CS 270 Homework 2

Neel Gupta

January 25, 2023

Problem 1. Arrange the following functions by asymptotic time complexity using big-O notation.

a) $2^{\log n}$

b) 2^{3n}

c) $n^{n \log n}$

d) $\log n$

e) $n \log(n^2)$

f) n^{n^2}

g) $\log(\log(n^n))$

Answer: $\log n \subset \log(\log(n^n)) \subset 2^{\log n} \subset n \log n^2 \subset 2^{3n} \subset n^{n \log n} \subset n^{n^2}$

$$\begin{aligned}
2^{\log n} &= n \\
2^{3n} &= (2^3)^n = 8^n \\
n^{n \log n} &= 2^{\log(n^{n \log n})} = 2^{n \log n * \log n} = 2^{n(\log n)^2} \\
n \log(n^2) &= 2n \log n \\
n^{n^2} &= 2^{\log n^{n^2}} = 2^{n^2 \log n} \\
\log(\log(n^n)) &= \log(n \log n)
\end{aligned}$$

Problem 2. Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, decide whether each statement is true or false and briefly explain why.

a) $f_1(n)/f_2(n) = O(g_1(n)/g_2(n))$

b) $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

c) $f_1^2(n) = O(g_1^2(n))$

d) $\log f_1(n) = O(\log g_1(n))$

a) *Answer:* True

$\exists N \in \mathbb{N}$ and $c_1, c_2 \in \mathbb{R}$ s.t. $\forall n \in \mathbb{N}$ and $n \geq N$,

$$\begin{aligned} \frac{f_1(n)}{f_2(n)} &\leq \frac{c_1 g_1(n)}{c_2 g_2(n)} \\ &= (c_1/c_2) g_1(n)/g_2(n) && (\because c_1/c_2 \in \mathbb{R}) \\ &\therefore f_1(n)/f_2(n) = O(g_1(n)/g_2(n)) \end{aligned}$$

b) *Answer:* True

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n)) \\ &\leq c_1 * \max(g_1(n), g_2(n)) + c_2 * \min(g_1(n), g_2(n)) \\ &\leq (c_1 + c_2) \max(g_1(n), g_2(n)) \\ &= O(\max(g_1(n), g_2(n)) \\ &\therefore f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))) \end{aligned}$$

c) *Answer:* True

$\exists N \in \mathbb{N}$ and $c \in \mathbb{R}$ s.t. $\forall n \in \mathbb{N}$ and $n \geq N$,

$$\begin{aligned} 0 \leq f_1(n) &\leq c g_1(n) && (\because f_1(n) = O(g_1(n))) \\ 0^2 \leq f_1^2(n) &\leq (c g_1(n))^2 \\ &= c^2 g_1^2(n) && (\because c^2 \in \mathbb{R}) \\ &\therefore f_1^2(n) = O(g_1^2(n)) \end{aligned}$$

d) *Answer:* False
Consider the following counterexample.

$$\begin{aligned} &\text{Let } f_1(n) = 1 \text{ and } g_1(n) = 2 \\ &\log_2 f_1(n) = 0 \neq O(1) = 1 = \log_2 g_1(n) \\ &\therefore \log f_1(n) \neq O(\log g_1(n)) \end{aligned}$$

Problem 3. Given an undirected graph G with n nodes and m edges, design an algorithm in $O(m+n)$ to detect whether G contains a cycle. Your algorithm should output a cycle if there is one.

Answer: The following problem can be solved using a depth-first search (DFS) approach where we go down successive levels and see if a neighbor of a deeper node has already been found as a neighbor of a previous node using a list of size n to keep store of visited nodes.

Algorithm 1 Detecting cycles within an undirected graph

```
Let $\bar{v}$ be a list of size $n$ where $v_i$ is $j$ if $x_j$ visited $x_i$, and 0 otherwise
procedure DETECTCYCLES($G$, $x_i$, $x_j$, $\bar{v}$)
    visit $x_i$, so set $v_i$ to $j$, the index of $i$'s parent
    for each neighbor $x_k$ of $x_i$ do
        if $v_k$ is already not 0 then
            if $x_k$ is not $x_i$'s child then
                return true
            end if
        else if detectCycles($G$,$x_i$,$x_k$, $\bar{v}$) then
            return true
        end if
    end for
    return false
end procedure
while $G$ has unvisited nodes do
    Let $x_1$ be the root or first unvisited node
    detectCycles($G$, 0, $x_1$, $\bar{v}$)
end while
```

The time complexity of this algorithm is $O(n+m)$, and the space complexity of the algorithm is $O(n)$. The algorithm terminates in a finite amount of time since for each connected undirected graph, the algorithm will return false after traversing all neighbors then all neighbors of those those neighbors until every piece of the connected graph has been visited. This DFS traversal takes $O(n+m)$ since only after visiting every node in G through potentially every edge can we deduce whether our graph has a cycle or not.

Problem 4. Given an array A of n integers, return an n-by-n matrix B in which $B[i,j]$ (for $i < j$) contains the sum of array entries $A[i]$ to $A[j]$.

Algorithm 2 Simple algorithm to achieve this

```
for $i = 1, 2, ..., n$ do
    for $j = i+1, i+2, ..., n$ do
        Add up array entires $A[i]$ through $A[j]$
        Store result in $B[i,j]$
    end for
end for
```

a) For some function f, give an upper bound in the form $O(f(n))$ on the runtime of this algorithm on an input of size n.

b) For this same function f, show that the runtime of this algorithm on an input of size n is also $\Omega(f(n))$.

c) Although this simple algorithm is the most natural way to solve this problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem with an asymptotically better running time.

Answer:

a) The first loop will make n iterations, and the second will loop will make n more iterations at each of those iterations. At each of these iterations, we want to sum up the ith to jth element which takes $O(j - i + 1)$. In the worst case, this will be take $O(n)$ time per each n^2 iterations, so an upper bound on the runtime of this algorithm could be $O(n^3)$.

b) By showing that this algorithm is $\Theta(n^3)$, we can imply that the runtime of this algorithm is also $\Omega(n^3)$.

Consider the time when $i \le n/4$ and $j \ge 3n/4$,

$$\text{then } j - i + 1 \ge \frac{3n - n}{4} + 1 > \frac{n}{2}$$
$$\therefore A[i] + ... + A[j] \ge n/2$$

There are $(\frac{n}{4})^2$ iterations where $i \le n/4$ and $j \ge 3n/4$, and at each of these iterations, at least $n/2$ work is done. $\Theta(f(n)) = \frac{n}{2}(\frac{n}{4})^2 = \frac{n^3}{32}$ $\therefore$ $f(n)$ is $\Omega(n^3)$.

c) This task can be completed in $O(n^2)$ time by precomputing the sums.

Algorithm 3 More optimized algorithm to achieve the same task

```
for $i = 1, 2, ..., n$ do
    $B[i, i+1] \leftarrow A[i] + A[i+1]$        ▷ computing the running sum
end for
for $j = 2, 3, ..., n$ do
    for $k = 1, 2, ..., n$ do
        $newIdx \leftarrow j + k$
        $B[i, newIdx] \leftarrow B[i, newIdx - 1] + A[j]$
    end for
end for
```

There are $O(n)$ operations to compute running sums for the first column of the matrix. Then for each of the n iterations, there are up to n more iterations, resulting in a runtime of $f(n) = n + (n-2) * n = O(n^2)$.

Problem 5. Consider a researcher's Erdos Number, which for Paul Erdos was 0, for researchers who wrote papers with him was 1, for researchers who wrote papers with researchers who wrote papers with him was 2, and so on, that represents how many degrees of freedom of writing are between writers and Erdos himself. Suppose we have a database of all mathematical papers ever written along with their authors.

a) Explain how to represent this data as a graph.

b) Explain how we would compute the Erdos number for a particular researcher.

c) Explain how we would determine all researchers with Erdos numbers at most two.

Answer:

a) Let G be an undirected graph where authors are nodes and an edge between nodes X and Y represent whether author X cowrote a paper with author Y. This graph could represent mathematicians and their authorship to be able to compute Erdos Numbers

b) To compute the Erdos Number of a particular researcher, find the number of minimum number edges which are taken to connect the particular researcher to Paul Erdos, then that number of edges is the researcher's Erdos number.

c) Beginning at the root who is Paul Erdos, conducting a breadth-first search on his node would mean going through the graph G level by level where Erdos number is given by level of depth. All researchers with Erdos numbers at most 2 would be whoever would be visited after running BFS beginning with Erdos's coauthors then visiting the coauthors of Erdos's coauthors, and upon visiting someone who is 3 levels away from Erdos, the search should terminiate, resulting with the set of authors visited as being those with Erdos number of at most 2.

Problem 6. Given a directed acyclic graph, give a linear time algorithm to determine if there is a simple path that visits all vertices.

Answer:

By conducting a topological sort on a DAG, we can see whether "all prerequisites have been met" or a simple path exists between all vertices.

This algorithm runs in $O(n)$. At each iteration, there can be up to $|E|$ or m iterations to neighboring nodes, so for each n nodes, there can be up to m amounts of work, so the runtime for the algorithm will be less than or equal to mn, but since m is a constant and cannot occur everytime, the runtime is $O(n)$. This algorithm is based on creating a topological ordering of all nodes, and if not, it returns false after doing some constant amount of work on each node, thus taking linear time.

Algorithm 4 Algorithm to see if there exists a path visits all verticies

```
Let $G$ be a DAG
Let $v$ be a list of in-degrees for $n$ nodes
Let $t$ be an empty list for the result ordering
for every node $i$ in $G$ do
    $v_i \leftarrow$ in-degree of node $i$
end for
Let $q$ be a queue initialized to all nodes with in-degree 0
while $q$ is not empty do
    if all nodes have been dequeued then
        return false
    end if
    dequeue from $q$ to get node $i$
    add $i$ to the topological ordering $t$
    for all out going edges $(i, j)$ do
        $v_j \leftarrow v_j - 1$
        if $v_j$ is 0 then
            add $u$ to $q$
        else if $v_j$ is -1 then
            return false
        end if
    end for
end while
if $t$ has $n$ elements then
    return true
else
    return false
end if
```