Advanced Algorithms
Freie Universitat Berlin
Winter 2023-24
László Kozma & Michaela Krüger

Exercise Sheet 5

Jatin Kansal, Sandip Sah

This homework answers the problem set sequentially.

Exercise 1

The pseudocode for the algorithm taught in class is as follow:

```
1. function minimumEditDistance(a, b):
2.     n = length of string a
3.     m = length of string b
4.
5.     if n is 0:
6.         return m
7.     if m is 0:
8.         return n
9.
10.     if a equals b:
11.         return 0
12.
13.     if last character of a equals last character of b:
14.         return minimumEditDistance(a without last character, b without last
                character)
15.
16.     return minimum of (
17.         1 + minimumEditDistance(a with last character removed, b),
18.         // Deletion in 'a'
19.         1 + minimumEditDistance(a, b with last character removed),
20.         // insertion in 'a'
21.         1 + minimumEditDistance(a without last character, b without last
            character)
22.         // last element of a changed to b
23.     )
```

a

		n	a	r	u	t	o
	0	1	2	3	4	5	6
k	1	1	2	3	4	5	6
a	2	2	1	2	3	4	5
r	3	3	2	1	2	3	4
u	4	4	3	2	1	2	3
m	5	5	4	3	2	2	3
a	6	6	5	4	3	3	4
n	7	7	6	5	4	4	4

The optimal sequence of operation is: (1,1)->(2,2)->(3,3)->(4,4)->(5,5)->(6,6)->(6,7).
The edit distance is 4

preblem b Create a matrix pf size 26×26 to store weight for each possible pair of letter. While calculating the distance between two last element of substrings, use respective weight instead of default value of 1.

To be precise, the algorithm can be modified in the following ways:

```
1. function minimumEditDistance(a, b):
2.     n = length of string a
3.     m = length of string b
4.
5.     if n is 0:
6.         return m
7.     if m is 0:
8.         return n
9.
10.     if a equals b:
11.         return 0
12.
13.     if last character of a equals last character of b:
14.         return minimumEditDistance(a without last character, b without last
                character)
15.
16.     return minimum of (
17.         0.5 + minimumEditDistance(a with last character removed, b),
18.         // Deletion in 'a'
19.         0.5 + minimumEditDistance(a, b with last character removed),
20.         // insertion in 'a'
21.         w[a[n], b[n]] + minimumEditDistance(a without last character, b
            without last character)
22.         // last element of a changed to b
23.     )
```

Since nothing is mentioned about cost for insertion and deletion, it is considered 0.5. In line 17 and line 19, 0.5 is used to represent deletion and insertion in a. For line 21, w matrix is used to assign the weight.

Exercise 2

When an element of matrix is corrupt, the probability that it will be detected is $1/2$ and that of not getting detected is $1/2$ as well. if we were to increase the number of corrupt to n element, the probabilty of not getting detected for all corrupt element is 0.5^n. This gives the probability of getting detected to $1 - 0.5^n$.

For the calculation of probability for each number of corrupt element, the algorithm was ran 1000 times and number of times it was detected was counted. num of detected divided by total number of runs was used as calculated probability.

The calculated probability was compared with expected probability for each number of corruption. The result can be seen in the graph attached. The graph for both are quite similar. This proves the assumption that for each corruption, the probability of not getting detected decreases by half.

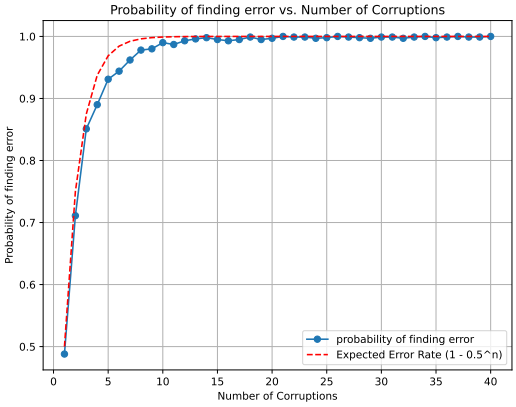

Figure 1: expected and obtained probability

Exercise 3

Choice of problem: part a
Let's first think about the sub-problem structure for this problem. We know that at every step, either the man moves one step, the dog moves one step or both of them move together one step. To minimize the total length of leash, we would want to minimize the length of additional leash needed at every step. For example, consider the last point (x_n, y_n). To reach this point, there are three possibilities.

1. They started from the position (x_{n-1}, y_n) and the man walked one unit.
2. They started from the position (x_n, y_{n-1}) and the dog walked one unit.
3. They started from the position (x_{n-1}, y_{n-1}) and both the dog and the man walked one unit.

Hence, to get the optimal value of the leash to reach (x_n, y_n), we must get the optimal values of each of these three possibilities and choose the one with the minimum length of the leash. Let's define a function $f(z)$ such that the value of the function is 0 for any

$z \leq 0$ and z for all other values. Also consider $OPT(x_i, y_j)$ to be the minimum length of the leash required to reach the step (x_i, y_j). Then, the length of the leash in each of these cases is given by:

1. $L_1 = OPT(x_{n-1}, y_n) + f(|x_n - y_n| - OPT(x_{n-1}, y_n))$
2. $L_2 = OPT(x_n, y_{n-1}) + f(|x_n - y_n| - OPT(x_n, y_{n-1}))$
3. $L_3 = OPT(x_{n-1}, y_{n-1}) + f(|x_n - y_n| - OPT(x_{n-1}, y_{n-1}))$

Here, we wrote the pairwise distance between x_i and y_j as $|x_i - y_j|$. Now, we can write the optimal value for the leash to reach (x_n, y_n) as:

$$OPT(x_n, y_n) = min(L_1, L_2, L_3) \tag{1}$$

We can see that the optimal solution depends on the optimal solutions for the steps before. We can further generalize this to formulate a recurrence relation in general.

$$\begin{aligned} OPT(x_i, y_j) \quad = \quad & min(OPT(x_{i-1}, y_j) + f(|x_i - y_j| - OPT(x_{i-1}, y_j)), \\ & OPT(x_i, y_{j-1}) + f(|x_i - y_j| - OPT(x_i, y_{j-1})), \\ & OPT(x_{i-1}, y_{j-1}) + f(|x_i - y_j| - OPT(x_{i-1}, y_{j-1})) \end{aligned}$$

The base case for this recurrence is that the $OPT(x_1, y_1) = |x_1 - y_1|$.
Now we can use dynamic programming to solve these subproblems in each recurrence. Let us think about the number of subproblems we will need to solve for a solution. We can see that a subproblem corresponds to a step that the pair of the man and dog may take. To go from (x_1, y_1) to (x_n, y_n) would take a maximum of $2n$ steps. This is because at every step, either the x_i or y_i must increase by one. Otherwise they are not moving. In some of the steps, they may even both increase by 1. Since they take $2n$ steps at maximum, we also need to solve $2n$ subproblems. For each subproblem, we use the previous solutions and some transformation. The total time cost for each recurrence step is a constant (c). Hence, the total time will be $2cn$ which gives us a time complexity of $O(n)$.