Homework 1

CS 4104 (Spring 2021)

Assigned on Wednesday, January 27, 2021.
Submit a PDF file containing your solutions on Canvas by the beginning of class on Wednesday, February 3, 2021.

My team-mate is: Joseph McAlister

Problem 1 (5 points) If a potential solution to the Stable Matching problem is a perfect matching but is not stable, then the number of rogue couples must be at least two. Just say "True" or "False". You do not have to provide any reasoning,

Solution: False

Problem 2 (10 points) In any input to the Stable Matching problem with n women and n men, what is the number of perfect matchings? Prove your answer.

Solution: The number of perfect matchings with n men and n women will be n.

proof:

Recall that a perfect matching is one in which each man is paired with exactly one woman and vice versa.

Suppose the set S of pairs returned by the proposed Gale-Shapley algorithm is of size $n-1$ for an input of shape $n \times n$.

Then there exists some man and woman m, w who are free in the input after termination.

However, in the Gale-Shapley algorithm, each free man proposes to every woman who he has not already proposed to, and each woman w becomes engaged to the proposing man m if she is not already engaged to another man $m' > m$.

So w must be engaged to m, contradicting our assumption that m, w were free after termination.

Therefore the number of perfect matchings will be n. ■

Problem 3 (25 points) Consider the following algorithm to determine if a given positive integer n is prime. For each integer i between 2 and $\lfloor\sqrt{n}\rfloor$, check if i is a factor of n, i.e., if dividing n by i leaves a remainder of 0. If the answer is "yes" for any i, return that n is composite. Otherwise, return that n is prime. Answer the following questions about this algorithm.

(a) (5 points) Let us use $f(n)$ to denote size of the input to this problem. What is $f(n)$? Is $f(n) = 1$ (because there is only number in the input)? Is $f(n) = n$ (because the value of the input is n)? Is it something else entirely? In the last case, write down what you think this alternative is.

Solution: If we assume that the algorithm takes the number n as it's input (e.g. `is_prime(n)`) then $f(n) = 1$.

(b) (10 points) What is the running time $g(n)$ of the algorithm in terms of the input size? You may assume that we can check whether if i is a factor of n in $O(1)$, i.e., constant, time.

Solution: $O(g(n)) = O(\sqrt{n})$ since the algorithm consists of iterating over each number $i \in [2, \sqrt{n}]$ and checking if i is a factor of n in constant time which is essentially $\sqrt{n}$ iterations for Big-O purposes involving sufficiently large n.

(c) (10 points) Prove that $g(n) = \Omega(f(n)^d)$ for any positive value d. In other words, prove that the running time of this algorithm for testing primality is lower bounded by any arbitrary polynomial function of the input size. You can start from any statement in the textbook or the slides but be sure to explain how you derive your conclusion from the starting statement.

Solution:
proof
Since $f(n) = 1$, $f(n)^d = 1, \quad \forall d \in \mathbb{R}^+$.
Recall that $O(g(n))$ is $O(\sqrt{n})$ which is greater than or equal to $\Omega(1)$ for all inputs.
Therefore, $g(n)$ is $\Omega(f(n)^d)$. ■

Note: It is likely that you will be able to prove part (c) only if you answer part (a) correctly.

Problem 4 (20 points) Solve exercise 5 in Chapter 2 (page 68) of "Algorithm Design" by Kleinberg and Tardos. In addition to what is stated in the problem, assume that $f(n)$ and $g(n)$ are increasing functions of n. If you decide that a statement is true, provide a short proof. Otherwise, provide a counterexample. *Note:* If you think one of the statements is true, you should not prove it for your own choices of $f(n)$ and $g(n)$. Your proof must hold for *any* pair of increasing functions $f(n)$ and $g(n)$ where $f(n)$ is $O(g(n))$.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$
Solution: True
Since f and g are increasing and $f(n)$ is $O(g(n))$, the statement holds for all n where $c \geq 1$, $n_0 > 2$, such that $O(\log_2 f(n))$ is $O(c \log_2 g(n))$.

(b) $2^{f(n)}$ is $O(2^{g(n)})$
Solution: False
Let $f(n) = 10n$ and $g(n) = n$.
Note that no constant $c > 0$ can be multiplied with $g(n) = n$ such that $c2^{g(n)} > 2^{f(n)}$
Explicitly: $2^{10n} > c2^n, \forall n, c$ with $n_0 = 1$.
Although both functions f and g can be increasing functions with $f(n)$ being $O(g(n))$, e.g. $f(n) = 10n$ and $g(n) = n$, $2^{f(n)}$ is not $O(2^{g(n)})$.

(c) $f(n)^2$ is $O(g(n)^2)$
Solution: True
Since $f(n)$ is $O(g(n))$, we know that there exist some constants $c > 0$, $n_0 \geq 0$ such that $f(n) \leq cg(n)$
Therefore, $f(n)^2 \leq c^2 g(n)^2$ also holds for all c, n of the same constraints.

Problem 5 (40 points) Solve exercise 8(a) in Chapter 2 (pages 69-70) of "Algorithm Design" by Kleinberg and Tardos. This problem involves stress-testing jars by throwing them off ladders by breaking as few bottles as possible.

(a) Suppose $k = 2$, describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ time, for some function $f(n)$ that grows slower than linearly.
Solution:
Of the sublinear Big-O function classes, the next slowest class is the fractional power class: $O(n^c)$ where $0 < c < 1$. For simplicity sake, we can choose $c = \frac{1}{2}$ such that our target is $O(\sqrt{n})$ which satisfies the sub-linear time requirement.
With this in mind, we can drop jars at increments of $\lfloor \sqrt{n} \rfloor$, starting from the bottom of the ladder and moving upwards until a jar breaks.
In this way, we are able to skip increasingly large intervals of rungs (meeting the sub-linear requirement) until we reach some rung $i\lfloor \sqrt{n} \rfloor$ where the jar breaks.
From this position, we can continue to drop the remaining jars from the last safe rung $(i-1)\lfloor \sqrt{n} \rfloor + 1$, and working our way upwards until our 2nd jar breaks at which point we know that the rung immediately prior was the highest safe rung.
Alternatively, if we reach the top rung without breaking our last jar, then that is our highest safe rung.

```
   **Input:** the number of rungs on the ladder $n$
   **Output:** the highest safest rung
 1 j ← 0;
 2 i ←j $\lfloor\sqrt{n}\rfloor$
 3 while $i < n$ do
 4     drop a jar
 5     if jar breaks then
 6         if $j = 0$ then
 7             return $j$ ;                                      /* the jars are weak */
 8         end
 9         $i \leftarrow (j-1)\lfloor\sqrt{n}\rfloor$;                   /* move back to the last, safe rung */
10         while jar not broken do
11             drop a jar ;    /* find highest rung between the last safe, and break point */
12             if jar breaks then
13                 return $i - 1$
14             end
15             $i \leftarrow i + 1$
16         end
17     end
18     $j \leftarrow j + 1$
19     i ←j $\lfloor\sqrt{n}\rfloor$
20 end
21 return $n$ ;                                   /* sturdy jars:  highest rung is safe */
```

Formally, an algorithm for this process is presented above, along with proofs of correctness and sublinear runtime complexity below.

proof of correctness

There are three cases to be examined. We will refer to them as the case of weak jars, the case of sturdy jars, and the case of a sufficiently large ladder

(a) weak jars
Suppose that our jars are weak and break after being dropped on from the lowest rung.
The proposed algorithm will terminate after correctly returning 0 at line 7, since none of the rungs are safe for the jars to be dropped from.

(b) sturdy jars
Suppose that our jars are strong and we reach the top of the ladder without breaking a single one.
The control statement on line 3 will evaluate to false and the program will return the height of the ladder n on line 21 before terminating.

(c) sufficiently large ladder
Suppose our jars are neither too strong nor too weak such that our first jar breaks somewhere in the middle of the ladder, $j\lfloor\sqrt{n}\rfloor$.
The control statement on line 5 will evaluate to true and we will enter the second chunk of logic to find the break point between $[(j-1)\lfloor\sqrt{n}\rfloor, j\lfloor\sqrt{n}\rfloor]$.
We iterate over each rung on the interval, increasing our rung position i by 1, rather than multiples of $\lfloor\sqrt{n}\rfloor$.
In this way, we are guaranteed to find and return the highest safe rung which will be the rung immediately prior to the next break point $i - 1$.

■

proof of sublinear runtime complexity

Since we iterate over the available ladder rungs in multiples of the square root of the input, we are able to eliminate larger intervals of safe rungs with each step than we otherwise would be able to using a linear approach.

In the case where the highest safe rung is not in the first interval of safe rungs that demand further investigation and with a minimum input size of $n_0 > 3$, the presented algorithm outperforms the intuitive linear approach.

Once the first jar is broken, and potentially all the rungs between between $[(j-1)\lfloor\sqrt{n}\rfloor, j\lfloor\sqrt{n}\rfloor-1]$ must be linearly scanned, our performance is still sublinear as the time saved by skipping swaths of $\sqrt{n}$ intervals compensates for this necessary sacrifice.

In the worst case (in terms of runtime, not jar strength) where the highest safe rung is $\lfloor\sqrt{n}\rfloor - 1$, the runtime complexity is $O(\sqrt{n} + \sqrt{n} - 1)$ which is $O(\sqrt{n})$.

Suppose $n = 25$, and our target rung x is 24.

The proposed algorithm would check the following rungs: $[0, 5, 10, 15, 20, 25, 12, 22, 23, 24]$ for a total of 10 rungs being checked which beats the linear approaches list of checked rungs: $[0, 1, \ldots, 23, 24]$ of length 25. The larger the input n, the greater amount of time is saved.

■