HW4

Name: Ayush Jain
UNI: aj2672

1 Problem 1

Exercise 16.2-7
We get the maximum payoff if the largest element a_i in the set A is paired with the largest element b_i in the set B, the second largest of set A with the second largest of B, and so on.

Proof: Lets assume that we have the optimal solution S of two element arrays A given by a_1, a_2 and B given by b_1, b_2 respectively such that $a_1 > a_2$ and $b_1 < b_2$. Then the optimal solution has the product as ${a_1}^{b_1} \times {a_2}^{b_2}$. If we consider another solution S' where a_1 is paired with b_2 and a_2 is paired with b_1 then the product in that case would be ${a_1}^{b_2} \times {a_2}^{b_1}$. If we take the ratio of the two products we get:

$$\frac{payoff(S')}{payoff(S)} = \frac{{a_1}^{b_2} \times {a_2}^{b_1}}{{a_1}^{b_1} \times {a_2}^{b_2}}$$

$$\frac{payoff(S')}{payoff(S)} = \frac{{a_1}^{b_2-b_1}}{{a_2}^{b_2-b_1}} > 1$$

Since we know that $b_2 > b_1$ and $a_1 > a_2$, we can definitely say that this quantity is greater than 1. This means that our assumption of the optimal solution was wrong and $S' > S$. So we want a solution such that ith largest element of a is paired with ith largest element of B. We could do this by sorting both A and B in increasing order or sorting both in decreasing order or any other order where such a pairing is maintained.
We could also see the proof in another way. Lets say we have a_1 the biggest element of A and another element a_p. Then we can maximize the contribution of ${a_1}^{b_1}$ only if b_1 is also the largest element of B. If we had any other elements a_i and b_i of the decreasing order sorted A and B, then the opposite pairing of a_1 with b_i and a_i with b_1 will always result in a contribution ${a_1}^{b_i} \times {a_i}^{b_1}$ smaller than ${a_1}^{b_1} \times {a_i}^{b_i}$ as seen above.

Algorithm:
A, B: arrays of a_i and b_i respectively.

MaximizePayoff(A, B)

1. MergeSort(A)
2. MergeSort(B)
3. product = 1
4. for(i = 1 to A.length)
5. product = $product \times (a[i]^{b[i]})$
6. return product

The time requirement of step 1 and 2 is $O(n \log n)$. The time complexity of step 4 and 5 is $O(n)$. Hence the total time $T(n) = O(n \log n) + O(n \log n) + O(n) = O(n \log n)$.

2 Problem 2

Problem 16-2 a.
The minimum running time can be achieved by scheduling our tasks in the increasing order of their completion time.

Proof: Lets assume that we have two schedules a_1, a_2 with processing times p_1, p_2 respectively such that $p_1 < p_2$. If we schedule a_1 and then a_2 then $c_1 = a_1$ and $c_2 = a_1 + a_2$. However if we schedule the other way then, $c_1 = a_2$ and $c_2 = a_1 + a_2$. So we can see that the last c does not change and is the sum of processing times of all schedules. But the initial c_i can be reduced by selecting the tasks with minimum running time first and then moving in the increasing order.

Algorithm:
P: array of processing times of tasks. p[i].
C array stores the completion time of each task. C is an int array assuming that all P[i] are integer values.

MinimizeAvgRunTime(P)

1. int n = P.length
2. MergeSort(P)
3. int C[n]
4. C[1] = P[1];
5. for(i = 2 to n)
6. C[i] = C[i-1] + P[i]
7. int c = 0
8. for(i=1 to n)
9. c = c + C[i]
10. return c/n

In the above algorithm, we first sort elements of P in increasing order. Once, that is done we create the array C, which stores the completion time of tasks as they are executed. For that, it is required to some over elements of P.
Once we have the array C, the average completion time is just the average of elements of C.
The running time of the above algorithm would be $O(n \log n)$ defined by merge sort, since all other operations take $O(n)$ time.

Problem 16-2 b.
At time = 1, we consider the tasks for which r[i] = 1. Lets say, we have a list of tasks T. We can follow the same algorithm as above and find the process with the minimum processing time and start running it. Lets say this process is x. Now, at time = 2, new processes get added to our task list T(those for which r[i] = 2). We will again have to go through our list and find the process with the minimum processing time in T. If there is a process y in T(other than x) which starting now can complete before x, we suspend x and add it back to the list T with the remaining completion time and start y, otherwise we let x run for another second till processes with r[i] = 3 get added to

the list. And then repeating this for all time intervals, we can define the execution sequence of all processes and hence figure out the completion times of our processes.

Algorithm:
P: array of processing times of tasks.
R: array of release times of tasks in P.
L: array of processing time left for each task at any point of time.
C: array of completion time of tasks.
T: total combined running time of all tasks.

```
MinimizeAvgRuntime(P, R)
1.  int n = P.length, T = 0
2.  int L[n], C[n]
3.  for(int i = 1 to n)
4.      T = T + P[i]
5.      L[i] = P[i]
6.      C[i] = 0
7.  int current = -1
8.  for(i = 1 to T)
9.      if (current != -1)
10.         L[current] = L[current] - 1
11.         if(L[current] == 0)
12.             C[current] = time
13.     minL = 0
14.     for(j = 1 to n)
15.         if(R[j] ≤ time && L[j] != 0 && minL > L[j]
16.             minL = L[j]
17.             current = j
18. int c = 0;
19. for(int i=0 to n)
20.     c = c + C[i]
21. return c/n
```

As discussed earlier we iterate over time and at each time = t, we decreases the remaining processing time which had been running latest by 1. If the remaining processignt time reduces to 0, then we save t as the completion time for that task. Then we consider the processes that have been released till now and find the process which will complete the fastest and begin that process. Once the loop has iterated over all processes, we will have the completion time for all processes, using which we can find the average completion time for the processes.
In the above algorithm the outer loop runs T times and there is one inner loop which runs n times on each iteration. Hence, we can say time complexity, T(n) = O(T*n)

3 Problem 3

Problem 17-2 a.
We can do search by running a binary search on all filled sorted arrays. The worst case running time will be when all the arrays are full. The time complexity of binary search on a single array is O($\log n_i$).

$$T(n) = \sum_{i=1}^{k-1} O(\log n_i)$$

$$T(n) = O(\sum_{i=1}^{k-1} \log 2^i)$$

$$T(n) = O(\sum_{i=1}^{k-1} i)$$

$$T(n) = O(\frac{k(k-1)}{2})$$

But, we know that k = $\lceil \log(n+1) \rceil$.

$$T(n) = O(\log^2 n)$$

Problem 17-2 b.
For any insertion, we start from A_0 and move up to A_{k-1}. to find the first array A_i which is empty. We then create a new array B of size 1 containing the new element and then merge B with A_0. Then we merge the resulting array of the above operation with A_1. We do this till the array A_1 and save the result into A_i. The complexity to perform such an operation is (2^i). The worst case running time for insertion would be when all k-1 arrays are completely filled. In such a case T(n) = O(n).

The insertion is similar to our binary counter problem given in CLRS. If $< n_{k-1}, n_{k-2}, ..., n_0 >$ represents our binary representation of an array being filled or empty, then the number $n_{k-1}n_{k-2}....n_0$ represents the number of elements in the array. Each insertion is like an addition of 1 to this number. At each addition only one bit changes from 0 to 1 and the cost of the insertion when the ith bit changes from 0 to 1 is the size of the array A_i which is equal to 2^i.
Lets do an aggregate analysis to find out our amortized cost. For n insertions, n_0 changes from 0 to 1 on every alternate insertion, n_1 on every 4th insertion, n_2 on every 8th insertions and so on. Thus any ith bit changes from 0 to 1 $n/2^i$ times and the cost associated with every change is 2^i.

$$T(n) = \sum_{i=0}^{k-1} 2^i \frac{n}{2^i}$$

$$T(n) = \sum_{i=0}^{k-1} n$$

$$T(n) = n \sum_{i=0}^{k-1}$$

$$T(n) = n(k)$$

$$T(n) = O(n \log n)$$

Thus, each insertion has the amortized cost: $O(\log n)$.

Lets now try to calculate the amortized cost using the accounting method. We can associate an amortized cost of k per element for an insertion. The 1 cost will be used to insert that element the 1st time and then k-1 credit for future transfer of that element from one array to another. Hence the amortized cost per insertion is $O(k) = O(\log n)$

Problem 17-2 c.
In order to perform a delete we first search for the array and the position on which that element is present. Once we find the element in our array, we delete the element from the array. Then we use the binary representation $< n_{k-1}, n_{k-2}, ..., n_0 >$ to find the right most 1 in the representation. Lets say that n_i is the right most 1 bit. We take one element from A_i and transfer it to the array from which the element was deleted. Then we split the array A_i and fill up all the arrays smaller than A_i. If i=0, then we simply delete this element from A_0

Algorithm:
A: the array of array $A_0, A_1, ...A_{k-1}$ and the total number of elements in A combined is n.
p is the array A_p and q is the index in array A_q on which key is found. Search is the algorithm defined in part a to find the key.
i is the right most bit set to 1.

Delete(A, n, key)

1. $k = \lceil \log(n+1) \rceil$
2. p, q = Search(A, key)
3. int i = 0
4. for (i=0 to k-1)
5. if(A[i].length != 0) break;
6. insert A[i][2^i-1] into A[p]
7. int a = 0
8. for(int x = 0 to i-1)
9. for(int y= 0 to $2^x - 1$)
10. A[x][y] = A[i][a]
11. a = a +1

In the above algorithm we first find the position of our key in A, which comes out to A[p][q]. Then we find the right most bit = 1 to in n which is i. In the array A_i, we transfer the last element of A[i] to A[p] while maintaining the sorting order. Then we split the elements of array A[i](except A[i][$2^i - 1$], the last element of A[i]) and distribute them into arrays A_0 to A_{i-1}.

The complexity of search as we saw in a. is $O(\log^2 n)$. The complexity to find i is at most O(k-1). The last two loops which distribute the elements of A[i] into arrays A_0 to A_{i-1} have a time complexity of O(n) as it runs for all the elements of A[i]. Summing over all the steps it can be said that the time complexity of the DELETE operation is O(n).