Anash Atul Mehta
USC ID: 6498-8280-04

CSCI 570 – Analysis of Algorithms
Homework #2

1. What is the worst-case runtime performance of the procedure below?

```
c = 0
i = n
while i > 1 do
    for j = 1 to i do
        c = c + 1
    end for
    i = floor(i/2)
end while
return c
```

   Provide a brief explanation for your answer.

   Answer:

   Here, the while loop will run for

   i, i/2, i/4, i/8, …, 1 times

   Hence, the complexity for while loop will be **log n**.

   Now, for the inner *for* loop, it will be running

   n, n/2, n/4, n/8, …., 1 times

   If we add the instances of running, it runs for exactly **2n-1** times (this is a case of Geometric Projection).

   The operation *i = floor(i/2)* runs at a linear **O (1)** time.

   Thus, the algorithm's worst time complexity would be:

   (2n – 1) + log n i.e., **O (n)**

2. Arrange these functions under the O notation using only = (equivalent) or (strict subset of):

   (a) $2^{\log n}$
   (b) 2^{3n}
   (c) $n^{n \log n}$
   (d) $\log n$
   (e) $n \log\left(n^2\right)$
   (f) n^{n^2}
   (g) $\log(\log(n^n))$

   Answer:
   We can further solve the functions in this way:
   $n^{n \log n} = 2^{n (\log n) \wedge 2}$, $2\log^{n} = n$, $n^{n \wedge 2} = 2^{n \wedge 2 * \log n}$, $n^{n \log n} = 2^{n (\log n) \wedge 2}$

Now, we can divide the functions into categories:

Logarithmic: d, g
Here, $log(log(n^n)) = log\ (n\ log(n))$

$log\ (n\ log\ n) \leq log\ (n^2)$ which can be written as 2 * $log\ (n)$
Hence, $log(log(n^n)) = O\ (log\ n)$.

Polynomial: a, e
Function a is n, and e is greater than n.
Hence, $O\ (2^{log\ n}) \subset O\ (n\ log(n^2))$

Exponential: b, c, f
Removing the exponentials' base, we observe for the functions (b), (c), (f):
$O(3n) \subset O\ (n\ (log\ n)^2) \subset O\ (n^2\ log\ n)$,

The rule is Logarithmic functions grow the slowest and Exponential functions grow the fastest.
Thus, the answer is:

$O(log(log(n^n)) = O\ (log\ n) \subset O\ (2^{log\ n}) \subset O\ (n\ log(n^2)) \subset O(2^{3n}) \subset O\ (n^{n\ log\ n}) \subset O(n^{n\wedge 2})$

3. Given functions *f1, f2, g1, g2* such that *f1(n) = O(g1(n))* and *f2(n) =O(g2(n)).* For each of the following statements, decide whether it is true or false and briefly explain why.

(a) *f1(n) · f2(n) = O (g1(n) · g2(n))*
(b) *f1(n) + f2(n) = O (max (g1(n), g2(n)))*
(c) *f1(n)2 = O(g1(n)2)*
(d) *log2 f1(n) = O (log2 g1(n))*

Answer:

(a) Let us assume $f1 = 2n+1$, $f2 = n^2$, $g1 = 5n+8$, $g2 = 3n^2 + 7n$
Therefore, $O(g1(n)) = n$ and $O(g2(n)) = n^2$

$f1(n) \cdot f2(n) = (2n + 1) \cdot (n^2) = 2n^3 + n^2$. This function is equal to $O(n^3)$.

Now, $g1(n) \cdot g2(n) = (5n + 8) \cdot (3n^2 + 7n) = 15n^3 + 24n^2 + 35n^2 + 56n = 15n^3 + 59n^2 + 56n.$

This function has a big O of n^3.
Therefore, *f1(n) · f2(n) = O (g1(n) · g2(n))*

Hence, it is **True.**

(b) We begin by solving the left side of the equation:

$f1(n) + f2(n) = n + n^2 = O(n^2)$

Now, out of the functions g1 and g2, we see that O(g2(n)) is greater than O(g1(n)) i.e., n^2.

Thus, *f1(n) + f2(n) = O (max (g1(n), g2(n))) i.e.,* n^2.

Hence, it is **True**.

(c) We know that *f1(n) = O(g1(n))* ----- (given)

Squaring both sides,

$f1(n)^2 = O(g1(n)^2)$.

Hence, it is **True**.

(d) Let us assume $f1 = 2$. Hence, $O(g(n)) = 1$.

Taking log on both sides,

$log_2\, f1(n) = O\ (log_2\, g1(n))$

$log_2\, 2 = O\ (\log_2 1)$

$O\ (1) = O\ (0)$

This is a contradiction, and hence, it is **False**.

4. Given an undirected graph G with n nodes and m edges, design an O(*m+n*) algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one.

Answer:

We use a modified version of Depth-First Search, where we will try to observe a back edge to a node's parent. If there exists a back edge, the graph contains a cycle. We take the help of another variable 'parent' to keep a check on the parent node if it creates a back edge. If we encounter that while visiting an adjacent node we visit a parent node, we declare that the graph has a cycle.

We assume these variables:

1. A graph G that contains *n* nodes and *m* edges.
2. '*parent*' that stores information of the parent node of the current node.

Algorithm:

```
For each node n that has not been visited:
        Mark n as visited and initialize parent = -1
        Call the function recursively for the next node n until all the nodes are visited:
                If n not equal to parent:
                        If n was previously visited:
                                Print "Graph G contains a cycle"
                                Return True
                        Else
                                Set parent to current node n and mark visited
                                Call the function recursively until all the nodes are exhausted
                Else
                        Ignore node n and consider another adjacent node of n.
                End If
        End for
End for
If no cycle is found, return False
```

Time complexity:
We are visiting each node n along with each edge m, so the time complexity for the algorithm hence is **O(m+n)**.

5. Solve Kleinberg and Tardos, **Chapter 3, Exercise 6.**

   We have a connected graph G = (V, E), and a specific vertex u ∈ V. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that G = T. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

   Answer:

   We can prove it with the help of proof by contradiction. Assume that we have an edge e = (x, y) that in G which does not exist in T. We know that T is a DFS tree, so either x or y is the parent of the either node. Since T is also a BFS tree, the nodes x and y must be separated by at most one layer. We proved earlier that either x or y is the parent and be separated by a maximum of one layer. Hence, the edge e = (x, y) must belong in the DFS and BFS tree T. This contradicts our assumption that there may be an edge out in graph G that do not belong to T.