Sorting

Comparison-based sorting

Bubble sort :

```
void BubbleSort(vector<int> arr, int n) {
    for(int i = 0; i < n-1 ; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

Improved bubble sort :

Idea : after every pass, if there is no swapping taking place, then there is no need for performing further loops

```
void BubbleSort(vector<int> arr, int n) {
    bool swapped = true;
    for(int i = n-1; i >= 0 && swapped; i--) {
        swapped = false;
        for(int j = 0; j <= i-1; j++) {
            if(arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
    }
}
```

Selection Sort :

```
void selectionSort(vector<int> arr, int n) {
    int i, j, min;
    for(i = 0; i < n-1; i++) {
        min = i;
        for(j = i+1; j < n; j++) {
            if(arr[j] < arr[i]) {
                min = j;
            }
        }
        swap(arr[min], arr[i]);
    }
}
```

Insertion Sort :

```
void insertionSort(vector<int> arr, int n) {
```

```
    int i, j, key;
    for(i = 1; i < n; i++) {
        key = arr[i];
        j = i;
        while(arr[j-1] > key && j >= 1) {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = key;
    }
}
```

Shell Sort :

```
void ShellSort(vector<int> arr, int n) {
    int i, j, h, v;
    for(h = 1; h = n/9; h = 3*h + 1);
    for(; h > 0; h=h/3) {
        for(i = h+1; i = n; i++) {
            v = arr[i];
            j = i;
            while(j > h && arr[j-h] > v) {
                arr[j] = arr[j-h];
                j = h;
            }
            arr[j] = v;
        }
    }
}
```

Merge Sort :

```
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 < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

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

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

```
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Heap Sort :

```
void heapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // Check if the left child is larger than the root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // Check if the right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If the largest is not the root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest); // Recursively heapify the affected
sub-tree
    }
}

void heapSort(int arr[], int n) {
    // Build the max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // Extract elements from the heap one by one
```

```
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the current root to the end
        heapify(arr, i, 0); // Call heapify on the reduced heap
    }
}
```

Quick Sort :

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivot element
    int i = low - 1; // Index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot in the correct
position
    return i + 1; // Return pivot index
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partitioning index
        quickSort(arr, low, pi - 1); // Sort elements before pivot
        quickSort(arr, pi + 1, high); // Sort elements after pivot
    }
}

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

    quickSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Tree Sort :

```
void treeSort(int arr[], int n) {
```

```
    Node* root = nullptr;

    // Build the BST
    for (int i = 0; i < n; i++) {
        root = insert(root, arr[i]);
    }

    // Retrieve sorted elements
    int index = 0;
    inorder(root, arr, index);
}
```

Linear sorting algorithms

Counting sort :

```
void countingSort(int arr[], int n) {
    int maxVal = *max_element(arr, arr + n);
    int minVal = *min_element(arr, arr + n);
    int range = maxVal - minVal + 1;

    vector<int> count(range, 0);
    vector<int> output(n);

    for (int i = 0; i < n; i++) {
        count[arr[i] - minVal]++;
    }

    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i] - minVal] - 1] = arr[i];
        count[arr[i] - minVal]--;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
```

Bucket sort :

```
void bucketSort(float arr[], int n) {
    vector<float> buckets[n];
```

```
    for (int i = 0; i < n; i++) {
        int index = n * arr[i];
        buckets[index].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) {
        sort(buckets[i].begin(), buckets[i].end());
    }

    int idx = 0;
    for (int i = 0; i < n; i++) {
        for (float val : buckets[i]) {
            arr[idx++] = val;
        }
    }
}
```

Radix sort :

```
void countingSortForRadix(int arr[], int n, int exp) {
    vector<int> output(n);
    vector<int> count(10, 0);

    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void radixSort(int arr[], int n) {
    int maxVal = *max_element(arr, arr + n);

    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSortForRadix(arr, n, exp);
```

```
    }
}
```

Topological Sorting

```
void topologicalSort(struct graph* G) {
    int topsort[G->V], indeg[G->V]; // arrays with (size = no of
vertices)
    int i;
    for (i = 0; i < G->V; i++) {
        indeg[i] = findIndegree(G, i);
        if (indeg[i] == 0) {
            enqueue(i);
        }
    }
    int j = 0;
    int del_node;
    while (!isEmpty(Queue)) {
        del_node = dequeue();
        topsort[j] = del_node;
        j++;
        for (i = 0; i < G->V; i++) {
            if (G->adjMatrix[del_node][i] == 1) {
                G->adjMatrix[del_node][i] = 0;
                indeg[i]--;
                if (indeg[i] == 0) {
                    enqueue(i);
                }
            }
        }
    }
    for (i = 0; i < j; i++) {
        cout << topsort[i];
    }
}
```

Problems

Problem 1 & 2:

Given an array A[0...n-1] of n numbers containing the repetition of some number. Give an algorithm for checking whether there are repeated elements or not. Assume that we are not allowed to use additional space (i.e., we can use a few temporary variables, O(1) storage).

```
int checkDuplicatesInArray(vector<int> arr, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
```

```
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Improved Complexity by sorting :

```
int checkDuplicatesInArray(vector<int> arr, int n) {
    Heapsort(arr, n);
    for(int i = 0; i < n-1; i++) {
            if(arr[i] == arr[i+1]) {
                return true;
            }
    }
    return false;
}
```

<u>Problem 3 & 4 :</u>

Given an array A[0 ...n −1], where each element of the array represents a vote in the election. Assume that each vote is given as an integer representing the ID of the chosen candidate. Give an algorithm for determining who wins the election.

```
int CheckWhoWinsTheElection(int A[], int n) {
    int i, j, counter = 0, maxCounter = 0, candidate = A[0];

    for (i = 0; i < n; i++) {
        candidate = A[i];
        counter = 0;

        for (j = i + 1; j < n; j++) {
            if (A[i] == A[j]) counter++;
        }

        if (counter > maxCounter) {
            maxCounter = counter;
            candidate = A[i];
        }
    }

    return candidate;
}
```

Improved TC :

```
int CheckWhoWinsTheElection(int A[], int n) {
    int currentCounter = 1, maxCounter = 1;
    int currentCandidate = A[0], maxCandidate = A[0];

    for (int i = 1; i < n; i++) {
        if (A[i] == currentCandidate) {
            currentCounter++;
        } else {
            currentCandidate = A[i];
            currentCounter = 1;
        }

        if (currentCounter > maxCounter) {
            maxCounter = currentCounter;
            maxCandidate = currentCandidate;
        }
    }

    return maxCandidate;
}
```

Problem 9 :

Let A and B be two arrays of n elements each. Given a number K, give an O(nlogn) time algorithm for determining whether there exists a ∈ A and b ∈ B such that a + b = K.

```
int Find(int A[], int B[], int n, int K) {
    int i, c;

    Heapsort(A, n);

    for (i = 0; i < n; i++) {
        c = K - B[i];
        if (BinarySearch(A, n, c))
            return 1;
    }

    return 0;
}
```

Problem 18 :

How do we find the number that appeared the maximum number of times in an array?

```
void FindMostFrequent(int A[], int n) {
```

```
    QuickSort(A, 0, n - 1); // Sort the array using QuickSort

    int count = 1, maxCount = 1, Number = A[0], mostFrequent = A[0];

    for (int i = 1; i < n; i++) {
        if (A[i] == A[i - 1]) {
            count++;
        } else {
            if (count > maxCount) {
                maxCount = count;
                mostFrequent = A[i - 1];
            }
            count = 1;
        }
    }

    // Final check for the last element
    if (count > maxCount) {
        maxCount = count;
        mostFrequent = A[n - 1];
    }

    cout << "Number: " << mostFrequent << ", Count: " << maxCount <<
endl;
}
```

Problem 27 :

Merge sort for linked lists

```
struct ListNode {
    int data;
    ListNode* next;
};

ListNode* LinkedListMergeSort(ListNode* first) {
    ListNode *list1HEAD = NULL, *list1TAIL = NULL;
    ListNode *list2HEAD = NULL, *list2TAIL = NULL;

    if (first == NULL || first->next == NULL)
        return first;

    while (first != NULL) {
        // Append logic for list1
        if (list1HEAD == NULL) {
            list1HEAD = list1TAIL = first;
        } else {
```

```
            list1TAIL->next = first;
            list1TAIL = first;
        }
        first = first->next;
        list1TAIL->next = NULL;

        // Append logic for list2
        if (first != NULL) {
            if (list2HEAD == NULL) {
                list2HEAD = list2TAIL = first;
            } else {
                list2TAIL->next = first;
                list2TAIL = first;
            }
            first = first->next;
            list2TAIL->next = NULL;
        }
    }

    list1HEAD = LinkedListMergeSort(list1HEAD);
    list2HEAD = LinkedListMergeSort(list2HEAD);

    // Merge logic
    ListNode* merged = NULL;
    if (list1HEAD == NULL) return list2HEAD;
    if (list2HEAD == NULL) return list1HEAD;

    if (list1HEAD->data <= list2HEAD->data) {
        merged = list1HEAD;
        merged->next = LinkedListMergeSort(list1HEAD->next);
    } else {
        merged = list2HEAD;
        merged->next = LinkedListMergeSort(list1HEAD);
    }

    return merged;
}
```

Problem 28 :

Quick sort for linked lists

```
struct ListNode {
    int data;
    ListNode* next;
};
```

```
void Qsort(ListNode** first, ListNode** last) {
    ListNode *lesHEAD = NULL, *lesTAIL = NULL;
    ListNode *equHEAD = NULL, *equTAIL = NULL;
    ListNode *larHEAD = NULL, *larTAIL = NULL;
    ListNode* current = *first;

    if (current == NULL) return;

    int pivot = current->data;
    // Append pivot node
    Append(current, equHEAD, equTAIL);

    while (current != NULL) {
        int info = current->data;
        if (info < pivot)
            Append(current, lesHEAD, lesTAIL);
        else if (info > pivot)
            Append(current, larHEAD, larTAIL);
        else
            Append(current, equHEAD, equTAIL);
    }

    // Recursively sort lesser and larger partitions
    Qsort(&lesHEAD, &lesTAIL);
    Qsort(&larHEAD, &larTAIL);

    // Join all partitions
    Join(lesHEAD, lesTAIL, equHEAD, equTAIL);
    Join(lesHEAD, larHEAD, larTAIL);

    *first = lesHEAD;
    *last = larTAIL;
}
```

Problem 33 :

There are two sorted arrays A and B. The first one is of size m + n containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m + n such that the output is sorted.

```
void Merge(int A[], int m, int B[], int n) {
    int count = m;
    int i = n - 1, j = count - 1, k = m + n - 1;

    for (; k >= 0; k--) {
```

```
        if (B[i] > A[j] || j < 0) {
            A[k] = B[i];
            i--;
            if (i < 0)
                break;
        } else {
            A[k] = A[j];
            j--;
        }
    }
}
```

Problem 38 :

Wiggle sort

```
void wiggleSort(vector<int>& nums) {
    for (int i = 0; i < nums.size() - 1; i++) {
        if ((i % 2 == 0 && nums[i] > nums[i + 1]) ||
            (i % 2 == 1 && nums[i] < nums[i + 1])) {
            swap(nums[i], nums[i + 1]);
        }
    }
}
```