CS2040S: Data Structures and Algorithms

Recitation 3

Problem 1. (Everyday I'm Shuffling)

We have seen by now that 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!

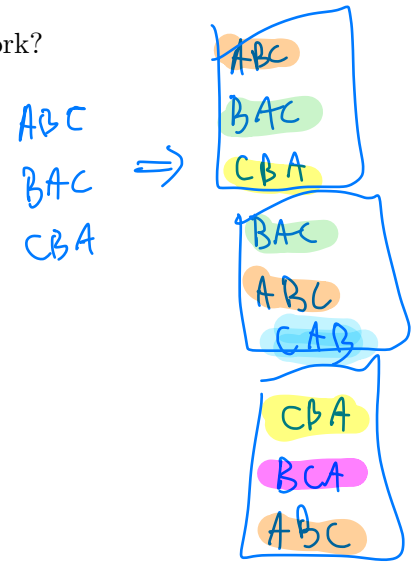
Let us look at the problem of generating permutations. Given an array `A` of n items (They might be integers, or they might be larger objects.), we want to come up with an algorithm which produces a *random permutation* of `A` on every run.

Problem 1.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 solution meets them?

Problem 1.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?

Problem 1.c. Does the following algorithm work?

```
for (i = 1 to n):
  Choose j = random(1,n)
  Swap(A, i, j)
```



Problem 1.d. Consider the following algorithm.

```
Source array A
Destination array B
for (i = 1 to n):           // Build permutation prefix from i to n
  Choose j = random(1, i-1) // Choose random position in current permutation prefix
  B[i] = B[j]               // Copy random element to end
  B[j] = A[i]               // Insert next item from A into the random slot
```

What is the idea behind this? Will this produce good permutations? If so why? If not, how do you fix it?

Problem 1.e. How can we turn the previous algorithm into an in-place one?

Now let's consider another possible application of permutations. To handle grading a 500 person 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. Generate a random permutation `B` of the students
2. Assign student `A[i]` to grade the homework of student `B[i]`

Problem 1.f. If you use solutions from 1.b or 1.e, what is the *expected* number of students that'll have to grade their own homework in one random permutation? What about the algorithm proposed in 1.d. Which is the *better* algorithm. What's the moral of the story here?

Problem 2. **(Zoom Woes)**

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

Problem 3. (Random Trees)

We saw how to generate a random permutation from an array. Now how would you generate a random binary tree?

But wait, what is a random tree?

Problem 3.a. Recall from your problem set that a *perfectly balanced* binary tree is defined to be a binary tree in which for every node, the sizes of its left and right subtrees differ by at most one. For the sake of simplicity in this problem, let's modify the definition such that both left and right subtrees under every node have exactly the same sizes.

For a perfectly balanced binary tree with n nodes, what can you say about n? Given an array `A` of n items, how do you randomly generate a perfectly balanced binary tree from the items?

Problem 3.b. Now what if you just want any random binary tree (i.e., doesn't need to be balanced). How will the algorithm look like and can you write down the time complexity recurrence for it?

Problem 3.c. Suppose the number of different binary trees you can construct with n nodes is counted by $c(n)$, express it as a recurrence relation. Try working out the answers for $c(1)$, $c(2)$, $c(3)$, $c(4)$, $c(5)$. (*Bonus:* Can you also derive the combinatorial formula for these numbers?) Do you know what is the significance of these numbers?

Problem 3.d. Consider the following strategy in constructing a random binary tree:

1. Assign each node a random number which we will call its *priority*.
2. Now build a tree that maintains priority hierarchy: the root of each subtree is the node in that subtree with the highest priority.

That should should also work, right? How would you implement it?

Problem 3.e. What is the sum of node depths in the tree from the previous part? What is the expected depth of the tree?

We shouldn't have to answer these two questions formally because something we have already seen should imply their answers. What is it?

If you are interested in more randomized tree goodness, take a look a Treaps!