perplexity

also give its pseudo code

Here are the pseudocodes and C++ implementations for each of the algorithms:

1. Insertion Sort

Algorithm Steps:

1. Start from the second element (key).
2. Compare key with previous elements.
3. Shift elements > key to the right.
4. Insert key into correct position.

Pseudocode:

```
INSERTION-SORT(A):
    for j = 2 to A.length:
        key = A[j]
        i = j - 1
        while i &gt; 0 and A[i] &gt; key:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

C++ Implementation:

```
#include &lt;iostream&gt;
using namespace std;

void insertionSort(int arr[], int size) {
    for (int i = 1; i &lt; size; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j &gt;= 0 &amp;&amp; arr[j] &gt; key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i &lt; size; i++) {
        cout &lt;&lt; arr[i] &lt;&lt; " ";
    }
```

```
    cout &lt;&lt; endl;
}

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

    cout &lt;&lt; "Original array: ";
    printArray(arr, size);

    insertionSort(arr, size);

    cout &lt;&lt; "Sorted array: ";
    printArray(arr, size);

    return 0;
}
```

2. Quick Sort

Algorithm Steps:

1. Choose pivot (e.g., last element).
2. Partition array into ≤ pivot and ≥ pivot.
3. Recursively sort subarrays.

Pseudocode:

```
QUICK-SORT(A, low, high):
    if low &lt; high:
        pivotIndex = PARTITION(A, low, high)
        QUICK-SORT(A, low, pivotIndex - 1)
        QUICK-SORT(A, pivotIndex + 1, high)

PARTITION(A, low, high):
    pivot = A[high]
    i = low - 1
    for j = low to high - 1:
        if A[j] &lt;= pivot:
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[high]
    return i + 1
```

C++ Implementation:

```
#include &lt;iostream&gt;
using namespace std;

int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing last element as pivot
```

```
    int i = low - 1;

    for (int j = low; j &lt; high; j++) {
        if (arr[j] &lt;= 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 &lt; high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i &lt; size; i++) {
        cout &lt;&lt; arr[i] &lt;&lt; " ";
    }
    cout &lt;&lt; endl;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout &lt;&lt; "Original array: ";
    printArray(arr, size);

    quickSort(arr, 0, size - 1);

    cout &lt;&lt; "Sorted array: ";
    printArray(arr, size);

    return 0;
}
```

3. Merge Sort

Algorithm Steps:

1. Divide array into halves.
2. Recursively sort each half.
3. Merge sorted halves.

Pseudocode:

```
MERGE-SORT(A, low, high):
    if low &lt; high:
        mid = (low + high) / 2
        MERGE-SORT(A, low, mid)
        MERGE-SORT(A, mid + 1, high)
        MERGE(A, low, mid, high)

MERGE(A, low, mid, high):
    n1 = mid - low + 1
    n2 = high - mid
    L = new array[n1]
    R = new array[n2]

    for i = 0 to n1 - 1:
        L[i] = A[low + i]
    for j = 0 to n2 - 1:
        R[j] = A[mid + 1 + j]

    i = 0, j = 0, k = low
    while i &lt; n1 and j &lt; n2:
        if L[i] &lt;= R[j]:
            A[k] = L[i]
            i = i + 1
        else:
            A[k] = R[j]
            j = j + 1
        k = k + 1

    while i &lt; n1:
        A[k] = L[i]
        i = i + 1
        k = k + 1
    while j &lt; n2:
        A[k] = R[j]
        j = j + 1
        k = k + 1
```

C++ Implementation:

```
#include &lt;iostream&gt;
using namespace std;

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i &lt; n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j &lt; n2; j++) {
        R[j] = arr[mid + 1 + j];
    }
```

```
    int i = 0, j = 0, k = left;

    while (i &lt; n1 &amp;&amp; j &lt; n2) {
        if (L[i] &lt;= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i &lt; n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j &lt; n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left &lt; right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i &lt; size; i++) {
        cout &lt;&lt; arr[i] &lt;&lt; " ";
    }
    cout &lt;&lt; endl;
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout &lt;&lt; "Original array: ";
    printArray(arr, size);

    mergeSort(arr, 0, size - 1);

    cout &lt;&lt; "Sorted array: ";
    printArray(arr, size);
```

```
    return 0;
}
```

4. Linear Search

Algorithm Steps:

1. Traverse array from start to end.
2. Compare each element with target.
3. Return index if found, else -1.

Pseudocode:

```
LINEAR-SEARCH(A, target):
    for i = 0 to A.length - 1:
        if A[i] == target:
            return i
    return -1
```

C++ Implementation:

```
#include &lt;iostream&gt;
using namespace std;

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i &lt; size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1; // Not found
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 10;

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        cout &lt;&lt; "Element found at index " &lt;&lt; result &lt;&lt; endl;
    } else {
        cout &lt;&lt; "Element not found in array" &lt;&lt; endl;
    }

    return 0;
}
```

5. Binary Search

Algorithm Steps:

1. Compare target with middle element.
2. If equal, return index.
3. If smaller, search left half; else, search right.

Pseudocode:

```
BINARY-SEARCH(A, target):
    low = 0
    high = A.length - 1
    while low &lt;= high:
        mid = (low + high) / 2
        if A[mid] == target:
            return mid
        elif A[mid] &lt; target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

C++ Implementation:

```
#include &lt;iostream&gt;
using namespace std;

int binarySearch(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

    while (low &lt;= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] &lt; target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1; // Not found
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 10;

    int result = binarySearch(arr, size, target);
```

```
    if (result != -1) {
        cout &lt;&lt; "Element found at index " &lt;&lt; result &lt;&lt; endl;
    } else {
        cout &lt;&lt; "Element not found in array" &lt;&lt; endl;
    }

    return 0;
}
```

Key Points for Pseudocodes and C++ Implementations

- **Insertion Sort**: Efficient for small datasets or nearly sorted arrays.
- **Quick Sort**: Fast but can be slow if pivot is chosen poorly.
- **Merge Sort**: Stable and efficient for large datasets.
- **Linear Search**: Simple but slow for large datasets.
- **Binary Search**: Fast but requires sorted data.

Practice Problems

1. Implement Quick Sort with different pivot selection strategies.
2. Optimize Merge Sort for linked lists.
3. Compare performance of Binary Search vs. Linear Search on large datasets.

These pseudocodes and C++ implementations provide a solid foundation for understanding and implementing these algorithms.