Yogendra Pokhrel
Student ID: 617585

1.Determine the asymptotic running time of the following procedure (an exact computation of number of basic operations is not necessary):

```
int[] arrays(int n) {
   int[] arr = new int[n];
   for(int i = 0; i < n; ++i){
       arr[i] = 1;
   }
   for(int i = 0; i < n; ++i) {
       for(int j = i; j < n; ++j){
           arr[i] += arr[j] + i + j;
       }
   }
   return arr;
}
```

Answer:

To analyze the asymptotic running time of the given procedure, let's break it down step by step:

1. **First Loop (Initialization):**

```
for(int i = 0; i < n; ++i){
    arr[i] = 1;
}
```

This loop runs **O(n)** times, setting each element of the array to 1.

2. **Nested Loops (Computation):**

```
for(int i = 0; i < n; ++i) {
    for(int j = i; j < n; ++j){
        arr[i] += arr[j] + i + j;
    }
}
```

   - The outer loop runs **O(n)** times.
   - The inner loop starts from `i` and goes up to `n - 1`. This means for each `i`, the number of iterations is `(n - i)`.
   - The total number of iterations across all `i` is: $O(n^2)$
   - The operations inside the inner loop are constant time, so the overall complexity of this nested loop is **$O(n^2)$**.

2. Consider the following problem: As input you are given two sorted arrays of integers. Your objective is to design an algorithm that would merge the two arrays together to form a new sorted array that contains all the integers contained in the two arrays. For example, on input [1, 4, 5, 8, 17], [2, 4, 8, 11, 13, 21, 23, 25] the algorithm would output the following array: [1,2,4,4,5,8,8, 11, 13, 17, 21, 23, 25]

Answer:

A.

```
Algorithm Merge(arr1, arr2):
    Input: Two sorted arrays arr1 and arr2
    Output: A new sorted array containing all elements from arr1 and arr2

    1. Let mergedArray be a new empty array of size (length of arr1 + length of arr2)
    2. Initialize index1 = 0, index2 = 0, mergedIndex = 0
    3. While index1 < length of arr1 AND index2 < length of arr2:
        a. If arr1[index1] <= arr2[index2]:
            i. mergedArray[mergedIndex] = arr1[index1]
            ii. Increment index1
        b. Else:
            i. mergedArray[mergedIndex] = arr2[index2]
            ii. Increment index2
        c. Increment mergedIndex
    4. While index1 < length of arr1:
        a. mergedArray[mergedIndex] = arr1[index1]
        b. Increment index1, mergedIndex
    5. While index2 < length of arr2:
        a. mergedArray[mergedIndex] = arr2[index2]
        b. Increment index2, mergedIndex
    6. Return mergedArray
```

B. Asymptotic Running Time Analysis

- The algorithm traverses both arrays once in a single pass.
- Each element is compared and inserted into the new array in **O(1)** time.
- The worst-case scenario involves iterating through all elements in both arrays, which results in **O(n + m)** complexity, where:
    - n is the length of `arr1`
    - `m` is the length of `arr2`
- Since we only use a constant amount of extra space aside from the output array, the space complexity is **O(n + m)**.

Thus, the **time complexity is O(n + m)**, and the **space complexity is O(n + m)**.

C. Java Implementation

```
import java.util.Arrays;

public class MergeSortedArrays {

    public static int[] merge(int[] sortedArrayOne, int[] sortedArrayTwo) {
        int lengthOfFirstArray = sortedArrayOne.length;
        int lengthOfSecondArray = sortedArrayTwo.length;
        int[] mergedSortedArray = new int[lengthOfFirstArray + lengthOfSecondArray];

        int firstArrayIndex = 0, secondArrayIndex = 0, mergedArrayIndex = 0;


        while (firstArrayIndex < lengthOfFirstArray && secondArrayIndex <
lengthOfSecondArray) {
            if (sortedArrayOne[firstArrayIndex] <= sortedArrayTwo[secondArrayIndex]) {
                mergedSortedArray[mergedArrayIndex++] =
sortedArrayOne[firstArrayIndex++];
            } else {
                mergedSortedArray[mergedArrayIndex++] =
sortedArrayTwo[secondArrayIndex++];
            }
        }


        while (firstArrayIndex < lengthOfFirstArray) {
            mergedSortedArray[mergedArrayIndex++] = sortedArrayOne[firstArrayIndex++];
        }


        while (secondArrayIndex < lengthOfSecondArray) {
            mergedSortedArray[mergedArrayIndex++] = sortedArrayTwo[secondArrayIndex++];
        }

        return mergedSortedArray;
    }

    public static void main(String[] args) {
        int[] arrayOne = {1, 4, 5, 8, 17};
        int[] arrayTwo = {2, 4, 8, 11, 13, 21, 23, 25};

        int[] mergedArray = merge(arrayOne, arrayTwo);
        System.out.println("Merged Sorted Array: " + Arrays.toString(mergedArray));
    }
}
```

3. Answer:

Yogendra Pokhrel
617885

6> Answer:

Statement A: $1 + 4n^2$ is $O(n^2)$

By defination, $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0.$$

Let $f(n) = 1 + 4n^2$ and $g(n) = n^2$.

for sufficiently large n:

$$1 + 4n^2 \leq 5n^2.$$

Choosing $c = 5$ and $n_0 = 1$ we see that $1 + 4n^2$ is bounded above by a constant multiple of n^2.

Thus $1 + 4n^2$ is $O(n^2)$.

Statement B: $n^2 - 2n$ is not $O(n)$

if $n^2 - 2n$ were $O(n)$ there would exist constants c and n_0 such that

$$n^2 - 2n <= c \cdot n \text{ for all } n >= n_0.$$

Rearranging:

$$n^2 <= (c+2)n.$$

Dividing by n (for large n).

$$n <= c+2.$$

This is false for arbitarily larger n since $n \to \infty$ Contradicting the assumption. Thus $n^2 - 2n$ cannot be bounded by $O(n)$.

Yogendra Pokhrel
617585

Statement C: $\log(n)$ is $O(n)$

By definition, $f(n)$ is $O(g(n))$ if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

let $f(n) = \log(n)$ and $g(n) = n$:

$$\lim_{n \to \infty} \frac{\log(n)}{n}$$

Since $\log(n)$ grows much slower than n, this fraction approaches 0 as $n \to \infty$ Thus $\log(n)$ is $O(n)$

Statement D: n is not $O(n)$

Checking the definition:

$$\lim_{n \to \infty} \frac{n}{n} = 1$$

Since the limit is not 0, n is not $O(n)$

Thus n is not $o(n)$.

4. Answer:

```java
import java.util.*;

public class PowerSetGenerator {

    public static List<Set<Integer>> powerSet(List<Integer> inputSet) {
        List<Set<Integer>> powerSet = new ArrayList<>();
        powerSet.add(new HashSet<>());

        for (Integer element : inputSet) {
            List<Set<Integer>> newSubsets = new ArrayList<>();
            for (Set<Integer> subset : powerSet) {
                Set<Integer> newSubset = new HashSet<>(subset);
                newSubset.add(element);
                newSubsets.add(newSubset);
            }
            powerSet.addAll(newSubsets);
        }

        return powerSet;
    }

    public static void main(String[] args) {
        List<Integer> inputSet = Arrays.asList(1, 2, 3);
        List<Set<Integer>> result = powerSet(inputSet);

        System.out.println("Power Set:");
        for (Set<Integer> subset : result) {
            System.out.println(subset);
        }
    }
}
```