**Question 1:**

**Discuss recursion in brief and write down its generic Algorithm.**

Recursion is a programming concept where a function calls itself to solve a smaller instance of a problem. It's a powerful and elegant technique used in various programming languages to break down complex problems into simpler, more manageable subproblems.

**Here are the key characteristics of recursion:**

1. **Base Case:**

Every recursive function must have a base case that defines the simplest possible problem or input that doesn't need further recursion. It prevents the function from calling itself indefinitely.

1. **Progress towards Base Case:**

Recursive calls must make progress towards the base case, meaning that each recursive call should reduce the problem size or move closer to the base case.

1. **Self-Calling:**

The function calls itself a smaller or simpler version of the problem.

Recursion is often used when a problem can be naturally divided into smaller, similar subproblems. Common examples include traversing hierarchical data structures like trees and graphs, searching, sorting, and problems involving repeated subdivisions.

**Here's a generic algorithm for a recursive function:**

|  |
| --- |
| **ALGORITHM** |
| **function recursiveFunction(parameters):**  **// Base case**  **if base\_case\_condition(parameters):**  **return base\_case\_result**  **// Recursive case**  **else:**  **// Break the problem into smaller subproblems**  **subproblems = break\_into\_subproblems(parameters)**  **// Recursive calls**  **result = recursiveFunction(subproblems)**  **// Combine results if necessary**  **final\_result = combine\_results(result, additional\_parameters)**  **return final\_result** |

**Let's break down the components:**

* **base\_case\_condition:**

This is a condition that checks if the current input has reached a state where further recursion is unnecessary, and the function can return a result directly.

* **base\_case\_result:**

The result is returned when the base case is reached.

* **break\_into\_subproblems:**

This function breaks down the current problem into smaller subproblems, creating new sets of parameters for recursive calls.

* **recursiveFunction:**

The recursive function calls itself with the new set of parameters.

* **combine\_results:**

If needed, this step combines the results of recursive calls, possibly along with additional parameters.

It's important to design recursive functions carefully to ensure termination and avoid infinite loops. Understanding the problem structure and identifying base cases are crucial for successful recursive algorithm implementation.

**Question 2:**

**C++ Code for Sorting Algorithms:**

**Insertion Sort:**

#include <iostream>

Using namespace std;

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; ++i) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

} }

int main() {

int arr[] = {5, 2, 3, 1, 4};

int n = sizeof(arr) / sizeof(arr[0]);

insertionSort(arr, n);

cout << "Insertion Sort Result: ";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0; }

**Selection Sort:**

#include <iostream>

Using namespace std;

void selectionSort(int arr[], int n) {

for (int i = 0; i < n - 1; ++i) {

int minIndex = i;

for (int j = i + 1; j < n; ++j) {

if (arr[j] < arr[minIndex])

minIndex = j;

}

swap(arr[i], arr[minIndex]);

} }

int main() {

int arr[] = {5, 2, 3, 1, 4};

int n = sizeof(arr) / sizeof(arr[0]);

insertionSort(arr, n);

cout << "Insertion Sort Result: ";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0; }

**Merge Sort:**

#include <iostream>

Using namespace std;

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0;

int j = 0;

int k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++; }

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

} }

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

} }

int main() {

int arr[] = {5, 2, 3, 1, 4};

int n = sizeof(arr) / sizeof(arr[0]);

mergeSort(arr, 0, n - 1);

cout << "Merge Sort Result: ";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0;

}

**Quick Sort:**

#include <iostream>

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(arr[i], arr[j]);

} }

swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

} }

int main() {

int arr[] = {5, 2, 3, 1, 4};

int n = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, n - 1);

cout << "Quick Sort Result: ";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0;

}

**Dry Run:**

Let's consider the array {5, 2, 3, 1, 4} and simulate each algorithm. The dry run involves step-by-step execution of the algorithm on this input to observe how the array changes at each stage. Due to space limitations, I'll provide the final sorted array for each algorithm:

**Insertion Sort Result:**

After each iteration, the array becomes more sorted.

**Result:** {1, 2, 3, 4, 5}

**Selection Sort Result:**

Selects the minimum element in each iteration and swap it with the element in the current position.

**Result:** {1, 2, 3, 4, 5}

**Merge Sort Result:**

Splits the array into halves, sorts each half, and then merges them.

**Result:** {1, 2, 3, 4, 5}

**Quick Sort Result:**

Chooses a pivot, partitions the array, and recursively applies the same process to each partition.

**Result:** {1, 2, 3, 4, 5}