CS2040S: Data Structures and Algorithms

Recitation 3

Problem 1. (String Sorting)

In the "DNA Sequence" problem from the previous recitation, we discussed how to modify MergeSort to sort a string with only 2 types of characters by using only substring reversals.

Problem 1.a. Suppose now, our input string comprise of arbitrary characters without duplicates. Your new task is to devise a QuickSort-like algorithm for sorting the string but still obeying the same rules as before: (1) only substring reversals are permitted and (2) inspecting the string is free. (*Hint:* use our modified MergeSort from before to help you).

What is the recurrence? What is the running time?

every time, choose median as pivot. regard all characters larger than pivot as 'T', otherwise 'S'. Use MergeSort on the array and we get "S... pivot T...".
T(n) = 2T(n/2) + O(nlogn) -> O(n*log^2(n))

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

yes we can, but the worst case: all characters are the same

We can use 3-way partitioning to deal with it: First, we partition the original array into two parts: the less than and equal part and larger part. Then, for the less than and equal part, we partition it again into less than part and equal part

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?

each permutation has a probability of 1 / n!

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?

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?

Problem 3. (***Bonus:*** **Zoom Woes**)

Note: Your instructor will only cover this question if time permits. The solution of this problem will be included in the recitation slides for your own perusal.

Suppose Seth is giving a Zoom lecture and the chat is bursting with great questions that are streaming in. Now obviously Seth do not have the time to answer all of them, so he decided to just randomly pick **one** question to answer in the end. In addition, he wants to be fair by ensuring that every question has an equal chance of being answered. However Seth don't intend to maintain a collection of all the questions asked at the back of his head while giving the lecture – he only intends to bear one question in mind at any one moment.

Problem 3.a. How will Seth accomplish this? Realize that if he didn't pick a question the moment it was posed then the opportunity will pass because he cannot come back to pick it later. How can Seth know which question to pick (as they are streaming in) without knowing the total number of questions ahead of time?

Problem 3.b. What if he wishes to answer k random questions instead?