Advanced Algorithmic Problem Solving (R1UC601B)
Assignment Solutions for MTE

Kushqgra Agarwal 22SCSE1010032

1. Explain the concept of a prefix sum array and its applications.

A prefix sum array is an array where each element at index i stores the sum of elements from index 0 to i in the original array. It allows for efficient range sum queries in constant time after a linear time preprocessing step. Applications include:

- Range sum queries
- Subarray sum problems
- Optimization in dynamic programming
- Frequency count optimizations

2. Write a program to find the sum of elements in a given range [L, R] using a prefix sum array.
Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Build the prefix sum array.
2. For a range [L, R], compute sum as prefix[R] - prefix[L-1] if L > 0, else prefix[R].

Program: public class

PrefixSumRangeQuery {

public static int[] buildPrefixSum(int[]

arr) { int[] prefix = new int[arr.length];

prefix[0] = arr[0]; for (int i = 1; i <

arr.length; i++) { prefix[i] = prefix[i -

1] + arr[i];

}

return prefix;

}

```
  public static int rangeSum(int[] prefix, int L, int
R) {        if (L == 0) return prefix[R];        return
prefix[R] - prefix[L - 1];
  }

  public  static  void  main(String[]
args) {        int[] arr = {2, 4, 6, 8, 10};
int[] prefix = buildPrefixSum(arr);
    System.out.println(rangeSum(prefix, 1, 3));
  }
}
```

Time Complexity: O(n) for building prefix sum, O(1) per query
Space Complexity: O(n)

Example: For array [2, 4, 6, 8, 10], range [1, 3] -> 4 + 6 + 8 = 18

3. Solve the problem of finding the equilibrium index in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Calculate total sum of array.
2. Iterate through array maintaining left_sum.
3. At each index, if left_sum == total - left_sum - arr[i], return index.

Program:

```
public class EquilibriumIndex {   public static
int findEquilibriumIndex(int[] arr) {       int total
= 0;       for (int num : arr) total += num;       int
leftSum = 0;       for (int i = 0; i < arr.length; i++) {
```

```
if (leftSum == total - leftSum - arr[i]) return i;
leftSum += arr[i];
    }
    return -1;
  }

  public static void main(String[] args) {
int[] arr = {1, 3, 5, 2, 2};
    System.out.println(findEquilibriumIndex(arr));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Example: For array [1, 3, 5, 2, 2], index 2 is equilibrium index.

4. Check if an array can be split into two parts such that the sum of the prefix equals the sum of the suffix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Calculate total sum.
2. Traverse array, keep track of left sum.
3. If at any point, left sum * 2 == total, return True.

Program:

```
public class EqualPrefixSuffixSum {    public
static boolean canSplitEqualSum(int[] arr) {
    int total = 0;       for (int num : arr)
total += num;       int leftSum = 0;
for (int i = 0; i < arr.length - 1; i++) {
```

```
leftSum += arr[i];          if (leftSum * 2
== total) return true;
      }
      return false;
   }

   public static void main(String[] args) {
int[] arr = {1, 2, 3, 3};
      System.out.println(canSplitEqualSum(arr));
   }
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Example: [1, 2, 3, 3] -> split at index 2: sum([1,2,3]) == sum([3])

5. Find the maximum sum of any subarray of size K in a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Use sliding window of size K.
2. Compute initial window sum.
3. Slide window and update max sum.

Program: public class

```
MaxSumSubarrayK {
   public static int maxSum(int[] arr, int k) {
int windowSum = 0;       for (int i = 0; i < k; i++)
windowSum += arr[i];
      int maxSum = windowSum;
```

```
    for (int i = k; i < arr.length; i++) {
windowSum += arr[i] - arr[i - k];          maxSum =
Math.max(maxSum, windowSum);
    }
    return maxSum;
  }

  public static void main(String[] args) {
int[] arr = {1, 4, 2, 10, 2, 3, 1, 0, 20};
int k = 4;
    System.out.println(maxSum(arr, k));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Example: Maximum sum of size 4 is 24 for subarray [2, 10, 2, 3]

6. Find the length of the longest substring without repeating characters. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Use sliding window with hash set.
2. Expand window until repeat. 3. Shrink from left until unique.

Program:

```
import java.util.HashSet;

public class LongestUniqueSubstring {
public static int longestUniqueSubstr(String
```

```
s) {       HashSet<Character> set = new
HashSet<>();       int l = 0, maxLen = 0;       for
(int r = 0; r < s.length(); r++) {          while
(set.contains(s.charAt(r))) {
set.remove(s.charAt(l));
            l++;
        }
        set.add(s.charAt(r));          maxLen =
Math.max(maxLen, r - l + 1);
    }
    return maxLen;
  }

  public static void main(String[] args) {
    String s = "abcabcbb";
    System.out.println(longestUniqueSubstr(s));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(n)

Example: Input = "abcabcbb", Output = 3 ("abc")

7. Explain the sliding window technique and its use in string problems.

The sliding window technique involves creating a window which can either be fixed or dynamic in size and moving it over the input to solve problems efficiently. It helps in reducing the time complexity of problems involving arrays or strings. In string problems, it's used for:

- Finding the longest/shortest substring with certain properties

- Detecting anagrams
- Pattern matching
- Optimizing space/time over brute-force solutions

8. Find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Expand around each character as center.
2. Track the longest palindrome.

Program: public class

LongestPalindromeSubstring { public static

String longestPalindrome(String s) { if (s ==

null || s.length() < 1) return ""; int start = 0,

end = 0; for (int i = 0; i < s.length(); i++) {

int len1 = expand(s, i, i); int len2 =

expand(s, i, i + 1); int len =

Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private static int expand(String s, int left, int right) { while (left >=

0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

```
        right++;
      }
      return right - left - 1;
    }

    public static void main(String[] args) {
      String s = "babad";
      System.out.println(longestPalindrome(s));
    }
}
```

Time Complexity: O(n^2)
Space Complexity: O(1)

Example: Input = "babad", Output = "bab" or "aba" **9.**

Find the longest common prefix among a list of strings.

Algorithm:

1. Sort the array of strings.
2. Compare characters of the first and last string until they differ.
3. Return the common characters.

Java Program:

java

CopyEdit import java.util.Arrays; public class Q9 {

public static String longestCommonPrefix(String[]

strs) {

```
    if (strs.length == 0) return "";
    Arrays.sort(strs);
    String first = strs[0];
```

```
    String last = strs[strs.length - 1];
int i = 0;
    while (i < first.length() && i < last.length() && first.charAt(i) == last.charAt(i)) i++;
return first.substring(0, i);
  }
  public static void main(String[] args) {
    String[] strs = {"flower", "flow", "flight"};
    System.out.println(longestCommonPrefix(strs));
  }
}
```

Time Complexity: O(N log N + M)
Space Complexity: O(1)
Example: Input: ["flower", "flow", "flight"] → Output: "fl"

---

10. Generate all permutations of a given string.

Algorithm:

1. Fix each character at the start and recursively permute the rest.
2. Swap back after recursion (backtracking).

Java Program:

```
java
CopyEdit public class Q10 {    public static
void permute(String s, int l, int r) {       if (l ==
r) System.out.println(s);       else {          for
(int i = l; i <= r; i++) {             s = swap(s, l, i);
permute(s, l + 1, r);             s = swap(s, l, i);
        }
    }
```

```
    }
    public static String swap(String s, int i, int
j) {        char[] arr = s.toCharArray();        char
temp = arr[i];        arr[i] = arr[j];        arr[j] =
temp;        return new String(arr);
    }
    public    static    void    main(String[]
args)  {                String  str  =  "ABC";
permute(str, 0, str.length() - 1);
    }
}
```

Time Complexity: O(n!)
Space Complexity: O(n)
Example: Input: "ABC" → Output: All 6 permutations

---

11. Find two numbers in a sorted array that add up to a target.

Algorithm:

1. Use two pointers from left and right.
2. Move left if sum < target, right if sum > target.

Java Program:

```
java
CopyEdit public class Q11 {    public static int[]
twoSum(int[] arr, int target) {        int i = 0, j =
arr.length - 1;        while (i < j) {            int sum =
arr[i] + arr[j];            if (sum == target) return new
int[]{arr[i], arr[j]};            else if (sum < target) i++;
        else j--;
```

```
    }
    return new int[]{};
  }
  public static void main(String[]
args) {      int[] arr = {1, 2, 4, 7, 11,
15};      int target = 15;      int[] result
= twoSum(arr, target);
System.out.println(result[0] + " " +
result[1]);
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [1,2,4,7,11,15], Target: 15 → Output: 4 11

---

12. Rearrange numbers into the lexicographically next greater permutation.

Algorithm:

1. Find the longest non-increasing suffix.
2. Swap pivot with the smallest greater element.
3. Reverse the suffix.

Java Program:

```
java
CopyEdit import java.util.Arrays; public class
Q12 {    public static void
nextPermutation(int[] nums) {      int i =
nums.length - 2;      while (i >= 0 && nums[i]
>= nums[i + 1]) i--;
```

```
    if (i >= 0) {
      int j = nums.length - 1;
while (nums[j] <= nums[i]) j--;
int temp = nums[i];          nums[i] =
nums[j];          nums[j] = temp;
    }
    int l = i + 1, r = nums.length -
1;      while (l < r) {          int temp
= nums[l];          nums[l++] =
nums[r];          nums[r--] = temp;

    }
  }
  public static void main(String[]
args) {          int[] nums = {1, 2, 3};
nextPermutation(nums);
    System.out.println(Arrays.toString(nums));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [1,2,3] → Output: [1,3,2]

---

13. Merge two sorted linked lists into one sorted list.

Algorithm:

1. Use recursion or iteration to merge node by node based on value.

Java Program:

java

```
CopyEdit class
ListNode {
   int val;
   ListNode next;
   ListNode(int x) { val = x; }
}
public class Q13 {    public static ListNode
merge(ListNode l1, ListNode l2) {       if (l1 == null)
return l2;       if (l2 == null) return l1;       if (l1.val <
l2.val) {
         l1.next = merge(l1.next, l2);
return l1;       } else {
         l2.next = merge(l1, l2.next);
         return l2;
      }
   }
   public static void printList(ListNode head) {
while (head != null) {
         System.out.print(head.val + " ");
head = head.next;
      }
   }
   public static void main(String[] args) {
ListNode a = new ListNode(1);
      a.next = new ListNode(3);
ListNode b = new ListNode(2);
```

```
    b.next = new ListNode(4);
ListNode res = merge(a, b);
printList(res);
  }
}
```

Time Complexity: O(n + m)
Space Complexity: O(n + m) (recursion)
Example: 1→3 and 2→4 → Output: 1→2→3→4

---

14. Find the median of two sorted arrays using binary search.

Algorithm:

1. Binary search on the smaller array.
2. Ensure partitioning conditions for median.

Java Program:

```
java CopyEdit public class Q14 {    public static
double findMedian(int[] A, int[] B) {       if
(A.length > B.length) return findMedian(B, A);
int m = A.length, n = B.length;       int imin = 0,
imax = m, halfLen = (m + n + 1) / 2;       while
(imin <= imax) {          int i = (imin + imax) / 2;
int j = halfLen - i;          if (i < m && B[j - 1] > A[i])
imin = i + 1;          else if (i > 0 && A[i - 1] > B[j])
imax = i - 1;          else {             int maxLeft = 0;
if (i == 0) maxLeft = B[j - 1];             else if (j == 0)
maxLeft = A[i - 1];             else maxLeft =
Math.max(A[i - 1], B[j - 1]);             if ((m + n) % 2
== 1) return maxLeft;             int minRight = 0;
```

```
if (i == m) minRight = B[j];              else if (j == n)
minRight = A[i];              else minRight =
Math.min(A[i], B[j]);              return (maxLeft +
minRight) / 2.0;
        }
      }
      return 0.0;
   }
   public static void main(String[] args) {
      int[] A = {1, 3};
int[] B = {2};
      System.out.println(findMedian(A, B));
   }
}
```

Time Complexity: O(log(min(m,n)))
Space Complexity: O(1)
Example: [1,3] and [2] → Output: 2.0

---

15. Find the k-th smallest element in a sorted matrix.

Algorithm:

1. Use binary search from smallest to largest element.
2. Count how many elements ≤ mid.

Java Program: java

```
CopyEdit public class Q15 {    public static int
kthSmallest(int[][] matrix, int k) {       int n =
matrix.length;       int low = matrix[0][0], high =
matrix[n - 1][n - 1];       while (low < high) {
```

```
int mid = (low + high) / 2;          int count = 0, j = n
- 1;          for (int i = 0; i < n; i++) {              while (j
>= 0 && matrix[i][j] > mid) j--;              count += (j
+ 1);
        }
        if (count < k) low = mid + 1;
else high = mid;
     }
     return low;
  }
  public   static   void   main(String[]
args) {       int[][] matrix = {          {1, 5,
9},
        {10, 11, 13},
        {12, 13, 15}
     };
     System.out.println(kthSmallest(matrix, 8));
  }
}
```

Time Complexity: O(n log(max - min))
Space Complexity: O(1)
Example: k=8 in given matrix → Output: 13

16. Find the majority element in an array that appears more than n/2 times.

Algorithm:

1. Use Boyer-Moore Voting Algorithm.
2. Maintain a count and a candidate.

3. Traverse through the array, adjusting count and updating candidate when necessary.

Java Program:

```
java

CopyEdit public class Q16 {    public static

int majorityElement(int[] nums) {       int

count = 0, candidate = -1;       for (int num :

nums) {          if (count == 0) {

candidate = num;

        }

        count += (num == candidate) ? 1 : -1;

    }

    return candidate;

  }

  public static void main(String[] args) {

int[] nums = {3, 3, 4, 2, 4, 4, 2, 4, 4};

    System.out.println(majorityElement(nums));

  }

}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [3, 3, 4, 2, 4, 4, 2, 4, 4] → Output: 4

---

17. Calculate how much water can be trapped between the bars of a histogram.

Algorithm:

1. Use two pointers from left and right.
2. Track maximum heights from both sides and calculate trapped water.

Java Program:

```
java
CopyEdit public class Q17 {    public static int trap(int[] height) {
int left = 0, right = height.length - 1, leftMax = 0, rightMax = 0, result =
0;       while (left <= right) {           if (height[left] < height[right]) {
if (height[left] >= leftMax) leftMax = height[left];              else result +=
leftMax - height[left];
            left++;
         } else {
            if (height[right] >= rightMax) rightMax = height[right];
else result += rightMax - height[right];
            right--;
         }
      }
      return result;
   }
   public static void main(String[] args) {
int[] height = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
      System.out.println(trap(height));
   }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [0,1,0,2,1,0,1,3,2,1,2,1] → Output: 6

---

18. Find the maximum XOR of two numbers in an array.

Algorithm:

1. Use a trie to store binary representations of numbers.

2. Traverse each number, finding the number with the maximum XOR.

Java Program:

java

CopyEdit class

```
TrieNode {
   TrieNode[] children = new TrieNode[2];
}
public class Q18 {    public static int
findMaximumXOR(int[] nums) {       TrieNode
root = new TrieNode();       int maxXOR = 0;
for (int num : nums) {          TrieNode node =
root;          int currentXOR = 0;          for (int i =
31; i >= 0; i--) {             int bit = (num >> i) & 1;
if (node.children[1 - bit] != null) {
currentXOR |= (1 << i);                node =
node.children[1 - bit];
            } else {
               node = node.children[bit];
            }
         }
         maxXOR = Math.max(maxXOR, currentXOR);
         node = root;          for (int
i = 31; i >= 0; i--) {             int bit
= (num >> i) & 1;
            if (node.children[bit] == null) node.children[bit] = new TrieNode();             node =
node.children[bit];
         }
```

```
    }
    return maxXOR;
  }
  public static void main(String[] args) {
int[] nums = {3, 10, 5, 25, 2, 8};
    System.out.println(findMaximumXOR(nums));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example: Input: [3, 10, 5, 25, 2, 8] → Output: 28

---

19. How to find the maximum product subarray.

Algorithm:

1. Use dynamic programming to track the maximum and minimum products up to each point.
2. Update the result as you go through the array.

Java Program:

java

CopyEdit public class Q19 { public static int

maxProduct(int[] nums) { int maxProd = nums[0],

minProd = nums[0], result = nums[0];

for (int i = 1; i < nums.length; i++)

{ if (nums[i] < 0) { int

temp = maxProd; maxProd =

minProd; minProd = temp;

}

```
        maxProd = Math.max(nums[i], maxProd * nums[i]);
minProd = Math.min(nums[i], minProd * nums[i]);
result = Math.max(result, maxProd);

    }
    return result;
  }
  public static void main(String[] args) {
int[] nums = {2, 3, -2, 4};
    System.out.println(maxProduct(nums));
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [2, 3, -2, 4] → Output: 6

---

20. Count all numbers with unique digits for a given number of digits.

Algorithm:

1. For a given number of digits n, the first digit has 9 choices (1–9), and subsequent digits have 9, 8, ... choices.
2. Multiply choices and sum for all possible lengths.

Java Program:

```
java
CopyEdit public class Q20 {    public static int
countNumbersWithUniqueDigits(int n) {
    if (n == 0) return 1;       int
result = 10, product = 9;
for (int i = 2; i <= n; i++) {
```

```
product *= (11 - i);          result
+= product;
      }
      return result;
   }
   public static void main(String[] args) {
      System.out.println(countNumbersWithUniqueDigits(3));
   }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: 3 → Output: 739

---

21. How to count the number of 1s in the binary representation of numbers from 0 to n.

Algorithm:

1. For each number from 0 to n, count 1s in its binary representation.

Java Program:

```
java
CopyEdit public class Q21 {
public static int countBits(int n) {
int count = 0;       for (int i = 0; i <=
n; i++) {          count +=
Integer.bitCount(i);
      }
      return count;
   }
   public static void main(String[] args) {
```

```
    System.out.println(countBits(5));
  }
}
```

Time Complexity: O(n log n)
Space Complexity: O(1)
Example: Input: 5 → Output: 7 (binary: 0, 1, 1, 2, 1, 2)

---

22. How to check if a number is a power of two using bit manipulation.

Algorithm:

1. A number is a power of two if it has only one bit set, i.e., n & (n - 1) == 0.

Java Program:

```
java
CopyEdit public class Q22 {    public
static boolean isPowerOfTwo(int n) {
return n > 0 && (n & (n - 1)) == 0;
  }
  public static void main(String[] args) {
    System.out.println(isPowerOfTwo(16));
  }
}
```

Time Complexity: O(1)
Space Complexity: O(1)
Example: Input: 16 → Output: true

---

23. How to find the maximum XOR of two numbers in an array (same as Q18).

---

24. Explain the concept of bit manipulation and its advantages in algorithm design.

Explanation:

Bit manipulation refers to the process of directly manipulating bits in a number using bitwise operators. It allows for efficient computation, especially for problems involving numbers and sets. Key advantages include:

- **Faster execution**: Bit operations are computationally inexpensive.
- **Space optimization**: Storing large sets or numbers using fewer bits.
- **Efficient algorithms**: Used in dynamic programming, graph algorithms, and finding subsets.

---

25. Solve the problem of finding the next greater element for each element in an array.

Algorithm:

1. Use a stack to keep track of indices.
2. Traverse the array from right to left and find the next greater element.

Java Program:

java

CopyEdit

```
import java.util.Arrays; import java.util.Stack;

public class Q25 {    public static int[]

nextGreaterElement(int[] nums) {       int[] result

= new int[nums.length];

    Arrays.fill(result, -1);

    Stack<Integer> stack = new Stack<>();       for (int i = 0; i

< nums.length; i++) {          while (!stack.isEmpty() &&

nums[i] > nums[stack.peek()]) {

          result[stack.pop()] = nums[i];

       }

       stack.push(i);

    }
```

```
        return result;
    }
    public static void main(String[] args) {
int[] nums = {4, 5, 2, 10};
        System.out.println(Arrays.toString(nextGreaterElement(nums)));
    }
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example: Input: [4, 5, 2, 10] → Output: [5, 10, 10, -1]

---

26. Remove the n-th node from the end of a singly linked list.

Algorithm:

1. Use two pointers, one ahead by n steps.
2. Move both pointers until the first pointer reaches the end.

Java Program:

```
java
CopyEdit class
ListNode {    int
val;
    ListNode next;
    ListNode(int x) { val = x; }
}
public class Q26 {    public static ListNode
removeNthFromEnd(ListNode head, int n) {        ListNode
dummy = new ListNode(0);        dummy.next = head;
        ListNode fast = dummy, slow = dummy;
```

```
    for (int i = 1; i <= n + 1; i++) fast =
fast.next;      while (fast != null) {         fast =
fast.next;          slow = slow.next;
    }
    slow.next = slow.next.next;
return dummy.next;
  }
  public static void main(String[] args) {
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
head.next.next.next = new ListNode(4);
head.next.next.next.next = new ListNode(5);
ListNode res = removeNthFromEnd(head, 2);
    while (res != null) {
      System.out.print(res.val + " ");
res = res.next;
    }
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: Input: [1, 2, 3, 4, 5], n = 2 → Output: [1, 2, 3, 5]

---

27. Find the node where two singly linked lists intersect.

Algorithm:

1. Use two pointers, one for each list.

2. Move both pointers through the lists, switching to the other list when they reach the end. The node where they meet is the intersection.

Java Program:

```
java

CopyEdit class

ListNode {

  int val;

  ListNode next;

  ListNode(int x) { val = x; }

}

public class Q27 {    public static ListNode

getIntersectionNode(ListNode headA, ListNode headB) {       if (headA

== null || headB == null) return null;       ListNode ptrA = headA, ptrB =

headB;       while (ptrA != ptrB) {          ptrA = (ptrA == null) ? headB :

ptrA.next;          ptrB = (ptrB == null) ? headA : ptrB.next;

     }

     return ptrA;

  }

  public static void main(String[] args) {

ListNode common = new ListNode(8);

common.next = new ListNode(4);

common.next.next = new ListNode(5);

ListNode headA = new ListNode(4);

headA.next = common;       ListNode headB

= new ListNode(1);       headB.next =

common;

     ListNode intersection = getIntersectionNode(headA, headB);
```

```
    System.out.println(intersection.val);
  }
}
```

Time Complexity: O(n + m)
Space Complexity: O(1)
Example: Intersection at node 8

28. Implement two stacks in a single array.

Algorithm:

1. Divide the array into two parts.
2. Each stack grows from opposite ends of the array.

Java Program: java

CopyEdit public class

Q28 { private int[] arr;

private int top1, top2;

public Q28(int size) {

arr = new int[size];

top1 = -1; top2 =

size;

  }

  public void push1(int val) { if (top1 + 1 == top2)

throw new StackOverflowError(); arr[++top1] = val;

  }

  public void push2(int val) {

    if (top2 - 1 == top1) throw new StackOverflowError();

arr[--top2] = val;

```
  }
  public int pop1() {        if (top1 == -1) throw new
StackUnderflowError();       return arr[top1--];
  }
  public int pop2() {       if (top2 == arr.length) throw new
StackUnderflowError();       return arr[top2++];
  }
  public    static    void    main(String[]
args) {        Q28 stack = new Q28(10);
stack.push1(10);     stack.push2(20);
     System.out.println(stack.pop1());
System.out.println(stack.pop2());
  }
}
```

Time Complexity: O(1) for both push and pop operations
Space Complexity: O(1)
Example: Push 10 to stack1, 20 to stack2 → Output: 10, 20

29. Write a program to check if an integer is a palindrome without converting it to a string.

Algorithm:

1. Reverse the second half of the number and compare it with the first half.

Java Program:

```
java CopyEdit public class Q29 {    public static
boolean isPalindrome(int x) {       if (x < 0 || (x %
10 == 0 && x != 0)) return false;       int
reversedHalf = 0;       while (x > reversedHalf) {
reversedHalf = reversedHalf * 10 + x % 10;
```

```
        x /= 10;
      }
      return x == reversedHalf || x == reversedHalf / 10;
    }
    public static void main(String[] args) {
      System.out.println(isPalindrome(121));
      System.out.println(isPalindrome(-121));
    }
}
```

Time Complexity: O(log n)
Space Complexity: O(1)
Example: Input: 121 → Output: true

30. Explain the concept of linked lists and their applications in algorithm design.

Explanation:
A **linked list** is a linear data structure where each element (node) contains data and a reference (or link) to the next node in the sequence. **Applications in algorithm design:**

- **Efficient insertion/deletion**: Insertion or deletion can be done in O(1) time if the node is given.
- **Dynamic data structures**: Linked lists can grow or shrink in size dynamically without reallocating or resizing.
- **Graph and tree representations**: Used in adjacency lists, and can be used for sparse matrices in graphs

31. Use a deque to find the maximum in every sliding window of size K.

Algorithm:

1. Use a deque to store indices of array elements.
2. Ensure the deque holds indices of elements in decreasing order.
3. Slide the window from left to right, adding elements to the deque and removing elements out of the window.

Java Program:

java

CopyEdit import

java.util.*;

```
public class Q31 {    public static int[]
maxSlidingWindow(int[] nums, int k) {       int n =
nums.length;       int[] result = new int[n - k + 1];
      Deque<Integer> deque = new ArrayDeque<>();

      for (int i = 0; i < n; i++) {
        // Remove elements not in the window
if (!deque.isEmpty() && deque.peek() < i - k + 1) {
deque.poll();
        }
        // Remove smaller elements from the deque          while
(!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
           deque.pollLast();
        }
        deque.offer(i);
if (i >= k - 1) {
           result[i - k + 1] = nums[deque.peek()];
        }
      }
      return result;
   }
```

```
   public static void main(String[]
args) {       int[] nums = {1,3,-1,-
3,5,3,6,7};       int k = 3;
      System.out.println(Arrays.toString(maxSlidingWindow(nums, k)));
   }
}
```

Time Complexity: O(n) **Space Complexity:** O(k) **Example:**
Input: [1,3,-1,-3,5,3,6,7], k = 3
Output: [3, 3, 5, 5, 6, 7]

---

32. Find the largest rectangle that can be formed in a histogram.

Algorithm:

1. Use a stack to keep track of bar indices.
2. For each bar, calculate the area formed with the bars in the stack.

Java Program:

```
java
CopyEdit import
java.util.Stack;

public class Q32 {    public static int
largestRectangleArea(int[] heights) {
Stack<Integer> stack = new Stack<>();       int
maxArea = 0, i = 0;
```

```
    while (i < heights.length) {          if (stack.isEmpty() ||
heights[i] >= heights[stack.peek()]) {
stack.push(i++);
        } else {
          int h = heights[stack.pop()];             int w =
stack.isEmpty() ? i : i - stack.peek() - 1;
maxArea = Math.max(maxArea, h * w);
        }
      }

      while (!stack.isEmpty()) {
        int h = heights[stack.pop()];          int w =
stack.isEmpty() ? i : i - stack.peek() - 1;
        maxArea = Math.max(maxArea, h * w);
      }

      return maxArea;
   }

   public static void main(String[] args) {
int[] heights = {2, 1, 5, 6, 2, 3};
      System.out.println(largestRectangleArea(heights));
   }
}
```

Time Complexity: O(n) **Space Complexity:** O(n) **Example:**
Input: [2, 1, 5, 6, 2, 3]

Output: 10 (area of rectangle formed by 5 and 6)

---

33. Explain the sliding window technique and its applications in array problems.

Explanation:
The **sliding window technique** involves maintaining a subset of elements in a sequence, usually in a contiguous block. The window "slides" over the array, expanding or contracting as needed.

Applications:

1. **Maximum/Minimum in a sliding window**
2. **Longest substring without repeating characters**
3. **Subarray sum or product problems**
4. **String matching problems**

For example, finding the maximum sum of a subarray of size k can be solved by maintaining a window of size k and moving it across the array.

---

34. Solve the problem of finding the subarray sum equal to K using hashing.

Algorithm:

1. Use a hash map to store the cumulative sum and its frequency.
2. At each element, check if the difference between the cumulative sum and K is present in the map. **Java Program:**

java

CopyEdit import

java.util.HashMap;

public class Q34 { public static int

subarraySum(int[] nums, int k) {

HashMap<Integer, Integer> map = new

```
HashMap<>();       map.put(0, 1); // base case: sum 0
occurs once        int sum = 0, count = 0;

      for (int num : nums) {
sum += num;           if
(map.containsKey(sum - k)) {
count += map.get(sum - k);
         }
         map.put(sum, map.getOrDefault(sum, 0) + 1);
      }

      return count;
   }

   public   static   void   main(String[]
args) {       int[] nums = {1, 1, 1};       int
k = 2;
      System.out.println(subarraySum(nums, k));
   }
}
```

Time Complexity: O(n) **Space Complexity:** O(n) **Example:**
Input: [1,1,1], k = 2
Output: 2 (subarrays [1,1] and [1,1])

---

35. Find the k-most frequent elements in an array using a priority queue.

Algorithm:

1. Use a hash map to count the frequency of each element.
2. Use a priority queue (max-heap) to get the top k frequent elements.

Java Program:

java

CopyEdit import

java.util.*;

```
public class Q35 {    public static int[]
topKFrequent(int[] nums, int k) {
    HashMap<Integer, Integer> freqMap = new HashMap<>();
    for (int num : nums) {          freqMap.put(num,
freqMap.getOrDefault(num, 0) + 1);
    }

    PriorityQueue<Map.Entry<Integer, Integer>> maxHeap = new
PriorityQueue<>((a, b) -> b.getValue() - a.getValue());
maxHeap.addAll(freqMap.entrySet());

    int[] result = new int[k];
for (int i = 0; i < k; i++) {
       result[i] = maxHeap.poll().getKey();
    }

    return result;
  }
```

```
   public static void main(String[]
args) {       int[] nums = {1,1,1,2,2,3};
int k = 2;
      System.out.println(Arrays.toString(topKFrequent(nums, k)));
   }
}
```

Time Complexity: O(n log k)
Space Complexity: O(n) **Example:**
Input: [1,1,1,2,2,3], k = 2
Output: [1, 2]

---

36. Generate all subsets of a given array.

Algorithm:

1. Use bit manipulation or backtracking to generate all possible subsets of the array.

Java Program:

```
java
CopyEdit import
java.util.*;

public class Q36 {    public static List<List<Integer>>
subsets(int[] nums) {       List<List<Integer>> result =
new ArrayList<>();       result.add(new ArrayList<>()); //
Add the empty subset

      for (int num : nums) {
int size = result.size();          for
(int i = 0; i < size; i++) {
```

```
            List<Integer> newSubset = new
ArrayList<>(result.get(i));            newSubset.add(num);
result.add(newSubset);
        }
      }

      return result;
   }

   public static void main(String[] args) {
int[] nums = {1,2,3};
      System.out.println(subsets(nums));
   }
}
```

Time Complexity: O(2^n) **Space Complexity:** O(2^n) **Example:**
Input: [1,2,3]
Output: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

---

37. Find all unique combinations of numbers that sum to a target.