Homework 3

Name: Tianyu Gu No: U89572036

Q1:

a. The answer will be [4, 3, 2, 6, 1, 5].

b. This can be proved by contradiction:
Let extracted [] be the output array by this algorithm and let O denote the correct array. If they are not the same, let j denote the number which makes extracted[j] the smallest value in the extracted [] array and fulfill the requirement extracted[j] != O[j]. It is obvious that O[j] is either bigger than extracted[j] or smaller than extracted[j]. let P = extracted[j] and P^* = O[j]

1. If $P^* < P$ (O[j] < extracted[j]):
   First, we can show that P^* is not in the array extracted [], otherwise P^* will be the smallest number in the extracted [] array makes the extracted [] incorrect. That means, we must have put P^* into the K_{m+1} set before we process P. On the other hand, P^* is in the array O[], it must initially lies in some set K_i such that i<=j. However, according to this algorithm, we must have processed P^* before P, since $P^* <= P$. Moreover, since extract[j] = P, we know that extracted[j] must have no value before we process P, then, we could not put P^* into the K_{m+1} set, since we have not union with K_j yet.
2. If $P^* > P$ (O[j] > extracted[j]):
   First, we can show that, P must initially lie in some set K_i which i <= j , since we only union with the sets afterward. We would never make extracted[j] = P, if P lies in the set before K_j. Moreover, the O array did not choose the P for O[j] but P^*, the only possible situation is the optimal method have chosen P in O[q], which q < j. otherwise, it must choose P, since $P < P^*$. But this means, extracted[q] must be larger than P, otherwise P will not be the smallest number that makes extracted [] incorrect. However, we cannot move P to the set K_j, since our algorithm process the P before extracted[q], and at that time extracted[q] must be empty, which means the set K_p still exists. Therefore, extracted[j] cannot equal to P.

Therefore, extracted[j] must equal to O[j] for all [j], which can be proved inductively. Thus, this algorithm is a correct one.

c. The answer for the data is:

```
[22,6,3,32,11,16,30,34,35,40,45,54,55,51,60,62,7,33,38,58,68,17,28,41,14,42,46,
 47,56,8,63,67,59,5,29,48,21,25,37,27,26,1,15,19,2,4,18,23,13,9]
```

The basic idea is use disjoint set to implement this algorithm. Additionally, we will need arrays which keep recoding the pairs of the number of set (K_1 ~ K_{m+1}) and the representative of each set. Therefore, we can always find the corresponding set number for each node and update them correctly.
The implementation is shown below:

```
1. // this is the typical disjoint_set class, which is the same as the textbo
   ok shows
2. // make one disjoint set cost O(1)
3. // find_set and union cost O(Î±(n))
4. public class Disjoint_Set{
5.     private int[] parent;
6.     private int[] rank;
7.     public Disjoint_Set(int n){
8.         parent = new int[n];
9.         rank = new int[n];
10.        for (int i=0; i<n;i++){
11.            parent[i] = i;
12.            rank[i] = 0;
13.        }
14.    }
15.
16.    public int find_set(int x){
17.        if (parent[x]!= x){
18.            parent[x] = find_set(parent[x]);
19.        }
20.        return parent[x];
21.    }
22.
23.    public void union(int i, int j){
24.        i = find_set(i);
25.        j = find_set(j);
26.        if (rank[i] > rank[j])
27.            parent[j] = i;
28.        else{
29.            parent[i] = j;
30.            if(rank[i]==rank[j])
31.                rank[j]++;
32.        }
33.
34.    }
35.
```

The implementation for OFF_LINE algorithm is:

```
1. // @para root: integer array used to store the representative of each s
   et. take set number(1~m)
2. //             as input, and return the correspoding representative.
3. // @para pos:  integer array used to store the set number. it takes the
    representative
4. //             as input and return the current set number of this root.

5. // @para extracted: the array used to store the extracted number.
6.
7.
8. public class offline{
9.
```

```
10.          public static void main(String[] args){
11.              int n = 100;
12.              int m = 50;
13.              int[] root = new int[m+2];
14.              int[] pos = new int[n+1];
15.              Disjoint_Set ds = new Disjoint_Set(n+1);
16.          // read in the file. nead In.class
17.              In in = new In("hw3test.txt");
18.              String current = in.readLine();
19.              int re = 0; // re is the representative
20.              int count = 0; // count the number of extract
21.          // this while loop read in all the number and E, then
22.          // it initialize the disjoint set, and set the representative

23.          // of each set to the first incoming number of each set.
24.          // this will take the O(n) running time.
25.               while(current!=null){
26.                       if(current.charAt(0)=='E'){
27.                           count++;
28.                           root[count]=re;
29.                           pos[re]=count;
30.                           re=0;
31.                       }
32.                       else{
33.                           int temp=Integer.parseInt(current);
34.                           if(re==0)
35.                               re=temp;
36.                           ds.union(re,temp); // in initialization, this
    will take constant time,
37.                       }                       // since re and temp are a
   ll reprsentative of each set.
38.                       current=in.readLine();
39.                   }
40.                   count++;
41.                   root[count]=re;
42.                   pos[re]=count;
43.                   re=0;
44.      // the algorithm described in the textbook. the algorithm takes O
   (3n * Î±(n)) running time.
45.
46.              int[] extracted = new int[m];
47.              for(int i=1;i<=n;i++){
48.                  int r=ds.find_set(i); //O(n*Î±(n))
49.                  int j=pos[r];
50.              // set extract and get the next non-empty set
51.                  if(j<m+1){
52.                      extracted[j-1]=i;
53.                      int k=j+1;
54.                      while(k<=m){
55.                          if(root[k]!=-1)
```

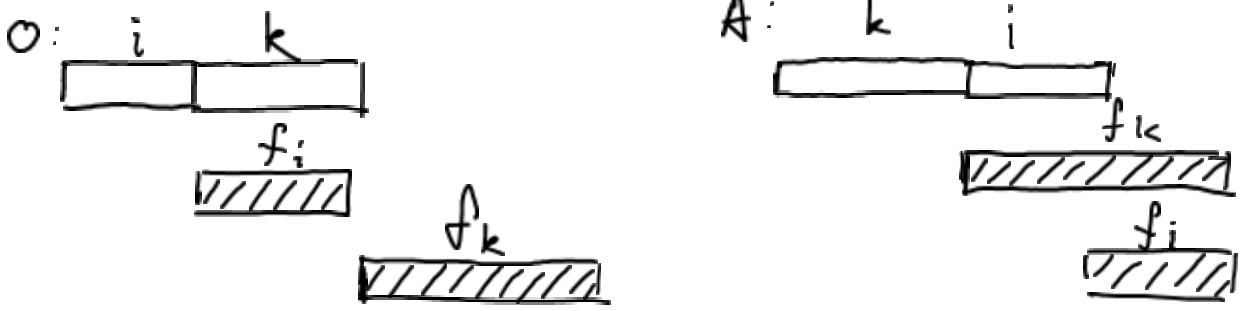
```
                                break;
                            k++;
                        }
                // union two sets and update the pos and representative a
   ccrodingly.
                        if(root[k]!=0){
                            ds.union(root[k],r);  //O(n*Î±(n))
                            int updated_root=ds.find_set(r); //O(n*Î±(n))

                            root[k] = updated_root;
                            pos[updated_root]=k;
                            root[j]=-1;
                        }
                        else{
                            root[k]=r;
                            pos[r]=k;
                            root[j]=-1;
                        }
                    }
                }

                for (int num=0;num<m;num++)
                    System.out.println(extracted[num]);

            }


        }
```

Since, we call find_set( ) 2n times, the union( ) n times, build_set( ) n times. The total running time will be O(n + 3n * α(n)) = O(α(n)).

Q2.

Schedule: we should run jobs in the order of decreasing finish time f_i.

Proof: let A be the schedule produced by our algorithm and O be the schedule generate be the optimal algorithm. I want to show by "exchange argument" that, A will be no worse than O. Thus, the A will be the optimal schedule.



Suppose in the O, job J_i and J_k are neighboring jobs, which J_i is scheduled before J_k, but $f_k >= f_i$.

The finishing time for these two jobs in O will be $T(o) = P_i + P_k + f_k$.

In schedule A, the finish time for these two jobs has two possibilities:

$T(a) = P_k + f_k$ or $T(a) = P_k + P_i + f_i$.

If $T(a) = P_k + f_k$, then we know that $T(a) = T(o) - P_i < T(o)$

If $T(a) = P_k + P_i + f_i$, then we know that $T(a) <= P_k + P_i + f_k = T(o)$

Thus, in either case, the finishing time of these two jobs in A is no worse than it in optimal schedule O. moreover, it is obvious that the finishing time for other jobs will remain the same. Therefore, by exchanging the jobs J_i and J_k in O, the total finishing time will stay the same (when there are other jobs that finished after these two) or become smaller. Therefore, by keep exchanging, we can get A. so that, A is no worse than O, thus an optimal schedule.

Meanwhile, if job J_i and J_k are not neighboring jobs in O, we can find that, for every jobs J_m between J_i and J_k. it can either switch with job J_i or job J_k. thus we can always switch the neighboring jobs first.

The running time of the algorithm to find this schedule basically depends on the sorting time. Thus, it will be a polynomial running time of n.

Q3.

This problem can be solved be Dijkstra's algorithm. Assume s is the starting node and we use d(u) to denote the minimum time we can reach node u from s. S be the set of nodes that we have explored. V is the set for all nodes. Therefore, the algorithm is:

Initially, S={s} and d(s) = 0.

While S ≠ V:

Select a node $v \notin S$ with at least one edge from S for which:

$$d'(v) = \min_{e=(u,v);\, u \in S} f_e(d(u))$$

Try all the possible v to make this formula as small as possible.

Add v to S and $d(v) = d'(v)$; $v.\pi = u$

End

The v.π is used to track the path, which will return the last node we should choose to get to v, so that we can know the exact path that takes the shortest time.

Pseudo code:

for each vertex $v \in G.v$:

Insert(v,∞)

v.π = Nil

end

Decrease_key(s, 0)

$S = \emptyset \;\; Q = G \cdot V$

While $Q! = \emptyset$:

u = Extract_min( )

$S = s \cup \{u\}$

for each vertex $v \in G\; adj\; j\{u\}$:

if v.key > $f_e(u.key)$:

Decrease_key (v, $f_e(u.key)$)

$v.\pi = u$

end

end for

end while

Running time: this can be implemented by the Fibonacci Heap, in which:

we call: Insert( ) |v| times

Extract_min( ) |v| times

Decrease_Key ( ) |E| times

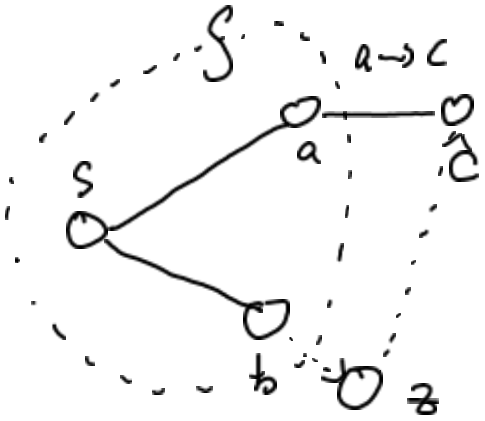
Thus the total running time will be $O\left(|V| \cdot lg^{|v|} + |E|\right)$.

Since |E| < $|V|^2$, suppose |V| = n, the running time will be $O(n^2)$, which is polynomial of n.

Proof:

This can be proved by induction:

1. When |S| = 1, it apparently holds.
2. Suppose when |S| = k, it holds, then consider the situation for |S| = k + 1:



As shown above, suppose we are going to choose path (a → c) and node c as the next destination. That means, we have determined that the shortest path to c from |S| is through a. if there exist another road, then, there must be at least a node out of the S, let's say z. so the path will be b → z →c, which makes $f_{(b,z)}(t_b) + f_{(z,c)}(t_c)$ smaller than$f_{(a\cdot c)}t(a)$. However, if z exists, the algorithm will include z to |S| instead of c. contradiction found. Therefore, when |S| = k + 1, it still holds

3. Therefore, finally when we reach the destination, it will be the shortest path we can find.