Advanced Algorithms Homework #2

Problem 1:

Solution:

In the given problem boxes arrive in an order. Let's say the order in which the boxes arrive are $B_1, B_2 \ldots\ldots\ldots B_n$. Each box has some weight. Let the weight of each box B_i be 'w_i'. It's also given in the problem that any truck can have a limit for its weight. That is, it can carry boxes only up to that mark. Let the maximum weight each truck carry is 'W'.

In order to pack the boxes into trucks we can also get to know from the question that-
-No truck can be overloaded. That is the total weight of all boxes in each truck must be less than or equal to the Weight capacity of the truck i.e. 'W'.
-The company also follows a policy of preserving the order. That is the order in which the boxes arrive is the order in which they are shipped. In other words, if box B_i is sent before the box B_j then accordingly B_i should arrive first than B_j (i.e. i < j).

In the question they asked us to prove that the greedy algorithm which the company uses is optimal. That is in other words it "stays ahead" of any other solution. In order to prove this, we consider any other solution. If greedy algorithm fits boxes $B_1, B_2 \ldots\ldots\ldots B_j$ into the first 'T' trucks and the other solution fits $B_1, B_2 \ldots\ldots\ldots B_i$ into the first 'T' trucks, then $i \le j$. This implies that the greedy algorithm is optimal by setting 'T' to be the number of trucks used by the company's algorithm.

We will prove this by induction on 'T'. The case T = 1 is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now let's assume it holds T-1: the greedy algorithm fits $B_1, B_2 \ldots\ldots\ldots B_{j'}$ into the first T-1 trucks and the other solution is $B_1, B_2 \ldots\ldots\ldots B_{i'}$ into the first T-1 trucks, then $i' \le j'$. Since $i' \le j'$, the greedy algorithm is able to fit more. This completes the induction step, the proof of the problem and hence the optimality of the algorithm.

Problem 2: Design an efficient (polynomial time) algorithm that, given $n, p,$ and q as inputs, determines the *minimum cost* among all possible traversal strategies. Your algorithm must make no assumption about $n, p,$ and q, except that they are all positive. *Hint: consider how a traversal splits the original tree into left and right subtrees, then express the cost in terms of the number of edges to traverse from the root to each sub-root, and finally try a dynamic programming approach.*

Solution:

My algorithm traverses recursively from the root node till it reaches the leaf nodes. Once it reaches the leaf nodes it starts unwinding the stack and while doing so calculates the minimum cost at each node until we reach the root node.

Algorithm:

```
Minimum_cost(root)
    //base condition for the recursion
    if (root. right = = Null && root. left = = Null)
        return [0];
    endif

    initialize left_set = Minimum_cost (root. left);
    initialize right_set = Minimum_cost (root. right);

    //Create an empty array
    initialize Return_set = [ ];

    return_set. append (left_set [0] + root. p)

    for i:1 to len(left_set)
        return_set. append (min(left_set[i]+root. p, right_set[i-1] +root. q)
    endfor

    return_set. append(right_set[len(right_set)-1] +root. q)

    return return_set
```

This is a variant of the DFS algorithm.

Problem 3*: Let $G = (V, E)$ be a graph on which each edge $e \in E$ has an associated value $0 \leq rel(e) \leq 1$ that represents the reliability of a communication channel through edge e. In other words, if $e = (u, v)$, then $rel(e)$ is the probability that the channel from u to v will not fail. Assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices and prove that it is correct.*

Solution:

We can get an efficient algorithm to find the most reliable path between two given vertices by simply modifying Dijkstra's Algorithm.

Algorithm:

```
Reliable_path(G):
    Set all the distance of all the vertices as '0' and the source vertex with '1';
    Save all the vertices in max_heap;
    Do until max_heap is not empty:
        Current_vertex = EXTRACT MAX_HEAP;
        For each neighbor of current_vertex:
            If ((current_vertex reliability) (currentEdge) > neighbor_reliability):
                Update the value of neighbor_reliability
```

The complexity for the above algorithm is O (V^2)
In order to prove that the given algorithm is correct.
This is a modification of Dijkstra's algorithm which is a greedy algorithm. In this algorithm we calculate the best path based on the reliability i.e. the higher the reliability the better is the path. Since the reliability to reach from one node to other is greater and all the values are positive, we can say that the algorithm is optimal.

Problem 4:
a) *Implement (in Java) Myers' $O(ND)$ algorithm to compute the edit distance between A and B*

Solution:

```
public class Diff {
    private int diff(int[] a, int[] b) {
        int N = a. length;
        int M = b. length;
        int max_val = N + M;
        int len = 2 * max_val + 1;
        int[] v = new int[len];
        v[1]=0;
        int x;
        int i;
        for (i=0; i<=max_val; i++) {
            for (int j=-i; j<=i; j+=2) {
                int k = j + max_val;
                if((j==-i) || (j! =i) && (v[k-1] < v[k+1])) {
                    x = v[k+1];
                }
                else {
                    x = v[k-1] +1;
                }
                int y = x - j;
                while ((x<N) && (y<M) && (a[x]==b[y])) {
                    x = x+1;
                    y = y+1;
                }
```

```
                                v[k]=x;
                                if ((x>=N) && (y>=M)) {
                                        return i;
                                }
                  }
                }
                    return i;
                }


public static void main (String[] args)
{
                int[] a = {1,2,3,5};
                int[] b = {6,8,9,3};
                Diff e = new Diff ();
                System.out.println("editdistance:" + e. diff (a, b));
}

}
```

b) Implement (in Java) the Wu-Manber-Myers-Miller $O(NP)$ algorithm for the same problem

Solution:

```
import java. util. Arrays;

public class EditDistance {
     public int Edit (int [] a, int [] b) {
        int len = a. length + b. length + 3;
        int diff = b. length - a. length;
        int off = a. length + 1;
        int v []   = new int[len];

        Arrays. fill(v, -1);
        int i =0;

        for (i=0; v [diff + off] < b. length; ++i) {
            for (int j=-i; j<=diff-1; ++j) {
                v [j + off] = snakes (a, b, j, v[j-1+off] + 1, v[j+1+off]);
            }
            for (int k = diff + i; k>=diff+1; --k) {
                v [k + off] = snakes (a, b, k, v [k-1+ off] +1, v[k+1+off]);
            }
            v [diff + off] = snakes (a, b, diff, v [diff - 1+off] + 1, v[diff+1+off]);
        }
        return (diff + 2 * (i-1));
    }

    private int snakes (int [] a, int [] b, int k, int c, int d) {
        int y = Math.max (c, d);
        int x = y - k;
        while (x < a. length && y < b. length && a[x] == b[y]) {
            ++x;
            ++y;
        }
        return y;
    }

    public static void main (String [] args) {
                int a [] = {1,2,3,5};
                int b [] = {6,8,9,3};
                EditDistance editdistance = new EditDistance ();
                System.out.println("editdistance:" + editdistance.Edit(a, b));
}


}
```

c) Generate 8 pairs of random sequences A and B, of lengths $M = 4000$ and $N = 5000$ elements respectively, with $D = 10, 50, 100, 200, 400, 600, 800,$ and 1000 randomly located deletions and $N - M + D$ insertions between them. Apply both algorithms above, then tally and report how many comparisons each algorithm performs between an element from A and an element from B.

Solution:

```
import java. util.ArrayList;
import java. util.Arrays;
import java.util.Random;

//O(NP)
public class EditDistance {
                public int counter_ND = 0;
                public int counter_NP = 0;
     public int Edit (int[] a, int[] b) {
        int len   = a. length + b.length + 3;
        int diff  = b.length - a.length;
        int off = a.length + 1;
        int v[]   = new int[len];
        Arrays.fill(v, -1);
        int i =0;
        for(i=0;v[diff + off] < b.length; ++i) {
            for (int j=-i;j<=diff-1;++j) {
                this.counter_NP++;
                v[j+off] = snakes (a, b, j, v[j-1+off]+1, v[j+1+off]);
            }
            for (int k=diff+i;k>=diff+1;--k) {
                this.counter_NP++;
                v[k+off] = snakes(a, b, k, v[k-1+off]+1, v[k+1+off]);
            }
            v[diff+off] = snakes (a, b, diff,v[diff-1+off]+1,v[diff+1+off]);
        }
                  return (diff + 2 * (i-1));
    }

    private int snakes (int[] a, int[] b, int k, int c, int d) {
        int y = Math.max(c, d);
        int x = y - k;
        while (x < a.length && y < b.length && a[x] == b[y]) {
            ++x;
            ++y;
        }

        return y;
    }

//generating arrays A and B
    private ArrayList<Integer> generate_list (int length) {
                ArrayList<Integer> a = new ArrayList<Integer>();
                Random rand = new Random();
                for(int i=0;i<length;i++) {
                                a.add(rand.nextInt(10000));
                }
                return a;
```

```
}

private void append_list(ArrayList<Integer> b,int length) {
        Random rand = new Random();
 for (int i=0; i<length; i++) {
        int x = rand.nextInt(10000);
        b.add(x);
        }
}

private ArrayList<Integer> mutate_list(ArrayList<Integer> b, int d, int num_inserts) {
        Random rand = new Random();
        for (int i = 0; i<d; i++) {
                int random_index = rand.nextInt(b.size());
                b.remove(random_index);
        }
        for (int i=0; i<num_inserts; i++) {
                int random_index = rand.nextInt(b.size());
                int new_int = rand.nextInt(10000);
                b.add (random_index, new_int);
        }
        return b;
  }

//O(ND)
private int diff (int [] a, int [] b) {
        int N = a.length;
        int M = b.length;
        int max_val = N + M;
        int len = 2 * max_val + 1;
        int[] v = new int[len];
        v[1]=0;
        int x;
        int i;
        for (i=0; i<=max_val; i++) {
            for (int j=-i; j<=i; j+=2) {
                int k = j + max_val;
                this. counter_ND++;

                if((j==-i) || (j! =i) && (v[k-1] < v[k+1])) {
                    x = v[k+1];
                  }
                  else {
                       this.counter_ND++;
                       x = v[k-1]+1;
                    }
                  int y = x - j;
                  while ((x<N) && (y<M) && (a[x]==b[y])) {
                         x = x+1;
                         y = y+1;

                  }
                  v[k]=x;
                  if ((x>=N) && (y>=M)) {
                       return i;
                  }
            }
        }
        return i;
        }
public static void main (String[] args) {
        int [] d = new int[] {10, 50, 100, 200, 400, 600, 800, 1000};
```

```
                for (int iter = 0; iter<8; iter++) {
                  EditDistance editdistance = new EditDistance();
                 ArrayList<Integer> a = editdistance.generate_list(4000);
                 ArrayList<Integer> b = new ArrayList<Integer>(a);
                 editdistance.append_list(b, 1000);
                 editdistance.mutate_list(b, d[iter], 5000-4000+d[iter]);
                 Integer [] temp_a = new Integer[a.size()];
                Integer[] temp_b = new Integer[b.size()];
                int[] new_a = new int[a.size()];
                int[] new_b = new int[b.size()];
                temp_a = a.toArray(temp_a);
                temp_b = b.toArray(temp_b);
                for (int i=0; i<a.size(); i++) {
                        new_a[i] = temp_a[i];
                }
                for (int i=0; i<b.size();i++) {
                        new_b[i] = temp_b[i];
                }
                System.out.print("d: " + d[iter] + " ");
                System.out.print("editdistance NP: " + editdistance.Edit(new_a, new_b) + " ");
                System.out.print("editdistance ND: " + editdistance.diff(new_a, new_b) + " ");
                System.out.print("counter NP: " + editdistance.counter_NP + " ");
                System.out.println("counter ND: " + editdistance.counter_ND);

                }
        }
}
```

Experimental results:

M	N	No of deletions	Edit Distance	O(NP) comparisons	O(ND) comparisons
4000	5000	10	2014	16064	2029113
4000	5000	50	2080	83681	2164281
4000	5000	100	2178	188100	2373021
4000	5000	200	2302	327104	2650905
4000	5000	400	2646	752976	3502305
4000	5000	600	2956	1187441	4370925
4000	5000	800	3286	1702736	5401185
4000	5000	1000	3602	2247204	6489805

The number of comparisons for O(NP) are less when compared to O(ND).

Problem 5*: You are given two sequences A and B, each containing n positive integers. You can reorder each set-in whatever way you want. After reordering, let a_i be the i-th element of sequence A and let b be the i-th element of sequence B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$ dollars. Give a greedy algorithm to maximize your payoff, prove that your algorithm is correct, and analyze its running time.*

Solution:

Let A = $[a_3, a_1, a_5, a_4 \ldots\ldots\ldots\ldots a_n]$
Let B = $[b_8, b_3, b_1, b_6 \ldots\ldots\ldots\ldots b_n]$

Given payoff $\prod_{i=1}^{n} a_i^{b_i}$ ***dollars.*** We need to find a greedy algorithm to maximize the payoff. In order to do so we should implement the algorithm in the following way-

Algorithm:

```
Initialize A = [$a_3, a_1, a_5, a_4 \ldots\ldots\ldots\ldots a_n$]; ---------- O (1)
Initialize B = [$b_8, b_3, b_1, b_6 \ldots\ldots\ldots\ldots b_n$]; ----------O (1)

Sort A in increasing order; -----------------------------O (nlogn)
Sort B in increasing order; ----------------------------- O (nlogn)
Initialize payoff = 0; ------------------------------------- O (1)
for i: 0 to A. length: --------------------------------------- O (n)
    payoff = payoff * pow(a[i], b[i]); -------------------- O (1)
Endfor
Return payoff;
```

The Time complexity for the algorithm is as follows-

As we are sorting the arrays the complexity will be O (nlogn) and as we are iterating the till the end of the array it takes O (n). Therefore, the complexity is – O(nlogn) + O (n) = O (nlogn)

T (n) = O (nlogn)

We need to show that the solution given by this algorithm is optimal. We know that $a_1 \le a_2 \le a_3 \le a_4 \ldots\ldots\ldots\ldots \le a_n$ from the solution. Since we know that the payoff is $\prod_{i=1}^{n} a_i{}^{b_i}$ ***dollars,*** the payoff will increase if $b_{i+1} \ge b_i$ i.e. if the array B is sorted in an increasing order format. Therefore, the solution obtained is an optimal solution.

Reference - https://www.udemy.com/

https://publications.mpi-cbg.de/Wu_1990_6334.pdf
https://www.geeksforgeeks.org/arrays-in-java/