Homework 1

COMP221 Spring 2024 - Suhas Arehalli

Complete the problems below. This assignment is out of 100 points, and the point values of each question. Note that point values are roughly inversely correlated with expected difficulty, so

- You may complete the implementation section using your choice of Python, C, or Java.
- Write up your solutions to all parts of this homework using some typesetting software (LaTeX/Overleaf is recommended, but any typeset PDF is acceptable).
- You'll submit this assignment through Github Classroom (link on Moodle), with your written solutions in a pdf named "[LASTNAME1]_[LASTNAME2]_Homework1.pdf".
- **The due date for this assignment will be posted on the course website.**
- **You may submit assignments in pairs.** Put both peoples' names at the top of your submission.
  - **Both people are expected to have contributed to all parts of the assignment** – work together, and make sure you and your partner are both convinced of your solutions.
  - If you discussed problems with other students, list their names at the top of your assignment (there will be no penalty for this — it's encouraged!).
  - However, don't look at other group's written solutions/code, or share your written solutions/code with other groups. Consult the syllabus for more details on academic integrity policies.

Note that we may not have covered all of the topics involved in this assignment at the point of release. This is to give you time to complete the parts you can as early as you can. I recommend you make note of parts we haven't covered yet so you can come back to them.

1 Implementation (*40pts*)

1. **Preliminaries**

   In class we talk primarily about formally bounding the runtime of our algorithms. I sometimes gesture at the fact that the asymptotic bounds we prove are relevant in practice, but this problem, and related future problems, will hopefully convince you that this is true.

   Familiarize yourself with standard functions for measuring real time passing in your language of choice (this will be something like System.nanotime() in java, or the time or timeit modules in python, or clock() in C). Make sure you're aware of the units you're measuring in (microseconds, nanoseconds, CPU cycles, etc.) — we'll ask you to convert those to *nanoseconds* for consistency. Also familiarize yourself with random number generation mechanisms in your language.

(a) Write a function to generate a random "array" of a numeric type of length n (if you're using python, just treat the list primitive as if it were an array). Whether these are integers or floats/doubles should not matter.

(b) Write a function that, for a particular sorting function and array, time how long it takes to sort in nanoseconds.

(c) Write a function that, for a particular n, k, and sorting function, will record how long it takes to sort a random array of length n k different times. That is, this function should return k different integers, each representing how long a particular random problem instance took to sort using the provided algorithm.

Try to make your methods as *general* as you can, so that you can use your functions to easily test a variety of different sorting functions.

2. **Implement BubbleSort** (*20pts*)

   Take a look at the BubbleSort algorithm described in Alg. 1.

Algorithm 1 Bubblesort

```
function BubbleSort(Array A)
    for i ← 1 to N do
        for j ← 1 to N − i do
            if A[j] > A[j + 1] then
                Swap(A[j], A[j + 1])
            end if
        end for
    end for
    return A
end function
```

Implement this algorithm in your language of choice, making sure you test for correctness (try and practice good software engineering principles: unit tests, etc.).

Use your timing methods to record 10 sample sort times for Ns = 50, 100, 500, 1000, 5000, 10000. Record this in a *csv* (comma separated value) file with columns for N, *sample_num*, and *time_ns* representing the N for the list, which of the 10 samples for this N you're recording, and the time in nanoseconds it took you to sort a random array of length N. See the homework release website for a sample csv file to see the format your code should generate. Your language of choice may have built-in libraries to make generating csvs easier (i.e., the csv library in Python). **Name this file "[LASTNAME1]_[LASTNAME2]_bubblesort_times.csv" as part of your submission.**

3. **Implement MergeSort** (*15pts*) Consider MergeSort, which is discussed both in class and in the textbook. Pseudocode for MergeSort is provided in Alg. 2

   Implementing this function might be a bit trickier than BubbleSort, but such is the cost of performance! There are a number of things in this pseudocode that make tricky things look simpler than they are, so make sure to proceed cautiously and set up tests for yourself to make sure the code is doing what you think it's doing.

Algorithm 2 Pseudocode for MergeSort

```
function MergeSort(Array A)
    if N ≤ 1 then
        return A
    end if
    midpoint ← ⌈N/2⌉
    L ← MergeSort(A[1...midpoint])
    R ← MergeSort(A[midpoint + 1...N])
    return Merge(L, R, N)
end function

function Merge(Array L, Array R, Int N)
    C ← Array[1 . . . N]
    i ← 1
    j ← 1
    for k ← 1 to N do
        if L[i] < R[j] or j out-of-bounds then
            C[k] ← L[i]
            i ← i + 1
        else (or i out-of-bounds)
            C[k] ← R[j]
            j ← j + 1
        end if
    end for
    return C
end function
```

After you have this implemented, do the same timing exercise! **Record 10 sample sort times for Ns = 50, 100, 500, 1000, 5000, 10000**. Record this in a *csv* (comma separated value) file with columns for N, *sample_num*, and *time_ns* representing the N for the list, which of the 10 samples for this N you're recording, and the time in nanoseconds it took you to sort a random array of length N. **Name this file "[LASTNAME1]_[LASTNAME2]_mergesort_times.csv" as part of your submission.**

4. **Sanity Check** (*5pts*)

   Look at the way the runtimes scale with N. What would you expect to see, given the Big-Oh analyses you saw in class? Do differences emerge at this scale? If things look weird, try running your code a few times with larger Ns.

2 Algorithmic Analysis

2.1 Building Back to Basics (*30pts*)

1. **Ignoring coefficients** (*15pts*): In COMP128, we learned that we can drop constant coefficients in time complexities I all we care about is Big-Oh. Formally, a (simplified) version of this can be written formally as if $f(n) \in \Theta(g(n))$, $kf(n) \in \Theta(g(n))$ for $k > 0$. Prove this to be correct using the c, n_0 definition!

   HINT: *If the method for doing this is seems weird, try and generalize from more concrete examples. Is $3n \in O(n)$? Is $3n^2 \in O(n^2)$? What about $9n^2$? Can you see a way to construct the c and n_0 to prove $9n^2 \in O(n^2)$ from the c' and n_0' that show $3n^2 \in O(n^2)$ and the fact that $9n^2 = 3(3n^2)$?*

2. **Counting for Quadratics** (*10pts*) In class we talked about how developed all of our Big-Oh formalisms so we could be adequately lazy in talking about time complexity. In class, I was often sloppy when counting simple operations when proving the time complexities of n^2 sorts. Formally, if we ignore the loops, the number of time steps in the inner and outer loops of these kinds of nested-loop algorithms creates time complexities of the form

   $$f(n) = a_1 n^2 + a_2 n + a_3$$

   where $a_1, a_2, a_3 \in \mathbb{Z}$ (i.e., they're integers) and $a_1 > 0$, It's a quadratic function with a postive n^2 term! I'll claim that if I'm sloppy with counting operations, this will affect the values of a_1, a_2, and a_3, but won't result in a time complexity growth function of a different form (convince yourself this is true!). **Prove that for *any* integers a_1, a_2, a_3 that $f(n) \in \Theta(n^2)$ using the c, n_0 definition of big-O and big-Ω.** You may use either definition of big-Θ we discussed. If helpful, you may assume that $f(n) > 0$ for all $n > 0$.

   HINT: *Try and get the quadratic to be something easily factorable. Try by cases: If $a_3 = 0$, then some factoring tricks should make the inequality solve-able! If $a_3 \neq 0$, you'll need 2 tricks: One you saw in the insertion sort example in class. For the other, it might be helpful to observe that if $n > 1$ and some number $k < 0$, then $kn \leq k$. See if you can find a variant of this trick!*

3. **Polynomials of Higher Degree** (*5pts*) Now show that with

$$\begin{aligned} f(n) &= \sum_{i=0}^{k} a_i n^i \\ &= a_k n^i + \cdots + a_1 n + a_0 \end{aligned}$$

$f(n) \in O(n^k)$ by the c, n_0 definition. Note I'm only asking for big-0, not big-Θ!

HINT: We'll need a slightly different strategy (one more general than the small bounding trick I suggested for quadratics). Consider $c = \sum_{i=0}^{n} |a_i|$. Can you see a way to show cn^k *must* be $\geq f(n)$ for all $n \geq$ some n_0?

2.2 Algorithm Design (*29pts*)

1. **Recursive Linear Search** (*16pts*)
   (a) Write pseudocode for a recursive linear search algorithm called RecursiveLinearSearch. This algorithm should take in an array A and some element e and return either an index i or $NULL$. If it returns an index i, the algorithm is correct if $A[i] == e$. If it returns $NULL$, $e \notin A$. (*8pts*)
   (b) Prove the correctness of this algorithm. Refer to the lecture notes for the problem specification for search we're using. (*8pts*)
   Hint: Use induction.

2. **Don't Divide, But Do Concur** (*13pts*)
   (a) Write pseudocode for an algorithm, IntDivision, that performs *integer division*. Formally, we take as arguments an integers $a, b > 0$ and we want to return integers m, r such that $a = bm + r$ and $0 \leq r < b$. Note that r (the remainder) is the same as $a \% b$ and $m = \lfloor \frac{a}{b} \rfloor$. Unfortunately our model computer for this problem has been un-*mod*-ded, so there are not modulus operations (or floor/division operations, for that matter) for your pseudocode to use! (*6pts*)
   (b) Prove your algorithm is correct. For this problem, assume correctness means that the returned m, r satisfy $a = mb + r$ with $0 \leq r < b$ (*5pts*)
   (c) Modify your algorithm to work for $a \leq 0$ (though still assume $b > 0$!). Prove it's correctness. (*2pts*)

3 A Small Puzzle *(1pt)*

Forget about computers and start thinking about game strategies.

A devious gambler approaches you with 8 balls. You are told that 1 of them is slightly heavier than the rest, but this difference can only be discerned by a balance that can compare the weights of two subsets of the balls and tell you which (if either) is heavier. A naive variant of a binary search can solve this problem in 3 weighings (convince yourself of this before moving forward!).

However, this gambler doesn't want you to have certainty (they're a gambler, after all!), and so wagers k dollars (a large sum of money!) that you can't select the right ball with only 2 weighings. You can see that with naive binary search, each weighing removes half of the possible options, and

so two weighings would leave your odds at 50-50. With an even wager, your expected winnings are $0!

Provide a strategy that guarantees victory under these conditions (i.e., with only two weighings). Argue that it is always correct (The formality of a proof isn't necessary, but the argument should be convincing!).