CSC165H1: Problem Set 4 Sample Solutions

Due April 4, 2018 before 10pm

Note: solutions may be incomplete, and meant to be used as guidelines only. We encourage you to ask follow-up questions on the course forum or during office hours.

1. [**4 marks**] **Binary representation and algorithm analysis**. Consider the following algorithm, which manually counts up to a given number n, using an array of 0's and 1's to mimic binary notation.[1]

```
1  from math import floor, log2
2
3
4  def count(n: int) -> None:
5      # Precondition: n > 0.
6      p = floor(log2(n)) + 1     # The number of bits required to represent n.
7      bits = [0] * p             # Initialize an array of length p with all 0's.
8
9      for i in range(n):  # i = 0, 1, ..., n-1
10         # Increment the current count in the bits array. This adds 1 to
11         # the current number, basically using the loop to act as a "carry" operation.
12         j = p - 1
13         while bits[j] == 1:
14             bits[j] = 0
15             j -= 1
16         bits[j] = 1
```

For this question, assume each individual line of code in the above algorithm takes constant time, i.e., counts as a single step. (This includes the `[0] * p` line.)

(a) [**3 marks**] Prove that the running time of this algorithm is $\mathcal{O}(n \log n)$.

> **Solution**
>
> First, the two lines before the `for` loop take constant time, so we count this as 1 step.
>
> For a fixed iteration of the outer loop, the inner `while` loop (on line 13) runs *at most* `p` times, since `j` starts at `p-1` and goes down by 1 at each iteration. Note that `p` has value $\lfloor \log n \rfloor + 1$. Each iteration of the while loop takes 1 step, for a total cost of *at most* $\lfloor \log n \rfloor + 1$ steps for the `while` loop per iteration of the outer loop.
>
> Then for a fixed iteration of the outer loop, we add 1 step for the other constant-time operations in the loop body (lines 12 and 16), for a total cost of *at most* $\lfloor \log n \rfloor + 2$ steps per iteration. The outer loop also takes (exactly) n iterations (for i going from 0 to $n-1$), leading to a total runtime of *at most* $n(\lfloor \log n \rfloor + 2) \in \mathcal{O}(n \log n)$ steps.

(b) [**1 mark**] Prove that the running time of this algorithm is $\Omega(n)$.

> **Solution**
>
> As we observed in the previous part, the outer `for` loop of this algorithm takes n iterations. Each iteration takes *at least* 1 step,[*] and so the cost of the `for` loop is *at least* n steps, which

[1] This is an extremely *inefficient* way of storing binary, and is certainly not how modern hardware does it. But it's useful as an interesting algorithm on which to perform runtime analysis.

is $\Omega(n)$.

*Note: *any* non-empty block of code takes at least 1 step!

2. **[10 marks] Worst-case and best-case algorithm analysis.** Consider the following function, which takes in a list of integers.

```
def myprogram(L: List[int]) -> None:
    n = len(L)
    i = n - 1
    x = 1
    while i > 0:
        if L[i] % 2 == 0:
            i = i // 2  # integer division, rounds down
            x += 1
        else:
            i -= x
```

Let $WC(n)$ and $BC(n)$ be the worst-case and best-case runtime functions of `myprogram`, respectively, where n represents the length of the input list `L`. You may take the runtime of `myprogram` on a given list `L` to be equal to the number of executions of the `while` loop.

(a) **[3 marks]** Prove that $WC(n) \in \mathcal{O}(n)$.

Solution

Let $n \in \mathbb{N}$, and consider running `myprogram` on an (arbitrary) input L of length n.

We claim that at each loop iteration i goes down by at least 1. This is because whenever $L[i]$ is even, then i decreases to $\lfloor i/2 \rfloor$ and thus decreases by at least one. When $L[i]$ is odd, then i goes from i to $i - x$. Since x starts at 1, and only increases, this means that in this case as well i decreases by at least 1.

Then since i starts equal to n and the loop stops when $i \leq 0$, the number of iterations of the while loop is at most n. We count the cost of a single loop iteration as 1 step (constant-time operation), and so the loop takes *at most* n steps in total, which is $\mathcal{O}(n)$.

(b) **[2 marks]** Prove that $WC(n) \in \Omega(n)$.

Solution

To prove that $WC(n) \in \Omega(n)$, we need to find an input family whose running time is $\Omega(n)$. Let $n \in \mathbb{N}$. The input family we'll choose is where L is a list of length n that contains all 1's.

In this case, since $L[i]$ is never even, the `else` branch will always execute, causing i to decrease by 1 at each iteration. (This is exact since x starts off at 1 and never changes value when L contains all odd numbers.) So then in this case, it will take exactly n iterations until the loop terminates (when $i = 0$). So for this input, there are n steps, which is $\Theta(n)$.

[Note: in fact, we only needed to prove that the runtime for this input family is $\Omega(n)$, but of course if the runtime is $\Theta(n)$, then it is also $\Omega(n)$.]

(c) **[2 marks]** Prove that $BC(n) \in \mathcal{O}(\log n)$.

Solution

To prove that $BC(n) \in \mathcal{O}(\log n)$, we need to find an input family whose running time is $\mathcal{O}(\log n)$. Let $n \in \mathbb{N}$. The input family we'll choose is where L is a list of length n that contains all 2's.

Then $L[i]$ is even for every i, and thus, at every iteration i will go down by half. Since i starts equal to n, after k iterations the value of i is at most $n/2^i$.* So after $k = \lceil \log n \rceil$ iterations, $i < n/2^{\lceil \log n \rceil} \leq 1$, and the loop stops.

This means that for this input, there are at most $\lceil \log n \rceil$ loop iterations; since each iteration takes one step, there are at most $\lceil \log n \rceil$ steps total, which is $\mathcal{O}(\log n)$.

*We need the "at most" here because the integer division rounds down.

(d) **[3 marks]** Prove that $BC(n) \in \Omega(\log n)$.

Solution

To prove that $BC(n) \in \Omega(\log n)$, we need to prove that for *every* input of length n, this program takes *at least* $c \log n$ steps (for some constant c that doesn't depend on n), for "large enough" n. Formally, we'll do this by showing that for all $n \geq 16$, the loop takes *at least* $\frac{\log n}{2}$ iterations. Before our main analysis, we have the following claims:

Claim 1. Let $j \in \mathbb{N}$. Then after j iterations of the `while` loop, $x \leq j + 1$.
To justify this, we note that x starts off as 1, and it can only increase by at most 1 during each iteration of the loop.

Claim 2. Let $n \in \mathbb{N}$. If $n \geq 16$ then $\log n \leq \sqrt{n}$.
Here is the key idea of how to prove this is to first prove that $m^2 \leq 2^m$ for all $m \geq 4$, which implies (through a change of variables) that $n \leq 2^{\sqrt{n}}$ for $n \geq 16$. Finally, taking the log of both sides results in $\log n \leq \sqrt{n}$.

We will now analyze the loop. Let $n \in \mathbb{N}$ and assume $n \geq 16$, and let `L` be an arbitrary list of length n. We'll prove the following claim by induction. We use the notation i_j to represent the value of the variable i just before the j-th iteration of the loop (where j counts starting from 1).

Claim: For all $j \in \mathbb{N}$, $1 \leq j \leq \frac{\log n}{2} \Rightarrow i_j \geq \frac{n-1}{2^{j-1}}$.

Proof. **Base case**: let $j = 1$. Just before the first iteration of the while loop, $i_1 = n - 1$. The right side of the inequality is $\frac{n-1}{2^{1-1}} = n - 1$, and so the inequality holds. □

Induction step: let $j \in \mathbb{N}$, and assume $1 \leq j < \frac{\log n}{2}$ and that $i_j \geq \frac{n-1}{2^{n-1}}$. We want to show that $i_{j+1} \geq \frac{n-1}{2^j}$.

By our induction hypothesis, the value of `i` immediately before the j-th iteration is $\geq \frac{n-1}{2^{j-1}}$. We want to prove that the value of `i` immediately after the j-th iteration (which is the same as its value immediately before the $(j+1)$-th iteration) is $\geq \frac{n-1}{2^j}$. To do this, we need to look at what happens during the j-th iteration.

There are two cases to consider. If $L[i_j]$ is even, then $i_{j+1} = i_j/2$. Thus $i_{j+1} \geq \frac{(n-1)}{2^{j-1}} \cdot \frac{1}{2} = \frac{(n-1)}{2^j}$.

The other case is when $L[i_j]$ is odd, and in that case $i_{j+1} = i_j - x$. Here we use our first claim to note that $x \leq j$, and since $j < \frac{\log n}{2}$,

$$\begin{aligned} i_{j+1} &= i_j - x \\ &\geq i_j - \frac{\log n}{2} \\ &\geq \frac{n-1}{2^{j-1}} - \frac{\log n}{2} && \text{(by the induction hypothesis)} \end{aligned}$$

Now we want to show that $\frac{n-1}{2^j} \geq \frac{\log n}{2}$. We start from $j < \frac{\log n}{2}$:

$$\begin{aligned} j &< \frac{\log n}{2} \\ 2^j &< 2^{\frac{\log n}{2}} \\ 2^j &< \sqrt{n} \\ \frac{n-1}{2^j} &> \frac{n-1}{\sqrt{n}} \\ \frac{n-1}{2^j} &> \sqrt{n} - \frac{1}{\sqrt{n}} \\ \frac{n-1}{2^j} &> \log n - \frac{1}{\sqrt{n}} && \text{(by Claim 2)} \\ \frac{n-1}{2^j} &> \frac{\log n}{2} \end{aligned}$$

Now using this inequality, we get:

$$\begin{aligned} i_{j+1} &\geq \frac{n-1}{2^{j-1}} - \frac{\log n}{2} \\ &\geq \frac{n-1}{2^{j-1}} - \frac{n-1}{2^j} \\ &= \frac{n-1}{2^j} \end{aligned}$$

This completes the inductive proof.

So then right before the $\frac{\log n}{2}$ iteration of the while loop, the value of i is at least $\frac{n-1}{2^{\frac{\log n}{2}}} > 1$, and therefore the while loop executes at least $\frac{\log n}{2} - 1$ times, and thus at least this many steps. This leads to a lower bound on the best-case running time of $\Omega(\log n)$.

Note: this is actually the hardest question of this problem set. A correct proof here needs to argue that the variable `x` cannot be too big, so that the line `i -= x` doesn't cause `i` to decrease too quickly!

3. **[14 marks] Graph algorithm**. Let $G = (V, E)$ be a graph, and let $V = \{0, 1, \ldots, n-1\}$ be the vertices of the graph. One common way to represent graphs in a computer program is with an **adjacency matrix**, a two-dimensional n-by-n array[2] M containing 0's and 1's. The entry $M[i][j]$ equals 1 if $\{i, j\} \in E$, and 0 otherwise; that is, the entries of the adjacency matrix represent the edges of the graph.

Keep in mind that graphs in our course are *symmetric* (an edge $\{i, j\}$ is equivalent to an edge $\{j, i\}$), and that no vertex can ever be adjacent to itself. This means that for all $i, j \in \{0, 1, \ldots, n-1\}$, $M[i][j] == M[j][i]$, and that $M[i][i] = 0$.

The following algorithm takes as input an adjacency matrix M and determines whether the graph contains at least one **isolated vertex**, which is a vertex that has no neighbours. If such a vertex is found, it then does a very large amount of printing!

```
1  def has_isolated(M):
2      n = len(M)    # n is the number of vertices of the graph
3      found_isolated = False
4
5      for i in range(n):   # i = 0, 1, ..., n-1
6          count = 0
7          for j in range(n):    # j = 0, 1, ..., n-1
8              count = count + M[i][j]
9          if count == 0:
10           found_isolated = True
11           break
12
13     if found_isolated:
14         for k in range(2 ** n):
15             print('Degree too small')
```

(a) **[3 marks]** Prove that the worst-case running time of this algorithm is $\Theta(2^n)$.

> **<u>Solution</u>**
>
> The outer loop executes n times, and the inner loop executes n times, for a total of n^2. Then if `found_isolated` is true, the for loop takes time 2^n. Thus the total runtime is $n^2 + 2^n \in O(2^n)$. To see that the worst-case runtime is $\Omega(2^n)$, consider a graph with no edges. Then `found_isolated` will get set to true in the outer for loop, and thus the runtime will be $\Omega(2^n)$.

(b) **[3 marks]** Prove that the best-case running time of this algorithm is $\Theta(n^2)$.

> **<u>Solution</u>**
>
> To determine a lower bound on the best-case runtime, we can ignore the constant cost from lines 2-3, and need only think about the runtime of lines 5-15.
>
> Every adjacency matrix either contains an isolated vertex or it doesn't.
>
> In the case that the given adjacency matrix does contain an isolated vertex, the loop in lines 13-15 executes, with a runtime of 2^n steps. So in this case there are at least n^2 steps when $n \geq 2$.
>
> In the case that the given adjacency matrix does not contain an isolated vertex, the loop starting in line 5 iterates exactly n times and it contains a loop starting in line 7 that also iterates exactly n times. Since these loops contain at least one step, the total runtime is at least n^2.

[2] In Python, this would be a list of length n, each of whose elements is itself a list of length n.

So, in either case, the total runtime is at least n^2 steps, and so the best-case runtime is in $\Omega(n^2)$.

To see that the best-case runtime is $\mathcal{O}(n^2)$, consider a graph with all possible edges. Then `found_isolated` will not get set to true in the outer for loop. The outer loop will iterate exactly n times, and since the inner loop will iterate exactly n times, the runtime for lines 5-11 will be at most $n(n+1)$ steps. (The $+1$ comes from the constant cost of running lines 6 and 9-11.)

Outside of the nested loop, there is 1 more step, from considering the cost of lines 2-3 and 13-15 (and remembering that `found_isolated` will not be set to true).

The total runtime will for a graph will all possible edges will be at most $n(n+1)+1$ steps, and so the best-case runtime is in $\mathcal{O}(n^2)$.

(c) **[1 mark]** Let $n \in \mathbb{N}$. Find a formula for the number of adjacency matrices of size n-by-n that represent valid graphs. For example, a graph $G = (V, E)$ with $|V| = 4$ has 64 possible adjacency matrices.

Note: a graph with the single edge $(1, 2)$ is considered different from a graph with the single edge $(2, 3)$, and should be counted separately. (Even though these graphs have the same "shape", the vertices that are adjacent to each other are different for the two graphs.)

Solution

The number of adjacency matrices of size n-by-n that represent valid graphs is $2^{n(n-1)/2}$.

(d) **[2 marks]** Prove the formula that you derived in Part (c).

Solution

We can prove this by induction on n.

Base case: let $n = 0$. In this case, there is only one graph with 0 vertices ($V = \emptyset$ and $E = \emptyset$). At the same time, $2^{0(-1)/2} = 2^0 = 1$, and so this formula is true.

Induction step: let $k \in \mathbb{N}$ and assume that the formula holds for k-by-k matrices, i.e., there are $2^{k(k-1)/2}$ distinct matrices that represent valid graphs on k vertices. We want to prove that the number of $(k+1)$-by-$(k+1)$ adjacency matrices representing valid graphs on $k+1$ vertices is $2^{(k+1)k/2}$.

To see this, we note that any $(k+1)$-by-$(k+1)$ adjacency matrix is formed by taking a k-by-k matrix and adding a new row and column. We claim that there are 2^k choices for the new row/column. To see this, note that the new row and column have length $k+1$, but the last entry (corresponding to very bottom-right corner of the matrix) must be a 0, while the other k entries could be 0 or 1, leading to 2^k possibilities.. Another way to see this is that we take a graph on k vertices and add a $(k+1)$-th vertex to it; there are 2^k ways to connect the new vertex to the existing k vertices.

By the induction hypothesis, there are $2^{k(k-1)/2}$ k-by-k adjacency matrices to start from. So then the total number of $(k+1)$-by-$(k+1)$ adjacency matrices is

$$2^{k(k-1)/2} \cdot 2^k = 2^{k(k-1)/2+k} = 2^{(k+1)k/2}$$

(e) **[2 marks]** Let $n \in \mathbb{N}$. Prove that the number of n-by-n adjacency matrices that represent a graph with at least one isolated vertex is at most $n \cdot 2^{(n-1)(n-2)/2}$.

Solution

Let $n \in \mathbb{N}$.

If $n = 0$, there are *no* graphs on 0 vertices that have at least one isolated vertex, and $0 \leq 0 \cdot 2^{(0-1)(0-2)/2}$.

If $n > 0$, then for each natural number i between 1 and n (inclusive), define the set $Isolated_i$ to be the set of n-by-n adjacency matrices where vertex i is isolated.

Let $i \in \{1, \dots, n\}$. We claim that $|Isolated_i| = 2^{(n-1)(n-2)/2}$. This is because all of the matrices in this set represent a different graph on $n-1$ vertices by removing vertex i, and from part (c) we know there are $2^{(n-1)(n-2)/2}$ different graphs on $n-1$ vertices.

So then the total number of adjacency matrices with at least one isolated vertex is at most the sum of the sizes of the $Isolated_i$ sets:*

$$\sum_{i=1}^{n} |Isolated_i| = \sum_{i=1}^{n} 2^{(n-1)(n-2)/2} = n \cdot 2^{(n-1)(n-2)/2}$$

*The reason this is an "at most" is because there might be duplicates among the sets.

(f) **[3 marks]** Finally, let $AC(n)$ be the average-case runtime of the above algorithm, where the set of inputs is simply all valid adjacency matrices (same as what you counted in part (c)).
Prove that $AC(n) \in \Theta(n^2)$.

<u>Solution</u>

Part 1: proving that $AC(n) \in \Omega(n^2)$.

To prove this, we use the fact that for any set of numbers S, $avg(S) \geq min(S)$, i.e., the set's average is greater than or equal to its minimum element.

So then by the definition of average-case and best-case running time, we know that for all $n \in \mathbb{N}$, $AC(n) \geq BC(n)$ (which implies that $AC(n) \in \Omega(BC(n))$). By part (b), we know that $BC(n) \in \Omega(n^2)$. So then we can conclude that $AC(n) \in \Omega(n^2)$.

Part 2: Proving that $AC(n) \in \mathcal{O}(n^2)$.

To prove this, we will prove an upper bound on the average runtime of the given algorithm.

Let $n \in \mathbb{N}$. From part (c), we know that there are $2^{n(n-1)/2}$ inputs of size n (i.e., the number of valid adjacency matrices of size n-by-n). Let S be the set of adjacency matrices that have an isolated vertex, and T be the set of adjacency matrices that don't have one. We know from part (e) that $|S| \leq n \cdot 2^{(n-1)(n-2)/2}$, while $|T| \leq 2^{n(n-1)/2}$.*

Finally, every input in S takes at most $n^2 + 2^n$ steps, using the same analysis in part (a). Every input in T takes just at most n^2 steps, because the `if found_isolated` branch at line 13 will never execute for these inputs.

So the average runtime, $AVG(n)$, is *at most*:

$$\begin{aligned} AVG(n) &\leq \frac{|S| \cdot (n^2 + 2^n) + |T| \cdot n^2}{2^{n(n-1)/2}} \\ &\leq \frac{(n \cdot 2^{(n-1)(n-2)/2}) \cdot (n^2 + 2^n) + 2^{n(n-1)/2} \cdot n^2}{2^{n(n-1)/2}} \\ &= (n^3 + n \cdot 2^n) \cdot \frac{2^{(n-1)(n-2)/2}}{2^{n(n-1)/2}} + n^2 \\ &= (n^3 + n \cdot 2^n) \cdot \frac{1}{2^{n-1}} + n^2 \\ &= \frac{n^3}{2^{n-1}} + 2n + n^2 \\ &\in \mathcal{O}(n^2) \end{aligned}$$

*This is, of course, an overestimate of $|T|$, but it'll do for our purposes.