CS570
Analysis of Algorithms
Fall 2014
Exam II

Name: _____

Student ID: _____

Email:_____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	20	
Problem 5	16	
Problem 6	12	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
2. If a description to an algorithm or a proof is required please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[/FALSE]

If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge (u, v), then in every later iteration, the flow through (u, v) is at least 1.

[TRUE/]

For the recursion T(n) = 4T(n/3) + n, the size of each subproblem at depth k of the recursion tree is $n/3^{k-1}$.

[/FALSE]

For any flow network *G* and any maximum flow on *G*, there is always an edge *e* such that increasing the capacity of *e* increases the maximum flow of the network.

[/FALSE]

The asymptotic bound for the recurrence T(n) = 3T(n/9) + n is given by $\Theta(n^{1/2} \log n)$.

[/FALSE]

Any Dynamic Programming algorithm with *n* subproblems will run in *O(n)* time.

[/FALSE]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[TRUE/]

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[/FALSE]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

[TRUE/]

For a divide and conquer algorithm, it is possible that the divide step takes longer to do than the combine step.

[TRUE/]

Maximum value of an $s-t$ flow could be less than the capacity of a given $s-t$ cut in a flow network.

2) 16 pts

Recall the Bellman-Ford algorithm described in class where we computed the shortest distance from all points in the graph to t. And recall that we were able to find all shortest distance to t with only O(n) memory.

How would you extend the algorithm to compute both the shortest distance and to find the actual shortest paths from all points to t with only O(n) memory?

We need an array of size n to hold a pointer to the neighbor that gives us the shortest distance to t. Initially all pointers are set to Null. Whenever a node's distance to t is reduced, we update the pointer for that node and point it to the node that is giving us a lower distance to t. Once all shortest distances are computed, to find a path from any node v to t, one can simply follow these pointers to reach t on the shortest path.

3) 16 pts

During their studies, 7 friends (Alice, Bob, Carl, Dan, Emily, Frank, and Geoffrey) live together in a house. They agree that each of them has to cook dinner on exactly one day of the week. However, assigning the days turns out to be a bit tricky because each of the 7 students is unavailable on some of the days. Specifically, they are unavailable on the following days (1 = Monday, 2 = Tuesday, …, 7 = Sunday):

- Alice: 2, 3, 4, 5
- Bob: 1, 2, 4, 7
- Carl: 3, 4, 6, 7
- Dan: 1, 2, 3, 5, 6
- Emily: 1, 3, 4, 5, 7
- Frank: 1, 2, 3, 5, 6
- Geoffrey: 1, 2, 5, 6

Transform the above problem into a maximum flow problem and draw the resulting flow network. If a solution exists, the flow network should indicate who will cook on each day; otherwise it must show that a feasible solution does not exist

Solution:
I will use the initials of each person's name to refer to them in this solution.

Construct a graph G = (V, E). V consists of 1 node for each person (let us denote this set by P = {A, B, C, D, E, F, G}), 1 node for each day of the week (let's call this set D = {1, 2, 3, 4, 5, 6, 7}), a source node s, and a sink node t. Connect *s* to each node *p* in P by a directed (*s*, *p*) edge of unit capacity. Similarly, connect each node *d* in D to *t* by a directed (*d*, *t*) edge of unit capacity. Connect each node *p* in P by a directed edge of unit capacity to those nodes in D when *p* is **available** to cook. This completes our construction of the flow network. I am omitting the actual drawing of G here.

Finding a max-flow of value 7 in G translates to finding a feasible solution to the allocation of cooking days problem. Since there can be at most unit flow coming into any node *p* in P, a maximum of unit flow can leave it. Similarly, at most a flow of value 1 can flow into any node *d* in D because a maximum of unit flow can leave it. Thus, a max-flow of value 7 means that there exists a flow-carrying *s*-*p*-*d*-*t* path for each *p* and *d*. Any (*p*, *d*) edge with unit flow indicates that person *p* will cook on day *d*.

The following lists one possible max-flow of value 7 in G:
Send unit flow on each (*s*, *p*) edge and each (*d*, *t*) edge. Also send unit flow on the following (*p*, *d*) edges: (A, 6), (B, 5), (C, 1), (D, 4), (E, 2), (F, 7), (G, 3)

4) 20 pts

Suppose that there are n asteroids that are headed for earth. Asteroid i will hit the earth in time t_i and cause damage d_i unless it is shattered before hitting the earth, by a laser beam of energy e_i. Engineers at NASA have designed a powerful laser weapon for this purpose. However, the laser weapon needs to charge for a duration ce before firing a beam of energy e. Can you design a dynamic programming based pseudo-polynomial time algorithm to decide on a firing schedule for the laser beam to minimize the damage to earth? Assume that the laser is initially uncharged and the quantities c, t_i, d_i, e_i are all positive integers. Analyze the running time of your algorithm. You should include a brief description/derivation of your recurrence relation. Description of recurrence relation = 8pts, Algorithm = 6pts, Run Time = 6pts

Solution 1 (Assuming that the laser retains energy between firing beams): Sort all asteroids by the time t_i. Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \cdots < t_n$. Also assume that if asteroid i is destroyed then it is done exactly at time t_i (if the laser continuously accumulates energy then the destruction order of the asteroids does not change even if i^{th} asteroid is shot down before time t_i).

Define $OPT(i, T)$ as the minimum possible damage caused to earth due to asteroids $i, i+1, \ldots, n$ if $\frac{T}{c}$ energy is left in the laser just before time t_i. We want the solution corresponding to $OPT(1, t_1)$. If $T \geq ce_i$ and the i^{th} asteroid is destroyed then $OPT(i, T) = OPT(i+1, T - ce_i + t_{i+1} - t_i)$, otherwise $OPT(i, T) = d_i + OPT(i+1, T + t_{i+1} - t_i)$. Hence,

$$OPT(i, T) = \begin{cases} d_i + OPT(i+1, T + t_{i+1} - t_i), & T < ce_i \\ \min\{d_i + OPT(i+1, T + t_{i+1} - t_i), OPT(i+1, T - ce_i + t_{i+1} - t_i)\}, & T \geq ce_i \end{cases}$$

Boundary condition: $OPT(n, T) = 0$ if $T \geq ce_n$ and $OPT(n, T) = d_n$ if $T < ce_n$, since if there is enough energy left to destroy the last asteroid then it is always beneficial to do so. Furthermore, $T \leq t_n$ since a maximum of $\frac{t_n}{c}$ energy is accumulated by the laser before the last asteroid hits the earth and $T \geq 0$ since the left over energy in the laser is always non-negative.

Algorithm:

i. Initialize $OPT(n, T)$ according to the boundary condition above for $0 \leq T \leq t_n$.
ii. For each $n - 1 \geq i \geq 1$ and $0 \leq T \leq t_n$ populate $OPT(i, T)$ according to the recurrence defined above.
iii. Trace forward through the two dimensional OPT array starting at $(1, t_1)$ to determine the firing sequence. For $1 \leq i \leq n - 1$, destroy the i^{th} asteroid with $\frac{T}{c}$ energy left if and only if $OPT(i, T) = OPT(i+1, T - ce_i + t_{i+1} - t_i)$. Destroy the n^{th} asteroid only if $T \geq ce_n$.

Complexity: Step (i) initializes t_n values each taking constant time. Step (ii) computes $(n-1)t_n$ values, each taking one invocation of the recurrence and hence is done in $O(nt_n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Initial sorting takes $O(n \log n)$ time. Thus overall complexity is $O(nt_n + n \log n)$.

Solution 2 (Assuming that the laser does not retain energy left over after firing and if asteroid i if destroyed then it is done exactly at time t_i): Sort all asteroids by the time t_i. Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \cdots < t_n$. In contrast to Solution 1, the destroying sequence will change if we are free to destroy asteroid i before time t_i (this case is not solved here).

Define $OPT(i)$ to be the minimum possible damage caused to earth due to the first i asteroids. We want the solution corresponding to $OPT(n)$. If the i^{th} asteroid is not destroyed either by choice or because $t_i < ce_i$ then $OPT(i) = d_i + OPT(i-1)$. On the other hand, if the i^{th} asteroid is destroyed ($t_i \geq ce_i$ is necessary) then none of the asteroids arriving between times $t_i - ce_i$ and t_i (both exclusive) can be destroyed. Letting $p[i]$ denote the largest positive integer such that $t_{p[i]} \leq t_i - ce_i$ (and $p[i] = 0$ if no such integer exists), we have $OPT(i) = OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j$ if $p[i] \leq i-2$ and $OPT(i) = OPT(i-1)$ if $p[i] = i-1$. Hence for $i \geq 1$,

$$OPT(i) = \begin{cases} OPT(i-1), & t_i \geq ce_i \text{ and } p[i] = i-1 \\ d_i + OPT(i-1), & t_i < ce_i \\ \min\left\{ d_i + OPT(i-1), OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j \right\}, & t_i \geq ce_i \text{ and } p[i] \leq i-2 \end{cases}$$

Boundary condition: $OPT(0) = 0$ since no asteroids means no damage.

Algorithm:

i. Form the array p element-wise. This is done as follows.
   a. Set $p[1] = 0$.
   b. For $2 \leq i \leq n$, binary search for $t_i - ce_i$ in the sorted array $t_1 < t_2 < \cdots < t_n$. If $t_i - ce_i < t_1$ then set $p[i] = 0$ else record $p[i]$ as the index such that $t_{p[i]} \leq t_i - ce_i < t_{p[i]+1}$.

ii. Form array D to store cumulative sum of damages. Set $D[0] = 0$ and $D[j] = D[j-1] + d_j$ for $1 \leq j \leq n$.

iii. Set $OPT(0) = 0$ and for $1 \leq i \leq n$ populate $OPT(i)$ according to the recurrence defined above, computing $\sum_{j=p[i]+1}^{i-1} d_j$ as $D[i-1] - D[p[i]]$.

iv. Trace back through the one dimensional OPT array starting at $OPT(n)$ to determine the firing sequence. Destroy the first asteroid if and only if $OPT(1) = 0$. For $2 \leq i \leq n$, destroy the i^{th} asteroid if and only if either $OPT(i) = OPT(i-1)$ with $p[i] = i-1$ or $OPT(i) = OPT(p[i]) + D[i-1] - D[p[i]]$ with $p[i] \leq i-2$.

Complexity: initial sorting takes $O(n \log n)$. Construction of array p takes $O(\log n)$ time for each index and hence a total of $O(n \log n)$ time. Forming array D is done in $O(n)$ time. Using array D, $OPT(i)$ can be populated in constant time for each $1 \leq i \leq n$ and hence step (iii) takes $O(n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Therefore, overall complexity is $O(n \log n)$.

5) 16 pts

Consider a two-dimensional array A[1:n,1:n] of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

a. One way to do this is to call binary search on each row (alternately, on each column). What is the running time of this approach? [2 pts]

b. Design another divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm. Your algorithm should take strictly less than O(n^2) time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.

a) O($n\log n$).

b) Look at the middle element of the full matrix. Based on this, you can either eliminate A[$1..\frac{n}{2}, 1..\frac{n}{2}$] or A[$\frac{n}{2}..n, \frac{n}{2}..n$]. If x is less than middle element then you can eliminate A[$\frac{n}{2}..n, \frac{n}{2}..n$]. If x Is greater middle element then you can eliminate A[$1..\frac{n}{2}, 1..\frac{n}{2}$]. You can then recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$ matrices. The total runtime is T(n) = 3T($\frac{n}{2}$)+O(1), T(n) = O($n^{\log_2 3}$).

6) 12 pts

Consider a divide-and-conquer algorithm that splits the problem of size n into 4 sub-problems of size n/2. Assume that the divide step takes O(n^2) to run and the combine step takes O(n^2 log n) to run on problem of size n. Use any method that you know of to come up with an upper bound (as tight as possible) on the cost of this algorithm.

Solution: Use the generalized case 2 of Master's Theorem. For $T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^{\log_b a} \log^k n\right)$ with $k \geq 0$, we have $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.

The divide and combine steps together take $O(n^2 \log n)$ time and the worst case is that they actually take $\Theta(n^2 \log n)$ time. Hence the recurrence for the given algorithm is $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ in the worst case. Comparing with the generalized case, $a = 4, b = 2, k = 1$ and so $T(n) = \Theta(n^2 \log^2 n)$. Since this expression for $T(n)$ is the worst case running time, an upper bound on the running time is $O(n^2 \log^2 n)$.

Additional Space