COMPSCI 311: Introduction to Algorithms

Daniel Chin

Challenge Problems 1

due 9/17/2023 at 11 : 59pm in Gradescope

Instructions. Limited collaboration is allowed while solving problems, but you must write solutions yourself. List collaborators on your submission.

You can choose which problems to complete, but must submit at least one problem per assignment. See the course page for information about how challenge problems are graded and contribute to your homework grade. Since you don't need to complete every problem, you are encouraged to focus your efforts on producing high-quality solutions to the problems you feel confident about. There is no benefit to guessing or writing vague answers.

If you are asked to design an algorithm, please (a) give a precise description of your algorithm using either pseudocode or language, (b) explain the intuition of the algorithm, (c) justify the correctness of the algorithm; give a proof if needed, (d) state the running time of your algorithm, (e) justify the running-time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

Problem 1. Stable Matching Running Time. In class, we saw that the Propose-and-reject algorithm terminates in at most n^2 iterations, when there are n students and n colleges.

1. It seems possible that the algorithm may complete in fewer rounds if the preference lists have a certain structure. Describe how to construct preference lists for any number n of colleges and students, such that the propose-and-reject algorithm will complete in only $O(n)$ rounds when run on these preference lists.

   The propose and reject algorithm would run in O(n) rounds when proposals are accepted within the first or second round. For example, if all students prefer different colleges and all colleges prefer different students, and they are matched with their first choice, then it would be O(n). Similarly, it would be O(n) if the proposals were accepted within the second round because it would be closer to a runtime of n, 2n, 3n... As the number of rounds increase it would approach $O(n^2)$.

2. Could it be the case that the running time is actually $O(n)$ for all preference lists? Show that this is not true by designing preference lists so that the number of rounds of the algorithm is $\Omega\left(n^2\right)$.

   The running time is not O(n) for all preferences lists because as more and more proposal rounds go on, the run time approaches $\Omega(n^2)$. Students may need to propose to up to n different colleges before being matched. This would lead to a worst case scenario of $\Omega(n^2)$.

Problem 2. Stable Matchings Consider a version of the stable matching problem where there are n students and n colleges as before. Assume each student ranks the colleges (and vice versa), but now we allow ties in the ranking. In other words, we could have a school that is indifferent two students s_1 and s_2, but prefers either of them over some other student s_3 (and vice versa). We say a student s prefers college c_1 to c_2 if c_1 is ranked higher on the s 's preference list and c_1 and c_2 are not tied.

1. Strong Instability. A strong instability in a matching is a student-college pair, each of which prefer each other to their current pairing. In other words, neither is indifferent about the switch. Does there always exist a matching with no strong instability? Either give an example instance for which all matchings have a strong instability (and prove it), or give and analyze an algorithm that is guaranteed to find a matching with no strong instabilities.

   There does not exist a perfect matching such that there is no strong instability. For colleges c_1 and c_2 and students s_1 and s_2, assume that both s_1 and s_2 prefer c_1 over c_2. Let c_1 prefer s_2 and c_2 prefer s_1. The possible matchings would be $(s_1, c_1) and (s_2, c_2)$ and $(s_1, c_2) and (s_2, c_1)$. These both create strong instabilities, which means there does not always exist a matching with no strong instabilities.

2. Weak Instability. A weak instability in a matching is a student-college pair where one party prefers the other, and the other may be indifferent. Formally, a student s and a college c with pairs c' and s' form a weak instability if either

- s prefers c to c' and c either prefers s to s' or is indifferent between s and s'.
- c prefers s to s' and s either prefers c to c' or is indifferent between c and c'.

Does there always exist a perfect matching with no weak instability? Either give an example instance for which all matchings have a weak instability (and prove it), or give and analyze an algorithm that is guaranteed to find a matching with no weak instabilities.

It is possible to find a perfect matching such that there is no weak instability. This is done by modifying the Gale Shapely algorithm so that there is a way

to break ties. This can be done by adding rules so that parties always have preferences over other options.There cannot be a weak instability where s prefers c to c' because s would have been rejected by c' earlier in the proposals.

Problem 3. Asymptotics Given an array A of n integers, you'd like to output a two-dimensional $n \times n$ array B in which $B[i, j] = \max\{A[i], A[i+1], \ldots, A[j]\}$ for each $i < j$. For $i \geq j$ the value of $B[i, j]$ can be left as is.

```
for $i=1,2, \ldots, n$
    for $j=i+1, \ldots, n$
        Compute the maximum of the entries $A[i], A[i+1], \ldots, A[j]$.
        Store the maximum value in $B[i, j]$.
```

1. Find a function f such that the running time of the algorithm is $O(f(n))$, and clearly explain why.

   The outer loop runs from $i - n$, which is $O(n)$, and the inner loop runs $j - (n - 1)$, which is also $O(n)$. And each time the inner loop runs, it needs to find the maximum value of the entries, which also runs in $O(n)$, which means that the algorithm runs in $O(n^3)$.

2. For the same function f argue that the running time of the algorithm is also $\Omega(f(n))$. (This establishes an asymptotically tight bound $\Theta(f(n))$.)

   This algorithm is also $\Omega(n^3)$ because the inner loop will run at least $\Omega(n^2)$ times, and finding the maximum would need $j-i+1$ comparisons ($\Omega(n)$). Thus it is also $\Omega(n^3)$ and creates an asymptotically tight bound of $\Theta(n^3)$.

3. Design and analyze a faster algorithm for this problem. You should give an algorithm with running $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

   We can make a faster algorithm that runs in $O(n^2)$ by determining the maximum value at each iteration of the inner loop. This would mean storing the current maximum value in a variable, then as we iterate through the array, then we can store it in B.

   ```
   for $i=1,2, \ldots, n$
       let currMax = A[i]
       for $j=i+1, \ldots, n$
           currMax = max(currMax, A[j])
           B[i, j] = currMax
   ```

   The outer loop runs in $O(n)$, as does the inner loop, and we do not need to iterate through the previous entries which makes the runtime $O(n^2)$

Problem 4. Highest Safe Rung. You want to stress test glass jars. You have a ladder with n rungs, and want to find the highest rung from which you can drop a jar and not have it break. We call this the highest safe rung.

Your goal is to find the highest safe rung with the fewest number of drops possible. However, you have a limited supply of jars, and need to find the highest safe rung before breaking all of them.

For example, with one jar you could drop it from the first rung, then the second, then the third, and so on, until it breaks, and you are guaranteed to find the highest safe rung with at most n drops. With any other strategy, you would risk breaking the jar before finding the highest safe rung.

Now, suppose you have two (identical) jars, so you can break one and still find the highest safe rung. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. (In other words, it should be the case that $\lim_{n\to\infty} f(n)/n = 0$.)

We can start by dividing the rungs into groups of approximately $\sqrt{n}$ rungs. Then we will use one of the jars to test which group the highest safe rung belongs to. We will do this by dropping the jar at the highest rung of each group. Once it breaks, we will then get the upper bound of our range. The highest safe rung will be between the rung where the first jar broke, and the bottom-most rung in that group. We can then start dropping the other jar from the bottom up inside of the group. Once the second jar breaks we will then know the highest safe rung. This algorithm runs in $\sqrt{n}$ time. $\lim_{n\to\infty} \sqrt{n}/n = 0$