COMP3011 Homework 3

ZHANG Caiqi 18085481d

November 16, 2020

1

1.1 a)

The only probability that the two will be compared during the execution is that one of the largest or the least is selected as the pivot in the first dividing iteration. If none of them is the pivot, then after the first iteration, they will be divided into two parts and will not be compared again. Therefore,

$$P_{compared\ at\ least\ once} = 1 - P_{not\ be\ compared} = \frac{2}{n}$$

1.2 b)

From the result, we now that in the output sorted array, the second smallest number in A and the third smallest number in A will be adjacent. Since they are adjacent in the sorted array, when doing partition, if they are in the same smallest partition, they will definitely be compared. On the other hand, if they are not in the same smallest partition, when doing combination, since they will be adjacent in the sorted array, they will also be compared. Therefore,

$$P_{be\ compared} = 1$$

2

In this question, the edge is either good or not good. Therefore, we may define a indicator random variable to help the proof.

$$I\{A\} = \begin{cases} 1, & if\ it\ is\ good\ edge, \\ 0, & if\ it\ is\ not\ good\ edge. \end{cases}$$

Let's define $P\{G\}$ as the probability of any two nodes with an edge and the edge is good. Since there are four colors in total and all the vertices are assigned uniformly at random and independently, then

$$P\{G\} = 1 - 4 \times \frac{1}{4} \times \frac{1}{4} = \frac{3}{4}$$

Then, for a random chosen edge, the probability that it is a good edge is $\frac{3}{4}$. According to the Lemma 5.1 in lecture 5, the $E[I\{G\}] = P\{G\} = \frac{3}{4}$. Also, we know that any graph can be colored by 4 colors, which is the optimal solution. Considering the maximization problem, the output is a relaxed 4-coloring of G with as many good edges as possible. Let $E[G]$ be the expected good edge of the random graph, $E[O]$ be the expected good edge pf the maximization output and n be the optimal solution, then

$$E[O] \geq E[G] = E[\sum_{i=1}^{n} G_i] = \sum_{i=1}^{n} E[G_i] = \frac{3}{4}n$$

Therefore, the expected number of good edges is at least $\frac{3}{4}$ times the number of good edges in an optimal solution.

3

- Author: Grenet, B. and Volkovich I.
- Paper: One (more) line on the most Ancient Algorithm in History. [1] Details in reference list.
- The paper focuses on the Euclidean Algorithm. The new analysis provides a clean, "one-line" upper bound which turns out to be optimal. The main idea is based on the so-called Potential Method. It would be nice to see if we could replace other kind of analyses that rely on Fibonacci numbers by a potential argument
- In the analysis, the author used the following potential function:

> **Definition 2.3.** *For $1 \leq i \leq m$, we define $s_i \triangleq x_i + y_i$.*
>
> Next, we will show that s_i is a *potential* function for this algorithm. In particular, the function loses a constant fraction of its mass in every step, yet it is bounded away from zero. The following is immediate from the definition, given the above observation.

4

The main idea of my algorithm is to maintain a window of the same size of P. When moving the window from the start of T, compare whether the string in the window is an anagram of P. The key points are 1. how to initialize an array to store the string. 2. how to maintain the window and 3. how to compare whether two strings are anagrams.

- Initialization: Because that all the characters in P is from the alphabet A to Z, therefore we can initialize an array Alphabet_P[] of length 26 to store the frequency number of each character in the corresponding place. For example, if letter "A" appears twice in P, then Alphabet_P[1] = 2. For the Window[], which is of the same length with P, we also create Alphabet_W[] for it.

- *Check*(Alphabet_W[], Alphabet_P[]): To check whether W and P is a pair of anagrams, we compare Alphabet_W[] and Alphabet_P[]. The method is to transverse two array and check whether the two array is the same.
- *Update*(Window[]): To update the window, we need to not only update the string in the window but also update the Alphabet_W[]. Actually, we only need to delete the first letter in W and add the next one.

The general algorithm is as follows.

Algorithm 1: Find Anagrams

Input: A string T and a string P, both over the alphabet A,B,C,...,Z
Output: Where the anagram of P occurs

```
1 Initialize Window[], Alphabet_W[], Alphabet_P[];
2 while i ≤ length(T)-length(P) do
3 |   Check(Alphabet_W[], Alphabet_P[]);
4 |   Update(Window[]);
5 |   Update(Alphabet_W[]);
6 end
7 return -1
```

The time complexity mainly consist of the following parts: initialization, transverse T, compare and update. The initialization need to transverse P so it is in $O(|P|)$ time. To transverse T, it takes $O(|T|)$ time. *Check*() takes constant time because it just compares two fixed length array. *Update*() also takes constant time. Therefore, the worst-case running time of this solution is $O(|T| + |P|)$.

5

This question is very similar to the Exact String Matching Problem. The only difference is that in P contains exactly one "?". However, we can still use the linear algorithm in the lecture with some modification. We now consider "?" as a wildcard, in other words, when comparing two characters, "?" equals to any other character. Then we can still use the string matching algorithm. Here is the reference Z-box based algorithm from lecture notes.

Fast string matching algorithm

1. Given T and P, construct a new string $S = P\$T$ consisting of the string P followed by the symbol "$" followed by the string T, where "$" is a symbol that appears in neither P nor T.
2. Run the Z-Box Algorithm for the Longest Matching Prefixes Problem on input S to obtain Z_k for $k \in \{2, \ldots, m+1+n\}$.
3. Output all values of $k - m - 2$ such that $Z_k = m$.

The worst-case running time of this solution is $O(|T| + |P|)$.

6

The iterative version is as follows.

Algorithm 2: FIND-SET(x)

```
1  Initialize j, k = x, r = x;
2  while r.p ≠ r do
3  |   r = r.p;
4  end
5  while k ≠ r do
6  |   j = k.p;
7  |   k.p = r;
8  |   k = j;
9  end
10 return r;
```

The first while loop is used to find the needed value. The second one is used to update the set. After running this modified procedure, the x.p-values have the same values as if the recursive Find-Set above had been used.

References

[1] B. Grenet and I. Volkovich, "One (more) line on the most ancient algorithm in history," in *Symposium on Simplicity in Algorithms*. SIAM, 2020, pp. 15–17.