Study Guide Midterm `ris:Global`

Reference

#Worksheet 1 + 2 `ris:Bookmark3`

1.1 Algorithm analysis, Big-O, Big-Omega, and Big-Theta


```
▪ Logarithmic algorithm – **O(logn)** – Binary Search.

▪ Linear algorithm – **O(n)** – Linear Search.

▪ Superlinear algorithm – **O(nlogn)** – Heap Sort, Merge Sort.

▪ Polynomial algorithm – **O(n^c)** – Strassen’s Matrix Multiplication,
Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.

▪ Exponential algorithm – **O(c^n)** – Tower of Hanoi.

▪ Factorial algorithm – **O(n!)** – Determinant Expansion by Minors, Brute
force Search algorithm for Traveling Salesman Problem.
```

1.2 Evaluating nested for loops

O(1) - Constant Complexity

This function runs in *O(1)* time (or *constant time*). Because this method takes the same amount of time to complete irrespective of the input size, we ignore *n* entirely when expressing the complexity. The input array could contain 1 item or 1,000 items, but this function would still just require one "step."

```
std::cout << "anhyeuem" << std::endl;

for (int i = 1; i <= c; i++)
{
    // some O(1) expressions
}
```

O(n) - Linear Complexity

This function runs in *O(n)* time (or *linear time*), where *n* is the number of items in the array. If the array has 10 items, the function calls `std::cout ...` 10 times. If it has 1,000 items, it calls it 1,000 times.

```
// loop 1
for (int i = 0; i < n; i++)
{
    std::cout << "anhyeuem" << std::endl;
}

// loop 2
for (int i = 1; i <= n; i += c)
{
    // some O(1) expressions
}

// loop 3
for (int i = n; i > 0; i -= c)
{
    // some O(1) expressions
}
```

O(n^2) - Quadratic Complexity

If our array has *n* items, our outer loop runs *n* times and our inner loop runs *n* times *for each iteration of the outer loop*, giving us *n^2* total calls to `std::cout...`. Thus this function runs in *O(n2)* time (or *quadratic time*). If the array has 10 items, we have to print 100 times. If it has 1,000 items, we have to print 1,000,000 times.

```
// loop in loop 1
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        // do something
        std::cout << "anhyeuem" << std::endl;
    }
}

// loop in loop 2
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= n; j++)
    {
        // some O(1) expressions
    }
}

// loop in loop 3
for (int i = n; i > 0; i--)
{
    for (int j = i + 1; j <= n; j++)
    {
        // some O(1) expressions
```

```
    }
}
```

O(log n) - Logarithmic Complexity

Here we have simple loop in which the post operation of the `for` statement multiples the current value of `i` by 2, so `i` goes from 1 to 2 to 4 to 8 to 16 to 32 and so on.

O(log n) is considered to be fairly efficient. The time taken increases with the size of the data set, but not proportionately so. This means the algorithm takes longer per item on smaller datasets relative to larger ones. 1 item takes 1 second, 10 items takes 2 seconds, 100 items takes 3 seconds. If your dataset has 10 items, each item causes 0.2 seconds latency. If your dataset has 100, it only takes 0.03 seconds extra per item. This makes *O(log n)* algorithms very scalable.

```
// loop 1
for (int i = 1; i <= n; i *= 2)
{
    // do something
}

// loop 2
for (int i = 1; i <= logn; i++)
{
    // do something
}

// loop 3
for (int i = 1; i <= n; i *= c)
{
    // some O(1) expressions
}

// loop 4
for (int i = n; i > 0; i /= c)
{
    // some O(1) expressions
}
```

1.3 Reorder functions from smallest growth rate to largest growth rate

1.4 L'Hˆopital's Rule

```
// Looking for the dominant term
f(n) = O(g(n)) -- f(n) <= g(n) asymtotically, Upper bound, Limiter

g(n) = Ω(f(n)) -- f(n) == g(n) asymtotically, Exact bound, Tie Breaker

f(n) = θ(g(n)) -- f(n) >= g(n) asymtotically, Lower Bound, Starter
```

```
lim       f(n)
         -----
n-> ∞     g(n)

- If the result is ∞                then f(n) = θ(g(n)) or f(n) = Ω (g(n))

- If the result is c > 0 (constant) then f(n) = Ω(g(n)  or f(n) = θ(g(n)
|| then f(n) = O(g(n)

- If the result is 0 (toward it)    then f(n) = O(g(n)) or g(n) = Ω(f(n))
```

![[commonderivatives.png]]

![[commonderivatives2.png]]

1.5 [[Sort Algorithm]]

#Worksheet 3 `ris:Bookmark3`

^533d88

Binary Tree height

[[Search Algorithm#^b6c7f9|Binary Search Tree]]

1. A perfect binary tree of height h has `2^(h + 1) − 1` node.
2. A perfect binary tree with n nodes has height `log(n + 1) − 1 = θ(ln(n))`.
3. A perfect binary tree of height h has `2^h` leaf nodes.
4. The average depth of a node in a perfect binary tree is `θ(ln(n))`.

Max

- Max amount of **leaves** for a binatry tree = `2^h`
- Max **nodes** = `2^(h + 1) − 1`
- Max **non leaves** nodes = (Max nodes - leaves)

The leftmost child
Amount of total nodes and leafs nodes for a balanced binary tree are **asymptotically** the same
Off by constant factor but not asymptotically different.

Min

The rightmost child
If the tree is unbalanced, then the root is the smollest

Best case : O(logn)
Worst case: O(n)

Write a function that calculate the **height** of the tree

```
int height(binTreeNode *root)
{
    if (root == nullptr)
    {
        return -1;
    }

    int lh = heigh(root->left);
    int rh = heigh(root->right);

    return std::max(lh, rh) + 1;
}

or

int height(binTreeNode *root) const
{

    if (!r)
    {
        return -1;
    }
    else
    {
        1 + std::max(height(root>left), height(root->right));
    }
}
```

![[heightTree.png|750]]

Traversals (inorder, preorder, postorder)

Preorder

```
void BinarySearchTree::print() const
{
    if (root != nullptr)
    {
        std::cout << root->data << std::endl;
        print(root->left);
        print(root->right);
    }
}
```

Inorder

```
void BinarySearchTree::print() const
{
    if (root != nullptr)
    {
        print(root->left);
        std::cout << root->data << std::endl;
        print(root->right);
    }

}
```

Postorder

```
void BinarySearchTree::print() const
{
    if (root != nullptr)
    {
        print(root->left);
        print(root->right);
        std::cout << root->data << std::endl;
    }

}
```

Compute the number of leaves, interior nodes, and total nodes

Full Nodes

Write the function that count the amount of **full nodes** (nodes that have 2 childrens)

```
int fullNodes(binTree Node *root)
{
    if (root == nullptr)
    {
        return 0;
    }

    if (root->left != nullptr && roo->right != nullptr)
    {
        return 1 + fullNodes(root->left) + fullNodes(root->right);
    }

    return fullNodes(root->left) + fullNodes(root->right);
}
```

Leaves Nodes

Write the function that count the **leaves nodes** (nodes that have no childrens)

```
int leafNodes(binTree Node *root)
{
    if (root == nullptr)
    {
        return 0;
    }

    if (root->left == nullptr && roo->right == nullptr)
    {
        return 1 + leafNodes(root->left) + leafNodes(root->right);
    }

    return leafNodes(root->left) + leafNodes(root->right);
}
```

Total nodes

Write the function that count the **total nodes** (everything)

```
int totalNodes(binTree Node *root)
{
    if (root == nullptr)
    {
        return 0;
    }
    return 1 + totalNodes(root->left) + totalNodes(root->right);
}
```

Interior nodes

Write the function that count the **interior nodes** (non leaf nodes)

```
int interiorNodes(binTreeNode *root)
{
    if ((root == nullptr) || (root->left == nullptr && root->right ==
nullptr))
    {
        return 0;
    }
    return 1 + interiorNodes(root->left) + interiorNodes(root->right);
}
```

Half nodes

Write the function that count the **half nodes** (nodes with one child)

```
int halfNodes(binTreeNode *root)
{
    if (root->nullptr)
    {
        return 0;
    }
    int half = 0;

    if ((root->left == nullptr && root->right != nullptr) || (root->left
!= nullptr && root->right == nullptr) )
    {
        half++;
    }

    half += (halfNodes(root->left) + halfNodes(root->right));
    return half;
}
```

Pray to god

#Worksheet 4 ris:Bookmark3

AVL Tree

Tree Reference

![[AVL Rotations.png]]

the height is always O(logn) -- prove for it (believe me ask anybody)

Worst case: 2 * log2n

rotations and resulting tree after the rotations

Write a function that insert a node with a key given

```
binTreeNode *insert (binTreeNode *root, int value)
{
    if (root == nullptr)
    {
        binTreeNode *temp = new binTreeNode;
        temp->data = value;
        temp->left = nullptr;
        temp->right = nullptr;
        return temp;
    }

    if (value > root->data)
    {
```

```
            root->right = insert(root->right, value);
        }
        else if (value < root->data)
        {
            root->left = insert(root->left, value);
        }

        return root;
    }
```

Random

binsearchtree

left < key
right > key

how to insert stuff into AVL tree,

min -- leftmost
max -- rightmost

if there is no right side, then the root is the largest, since it is the rightmost at the moment

15-20 questions

Reorder the following function from the smallest to largest

n + n√n - 3 --> Θ(n√n)

lnn! --> Θ(nlogn)

2.1^n --> Θ(2.1^n)

36000 --> O(1)

nlog2n + 3n√n - n --> Θ(n√n)

2^n+1 --> Θ(2^n)

+1 is not gonna be a big enough factor too affect the growth rate

`Reorder them as of right now`

1st: 36000

2nd: lnn!

3rd: n + n√n - 3 && nlog2n + 3n√n - n

4th: 2.1^n

5th: 2^n+1

For **case 3** If the base is different, the growth is bigger as well when it comes to exponential

---