CA LAB FINAL

▼ Bubble Sort

```
// Bubble Sort Best Case Time Complexity O(n)
#include <bits/stdc++.h>
using namespace std;

void swaps (int x, int y, int A[]){
   int temp = A[x];
   A[x] = A[y];
   A[y] = temp;
   return;
}

int bubble(int n, int A[]){

   for(int p=1; p<=n-1;p++){
      int flag=0;
      for(int i=0; i<= n-2;i++){
         if(A[i]> A[i+1]){
            swaps(i, i+1, A);
            flag=1;
         }
      }
      if(flag == 0) break;
   }
}



int main(){
   int n;
   cin >> n;
   int A[n];
   for(int i = 0; i < n; i++) {
      cin >> A[i];
   }

   bubble(n, A);

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

```
    return 0;
}
```

Optimized Bubble Sort | Best Case | O(n) Time Complexity | Sorting Algorithms |DAA

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶Design and Analysis of algorithms (DAA) (Complete Playlist):

https://youtu.be/g6hr8B3OWio?si=F3RRPnbi1JsdOCpi

🧠 Bubble Sort – Theory

🔷 What is Bubble Sort?

Bubble Sort is a **simple comparison-based sorting algorithm** that repeatedly:

- **Compares adjacent elements**
- **Swaps them if they are in the wrong order**

This process **"bubbles" the largest (or smallest) element** to the end (or beginning) of the list in each pass.

🔶 Characteristics:

Property	Value
Category	Comparison-based
Time Complexity (Best)	O(n)
Time Complexity (Avg)	$O(n^2)$
Time Complexity (Worst)	$O(n^2)$
Space Complexity	O(1) (in-place)
Stable	✅ Yes
Adaptive	✅ Yes (with optimization)

🔷 When to Use?

✅ For **small** or **nearly sorted** datasets

❌ Not suitable for large lists due to $O(n^2)$ time

📜 Bubble Sort – Algorithm

Here's the step-by-step **algorithm**:

1. Start from the first element.
2. Compare the current element with the next one.

3. If they are in the wrong order, swap them.
4. Repeat the process for all elements.
5. Each pass places the next-largest element at the end.
6. Repeat passes until the array is sorted.

📘 Pseudocode:

```
procedure bubbleSort(arr, n)
    for i from 0 to n - 1
        swapped = false
        for j from 0 to n - i - 2
            if arr[j] > arr[j + 1]
                swap(arr[j], arr[j + 1])
                swapped = true
        if not swapped
            break
```

✅ The `swapped` flag helps optimize and **stop early** if no swaps happen (best case).

🧪 Example:

Input: `[5, 3, 1, 4]`

Pass 1:

- 5 > 3 → swap → `[3, 5, 1, 4]`
- 5 > 1 → swap → `[3, 1, 5, 4]`
- 5 > 4 → swap → `[3, 1, 4, 5]`

Pass 2:

- 3 > 1 → swap → `[1, 3, 4, 5]`
- 3 < 4 → no swap
- 4 < 5 → no swap

Pass 3:

- 1 < 3 → no swap
- No swaps in full pass → **sorted early**

✅ Final Notes for Exam:

- **Stable:** Equal elements stay in same relative order.
- **In-place:** Uses no extra space.
- **Best Case:** O(n) when array is already sorted.
- **Worst/Average Case:** O(n^2)

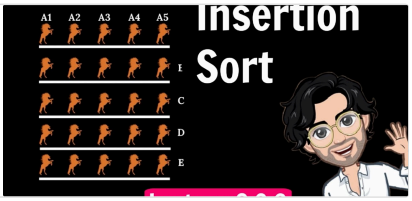
▼ Insertion Sort

```
// Insertion Sort Best Case Time Complexity O(n)
#include <bits/stdc++.h>
using namespace std;

int main()
{

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

   for (int i = 1; i < n; i++)
   {
      int current = arr[i];
      int j = i - 1;
      while (arr[j] > current && j >= 0)
      {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j+1] = current;
   }

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

   return 0;
}
```

8.3.3 Insertion Sort | Sorting Algorithms | C++ Placement Course |

Notes: Same as Lecture 8.3.1

https://youtu.be/3GC83dh4cf0?si=MlIA8Qjs2_M8_N3Y



🧠 Insertion Sort – Theory

🔷 What is Insertion Sort?

Insertion Sort is a simple and intuitive **comparison-based sorting algorithm**. It works like the way you sort playing cards in your hands:

- Start with the first card
- Pick the next card and insert it into the correct position among the sorted cards

✅ It builds the sorted array one element at a time.

🔶 Characteristics:

Property	Value
Category	Comparison-based
Time Complexity (Best)	O(n)
Time Complexity (Avg)	$O(n^2)$
Time Complexity (Worst)	$O(n^2)$
Space Complexity	O(1) (in-place)
Stable	✅ Yes
Adaptive	✅ Yes

🔷 When to Use?

✅ Best for **small** or **nearly sorted** arrays

✅ Performs better than Bubble Sort in practice

❌ Not suitable for large datasets

📜 Insertion Sort – Algorithm

✍️ Step-by-step:

1. Start from the **second element** (index 1).
2. Compare it with elements before it.
3. **Shift larger elements** one position to the right.
4. Insert the current element into the correct position.
5. Repeat for all elements.

📘 Pseudocode:

```
procedure insertionSort(arr, n)
   for i from 1 to n-1
      key = arr[i]
      j = i - 1
```

```
        while j >= 0 and arr[j] > key
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

🧪 Example:

Input: `[5, 3, 1, 4]`

- i = 1 → key = 3 → 5 > 3 → shift → `[5, 5, 1, 4]` → insert 3 → `[3, 5, 1, 4]`
- i = 2 → key = 1 → shift 5, 3 → `[3, 5, 5, 4]` → `[1, 3, 5, 4]`
- i = 3 → key = 4 → shift 5 → `[1, 3, 5, 5]` → insert 4 → `[1, 3, 4, 5]`

✅ Final array: `[1, 3, 4, 5]`

✨ Advantages:

- ✅ Simple to implement
- ✅ Works well for small and nearly sorted lists
- ✅ Stable and in-place

📝 Key Points for Exams:

- **Best case (already sorted):** O(n)
- **Worst case (reverse sorted):** $O(n^2)$
- **Stable:** Yes
- **Space complexity:** O(1)
- **In-place:** Yes

▼ Merge Sort

```
// Merge Sort Best Case Time Complexity O(n log n)
#include <bits/stdc++.h>
using namespace std;

void merge(int arr[], int l, int mid, int r)
{
    int n1 = mid - l + 1;
    int n2 = r - mid;
    int a[n1];
    int b[n2];
    for (int i = 0; i < n1; i++)
    {
```

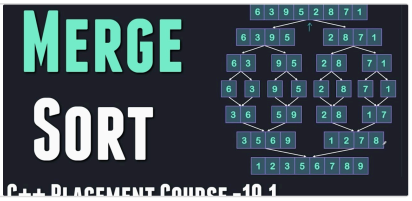
```
        a[i] = arr[l + i];
    }
    for (int i = 0; i < n2; i++)
    {
        b[i] = arr[mid + 1 + i];
    }
    int i = 0;
    int j = 0;
    int k = l;
    while (i < n1 && j < n2)
    {
        if (a[i] < b[j])
        {
            arr[k] = a[i];
            k++;
            i++;
        }
        else
        {
            arr[k] = b[j];
            k++, j++;
        }
    }
    while (i < n1)
    {
        arr[k] = a[i];
        k++;
        i++;
    }

    while (j < n2)
    {
        arr[k] = b[j];
        k++, j++;
    }
}

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

    if (l < r)
    {
        int mid = (l + r) / 2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid + 1, r);
        merge(arr, l, mid, r);
    }
```

```
}

int main()
{

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

    mergeSort(arr, 0, n - 1);

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

    return 0;
}
```

Merge Sort | Code and Explanation | C++ Course - 19.1

Complete C++ Placement Course (Data Structures+Algorithm)
:https://www.youtube.com/playlist?list=PLfqMhTWNBTe0b2nM6JHVCnAkhQRGiZMSJ
Telegram: https://t.me/apnikakshaofficial

https://youtu.be/4z9l6ZmeLOQ?si=9XY7VuD-bv2MvMOx



🧠 Merge Sort – Theory

🔷 What is Merge Sort?

Merge Sort is a **divide-and-conquer** sorting algorithm that:

1. **Divides** the array into two halves
2. **Sorts** each half recursively
3. **Merges** the two sorted halves into one

It continues dividing until each subarray has only **one element**, and then merges them back in **sorted order**.

🔶 Characteristics

Property	Value
Category	Divide & Conquer
Time Complexity (Best)	O(n log n)
Time Complexity (Avg)	O(n log n)
Time Complexity (Worst)	O(n log n)
Space Complexity	❌ O(n) extra space
Stable	✅ Yes
In-place	❌ No

🔷 When to Use?

✅ When consistent performance is needed

✅ For large datasets

✅ When stability matters

❌ Not ideal when **in-place sorting** or **low memory usage** is required

📜 Merge Sort – Algorithm

✍️ Step-by-step:

1. Divide the array into two halves
2. Recursively sort both halves
3. Merge the sorted halves

📘 Pseudocode

```
procedure mergeSort(arr[], left, right)
    if left < right
        mid = (left + right) / 2
        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)

procedure merge(arr[], left, mid, right)
    create temp arrays L[] and R[]
    copy data into L[] and R[]
    merge the two arrays into arr[]
```

🧪 Example:

Input: `[5, 3, 1, 4]`

Step 1: Divide

- `[5, 3]` and `[1, 4]`

Step 2: Sort

- `[3, 5]` and `[1, 4]`

Step 3: Merge

- Merge `[3, 5]` and `[1, 4]` → `[1, 3, 4, 5]`

✅ Advantages

- ✅ Guaranteed **O(n log n)** time
- ✅ **Stable** sort (does not change order of equal elements)
- ✅ Works well with **linked lists**

❌ Disadvantages

- ❌ Uses extra memory (O(n))
- ❌ Not in-place

✅ Key Points for Exam:

Concept	Value
Type	Divide & Conquer
Time Complexity	O(n log n) all cases
Space	O(n) (not in-place)
Stable	✅ Yes
Use Case	Large, stable sort required

▼ Quick Sort

```
// Quick Sort Best Case Time Complexity O(n log n)
#include <bits/stdc++.h>
using namespace std;

void swap(int arr[], int i , int j){
    int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
}

int partition(int arr[], int l, int r){
    int pivot = arr[r];
```

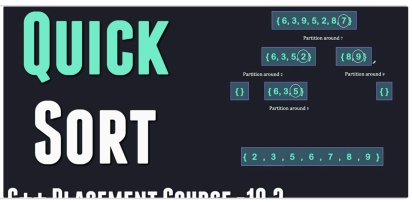
```
    int i = l-1;
    for (int j=l; j<r; j++){
        if(arr[j] < pivot){
            i++;
            swap(arr, i, j);
        }

    }
    swap(arr,i+1,r);
    return i+1;
}

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

    if(l<r){
        int pi = partition (arr, l ,r );
        quickSort(arr,l,pi-1);
        quickSort(arr,pi+1,r);
    }
}

int main(){

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

    quickSort(arr,0,n-1);

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


    return 0;
}
```

Case	Time Complexity	Explanation
Best Case	O(n log n)	Pivot divides array in half
Average	O(n log n)	Good balance in partitions
Worst	O(n^2)	Already sorted / reverse sorted

Quick Sort | Code and Explanation | C++ Course - 19.2

Complete C++ Placement Course (Data Structures+Algorithm)
:https://www.youtube.com/playlist?list=PLfqMhTWNBTe0b2nM6JHVCnAkhQRGiZMSJ
Telegram: https://t.me/apnikakshaofficial

https://youtu.be/DI6HT-NM_q4?si=gYfEPyatB96kr73y



🧠 Quick Sort – Theory

🔷 What is Quick Sort?

Quick Sort is a **divide-and-conquer sorting algorithm** that:

1. Picks a **pivot element**
2. **Partitions** the array into two:
   - Elements **less than pivot**
   - Elements **greater than or equal to pivot**
3. Recursively applies the same steps to the left and right subarrays

🚀 Known for being **very fast in practice** and **in-place**

🔶 Characteristics

Property	Value
Category	Divide & Conquer
Time Complexity (Best)	O(n log n)
Time Complexity (Avg)	O(n log n)
Time Complexity (Worst)	❌ $O(n^2)$ (when poorly balanced)
Space Complexity	✅ O(log n) (stack)
In-place	✅ Yes
Stable	❌ No (by default)

🔷 When to Use?

✅ Large datasets

✅ When **memory efficiency** is important

❌ Not great when **stability** is required

📜 Quick Sort – Algorithm

✍️ Step-by-step:

1. Select a **pivot** (commonly the last or first element)
2. Partition the array:

- Move smaller elements to left of pivot
- Move larger elements to right

3. Recursively apply Quick Sort to left and right subarrays

📘 Pseudocode

```
procedure quickSort(arr[], low, high)
   if low < high
      pivotIndex = partition(arr, low, high)
      quickSort(arr, low, pivotIndex - 1)
      quickSort(arr, pivotIndex + 1, high)

procedure partition(arr[], low, high)
   pivot = arr[high]
   i = low - 1
   for j = low to high - 1
      if arr[j] < pivot
         i++
         swap arr[i] and arr[j]
   swap arr[i + 1] and arr[high]
   return i + 1
```

🧪 Example:

Input: `[5, 3, 1, 4]`

Pivot = 4

Partition:

- Elements < 4 → `[3, 1]`
- Elements > 4 → `[5]`

Subarrays:

- `[3, 1]` and `[5]`
- Recur and sort them → `[1, 3, 4, 5]`

✅ Advantages

- ✅ **Fast in practice** — one of the fastest sorting algorithms
- ✅ **In-place**, minimal memory
- ✅ Average time: O(n log n)

❌ Disadvantages

- ❌ **Worst-case** is $O(n^2)$ if array is already sorted

- ❌ **Not stable** (unless modified)

📝 Key Points for Exam

Feature	Detail
Type	Divide & Conquer
Time Complexity	O(n log n) average
Worst Case	O(n²)
Space Complexity	O(log n) stack
Stable	❌ No
In-place	✅ Yes

▼ Heap Sort

```
#include <bits/stdc++.h>
using namespace std;

void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   if (l < n && arr[l] > arr[largest])
      largest = l;
   if (r < n && arr[r] > arr[largest])
      largest = r;

   if (largest != i)
   {
      swap(arr[i], arr[largest]);
      heapify(arr, n, largest);
   }
}

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

   for (int i = n - 1; i >= 0; i--)
   {
      swap(arr[0], arr[i]);
      heapify(arr, i, 0);
```

```
    }
}

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

    heapSort(arr, n);

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

    return 0;
}
```

L-3.8: Introduction to Heap Tree with examples | Max Min Heap

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

0:00 - Heap Tree

https://youtu.be/uuot9ltgTEI?si=Km9pm_VOH0YSz9RQ

L-3.11: Build Heap in O(n) time complexity | Heapify Method | Full Derivation with example

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

0:00 - Heapify method

https://youtu.be/8noP3YjjJCM?si=QrEPaNoT7Hm4L-vw

L-3.13: Heap sort with Example | Heapify Method

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶Heap Tree: https://youtu.be/uuot9ltgTEI

https://youtu.be/nJ6FdAIr_6g?si=mfUOTfmoL9sbD7pl

🧠 Heap Sort – Theory

🔷 What is Heap Sort?

Heap Sort is a **comparison-based**, **in-place** sorting algorithm that uses a **binary heap data structure**.

✅ It converts the input array into a **max heap**, and then:

1. Repeatedly extracts the **maximum element** (at the root)
2. Swaps it with the last element
3. Heapifies the reduced hea

🔶 Key Concepts

- A **heap** is a complete binary tree
- A **max heap** has the largest element at the root
- Sorting is done by repeatedly removing the root

🔶 Characteristics

Property	Value
Type	Comparison-based
Time Complexity (Best)	O(n log n)
Time Complexity (Avg)	O(n log n)
Time Complexity (Worst)	O(n log n)
Space Complexity	✅ O(1) (in-place)
Stable	❌ No
In-place	✅ Yes

📝 When to Use?

✅ When **in-place sorting** is required

✅ When **O(n log n)** worst-case time is needed

❌ Not preferred if **stability** is important

📜 Heap Sort – Algorithm

✍️ Step-by-step:

1. **Build Max Heap** from the input array
2. For each element from end to start:
    - Swap it with the root (max element)
    - Reduce the heap size
    - Heapify the root

📘 Pseudocode

```
procedure heapSort(arr[], n)
    // Step 1: Build Max Heap
    for i from n/2 - 1 down to 0
        heapify(arr, n, i)

    // Step 2: Extract elements one by one
    for i from n-1 down to 1
        swap(arr[0], arr[i])       // move max to end
        heapify(arr, i, 0)         // fix heap

procedure heapify(arr[], n, i)
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]
        largest = left
    if right < n and arr[right] > arr[largest]
        largest = right

    if largest != i
        swap(arr[i], arr[largest])
        heapify(arr, n, largest)
```

🧪 Example:

Input: `[4, 10, 3, 5, 1]`

- Build max heap → `[10, 5, 3, 4, 1]`
- Swap 10 with last → `[1, 5, 3, 4, 10]`
- Heapify root → `[5, 4, 3, 1, 10]`
- Swap 5 with index 3 → `[1, 4, 3, 5, 10]` and so on...

✅ Final output: `[1, 3, 4, 5, 10]`

✅ Advantages

- ✅ **Worst-case time is O(n log n)** (better than Quick Sort in worst case)
- ✅ **In-place** — no extra memory needed
- ✅ Good for data stored in arrays

❌ Disadvantages

- ❌ **Not stable** (order of equal elements can change)
- ❌ Slightly slower than Quick Sort in practice (due to more comparisons/swaps)

✅ Key Points for Exams

Feature	Value
Type	Comparison-based, in-place
Time Complexity	O(n log n)
Space	O(1)
Stable	❌ No
Heap Used	✅ Max Heap

▼ Redix Sort

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

// Counting sort based on digit represented by exp
void countingSort(int arr[], int n, int exp) {
    int output[n]; // Output array
    int count[10] = {0}; // Count array for digits 0-9

    // Count occurrences of digits
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Update count[i] to be position index
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array (stable sort)
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy back to original array
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

// Main radix sort function
void radixSort(int arr[], int n) {
    // Find the maximum number to know number of digits
    int maxNum = arr[0];
    for (int i = 1; i < n; i++)
```

```
        if (arr[i] > maxNum)
            maxNum = arr[i];

    // Apply counting sort for every digit
    for (int exp = 1; maxNum / exp > 0; exp *= 10)
        countingSort(arr, n, exp);
}

int main() {
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++) cin >> arr[i];

    radixSort(arr, n);

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

    return 0;
}
```

Case	Time Complexity
Best	O(n * k) where k = # of digits
Average	O(n * k)
Worst	O(n * k)

Radix Sort | Easiest explanation with example

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

👉Links for DAA Notes:

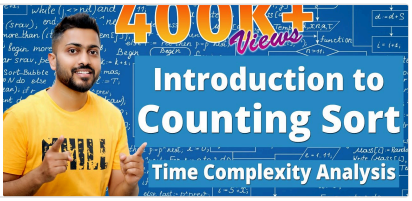
https://youtu.be/9QSgBO9yjKU?si=ZhxC2f2GvHTlEbC_

Counting Sort | Easiest explanation with example

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

👉Links for DAA Notes:

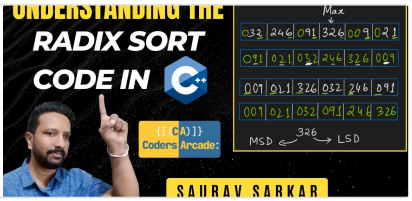
https://youtu.be/mowMVn9wTnE?si=RjVHHHaElJn7iv5c



🔥Mastering Radix Sort Code In C++ || Data Structures And Algorithms🔥

Title: Mastering Radix Sort: Code Demystified || Unraveling Algorithmic Brilliance, Part 2 || DSA

https://youtu.be/NqvXGtz9kr4?si=tKNYJwXWlxYUpxil



🧠 Radix Sort – Theory

🔷 What is Radix Sort?

Radix Sort is a **non-comparison-based** sorting algorithm that sorts integers (or strings) **digit by digit**, starting from the **least significant digit (LSD)** to the **most significant digit (MSD)**.

✅ It uses a **stable sorting algorithm** (like Counting Sort) as a subroutine to sort digits.

🔶 Key Characteristics

Property	Value
Type	Non-comparison sort
Time Complexity (Best)	O(nk)
Time Complexity (Avg)	O(nk)
Time Complexity (Worst)	O(nk)
Space Complexity	O(n + k)
Stable	✅ Yes
In-place	❌ No (uses extra space)

Here,

- `n` = number of elements
- `k` = number of digits in the largest number

🔶 When to Use?

✅ Best when:

- Sorting integers (or strings)
- You need **linear-time sorting**
- Numbers have **fixed number of digits**

  ❌ Not suitable for floating point numbers or huge values with many digits

---

📜 Radix Sort – Algorithm

✍️ Step-by-step:

1. Find the **maximum number** to know the number of digits `k`.
2. Use **Counting Sort** to sort the array **by each digit**, starting from **least significant digit (LSD)**.
3. Repeat for each digit place (units, tens, hundreds...).

📘 Pseudocode

```
procedure radixSort(arr[], n)
    maxVal = find maximum value in arr[]
    for exp = 1; maxVal / exp > 0; exp *= 10
```

```
        countingSortByDigit(arr, n, exp)

procedure countingSortByDigit(arr[], n, exp)
    initialize count[10] = {0}
    output[n]

    for i = 0 to n-1
        count[(arr[i]/exp) % 10]++

    for i = 1 to 9
        count[i] += count[i-1]

    for i = n-1 downto 0
        place arr[i] in correct output[] position using count
        decrease count

    copy output[] back to arr[]
```

🧪 Example:

Input: `[170, 45, 75, 90, 802, 24, 2, 66]`

Step 1: Sort by 1s place → `[170, 90, 802, 2, 24, 45, 75, 66]`

Step 2: Sort by 10s place → `[802, 2, 24, 45, 66, 170, 75, 90]`

Step 3: Sort by 100s place → `[2, 24, 45, 66, 75, 90, 170, 802]`

✅ Final sorted array: `[2, 24, 45, 66, 75, 90, 170, 802]`

✅ Advantages

- ✅ **Linear time** for fixed-length integers
- ✅ **Stable**
- ✅ Very efficient for large lists of numbers

❌ Disadvantages

- ❌ Not in-place (uses extra space)
- ❌ Only works well with **integers** or **fixed-length strings**
- ❌ Harder to implement than comparison-based sorts

✅ Key Points for Exams

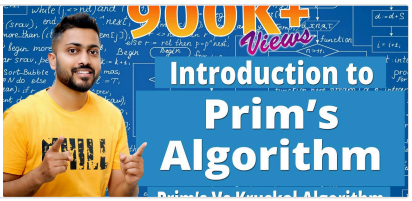
Feature	Value
Category	Non-comparison sort

Stable	✅ Yes
Time Complexity	O(nk)
Space Complexity	O(n + k)
Uses	Counting Sort (as subroutine)

▼ Minimum Spanning Tree (Prim's or Kruskal's)

L-4.9: Prim's Algorithm for Minimum Cost Spanning Tree | Prims vs Kruskal
👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

👉Links for DAA Notes:
https://youtu.be/_KX8GDvRzBc?si=xbWQXmA1Nn4gXrAT



L-4.8: Kruskal Algorithm for Minimum Spanning Tree in Hindi | Algorithm
#KruskalAlgorithm#MinimumSpanningTree#Algorithm

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots
https://youtu.be/huQojf2tevI?si=tWEnSbW3SEqiAXG8

🌲 What is a Minimum Spanning Tree (MST)?

A **Minimum Spanning Tree (MST)** of a **connected, undirected, weighted graph** is a subset of edges that:

- Connects **all vertices**
- Has **no cycles**
- Has the **minimum possible total weight**

For a graph with `V` vertices, MST has exactly **V - 1** edges.

✅ Use Cases of MST:

- Network design (telecom, water pipelines, electric grids)
- Clustering algorithms
- Approximation algorithms (like TSP)

📘 Common MST Algorithms:

Algorithm	Type	Works With	Time Complexity	Extra Notes
Prim's	Greedy	Adjacency Matrix / List	$O(V^2)$ or O(E log V)	Good for dense graphs
Kruskal's	Greedy	Edge List	O(E log E)	Good for sparse graphs

🔷 Prim's Algorithm – Theory & Algorithm

🧠 Theory

- Start with any vertex
- Grow the MST by **adding the smallest edge** that connects a visited vertex to an unvisited vertex
- Repeat until all vertices are in the MST

✅ Uses **priority queue (or minKey array)**

✅ Keeps track of included vertices

📘 Prim's Pseudocode

```
procedure primMST(graph, V)
    key[] = ∞ for all vertices
    mstSet[] = false for all vertices
    key[0] = 0, parent[0] = -1

    for count = 0 to V-1
        u = vertex with minimum key not in mstSet
        mstSet[u] = true

        for each v adjacent to u
            if graph[u][v] and !mstSet[v] and graph[u][v] < key[v]
                parent[v] = u
                key[v] = graph[u][v]
```

🔷 Kruskal's Algorithm – Theory & Algorithm

🧠 Theory

- Sort all edges in **ascending order** of weight
- Add the smallest edge **that doesn't form a cycle**
- Use **Union-Find (Disjoint Set Union)** to detect cycles
- Stop when you've added **V-1 edges**

✅ Works on **edge list**, not adjacency matrix

✅ Best for sparse graphs

📘 Kruskal's Pseudocode

```
procedure kruskalMST(edges, V)
    sort edges by weight
    parent[] = makeSet(V)
```

```
    for edge in sorted edges
        if find(edge.u) != find(edge.v)
            add edge to MST
            union(edge.u, edge.v)
```

🧪 Example Graph:

```
Vertices: 4
Edges:
0-1: 10
0-2: 6
0-3: 5
1-3: 15
2-3: 4
```

MST:

- Edges: 2-3, 0-3, 0-1
- Total Weight: 4 + 5 + 10 = **19**

✅ Summary Table

Feature	Prim's	Kruskal's
Approach	Greedy (Grow MST)	Greedy (Add edges)
Data Structure	Adjacency matrix/list	Edge list + DSU
Best for	Dense graphs	Sparse graphs
Time Complexity	$O(V^2)$ / O(E log V)	O(E log E)
Stable	Not applicable	Not applicable

▼ Activity Selection Problem

Step	Time
Sort	O(n log n)
Selection	O(n)
Total	✅ O(n log n)

Activity Selection Problem - Greedy Algorithm | C++ Placement Course | Lecture 33.3

Complete C++ Placement Course (Data Structures+Algorithm)
:https://www.youtube.com/playlist?list=PLfqMhTWNBTe0b2nM6JHVCnAkhQRGiZMSJ
Telegram: https://t.me/apnikakshaofficial

https://youtu.be/DHr-Mn_vzs0?si=hyev3qA9EFhaVsg1

```
#include <bits/stdc++.h>
using namespace std;

int main (){

    int n;
    cin>>n;
    vector<vector<int>> V;
    for(int i=0; i<n; i++){
        int start, end;
        cin >> start >> end;
        V.push_back({start,end});

    }
    sort(V.begin(),V.end(),[&](vector<int> &a, vector<int> &b){

        return a[1] < b[1];

    });

    int take = 1;
    int end = V[0][1];
    for(int i = 1; i<n;i++){
        if(V[i][0] >= end){
            take++;
            end = V[i][1];
        }
    }
    cout << take << "\n";

    return 0;
}
```

🧠 Activity Selection Problem – Theory

🔷 What is the Problem?

You're given:

- A list of `n` activities
- Each activity has a **start time** and an **end time**

🎯 **Goal:** Select the **maximum number of non-overlapping activities** that can be performed by a single person.

🔶 Real-Life Example:

- Scheduling interviews in a room
- Choosing the maximum number of TV shows to watch without overlap

---

🔶 Characteristics

Feature	Value
Problem Type	Greedy Algorithm
Input	Start & End times
Goal	Maximize activity count without overlaps
Strategy	Always choose the activity that **ends earliest**

📜 Greedy Strategy

Why choose the activity that ends earliest?

Because this **frees up time** for the rest of the activities.

This strategy ensures we can accommodate the **maximum number** of future activities.

📘 Activity Selection – Algorithm

✍️ Step-by-step:

1. **Sort all activities by their end time**
2. Select the **first activity** (it ends the earliest)
3. For each next activity:
    - If its **start time ≥ end time** of the last selected activity, **select it**
4. Continue until all activities are considered

📘 Pseudocode:

```
procedure activitySelection(start[], end[], n)
    sort activities by end time
    select first activity
    for i = 1 to n-1
        if start[i] >= end[last_selected]
            select activity i
```

🧪 Example:

Input:

```
Activities:
Start  = [1, 3, 0, 5, 8, 5]
End    = [2, 4, 6, 7, 9, 9]
```

Step-by-step:

- Sort by end:

  Activities become → [(1,2), (3,4), (0,6), (5,7), (8,9), (5,9)]
- Select (1,2)
- Next: (3,4) → start ≥ 2 → ✅
- Next: (0,6) → start < 4 → ❌
- Next: (5,7) → start ≥ 4 → ✅
- Next: (8,9) → start ≥ 7 → ✅

✅ Selected: **(1,2), (3,4), (5,7), (8,9)**

✅ **Total = 4 activities**

⏱️ Time Complexity

Step	Time
Sorting	O(n log n)
Selection	O(n)
Total	✅ O(n log n)

✅ Advantages

- ✅ Simple and fast
- ✅ Optimal for single resource (room, machine, person)
- ✅ Excellent example of greedy strategy

❌ Limitations

- ❌ Works only if activities are sorted by end time
- ❌ For multiple people/machines, this won't give optimal solution (use interval partitioning)

✅ Final Notes for Exam

Point	Description
Problem Type	Greedy
Input	Start and end time arrays
Strategy	Sort by end time, pick earliest finish

Time Complexity	O(n log n)

▼ Fractional Knapsack Problem

L-4.2: Knapsack Problem With Example| Greedy Techniques| Algorithm
👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

👉Links for DAA Notes:
https://youtu.be/M79iHjAG1tg?si=JtCtbGYTKHAPbcF9

🧠 Fractional Knapsack – Theory

🔷 What is the Problem?

You're given:

- A list of `n` items
- Each item has a **value** and a **weight**
- A **knapsack** with maximum capacity `W`

🎯 **Goal:** Maximize total value in the knapsack

✅ You **can take fractions** of an item (unlike 0/1 Knapsack)

🔶 Real-Life Example

Imagine filling a backpack with gold, sugar, or liquid — you can take any portion of it.

✅ Greedy Strategy

To get maximum value:

1. Calculate **value/weight ratio** for each item
2. Sort items by this ratio in **descending order**
3. Pick items starting from the top until the knapsack is full
4. If the next item can't fit, take the **fraction** of it

📘 Characteristics

Feature	Value
Problem Type	Greedy
Time Complexity	O(n log n)
Space Complexity	O(1) (excluding input)

Fractional allowed?	✅ Yes
Optimal solution?	✅ Yes (Greedy works)

📜 Fractional Knapsack – Algorithm

✍️ Step-by-step:

1. Calculate **value/weight ratio** for each item
2. Sort all items by ratio in descending order
3. Initialize `totalValue = 0`
4. For each item:
   - If it fits, take all of it
   - If not, take the fraction that fits
5. Stop when knapsack is full

📘 Pseudocode

```
procedure fractionalKnapsack(W, items[])
    sort items by value/weight ratio in descending order
    totalValue = 0

    for each item in sorted list:
        if item.weight <= W:
            W -= item.weight
            totalValue += item.value
        else:
            totalValue += item.value * (W / item.weight)
            break

    return totalValue
```

🧪 Example

Input:

```
Items: (value, weight)
(60, 10), (100, 20), (120, 30)
Knapsack capacity = 50
```

- Value/weight: 6, 5, 4
- Pick:
  - (60,10) → full

- (100,20) → full
- (120,30) → take 20/30 = ⅔ → value = 80

✅ Output:

```
Total max value = 60 + 100 + 80 = 240
```

⏱️ Time & Space Complexity

Step	Time
Sort	O(n log n)
Selection	O(n)
Total	✅ O(n log n)

Space: O(1) (if in-place sort used)

✅ Final Notes for Exam

Feature	Detail
Type	Greedy
Can take fraction?	✅ Yes
Uses	value/weight ratio
Time Complexity	O(n log n)
Space Complexity	O(1)
Optimal?	✅ Yes, greedy works perfectly
