Francisco Rocha
CMPS 102
2/24/15

Homework #3 (Dynamic Programming)

1. We want to get to our transfer gate from an arriving flight as fast as possible given that we have an option of taking two walkways R and L that travel at different speeds at certain segments. Switching from one walkway to another will cost us k minutes so sometimes it is better to just stay in a slower moving segment rather than switching over. One key fact that we must observe is that at the end of the n segments we will either end up on walkway R or walkway L. Therefore we design two algorithms that returns an optimal solution for each of these two scenarios then take the fastest of the two in order to calculate the fastest travel time. With that being said the algorithm is as follows:

$$OPT_R(j) = best\ time\ for\ getting\ to\ end\ of\ j^{th}\ segement\ if\ you\ end\ on\ rigth\ walkway.$$

$$OPT_L(j) = best\ time\ for\ getting\ to\ end\ of\ j^{th}\ segement\ if\ you\ end\ on\ left\ walkway.$$

Mathematically, the identity would be:

Defines the following optimal solutions for weather we end up on the right walkway R or the left walkway L:

1st:

$$OPT_R(j) = \begin{cases} R_1 & if\ j = 1 \\ R_j + \min\big(OPT_R(j-1), k + OPT_L(j-1)\big), & o.w. \end{cases}$$

$$OPT_L(j) = \begin{cases} L_1 & if\ j = 1 \\ L_j + \min\big(OPT_L(j-1), k + OPT_R(j-1)\big), & o.w \end{cases}$$

2nd:

$$\min\ (OPT_L(n), OPT_R(n))$$

Proof of Correctness

As I stated before the only possible outcomes are that at the end of our route to the departure gate we are on the left walkway or on the right walkway. We are talking whichever preceding segments minimize our traveling time to the jth segment, this will depend on whether we waste the k amount of time switching sides rather than staying on the slower moving walkway or not. Thus we are calculating a minimizing cost for both scenarios, one ending on the right and one ending the left and then choosing the lesser of the two in order to optimize our traveling time. Notice that this is indeed possible since we are only looking at previous "optimal calculations" in two supplemental arrays in which we store the data returned from $OPT_L(j)\ and\ OPT_R(j)$, filling them up accordingly **in order**, so that by the time we calculate $OPT_L(n)$ and $OPT_R(n)$ and we can take the lesser of the two to get our desired optimal answer. Also note that if there is only one segment it will simply just choose the lesser of the two, proving our base case.
($I\ have\ ommited\ the\ j = 0\ case\ since\ I\ have\ assumed\ that\ we\ will\ aways\ be$

$travielng\ from\ one\ loaction\ to\ another\ in\ this\ scenario)$

<u>Data Structure and Runtime:</u>

As previously stated we will be filling out two n sized arrays, lets call them $L\ and\ R$ with our calculations of $OPT_L(j)\ and\ OPT_R(j)$ respectively up until we fill the nth cell of each array. From there we will take the nth element of $L\ and\ R$ and choose the least of the two to get our answer. In other words we are going to the taking the minimum of $L[n]\ and\ R[n]$. Each entry takes constant work to fill in since we are only comparing two cells of each array, therefor going though $2n$ entries giving us a runtime of $O(n)$.

2. We wish to get the schedule that gives us the minimum cost of our sightseeing expedition. A few key observations are that we can't definitively choose a limo up until we calculate how much it would cost to get to the 5th location on a taxi, then compare the two and choose the minimum. With that bit of insight stated let us define the algorithm as follows:

$$\boxed{OPT(j) = best\ price\ for\ getting\ up\ to\ the\ j^{th}\ location}$$

With this being stated we are either going to get to said j^{th} location on a taxi or on a limo and we will pay the respective price for either where if we arrive by taxi we must have taken one at the prior destination, $(i-1)$, but if we were to arrive my limo then he must have also arrived by limo at the previous 4 contiguous destinations. From this observation we get the following mathematical identity:

$$\boxed{OPT(j) = \begin{cases} r \times \sum_{i=1}^{j} L_i\,, & j < 5 \\ \min\left(r_j + OPT(j-1), b + OPT(j-5)\right), & j \geq 5 \end{cases}}$$

<u>Proof of Correctness and Runtime:</u>

We can see that for the first for trips we are only allowed to take a taxi since a limo ride must be used in 5 contiguous destinations. When we have at least 5 though, we can compute the minimum cost by comparing the previously computed entries to the possibility of us using a limo, and picking the cheaper option. With that being said it should be obvious that we are record keeping our spending in an array, let us call it Q. In other words $Q[j]$ will store the to our $OPT(j)$ sub-problem, and what it will be storing is the minimal cost of getting to the jth destination. We aren't finished yet though, this merely will give us the minimum cost of traveling around the city in a given manner but we want to find the schedule! I will proceed to

explain how we will figure out this conundrum in the following data structure analysis.

We know that as we traverse our array Q we are using only previously calculated amounts to determine the least cost up to the jth destination. In other words we are either going to be looking at position $Q[j-1]$ and $Q[j-5]$ and since our mathematical identity fills in Q from left to right ($ie. OPT(j)\ begins\ at\ 1\ and\ ends\ at\ n$) we will always have the necessary predetermined information to pick the least costly choice when we want to move from location to location and we will have the minimum cost of our sightseeing trip by the time we reach the nth recurrence of our mathematical identity. Furthermore, since are guaranteed to have a correct minimal cost solution, by filling in array S in parallel with Q, our backtracking technique on S will give us the wanted traveling schedule since we are filling out S with whether we decided to take a limo or taxi at the j^{th}location and we will be traversing our array backwards in the way described above.

Furthermore, the running time for this algorithm will be $O(n)$ since the total cost to fill out any of the arrays ($Q\ or\ S$) will take a constant amount of work due to our algorithm just each doing one comparison then recording and we will go though Q just once $O(n)$ and since we back track on S we will go through it at $O(2n)$ which results in a total runtime of $O(n)$.

<u>Data Structure:</u>

The way to go from the minimal cost solution to our wanted solution, which is the <u>schedule</u> of our trips from location to location. A very easy way to do this is to fill in an array in parallel with our Q array, which we can call S. We will denote weather we take a limo at a given point by writing a L in S and weather we took a taxi by writing a T. At the end we will back track from the last element in the array by the specifications that each of our modes of transportation abides by. So if at $S[n]$ we see a L we will take note of that L then proceed to moving to $S[n-5]$ to get the next ($or\ in\ this\ case\ precedding\ since\ we\ are\ backtracking$) item on our schedule. Similarly, if we see a T we will precede to look up $S[n-1]$ and so on and so forth until we get to the beginning S. By this mode of "backtracking" the solution array we will get the schedule that we must stick to in order to minimize traveling cost.

3. We want to maximize our fact gain at the end of our time at the museum by figuring out how much time we need to spend at each exhibit. In order to do this we must define our optimal solution to take in two parameters to effectively solve this problem. Lets us define our algorithm in the following way:

$$OPT(j,t) = the\ amount\ of\ time\ t\ to\ spend\ at\ first\ j\ exhibits\ to\ maximize\ facts\ learned.$$

Also note:

$$f_j(x) = the\ number\ of\ facts\ you\ will\ learn\ by\ spending\ x\ minutes\ at\ exhibit\ j$$

From the above definition we can construct the following mathematical identity:

$$OPT(j,t) = \begin{cases} 0, & if\ j = 0, \quad t \geq 0 \\ \sum_{i=1}^{j} f_i(0), & if\ j \geq 1, \quad t = 0 \\ \max\limits_{0 \leq k \leq t} \quad (f_j(k) + OPT(j-1, t-k)), & if\ j \geq 1, \quad t \geq 0 \end{cases}$$

<u>Proof of correctness:</u>

Again, like our previous problem, in order to show that our algorithm gives us correct answer when we calculate $OPT(n,m)\ ie. the\ total\ number\ of\ facts\ after\ spending\ \ all\ of\ out\ aloted\ time\ at\ the\ n\ exhibits$, we must show that it fills in our two dimensional solution array accurately. First notice that we are filling out the first row of our two dimensional array, lets call it S, in the very first case of our recurrence with zeros. That is when there are no exhibits to consider ($j = 0$). Then we also fill out the first column of S with our second equation listed above, when we consider the amount of information when we consider not spending any time at the exhibits. Henceforth we will be looking at the $j - 1\ and\ t - k\ for\ 0 \leq k \leq t$ cells. We can continue filling in our array row by row since we only need to access cells above or before the cell we are currently trying to fill. We finish with an "*optimal answer*" when we reach $S[n,m]$. Note that this is not our optimal solution to the given problem since we need to get the schedule of what exhibits we want to visit! We again will use the backtracking method to calculate the amount of time to spend at each exhibit.

<u>Backtracking and Running time</u>

As mentioned before we are to be filling in S an $(n+1) \times (m+1)$ table this on its own would account for a runtime of $O(nm)$. But every time that we fill in a cell we are doing up to $O(s)$ work where s is the maximum value of any given fact function $f_j(x)$ yielding an overall runtime of $O(nms)$.

We also now have the back tracking problem to get our itinerary for our visit to the museum. Like most DP problems we will figure our the actual solution in which we will fill out in parallel a table also of size $(n+1) \times (m+1)$, which we will call Q. Notice that filling in Q will take $O(nm)$ time which wont affect the over all runtime of our algorithm. Furthermore, the information that our table Q stores will be pointers to the location from of the time that maximized our fact gain beginning at location $Q[n,m]$ and working backwards. In other words, we are to be writing down the time k in the range $0 \leq k \leq t$, that maximized our $f_j(k)$ for the given exhibit j. By this back tracking notion we will know exactly how much time to spend at each exhibit by walking down the last row of our table Q since that row considers us going through all the n exhibits and using all of our allowed allotted time m.

4. Given an array of length n, we wish to design an algorithm which gives us the <u>length</u> of the longest monotonically increasing subsequence that includes the last element in the array ($ie. A[\ n\ ]$) in $O(n^2)$ time.

One of the key observations that I first made just working out this problem on paper is that you are going thought the array from left to right solving smaller subproblems in order to assist in finding the solution once you traverse the entire array. We also are checking all previous solutions in order to maximize the length of the longest monotonically increasing subsequence which includes the last element of the subarray you are currently inspecting.
With that being said algorithm is as follows:

By previous explanation let us define OPT(j) to be:

$$OPT(j) = the\ length\ of\ longest\ monotonically\ increasing\ subsequence\ if\ we\ only\ consider\ elements\ from\ 1\ to\ j, \boldsymbol{ending}\ on\ index\ j.$$

Mathematically it will satisfy the following identity:

Consider an array A of size n then the following mathematical identity will hold:

$$OPT(j) = \begin{cases} \max\big(OPT(i)\big) + 1) & \begin{matrix} where\ 1 \le i < j \\ and\ A[i] < A[j] \end{matrix} \\ 1 & otherwise \end{cases},$$

<u>Proof of Correctness and Runtime:</u>

Let us first assert that fact that our solutions of said smaller subproblems are being stored in an array also of size n and let is define it to be called $S[n]$. The base case of our definition will be when we are inspecting $OPT(1)$ in which case there will be no subproblem that satisfies the condition where $A[i] < A[j]$ therefore stated in our mathematical identity $OPT(1)$ will store 1 in S[1] and with this being proven I will continue to convince you of the fact that we are set for all further recurrences of greater sized sub problems.

The way that we have defined our "mathematical identity" to work is by using previous solved subproblems and picking out the one that when we append $A[j]$ to said monotonically increasing subsequence we will maximize its length. You can see that our algorithm looks at all previously solved sub problems $S[i]$ where $1 \le i < j$ and only considering those that meet the criteria $A[i] < A[j]$ and picks one that maximizes the length when its appended with $A[j]$. Furthermore if no such previously solved subproblem exists we assign $S[j]$ to be 1. Doing this we will have figured out all monotonically increasing subsequences considering elements up to j so by the time we get to wanting to solve $OPT(n)$ we will have all the necessary information from which to get the length of the longest monotonically increasing subsequence in which $A[n]$ is included. If all posterior subproblems are grater than

element at $A[n]$ then we again would simply assign $S[n]$=1, and that will be our solution.

Also we will meet the running time criteria since our solution array $S[n]$ is n long and the worst work it will take to fill out a cell can be up to $O(n)$ giving us a runtime of $O(n^2)$.

<u>Data Structure</u>

As I mentions above S will me our solution array and will be of size n in which we will be storing $OPT(j)$ from $1 \leq j \leq n$. At $S[n]$ we will find our solution which will be the length of the longest monotonically increasing subsequence including $A[n]$.

5. (*Help was received from tutor John Roberts when figuring answer for this question*)
We are considering an inventory problem where we wish to minimize the cost of the sum of our shipping (K) and shortage (C) while meeting all of our demands(d_i) for a said month. Our storage unit only allows a limit of S trucks per month, cost K is fixed for any number of trucks you order but C is charged for each truck you wish to store. Some insights that are imperative to observe in order to find a starting point towards the optimal solution is that at the end of n months we wish for there to be no left over trucks cause we have met all of our yearly demands thus it is wasteful to store any leftover trucks. Observing these key facts lets us at least get a starting point as to how to go about checking the minimal cost for a certain subproblems. Let us also define r to be the remainder trucks after a given month i.
Therefore our algorithm is as follows:

$$OPT(i,r) = how\ well\ can\ we\ do\ up\ to\ months\ i\ in\ order\ to\ have\ considering\ a\ remainder\ of\ r\ trucks, minimizing\ sum\ of\ costs.$$

Further notation will be needed though in order to properly solve subproblems. Let us define $\boldsymbol{p}$ to be the remainder of the first $j-1$ months.

With that defined we can now state our mathematical identity:

Consider an array A of size n then the following mathematical identity will hold:

$$OPT(j,r) = \begin{cases} \infty, & if\ p - d_j > r \\ \min\limits_{0 \leq p \leq S} (OPT(j-1,p) + Cr), & if\ \ p - d_j = r \\ \min\limits_{0 \leq p \leq S} (OPT(j-1,p) + K + Cr) & if\ p - d_j < r \end{cases}$$

Proof of Correctness

We know that we are interested in finding $OPT(n, 0)$ because the optimal solution results in us meeting all of the demands and having no remainder trucks in storage. Therefore, when solving $OPT(j, r)$, for each preceding problem we are interested in minimizing the cost up to said point by considering what $OPT(j-1, p)$ could have been for all $0 \leq p \leq S$ where p is the number of trucks left over from the pervious period and S is the inventory limit. As seem above we must consider three different cases:

1. $p - d_j > r$: Obviously this problem is impractical so we declare the cost to be infinity to not consider it when minimizing our optimal solution.
2. $p - d_j = r$: this case implies that we have enough inventory to supply the preceding demand plus the current demand therefore we don't need to order any more trucks and the cost will come only from the storage fee for the r trucks during this month.
3. $p - d_j < r$: this implies that we wont have enough trucks to satisfy the demand therefore we should order more $ie. p - d_j + r$to be able to meet the demand for $OPT(j, r)$ therefor in this case we will pay to order and to store.

Moreover it is clear to see that it is possible to fill the table since are always using previously computed sub-solutions much like problem #4.

Data Structure and Runtime

It should be obvious that we are filling in a 2 dimensional array with the dimensions $(n+1)\times(S+1)$, let us call it C. As we fill out C every cell $C[j, x] = OPT(j, r)$. Starting at the upper most left one we will assign a 0 since we are computing $OPT(0,0)$ and continuing to fill in the entire column $OPT(0, r)$ with ∞ by (1). We can continue hence forth to fill in array row by row since at any point we just need the preceding values of the current row and we shall find our minimal cost answer at $C[n, 0]$. In order to find the solution we are looking for though, which is the actual schedule of ordering times we need to create a duplicate array in which we record z in every step j of our recurrence. From this we can back track to the array and look up the places were $z = 0$ and look at the following cell and record its location and its number to know when and how much to order in order to minimize our costs.

Furthermore, since we are filling out an $(n+1)\times(S+1)$ array we will have a $O(nS)$ amount of work to do just to completely fill our tables. Each cell will require at most $O(S)$ amount of work to fill leaving us with a total runtime of $O(nS^2)$.