Stefan Emmons

COSC-3020-01

Lab 04

10-04-2019

Part 2: Runtime analysis

Out of the three function in this lab, I have one that I know does the bulk of the work, the rest simply assign and compare. Stepping through our code in "discover" we have the following:

If (a.length <= 1): **linear base case**

var permArray = []: **constant, but shows that this is not a "in-place" algorithm**

const excludedAnchor = discover(a.slice(1)): **n-1, and is called independently of the following loops**

const firstOption = a[0]: **constant**

First for loop: **n, no recursion occurs here**

Second, nested for loop: **Also n, as no recursion occurs, but since it is this is a nested for loop sequence, our overall runtime for both loops is n^2.**

return permArray: **constant**

Given this information, we can start to build the following recurrence relation.

Recurrence relation:

$$T(n) = \begin{cases} 1 & if\ n \leq 1 \\ T(n-1) + n^2 & if\ n > 1 \end{cases}$$

Solve by substitution:

$$T(n) = T(n-1) + n^2$$

$$= T(n-1)^2 + 2n^2$$

$$= T(n-1)^3 + 3n^2$$

$$\ldots and\ so\ on, we\ can\ see\ a\ pattern\ here, so$$

$$T(n-1)^i + in^2$$

The 'i" here is the recurrence relation representation.

When considering the best case with this algorithm, the array is already sorted, and no more permutations need to be executed. This will give us a best case of $\theta(n)$. Linear and easy, no swapping needed. Our worst case, which is actually our average case as well, is going to be $\theta(n!)$. This is because even if our input array is half sorted, every single unique permutation is

going to be calculated, and then passed through condition branches to filter the variation that is correctly sorted. Since we only get unique, systemically generated permutations, we are guaranteed an upper bound of (n!).

This would not be the case if we were swapping without memory, and randomly generating permutations. The best case would still be $\theta(n)$, but our worst case could be $\theta(\infty)$. This is because we have the (slim) chance that we just keep generating random permutations that are never correctly sorted. This is the strength of systematically generating permutations. While it's still a terrible sorting method, it will stop at SOME point.