Sheetal Parikh
Foundations of Algorithms
EN.605.621.81.SU19
Homework #4

1. *Give a polynomial-time algorithm that takes a sequence of supply values s1, ..., sn and returns a schedule of minimum cost. For example, suppose r = 1, c = 10, and the sequence of values is 11, 9, 9, 12, 11, 12, 12, 9, 9, 11. Then the optimal schedule would be to chose company A for the first three weeks, company B for the next block of four contiguous weeks, and then company A for the final three weeks.*

We are given that company A charges a fixed rate r per pound and costs rs_i to ship a week's supply of s_i whereas company B charges a fixed amount of c per week and a contract must be made in blocks of four consecutive weeks at a time giving a total cost of $4c$ per contract.

For an optimal schedule we would want to use company A or B during the ith week. If using company A for the ith week and taking into account the previous weeks, we would behave optimally for up to week $i-1$ and pay rs_i. Otherwise, if we use company B for the ith week and take into account the previous weeks, we would behave optimally for up to week $i-4$ and pay $4c$. The minimum cost would be the minimum value of these two costs.

The recurrence to solve would be determining the optimal cost of weeks 1 to 3. The optimal cost of the recurrence would be the minimum cost of the different possibilities each of which could be defined as an array. When determining the optimum schedule the minimum value of these arrays would be returned. There would be 4 possibilities for weeks 1 to 3.

1.) company A weeks 1 to 3
2.) company B weeks 1 to 3
3.) company A weeks 1 to 2 and company B week 3
4.) company A week 1 and company B weeks 2 to 3.

Constant time would be spent per iteration as we iterate i.

2. ***Collaborative:*** *Given a long string of letters y = y1y2...yn, a segmentation of y is a partition of its letters into contiguous blocks of letters, each block corresponding to a word in the segmentation. The total quality of a segmentation is determined by adding up the qualities of each of its blocks. (So we would need to get the right answer above provided that quality("meet")+quality("at")+quality("eight") was greater than the total quality of any other segmentation of the string.) Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. Prove the correctness of your algorithm and analyze its time complexity.*

We take an array to hold the score and run the loop from from *i* until the length of the string. We run another loop from *j* until the end of the string and calculate the highest quality score. If the score of the right hand side is greater than the initial score, we replace the new score with the initial score. We backtrack the string to find where to put a segmentation. To find the optimal solution, we are searching through the right hand side because the left hand side consists of only one word.

Let QS be the total quality score of any segment in the string $y_1 .... y_i$

$QS[i] =$

$$\begin{array}{ll} 0 & if\ i = 0 \\ max\ QS(j-1) + Q(y_1 .... y_i) & if\ i \geq 0 \end{array}$$

Function QualityScore $(y_1 .... y_i)$

```
$QS[0] = 0$
$for\ i\ in\ n\ \{$
    $var\ QS[i] = \infty$
    $for\ j\ in\ i\ \{$
        $var\ previous = QS[j-1] + Q[y_1 .... y_i]$
            $if\ QS[i] < previous$
            $QS[i] = previous$
            $Record\ [i] = j$
$Return\ QS$
```

Function BackTrack $(track)$

```
$val\ seg = emptyList()$
$var\ i = n$
$while\ (\ i > 0)$
        $var\ j = track\ [i]$
        $add\ seg\ \ (y_1 .... y_i)\ to\ beginning\ of\ segment$
  $return\ seg$
```

The algorithm will have a time complexity of $O(n^2)$.
Each operation inside the loop will take constant time. Furthermore, the function first loops from 1 until the end of the string and then loops from j until the end of the string (not including the one word from the left hand side). Therefore, the function approximately goes through the length of the string twice and the increase in the number of steps as the function goes through both loops could be described by an arithmetic series: $\frac{1}{2}\ n(n+1)$.

$\frac{1}{2}\ n(n+1)$ can simplify to $\frac{1}{2}\ n^2 + \frac{1}{2}$ which if only valuing the large ordered elements would give us a time complexity of $O(n^2)$.

3. *A greedy algorithm you might use for this is the following. Start with an empty truck and begin piling containers 1,2,3,... onto it until you get to a container that would overflow the weight limit. (These containers might not be sorted by weight.) Now declare this truck "loaded" and send it off. Then continue the process with a fresh truck. By considering trucks one at a time, this algorithm may not achieve the most efficient way to pack the full set of containers into an available collection of trucks. (*

a) *Give an example of a set of weights and a value for K where this algorithm does not use the minimum number of trucks.*

Let's say a shipment arrives with 6 containers of weight $\{w_1, w_2, w_3, w_4, w_5, w_6\} = \{1, 10, 1, 10, 1, 10\}$. The weight restriction of the trucks or the value of K in this example is 10. As per the algorithm described in the description of the problem, we first start with an empty truck and pile w_1 into the truck. We cannot pile w_2 into the truck as an additional weight of 10 would overflow the restriction. Therefore the first truck is loaded and is sent off. If we follow the algorithm, we would load 5 trucks since if we go in order of the containers, any additional container would cause us to overflow the restriction. This algorithm proves to be inefficient, because the minimum number of trucks should have been 3 since containers w_1, w_3, and w_5 could have been piled into the same truck.

(b) *Show that the number of trucks used by this algorithm is within a factor of two of the minimum possible number for any set of weights and any value of K.*

If *W* is the total weight of all trucks and as given, *K* is the maximum weight held by a truck, we can say $\frac{W}{K}$ is the minimum number of trucks needed. We want to prove that the number of trucks needed to carry the containers is within a factor of 2 of the minimum number of trucks is within 2 $\frac{W}{K}$ for any value of *W* and *K*. We saw in the example given in 3a that the average load per truck was approximately $\frac{K}{2}$. For example, the average load from the example in 3a would be $\frac{33}{60}$ which is approximately $\frac{K}{2}$. Therefore, if we show that the average load is approximately $\frac{K}{2}$ for any situation of trucks, we will be able to prove that the number of trucks needed to carry the containers is within a factor of 2 of the minimum number of trucks. Based on our example, we can see that the weight of 2 consecutive trucks will be greater than the limit *K*. Therefore, if we are loading truck *a*, this can be shown as $T_a + T_{a+1} > K$. To prove this further, if we are loading a container C_x it will be put into truck T_a . We know that loading the next container C_{x+1} into truck T_a will cause us to go over the weight restriction or $T_a + C_{x+1} > K$. Therefore, container C_{x+1} will be loaded into the next truck T_{a+1} and will be the first container in T_{a+1} or $T_{a+1} \geq C_{x+1}$. Because the average of two consecutive trucks will also be approximately $\frac{T_a+T_{a+1}}{2}$ and we know that $T_a + T_{a+1} > K$, we can say that $\frac{T_a+T_{a+1}}{2} \geq \frac{K}{2}$. As shown by the example in 3a, the equation can be applied towards an entire set of containers and trucks. If the average of two consecutive trucks is $\frac{K}{2}$ we can say that the average of the entire set of trucks used is also $\frac{K}{2}$. Therefore, because the average load of trucks is approximately $\frac{K}{2}$, we can say that the number of trucks needed to carry the containers is within a factor of 2 of the minimum number of trucks.

4. ***Collaborative:***

(a) *Prove that the expected value represented by the counter after n Increment operations have been performed is exactly n.*

We need to find the expected value of the counter value. In general the expected value $E[X_j]$ is the weighted average of the possible values of X or in other words is the sum of the possible values of X multiplied but its probability. The possible counter values are 0, when the counter stays the same, and 1, when the counter is incremented. We have already been given the probabilities of both counter values.

Using the information we have above, we will find the expected value $E[X_j]$ of one increment operation:

$$E[X_j] = 1 * \left[(n_{i+1} - n_i) * \frac{1}{n_{i+1} - n_i}\right] + 0 * \left[1 - \frac{1}{n_{i+1} - n_i}\right] = 1$$

As we are given that there will be *n* increment operations, the expected value represented by the counter after *n* Increment operations will be *1*n operations = n.*

(b) *The analysis of the variance of the count represented by the counter depends on the sequence of the ni Let us consider a simple case: ni = 100i for all i ≥ 0. Determine the variance in the value represented by the register after n Increment operations have been performed.*

We need to analyze the variance of the count of a sequence of n_i. Generally the variance is a measure of how spread the numbers are and is the average of the squared differences from the mean. The equation for the variance is $V[X] = E[X^2] - u^2$ where *u* is the expected value, $E[X] = u$. Therefore the variance equation we will use is: $V[X_j] = E[X_j^2] - E[X_j]^2$. We are given that the simple case we are considering is $n_i = 100i$ for all $i \geq 0$. The n_i represents the increasing sequence of nonnegative values. The equation $n_i = 100i$ can be used to find the probability of increasing the counter by 1 and the probability of no increase when the counter is 0.

We will use the information above to find the variance:

Probability of incrementing the counter value by 1 = $\frac{1}{n_{i+1} - n_i}$

$$\begin{aligned} n_{i+1} - n_i &= 100(i+1) - 100i \\ &= 100i + 100 - 100i \\ &= 100 \end{aligned}$$

$$\text{Probability} = \frac{1}{n_{i+1} - n_i} = \frac{1}{100}$$

Probability of no change in counter value = 1 - $\frac{1}{n_{i+1} - n_i}$

$$1 - \frac{1}{n_{i+1} - n_i} = 1 - \frac{1}{100} = \frac{99}{100}$$

Finding the Variance:

We already found in part a that $E[X_j]$ = 1

$$\begin{aligned} V[X_j] &= E[X_j^2] - E[X_j]^2 \\ &= \left[\left(0^2 * \frac{99}{100}\right) + \left(100^2 * \frac{1}{100}\right)\right] - 1^2 \\ &= 100 - 1 \\ &= 99 \end{aligned}$$

Therefore the variance in the value represented by the register after *n* Increment operations have been performed would be $99 * n = 99n$.

5. *Assume that A and B are implemented using doubly-linked lists such the PushA and PushB, as well as a single pop from A or B, can be performed in O(1) time worst-case.*

(a) What is the worst-case running time of the operations MultiPopA, MultiPopB, and Transfer?

MultiPopA involves popping the minimum of *k* items out of the *n* total elements from Stack A. The worst-case time would occur if *k=n* and we are popping all *n* elements from Stack A. Therefore, the worst case running time would be $O(n)$.

Very similar to *MultiPopA, MultiPopB* involves popping the minimum of *k* items out of the *m* total elements from Stack B. The worst-case time would occur if *k=m* and we are popping all *m* elements from Stack A. Therefore, the worst case running time would be $O(m)$.

Transfer involves popping element from Stack A and pushing that element into Stack B until either *k* elements have been moved or Stack A is empty. The worst case scenario would occur if *k=n* meaning we will be popping all *n* elements from Stack A and pushing all *n* elements into Stack B. As we know that push and pop require O(1) time, pushing and popping *n* elements for *Transfer* would require O(n) time.

(b) *Define a potential function Φ(n, m) and use it to prove that the operations have amortized running time O(1).*

Let's define the potential function $\Phi(n, m)$, as $\Phi(n, m) = 2n + m$.

As per our textbook, the total amortized cost ĉ of the *ith* operation is the following:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The amortized cost of *n* operations needs to take into account the actual cost and the change in potential due to the operation. The updated formula is the following:

$$\sum_{i-1}^{n} \hat{c} = \sum_{i-1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

where D_0 is the initial element and D_n is the following element.

To prove that the operations *PushA, PushB, MultiPopA, MultiPopB, & Transfer* have amortized running time *O(1)* we will use our function $\Phi(n,m) = 2n + m$ and the equation of the amortized cost of *n* operations we described above.

PushA - push element on stack A

$\sum_{i-1}^{n} c_i = 1$
$\Phi(D_n) = 2(n+1) + m$
$\Phi(D_0) = 2n + m$

$$\begin{aligned}\hat{c} &= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\\ &= 1 + 2(n+1) + m - (2n+m)\\ &= 1 + 2n+2 + m - 2n - m\\ &= 3\end{aligned}$$

Amortized Running Time: *O(1)*

PushB - push element on Stack B

$\sum_{i-1}^{n} c_i = 1$
$\Phi(D_n) = 2n + (m+1)$
$\Phi(D_0) = 2n + m$

$$\begin{aligned}\hat{c} &= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\\ &= 1 + 2n + (m+1) - (2n+m)\\ &= 1 + 2n + m + 1 - 2n - m\\ &= 2\end{aligned}$$

Amortized Running Time: *O(1)*

MultiPopA - popping the minimum of *k* items out of the *n* total elements from Stack A

$\sum_{i-1}^{n} c_i = k$
$\Phi(D_n) = 2(n-k) + m$
$\Phi(D_0) = 2n + m$

$$\begin{aligned}\hat{c} &= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\\ &= k + 2(n-k) + m - (2n+m)\\ &= k + 2n - 2k + m - 2n - m\\ &= -k\end{aligned}$$

Amortized Running Time: *O(1)*

MultiPopB - popping the minimum of *k* items out of the *m* total elements from Stack B

$\sum_{i-1}^{n} c_i = k$
$\Phi(D_n) = 2n + (m-k)$
$\Phi(D_0) = 2n + m$

$$\begin{aligned}\hat{c} &= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\\ &= k + 2n + (m-k) - (2n+m)\\ &= k + 2n - m - k - 2n - m\end{aligned}$$

$= 0$

Amortized Running Time: *O(1)*

Transfer - popping element from Stack A and pushing that element into Stack B until either *k* elements have been moved or Stack A is empty

$$\sum_{i-1}^{n} c_i = k + k$$
$$\Phi(D_n) = 2(n-k) + (m+k)$$
$$\Phi(D_0) = 2n + m$$

$$\begin{aligned}
\hat{c} &= \sum_{i-1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\
&= k + k + 2(n-k) + (m+k) - (2n+m) \\
&= 2k + 2n - 2k + m + k - 2n - m \\
&= k
\end{aligned}$$

Amortized Running Time: *O(1)*