Linear Search

```
#include <iostream>
using namespace std;

int main() {
    // Input array
    int arr[] = {3, 5, 7, 9, 11};
    int target = 7; // Element to search
    int n = sizeof(arr) / sizeof(arr[0]); // Calculate array size

    // Iterate through array to find the target
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            cout << "Found at index " << i << endl; // Element found
            break; // Exit loop once found
        }
    }

    return 0;
}

/*
Time Complexity: O(n) - we may have to check each element
Space Complexity: O(1) - no extra space used
*/
```

Binary Search

```
#include <iostream>
using namespace std;

int main() {
    // Array must be sorted for Binary Search
    int arr[] = {2, 4, 6, 8, 10};
    int target = 6;
    int low = 0, high = 4; // Start and end indexes

    // Continue until search space is valid
    while (low <= high) {
        int mid = (low + high) / 2; // Find mid index
        if (arr[mid] == target) {
            cout << "Found at index " << mid << endl;
            break;
        } else if (arr[mid] < target) {
            low = mid + 1; // Search in right half
        } else {
            high = mid - 1; // Search in left half
        }
    }

    return 0;
}
```

```
/*
Time Complexity: O(log n) - halving the search space each step
Space Complexity: O(1)
*/
```

Bubble Sort

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {5, 3, 8, 4};
    int n = sizeof(arr) / sizeof(arr[0]);   // total elements

    /* ------------- Outer loop -------------
       After each full pass, the largest
       remaining element “bubbles” to the end */
    for (int i = 0; i < n; ++i) {

        /* --------- Inner loop ---------
           Compare pairs (j, j+1) that
           are still unsorted (≤ n-i-2)     */
        for (int j = 0; j < n - i - 1; ++j) {

            // If current element is bigger than next, swap them
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }

    // Print sorted array
    for (int num : arr) cout << num << " ";
    return 0;
}

/*
(two nested loops over n)  Time Complexity: O(n²)
(in-place swapping)Space Complexity: O(1)
*/
```

Selection Sort

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    /* Outer loop: step through each index i,      *
     * assuming that position i should hold the    *
     * smallest remaining element in the suffix.   */
    for (int i = 0; i < n; ++i) {
        int minIdx = i;  // candidate index for minimum

        // Scan the unsorted suffix to find real minimum
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }

        // Put the found minimum into correct spot i
        swap(arr[i], arr[minIdx]);
    }

    for (int num : arr) cout << num << " ";
    return 0;
}

/*
(two nested loops)   Time Complexity: O(n²)
(in-place)   Space Complexity: O(1)
*/
```

Insertion Sort

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    /* Treat arr[0..i-1] as a sorted sub-array;   *
     * insert arr[i] into its correct position.   */
    for (int i = 1; i < n; ++i) {
        int key = arr[i];     // element to be inserted
        int j  = i - 1;       // last index of sorted part

        // Shift elements right until the spot for key is found
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;     // place key
    }

    for (int num : arr) cout << num << " ";
    return 0;
}
```

```
/*
 Time Complexity: O(n²) worst, O(n) best (already sorted)
 Space Complexity: O(1)
*/
```

Merge Sort

```
#include <iostream>
#include <vector>
using namespace std;

/* ---------- Helper: merge two sorted halves ---------- */
void merge(vector<int>& arr, int l, int m, int r) {
    /* Create temp arrays:
       Left  = arr[l .. m]
       Right = arr[m+1 .. r]                                */
    int n1 = m - l + 1, n2 = r - m;
    vector<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];

    // Merge back into arr
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    }
    // Copy leftovers (one of these loops will execute 0 times)
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

/* -------------- Recursive merge-sort -------------- */
void mergeSort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;   // safe midpoint
        mergeSort(arr, l, m);      // sort left half
        mergeSort(arr, m + 1, r);  // sort right half
        merge(arr, l, m, r);       // merge halves
    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7};

    mergeSort(arr, 0, arr.size() - 1);

    for (int x : arr) cout << x << " ";
    return 0;
}

/*
 Time Complexity: O(n log n)  (divide-and-conquer)
 Space Complexity: O(n) extra (temporary arrays or recursion stack)
*/
```

Quick Sort]

```
#include <iostream>
#include <vector>
using namespace std;

/* Lomuto partition: chooses last element as pivot and
   places it in correct spot, so left ≤ pivot < right   */
int partition(vector<int>& a, int low, int high) {
    int pivot = a[high];
    int i = low - 1;   // tracks “last index of smaller element”

    for (int j = low; j < high; ++j) {
        if (a[j] <= pivot) {  // belongs on the left
            ++i;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[high]); // place pivot
    return i + 1;            // pivot index
}

void quickSort(vector<int>& a, int low, int high) {
    if (low < high) {
        int pi = partition(a, low, high);
        quickSort(a, low, pi - 1);   // sort left subarray
        quickSort(a, pi + 1, high);  // sort right subarray
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr, 0, arr.size() - 1);

    for (int x : arr) cout << x << " ";
    return 0;
}

/*
 Time Complexity: average O(n log n), worst O(n²) (when pivot poorest)
 Space Complexity: O(log n) expected (recursion), worst O(n)
*/
```

Two-Pointers Technique

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {1, 2, 3, 4, 6};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 6;

    int left = 0, right = n - 1;   // two ends of the sorted array

    // Move inward until pointers cross
    while (left < right) {
        int sum = arr[left] + arr[right];

        if (sum == target) {          // pair found
            cout << arr[left] << " " << arr[right] << endl;
            break;
        } else if (sum < target) {   // need bigger sum → move left
            ++left;
        } else {                      // need smaller sum → move right
            --right;
        }
    }
    return 0;
}

/*
 Time Complexity: O(n)
 Space Complexity: O(1)
*/
```

Sliding Window

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;                             // window size

    // Compute sum of first window [0..k-1]
    int windowSum = 0;
    for (int i = 0; i < k; ++i) windowSum += arr[i];

    int maxSum = windowSum;                // initialize answer

    // Slide window: add next element, subtract exiting element
    for (int i = k; i < n; ++i) {
        windowSum += arr[i] - arr[i - k];
        maxSum = max(maxSum, windowSum);  // track best
    }

    cout << maxSum << endl;
    return 0;
}

/*
 Time Complexity: O(n)
```

```
 Space Complexity: O(1)
*/
```

Kadane's Algorithm

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    int maxEndingHere = arr[0];   // best subarray ending at index i
    int maxSoFar      = arr[0];   // best subarray seen overall

    for (int i = 1; i < n; ++i) {
        // Extend previous subarray or start new at i
        maxEndingHere = max(arr[i], maxEndingHere + arr[i]);
        maxSoFar      = max(maxSoFar, maxEndingHere);
    }

    cout << maxSoFar << endl;   // maximum subarray sum
    return 0;
}

/*
 Time Complexity: O(n)
 Space Complexity: O(1)
*/
```

Prefix Sum

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    int prefix[n + 1];      // prefix[0] = 0, prefix[i] = sum up to i-1
    prefix[0] = 0;

    // Build prefix array
    for (int i = 0; i < n; ++i) {
        prefix[i + 1] = prefix[i] + arr[i];
    }

    // Example: sum of subarray [1, 3] (2 + 3 + 4) = prefix[4] - prefix[1]
    cout << prefix[4] - prefix[1] << endl;   // prints 9
    return 0;
}
```

```
/*
 Time Complexity: O(n) to build, O(1) per range query
 Space Complexity: O(n) for prefix array
*/
```

Flood Fill (DFS)

```
#include <iostream>
using namespace std;

const int ROW = 3, COL = 3;

/* Recursive DFS: recolor connected components that match origColor */
void floodFill(int grid[ROW][COL], int r, int c,
               int newColor, int origColor) {

    // Out-of-bounds OR cell doesn’t match original color → stop
    if (r < 0 || c < 0 || r >= ROW || c >= COL || grid[r][c] != origColor)
        return;

    grid[r][c] = newColor;          // paint current cell

    // Recurse 4-directionally (up, down, left, right)
    floodFill(grid, r + 1, c, newColor, origColor);
    floodFill(grid, r - 1, c, newColor, origColor);
    floodFill(grid, r, c + 1, newColor, origColor);
    floodFill(grid, r, c - 1, newColor, origColor);
}

int main() {
    int grid[ROW][COL] = { {1, 1, 1},
                           {1, 1, 0},
                           {1, 0, 1} };

    int startR = 1, startC = 1;
    floodFill(grid, startR, startC, 2, grid[startR][startC]);

    // Print result
    for (int i = 0; i < ROW; ++i) {
        for (int j = 0; j < COL; ++j) cout << grid[i][j] << " ";
        cout << endl;
    }
    return 0;
}

/*
 Time Complexity: O(n × m) — visits every cell at most once
 Space Complexity: O(n × m) worst (recursion stack in deepest cases)
*/
```