Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

Structuri de date si algoritmi

A efectuat: st. gr.FAF-212 Lupascu Felicia

A verificat: dr. conf. Univ S. Corlat

1. *Simple sorting algorithms*

1.1 Selection algorithm

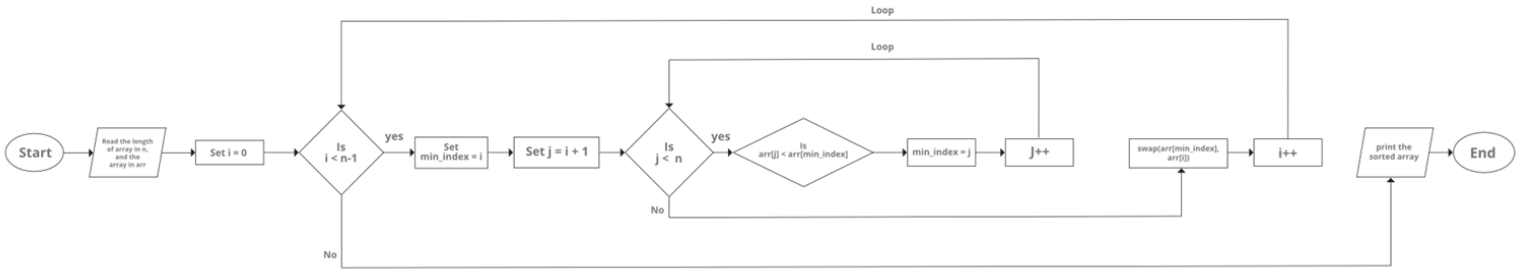
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```
arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```



Flowchart for Selection Sort

```cpp
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

***Time Complexity*:** $O(n^2)$ as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

1.2 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

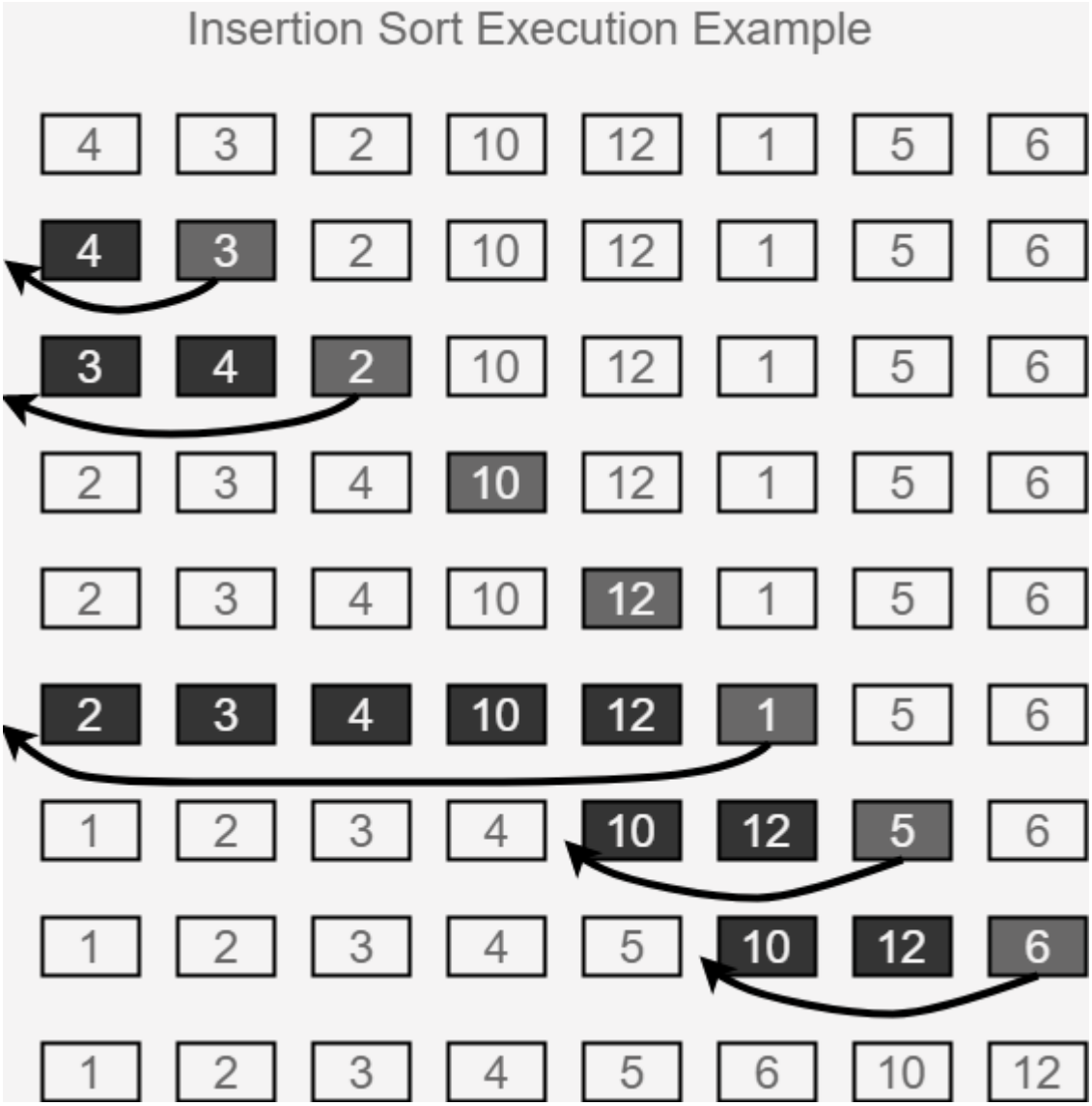
Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



```cpp
// C++ program for insertion sort
#include <bits/stdc++.h>
using namespace std;

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;

      /* Move elements of arr[0..i-1], that are
      greater than key, to one position ahead
      of their current position */
      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      cout << arr[i] << " ";
   cout << endl;
}

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

   insertionSort(arr, n);
   printArray(arr, n);

   return 0;
}
```

Output:

```
5 6 11 12 13
```

Time Complexity: O(n^2)
Auxiliary Space: O(1)
Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
Algorithmic Paradigm: Incremental Approach

1.3 Binary Insertion Sort

Binary insertion sort is a sorting algorithm similar to insertion sort, but instead of using linear search to find the position where the element should be inserted, we use binary search. Thus, we reduce the number of comparisons for inserting one element from O(N) to O(log N).

It is an adaptive algorithm, which means that it works faster when the given array is already substantially sorted, i.e., the current position of the element is near its actual position in the sorted array.

It is a stable sorting algorithm — the elements with the same values appear in the same order in the final array as they were in the initial array.

Binary Insertion Sort Example

Suppose we need to sort the following array:

1. We assume that the first element is already sorted.
2. We take the second element and store it in a variable (key).
3. Now, we use binary search to find the element on the left of the current element, which is just greater than it.
4. In this case, we have only one element, 8, and it is greater than 6. So, we shift 8 one index towards the right and place 6 at its position.

The array now looks like this:

5. Now, we take the third element, 1. Note that all the elements before the current element are sorted.
6. We store 1 in key and find the element just greater than 1 in the sorted part using binary search.
7. Here, the required element is 6. So, we shift 6 and 8 one index towards the right and place 1 at the position of 6 before shifting.

The array now looks like this:

8. We now take the 4th element, 5, and store it in key.
9. Using binary search, we find the element just greater than 5 in the sorted part. In this case, the required element is 6.

10. Again, we shift 6 and 8 one index towards the right and place 5 at the position of 6 before shifting.

The array now looks like this:

11. We now take the last (5th) element, which is 3, and find the element just greater than it in the sorted part.
12. The required element is 5. We shift 5, 6, and 8 one index towards the right and place 3 at the position of 5 before shifting.

The resulting array is:

We have sorted the given array using binary insertion sort.

Binary Insertion Sort Algorithm

Binary insertion sort for array A:

- **Step 1:** Iterate the array from the second element to the last element.
- **Step 2:** Store the current element A[i] in a variable key.
- **Step 3:** Find the position of the element just greater than A[i] in the subarray from A[0] to A[i-1] using binary search. Say this element is at index pos.
- **Step 4:** Shift all the elements from index pos to i-1 towards the right.
- **Step 5:** A[pos] = key.

```
#include <iostream>
using namespace std;
int binarySearch(int arr[], int item, int low, int high) {
   if (high <= low)
      return (item > arr[low])? (low + 1): low;
      int mid = (low + high)/2;
   if(item == arr[mid])
      return mid+1;
   if(item > arr[mid])
      return binarySearch(arr, item, mid+1, high);
      return binarySearch(arr, item, low, mid-1);
}
void BinaryInsertionSort(int arr[], int n) {
   int i, loc, j, k, selected;
   for (i = 1; i < n; ++i) {
      j = i - 1;
      selected = arr[i];
      loc = binarySearch(arr, selected, 0, j);
      while (j >= loc) {
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1] = selected;
   }
```

```
}
int main() {
   int arr[] = {8, 6, 1, 5, 3};
   int n = sizeof(arr)/sizeof(arr[0]), i;
   BinaryInsertionSort(arr, n);
      cout<<"Sorted array is : \n";
   for (i = 0; i < n; i++)
      cout<<arr[i]<<"\t";
   return 0;
}
```

```
Sorted array is :
1   3   5   6   8
```

2. *Exotic sorting algorithms*

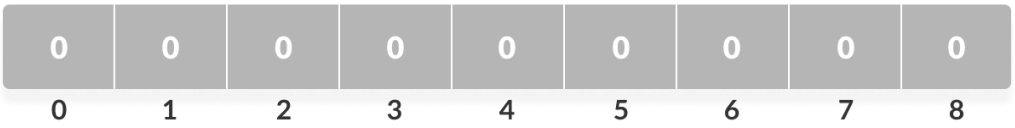
2.1 Counting sort

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Working of Counting Sort

1. Find out the maximum element (let it be `max`) from the given array.

2. Initialize an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array.

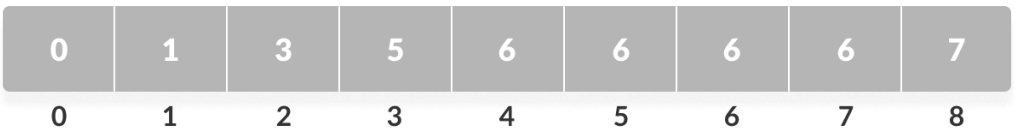


Store the count of each element at their respective index in the `count` array

3. For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of the count array. If element "5" is not present in the array, then 0 is stored in the 5th position.

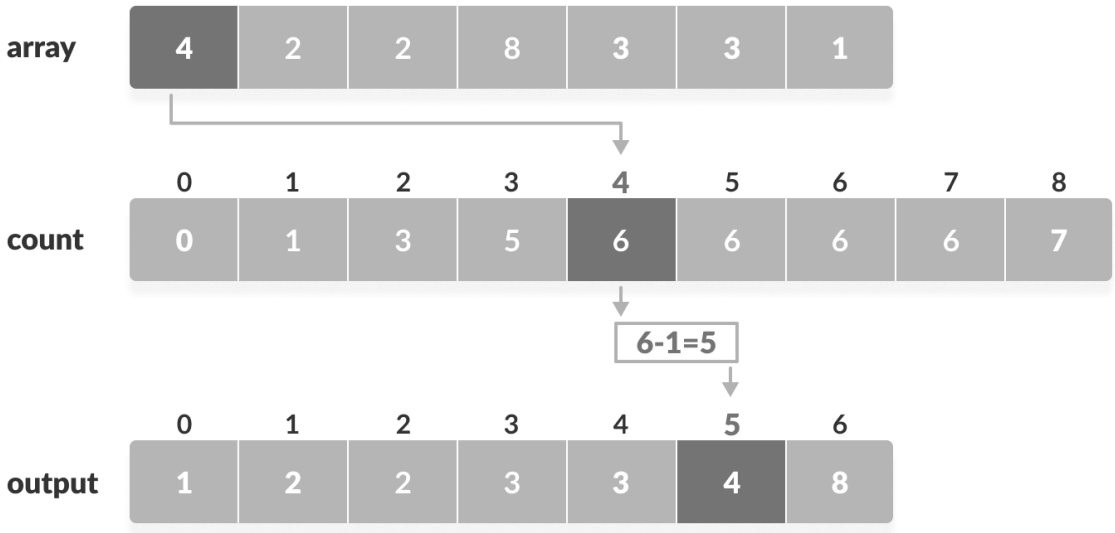
Count of each element stored

4. Store the cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



Cumulative count

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in the figure below.



6. After placing each element at its correct position, decrease its count by one.

Counting Sort Algorithm

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

```
// Counting sort in C++ programming

#include <iostream>
using namespace std;
void countSort(int array[], int size) {
  // The size of count must be at least the (max+1) but
  // we cannot assign declare it as int count(max+1) in C++ as
  // it does not support dynamic memory allocation.
  // So, its size is provided statically.
  int output[10];
  int count[10];
  int max = array[0];
  // Find the largest element of the array
  for (int i = 1; i < size; i++) {
    if (array[i] > max)
      max = array[i];
  }
  // Initialize count array with all zeros.
  for (int i = 0; i <= max; ++i) {
    count[i] = 0;
  }
  // Store the count of each element
  for (int i = 0; i < size; i++) {
    count[array[i]]++;
  }
  // Store the cummulative count of each array
  for (int i = 1; i <= max; i++) {
    count[i] += count[i - 1];
```

```
  }
  // Find the index of each element of the original array in count array, and
  // place the elements in output array
  for (int i = size - 1; i >= 0; i--) {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
  }
  // Copy the sorted elements into original array
  for (int i = 0; i < size; i++) {
    array[i] = output[i];
  }
}
// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
// Driver code
int main() {
  int array[] = {4, 2, 2, 8, 3, 3, 1};
  int n = sizeof(array) / sizeof(array[0]);
  countSort(array, n);
  printArray(array, n);
}
```

Complexity

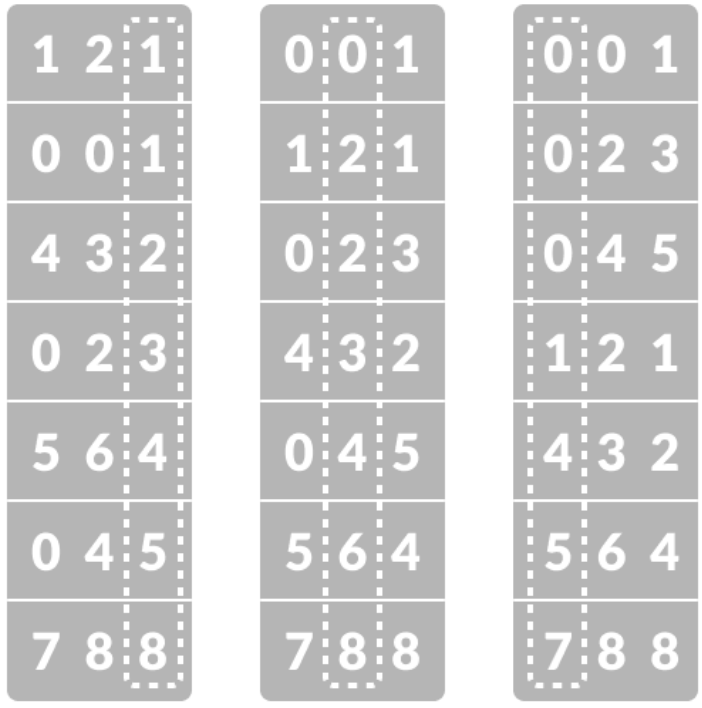
Time Complexity	
Best	O(n+k)
Worst	O(n+k)
Average	O(n+k)
Space Complexity	O(max)
Stability	Yes

2.2 Radix Sort

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same **place value**. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.



sorting the integers according to units, tens and hundreds place digits

Working of Radix Sort

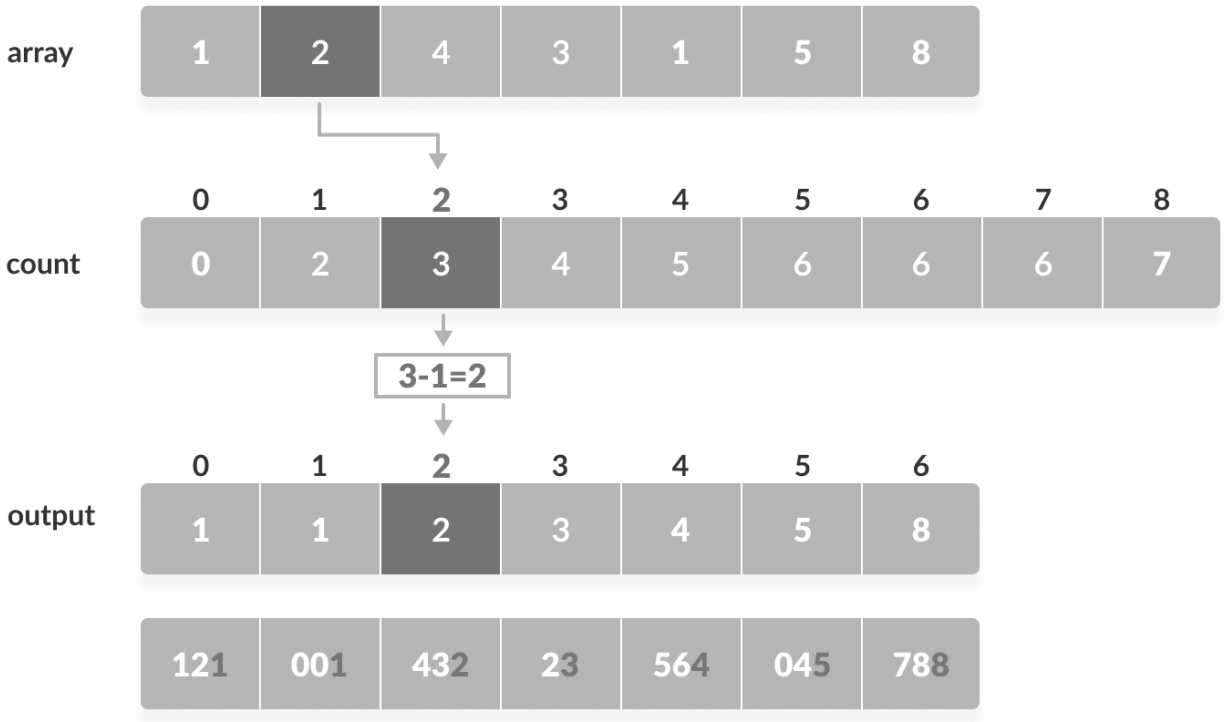
1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.

In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds of places (3 times).

2. Now, go through each significant place one by one.

   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.

   Sort the elements based on the unit place digits (X=0).



3. Now, sort the elements based on digits at tens place.

4. Finally, sort the elements based on the digits in hundreds of places.

Radix Sort Algorithm

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

```
// Radix Sort in C++ Programming
#include <iostream>
using namespace std;
// Function to get the largest element from an array
int getMax(int array[], int n) {
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}
// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
  const int max = 10;
  int output[size];
  int count[max];
  for (int i = 0; i < max; ++i)
    count[i] = 0;
  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;
```

```
  // Calculate cumulative count
  for (int i = 1; i < max; i++)
    count[i] += count[i - 1];
  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--) {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }
  for (int i = 0; i < size; i++)
    array[i] = output[i];
}
// Main function to implement radix sort
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);
  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
// Print an array
void printArray(int array[], int size) {
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
// Driver code
int main() {
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

Time Complexity

Best	O(n+k)
Worst	O(n+k)