Name & Surname:
ID:

CSE 321 - Quiz #3

November 8^{th}, 2024

1. ***45 pts.*** Solve the following recurrence relations by using Master Theorem and determine Θ bound for each of them. If any of the relations cannot be solved by using Master Theorem, explain why.

   (a) $T(n) = 4 \cdot T(\frac{n}{2}) + \sqrt{3n}$

   (b) $T(n) = 3 \cdot T(\frac{n}{9}) + \sqrt{n}$

   (c) $T(n) = 3 \cdot T(\frac{n}{2}) + 5 \cdot n^3$

Solution:

$$T(n) = aT(n/b) + f(n), \qquad \textbf{(5.1)}$$

(Here, a and b are constants; $a \geq 1$ and $b > 1$.)

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

(a) $a = 4 \geq 1$ ✓
$b = 2 > 1$ ✓
$f(n) \in \Theta(\sqrt{n})$, thus, $d = \frac{1}{2} \geq 0$ ✓

Since $a = 4$ and $b^d = \sqrt{2} \approx 1.41$, $a > b^d$.
Therefore, the solution is $\Theta(n^{log_b a}) = \Theta(n^{log_2 4}) = \Theta(n^2)$

(b) $a = 3 \geq 1$ ✓
$b = 9 > 1$ ✓
$f(n) \in \Theta(\sqrt{n})$, thus, $d = \frac{1}{2} \geq 0$ ✓

Since $a = 3$ and $b^d = \sqrt{9} = 3$, $a = b^d$.
Therefore, the solution is $\Theta(n^d \cdot logn) = \Theta(\sqrt{n} \cdot logn)$

(c) $a = 3 \geq 1$ ✓
$b = 2 > 1$ ✓
$f(n) \in \Theta(n^3)$, thus, $d = 3 \geq 0$ ✓

Since $a = 3$ and $b^d = 2^3 = 8$, $a < b^d$. Therefore, the solution is $\Theta(n^d) = \Theta(n^3)$.

2. ***45 pts.*** Following algorithms calculate a^n for given integers a and n, where $n > 0$. Analyze the average-case time complexity of these algorithms.

(a)

```
PowCalculator(a, n):
    Input: An integer a and a positive integer n.
    Output: a^n.
    begin
        if n == 1 then
            return a
        end if
        if n % 2 == 1 then
            return PowCalculator(a, n-1) * a
        end if
        b ← PowCalculator(a, n/2)
        return b*b
    end
```

(b)

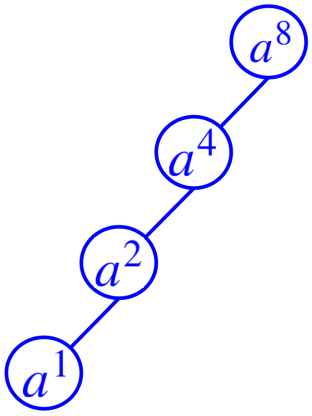
```
PowCalculator(a, n):
    Input: An integer a and a positive integer n.
    Output: a^n.
    begin
        if n == 1 then
            return a
        end if
        if n % 2 == 1 then
            return PowCalculator(a, n-1) * a
        end if
        return PowCalculator(a, n/2)*PowCalculator(a, n/2)
    end
```

(c)

```
PowCalculator(a, n):
    Input: An integer a and a positive integer n.
    Output: a^n.
    begin
        if n == 1 then
            return a
        end if
        return a*PowCalculator(a, n-1)
    end
```

Solution:

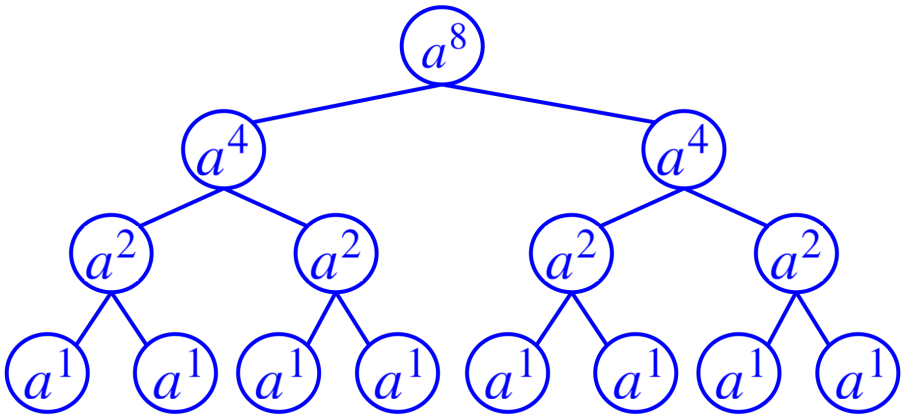
(a) This algorithm follows a tree-like structure to compute a^n. For instance, if $n = 8$, we have the following tree:



In order to calculate a^8 we should reach the leaf and then go upwards:
- By multiplying a^1 with itself, we obtain a^2.
- By multiplying a^2 with itself, we obtain a^4.
- By multiplying a^4 with itself, we obtain a^8.

As seen, the number of steps is equal to the height of this tree. Or, to be more specific, it is equal to the $2 \cdot height$ since we go to the leaf and then come back to the root. Since such a binary tree has a height of $logn$, the average and worst-case time complexity is equal to $\Theta(logn)$. The best case time complexity is the special case where we have only 1 node in the tree (where $n = 1$) and is constant time.

(b) This algorithm also follows a tree-like structure. It is quite similar to the algorithm in question $2 - a$. The only difference is that we make the recursive function call twice. In question $2 - a$, we do this only once and save the value in variable b. But here, we don't do that. Therefore, we have the following tree:



As seen, one single code line changed the whole structure of the tree. Instead of following a single path as in question $2 - a$, now we have to reach all of the leaves. For instance: in order to compute the leftmost a^2 node, we should calculate both of its child nodes. We can only calculate a^2 by multiplying a^1 from the left child and a^1 from the right child. This means that we have to go over all of the nodes. As known, in a perfect binary tree, the number of internal nodes is $n - 1$, where n is the number of leaves. Thus, in this tree, we have $2 \cdot n - 1$ nodes and we have to visit all of them. Again, to be more specific, we visit the internal nodes twice, since

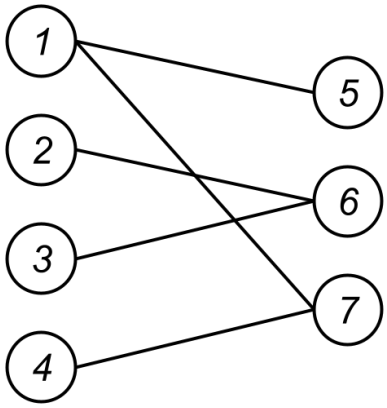
we first reach to leaves and then come back to the root. But it will converge to $\Theta(n)$ anyways (in average and worst-case). The best-case time complexity is constant since we only have one node in the tree in the best-case scenario (where $n = 1$).

(c) This algorithm follows a sequential logic where we repeatedly multiply a by itself n times. Therefore, the average and the worst-case time complexity is $\Theta(n)$ while the best-case is a specific scenario with constant time.

It must be noted that the algorithm in question $2 - b$ has also $\Theta(n)$ complexity but in practice, it requires much more operations.

3. ***10 pts.*** A bipartite graph is a graph whose vertex set can be partitioned into two disjoint sets of vertices such that no two vertices in the same set are adjacent.

   For instance, the following is a bipartite graph. We can say that the vertex set can be partitioned into two disjoint sets, $X = \{1, 2, 3, 4\}$ and $Y = \{5, 6, 7\}$. As seen, the vertices of X are not adjacent. The same thing can also be said for Y.



   Design a BFS-based algorithm to check whether a given graph G is bipartite or not. Provide the pseudo-code of your algorithm and analyze its worst-case time complexity.

   Solution: A bipartite graph is also called a 2-colorable graph. A 2-colorable graph is a graph that can be colored using two distinct colors such that no two adjacent vertices share the same color. In other words, we use the same color for the vertices of a disjoint subset. Now, we can provide a BFS-based solution to the graph coloring problem as below:

```
IsBipartite(G):
    Input: A graph G.
    Output: True or False, indicating that the graph G is bipartite or not.
    begin
        colors ← [-1] * V
        for each vertex v in G repeat
            if color[v] == -1 then
                Initialize a queue, with v
                color[u] ← 0
                while queue is not empty repeat
                    u ← dequeue q
                    for each neighbor v of u repeat
                        if color[v] == -1 then
                            color[v] = 1 - color[u]
                            enqueue v to queue
                        else
                            if color[v] == color[u] then
                                return False
                            end if
                        end if
                    end for
                end while
            end if
        end for
        return True
    end
```

This algorithm follows a BFS logic and colors the vertices one by one (with color 0 and color 1). If two adjacent vertices have the same color, the algorithm returns *False*. The worst-case scenario is when the graph is bipartite. In this case, the algorithm has to finish all of the loops and then return *True*. In such a case, even though there are multiple loops in the provided solution, we can see that the algorithm checks every node and every edge only once. The outer for loop visits every node, while the inner for loop checks the edges in order to find the neighbors. Thus, the worst-case time complexity becomes $O(|V| + |E|)$, where V is the set of vertices of graph G and E is the set of edges of graph G.