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
```

O(n). i is cut in half during every loop O(logn), but also we loop from 1 to i each time → O(n). This might become nlogn. However, there is a tighter bound because it runs n + n/2 + n/4 + ….. = 2n - 1 = O(n).

2. **Arrange these functions under the O notation using only = (equivalent) or ⊂ (strict subset of):**
   - (a) 2^{logn}
   - (b) 2^{3n}
   - (c) n^{nlogn}
   - (d) $logn$
   - (e) $nlog(n^2)$
   - (f) n^{n^2}
   - (g) $log(log(n^n))$

Answer: $O(logn) \subset O(log(log(n^n)) \subset O(2^{logn}) \subset O(nlog(n^2)) \subset O(2^{3n}) \subset O(n^{nlogn}) \subset O(n^{n^2})$
(d) ⊂ **(g)** ⊂ **(a)** ⊂ **(e)** ⊂ **(b)** ⊂ **(c)** ⊂ **(f)**

First I put them in order (slowest to fastest)

$logn \ log(log(n^n)) \ 2^{logn} \ nlog(n^2) \ 2^{3n} \ n^{nlogn} \ n^{n^2}$

Simplified:

$logn \quad log(nlogn) \quad n \quad nlogn \quad 2^n \quad n^{nlogn} \quad n^{n^2}$

- $2^{logn} < 2^{3n}$ because $logn$ < 3n   Also → 2^{logn} = n
- $n^{nlogn} < n^{n^2}$ because nlogn < n^2
- $log(log(n^n)) < nlog(n^2)$ ... Take e power. $e^{nlog(n^2)} = e^n + e^{log(n^2)} = e^n + n^2 >. log(n^n)$ .log(nlogn)
- $logn < log(log(n^n))$ since $e^{e^{logn}} = e^n < n^n = e^{e^{log(log(n^n))}}$
- $log(log(n^n)) < 2^{logn} \rightarrow log(log(log(n^n))) < log(n)$

- $2^{logn} < nlog(n^2) \rightarrow log(2^{logn}) \ = \ logn \ < \ log(n) \ + \ log(log(n^2))$
- $nlog(n^2) < 2^{3n} \ \rightarrow log(2^{3n}) \ = \ 3n \ > \ log(n) \ + \ log(log(n^2))$

Other:

- $nlog(n^2) < n^{n^2}$. Try taking the log $\rightarrow log(n^{n^2}) \ = \ n^2 log(n)$ we can see this is larger than $n \ log(n^2)$

3. **Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.**
    (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) $f_1(n)^2 \ = \ O(g_1(n)^2)$
    (d) $log_2 f_1(n) \ = \ O(log_2 g_1(n))$

(a) **True**. If $f_1(n) = O(g_1(n))$, this means there exists a constant c such that $f_1(n)$<= c1 *$g_1(n)$. Similarly, if $f_2(n) = O(g_2(n))$, this means there exists a constant c such that $f_2(n)$<= c2 *$g_2(n)$. Thus, there exists a constant d such that $f_1(n)$ * $f_2(n)$ <= c1 * c2 * $g_1(n)$ *$g_2(n)$. e.g. $f_1(n)$ = 2n and $f_2(n)$= n, then O(n) will be n and $f_1(n)$ *$f_2(n) \ = \ 2n^2$ which satisfies $O(n^2)$.

(b) **True**. In this case, we would have, by big O definition, $f_1(n) + f_2(n)$ <= c1 *$g_1(n)$ + c2 *$g_2(n)$. If we include the max() function, it would look like:
$f_1(n) + f_2(n)$ <= c1 *$max(g_1(n), \ g_2(n))$ + c2 *$max(g_1(n), \ g_2(n))$.
$f_1(n) + f_2(n)$ <= (c1 + c2) *$max(g_1(n), \ g_2(n))$.
Thus, by definition, the equation holds.

(c) **True**. $f_1(n)^2 \ = \ O(g_1(n)^2)$. With big O notation, this means there exists a constant c such that $f_1(n)$<= c *$g_1(n)$. Now we would have $f_1(n)^2$<= c *$g_1(n)^2$. Take f(n) = 2n and g(n) = n for example. Then $4n^2 \ <= \ c \ * \ n^2$ for c >= 4.

(d) **False**. With big O notation, again, this means there exists a constant c such that $f_1(n)$<= c *$g_1(n)$. So is it always the case that $log_2 f_1(n) \ <= \ c \ * \ log_2(\ g_1(n))$? Not necessarily - consider f(n) = 5 and g(n) = 1 (constant time). Then we'd have $log_2(5) \ <= \ c \ * \ log_2(1)$. We have a conflict here because log2(1) = 0. And no constant can make that greater than log2(5).

4. **Given an <u>undirected</u> 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.**

Let us use BFS. Note that a cycle will not necessarily exist if we reach a node that we have previously visited. a -- b -- c . This has no cycle but b will check a (visited) then c. So we are looking for a <u>visited node that is not the parent node of the current node</u>, that will mean there is a cycle.

```
FindCycle(node):
    Create a boolean array Visited for visited elements with all set to False
    Set Visited[node] = True
    Create an array of nodes L
    Set L[0] = node
    set parent = node
    while L is not empty
        curr = remove node from L
        parent = curr
        Visited[curr] = True

        for all neighbor nodes of curr:
            if neighbor is not visited:
                add to L
            elif neighbor is visited and is NOT the same as the parent:
                return True
    return False
```