আন্তর্জাতিক ইসলামী বিশ্ববিদ্যালয় চট্টগ্রাম

International Islamic University Chittagong

Department of Computer Science & Engineering

CSE-2422 : Algorithm Lab

Session : Spring-2025

Lab Report

Submitted to:

Tuly

Submitted by:

Tahasina Tasnim Afra - C233456

Semester: 4th

Section: 4BF

Contents:

Linear Search	3
Binary Search	4
Bubble Sort	5
Insertion Sort	6
Selection Sort	7
Merge Sort	8
Counting Sort	10
Quick Sort	11
Heap Sort	12
Radix Sort	14

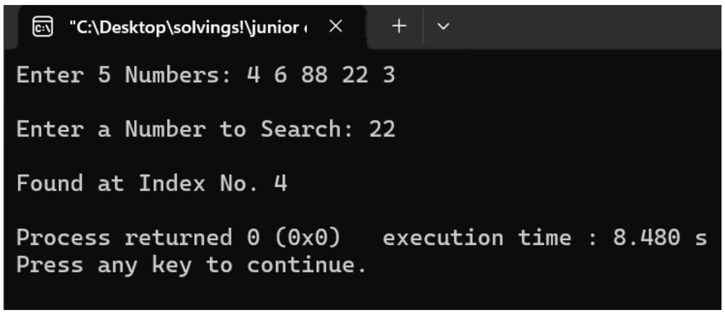
Linear Search

Objective: Search a number in the array by checking each element one by one.

Code:

```
for(int i = 0; i < n; i++) {
    if(arr[i] == num) {
      index = i;
      break;
   }
}
```

Output:



```
Enter 5 Numbers: 4 6 88 22 3

Enter a Number to Search: 22

Found at Index No. 4

Process returned 0 (0x0)   execution time : 8.480 s
Press any key to continue.
```

Description:

Linear Search works by sequentially checking each element of the array until the target element is found or the end of the array is reached. 1. Start at the first element of the array. 2. Check if it matches the target. 3. Move to the next element and repeat until the target is found or the end of the array is reached.

Time Complexity:

- Best Case: O(1) – The target is found at the first element.
- Worst Case: O(n) – The target is at the last element or not present at all.
- Average Case: O(n) – On average, half of the elements are checked.

Space Complexity:

- O(1) – Linear Search is performed in-place, using constant space.

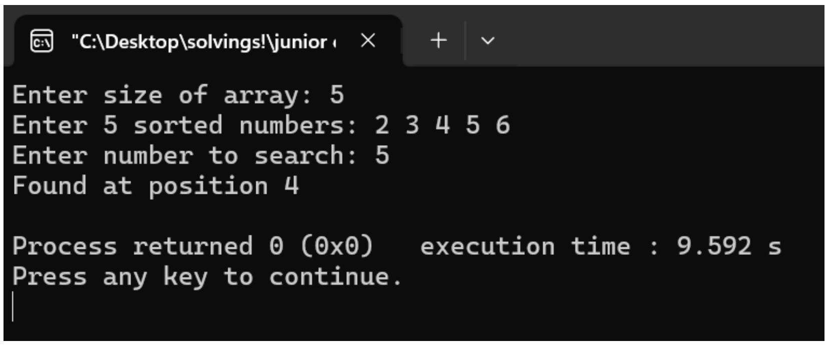
Binary Search

Objective: Efficiently search a sorted array by repeatedly dividing the search range in half.

Code:

```
cin >> num;
    int low = 0, high = n - 1, mid, found = 0;
   while(low <= high) {
      mid = (low + high) / 2;
      if(arr[mid] == num) {
         cout << "Found at position " << mid + 1 << endl;
         found = 1;
         break ;  }
      else if(arr[mid] < num)
         low = mid + 1;
      else
         high = mid - 1;  }
   if(!found) cout << "Not found!" << endl;
```

Output:



```
Enter size of array: 5
Enter 5 sorted numbers: 2 3 4 5 6
Enter number to search: 5
Found at position 4

Process returned 0 (0x0)   execution time : 9.592 s
Press any key to continue.
```

Description:

Binary Search is an efficient algorithm for finding an element in a sorted array. It works by repeatedly dividing the search interval in half. If the target value is less than the value at the middle of the interval, the search continues in the lower half; otherwise, it continues in the upper half.

1. Start with the entire array as the search interval.
2. Find the middle element.
3. If the middle element is the target, return it.
4. If the target is less than the middle element, repeat the process on the left half. 5. If the target is greater, repeat on the right half.

Time Complexity:

- Best Case: O(1) – If the middle element is the target.
- Worst Case: O(log n) – In each step, the search space is halved.
- Average Case: O(log n) – The algorithm always halves the search space.

Space Complexity: O(1) & O(log n) - If using a recursive version, due to the call stack.

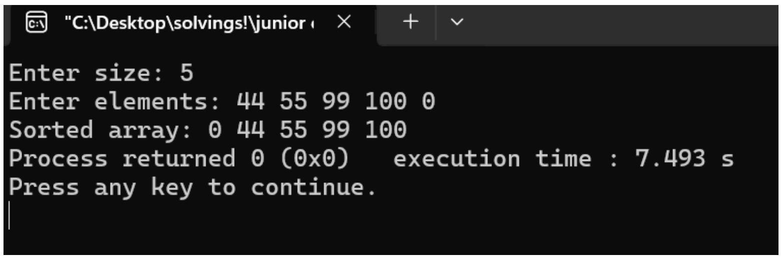
Bubble Sort

Objective: Sort elements by repeatedly swapping adjacent elements if they are in the wrong order.

Code:

```
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]);
```

Output:



```
Enter size: 5
Enter elements: 44 55 99 100 0
Sorted array: 0 44 55 99 100
Process returned 0 (0x0)   execution time : 7.493 s
Press any key to continue.
```

Description:

Bubble Sort works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. After each pass, the largest (or smallest) element "bubbles" up to its correct position.

1. Compare adjacent elements and swap them if they are in the wrong order.
2. Repeat the process for the entire array but reduce the comparison range after each pass (since the largest element moves to the end).

Time Complexity:

- Best Case: O(n) – If the array is already sorted, only one pass is needed. Also, optimization with a flag to break early if no swaps are made.
- Worst Case: O(n^2)
- Average Case: O(n^2)

Space Complexity: O (1)

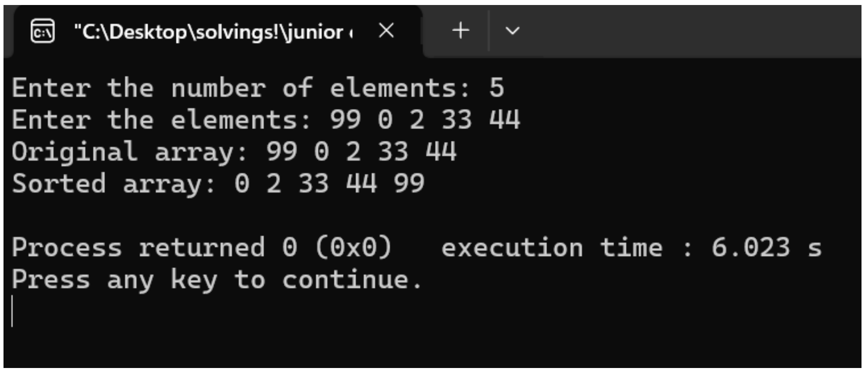
Insertion Sort

Objective: Build the sorted array one element at a time by inserting elements into the correct position.

Code:

```
void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
    int key = arr[i]; // Element to be placed at its correct position
    int j = i - 1;
    // Move elements of the sorted portion that are greater than key
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;}
    arr[j + 1] = key; // Place the key in its correct position} }
```

Output:



```
Enter the number of elements: 5
Enter the elements: 99 0 2 33 44
Original array: 99 0 2 33 44
Sorted array: 0 2 33 44 99

Process returned 0 (0x0)   execution time : 6.023 s
Press any key to continue.
```

Description:

Insertion Sort works by building a sorted portion of the array one element at a time. Starting from the second element, each element is compared with the elements in the sorted portion (to the left) and inserted into its correct position. This continues until all elements are sorted.

1. Start with the second element of the array.
2. Compare the current element with the elements before it and insert it into its correct position in the sorted portion. If the element is smaller (in-case of increasing sort) than an element to its left, right-shift the larger element(s) to make space for the current element.
3. Move to the next element and repeat the process.

Time Complexity:

- Best Case: O(n) – The array is already sorted, and only a single pass is needed with no swaps
- Worst Case: O(n^2) – The array is sorted in reverse order, and for each element, all previous elements need to be compared.

AFRA C233456

- Average Case: O(n^2) – In the general case, about half of the comparisons result in swaps.

Space Complexity: O (1) – Insertion Sort sorts the array in-place, using a constant amount of extra space.

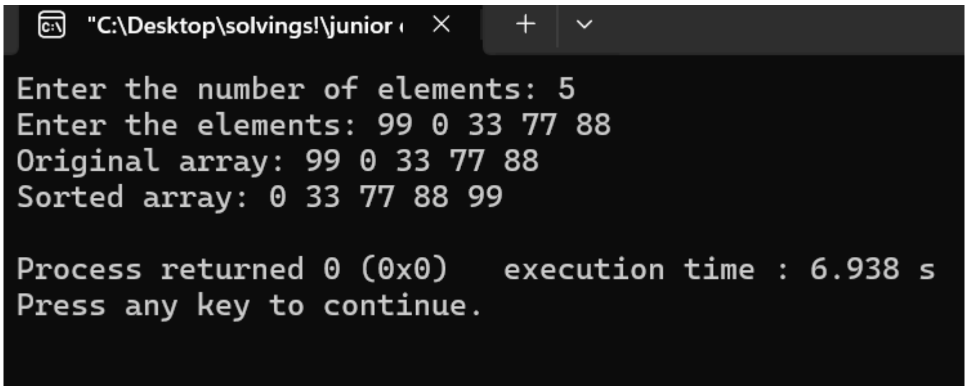
Selection Sort

Objective:
Sort elements by repeatedly selecting the smallest (or largest) element and placing it in correct position.

Code:

```
void selectionSort(int arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int minIndex = i; // Assume the first unsorted element is the smallest
    for (int j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) { minIndex = j; // Update the indx of the smallest elem }}
   // Swap the smallest element with the first unsorted element
    swap(arr[i], arr[minIndex]);
  }}
```

Output:



```
"C:\Desktop\solvings!\junior

Enter the number of elements: 5
Enter the elements: 99 0 33 77 88
Original array: 99 0 33 77 88
Sorted array: 0 33 77 88 99

Process returned 0 (0x0)   execution time : 6.938 s
Press any key to continue.
```

Description:

Selection Sort works by dividing the array into a sorted and an unsorted part. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion and swaps it with the first unsorted element. This process continues until the entire array is sorted. One notable advantage of Selection Sort is that it performs only one swap per pass, making it useful in scenarios where swapping is more expensive than comparing (e.g., with large data structures or memory-limited systems).

1. Start at the beginning of the array.
2. Find the minimum element in the unsorted part.

3. Swap it with the first unsorted element.
4. Move the boundary of the sorted part and repeat until all elements are sorted.

Time Complexity:

- Best Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$

Space Complexity: O (1)

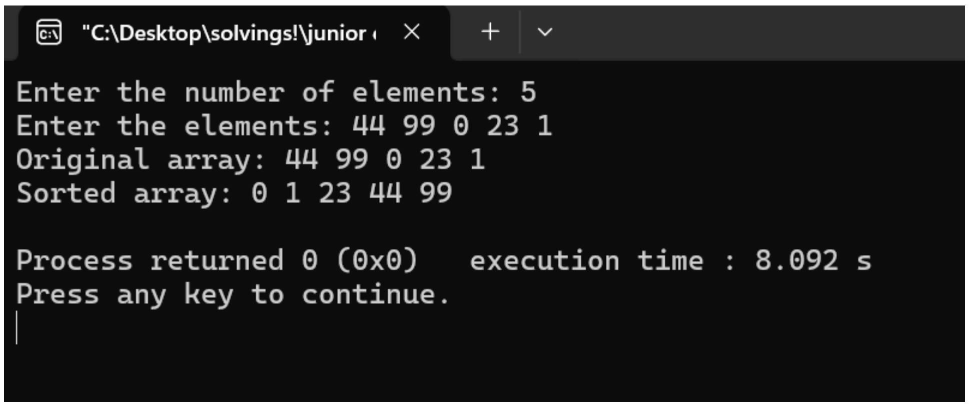
Merge Sort

Objective: Sort an array by dividing it into halves, sorting them recursively, and then merging them.

Code:

```
// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
  int n1 = mid - left + 1; // Size of left subarray
  int n2 = right - mid;   // Size of right subarray

  // Create temporary arrays
  int L[n1], R[n2];

  // Copy data to temporary arrays
  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];

  // Merge the temporary arrays back into arr[left..right]
  int i = 0, j = 0, k = left;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;  }
    k++;  }
  // Copy remaining elements of L[], if any
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;  }
  // Copy remaining elements of R[], if any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;  }}
// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2; // Find the midpoint

    // Recursively divide the array
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);

    // Merge the sorted halves
    merge(arr, left, mid, right);
  }}
```

Output:



```
Enter the number of elements: 5
Enter the elements: 44 99 0 23 1
Original array: 44 99 0 23 1
Sorted array: 0 1 23 44 99

Process returned 0 (0x0)   execution time : 8.092 s
Press any key to continue.
```

Description:

Merge Sort is a divide-and-conquer algorithm. It recursively divides the array into two halves, sorts each half, and then merges the sorted halves into a single sorted array.

1. Divide: Recursively split the array into two halves until each sub-array has only one element.
2. Conquer: Sort the two halves (the base case for recursion is an array with one element).
3. Combine: Merge the two sorted halves into one sorted array.

Time Complexity:

Merge Sort divides the array in half, resulting in logarithmic time complexity (log n levels) and performs linear-time merging at each level, resulting in a time complexity of O(n log n).

- Best Case: O(n log n)
- Worst Case: O(n log n)
- Average Case: O(n log n)

Space Complexity: O(n) – Merge Sort requires extra space for the temporary arrays used during the merge process.

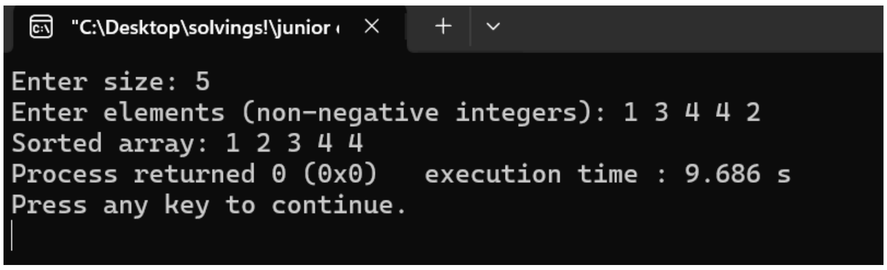
Counting Sort

Objective: Sort non-negative integers by counting occurrences and placing them in correct order.

Code:

```
vector<int> arr(n);  cout << "Enter elements (non-negative integers): ";
   int maxVal = 0;
   for(int i = 0; i < n; i++) {
      cin >> arr[i];
      maxVal = max(maxVal, arr[i]);      }
   vector<int> count(maxVal+1, 0);
   for(int i = 0; i < n; i++) count[arr[i]]++;
   cout << "Sorted array: ";
   for(int i = 0; i <= maxVal; i++)
      while(count[i]--) cout << i << " ";
```

Output:



```
Enter size: 5
Enter elements (non-negative integers): 1 3 4 4 2
Sorted array: 1 2 3 4 4
Process returned 0 (0x0)   execution time : 9.686 s
Press any key to continue.
```

Description:

AFRA C233456

Counting Sort works by counting the occurrences of each element in the array and then using these counts to place the elements in the correct position in the sorted array. It is efficient when the range of input elements is small. Note that, Counting Sort is not comparison-based and works well for integers or small-range data.

1. Count the occurrences of each element in the array.
2. Accumulate the counts to determine the position of each element.
3. Place the elements in their sorted positions.

Time Complexity: Where n is the number of elements and k is the range of input.

- Best Case: O(n + k)
- Worst Case: O(n + k)
- Average Case: O(n + k)

Space Complexity: O(k) – Additional space is needed for the count array, which has a size equal to the range of input values.

Quick Sort

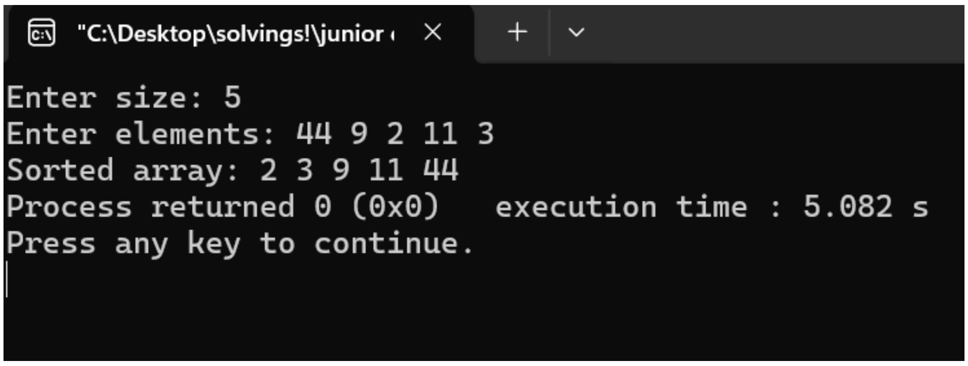
Objective: Sort elements using a divide-and-conquer approach by selecting a pivot and partitioning.

Code:

```
int partition(int arr[], int low, int high) {
 int pivot = arr[high], i = low-1;
```

```
  for(int j=low;j<high;j++) {
    if(arr[j]<pivot)
      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);
  }
} // call quickSort() in main
```

Output:



```
Enter size: 5
Enter elements: 44 9 2 11 3
Sorted array: 2 3 9 11 44
Process returned 0 (0x0)   execution time : 5.082 s
Press any key to continue.
```

Description:

Quick Sort is a divide-and-conquer algorithm that works by selecting a 'pivot' element from the array, partitioning the array into two parts (elements less than the pivot and elements greater than the pivot), and then recursively sorting the two partitions.

1. Select a pivot element from the array.
2. Partition the array such that all elements less than the pivot come before it, and all elements greater than the pivot come after it.
3. Recursively sort the two partitions (excluding the pivot, which is already in its final sorted position).

Note that, the efficiency of Quick Sort heavily depends on how balanced the partitions are, which in turn depends on the choice of pivot. To avoid worst-case scenarios (e.g., sorted or reverse-sorted arrays), pivot selection can be improved using strategies like: Randomized Pivot and Median-of-Three (picking the median of the first, middle, and last elements as pivot for more balanced partitions)

Time Complexity:

AFRA C233456

- Best Case: O(n log n) – The pivot divides the array into two nearly equal halves. Worst Case: O(n^2) – This occurs if the pivot is always the smallest or largest element, causing unbalanced partitions.
- Average Case: O(n log n) – With random pivot selection, the partitions tend to be balanced.

Space Complexity: Quick Sort is done in-place, with O(log n) space on average due to recursion; worst-case stack depth is O(n).

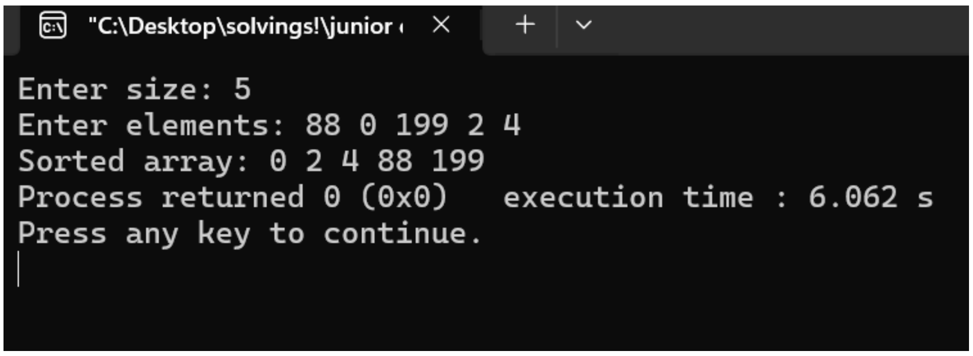
Heap Sort

Objective: Sort elements by building a heap data structure and extracting the maximum repeatedly.

Code:

```
 void heapify(int arr[], int n, int i) {
 int largest = i, l = 2i+1, r = 2i+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);
 }}
//call heapSort(arr,n) in main
```

Output:



```
Enter size: 5
Enter elements: 88 0 199 2 4
Sorted array: 0 2 4 88 199
Process returned 0 (0x0)   execution time : 6.062 s
Press any key to continue.
```

Description:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It first builds a max heap (or min heap), where the largest (or smallest) element is at the root. Then, it repeatedly extracts the root and rebuilds the heap until all elements are sorted.

Time Complexity:

Best Case: O(n log n) – Even if the array is partially sorted, Heap Sort takes O(n log n) due to the heapification process.

Worst Case: O(n log n) – The worst case still involves heapifying the array and performing n extractions.

Average Case: O(n log n) – Heap Sort performs consistently with logarithmic heap operations.

Space Complexity: O(1) – no additional space required beyond the input array

Radix Sort

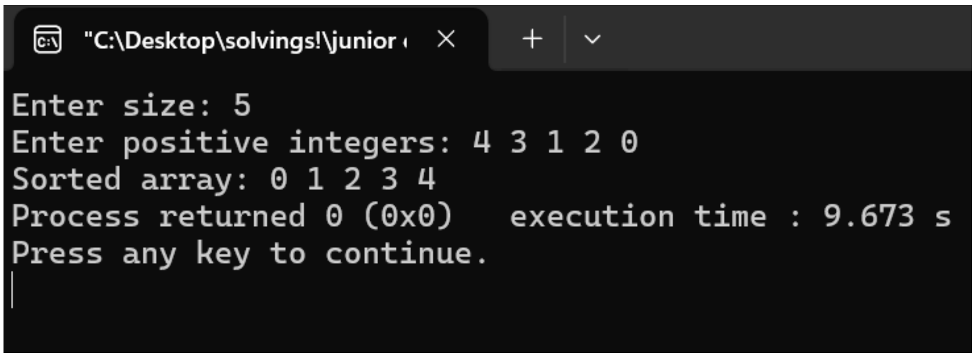
Objective:

To sort numbers by processing individual digits, starting from the least significant digit to the most significant, using a stable sorting algorithm like counting sort at each digit level.

Code:

```
 int getMax(int arr[], int n) {
  int mx = arr[0];
  for(int i=1;i<n;i++)
    if(arr[i]>mx) mx = arr[i];
  return mx;
}
void countingSort(int arr[], int n, int exp) {
  int output[n], count[10] = {};
  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 m = getMax(arr,n);
  for(int exp = 1; m/exp > 0; exp *= 10)
    countingSort(arr,n,exp);
}
```

Output:



```
Enter size: 5
Enter positive integers: 4 3 1 2 0
Sorted array: 0 1 2 3 4
Process returned 0 (0x0)   execution time : 9.673 s
Press any key to continue.
```

AFRA C233456

Description:

Radix Sort sorts numbers digit by digit, starting from the least significant digit (LSD) to the most significant. It uses a stable sorting algorithm (often Counting Sort) to sort the elements based on each digit.

1. Start with the least significant digit (LSD).
2. Sort the elements based on the current digit using a stable sorting algorithm (like Counting Sort).
3. Move to the next more significant digit and repeat until all digits are sorted.

   It's best suited for fixed-length integers or strings, and depends on the size of k.

Time Complexity: Where n is the number of elements and k is the number of digits.

- Best Case: O(nk)
- Worst Case: O(nk)
- Average Case: O(nk)

Space Complexity: O (n + k) – Radix Sort requires space for the counting array and the output array, where k is the number of possible digit values.