CMPS 2200 Assignment 4

In this assignment we'll look at randomness in computation, both in theory and in practice.

As with previous assignments, your code implementations will go in `main.py`. Please add your written answers to `answers.md` which you can convert to a PDF using `convert.sh`. Alternatively, you may scan and upload written answers to a file names `answers.pdf`.

Part 1: Deviation from Expectations

We learned in lecture that Quicksort takes $O(n \log n)$ expected work. A fair question is how tight that expectation is. Luckily we have some bounds that allow us to look at this question. For a random variable X, Markov's inequality states that:

$$\mathbf{P}[X \geq \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}$$

1a) What is the probability that Quicksort does $\Omega(n^2)$ comparisons? **Hint:** Let X be a random variable which outputs the amount of work done by Quicksort. Then to calculate the probability that the amount of work performed by Quicksort is $\Omega(n^2)$, you should decide what to set α to and what $\mathbf{E}[X]$ should be. .

.

Enter answer in `answers.md` .

.

1b) What is the probability that Quicksort does $10^c n \lg n$ comparisons, for a given $c > 0$? What does this say about the deviation of the actual work from the expected work for Quicksort? **Hint:** Once again, use Markov's inequality deciding what α and $\mathbf{E}[X]$ should be. .

.

Enter answer in `answers.md` .

.

Part 2: From "Maybe" to "Definitely"

At your new job designing algorithms for really hard problems, you're put to work solving problem X. Your predecessor has left you with an algorithm $\mathcal{A}$ for problem X that has a deterministic worst-case work of $O(w(n))$, but only produces the correct output with a certain probability of success. Moreover, we can also verify whether the correct result was produced with $O(v(n))$ work. $v(n) \in O(w(n))$.

Let $\mathcal{A}(\mathcal{I})$ denote the output of an algorithm $\mathcal{A}$ on input $\mathcal{I}$. So $\mathcal{A}(\mathcal{I})$ has a probability of ϵ of being correct and a failure probability of $1 - \epsilon$. Furthermore let $\mathcal{C}(\mathcal{A}(\mathcal{I}))$ denote the output of (deterministically) checking $\mathcal{A}$'s solution.

2a) You find that ϵ is too small to be reliable. You want to be able to have *any* guaranteed success probability δ, for $\epsilon < \delta < 1$. Use $\mathcal{A}$ to construct an algorithm $\mathcal{A}'$, where $\mathcal{A}'(\mathcal{I}, \delta)$ is the correct output with probability δ. It is sufficient to give a high level description of $\mathcal{A}'$. What is the work of $\mathcal{A}'$ in terms of n, δ, and ϵ? **Hint**: Each run of $\mathcal{A}$ is independent and does not depend on previous runs.

.

.

Enter answer in `answers.md` .

.

2b) Your boss and co-workers are impressed, but you want to do even better. Show how to convert $\mathcal{A}$ into an algorithm that always produces the correct result, but has an expected runtime that depends on $w(n)$ and a success probability ϵ.

.

.

Enter answer in `answers.md`

Part 3: Determinism versus Randomization in Quicksort

In lecture we saw that adding a random choice of pivot reduced the probability of worst-case behavior for any given input in selection. Let's look at how pivot choices affect Quicksort. For this question, refer to the code in `main.py`

3a)

Complete the implementations of `qsort` and `compare_sort` stubs. Feel free to take from code given in the lectures to help you perform list partitioning. Note that the pivot choice function is input to `qsort`, so you will have to curry `qsort` to test different methods of choosing pivots. Implement variants of `qsort` that correspond to selecting the first element of the input list as the pivot, and to selecting a random pivot. .

.

.

.

3b)

Compare running times using `compare-qsort` between variants of Quicksort and the provided implementation of selection sort (`ssort`). Perform two different comparisons: a comparison between sorting methods using random permutations of the specified sizes, and a comparison using already sorted permutations. How do the running times compare to the asymptotic bounds? How does changing the type of input list change the relative performance of these algorithms? Note that you may have to modify the list sizes based on your system memory; compare at least 10 different list sizes. The `print_results` function in `main.py` gives a table of results, but feel free to use code from Lab 1 to plot the results as well.

Enter answers in `answers.md`

3c)

Python uses a sorting algorithm called *Timsort*, designed by Tim Peters. Compare the fastest of your sorting implementations to the Python sorting function `sorted`, conducting the tests in 3b above.

Enter answers in `answers.md`