2️⃣

HW2

▼ **Question 1**

1. What is the worst-case runtime performance of the procedure below?

```
c = 0
i = n
while i > 1 do
    for j = 1 to i do
        c = c + 1
    end for
    i = floor(i/2)
end while
return c
```

Solution

The line $c = c + 1$ executes for $n + n/2 + n/4 + ...$ times. Let $T(n) = n + n/2 + n/4 + ...$ This is a geometric progression. Applying formula, we get $n(\frac{1-(1/2)^n}{1-1/2})$, solving which, we get $2n.(1 - (\frac{1}{2})^n)$ which is $O(n)$ for asymptotic values of n.

> 💡 Total worst case time complexity is $O(n)$

▼ **Question 2**

2. Arrange these functions under the O notation using only $=$ (equivalent) or $\subset$ (strict subset of):

   (a) $2^{\log n}$
   (b) 2^{3n}
   (c) $n^{n \log n}$
   (d) $\log n$
   (e) $n \log (n^2)$
   (f) n^{n^2}
   (g) $\log(\log(n^n))$

   E.g. for the function n, $n + 1$, n^2, the answer should be

$$O(n + 1) = O(n) \subset O(n^2).$$

General Time Complexity Trend -

$1 < log(n) < \sqrt{n} < n < nlog(n) < n^2 < n^3 < ... < 2^n < 3^n < ... < n^n$

1. Comparing (c) and (f), we get $n^{nlog(n)} \subset n^{n^2}$, for comparing exponents, since the bases are same, we compare the powers. We know from the texts that logarithms grow slower than polynomials
2. Comparing (a) and (b), we get $2^{log(n)} \subset 2^{3n}$, same base, comparing exponents, we know that logarithms grow slower than polynomials, hence the ordering. Further, we can simplify $2^{log(n)}$ as n
3. Then, we have $nlog(n^2)$ which simplifies to $2n \cdot log(n)$ which is essentially $O(n \cdot log(n))$
4. Comparing (d) and (g), we get $log(n) \subset log(log(n^n))$ as $log(log(n^n))$ simplifies to $log(n \cdot log(n))$

Combining 1, 2, 3 and 4, we get →

> 💡 $log(n) \subset log(log(n^n)) \subset 2^{log(n)} \subset nlog(n^2) \subset 2^{3n} \subset n^{nlog(n)} \subset n^{n^2}$

▼ **Question 3**

3. Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

   (a) $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
   (b) $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
   (c) $f_1(n)^2 = O(g_1(n)^2)$
   (d) $\log_2 f_1(n) = O(\log_2 g_1(n))$

Solution A = TRUE

From definition, $f(n) = O(g(n))$ implies iff $\exists c$ such that $f(n) = c.g(n)$. Applying the same to the above definitions,

- $f_1(n) = O(g_1(n))$ → $f_1(n) = c1.g_1(n)$
- $f_2(n) = O(g_2(n))$ → $f_2(n) = c1.g_2(n)$

Multiplying LHS of both sides, we get

→ $f_1(n).f_2(n) = c1.g_1(n) * c2.g_2(n)$

→ $f_1(n).f_2(n) = (c1.c2).(g_1(n).g_2(n))$

→ $f_1(n).f_2(n) = C.(g_1(n).g_2(n))$, where $C = c1.c2$

→ By definition, $f_1(n).f_2(n) = O(g_1(n).g_2(n))$

Hence Proved!

Solution B = TRUE

From definition, $f(n) = O(g(n))$ implies iff $\exists c$ such that $f(n) = c.g(n)$. Applying the same to the above definitions,

- $f_1(n) = O(g_1(n))$ → $f_1(n) = c1.g_1(n)$
- $f_2(n) = O(g_2(n))$ → $f_2(n) = c1.g_2(n)$

Adding LHS of both sides, we get

→ $f_1(n) + f_2(n) = c1.g_1(n) + c2.g_2(n)$

Since, we are considering the Big-Oh notation, we will consider the $max(g_1(n), g_2(n))$ during complexity analysis

→ $f_1(n) + f_2(n) = (c1 + c2).(max(g_1(n), g_2(n)))$

→ $f_1(n).f_2(n) = C.(max(g_1(n), g_2(n)))$, where $C = c1 + c2$

→ By definition, $f_1(n) + f_2(n) = O(max(g_1(n), g_2(n)))$

Hence Proved!

Solution C = TRUE

From definition, $f(n) = O(g(n))$ implies iff $\exists c$ such that $f(n) = c.g(n)$. Applying the same to the above definitions,

- $f_1(n) = O(g_1(n))$ → $f_1(n) = c1.g_1(n)$

Squaring both sides, we get

→ $f_1(n)^2 = c1^2.g_1(n)^2$

→ $f_1(n)^2 = C.g_1(n)^2$, where $C = c1^2$

→ Hence, by definition, $f_1(n)^2 = O(g_1(n)^2)$

Hence Proved!

Solution D = FALSE

From definition, $f(n) = O(g(n))$ implies iff $\exists c$ such that $f(n) = c.g(n)$. Applying the same to the above definitions,

- $f_1(n) = O(g_1(n))$ → $f_1(n) = c1.g_1(n)$

Taking log both sides, we get

→ $log_2 f_1(n) = log_2 c1 + log_2 g_1(n)$

→ For $log_2 f_1(n) = O(log_2 g_1(n))$ to be true, we need to find a d such that, $log_2 f_1(n) <= d.log_2 g_1(n)$

→ This means that, $log_2 c1 + log_2 g_1(n) <= d.log_2 g_1(n)$

→ Factorizing the above equation, taking $log_2 g_1(n)$ common, we get $\frac{log_2 c1}{log_2 g_1(n)} + 1 <= d$

→ Consider the following counter-example - $2(1 + \frac{1}{n}) \in O(1 + \frac{1}{n})$

→ Taking log on both sides, we get, $log_2 2 + log_2(1 + \frac{1}{n}) \notin O(log_2(1 + \frac{1}{n}))$

Hence Proved!

▼ **Question 4**

4. Given an undirected graph G with n nodes and m edges, design an $O(m+n)$ algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one.

Pseudo Code

```
For a given graph G = {V, E} represented by an adjacency list, we perform a DFS traversal on the graph to identify whether a cycle

Function: it returns a boolean. true, if a cycle is present, and false, if a cycle is not present
isCycleDetected(node, parent, pathSoFar) {
  visited[node] = true;
  pathSoFar += node;
  for (each edge that starts from this node, e) {
    if (visited[e.neighbor] == false) {
      if (dfs(e.neighbor, e.parent, pathSoFar + e.neighbor) == true) {
        print pathSoFar; // printing the cycle path as the cycle is present
        return true;
      }
    } else {
      if (child != parent) { // this condition checks whether a node is referring to its parent or not.
        print pathSoFar; // printing the cycle path as the cycle is present
      }
    }
  }
  return false; // cycle is not present
}
```

💡 This code does a BFS traversal of a graph, which is represented by an adjacency list and hence, the time complexity of the aforementioned algorithm is O(V + E)

▼ **Question 5**

6. Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$—that is, the sum $A[i] + A[i+1] + \cdots + A[j]$. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

Here's a simple algorithm to solve this problem.

```
For i = 1, 2, ..., n
  For j = i + 1, i + 2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

(c) Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem—after all, it just iterates through the relevant entries of the array B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n \to \infty} g(n)/f(n) = 0$.

a. Consider the following -

Complexity Analysis

1. The outer for-loop (i) runs for n times.
2. The inner for-loop (j) runs for at max n times.
3. The step "add up array entries A[i] through [j]" runs takes exactly O(j - i + 1) time which means that it takes O(n) time as its upper bound.
4. The step "store the result in B[i, j]" takes up constant time i.e. O(1)
5. Total time taken = $O(n^2) * [(j - i + 1) + constant(K)]$
6. Converting and simplifying into Big-Oh notation, we get $O(n^3)$

b. Answer -

Proof

- Consider cases when $i <= n/4$ and $j >= 3n/4$
- from (a) we know that addition step's amortized complexity is $j - i + 1$
- using the assumed values in the equation, we get, $j - i + 1 >= 3n/4 - n/4 + 1 > n/2$
- the addition step executes $n/2$ times atleast
- considering $i = n/4$ and $j = 3n/4$ for calculating total number of elements that satisfy $i <= n/4$ and $j >= 3n/4$, we get $(n/4)^2$ elements
- total operations performed by the algorithm - $n/2 * (n/4)^2$ = $n^3/32$
- considering lower bound of the algorithmic complexity = $\Omega(n^3)$

💡 Big-Omega notation = $\Omega(n^3)$

c. Answer -

Code

```
import java.util.Arrays;

public class Problem006 {

    static int[] a = {1,2,3,4,5,6}; // prefix sum for this array would be = {1, 3, 6, 10, 15, 21}
    static int n = 6;
    static int[][] b = new int[n][n];
    static int[] prefix_sum;

    public static void main(String[] args) {
        getPrefixSum(a);
        bruteForce();
        optimizedAlgo();
        printAns();
    }

    private static void printAns() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(b[i][j] + "\t");
            }
            System.out.println();
        }
    }

    private static void optimizedAlgo() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i >= j) {
                    b[i][j] = 0;
                } else {
                    if (i == 0) {
                        b[i][j] = prefix_sum[j];
                    } else {
                        b[i][j] = prefix_sum[j] - prefix_sum[i - 1];
                    }
                }
            }
        }
    }

    private static void bruteForce() {
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int sum = 0;
                for (int k = i; k <= j; k++) {
                    sum += a[k];
                }
                b[i][j] = sum;
            }
        }
    }

    private static void getPrefixSum(int[] a) {
        int[] ps = new int[a.length];
        ps[0] = a[0];

        for (int i = 1; i < a.length; i++) {
            ps[i] = ps[i - 1] + a[i];
        }
        prefix_sum = ps;
    }
}
```

```
OUTPUT:

PREFIX SUM ARRAY: [1, 3, 6, 10, 15, 21]
0 3 6 10  15  21
0 0 5 9   14  20
0 0 0 7   12  18
0 0 0 0   9   15
```

```
0 0 0 0   0   11
0 0 0 0   0   0
```

Optimized Algorithm

```
Let A be the input array and B be the output array of size n. Prefix_sum is the array calculated such that for any index k in
prefix_sum[k] = prefix_sum[k - 1] + a[k]


private static void optimizedAlgo() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i >= j) {
                    b[i][j] = 0;
                } else {
                    if (i == 0) {
                        b[i][j] = prefix_sum[j];
                    } else {
                        b[i][j] = prefix_sum[j] - prefix_sum[i - 1];
                    }
                }
            }
        }
    }
```

Complexity Analysis

The time complexity for each step can be stated as follows -

- creating prefix_sum array = $O(n)$, where n is the size of the array
- looping through the output array *B* gives $O(n^2)$
- populating the output array requires constant time operations $O(1)$
- amortized complexity of the code = $O(n + n^2)$
- hence, the worst time complexity (Upper Bound - Big-Oh) of this algorithm is $O(n^2)$

Answer

💡 Optimized Big-Oh time complexity = $O(n^2)$

▼ Question 6

6. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Assumption:

Let us assume that there exists an edge E = {a, b} that does not belong to T.

Proof by Contradiction

Now, from the question, we know that T has been obtained by finding the DFS of the graph G. So, at least one vertex of a given edge must be the ancestor of another. Now, BFS of T at node *u* means distance of node b from u and distance of node a from u can differ by at most 1.

But, as per our assumption, since a is the ancestor of b, then the distance between u to a and distance between u to b is at most more than 1. This means that a is in fact a direct parent of b which belongs to T. This means that {a, b} is in fact an edge in the graph G, which contradicts our previous assumption that it is not.

Hence, the graph is same as the tree i.e. G = T. Proved!