x coordinate. δ_1 = CP(Q[1..n/2]), δ_2 = CP(Q[n/2+1..n]).
3. R = Sort P by y coordinate. δ_3 = CP(R[1..n/2]), δ_4 = CP(R[n/2+1..n]).
4. $\delta = \min(\delta_1, \delta_2, \delta_3, \delta_4)$.
5. ...

(a)[10 marks] Suppose the distance between the closest pair of points is less than δ calculated above. Give a precise characterization of where this pair of points must be.

(b)[10 marks] Use the above characterization to fill in the right code in place of "...". should do as few distance calculations as possible. Estimate this and justify your answer. You may assume that all coordinates are distinct. Write a recurrenc and estimate the time taken by your algorithm.

Let $x_0 = (x[n/2] + x[n/2+1])/2$ and similarly y_0. Now the only pairs of points that havent been considered are those that lie in diagonally opposite quadrants (5 marks). For such a pair to be closer than δ it must be in the square with coordinates $(x_0 \pm \delta, y_0 \pm \delta)$. 5 marks.

Algorithm: So scan over the points and find the points in this square. There must be a constant number, and hence there are only a constant number of distance calculations. 5 marks.

$T(n) = 4T(n/2) + O(n)$

$= n + 4(n/2 + 4T(n/2))$

$= O(n^2)$

3 marks for writing and 2 marks for solving the recurrence.

Problem 2: [30 marks] The input to this problem consists of two integer arrays $D[1..n]$ and $T[1..n]$ with $D[j], T[j]$ respectively denoting the deadline and time required for the jth job. A schedule is an array $S[1..m]$ (for $m = \sum_j T[j]$) where $S[i] = j$ denotes the fact that during the ith time step, job j is scheduled. Each job j must be scheduled for $T[j]$ steps, not necessarily contiguously. A job j is said to be on time in a schedule S if all its $T[j]$ steps are scheduled in the first $D[j]$ steps of S. Say a job j is *important* if $D[j] \geq T[j]$.

Consider the following greedy strategies.

1. Find an important job with the least deadline, and schedule its slots as late as possible with respect to its deadline. Suitably recurse.

2. Find an important job with the least time requirement, and schedule its slots as late as possbile with respect to its deadline. Suitably recurse.

3. Find an important job with the least time requirement, and schedule its slots at the beginning. Suitably recurse.

For each strategy, either prove that it is correct or give an instance to show that the strategy fails to produce an optimal solution. Analyze the time taken by your algorithm. An $O(n^2)$ algorithm is fine.

1. Instance: job with deadline 2 and duration 2. 3 jobs with duration 1 and deadline 3. 5 marks

3. Instance: 1 job with deadline 3 and duration 1. 1 job with deadline 2 and duration 2. 5 marks

2. Suppose jobs are renumbered as above. If $T[1] > D[1]$, then job 1 cannot be scheduled, so we move on to job 2. Assume $T[1] \leq D[1]$. Suppose S is an optimal schedule in which job 1 is (a) absent, or (b) present but not in its last position.

(a) Suppose i is a job that finishes earliest in S at some time t. We remove job i. This frees up at least T[1] slots since $T[1] \leq T[i]$ for all i. Now we reschedule jobs in the interval [1..t] so that the empty slots come to the beginning. This will not upset the deadlines of any jobs since they were finishing after t anyway. Now job 1 can be scheduled in the empty region at the beginning. Now move job 1 to its latest position, sliding the other jobs earlier. Thus we have an optimal in which job 1 is present at its last possible position. (b) If job 1 is present, then we can simply slide it to the end, processing other jobs earlier – this will not upset any deadlines.

8 marks for greedy choice.

To recurse: reduce the deadlines of jobs with deadlines larger than D[i] by T[i]. Reduce the deadlines of jobs with deadlines between D[i]-T[i] and D[i] to D[i]-T[i]. Suppose S and T are optimal schedules for before and after. Removing job 1 from S and sliding back the later jobs we know that the new schedule S' valid for the after problem. Thus $n(S') \geq n(S) - 1$. But we can also add job 1 to S' and slide jobs ahead. Thus $n(S') + 1 \leq n(S)$. Thus $n(S') = n(S) - 1$. Thus we can recurse. 8 marks for optimal substructure.

Time taken $= O(n \log n)$ for initial sort. In the recursive part we dont need to resort but may have to do $O(n)$ work. Thus the time is $O(n^2)$. This can be reduced to $O(n \log n)$ using appropriate data structures, but was not expected. 4 marks for time taken.

Problem 3:[30 marks] Suppose you sketch a sequence of connected line segments on the screen using a mouse, say using the "freehand" feature in Xfig (or Paint or whatever favourite program you use). Your program will receive as input a sequence of n points $P[1..n]$ that get sampled as the mouse moves. You are to devise an algorithm that takes this sequence of points and an additional input m, and determines the endpoints of the $m - 1$ line segments that you likely intended to draw.

Specifically you are to find an increasing sequence of integers $i_1, i_2, \ldots, i_m$ with $i_1 = 1$ and $i_m = n$ such that the following is minimized:

$$\sum_{j=1}^{j=m-1} S(P[i_j..i_{j+1}])$$

where S is a procedure that returns an integer that is a measure of the straightness of the given set of points. The exact definition of straightness is not important. For example, the straightness of the sequence $P[j..k]$ could be defined as the (negative of the) sum of the distance of each point in the sequence to the straight line joining $P[j]$ and $P[k]$. You may assume that S implements some appropriate definition and runs in time proportional to the length of the sequence passed to it. Give an algorithm to solve this problem in time polynomial in n.

Let the optimal solution be $I[1..m]$, where $I[j] = i_j$ as above.

Now I[2..m] must be an optimal solution for instance $P[I[2]..n]$. If not, suppose some J[2..m] is, and has smaller total straightness. Then I[1]——J[2..m] is also a solution for P. Since P[J[2]..n]=P[I[2]..n] we must have J[2]=I[2]. Hence the new solution will have smaller straightness.

So then we know that Opt solution of P[1..n],m = 1 —— optimal solution to P[I[2]..n] for some unknown I[2].

So defining the cost of the optimal solution of instance P[i..n],m as C[i,m] we get

$$C[1, m] = \min_{j=2}^{j=n-m+2} S(P[1..j]) + C[j, m-1]$$

Or in general:

$$C[i, m] = \min_{j=i+1}^{j=n-m+2} S(P[i..j]) + C[j, m-1]$$

12 marks for a correct recurrence.

The base case is $C[i, 2] = S(P[i..n])$. The time required for this is $O(n - i)$. Thus the total time for setting the base cases is $O(n^2)$. Filling a single table entry requires $n - m$ calculations each requiring $O(n)$ time. Thus the total time for a single entry is $O(n^2)$. Finally, there are nm table entries. So the total time is $O(n^3 m)$. 13 marks.

Note that $S(P[i..j])$ can be calculated just once and saved for all i, j in time $O(n^3)$. Given this, a single table entry requires only $O(n)$ time. Thus the time can be reduced to $O(n^2 m)$ for the main algorithm; giving a total time of $O(n^3 + n^2 m) = O(n^3)$. 5 marks.

Problem 4:[20 marks] Consider the knapsack problem in which the input consists of integers $V[1..n], W[1..n]$ where $V[i], W[i]$ respectively denote the value and weight of item i, and an additional input C denotes the knapsack capacity. In the usual knapsack problem you are expected to find a subset of items such that its total weight is at most C and the total value is largest possible. Consider now, the problem of finding not just the best such solution, but the best and the second best. Give an algorithm that also runs in time $O(Cn)$.

In each entry of the table store the best as well as the second best.

Let S(i,c) be the best and second best value for items i through n and capacity c. The S(i,c).best = (S(i+1,c).best, v[i]+S(i+1,c-w[i]).best).

S(i,c).secondbest = secondbest(S(i+1,c).best, v[i]+S(i+1,c-w[i]).best, S(i+1,c).secondbest, v[i]+S(i+1,c-w[i]).secondbest).