**Boris Bespartochnyy, bpb56, Written Assignment 2**

# Question 1:

## Prove $3n^2 - 7n = \Theta(n^2)$ using the inequality definition.

**Step 1: Identify $f(n)$ and $g(n)$**

Let $f(n) = 3n^2 - 7n$ and $g(n) = n^2$. We need to find positive constants $c_1, c_2$, and $n_0$ such that:

$$c_1 n^2 \leq 3n^2 - 7n \leq c_2 n^2 \quad \text{for all } n \geq n_0.$$

**Step 2: Establish the upper bound**

We show that $3n^2 - 7n \leq c_2 n^2$. Factoring out $n^2$:

$$3n^2 - 7n = n^2 \left(3 - \frac{7}{n}\right).$$

For large enough $n$, the term $\frac{7}{n}$ becomes small. Since $3 - \frac{7}{n} \leq 3$ for all $n \geq 1$, we can choose $c_2 = 3$, ensuring:

$$3n^2 - 7n \leq 3n^2 \quad \text{for all } n \geq 1.$$

**Step 3: Establish the lower bound**

We show that $3n^2 - 7n \geq c_1 n^2$. Again, rewriting:

$$3n^2 - 7n = n^2 \left(3 - \frac{7}{n}\right).$$

For large enough $n$, $\frac{7}{n}$ is small, so $3 - \frac{7}{n} \geq 2.5$ for all $n \geq 7$. Thus, choosing $c_1 = 2.5$, we get:

$$3n^2 - 7n \geq 2.5n^2 \quad \text{for all } n \geq 7.$$

**Conclusion**

We have shown that for $n \geq 7$,

$$2.5n^2 \leq 3n^2 - 7n \leq 3n^2.$$

Thus, by the definition of Big-Theta notation,

$$3n^2 - 7n = \Theta(n^2).$$

# Question 2: Part A

## Summation expression or recurrence relation:

In the provided code snippet, there are two loops:

- The outer loop runs from 1 to $n$, iterating $n$ times.
- The inner loop runs from 1 to 3, iterating 3 times for each iteration of the outer loop.

For each iteration of the outer loop, the inner loop runs a constant number of times (3). Therefore, the total number of operations is:

$$T(n) = \sum_{i=1}^{n} 3 = 3n$$

## Simplify the summation or recurrence to a closed form:

The closed form is simply the summation evaluated:

$$T(n) = 3n$$

## Give a tight bound on the closed form solution:

The tight bound for this closed form is $O(n)$, because the runtime grows linearly with $n$.

# Question 2: Part B

## Summation expression or recurrence relation:

The function foo2Rec is a recursive function that processes each index $i$ from 0 to $n-1$. It performs two recursive calls for each index:

- One for the next index $i+1$ after setting the value to true.
- Another for the next index $i+1$ after setting the value back to false.

Thus, at each step, the function makes two recursive calls. Therefore, the recurrence for the runtime is:

$$T(n) = 2T(n-1) + O(1)$$

With a base case of $T(0) = O(1)$, since the function terminates when $i = n$.

## Simplify the summation or recurrence to a closed form:

We will solve this recurrence by expanding it:

$$T(n) = 2T(n-1) + O(1)$$
$$T(n-1) = 2T(n-2) + O(1)$$
$$T(n-2) = 2T(n-3) + O(1)$$

...

If we expand this until we reach $T(0)$, we get:

$$T(n) = 2^n T(0) + O(2^n)$$

Since $T(0) = O(1)$, this simplifies to:

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

## Give a tight bound on the closed form solution:

The tight bound for this recurrence is $O(2^n)$, because the runtime grows exponentially with $n$.

## Question 3:

Given a list $L$ with $n$ unique positive integers, check whether there exists two pairs of integers $(p, q)$ and $(r, t)$, where $p, q, r, t \in L$ and are less than some constant $k$, such that: $p + q = r + t$

We can solve this problem by using a hash table to store sums of all pairs of integers in the list $L$. For each pair $(p, q)$, we check if there exists another pair $(r, t)$ such that $p + q = r + t$.

The idea is to iterate over all pairs of numbers in the list $L$ and compute their sums. For each sum, we check if the sum has appeared before. If it has, then we have found two pairs with equal sums and return true. Otherwise, we store the sum in the hash table and continue.

```
def find_equal_sum_pairs(L, k):
# Dictionary to store sums of pairs
sum_map = {}

# Iterate over all pairs (p, q)
for i in range(len(L)):
    for j in range(i + 1, len(L)):
        p = L[i]
        q = L[j]

        # Ensure p and q are both less than k
        if p < k and q < k:
            pair_sum = p + q

            # Check if the sum already exists in the dictionary
            if pair_sum in sum_map:
                # We found a match, return True
                return True
            else:
                # Store the sum with the pair (p, q)
                sum_map[pair_sum] = (p, q)

# If no pairs with equal sum are found, return False
return False
```

## Time Complexity Analysis

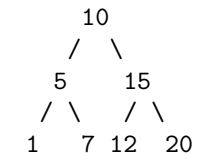
The time complexity of this algorithm is $O(n^2)$, where $n$ is the number of elements in the list $L$. This is because we are iterating through all pairs of integers in $L$, which results in $O(n^2)$ pairs. Inserting and checking the hash table takes constant time on average, assuming no collisions.

# Question 4:

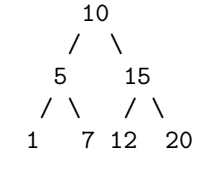
## (a) Sequences of unique integers generating the same BST

Let A = (10, 5, 1, 7, 15, 12, 20) and $A_0$ = (5, 1, 7, 10, 12, 20, 15).

Inserting the elements of A into an empty binary search tree in the given order results in the following tree structure:




Inserting the elements of $A_0$ into an empty binary search tree in the given order results in the same tree structure:




## (b) Algorithm to Build an Almost Balanced BST from a Sorted Array

- Choose the middle element of the array, $a_{\lfloor \frac{n}{2} \rfloor}$, as the root of the BST.
- Recursively build the left subtree using the left half of the array, and the right subtree using the right half.

The algorithm is as follows:

- **Input:** A sorted array $A$ of $n$ unique integers.
- **Output:** An almost balanced binary search tree.
- Base case: If the array has one element, create a node with that element.
- Recursive step: Select the middle element $A[\lfloor \frac{n}{2} \rfloor]$ as the root.
- Recursively build the left and right subtrees.

**Why this works:** This approach ensures that for each node, the left and right subtrees will have a size difference of at most 1, resulting in an almost balanced BST.

**Runtime analysis:** The algorithm recursively divides the array into two halves, and at each step, it processes a constant number of operations. Therefore, the total time complexity is:

$$\Theta(n)$$

## (c) Algorithm to Convert a BST into an Almost Balanced BST

To convert a given binary search tree (BST) into an almost balanced BST, the following algorithm is used:

- First, perform an in-order traversal of the BST to retrieve the elements in sorted order.
- Then, build an almost balanced BST from the sorted array of elements, using the method described in part (b).

The algorithm is as follows:

- **Input:** A BST with $n$ nodes.
- **Output:** An almost balanced BST.
- Step 1: Perform an in-order traversal of the BST to get the sorted array.
- Step 2: Build an almost balanced BST from the sorted array using the method from part (b).

**Why this works:** The in-order traversal guarantees that we get the elements of the BST in sorted order. Rebuilding the tree from this sorted array ensures that the new tree is almost balanced.

**Runtime analysis:** The in-order traversal takes $\Theta(n)$ time, and the construction of the almost balanced BST from the sorted array also takes $\Theta(n)$ time. Therefore, the total time complexity is:

$$\Theta(n)$$

# Question 5:

# Priority Queue Implementation Using Stacks

Since stacks allow for operations such as push(x), pop(), and isEmpty(), we can use two stacks to simulate the behavior of a priority queue.

**1. Insert Operation:**

To insert an element, we will use a stack to maintain the priorities in sorted order. This ensures that the minimum priority is always at the top of the stack. The procedure is as follows:

1. Start by pushing the new priority into stack1.
2. Compare the new priority with the top element of stack2. If the new priority is smaller, push it onto stack2.
3. If the new priority is larger, pop elements from stack2 and push them onto stack1 until the new priority is less than or equal to the top element of stack2. Then, push the new priority onto stack2.
4. Push any remaining elements from stack1 back into stack2.

**Time Complexity:** The insert operation has a worst-case time complexity of $O(n)$ because in the worst case, we need to move all elements between the two stacks.

**2. ExtractMin Operation:**

To extract the minimum element, we simply pop the top element from stack2, as it is always the minimum element in the priority queue. After the element is removed, we return it.

**Time Complexity:** The extractMin operation has a time complexity of $O(1)$ because we are always popping the top element from stack2, which takes constant time.

## Runtime Analysis:

- insert(priority): $O(n)$ in the worst case, as we may need to push elements between the two stacks.
- extractMin() : $O(1)$ because we only pop the top element of stack2.

In conclusion, the insertion operation is less efficient with a time complexity of $O(n)$ due to the need to maintain the stack's sorted order. However, the extractMin() operation is efficient with a time complexity of $O(1)$, making this implementation suitable for scenarios where we need to frequently extract the minimum priority.

# Question 6:

# Heap Insertion Algorithm

## Insertion Algorithm:

Given the heap with $n$ nodes and a new value to insert, the basic idea is to perform an efficient insertion while maintaining the heap property. We can leverage the following steps:

1. Start by inserting the new value as a leaf node. This can be done by adding the new node as the left or right child of an appropriate leaf node. We do not need to traverse all the way to the last level of the tree; instead, we can insert the new node at any leaf and then perform the necessary adjustments to restore the heap property.

2. After inserting the new node, perform the upheap operation starting from the newly inserted node. The upheap operation compares the value of the new node with its parent and swaps them if the heap property is violated. This process continues until the heap property is restored, i.e., the value of the node is no longer smaller (for min-heap) or larger (for max-heap) than its parent, or the node becomes the root.

3. The key observation here is that we do not need to traverse the entire tree to find the correct position for the new node. By choosing an appropriate leaf node, the insertion can be done efficiently.

## Time Complexity Analysis:

Let $h$ be the height of the heap. Since a binary heap is a complete binary tree, the height $h$ of the heap is $O(\log n)$, where $n$ is the number of nodes in the heap.

- Insert Operation: The insertion step involves adding the new node as a leaf, which can be done in constant time, $O(1)$.

- Upheap Operation: The upheap operation starts from the inserted node and compares it with its parent, moving up the tree and swapping if necessary. In the worst case, this operation might need to traverse the height of the tree, which is $O(\log n)$.

Thus, the overall time complexity for the insertion operation is $O(\log n)$.

## Conclusion:

The insertion operation can be performed efficiently in $O(\log n)$ time by adding the new node as a leaf and then using the upheap operation to restore the heap property. This approach is more efficient than a linear-time insertion because it exploits the structure of the binary heap and reduces the number of operations required.