MATH 416, PROBLEM SET 6

Comments about homework.

- Solutions to homework should be written clearly, with justification, in complete sentences. Your solution should resemble something you'd write to teach another student in the class how to solve the problem.
- You are encouraged to work with other 416 students on the homework, but solutions must be written independently. Include a list of your collaborators at the top of your homework.
- You should submit your homework on Gradescope, indicating to Gradescope where the various pieces of your solutions are. The easiest (and recommended) way to do this is to start a new page for each problem.
- Attempting and struggling with problems is **critical** to learning mathematics. Do not search for published solutions to problems. I don't have to tell you that doing so constitutes academic dishonesty; it's also a terrible way to get better at math.

  If you get stuck, ask someone else for a hint. Better yet, go for a walk.

Warning. These are not necessarily *model solutions*; they are meant to help you understand the problems you didn't totally solve and maybe to give you alternative solutions. Sometimes I will give less or more detail here than I would expect from you.

Problem 1. A small business (say, a small photocopying service with a single large machine) faces the following scheduling problem. Each morning they receive a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer k's job will take time t_k to complete. Given a schedule (i.e., an ordering of the jobs), let C_k denote the finish time of job k. For example, if job j is the first to be done, we would have $C_j = t_j$. Each job k also has a given weight w_k that represents the job's importance to the business. So the company decides to order the jobs in order to minimize the weighted sum of the completion times, $\sum_{k=1}^{n} w_k C_k$.

Design an efficient algorithm to solve this problem. That is, you are given a list of n jobs, the k^{th} with processing time t_k and a weight w_k. Order the jobs so as to minimize the weighted sum of the completion times, $\sum_{k=1}^{n} w_k C_k$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$; the second takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

Solution. The optimal algorithm is to schedule jobs in decreasing order of w_i/t_i. We prove the correctness of this algorithm by an exchange argument.

Consider any schedule other than the one produced by this greedy algorithm. As is standard in exchange arguments, we observe that this schedule must contain an *inversion*, a pair of jobs i, j for which i comes before j in the alternative solution, and j comes before i in the greedy solution. Arguing as we did in the interval-scheduling exchange argument, we see that there must be such a pair i, j of jobs that are adjacent in the alternative schedule. By the definition of the greedy schedule, we must have $w_j/t_j \geq w_i/t_i$. If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can repeatedly do this until no inversions remain, arriving at the greedy schedule without having increased the function we're trying to minimize. It will then follow that the greedy solution is optimal.

So consider the effect of swapping i and j. The completion times of all other jobs remain the same. Suppose the completion time of the job before i and j is c. Then, before the swap, the contribution to the sum was

$$w_i(c + t_i) + w_j(c + t_i + t_j), \tag{0.1}$$

while after the swap it is

$$w_j(c + t_j) + w_i(c + t_i + t_j). \tag{0.2}$$

The difference between quantity (**??**) and quantity (**??**) is (after canceling) $w_i t_j - w_j t_i$. This is ≤ 0 since $w_j/t_j \geq w_i/t_i$, so we have shown that the swap does not increase the total weighted sum of the completion times, as desired. □

Problem 2. A sequence is *palindromic* if it is the same whether read left-to-right or right-to-left. For instance, the sequence

A C G T G T C A A A A T C G

has many palindromic subsequences, including ACGCA and AAAA (but not ACT). (Notice that subsequences are not required to be contiguous.) Devise an algorithm that takes as input a sequence $x[1 \dots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Solution. The subproblems to consider are contiguous subsequences of $x[1 \dots n]$, i.e., $x[i \dots j]$. Let $P(i, j)$ be the length of the longest palindromic subsequence of $x[i \dots j]$. We want $P(1, n)$.

To find the key relation between subproblems, we consider whether $a_i = a_j$. If $x_i = x_j$, then x_i and x_j are part of a longest palindromic subsequence of $x[i \dots j]$ obtained by prepending x_i and appending x_j to a longest palindromic subsequence of $x[i + 1 \dots j - 1]$. In this case, $P(i, j) = 2 + P(i + 1, j - 1)$. On the other hand, if $x_i \neq x_j$, then we can use at most one in the longest palindromic subsequence, so $P(i, j) = \max(P(i + 1, j), P(i, j - 1))$. In summary:

$$P(i,j) = \begin{cases} 1 & \text{if } i = j \\ 2 + P(i+1, j-1) & \text{if } x_i = x_j \\ \max(P(i+1,j), P(i,j-1)) & \text{if } x_i \neq x_j. \end{cases} \tag{0.3}$$

As in the matrix-multiplication example, we fill a table by starting with the diagonal entries and working outward, using (**??**):

Algorithm 1: Palindromic subsequences

```
P will be an n × n table
foreach i = 1 to n do
    P[i, i] = 1
// now solve problems of size 2, then of size 3, etc.
foreach l = 1 to n − 1 do
    foreach i = 1 to n − l do
        j = i + l
        if x_i = x_j then
            P[i, j] = 2 + P[i + 1, j − 1]
        else
            P[i, j] = max(P[i + 1, j], P[i, j − 1])
return P[1, n]
```

(The algorithm solves n^2 subproblems that each require $O(1)$ work, so the total running time is $O(n^2)$.)

Problem 3. A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several ways:

BANANAANANAS BANANAANANAS BANANAANANAS.

(a) Given three strings $A[1 \dots m]$, $B[1 \dots n]$, $C[1 \dots m+n]$, describe an efficient algorithm to determine whether C is a shuffle of A and B. How fast does it run?

(b) A **smooth** shuffle of X and Y is a shuffle of X and Y that never uses more than two consecutive symbols of either string. For example, the shuffling BANANAANANAS. Describe an efficient algorithm to decide, given three strings X, Y, and Z, whether Z is a smooth shuffle of X and Y. How fast does it run?

Solution. (a) We have the following backtracking relation: if $C[1 \dots i+j]$ is a shuffle of $A[1 \dots i]$, $B[1 \dots j]$ then EITHER

(1) ($C[1 \dots i+j-1]$ is a shuffle of $A[1 \dots i-1]$, $B[1 \dots j]$ AND $A[i] = C[i+j]$)
(2) OR ($C[1 \dots i+j-1]$ is a shuffle of $A[1 \dots i]$, $B[1 \dots j-1]$ AND $B[j] = C[i+j]$).

So in words, the algorithm can be described as follows. We would like to fill in a table with the information of whether $A[i], B[j]$ is a shuffle of $C[i+j]$. It takes $O(1)$ to fill in each entry given the answers to the predecessors $(i-1, j)$ and $(i, j-1)$. So in total the algorithm takes $O(m \cdot n)$. The final entry answers the question asked by the algorithm.

(b) We have the following backtracking relation. If $C[1 \dots i+j]$ is a smooth shuffle of $A[1 \dots i]$, $B[1 \dots j]$ then EITHER

(1) $C[i+j] = A[i]$, $C[1 \dots i+j-1]$ is a smooth shuffle of $A[i-1]$, $B[j]$,
(2) OR $C[i+j] = B[i]$, $C[1 \dots i+j-1]$ is a smooth shuffle of $A[i]$, $B[j-1]$.

There is still something to be checked though, that we don't have too many As in a row (!).

We execute the algorithm as follows. We create a grid where the (i, j)th entry keeps track of whether or not $C[1 \dots i+j]$ is a smooth shuffle of $A[i]$, $B[j]$. If (i, j) is a smooth shuffle then we also keep track of the minimal number of As and Bs in a row at the end of a smooth shuffle that gives (i, j). When filling in the (i, j) entry we ask if $A[i] = C[i+j]$. If so, we need to check also that the minimal number of As in a row at entry $(i-1, j)$ is at most 1. Similarly if $B[j] = C[i+j]$. We can calculate the minimum ending As, and Bs in a smooth shuffle at (i, j). Calculating whether (i, j) is in a smooth shuffle and calculating the minimums can be done in $O(1)$ time using the neighboring minimums. The whole algorithm runs in $O(mn)$.

Problem 4. Recall the interval-scheduling problem in which we sought to minimize the maximum lateness. There are n jobs, the i^{th} with a deadline d_i and a required processing time t_i, and all jobs are available to start at time $t = 0$. Each job needs to be *scheduled*, i.e., assigned a start time s_i and finish time $f_i = s_i + t_i$, and different jobs should be assigned non-overlapping intervals. As usual, such an assignment of times will be called a *schedule*.

In this problem, we consider a problem with the same setup but a different objective. We assume that each job must be completed by its deadline or not at all. We'll say that a subset J of the jobs is *schedulable* if there is a schedule for the jobs in J so that each of them finishes by its deadline. Your task is to select a schedulable subset of maximum possible size and give a schedule your subset that allows each job (in your subset) to finish on time.

(a) Prove that there is an optimal solution J (i.e., a schedulable set of maximal size) in which the jobs in J are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines d_i and time durations t_i are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number n of jobs and the maximum deadline $D = \max_i d_i$.

Solution. (a) Let J be the optimal subset. By definition all jobs in J can be scheduled to meet their deadline. We know that the minimum lateness of J is 0, and we showed that the greedy algorithm of scheduling jobs in the order of their deadline, is optimal for minimizing maximum lateness. Hence ordering the jobs in J by the deadline generates a feasible schedule for this set of jobs.

(Or you could prove this from scratch, but you're just reproving the optimality of the greedy algorithm.)

(b) We need to assume that the jobs are given in increasing order of deadline: $d_1 \leq \cdots \leq d_n$.

The subproblems are jobs $\{1, \ldots, m\}$ with deadline d for $m \leq n$ and $d \leq D$. (These are like the subproblems in one of the knapsack problems.) For a time $0 \leq d \leq D$ and $m = 0, \ldots, n$ let $M(m, d)$ be the maximum number of jobs among $1, \ldots, m$ that can be scheduled before deadline d (even if the deadline of the job is $> d$). The crucial relation between subproblems is this: if job m is not in the optimal solution, then $M(m, d) = M(m-1, d)$, whereas if job m is in the optimal solution, then $M(m, d) = M(m-1, d-t_m) + 1$. Now the algorithm writes itself:

Its running time is $O(nD)$.

Algorithm 2: Job selection

```
Array M[0 ··· n, 0 ··· D] ;
foreach i = 0 to D do
    set M[0, d] = 0 ;
foreach m = 1 to n do
    foreach d = 0 to D do
        set M[m, d] := max(M[m − 1, d], M[m − 1, d − t_m] + 1) ;
return M[n, D]
```

Problem 5. You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (located at distance a_n), which is your destination.

You'd ideally like to travel 200 miles per day, but this may not be possible, depending on the spacing of the hotels. If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty, that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Solution. For $j = 1, 2, \ldots, n$ let $p(j)$ be the minimal penalty for a trip finishing at hotel j. The crucial relation between subproblems is this:

$$p(j) = \min_{i<j}(p(i) + (200 - (a_j - a_i))^2).$$

Algorithm 3: hotel trip

```
Array P[0 ··· n] ;
P(1) = (200 − a_1)^2 ;
foreach j = 2 to n do
    p(j) = min_{i<j}(p(i) + (200 − (a_j − a_i))^2) ;
return p(n)
```

There are n subproblems, each taking $O(n)$ work, so the running time is $O(n^2)$.