(1) The problem is a variation of the Fractional Knapsack problem.

Steps:

i. For each exam question, compute the ratio of marks to time (marks/time). This ratio reflects the efficiency of attempting that question.

ii. Sort all questions in descending order of their ratio. This ensures that the most efficient questions are attempted first.

iii. Traverse the sorted list and pick questions greedily:- If the remaining capacity (time) is enough to complete the question fully, take it all.- If not, take the fraction of the question that fits in the remaining time.

iv. Keep track of the marks accumulated. This greedy approach guarantees the maximum possible marks.

v. Finally, compute and display results for two scenarios:

Alone: Time = T

With friend: Time = 2T

(2)This is basically Interval Scheduling (Greedy Algorithm) with an extra rest time X.

Steps:

1. Sort the seminars by their finishing time . This ensures we always get the earliest-finishing seminar.
2. Always pick the first seminar.
3. For each subsequent seminar:
    - If the current seminar ends plus the buffer time x is less than or equal to the start of the next seminar, we can attend it.
    - Otherwise, skip it.
4. Keep counting and printing the selected seminar IDs.

(3)A warehouse has n boxes with `name`, `weight`, and `value`; k thieves with capacities `Wi` arrive, each taking divisible items greedily to maximize profit, and the program computes each thief’s profit and the remaining items.

Steps:

1.Compute value-to-weight ratio for each item:

2.Sort items in descending order of `ratio`.
3.For each thief:

- Pick items greedily:
  - If knapsack can fit the entire item → take it fully.
  - Otherwise → take a fraction of the item proportional to remaining capacity.
- Update remaining weight and value of the item.

4.Repeat for all thieves.

5.After all thieves, list the items that remain in the warehouse.

(4)a divide and conquer program that takes X and Y as input and calculates X^Y.

Steps:

1. **Base case:** If y=0, return 1. (Because any number raised to 0 is 1).
2. **Divide the problem:**Compute `temp = power(x, y/2)`.
3. **Combine results:**
   - If y is even → x^y=(x^y/2)^2=temp×temp
   - If y is odd → x^y=x×(x^y/2)^2=x×temp×temp

(5) Given an array of integers, I need to sort them using merge sort (a divide and conquer algorithm).

Steps:

1.Divide: Recursively split the array into halves until only one element remains.

2.Conquer (Merge): Compare elements from the left and right halves and merge them into a sorted array.

- Since we use `if (L[i] >= R[j])`, the array is merged in descending order.
- Continue merging until the whole array is sorted.

(6)Given an inversion is a pair (i,j) such that ,I need to sort them using merge sort (a divide and conquer algorithm).

Steps:

1.Divide the array into halves recursively.

2.Merge Step: While merging:

- If `L[i] <= R[j]`, no inversion → copy `L[i]`.
- If `L[i] > R[j]`, then all remaining elements in L from i → end form inversions with `R[j]`, because both halves are already sorted.Than,Count this into `(m - i)`.
  Accumulate inversions while merging, and return total.

(7)Given an array of integers (may contain negatives), find the contiguous subarray that has the maximum sum.

Steps:

1. **Base Case:**If the array has only one element, that element itself is the maximum subarray.
2. **Divide:**Split the array into two halves around a midpoint.
3. **Conquer:**
   Recursively compute:
   - `lsum`: the maximum subarray entirely in the left half.
   - `rsum`: the maximum subarray entirely in the right half.
   - `csum`: the maximum subarray that **crosses the midpoint**.
4. **Combine:**Compare `lsum.sum`, `rsum.sum`, and `csum.sum`.
   Return whichever one is largest.

(8)Write a recursive function to calculate the sum of digits of an integer.

Steps:

1.Base case:If n=0n = 0n=0, return `0`.

2.Recursive step:

- Extract the last digit: `n % 1`
- Add it to the result of the sum of digits of the remaining number: `sumDigit(n / 10)`

`3.Return the sum.`

(9)A program that checks whether an integer is a palindrome using recursion.

Steps:

1.Convert the integer to a **string** (so we can easily compare characters).

2.Use recursion to check characters from **both ends**:

- Base case: if `start >= end`, return `true` (whole string checked).
- Recursive case:
    - If `s[start] != s[end]` → return `false`.
    - Else, check the substring `(start+1, end-1)`.

(10)The Tower of Hanoi is a mathematical puzzle with three rods and n disks of different sizes.

Steps:

- Base Case:If n=1, move the single disk from the source to the target.
- Recursive Case: To move nnn disks from source → target:
    1. Move n−1 disks from source to auxiliary (using target).
    2. Move the n-th (largest) disk from source to target.
    3. Move the n−1 disks from auxiliary to target (using source).

(11)Two players, Sereja and Dima, take turns picking the larger of the leftmost or rightmost cards from a row of n numbered cards, with Sereja going first, and the goal is to compute their final scores.

Steps:

1. Initialize two pointers:
    - `l = 0` (left), `r = n-1` (right)
2. Use a boolean `istrue` to track whose turn it is:
    - `true` → Sereja's turn
    - `false` → Dima's turn

3. While `l <= r`:
    - Compare `arr[l]` and `arr[r]`
    - Pick the **larger**:
        - If left is larger → take `arr[l]` and increment `l`
        - Else → take `arr[r]` and decrement `r`
    - Add the value to the current player’s score
    - Switch the turn (`istrue = !istrue`)
4. Stop when all cards are taken.

(12)A new email service registers n usernames, responding `"ok"` if a name is new, or appending the smallest positive integer to make it unique if it already exists.

Steps:

1. Use an **unordered_map<string, int>** to store usernames and counts.
2. For each new username:
    - If it **does not exist** in the map → print `"ok"` and set count to 0.
    - Else → increment the count, append it to the name, and print (e.g., `name1`, `name2`).
3. This ensures **unique usernames** efficiently.