Introduction to Algorithms	Problem Set 3 (3 problems)
CS 4820 Fall 2019	Due 11:59pm Thursday, September 26

(1) Highway Planning. You are helping plan a new stretch of highway that is M miles long. One issue that came up is the placements of gas-stations. To help drivers not get off the road, you agreed to have a gas station at least every m miles (that is, you need to have a gas station with m miles of the start and end of the highway stretch, and two consecutive stations cannot be more than m miles apart. You also have estimates on costs of building gas-stations at various points along the high way. The costs vary due to differences in the land around the highway, and the proximity of town. The possible options (measured from the start of the highway) are at miles $x_1, \ldots, x_n$, and location x_i has cost estimate $c_i > 0$. You may assume that you have enough possibilities to satisfy the requirements, that is $x_1 \leq m$, $M - x_n \leq m$, and $x_{i+1} - x_i \leq m$ for all $i \leq n-1$. Design a $O(n \log n)$ dynamic programming algorithm for finding a set of gas stations of minimum cost satisfying the requirements, and implement the algorithm in Java. We suggest using subproblem $Opt[i]$ to be the minimum cost solution for the stretch of highway up to position x_i **with a gas-station at location** x_i. Your algorithm needs to output the total cost (sum of costs of the selected locations), as well as the list of locations selected in increasing order of distances.[1] Algorithms that find the correct solution, but run in $O(n^2)$ time will be graded out of 7 points. Algorithms that are faster than $O(n \log n)$ can earn 3 bonus points.

Solution: Using the suggested subproblems we get the following algorithm:

- Base case: $Opt(i) = c_i$ for i such that $x_i \leq m$.
- recurrence: For i such that $x_i > m$, we use the recurrence $Opt(i) = c_i + \min_{j<i:x_i-m\leq x_j} Opt(j)$
- Opt values are computed in order $i = 1, \ldots, n$
- The optimal value to return is $\min_{j:x_j \leq M-m} Opt(j)$
- to get the set of locations we can add extra field for each i and store also the optimal previous choice: For i such that $x_i \leq m$ let $Prev[i] = 0$, for all other i let $Prev[i] = argmin_{j<i:x_i-m\leq x_j} Opt(j)$. Using this extra information, we can find the optimal solution going backwards from the end: the last station is at x_j for the $j = argmin_{j:x_j \geq M-m} Opt(j)$. Then for a station already added at x_j already added, add a station at x_i for $i = Prev[j]$ unless $Prev[j] = 0$, and iterate.

Correctness (not required for this problem): by induction on i. The base case is the correct value by the definition of the subproblems. To prove that the computer value is correct for all i values, consider a value i, the solution up to mile x_i including the last station must have a previous station at a place $j < i$ such that $x_j \geq x_i - m$. By the induction hypothesis $Opt(j)$ is the min cost solution value for all such j, so the min-cost solution for i is indeed $c_i + \min_{j<i:x_i-m\leq x_j} Opt(j)$. Similarly, the returned value is correct follows from the induction hypothesis.

Running time: To implement this $O(n \log n)$ time we need to be able to find each $\min_{j<i:x_i-m\leq x_j} Opt(j)$ in $O(\log n)$ time. To do this, we can keep a priority queue of the opt values already computed. Adding a new value and extracting the min from the priority queue is $O(\log n)$ time. If the min extracted doesn't satisfy the $x_i - m \leq x_j$ condition, this station is too far by now, and we keep extracting a new min till we find one that satisfies this.

To implement this in $O(n)$ time we will keep a queue rather than a priority queue of the stations within m miles in decreasing order of cost. When computing a new $Opt[i]$ value, we take the first item on the queue. If that is more than m miles away, we delete it from the queue, and take then next one. Once the new $Opt[i]$ is computed, we need to add this to the queue. Notice that if $Opt[i] \leq Opt[j]$ for some $j < i$ than we will not use j as the selected index on any later $k > i$, as if x_j is within m miles

[1] You can assume that solution with optimum cost is unique.

than so is x_i as $j < i$. So in adding i to the queue, we can delete all entries with value $\leq Opt[i]$ and then add i at the end of the queue.

(2) Matrix Multiplication. Consider the problem of multiplying a sequence of matrices $A_1, A_2, \ldots, A_m$ that was mentioned at the beginning of class on Wednesday, September 18th. Assume matrix A_i is an $n_{i-1} \times n_i$ matrix. To multiply two matrices of size $n \times m$ and $m \times k$, the resulting matrix will be $n \times k$, and we get each entry by m multiplications and $m-1$ additions, so this requires a total of mnk multiplications and $\leq mnk$ additions. In multiplying a sequence of matrices, by associativity of multiplication, we have many orders we can do the multiplication. For example $(A_1 \cdot A_2) \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$, and doing it different order has different running times. For example, suppose A_1 is 2×4, A_2 is 4×10, and A_3 is 10×2. If we multiply $A_1 \cdot A_2$ and then multiply the result by A_3 the number of multiplications is $2 \times 4 \times 10 = 80$ for the $A_1 \cdot A_2$ and then $2 \times 10 \times 2 = 40$ for multiplying the result by A_3 for a total of 120 multiplications. In contrast, multiplying $A_2 \cdot A_3$ first is $4 \times 10 \times 2 = 80$ multiplication, and then A_1 times this product is another $2 \times 4 \times 2 = 16$ for a total of only 96 multiplications, so this is a better order. Given a sequence of m matrices, $A_1, A_2, \ldots, A_m$ (where A_i is an $n_{i-1} \times n_i$ matrix), give a polynomial time algorithm to find the order to do the multiplications that require the fewest multiplications. Your algorithm needs to output the order of multiplications, as well as the number of multiplications needed.

Solution: As subproblems use $Opt[i, j]$ for $i \leq j$ to be the number of multiplications needed to get the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

- Base case: For $j = i$ we have $Opt[i, i] = 0$.
- recurrence: For $j > i$ we have to consider the last multiplication and get that

$$Opt[i, j] = \min_{i \leq k < j} Opt[i, k] + Opt[k + 1, j] + n_{i-1} \cdot n_k \cdot n_j$$

- Opt values are computed in increasing order of $j - i$, that is, for $\ell = 0, \ldots, n - 1$ compute all $Opt[i, i + \ell]$ in this order. To help return the order and not just the value also save the k on which the min occurred as $Last[i, j]$ for all $j > i$.
- The minimal number of multiplications needed is $Opt[1, m]$
- to get order of multiplications, we expand recursively.
  - if $j = i$ then $Order[i, i] = A_i$
  - Else let $k = Last[i, j]$ then $Order[i, j] = (Order[i, k]) \cdot (Order[k + 1, j])$

Running time: There are $O(m^2)$ Opt values we want to compute, and each computation involves taking the minimum over at most m options, so running time is at most $O(m^3)$. Writing out the solution with all the matrices is longer, takes $O(\sum_{i=1}^{m} n_{i-1} n_i)$ time.

Correctness: we prove correctness by induction on $\ell = j - i$. The base case, when $j = i$, there is nothing to do, so the number of multiplication is 0 indeed. For the case $\ell \geq 1$ consider the last multiplication to get the result, so $A_i \ldots A_j = (A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$. By the induction hypothesis the optimal way to get $A_i \ldots A_k$ is $Opt[i, k]$ multiplications, and the optimal way to get $A_{k+1} \ldots A_j$ is $Opt[k + 1, j]$ multiplications, the last multiplication is for the product of these two results, multiplying a $n_{i-1} \times n_k$ matrix with an $n_k \times n_j$ matrix, which takes $n_{i-1} \cdot n_k \cdot n_j$ multiplications. So the best way to get $A_i \ldots A_j$ is to use the k that attains the minimum as the last multiplication.

Solutions that don't work:

Some students tried a **A 1-dimensional dynamic program** using $Opt[i]$ as the number of multiplications needed to get the result of $A_1 \ldots A_i$ (so always start from the beginning). We don't think these subproblems can be made to work: For example, some people tried this recurrence

$$Opt[i] = \min(Opt[i - 1] + n_0 n_{i-1} n_i, Opt[i - 2] + n_{i-2} n_{i-1} n_i + n_0 n_{i-2} n_i)$$

This offers two options as last multiplication the $(A_1 \ldots A_{i-1}) \cdot A_i$ and the one alternate of $(A_1 \ldots A_{i-2}) \cdot (A_{i-1} A_i)$, and not consider any other option for the last multiplication.

Some students tried a **Greedy algorithm**, suggesting that the first multiplication to do is the one with the middle size is the largest: so if n_i is the maximum size then start by multiplying A_iA_{i+1}. This greedy is not guaranteed to give the optimal order: For example, if the size of the matrices are $5 \cdot 10 \cdot 8 \cdot 2$. Doing the multiplication of A_1A_2 first, as suggested by the above greedy, we get $5 \cdot 10 \cdot 8 + 5 \cdot 8 \cdot 2 = 480$ multiplications, while the other option of doing A_2A_3 first gives us only $10 \cdot 8 \cdot 2 + 5 \cdot 8 \cdot 2 = 240$ multiplications.

(3) Min-cost path with negative costs. Consider a directed graph $G = (V, E)$ where the cost of edge e, c_e, can be negative, and this graph may have negative cycles. Given two nodes s and t, we have seen in class how to compute the length of the min-cost path in G assuming G has no negative cost cycles. In this problem, we will want to find low cost path, and also detect if the graph has negative cycles.

(a) *(2 points)* Give a polynomial time algorithm that computes the minimum cost path between two nodes $s, t \in V$ using at most k edges. If the graph has negative cycles, you may want to use edges more than once, which is OK as long as you don't use more than k edges.

Solution The algorithm from class works perfectly for this, use $Opt[k, v]$ as the min-cost path using at most k edges from s to v, and update as we did in class.
What we had in class used the update $Opt[i, v]$ for the min-cost path from s to v using at most i edges with the recurrence for $i \geq 0$

$$Opt[i, v] = \min(Opt[i-1, v], \min_{(u,v)\in E}(c_{uv} + Opt[i-1, u]))$$

The running time is $O(m)$ for one value of i, so a total of $O(mk)$. The correctness of this algorithm was covered in class and is in the book.
It is also OK to drop the $Opt[i-1, v]$ term from this, as long as you do one of a few options:

(i) add this option at node $v = s$, that is use the recurrence with $Opt[i-1, v]$ included for $v = s$ only. This works well as at all other nodes, even if the path has fewer edges, it still must have a last edge (u, v), so the $Opt[i, v] = \min_{(u,v)\in E}(c_{uv} + Opt[i-1, u])$ get the right value with $Opt[i-1, u)$ possibly a path of fewer edges than $i-1$.

(ii) Initialize $Opt[i, s] = 0$ for all i, that is in the update for node s also include 0 as an option, and only update it if the min is below 0.

(iii) without doing any of this $Opt[i, v]$ would be the min cost path using **exactly** i edges, and so we want to then output $\min_{i\leq k} Opt[i, t]$.

Runtime Runtime is $O(mk)$ or $O(n^2k)$. Note it could be the case that $k > n$ so the runtime $O(mn)$ (the same as Bellman Ford) does not apply.

(b) *(4 points, optional)* Give a polynomial time algorithm that either finds the minimum cost path from s to t or finds a negative cycle in the graph G.

Solution There are two interpretations of the questions

(1) if we are looking for a min-cost path (with repeat nodes allowed), then it seems best to test if the graph has a negative cycle reachable from s. Run the algorithm from part (a) till $k = n$. Recall that in a graph with no negative cycles $Opt[n, v] = Opt[n-1, v]$ should be true for all nodes v. We claim two parts

(i) if $Opt[n, v] < Opt[n-1, v]$ for any node v, than the min-cot path using n edges from s to v using at most n edges, must contain n edges, hence must contain a cycle, and this cycle must be of negative cost. You can test this in $O(mn)$ time, and return the negative cycle if you found one.

(ii) if $Opt[n, v] = Opt[n-1, v]$ for **all nodes** $v \in V$ then the graph has no negative cycles reachable from s, and the path corresponding to $Opt[n-1, t]$ is the min-cost path. This is true as you may note that once $Opt[i, v] = Opt[i+1, v]$ for all nodes v for any value i, than all further $k > i$ the Opt values will not change, that is, we will have $Opt[k, v] = Opt[i, v]$, so hence the current min-cost path is the true min-cost. Note you must check this for all nodes to account for negative cycles that require many go-arounds before becoming part of the min cost path, as in the example below.

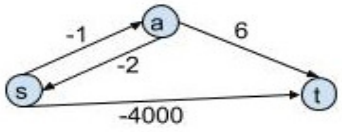

Figure 1: You would need $> n$ edges for the cycle s,a,s to be added to the min cost path from s to t

(2) if we are looking for a min-cost **simple** path (this is not that the question said, but it is consistent with some Piazza answers we gave) than you can do the following:
Find the min-cost of an s, t path using at most $n-1$ edges: $Opt[n-1, v]$. Find the path that the solution corresponds to. If the path has a cycle, check the cost of the cycle. If the cycle has negative cost, output the negative cycle. If the cycle has 0 cost, just delete the cycle from the path, which then gives us another min-cost path with fewer edges. Repeat till you either find a negative cycle or a min-cost path with no cycle in it.
If you found a negative cycle, that is a good output. If you found a simple path s to t of cost $Opt[n-1, t]$ we claim that is the min-cost **simple** path from s to t, as it is the min-cost path using at most $n-1$ edges even if we were to allow repeats, and a simple path cannot have more than $n-1$ edges.

(c) *(4 points)* To be more robust, you would like to know if the minimum cost path is unique. Modify the algorithms in (a+b) to also decide if there is only one min-cost path or are there multiple min-cost options.

Solution For part (a) we can add a extra binary value to $Unique[k, v]$, which will be 1 if the min-cost path of at most k edges is unique and 0 otherwise.
The simplest way to do this, would change the opt above to be about min-cost with **exactly** i edges (as suggested by version (iii) in part (a)), that is use the recurrence for the modifies $\tilde{Opt}$ as

$$\bar{Opt}[i, v] = \min_{(u,v) \in E} c_{u,v} + \bar{Opt}[i-1, u]$$

for all nodes $v \in V$ and all $i = 1, \ldots, k$ with the initialization of $\bar{Opt}[0, s] = 0$ and $\bar{Opt}[0, v] = +\infty$ all nodes $v \neq s$.
With this version, there are two or more the min-cost path from s to v using **exactly** i **edges** if one of two cases happens

(i) the minimum in $\min_{(u,v) \in E} c_{u,v} + \bar{Opt}[i-1, u]$ is attained on two different edges (u, v) and (u', v) entering v,

(ii) or if the minimum above is uniquely obtained on some edge (u, v), but the minimum cost path using at most $i-1$ edges from s to u is not unique.

with this version of the algorithm, we can add a new binary variable to our table $\bar{Opt}[i, v]$ called $Unique[i, v]$, which will be set as 1 if the min-cost path using exactly i edges is unique, and can update it along with filling out the table.

Finally, the min-cost path is unique, if the minimum of $\min_{i \le k} \bar{Opt}[i,t]$ is obtained on a single i, and for this value of i we have that the path for $\bar{Opt}[i,t]$ is unique.
Adding this extra variable doesn't increase the $O(.)$ running time, so its still $O(mk)$ only.

Alternate way to do this is to just compute Opt as in part (a) but only mark $Unique[k,v]$ as not unique if one of the following happens

(i) the minimum in $\min_{(u,v)\in E} c_{u,v} + Opt[i-1,u]$ is attained on two different edges (u,v) and (u',v) entering v,

(ii) the minimum in $\min_{(u,v)\in E} c_{u,v} + Opt[i-1,u]$ is attained on a unique u, and $Unique[i-1,u] = 0$

(iii) $v = s$ and the minimum is 0, and is attained both at $Opt[i-1,s]$ as well as on $c_{u,s} + Opt[i-1,u]$ an edge (u,s) entering s.

This version makes it a bit harder to argue that the value of $Unique[i,v]$ is indeed 1 if the min-cost path of at most i edges in unique. We can do this by induction. The base case is easy. If there are two or more min-cost path to u using at most i edges, these two path can differ on their last edge, this is case (i), or agree on their past edge, which is case (ii), or finally if $v = s$, one can have no edges, the other has some. This is case (iii) as this can only happen at node s.

Runtime Runtime is $O(mk)$ or $O(n^2k)$. Note it could be the case that $k > n$ so the runtime $O(mn)$ (the same as Bellman Ford) does not apply.

A solution that don't work Some students are trying to do this maintaining the uniqueness **without** switching to path using **exactly** i edges, that is $Unique[i,v]$ to be as unique if the minimum on the usual recurrence from part (a) is obtained on exactly one item. This won't work OK. Consider the example below. We will get $Opt[0,s] = 0, Opt[1,v] = 10, Opt[1,t] = 5$ and the

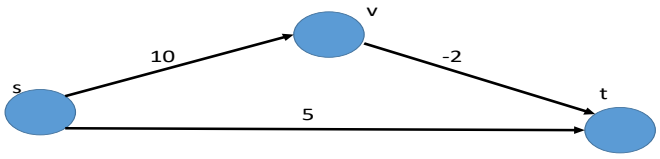

Figure 2: Last case of the exchange argument.

1 long path to v and t is unique. Then consider the path using at most 2 edges: we get

$$opt[2,t] = \min(Opt[1,t], \min(-2 + Opt[1,v], 5 + Opt[0,s]))$$

This minimum is obtained both at $Opt[1,t]$ and $5 + Opt[0,s]$, yet the min-cost path using at most 2 edges is actually unique, just the simple edge path (s,t).