CS3000: Algorithms & Data — Summer 2023 — Laney Strange

Recitation 5
Due Tuesday June 20 @ 9pm Gradescope

Name: Laney Strange
Collaborators: the Heartbreakers

- One recitation problem each week is graded; the rest are there for practice. That problem will be graded on completeness – full credit for making an honest effort. It is also closely linked to the upcoming Long Homework, so be sure you read the feedback from your grader!
- Recitations can be written by hand, or typeset if you prefer.
- Put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This recitation is due Tuesday June 20 @ 9pm Gradescope. If you miss the in-person recitation, or need to submit your solution later than the end of your section, please fill out this form: https://forms.gle/CLrhrkVauXYzC7U57
- Collaboration is strongly encouraged during recitation! Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.

Problem 1. *Randomization (graded)*

Recall that one of the ways to randomize an algorithm that operates on an array is to ”shuffle” the array – i.e., randomly permute it

(a) Below is an approach to shuffling that does not generate all possible permutations with equal likelihood. For an array of size 3, such as $A =< 1, 2, 3 >$, there should be 6 permutations, each equally likely. Describe what this algorithm does instead, using A as an example.

```
SHUFFLE(A)
for i = 1 to A.length
    r = random integer in range [1..n] inclusive
    swap A[i], A[r]
```

Solution:

(b) Here’s another attempt at shuffling. The problem with this version is that not all permutations are generated *at all*, without even worrying about equal probability. Starting with our same input array $A =< 1, 2, 3 >$, given an example of a permutation this code would never produce.

```
SHUFFLE(A)
for i = 1 to A.length − 1
    r = random integer in range [i + 1..n] inclusive
    swap A[i], A[r]
```

Solution:

Problem 2. *NP Complete Problems(not graded; for practice)*

We know that the Traveling Salesman Problem (TSP) is NP-Complete, i.e., the only known way to solve it is brute-force, which runs in $O(n!)$ time.

(a) Suppose you're running TSP on a computer that executes 1,000,000 operations per second. We'll assume that computing the cost of one permutation of cities takes one operation. How long would TSP take on each of these input sizes?

n	$n!$	Wall-clock time
4	24	
8	40320	
12	479001600	
14	87178291200	
16	20922789888000	
18	6402373705728000	
20	2432902008176640000	

Solution:

(b) Obviously, a company like FedEx, which needs to find an efficient way to get to all its required stops, is not going to wait that long just to figure out how to visit 20 addresses. Companies like that rely on approximations. What are some ideas for how you might be able to get a reasonable solution for 'real life', even if it's not optimal?

Solution: