CS2040S: Data Structures and Algorithms

Recitation 3

Problem 1. (DNA Sequence)

Imagine, instead of pancakes, you are given a DNA string. You can reverse any segment of the string, and the cost of reversing a segment of length k is k. For example, below is a small segment of the COVID-19 sequence obtained from GenBank:

C A G T G A C A A T

And if you reverse $[2, 5]$ (i.e., items at indices 2, 3, 4, 5 in the sequence), then you get:

C A A G T G C A A T

This reversal costs 4 (length of reversal). For today, **assume you can examine the string for free** before making any reversals. In addition, **the only legal operations are reversals**.

Problem 1.a. Assume your string is binary: only made up of letters 'A' and 'T'. Devise a divide-and-conquer algorithm for sorting. What is the recurrence? What is the running time?

Note: You cannot simply count the 'A's and rebuild the string! because string reversal is the only valid operation. You may examine the string as much as you want before deciding which reversals to do. That is, reading the string is free, only reversals have cost.

Problem 1.b. Now, assume your string consists of arbitrary characters(i.e. string can consist of any number of different characters). Devise a QuickSort-like algorithm for sorting (*Hint:* use the binary algorithm above to help you implement partition). What is the recurrence? What is the running time? Assume for today that each element in the string is unique, i.e., there are no duplicates.

Problem 1.c. Extra. What if there are duplicate elements in the string? (*Hint:* think the most extreme case for duplicate elements) Can you still use the exact routine from the previous part? If not, what would you now do differently?

Problem 2. **(Everyday I'm Shuffling)**

We have seen by now that the role of randomness in algorithms can be a useful and important one. One such example is QuickSort: if we randomly permute the array first, then we can run a deterministic QuickSort algorithm (where the first element is the pivot) and it will ensure good performance, with high probability. Of course, this is a terrible idea if our array is initially almost sorted. In the real world, randomness can also be a crucial feature in applications. For instance, casinos needs to ensure that their game instances should be generated completely at random, else they risk players exploiting the games. There are quite a number of prolific cases where players successfully exploited casino games after observing flaws in their algorithms. Clearly, randomization algorithm is serious business!

Let us look at the problem of generating permutations. Given an array `A` of n items (They might be integers, or they might be larger objects.), we want to come up with an algorithm which produces a *random permutation* of `A` on every run.

Problem 2.a. Recall the problem solving process in recitation 1. Before we come up with a solution, we should be clear about what the objectives are. What are our objectives here and what should be our metrics to evaluate how well a permutation-generation algorithm performs?

Problem 2.b. Come up with a simple permutation-generation algorithm which meets the metrics defined in the previous part. What is the time and space complexity of your algorithm?

Note: It doesn't have to be an in-place algorithm.

Problem 2.c. Does the following algorithm work?

```
for (i from 1 to n) do
  Choose j = random(1,n)
  Swap(A, i, j)
end
```

Problem 2.d. Consider the Fisher-Yates / Knuth Shuffle algorithm:

```
for (i from 2 to n) do
  Choose r = random(1, i)
  Swap(A, i, r)
```

```
end
```

What is the idea behind this algorithm? Will this produce good permutations? If so, are you able to come up with a simple proof of correctness?

Test Yourself - Optional Parts
Now let's consider a problem that can be approached by permutation-generation. There are currently 650 students enrolled in CS2040S. To handle grading such a large class without exhausting the tutors, suppose we decided to have each student grade *another* student's work. So, for PS5, we will do as follows:

1. Given a roster `A` of the class, generate a random permutation `B` of the students
2. Assign student `A[i]` to grade the homework of student `B[i]`

Problem 2.e. If you use solutions from 2.b or 2.d, what is the *expected* number of students that'll have to grade their own homework in one random permutation?

Problem 2.f. How might we modify and adapt Knuth Shuffle to this problem?

Problem 2.g. So which is the better permutation generating algorithm. What's the moral of the story here?