Analysis of Algorithms
CS 375, Fall 2022

Problem Set 2

Due **AT THE BEGINNING OF CLASS** Monday, October 24

- For this assignment, standard file naming conventions apply: Please submit your typewritten answers in a PDF file named `CS375_PS2_<userid>.pdf` where `<userid>` is replaced by your full Colby userid, and submit it to your `SubmittedWork` folder. Please reach out to me right away with any questions or concerns about this!
- From your textbook (CLRS), please read: Chapters 2.1, 2.2, and 3. **NOTE:** Some material there is more mathematically intense than we need—don't worry about those parts!—but please do read over the parts that give basic ideas behind content covered in lecture (e.g., O, Ω, Θ notation, functions' orders of growth, loop invariants).
- **IMPORTANT:** Some of these exercises may build upon topics covered in our Oct. 17 class meeting; they are included here "early" so you can see all of the exercises on this assignment.
- Please recall the essential, general style guidelines for writing algorithms in CS375—including restrictions on `break` and other flow-of-control statements—as presented on PS1. Unless otherwise specified, they apply for all work in CS375.
- *A general note for CS375:* When writing up your homework, please present your answers neatly and **explain your answers clearly**, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or hard to understand solutions. Clear communication *is* the point, on every assignment.

  In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Exercises

1. CLRS, Exercise 2.1-3 (page 22)—with slight modifications / clarifications:
   - As stated in the CLRS exercise, write pseudocode for linear search that meets the given specifications. (Please recall CS375 instructions not to use `break` or similar flow-of-control statements in your algorithms.)
   - As stated in the CLRS exercise, give a correctness argument for your pseudocode using a loop invariant that you create specifically for that purpose. Be sure to clearly state your loop invariant.
   - Then, follow the three steps shown in lecture notes to help explain the correctness of your linear search algorithm:
     (a) Give a very short and convincing explanation of how the loop invariant is true before the first iteration of the loop.

(b) Give a concise and convincing explanation of how your pseudocode ensures the invariant is true after each successive iteration. Refer directly to the pseudocode, citing specific lines of pseudocode in your explanation.

(c) Give a concise and convincing explanation of how the algorithm meets its specifications. Refer specifically to both the invariant property and the specifications as part of this explanation—referring to specifications is essential for establishing algorithm correctness!

Your loop invariant should be created so that it gives helpful information about what is known to be true when the loop is finished—please be sure to use that information in your correctness argument!

2. Consider this pseudocode algorithm for the sorting method *Selection Sort*:

```
SELECTIONSORT(A[1...n])
1.  for i = 1 to length[A] − 1
2.      min = i
3.      for j = i + 1 to length[A]
4.          if A[j] < A[min]
5.              min = j
6.      // the next 3 lines swap A[i] and A[min], using a temporary variable
7.      temp = A[i]
8.      A[i] = A[min]
9.      A[min] = temp
```

Here's a proposed loop invariant for the outer **for** loop in this algorithm:

> Subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order, and $A[i..n]$ consists of the remaining values of A (no constraint on order).

Recall that by convention, an array or a sequence for which the specification has a first index that is greater than the last, such as $A[n+1..n]$ is considered to be empty.

Working with that proposed loop invariant for the outer for loop of SELECTIONSORT, follow the three steps shown in lecture notes (and in Exercise 1) to explain the correctness of SELECTIONSORT. Your answers for those three parts do not need to be lengthy answers—a few sentences each could be enough, as long as those sentences contain the key details.

> **Hint:** For the first of the three parts (*a very short and convincing explanation of how the invariant is true before the first iteration of the loop*), in this case, note that before the first iteration, i is 1. What can be said about subarray $A[1..i-1]$?

Also, recall the specification for the *sorting problem*, also repeated here for convenience:

- *Input*: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$
- *Output*: Permutation (reordering) $\langle b_1, \ldots, b_n \rangle$ of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$.

This specification is the same as the one presented in lecture notes, capturing our intuitions of what it means for a list to be sorted. Please see me if there are questions about it, or about any part of the exercise!

3. **Anagrams!** Prof. Ram Sagan at the Portland Institute of Technology (which does not exist) was recently listening to these song lyrics (which actually do exist!)

   Miracles will have their claimers
   More will bow to Rome ...

   (Lyrics written by Neil Peart. From track 8 on the album *Presto*, by the band *Rush*.)

   and was suddenly inspired to ask you for help with the *Anagrams* problem!

   Our definition of an *anagram* is:

   > *Definition*: Two strings `s` and `t` are *anagrams* exactly when `s` is formed from the same letters as `t`, used the same number of times as in `t`.
   >
   > For examples, "miracles" and "claimers" are anagrams, as are "more" and "rome." But "secures" is not an anagram of "rescue"—even though "secures" uses the same letters as "rescue" (the letters "c", "e", "r", "s", and "u"), it does not use them the same number of times, because "secures" uses "s" twice and "rescue" uses "s" only once. Additionally, "comic" is not an anagram of "cosmic", because the two words do not use exactly the same letters—"cosmic" uses an "s" but "comic" does not.

   Using that definition, here are specifications of the Anagrams problem:

   - *Input*: strings s, t of lowercase letters from the alphabet for English (i.e., the 26 letters of the Latin / Roman alphabet). Assume there are no spaces, punctuation, capital letters, etc. in the strings—only lowercase letters. For complexity analyses, let m stand for the length of s, and let n stand for the length of t.
   - *Output*: True if s is an anagram of t, False otherwise.

   To help Prof. Sagan, answer the following questions:

   (a) What simple check would you do before checking any arrangements of letters? Your answer should be something that is constant time (in fact, it must be constant time in Java) but still very helpful for this problem! In what way(s) would it help with all of the algos you give below?

   (b) Then, give a brute force algorithm to solve this problem that checks all permutations of either s or t. What is its asymptotic time complexity, expressed as a Θ bound?

   (c) Then, give a $\Theta(n^2)$ algorithm to solve this problem.

   (d) Then, give a $\Theta(n \lg n)$ algorithm to solve this problem.

   (e) Then, give a $\Theta(n)$-time, $O(1)$-space complexity algorithm to solve this problem. (*Hint:* You may want to use some auxiliary data structure(s) for this algorithm, but you'll only need $O(1)$ additional space.)

As is conventional, we're asking about worst case time complexity unless otherwise specified.

To earn full credit, be sure to include all of the following for each of the four algorithms:

- Pseudocode, accompanied by a brief English explanation of what the algorithm does
- A concise but convincing explanation of correctness
- A concise but convincing explanation of its time complexity
- A concise but convincing explanation of its space complexity

Please be sure to document any important assumptions made or used in your complexity analyses!

Hint: You may notice that the complexities of the algorithms above are given as functions of only n, rather than as functions of both m and n. The "simple check" as the answer to part 3a makes it sensible to do this! If you don't come up with that particular "simple check" or don't see why these complexities can be given as functions of only n, you are welcome to give a $\Theta(mn)$ algorithm for part 3c, a $\Theta(m \lg m + n \lg n)$ algorithm for part 3d, and a $\Theta(m + n)$-time, $O(1)$-space algorithm for part 3e ... or, even better, you can ask me or a TA for help!

Optional Exercise

The exercise below is one that I think is a good learning experience, if you'd like to do it. **Do not submit this exercise. It will not be counted for a grade.** If you're looking for a little more experience with a different take on complexity analysis, however, it could be worthwhile for you to work on it—and I'll be happy to work with you on it or answer your questions, just let me know!

- CLRS, Exercise 2.2-3 (page 29).

  This exercise asks you to consider an *average case* time complexity analysis, as well as worst-case analysis. Answer all of the questions posed in the exercise, not just the last question involving Θ notation, and as always, please give short but convincing explanations of your answers.

  The application of average case analysis to this exercise is meant to be relatively intuitive—please read the chapter about it (pages 27–28), and if there are questions about average case analysis, please feel free to ask me!