**Min Max of the Array using Divide and Conquer :-**

 **Define Structure: Pair**

* max: Integer to store the maximum value.
* min: Integer to store the minimum value.

 **Function** maxMinDivideConquer(arr, low, high):

* **Input**: Array arr, integers low, high.
* **Output**: A Pair containing max and min.

**Steps**:

1. **Base Case 1**:  
   If low == high,
   * Create result with max = arr[low] and min = arr[low].
   * Return result.
2. **Base Case 2**:  
   If high == low + 1:
   * Compare arr[low] and arr[high].
   * Assign result.min to the smaller value and result.max to the larger value.
   * Return result.
3. **Divide**:
   * Compute mid = (low + high) / 2.
   * Recursively call maxMinDivideConquer for left half (low to mid) → left.
   * Recursively call maxMinDivideConquer for right half (mid + 1 to high) → right.
4. **Conquer**:
   * result.max = max(left.max, right.max).
   * result.min = min(left.min, right.min).
   * Return result.

 **Main Function**:

* **Input**: Array arr[] = {6, 4, 26, 14, 33, 64, 46}.
* Compute size n = sizeof(arr) / sizeof(arr[0]).
* Call maxMinDivideConquer(arr, 0, n - 1) → result.
* Print:
  + "Maximum element is: result.max".
  + "Minimum element is: result.min".

**Closest Pair of Points :-**

**Point Class**

* Define a class Point:
  + x: Integer, x-coordinate of the point.
  + y: Integer, y-coordinate of the point.

**Helper Functions**

1. **compareX(a, b)**:
   * **Input**: Two points a and b.
   * **Output**: Difference between a.x and b.x.
2. **compareY(a, b)**:
   * **Input**: Two points a and b.
   * **Output**: Difference between a.y and b.y.
3. **dist(p1, p2)**:
   * **Input**: Two points p1 and p2.
   * **Output**: Euclidean distance between p1 and p2.
4. **min(x, y)**:
   * **Input**: Two floats x and y.
   * **Output**: The smaller of the two values.

**Core Functions**

1. **bruteForce(P, n, closest\_pair)**:
   * **Input**: Array P of points, size n, reference to closest\_pair.
   * **Output**: Minimum distance between any two points in P.
   * Initialize min = ∞.
   * For each pair of points (P[i], P[j]) in P:
     + Compute distance d = dist(P[i], P[j]).
     + If d < min:
       - Update min = d.
       - Update closest\_pair = (P[i], P[j]).
   * Return min.
2. **stripClosest(strip, size, d, closest\_pair)**:
   * **Input**: Array strip of points, size size, current minimum distance d, reference to closest\_pair.
   * **Output**: Minimum distance between points in the strip.
   * Sort strip by y-coordinate using compareY.
   * Initialize min = d.
   * For each point strip[i]:
     + Compare it with the next points strip[j] such that (strip[j].y - strip[i].y) < min:
       - Compute distance d = dist(strip[i], strip[j]).
       - If d < min:
         * Update min = d.
         * Update closest\_pair = (strip[i], strip[j]).
   * Return min.
3. **closestUtil(P, n, closest\_pair)**:
   * **Input**: Array P of points, size n, reference to closest\_pair.
   * **Output**: Minimum distance between points in P.
   * If n ≤ 3:
     + Return bruteForce(P, n, closest\_pair).
   * Divide P into two halves:
     + Compute midpoint mid = n / 2.
     + Compute dl = closestUtil(P[0..mid-1], mid, left\_pair).
     + Compute dr = closestUtil(P[mid..n-1], n - mid, right\_pair).
     + Let d = min(dl, dr).
     + Set closest\_pair to the closer of left\_pair and right\_pair.
   * Create strip containing points within distance d of the midpoint.
   * Compute strip\_dist = stripClosest(strip, size, d, closest\_pair).
   * If strip\_dist < d:
     + Update d = strip\_dist.
   * Return d.
4. **closest(P, n, closest\_pair)**:
   * **Input**: Array P of points, size n, reference to closest\_pair.
   * **Output**: Minimum distance between points in P.
   * Sort P by x-coordinate using compareX.
   * Return closestUtil(P, n, closest\_pair).

**Main Function**

1. **Input**: Array P[] of points.
2. Compute n = sizeof(P) / sizeof(P[0]).
3. Initialize closest\_pair.
4. Call closest(P, n, closest\_pair) → min\_distance.
5. Print:
   * "The smallest distance is min\_distance".
   * "The points are: (closest\_pair.first.x, closest\_pair.first.y) and (closest\_pair.second.x, closest\_pair.second.y)".

**K-th Larget Element :-**

**partition\_algo(nums, L, R)**

**Input**:

* nums: List of integers.
* L: Left boundary index of the list.
* R: Right boundary index of the list.

**Output**:

* Index j where pivot is placed after partitioning.

**Steps**:

1. Set P = nums[L] (pivot).
2. Initialize i = L + 1 and j = R.
3. While i ≤ j:
   * If nums[i] < P and nums[j] > P:
     + Swap nums[i] and nums[j].
     + Increment i and decrement j.
   * If nums[i] ≥ P:
     + Increment i.
   * If nums[j] ≤ P:
     + Decrement j.
4. Swap nums[L] with nums[j].
5. Return j.

**findKthLargest(nums, k)**

**Input**:

* nums: List of integers.
* k: Integer, the position (from largest) of the desired element.

**Output**:

* The k-th largest element in nums.

**Steps**:

1. Set L = 0, R = nums.size() - 1.
2. Initialize pivot\_idx = 0.
3. Repeat until the pivot index matches k - 1:
   * Set pivot\_idx = partition\_algo(nums, L, R).
   * If pivot\_idx == k - 1:
     + Break.
   * Else if pivot\_idx > k - 1:
     + Update R = pivot\_idx - 1.
   * Else:
     + Update L = pivot\_idx + 1.
4. Return nums[pivot\_idx].

**Main Function**

1. Define a Solution class instance.
2. Initialize nums = [3, 2, 1, 5, 6, 4] and k = 3.
3. Call solution.findKthLargest(nums, k) to get the k-th largest element.
4. Print the result: "Kth largest element is: <result>".

**Maximum Subarray Sum Using Divide and Conquer**

**maxCrossingSum(arr, low, mid, high)**

**Input**:

* arr: Array of integers.
* low: Starting index of the current subarray.
* mid: Middle index of the current subarray.
* high: Ending index of the current subarray.

**Output**:

* Maximum sum of the subarray crossing the middle point.

**Steps**:

1. Initialize left\_sum = -∞ and sum = 0.
2. Iterate from mid to low (inclusive):
   * sum += arr[i].
   * If sum > left\_sum, update left\_sum = sum.
3. Initialize right\_sum = -∞ and sum = 0.
4. Iterate from mid + 1 to high (inclusive):
   * sum += arr[i].
   * If sum > right\_sum, update right\_sum = sum.
5. Return left\_sum + right\_sum.

**maxSubArraySum(arr, low, high)**

**Input**:

* arr: Array of integers.
* low: Starting index of the current subarray.
* high: Ending index of the current subarray.

**Output**:

* Maximum sum of any subarray within the range [low, high].

**Steps**:

1. If low == high:
   * Return arr[low] (base case: only one element).
2. Calculate mid = (low + high) / 2.
3. Recursively find:
   * left\_sum = maxSubArraySum(arr, low, mid) (maximum sum in the left half).
   * right\_sum = maxSubArraySum(arr, mid + 1, high) (maximum sum in the right half).
   * cross\_sum = maxCrossingSum(arr, low, mid, high) (maximum sum crossing the middle).
4. Return max(left\_sum, right\_sum, cross\_sum).

**Main Function**

1. Define an array arr of integers.
2. Set n = arr.length.
3. Call maxSubArraySum(arr, 0, n - 1) to compute the result.
4. Print the result.

**Search Element in Infinite Sorted Array :-**

**Pseudo-code:**

1. Define a function `getElement(arr, index)`:

- If `index` is out of bounds (greater than or equal to the size of `arr`), return a special symbol (simulate '@' with INT\_MAX).

- Otherwise, return the element at `index` in `arr`.

2. Define a function `findPosition(arr, K)`:

- Initialize `low` to 0 and `high` to 1.

- Step 1: Perform Exponential Search to find bounds:

- While the value at `high` is not INT\_MAX and is less than `K`:

- Set `low` to `high`.

- Double the value of `high`.

- Step 2: Perform Binary Search within the identified range:

- While `low` is less than or equal to `high`:

- Calculate `mid` as `low + (high - low) / 2` to prevent overflow.

- Get the value at `mid` using `getElement(arr, mid)`.

- If the value at `mid` equals `K`, return `mid` (element found).

- If the value at `mid` is greater than `K` or equals INT\_MAX:

- Set `high` to `mid - 1` (search left half).

- Otherwise:

- Set `low` to `mid + 1` (search right half).

- If no match is found, return -1.

3. In the `main` function:

- Define a sample sorted array `arr` (simulate an infinite sorted array with a finite array).

- Prompt the user to enter the value `K` to search for.

- Call `findPosition(arr, K)` to search for the position of `K`.

- If a valid position is returned:

- Print the index where `K` was found.

- Otherwise:

- Print that `K` was not found in the array.

4. End of program.

**Maximum Subarray Sum Using Greedy :-**

- The sum of `max\_current` and the current element.

- Update `max\_global` to the maximum of:

- `max\_global`.

- `max\_current`.

- Return `max\_global`.

2. In the `main` function:

- Define an array `nums` with sample integers (e.g., {-2, 1, -3, 4, -1, 2, 1, -5, 4}).

- Call Maximum\_Subarray\_Sum(nums) and store the result in max\_subarray\_sum.

- Print "Maximum Subarray Sum Is" followed by the value of max\_subarray\_sum.

3. End of program.

**Maximum Subarray Sum Using Kedans Algorithm :-**

Pseudo-code:

1. Define the function `Kedans\_Algorithm(nums)`:

- Input: An array `nums` of integers.

- Initialize `max\_sum` to the smallest possible integer (simulate `INT\_MIN`).

- Initialize `current\_sum` to 0.

- For each element `num` in `nums`:

- Add `num` to `current\_sum`.

- Update `max\_sum` to the maximum of:

- `current\_sum`

- `max\_sum`.

- If `current\_sum` becomes less than 0:

- Reset `current\_sum` to 0.

- Return `max\_sum`.

2. In the `main` function:

- Define an array `nums` with sample integers (e.g., `{-2, 1, -3, 4, -1, 2, 1, -5, 4}`).

- Call `Kedans\_Algorithm(nums)` and store the result in `max\_subarray\_sum`.

- Print "Maximum Subarray Sum Is" followed by the value of `max\_subarray\_sum`.

3. End of program.

**Sudoku using Backtracking :-**

Pseudo-code:

1. Define constants:

- UNASSIGNED = 0 (represents empty cells)

- N = 9 (size of the Sudoku grid)

2. Function `FindUnassignedLocation(grid, row, col)`:

- Loop through all rows and columns of the grid:

- If an unassigned cell (value = UNASSIGNED) is found:

- Set `row` and `col` to the location.

- Return true.

- If no unassigned cells are found:

- Return false.

3. Function `UsedInRow(grid, row, num)`:

- Check if `num` is present in the specified row.

- Return true if found; otherwise, return false.

4. Function `UsedInCol(grid, col, num)`:

- Check if `num` is present in the specified column.

- Return true if found; otherwise, return false.

5. Function `UsedInBox(grid, boxStartRow, boxStartCol, num)`:

- Check if `num` is present in the 3x3 subgrid starting at `boxStartRow` and `boxStartCol`.

- Return true if found; otherwise, return false.

6. Function `isSafe(grid, row, col, num)`:

- Return true if:

- `num` is not in the current row.

- `num` is not in the current column.

- `num` is not in the current 3x3 subgrid.

- The cell at `row, col` is unassigned.

- Otherwise, return false.

7. Function `SolveSudoku(grid)`:

- Find an unassigned cell using `FindUnassignedLocation(grid, row, col)`.

- If no unassigned cells are found:

- Return true (solution found).

- For each number from 1 to 9:

- If `isSafe(grid, row, col, num)`:

- Assign `num` to the cell at `row, col`.

- Recursively call `SolveSudoku(grid)`:

- If the recursive call returns true:

- Return true (solution found).

- If not successful:

- Unassign the cell at `row, col` (backtrack).

- If no number can be placed in the current cell:

- Return false (no solution exists).

8. Function `printGrid(grid)`:

- Loop through each cell in the grid and print its value.

9. Main function:

- Initialize the Sudoku grid with some pre-filled values and unassigned cells (represented by 0).

- Call `SolveSudoku(grid)`:

- If it returns true, call `printGrid(grid)` to display the solution.

- Otherwise, print "No solution exists".

**Printing all solutions in N-Queen Problem :-**

Function nQueenUtil(j, n, board, result, rows, diag1, diag2):

If j > n:

# A solution is found, add the current board configuration to result

Add board to result

Return

For i from 1 to n:

If rows[i] is false, diag1[i + j] is false, and diag2[i - j + n] is false:

# Place the queen in the current position

Set rows[i] = true, diag1[i + j] = true, diag2[i - j + n] = true

Add i to board

# Recurse to the next column (j + 1)

Call nQueenUtil(j + 1, n, board, result, rows, diag1, diag2)

# Backtrack: Remove the queen and reset the flags

Remove last element from board

Set rows[i] = false, diag1[i + j] = false, diag2[i - j + n] = false

Function nQueen(n):

Create empty result (list of lists)

Create empty board (list)

# Initialize rows, diag1, and diag2 as empty and false

Initialize rows as a list of n+1 elements, all set to false

Initialize diag1 as a list of 2n elements, all set to false

Initialize diag2 as a list of 2n elements, all set to false

# Start solving the N-Queens problem from the first column (1)

Call nQueenUtil(1, n, board, result, rows, diag1, diag2)

Return result

Main:

Set n = 4 # Number of queens (can be modified)

# Call nQueen function to get all valid configurations

Set result = nQueen(n)

# Print the result

For each configuration in result:

Print configuration as a list of positions

**Diet Plan of Mid Sem Exam :-**

Define FoodChart as a list of tuples with (FoodName, Calories, FatContent):

FoodChart = [

("Milk", 65, 4),

("Butter", 740, 82),

("Cheese", 310, 25),

("Ice Cream", 170, 7),

("Roasted Peanuts", 570, 49),

("Chocolate Biscuits", 520, 28),

("Cream", 210, 21)

]

Function CalculateCalories(foodChart, maxFat):

# Sort the foods based on the calorie-to-fat ratio (Calories/Fat)

Sort foodChart by Calories/Fat in descending order

# Initialize total calories and total fat consumed to 0

totalCalories = 0

totalFat = 0

selectedFoods = []

# Iterate through the sorted food chart

For each food in foodChart:

FoodName = food[0]

Calories = food[1]

FatContent = food[2]

# Check if adding this food would exceed the fat limit

If totalFat + FatContent <= maxFat:

# Add the food to the selected foods list

selectedFoods.append(FoodName)

totalCalories += Calories

totalFat += FatContent

# Return the total calories and the selected foods

Return totalCalories, selectedFoods

Main:

# Define the maximum fat consumption

maxFat = 151

# Call the function to calculate the maximum calories within the fat limit

totalCalories, selectedFoods = CalculateCalories(FoodChart, maxFat)

# Print the result

Print "Total Calories: ", totalCalories

Print "Selected Foods: ", selectedFoods

**Rabin-Karp String Matching Algorithm :-**

Pseudo-code :-

Define constants PRIME = 101 and MOD = 1e9 + 7

Function calculateHash(str, len):

Initialize hash = 0 and power = 1 # power = PRIME^0

For i from 0 to len-1:

hash = (hash + str[i] \* power) % MOD # Update hash

power = (power \* PRIME) % MOD # Update power to PRIME^(i+1)

Return hash # Return calculated hash value

Function recalculateHash(text, oldIndex, newIndex, oldHash, patternLen, primePower):

oldHash = (oldHash - text[oldIndex] + MOD) % MOD # Remove old character from hash

oldHash = (oldHash / PRIME) % MOD # Divide by PRIME to shift the hash

oldHash = (oldHash + text[newIndex] \* primePower) % MOD # Add new character to hash

Return oldHash # Return recalculated hash value

Function Rabin\_Karp(text, pattern):

Initialize an empty list result

Initialize textLen = length of text

Initialize patternLen = length of pattern

If patternLen > textLen:

Return empty list result # No matches possible if pattern is longer than text

# Calculate hash of the pattern and the initial substring of the text

patternHash = calculateHash(pattern, patternLen)

textHash = calculateHash(text, patternLen)

# Calculate primePower = PRIME^(patternLen-1) for hash recalculation

primePower = 1

For i from 1 to patternLen-1:

primePower = (primePower \* PRIME) % MOD

# Slide over the text and compare hashes

For i from 0 to textLen - patternLen:

If textHash == patternHash AND text substring from i to i + patternLen equals pattern:

Add i to the result list # Pattern found at index i

If i < textLen - patternLen:

# Recalculate the hash for the next substring

textHash = recalculateHash(text, i, i + patternLen, textHash, patternLen, primePower)

Return result # Return list of all occurrence indices of the pattern in the text

Function main:

Initialize Reference\_string = "ATGC"

Initialize Input\_String as an empty string

Initialize pattern = "AGC"

# Generate a random Input\_String by picking random characters from Reference\_string

For i from 0 to 24:

Append a random character from Reference\_string to Input\_String

Print "Random String: ", Input\_String

Print "Pattern: ", pattern

# Call Rabin\_Karp function to find all occurrences of the pattern

occurrences = Rabin\_Karp(Input\_String, pattern)

If occurrences is empty:

Print "No occurrences found for the pattern."

Else:

Print "Number of occurrences: ", length of occurrences

Print "Occurrences Founded At Indices: ", join occurrences into a space-separated string

**Coin Denomination :-**

Pseudo-code :-

Define constants INT\_MAX = a large number (infinity)

Function main:

// Step 1: Input the number of denominations

PRINT "Enter the Number of Denominations: "

INPUT denomination

// Step 2: Declare an empty list to store coin values and quantities

Initialize storage as an empty list

// Step 3: Input the coin values and quantities

FOR i from 0 to denomination - 1 DO:

PRINT "Enter The Details for (i + 1) Coins: "

// Input coin value and quantity

PRINT "Enter the Value of Coin: "

INPUT val

PRINT "Enter the Quantity of Coins: "

INPUT quantity

// Store the coin details (value and quantity) in the storage list

ADD (val, quantity) to storage

END FOR

// Step 4: Sort the storage list by coin values in descending order

SORT storage by the first element (coin value) in descending order

// Step 5: Input the amount for which we need to find the minimum number of coins

PRINT "Enter the Amount: "

INPUT amount

// Step 6: Initialize dynamic programming table dp

Initialize dp as an array of size (amount + 1), with all values set to INT\_MAX

Set dp[0] = 0 // 0 coins are required to make the amount 0

// Step 7: Update the dp table using dynamic programming

FOR i from 0 to denomination - 1 DO:

FOR j from storage[i].first to amount DO:

IF dp[j - storage[i].first] != INT\_MAX THEN:

dp[j] = MIN(dp[j], dp[j - storage[i].first] + 1)

END IF

END FOR

END FOR

// Step 8: Output the result

IF dp[amount] == INT\_MAX THEN:

PRINT "Minimum Number of Coins Required is Not Possible"

ELSE:

PRINT "Minimum Number of Coins Required is ", dp[amount]

END Function

**Segment Partition Q-2(B):-**

**Pseudo Code**

Function maxSegments(n, x, y, z):

// Initialize a dp array of size n+1 with all values set to -1

Initialize dp as an array of size (n + 1) with all values set to -1

Set dp[0] = 0 // Base case: no segments needed for length 0

// Loop through all lengths from 1 to n

FOR i from 1 to n DO:

// Check if a segment of length x can be cut

IF i >= x AND dp[i - x] != -1 THEN:

dp[i] = MAX(dp[i], dp[i - x] + 1)

END IF

// Check if a segment of length y can be cut

IF i >= y AND dp[i - y] != -1 THEN:

dp[i] = MAX(dp[i], dp[i - y] + 1)

END IF

// Check if a segment of length z can be cut

IF i >= z AND dp[i - z] != -1 THEN:

dp[i] = MAX(dp[i], dp[i - z] + 1)

END IF

END FOR

// If dp[n] is -1, it means it's not possible to cut the segment into valid pieces, return 0

IF dp[n] != -1 THEN:

RETURN dp[n]

ELSE:

RETURN 0

END IF

Function main:

PRINT "Enter the length of the line segment (n): "

INPUT n

PRINT "Enter the first cut length (x): "

INPUT x

PRINT "Enter the second cut length (y): "

INPUT y

PRINT "Enter the third cut length (z): "

INPUT z

// Call the maxSegments function to calculate the result

result = maxSegments(n, x, y, z)

PRINT "The maximum number of segments that can be cut from length ", n

PRINT "using lengths ", x, ", ", y, " and ", z, " is: ", result

**Critical Connections :-**

**Pseudo Code**

Function findCriticalConnections(graph, node):

Initialize min\_degree as INT\_MAX

Initialize src, dest as -1

Initialize criticalConnections as an empty list

// Loop through all nodes in the graph

FOR i from 0 to node - 1 DO:

Initialize curr\_degree as 0

// Loop through each possible edge from node i

FOR j from 0 to node - 1 DO:

IF graph[i][j] is not 9999 and graph[i][j] is not 0 THEN:

Increment curr\_degree by 1

// If the node i has exactly one connection, consider it as a critical connection

IF curr\_degree is 1 THEN:

// Find the neighboring node of i and store the critical connection

FOR j from 0 to node - 1 DO:

IF graph[i][j] is not 9999 and graph[i][j] is not 0 THEN:

Set src = i

Set dest = j

Add the pair (src, dest) to the criticalConnections list

END IF

END FOR

END IF

END FOR

// Print all critical connections found

FOR each connection in criticalConnections DO:

PRINT "The Critical Connection is from connection.first to connection.second"

END FOR

Function main:

PRINT "Enter The Number of Nodes :- "

INPUT node

// Initialize the adjacency matrix for the graph with default value 9999

Initialize graph as a 2D array of size node x node filled with 9999

PRINT "Enter The Adjacency Matrix :- "

// Take input for the adjacency matrix

FOR i from 0 to node - 1 DO:

FOR j from 0 to node - 1 DO:

PRINT "Enter The Edge from i to j (9999 for no edge) :- "

INPUT graph[i][j]

END FOR

END FOR

// Call the function to find and print critical connections

CALL findCriticalConnections(graph, node)

**Dijkstra Algorithm :**

**Algorithm**:

* Create a set **sptSet**(shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
* Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign the distance value as 0 for the source vertex so that it is picked first.
* While **sptSet**doesn’t include all vertices
  + Pick a vertex **u**that is not there in **sptSet**and has a minimum distance value.
  + Include u to **sptSet**.
  + Then update the distance value of all adjacent vertices of **u**.
    - To update the distance values, iterate through all adjacent vertices.
    - For every adjacent vertex **v,**if the sum of the distance value of **u**(from source) and weight of edge **u-v**, is less than the distance value of **v**, then update the distance value of **v**.

**Pseudo Code :**

Define constants:

V = 9 // Number of vertices in the graph

Function minDistance(dist[], sptSet[]):

Initialize min to a very large number (INT\_MAX)

For each vertex v from 0 to V-1:

If v is not in the shortest path tree (sptSet[v] is false)

and dist[v] is less than or equal to min:

Set min to dist[v] and min\_index to v

Return min\_index (the vertex with the smallest distance)

Function printSolution(dist[]):

Print "Vertex \t Distance from Source"

For each vertex i from 0 to V-1:

Print i and dist[i]

Function dijkstra(graph[V][V], src):

Initialize dist[] to hold the shortest distances, set all to INT\_MAX

Initialize sptSet[] to false (vertex not yet included in the shortest path tree)

Set dist[src] to 0 (distance of the source from itself is 0)

For count = 0 to V-2:

Find the vertex u with the minimum distance from the source (using minDistance function)

Mark vertex u as included in the shortest path tree (set sptSet[u] = true)

For each vertex v from 0 to V-1:

If vertex v is not in the shortest path tree (sptSet[v] is false)

and there is an edge between u and v (graph[u][v] > 0)

and the path through u provides a shorter distance to v:

Update dist[v] to dist[u] + graph[u][v]

Call printSolution(dist) to print the shortest distances from the source

Main:

Define the graph as a 2D array representing the adjacency matrix

Call dijkstra function with the graph and source vertex 0