Homework #1 Solutions

CS 344: Design and Analysis of Computer Algorithms (Fall 2022)

Problem 1

For each of the following functions, state whether $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$, or if both are true, then write $f(n) = \Theta(g(n))$. No proofs required for this problem.

(1) $\boxed{f(n) = \Omega(g(n))}$ when $f(n) = n^4 - 7n$ and $g(n) = n^3 + 10n^2$.

(2) $\boxed{f(n) = O(g(n))}$ when $f(n) = (\sqrt{n})^3$ and $g(n) = n^2 - (\sqrt{n})^3$.

(3) $\boxed{f(n) = O(g(n))}$ when $f(n) = n\log^3(n)$ and $g(n) = n^{log_2(5)}$.

(4) $\boxed{f(n) = \Theta(g(n))}$ when $f(n) = 2^{\log_2(n)}$ and $g(n) = n$.

(5) $\boxed{f(n) = \Omega(g(n))}$ when $f(n) = n/\log^2(n)$ and $g(n) = log^6(n)$.

(6) $\boxed{f(n) = O(g(n))}$ when $f(n) = 4^n$ and $g(n) = 5^n$.

(7) $\boxed{f(n) = \Theta(g(n))}$ when $f(n) = \log_3(n)$ and $g(n) = \log_5(n)$.

(8) $\boxed{f(n) = O(g(n))}$ when $f(n) = n^3$ and $g(n) = 2^n$.

(9) $\boxed{f(n) = O(g(n))}$ when $f(n) = \log^2(n)$ and $g(n) = \sqrt{n}$.

(10) $\boxed{f(n) = O(g(n))}$ when $f(n) = n\log(n)$ and $g(n) = n^2$.

Problem 2

(1) Prove by induction that $\sum_{i=0}^{k} i2^i = (k-1)2^{k+1} + 2$.

Base case: When k is 0, the following is true: $0 \cdot 2^0 = (0-1)2^{0+1} + 2$.

Inductive hypothesis: $\sum_{i=0}^{k'} i2^i = (k'-1)2^{k'+1} + 2$ for all $0 \le k' < k$.

We want to prove: $\sum_{i=0}^{k} i2^i = (k-1)2^{k+1} + 2$.

Proof / Inductive Step:

$$\begin{aligned}
\sum_{i=0}^{k} i2^i &= \left(\sum_{i=0}^{k-1} i2^i\right) + k2^k \\
&= (k-2)2^k + 2 + k2^k && \text{(Inductive hypothesis on } \left(\textstyle\sum_{i=0}^{k-1} i2^i\right)\text{)} \\
&= (k-2+k)2^k + 2 \\
&= (2k-2)2^k + 2 \\
&= 2(k-1)2^k + 2 \\
&= (k-1)2^{k+1} + 2.
\end{aligned}$$

(2) Prove that $\sum_{i=1}^{n} \log(i) = \Theta(n \log(n))$.

First, we show an upper bound for $\sum_{i=1}^{n} \log(i)$.

$$\begin{aligned}
\sum_{i=1}^{n} \log(i) &\le \sum_{i=1}^{n} \log(n) && (i \le n) \\
&= n \log(n) \\
&= O(n \log(n)). && \text{(If } x \le y \text{ and } y = O(f)\text{, then } x = O(f)\text{)}
\end{aligned}$$

Second, we show a lower bound for $\sum_{i=1}^{n} \log(i)$.

$$\begin{aligned}
\sum_{i=1}^{n} \log(i) &\ge \sum_{i=\lceil n/2 \rceil}^{n} \log(i) \\
&\ge \sum_{i=\lceil n/2 \rceil}^{n} \log(n/2) && (i \ge n/2) \\
&\ge (n/2 - 1) \log(n/2) \\
&= \Omega(n \log(n)). && \text{(If } x \ge y \text{ and } y = \Omega(f)\text{, then } x = \Omega(f)\text{)}
\end{aligned}$$

It thus follows that $\sum_{i=1}^{n} \log(i) = \Theta(n \log(n))$.

(3) What is $\sum_{i=0}^{\log_2(n)} 8^i$ equal to in Θ-notation? (No formal proof necessary, just a brief explanation.)

$$\begin{aligned}
\sum_{i=0}^{\log_2(n)} 8^i &= \frac{8^{\log_2(n)+1}-1}{7} && \text{(Geometric sum formula for radius larger than 1)} \\
&= \frac{8^{\log_2(n)}\cdot 8-1}{7} \\
&= \frac{n^{\log_2(8)}\cdot 8-1}{7} && (x^{\log_b(y)} = y^{\log_b(x)}) \\
&= \frac{8n^3-1}{7} \\
&= \Theta(n^3).
\end{aligned}$$

Problem 3

(1) Simplify $64^{\log_{16}(n)}$; that is, write it as n to the power of some number.

$$\begin{aligned}
64^{\log_{16}(n)} &= n^{\log_{16}(64)} && (x^{\log_b(y)} = y^{\log_b(x)}) \\
&= n^{6\log_{16}(2)} \\
&= n^{3/2}.
\end{aligned}$$

(2) Simplify $5^{\log_7(n)}$ – in particular write it as n to the power of some number.

$$\begin{aligned}
5^{\log_7(n)} &= n^{\log_7(5)} && (x^{\log_b(y)} = y^{\log_b(x)}) \\
&= n^{0.82708\ldots}.
\end{aligned}$$

(3) Prove that for any constants c, c', $\log_c(n) = \theta(\log_{c'}(n))$.

By the change-of-base formula,

$$\log_c(n) = \frac{\log_{c'}(n)}{\log_{c'}(c)},$$

and after rearranging, we see that

$$\frac{\log_c(n)}{\log_{c'}(n)} = \frac{1}{\log_{c'}(c)}.$$

Thus,

$$\lim_{n\to\infty} \frac{\log_c(n)}{\log_{c'}(n)} = \frac{1}{\log_{c'}(c)}$$

and hence, since the right hand side is a constant, $\log_c(n) = \Theta(\log_{c'}(n))$ using the definition for Θ-notation.

Problem 4

Consider the following problem:

- Input: An array A with n distinct (non-equal) elements
- Output: numbers x and y in A that minimize $|x - y|$, where $|x - y|$ denotes absolute-value(x-y). (If there are multiple closest pairs, you only have to return one of them.)

Write pseudocode for an algorithm for the above problem whose running time is $o(n^2)$. Note that this is little-o; in words, your running time must be *better* than $O(n^2)$. So an algorithm with running time $O(n^2)$ will receive very few points.

At a high level, the algorithm first sorts the input array and then finds the two adjacent elements with the smallest difference. The following pseudocode provides a more detailed description.

```
ClosestPair(A[0..n − 1]):
  Sort A in increasing order
  x ← A[0]
  y ← A[1]
  for i = 2 to n − 1:
      if A[i] − A[i − 1] < y − x
          x ← A[i − 1]
          y ← A[i]
  return (x, y)
```

Using an $O(n \log(n))$ time sorting algorithm such as MergeSort, the ClosestPair problem takes $O(n \log(n) + n)$ time, which is $o(n^2)$.

Problem 5

Consider the following problem:

- INPUT: An array $A[0...n - 1]$, where each $A[i]$ is either a 0 or a 1. Also, you are guaranteed that the length n of the array A is a multiple of 5.
- OUTPUT: Index $k \leq 4n/5$ such the subarray $A[k], A[k+1], ..., A[k+n/5-1]$ contains as many 1s as possible. If there exist multiple indices k that achieve this maximum, you only have to return one of them.

For example, say that $A = 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0$. Note that $n = 15$ and $n/5 = 3$, so you are looking for the subarray of length 3 with the most number of 1s. The correct output is $k = 6$ because the subarray $A[6], A[7], A[8] = 1, 0, 1$ contains the maximum possible number of 1s among all subarrays of length 3.

The Problem: Write pseudocode for an algorithm that solves the above problem in $O(n)$ time.

HINT: Say that you already figured out the number of 1s in subarray $A[i]...A[i+n/5-1]$ for some i. How can you use this information to very quickly figure out the number of 1s in the next subarray $A[i+1]...A[i+n/5]$?

Naively, one may compute $A[i] + A[i+1] + \cdots + A[i+n/5-1]$ for all $i \leq 4n/5$. This, however, will take $\Omega(n^2)$ time in total.

Let $c = A[i] + A[i+1] + \cdots + A[i+n/5-1]$. Notice that if we know c, we can find $A[i+1] + A[i+2] + \cdots + A[i+n/5]$ without recomputing $A[i+1] + A[i+2] + \cdots + A[i+n/5-1]$. Just subtract $A[i]$ from c and add $A[i+n/5]$, which is done with only a constant amount of computation.

```
MostOnes(A[0..n − 1]):
  c ← A[0] + A[1] + · · · + A[n/5 − 1]
  m ← c // Max count seen so far.
  k ← 0
  for i = 0 to 4n/5 − 1:
      c ← c − A[i] + A[i + n/5]
      if c > m
          m ← c
          k ← i + 1
  return k
```

The initialization of c takes $O(n)$ time. Each computation in the loop takes $O(1)$ time, and the loop runs $O(n)$ times. The total running time is thus $O(n)$.

Problem 6

Consider the following input problem

- INPUT: a 2-dimensional array $A[0...n-1][0...n-1]$ with n rows and n columns. Note that you can use $A[i][j]$ to refer to the element in row i and column j, and that you can access any particular $A[i][j]$ in constant time. Each entry $A[i][j]$ is either 0 or 1.
- OUTPUT: find an index i such that for *all* $j \neq i$ it is the case that $A[i,j] = 1$ and $A[j,i] = 0$. If no such index exists, return "no solution".

Interpretation in words: The problem might seem more intuitive if you think of at as follows. Say that you have n people $p_0, ..., p_{n-1}$ and think of $A[i][j]$ as representing who follows whom on twitter: $A[i][j] = 1$ means that p_i follows p_j and $A[i][j] = 0$ means that p_i does not follow p_j. Note that it is possible that $A[i][j] = 0$ but that $A[j][i] = 1$. Your goal is to find the person p_i such that they follow everyone but no one follows them. If no such person exists you return "no solution".

(1) Say that $A[i][j] = 1$. From this piece of information alone, which index do you know is definitely NOT the final answer.

> j is definitely not the final answer. Using the analogy of people following people, we are looking for someone who is not followed, but here p_j is followed by p_i.

(2) Say that $A[i][j] = 0$. From this piece of information alone, which index do you know is definitely NOT the final answer.

i is definitely not the final answer. Using the analogy of people following people, we are looking for someone who follows everyone, but here p_i does not follow p_j.

(3) Write pseudocode that solves the above problem in $O(n)$ time. Note that the runtime should be $O(n)$, not $O(n^2)$.

```
FindLurker(A[0..n − 1][0..n − 1]):
  c ← 0 // Candidate or, equivalently, Column.
  r ← 1 // Row.
  while r ≤ n − 1
      if A[r][c] = 1
          c ← r
      r ← r + 1
  if any A[c][j] = 0 for j ≠ c
      return NoSolution
  if any A[i][c] = 1 for i ≠ c
      return NoSolution
  return c
```

Running Time. The while loop iterates $O(n)$ times since each iteration increments r, which can happen only $O(n)$ many times before the loop terminates. The final checks for NoSolution takes $O(n)$ time each. The total running time is thus $O(n)$.

Correctness. Observe first that there can be at most one lurker; if p_i is a lurker, then all p_j for $j \neq i$ are followed by p_i and, by virtue of being followed, p_j is not a lurker.

FindLurker stores a candidate lurker c. The loop maintains the following two invariants:

(i) $c < r$.

(ii) Every p_j where $j < r$ and $j \neq c$ is not a lurker.

Invariant (i) can immediately be seen to hold. How is Invariant (ii) maintained? At each iteration of the loop, if $A[r][c] = 0$, we know by Part 2 that r is not a lurker and thus setting $r \leftarrow r+1$ maintains Invariant (ii). If, on the other hand, $A[r][c] = 1$, we know by Part 1 that c is not a lurker. Since also $c < r$, setting $c \leftarrow r$ and $r \leftarrow r + 1$ maintains Invariant (ii).

Once the loop terminates, $r = n$ and so by both the uniqueness of the lurker and Invariants (i,ii), c is the only possible lurker. FindLurker then checks if c is indeed the lurker, and returns a result accordingly.

Problem 7 (Extra Credit)

Consider the algorithm Foo(n). What is the running time in Θ notation? *For this problem, you must briefly justify your answer.* By briefly justify, I mean that you need to write enough that a knowledgeable reader would understand why your answer is correct.

Foo(n)

- For $i = 1$ to n
  - $x = n$.
  - While $(x \geq 2)$
    - $x \leftarrow \sqrt{x}$
    - Do placeholder stuff that takes $O(1)$ time.

NOTE: for this problem, you can assume that $\sqrt{x}$ can always be computed in $O(1)$ time.

The running time of Foo is $\Theta(n \log(\log(n)))$.

To see this "intuitively", observe that the binary representation of n is $\Theta(\log(n))$ bits long. Each square-root operation halves the number of bits. After logarithmically many steps (in the length of the thing initially being halved), the number of bits becomes $\Theta(1)$. Each iteration thus runs for $\Theta(\log(\log(n)))$ steps. The running time follows from there being n iterations.