CSE 321 HOMEWORK 1

1) In each of the following situations, determine whether $F \in O(g)$, $f \in \Omega(g)$ or both (in which case $f \in \Theta(g)$). Provide explicit explanations for your answers. For at least half of the examples, perform a limit analysis.

a) $f(n) = 2^n$ $g(n) = 2^{2n}$

in case of $O(n)$

$0 \le 2^n \le c \cdot 2^{2n}$

for all $n > n_0$ there exist positive c, so that $f(n) \in O(g(n))$

in case of $\Omega(n)$

$0 \le c \cdot g(n) \le f(n)$

$0 \le c \cdot 2^{2n} \le 2^n$

not all $n > n_0$ there exist positive c, so that $f(n) \notin \Omega(g(n))$

$\lim_{n \to \infty} \frac{2^n}{2^{2n}} = 2^n \cdot \frac{1}{2^n} = 0$

so that $f(n) \in O(g(n))$

so, $f(n) \in O(g(n))$

b) $f(n) = n^2$ $g(n) = n^3$

$0 \le n^2 \le c \cdot n^3$

there is positive c for all $n > n_0$

$f(n) \in O(g(n))$

$n_0 = 1$

$n = 2$ | $n = 3$

$4 \le c \cdot 8$ | $8 \le c\ 27$

$0 \le c \cdot n^3 \le n^2$

there is not positive c for all $n > n_0$

$f(n) \in \Omega(g(n))$

$\lim_{n \to \infty} \frac{n^2}{n^3} = \frac{1}{n} = 0$

so that $f(n) \in O(g(n))$

so, $f(n) \in O(g(n))$

c) $f(n) = 3n + 1$ $g(n) = 2n - 5$

$0 \le 3n+1 \le 2n-5 \cdot c$

there exists for all positive c for all $n > n_0$ so that

$f(n) \in \Theta(g(n))$

$0 \le c \cdot (2n-5) \le 3n+1$

there exists for all positive c for all $n > n_0$ so that

$f(n) \in \Omega(g(n))$

$\lim_{n \to \infty} \frac{3n+1}{2n-5} = \frac{3}{2}$

constant, thus:

$f(n) \in \Theta(g(n))$

So that $f(n) \in \Theta(g(n))$

d). $f(n) = 4n^2$ $g(n) = n^2$

for all $n > n_0$ there exist positive constant c,
$$0 \le 4n^2 \le c \cdot n^2$$
so that
$f(n) \in O(g(n))$

$$0 \le c \cdot n^2 \le 4n^2$$
for all $n > n_0$ there exists positive constant c,
So that $f(n) \in \Omega(g(n))$

$$\lim_{n \to \infty} \frac{4n^2}{n^2} = 4$$
So that
$f(n) \in \Theta(g(n))$

so, $f(n) \in \Theta(g(n))$

e) $f(n) = \log 2(n)$ $g(n) = \log_{10}(n)$

$$0 \le \log_2(n) \le c \cdot \log_{10}(n)$$
for all $n > n_0$ there exists positive constant c,
so that
$f(n) \in O(g(n))$

$$0 \le c \cdot \log_{10}(n) \le \log_2(n)$$
for all $n > n_0$ there exists positive constant c,
that
$f(n) \in \Omega(g(n))$

$$\lim_{n \to \infty} \frac{\log_2(n)}{\log_{10}(n)} = \frac{\log_2(n)}{\frac{\log_2(n)}{\log_2(10)}}$$
$$= \lim_{n \to \infty} \log_2(10)$$
constant
so, $f(n) \in \Theta(g(n))$

So that $f(n) \in \Theta(g(n))$

f) $f(n) = 2^n$ $g(n) = 3^n$

$$0 \le 2^n \le c \cdot 3^n$$
$n > n_0$ $n_0 = 1$
$c = 1$
$n = 2$
$4 \le 9$
$n = 3$
$8 \le 27$
So that
$f(n) \in O(g(n))$

$$0 \le c \cdot 3^n \le 2^n$$
There are no n value such that $n > n_0$ which there exist positive c constant.
$f(n) \notin \Omega(g(n))$

$$\lim_{n \to \infty} \frac{2^n}{3^n} \quad \lim_{n \to \infty} \left(\frac{2}{3}\right)^n = 0$$
$f(n) \in O(g(n))$

$f(n) \in O(g(n))$

g) $n^3 = f(n)$, $1000n^2 = g(n)$

$$0 \le n^3 \le c \cdot 1000n^2 \quad n > n_0$$
$$0 \le n \le 1000 \cdot c$$
there are no constants C and n_0 satisfy the condition.
so that
$f(n) \notin O(g(n))$

$$0 \le c \cdot 1000n^2 \le n^3$$
$$c \cdot 1000 \le n$$
For all n such that $n > n_0$ there exists constant positive c
so that
$f(n) \in \Omega(g(n))$

$f(n) \in \Omega(g(n))$

h) $5n+4 = f(n)$, $g(n) = 2n+2$

For all $n > n_0$

$5n+4 \leq c \cdot (2n+2)$

There exists positive constant c,

so that $f(n) \in O(g(n))$

$n_0 = 1$ $9 \leq 40$ $n=1$

$c = 10$ $14 \leq 60$ $n=2$

$29 \leq 120$ $n=5$

$0 \leq c \cdot (2n+2) \leq 5n+4$

For all $n > n_0$

$c\,(2n+2) \leq 5n+4$

There exists positive constant c,

so that $f(n) \in \Omega(g(n))$

$f(n) \in \Theta(g(n))$

---

i) $f(n) = \sqrt{n}$, $g(n) = \log_2(n)$

$$\lim_{n\to\infty} \frac{\sqrt{n}}{\log_2(n)} = \frac{\frac{1}{2\sqrt{n}}}{\frac{1}{\ln 2 \cdot n}} = \frac{\ln 2 \cdot n}{2\sqrt{n}} = \frac{\ln 2 \cdot n^{\frac{1}{2}}}{2} = \sqrt{n} \to \infty$$

so that $f(n) \in \Omega(g(n))$

---

j) $2^n = f(n)$, $g(n) = 2^{n+1}$

$$\lim_{n\to\infty} \frac{2^n}{2^{n+1}} = 2^{-1} = \frac{1}{2} \to f(n) \in \Theta(g(n))$$

---

2) List the following functions in order of their growth and provide proof for your claims.

$\frac{1}{2n} < \log(n) < \sqrt{n+5} < n+1 < n^2\log(n) < 2^n < 10^n < n! < n^{2^n}$ → double exponential functions are super fast.

$\lim_{n\to\infty} \frac{1}{2n}$ → it is a decreasing function. → So it is even not growing. → $n=1 \to \frac{1}{2}$, $n=2 \to \frac{1}{4}$, $n=8 \to \frac{1}{16}$

logarithmic order: it is a growing function. but it grows very slow → $n=1 \to 0$, $n=20 \to 1$, $n=100 \to 2$, $n=1000 \to 3$

it grows faster than logarithmic functions. → $n=4 \to 3$, $n=20 \to 5$, $n=31 \to 6$, $n=95 \to 10$

it is a linear function. → $n=4 \to 5$, $n=100 \to 101$

Exponential functions grows fast. → $n=1 \to 10$, $n=2 \to 100$, $n=5 \to 100000$

Quadratic order multiplied by growing function is for sure faster than just quadratic.

Factorial order functions are very fast compared to exponential.

3) Provide pseudo code for the following operations on a given binary search tree (BST) with a height of n. Analyze the time complexity (with Big-OH notation) of your code for each of the following:

a) Merging with another BST of height n.

```
def merge_bst (root1, root2): // Get the numbers from the trees and merge them into
                              // one list and reconstruct the tree.
    result1 = []
    result2 = []
    inorder_traversal(root1, result1) -> O(n)
    inorder_traversal (root2, result2) -> O(n)
    merged = sorted (result1 + result2) -> O(nlogn)
    return create_balanced_bst (merged) -> O(nlogn)
```

$T(n) = O(2n + 2n\log n) = O(n\log n)$

```
def inorder_traversal(root, result):
    if not root:
        return
    inorder_traversal(root.left, result)
    result.append(root.val)
    inorder_traversal(root.right, result)
```

Traverses all the elements of the BST recursively because of that $\Rightarrow T(n) = O(n)$

```
def create_balanced_bst (a_list):
    a_list = sorted (a_list)
    return create_balanced_bst_helper (a_list) -> O(n)
```

it sorts the list = $O(n \log n)$

$T(n) \Rightarrow O(n\log n + n) \Rightarrow O(n\log n) = T(n$

```
def create_balanced_bst_helper (a_list):
    if not a_list:
        return
    mid_idx = int (len(a_list) / 2)
    root = Node (val = a_list [mid_idx])
    root.left = create_balanced_bst_helper(a_list [:mid_idx])
    root.right = create_balanced_bst_helper (a_list [mid_idx+1 :])
    return root
```

Recursively constructs BST from a list at each level of recursion it processes the half of the list. Eventually every element is processed once. So it is $O(N)$ where N is the size of the list

b) Finding the kth smallest element in the BST.

```
def find_kth_smallest (root, k):
    stack = [ ]
    while True:
        while root:
            stack.append (root)
            root = root.left

        if not stack:
            return

        root = stack.pop()
        k -= 1
        if k == 0:
            return root.val
        root = root.right
```

→ Stack operations in while loop will iterate through all the nodes in the bst in the worst case.

$T(n) = O(n)$

C) Balancing the BST.

```
def balance_BST (root):

    result = [ ]
    inorder_traversal (root, result)   → O(n)
    root = create_balanced_bst_helper (result)
    return root
```

$T(n) = O(2n) = O(n)$

I used helper one because it doesn't sort. ⇒ $O(n)$

d) Finding elements within a specified value range

```
def find_elements_within_a_range(root, lower_bound, upper_bound):

    if lower_bound > upper_bound:
        return

    stack = [ ]
    result = [ ]
    while True:
        while root:
            stack.append(root)
            root = root.left

        if not stack:
            return result

        root = stack.pop()

        if lower_bound <= root.val <= upper_bound:
            result.append(root.val)

        if root.val > upper_bound:
            return result

        root = root.right
```

→ For worst case the while loop will iterate all the nodes of the tree so that:

$T(n) \in O(n)$

---

4) Calculate the time complexity in O (big oh) of the following program.

```
i=2
while i <= n:
    if i % 2 != 0:
        i = i - 1
    else:
        i = i * i
        i = i + 1
print(i)
```

→ if the $i \leq n$ condition satisfied at the first iteration i value is going to be 5.

→ And then we will check the condition again, if it is satisfied i value is going to be 4

→ satisfied
i value = 17

→ satisfied
i value = 16

→ satisfied
i value = 257

↓
continues like this

5
4
17
16
257
256
65537
65536
⋮

n=1	n=2	n=3	n=4	n=5	n=17	n=257
None	5	5	5	5 4 17	5 4 17 16 257	5 4 17 16 257 256 65537

the growth rate of number of iterations are very slow.
So that,
we can say: $T(n) = \log_2(n-1)$

$T(n) \in O(\log_2(n))$

5) Suppose you have an array of n elements, where each element can be either even or odd with a probability distribution of %20 even and %80 odd. Propose an algorithm that identifies the first even element in the array. Describe the algorithm and analyze its time complexity of average-case.

$$A(n) = \sum_{i=1}^{n} i \cdot P_i$$

$i =$ Number of comparisons at the i'th item.

$P_i =$ Probability for the item at the i'th to be found.