HW3

Name: Ayush Jain
UNI: aj2672

1 Problem 1

Problem 9-4

a. Considering this problem in a similar manner as that of RANDOMIZED_QUICKSORT, the probabilities in this case will depend on the relative positions of i, j and k.
We can define three cases, based on these relative positions:

i) $k \leq i < j$
In such a case, the first pivot that we chose from k to j index should be either z_i or z_j. If the element is anything other than z_i or z_j inside the range k to j, then these two elements will never be compared. If on the other side, the pivot chosen is less than k or greater than j then it does not matter and we simply try again.
Hence, the probability for this case is given by: $\dfrac{2}{j-k+1}$

ii) $i \leq k < j$ or $i < k \leq j$
(Both i= k and j=k conditions cannot hold true at the same time because that would mean i=j)
In this case also, the element chosen should be either z_i or z_j if it lies between z_i and z_j. If the element lies to the left of z_i or to the right of z_j, we simply select again. Although the favourable cases are still the same but the number of total cases have changed and are now dependent on i and j.
The probability is given by: $\dfrac{2}{j-i+1}$.

iii) $i < j \leq k$
This is similar to a) as in that the pivot selected should be either z_i or z_j if it lies between i and k, otherwise it can be anything since we will get to chose again.
The probability is given by: $\dfrac{2}{k-i+1}$.

b. We derived the values of $E[X_{ijk}]$ in part a. In order to calculate $E[X_k]$ we sum the expressions over the range of i and j.

$$E[X_k] \leq \sum_{i=k}^{j-1} \sum_{j=k+1}^{n} \frac{2}{j-k+1} + \sum_{i=1}^{k} \sum_{j=k}^{n} \frac{2}{j-i+1} + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k} \frac{2}{k-i+1}$$

$$E[X_k] \leq 2(\sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k} \sum_{j=k}^{n} \frac{1}{j-i+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1})$$

c. When we partition our array by choosing a pivot i we perform n-1 comparisons. The number of comparisons in the second iteration depends on the relative values of k and i. To find an expected value we can find the average number of comparisons that will happen. In order to find an upper bound let us assume that k is always in the bigger partition achieved after the first division.

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=n/2}^{n-1} T(i)$$

Lets derive our result using substitution method. Lets assume T(i) = O(4n). Substituting the same

$$T(n) \leq n - 1 + \frac{2}{n} \sum_{i=n/2}^{n-1} (4ci)$$

$$T(n) \leq n - 1 + \frac{8c}{n}(n/2 + (n/2 + 1) + (n/2 + 2) + ... + (n - 1))$$

The quantity (n/2 + (n/2+1) + (n/2+2) + ... (n-1))/n/2 is equal to $3n^2/8 - (n - 1)$. Hence, we can write:

$$T(n) \leq n - 1 + \frac{8c}{n}(3n^2/8))$$

If c = 1,

$$T(n) \leq n - 1 + \frac{8c}{n}(3n^2/8))$$

$$T(n) \leq n - 1 + 3n$$

$$T(n) \leq 4n$$

or,

$$T(n) = O(4n)$$

d. The majority of the time taken by a randomized select algorithm is consumed in the partitioning of the element around a pivot. We select a pivot and then partition the elements around that pivot. Then if our pivot is the element we were looking for, then we return otherwise w repeat the algorithm again. So in one cycle the majority of the time is taken by the partitioning and the rest of the algorithm takes O(1) time. Thus, the total time over the entire algorithm can be equated to O(n), assuming n loops in the worst case and the total number of comparisons over all loops.

$$T(n) = O(n) + E[X_k]$$

$$T(n) = O(n) + O(4n)$$

$$\boxed{T(n) = O(n)}$$

2 Problem 2

Problem 7-5

a. For an element A'[i] to be chosen as a pivot, the three numbers selected should obviously have A'[i] as its middle element. The other two elements should lie on the either side of A'[i]. So, the favourable cases would be 1 for selecting A'[i], (i-1) options for selecting one element to the left of A'[i] and (n-i) for selecting one element to the right. The total possible cases could be choosing 3 numbers out of our array which can be done in $\binom{n}{3}$ ways.

Hence, overall probability: $\frac{(i-1)(n-i)}{\binom{n}{3}} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$

b. For choosing $\lfloor \frac{n+1}{2} \rfloor$ as the pivot element, we substitute i= $\lfloor \frac{n+1}{2} \rfloor$ in the above equation. Also, in the normal RANDOMIZED_QUICKSORT, the probability to choose the median as the pivot element is 1/n.
The ratio of the two probabilities would be:

$$\frac{6(\lfloor \frac{n+1}{2} \rfloor - 1)(n - \lfloor \frac{n+1}{2} \rfloor)}{(n-1)(n-2)}$$

Applying $\lim_{n\to\infty}$ to the above term, we get:

$$\lim_{n\to\infty} \frac{6(\lfloor \frac{n+1}{2} \rfloor - 1)(n - \lfloor \frac{n+1}{2} \rfloor)}{(n-1)(n-2)}$$

n being odd or even would not really matter, in the event of $n \to \infty$. For simplification, assuming n be odd, so that $\lfloor \frac{n+1}{2} \rfloor = \frac{n+1}{2}$. The above expression would become:

$$\lim_{n\to\infty} \frac{6(\frac{n+1}{2} - 1)(n - \frac{n+1}{2})}{(n-1)(n-2)}$$

$$\lim_{n\to\infty} \frac{6(\frac{n-1}{2})^2}{(n-1)(n-2)}$$

=

$$\lim_{n\to\infty} \frac{6(n-1)}{4(n-2)} = 3/2$$

Even if we assume n to be even we arrive at the same result.

c. For a good split, we define $i \in [n/2, 2n/3]$. If we sum our probability for i over these values, we get the probability of a good split. The ratio would be:

$$\sum_{i=n/3}^{2n/3} \frac{\frac{6(i-1)(n-i)}{n(n-1)(n-2)}}{1/n}$$

$$\sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{(n-1)(n-2)}$$

If we define a new variable, t = i/n, we can approximate the above sum to an integral. The above sum cn be approximated as:

$$\int_{1/3}^{2/3} 6t(1-t)dt$$

$$\int_{1/3}^{2/3} (6t - 6t^2)dt$$

$$(\frac{6t^2}{2} - \frac{6t^3}{3})\Big|_{1/3}^{2/3}$$

$$3(\frac{4}{9} - \frac{1}{9}) - 2(\frac{8}{27} - \frac{1}{27})$$

$$1 - \frac{14}{27} = \frac{13}{27}$$

d. As discussed in the text book, the running time of quicksort is given by $O(n) + T_{comp}$. O(n) represents the total time in selecting the pivot in n iterations of the algorithm and T_comp represents the total number of comparisons that happen during the execution of the algorithm and is given by $\Omega(n \log n)$. When we use the median-of-3-method instead of random selection to select our pivot, only the O(n) part of the execution time is affected, the number of comparisons are still the same and hence the running time is still $\Omega(n \log n)$.

3 Problem 3

Exercise 15.2-1 As given in the textbook, we create a matrix m where m[i,j] represents the minimum number of multiplications required to compute $A_{i..j}$

$$m[i,j] = \begin{cases} 0 & \text{if i = j} \\ min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if i < j} \end{cases}$$

Computing all the caluculations:

$$m[0,1] = \min\{m[0,0] + m[1,1] + 5 \times 10 \times 3\} = 150$$

$$m[1,2] = \min\{m[1,1] + m[2,2] + 10 \times 3 \times 12\} = 360$$

$$m[2,3] = \min\{m[2,2] + m[3,3] + 3 \times 12 \times 5\} = 180$$

$$m[3,4] = \min\{m[3,3] + m[4,4] + 12 \times 5 \times 50\} = 3000$$

$$m[4,5] = \min\{m[4,4] + m[5,5] + 5 \times 50 \times 6\} = 1500$$

$$m[0,2] = \min \begin{cases} m[0,0] + m[1,2] + 5 \times 10 \times 12 \\ m[0,1] + m[2,2] + 5 \times 3 \times 12 \end{cases} = 330$$

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + 10 \times 3 \times 5 \\ m[1,2] + m[3,3] + 10 \times 12 \times 5 \end{cases} = 330$$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + 3 \times 12 \times 50 \\ m[2,3] + m[4,4] + 3 \times 5 \times 50 \end{cases} = 930$$

$$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + 12 \times 5 \times 6 \\ m[3,4] + m[5,5] + 12 \times 50 \times 6 \end{cases} = 1860$$

$$m[0,3] = \min \begin{cases} m[0,0] + m[1,3] + 5 \times 10 \times 5 \\ m[0,1] + m[2,3] + 5 \times 3 \times 5 \\ m[0,2] + m[3,3] + 5 \times 12 \times 5 \end{cases} = 405$$

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + 10 \times 3 \times 50 \\ m[1,2] + m[3,4] + 10 \times 12 \times 50 \\ m[1,3] + m[4,4] + 10 \times 5 \times 50 \end{cases} = 2430$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + 3 \times 12 \times 6 \\ m[2,3] + m[4,5] + 3 \times 5 \times 6 \\ m[2,4] + m[5,5] + 3 \times 50 \times 6 \end{cases} = 1770$$

$$m[0,4] = \min \begin{cases} \min\{m[0,0] + m[1,4] + 5 \times 10 \times 50\} \\ \min\{m[0,1] + m[2,4] + 5 \times 3 \times 50\} \\ \min\{m[0,2] + m[3,4] + 5 \times 12 \times 50\} \\ \min\{m[0,3] + m[4,4] + 5 \times 5 \times 50\} \end{cases} = 1655$$

$$m[1,5] = \min \begin{cases} \min\{m[1,1] + m[2,5] + 10 \times 3 \times 6\} \\ \min\{m[1,2] + m[3,5] + 10 \times 12 \times 6\} \\ \min\{m[1,3] + m[4,5] + 10 \times 5 \times 6\} \\ \min\{m[1,4] + m[5,5] + 10 \times 50 \times 6\} \end{cases} = 1950$$

$$m[0,5] = \min \begin{cases} \min\{m[0,0] + m[1,5] + 5 \times 10 \times 6\} \\ \min\{m[0,1] + m[2,5] + 5 \times 3 \times 6\} \\ \min\{m[0,2] + m[3,5] + 5 \times 12 \times 6\} \\ \min\{m[0,3] + m[4,5] + 5 \times 5 \times 6\} \\ \min\{m[0,3] + m[4,5] + 5 \times 50 \times 6\} \end{cases} = 2010$$

Thus we arrive at the table m[i,j]:

	0	1	2	3	4	5
0	0	150	330	405	1655	2010
1		0	360	330	2430	1950
2			0	180	930	1770
3				0	3000	1860
4					0	1500
5						0

And the table s[i,j] for the optimum solutions looks like:

	0	1	2	3	4	5
0	0	0	1	1	3	1
1		0	1	1	1	1
2			0	2	3	3
3				0	3	3
4					0	4
5						0

Thus using the s[i,j] table, we can derive the following optimal parenthesization: $(A_0A_1)((A_2A_3)(A_4A_5))$

Exercise 15.2-3

$$P(n) = \begin{cases} 1 & \text{if n} = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if n} \geq 2 \end{cases}$$

By susbstitution method, if we assume that $P(k) = \Omega(2^k)$ and $P(n-k) = \Omega(2^{n-k})$ we should be able to prove for P(n).

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

$$\geq \sum_{k=1}^{n-1} c2^k c2^{n-k}$$

$$= c^2 2^n \sum_{k=1}^{n-1} 1 = (n-1)c^2 2^n$$

Now this expression should be greater than $c2^n$.

$$(n-1)c^2 2^n \geq c2^n$$

i.e

$$c(n-1) \geq 1$$

. So, for any n we have a threshold value for c. If c is greater than $1/(n-1)$ the above relation holds true which proves that $P(n) = \Omega(2^n)$

4 Problem 4

Exercise 15.4-4
The recurrence of the LCS of two strings X and Y is of the following form:

$$c[i,j] = \begin{cases} 0 & \text{if i=0 or j=0} \\ c[i-1,j-1]+1 & \text{if i, j} > 0 \text{ and } x_i = y_j \\ max(c[i-1,j], c[i,j-1]) & \text{if i, j} > 0 \text{ and } x_i \neq y_j \end{cases}$$

Thus, if we see clearly, while calculating the subsequence we need only two rows i and i-1. Rest all the rows can be discarded. Hence, the solution can be achieved using only 2.min(m,n) entries in the c table. So, we can modify the algorithm using two arrays prevRow and currRow of lengths min(m,n). During the iteration, the prevRow, as the name suggests holds the c[i-1] elements, while the $c[i]^{th}$ for the current row can be stored in the currRow. At the end of the loop we set prevRow = currRow and initialize currRow equal to zero.
Now, again if we see, we do not even use the entire two rows for our calculations. For the calculation of c[i][j] we need only the c[i-1][j-1], c[i-1][j] and c[i][j-1] elements. Once the calculation of c[i][j] is done, we can discard the element c[i-1][j-1]. For the next calculation, that is, c[i][j+1] we only need c[i-1][j], c[i-1][j+1] and c[i][j]. Hence we can achieve this by using one array cost of length min(m,n) + 1. This will store elements from c[i-1][j-1] to c[i][j-1] at a given point of time for calculation of c[i][j]. When c[i][j] is calculated we can overwrite c[i-1][j-1] with c[i][j] as that is not needed any more.

Exercise 15.4-4
Lets suppose we have a sequence A of n integers. We can sort this array and save it to another array B. Now, if we try to find the LCS of A and B, we will get our longest monotonically increasing subsequence of A.
We can prove this by contradiction. Lets say in the LCS of A and B, we do not get the longest monotonically increasing subsequence of A. Lets say the subsequence had the z_n as the last element. But we have another element z_{n+1} which occurs after z_n in A and is greater than z_n. But, if z_{n+1} is greater than z_n then it should be present in B as well and at some position after z_n. Hence, this should be a part of the LCS of A and B. Thus our assumption was wrong.

5 Problem 5

Problem 15-9 Lets say we have a string A of size n. If we were to split this string at a single point, the cost is always n, irrespective of the point at which we split it. But, if we had more points at which the string is to be split, we can reduce it to smaller problems. We try to select a point which minimize the cost of splitting for the two smaller strings produced from the split. Since, the cost of splitting the string at each level is the sum of the size of string with the cost of splitting the substrings, the optimal solution will be the one in which splitting the sustrings is optimized. Following is a recursive solution to the problem:

n is the size of the string to be split.
L is the array containing the points at which the string is to be split. 1 and m are the first and last indices of L.
a is an array which stores the order in which the string should be split.

SplitString(n, L[1..m], a)

1. $q = \infty$
2. minIndex = -1
3. for(i = 1 to m)
4. p = SplitString(L(i), L[1..i], a) + SplitString(n-L(i), L[i+1..m], a) + n
5. $if(p < q)$
6. q = p
7. minIndex = i
8. a.append(L[minIndex])
9. return q, a

For the non recursive solution we start the problem bottom up. We look at the last level splitting of our string and try to optimize that to achieve a better solution for second last level. We know that, if we decide to split our string at L[i] position in the last level, then the cost of such split will always be L[i+1]-L[i-1]. This is because our string has already been partitioned at all other points in L. According to the question, if our string was to be split in the order 10, 8, 2, then we consider 2 and determine that the cost for such a split would be 8-0=8. Otherwise, if we decided to split at 8 in the last level, the cost would be 10-2 = 8.
For the non-recursive solution, we define a new array L' of size m+2, which has 0 as the first element and n as the last elements and contains L array in between. L' = {0, L[1..m], n}. This simplifies our edge cases. If splitting was to happen at L[1], it means that we are splitting the portion of the string from 1st index to L[2] index at L[1] point, or in terms of L', it can be said that we are splitting from L'[1] to L'[3] at L'[2]. This helps to remove if else conditions.
The third loop with k is just to find the most optimal splitting by varying the split point between j and j+i.

c[i][j] stores the optimum cost of splitting the string between L[i] and L[j]
p[i][j] stores the key k of L, saying that splitting at L[k] would give us the most optimum solution.
the first loop iterates over the difference between indexes of L between which the string should be split. In the first iteration i=2, so with values of j it gives us the solution to optimal splitting at L[2], at L'[3], etc. In the next iteration for i=3, we achieve optimal solution for splitting between

L'[1] to L'[4] by summing the split at L'[2] and L'[3] in different orders and so on.

SplitString(n, L)

1. L' = 0, L, n
2. c = zeros[m+2][m+2], p = zeros[m+2][m+2] //initialize all elements to zero
3. for(i=2 to m+2)
4. for(j=1 to m-i+2)
5. c[j][j+i] = ∞
6. for(k=j+1; k¡j+i; k++)
7. q = c[L'[j]][L[k]] + c[L'[k]][L[j+i]] + L'[j+i]-L'[j]
8. if (q < c[j][j+])
9. c[j][j+i] = q
10. p[j][j+i] = k
11. print "Optimal Cost is" : c[1][m+2]
12. Print-Optimal-Order(p, 1, m+2, L)

Print-Optimal-Order(p, i, j, L)

1. k = p[i][j]
2. print L[k]
3. if(k-i $\geq$ 2)
4. Print-Optimal-Order(p, i, k, L)
5. if(j-k $\geq$ 2)
6. Print-Optimal-Order(p, k, j, L)

The if conditions are required because when the difference is less than 2, then there can be no further splitting between i and i+1. Minimum values should be i and i+2 to be able to split at i+1.

6 Problem 6

To reach the solution for this problem we need to start from the first month and then continuously derive the optimal solution for the successive months. if we have the optimal distribution of the demand till i months, we can optimize for the d[i+1] using this solution. If d[i+1] $\leq$ m, then we do not have to do anything. However, if not, then we vary our distribution of d[i]-m machines between this month and previous months to arrive at the most optimal solution.
The recursive procedure for this solution can be written as:

$$MinCost(i) = min(MinCost(i-1) + c(k) + h(d[i] - m - k)$$

That is to arrive at the MinCost for ith month, we calculate the ith cost of i-11 months and try to divide the current month's demand surplus of m, as k extra machines in the current month and d[i]-m-k machines distributed over previous months. We iterate for different values of k and try to derive the most optimal solution.

The non-recursive solution for the above procedure is:

d[i] is the demand for month i.
m is the machine production capacity of the company.
n is the total number of months.
h[i] gives the holding cost of i machine per month.
c is the cost of producing extra machine after m.
plan is our solution which gives the optimal distribution of machines produced in each month.

The first loop runs for the different months and tries to obtain the most optimal distribution of demand till the i^{th} month. The inner loop starts at a particular month previous to the current month and then tries to obtain an optimal plan by distributing the surplus demand. At a particular month 'mon' where plan[mon] is less than m, we increase the quantity of machines created by 1 and reduce the same from the i^{th} month whose optimal distribution we are trying to achieve. Thus our cost reduces by c but changes by the machine holding cost over the duration of months (i-m). If the month mon reaches its limit of m machines we again reduce mon to arrive at a new month which has scope of additional machines and try to fit the rest of the surplus demand here.

InventoryPlan(d, h, c, m, n)

1. plan = zeros(n)
2. plan[1] = d[1]
3. for(i = 2 to n)
4. if(d<m)
5. plan[i] = d[i]
6. else
7. p = q = plan
8. $cost = minP = c * (d[i] - m)$
9. for(j=i-1 to 0)
10. if(q[j] < m)

```
11.                     mon = j
12.                     q = plan
13.                     cost = c * (d[i] - m)
14.                     for(k = 1 to d[i]-m)
15.                         q[mon]++
16.                         cost = cost - c
17.                         if(q[mon] == plan[mon]+1)
18.                             cost = cost + h[1] × (i - mon)
19.                         else
20.                             cost = cost + (h[q[mon] - plan[mon]] - h[q[mon] - plan[mon] - 1]) × (i - mon)
21.                         if(cost < minP)
22.                             minP = cost
23.                             p = q
24.                         if(q[mon] == m)
25.                             while(q[mon] ≥ m && mon > 0)
26.                                 mon - -
27.             plan = p
28. return plan
```