Data Structure

2. Searching & Sorting

Handwritten by pankaj kumar

Searching & Sorting

Date

Page No. 121

(1) Linear Search:

Given an array of n element, write a function to search a given element x in array.

ex:- I/P: arr[] = {10, 20, 80, 30, 60, 50, 110, 130, 170}, x = 110
O/P: 6 (index)

I/P: arr[] = {10, 20, 80, 30, 60, 50, 110, 130, 170}, x = 175
O/P:- -1 // x is not present in array

Approach: start from left most element of arr[] and one by one compare x with each element of arr[].
- If x match return index else return -1.

Program:
```
int linear_search (int arr[], int n, int x){
        for(int i=0; i<n; i++)
            if (arr[i]==x)
                return i;
        return -1;
}
```

Time complexity :- O(n)

(2) Binary Search:

Given a sorted array of n elements, write a function to search a given element x in array.

Binary-Search: Search in a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the middle element of interval, narrow the interval to lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Recursive Implementation:
```
int binary_search (int arr[], int l, int r, int x){
        if (r<l) { return -1; }
        int mid = l+(r-l)/2;
        if (arr[mid]==x) return mid;
```

```
else if (arr[mid] > x)
        r = mid-1;
else
        l = mid+1;
return binary_search(arr, l, r, x);
}
```

Iterative implementation:

```
int binary_search(int arr[], int l, int r, int x){
    while (l<=r){
        int mid = l+(r-l)/2;
        if (arr[mid]==x) return mid;
        if (arr[mid]>x) r=mid-1;
        else l=mid+1;
    }
    return -1;
}
```

Time complexity: $O(\log n)$ | Space: $O(1)$

$T(n) = T(n/2) + c \Rightarrow O(\log n)$ | Algorithm: Divide & conquer.

(3) Jump Search:

- like binary search jump search is a searching algorithm for sorted array. the basic idea is to search fewer element by fixed step or skipping some element in place of searching all element. when we find the interval, we perform a linear search operation from the one step back.

Ex:- arr = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610].
(indices 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

length of array is 16. Find 55.

Soln:- let jump step is 4.

① Jump from 0 to 4;
② Jump from 4 to 8;
③ Jump from 8 to 12;

∵ index 12 is greater than 55 start linear search from 8 index.

* optimal block size to skipped ?
- let m is block size then time complexity will be (n/m) and for linear search $(2_m + m - 1)$ in worst case
- The minm value of function $(2_m + m - 1)$ will be if $m = \lceil n \rceil$

* Program:

```
int JumpSearch (int arr[], int x, int n) {
        int step = sqrt(n);
        int prev = 0;
        while (arr[min (step, n) - 1] < x) {
                prev = step;
                step += sqrt (n);
                if (prev >= n) return -1;
        }
        while (arr[prev] < x) {
                prev++;
                if (prev == min (step, n)) return -1;
        }
        if (arr[prev] == x) return prev;
        return -1;
}
```

Time Complexity: $O(\sqrt{n})$ between linear and binary-search.

(4.) Interpolation Search:

- The interpolation search is an improvement over Binary search for instances, where the value of sorted array are uniformly distributed. Binary search will always goes to middle element
- on the other hand interpolation search may go to different location according to the value of key element (near to key element)

* Find that location.

$$pos = l_0 + [(x - arr[l_0]) * (hi - l_0) / (arr[hi] - arr[l_0])]$$

Derivation :-

- Assume array are linearly distributed.

- General equation of line: $y = m * x + c$
- y is the value of array and x is the index.

∴ No putting value of lo, hi and X in the equation

$arr[hi] = m * hi + c$ —— ①

$arr[lo] = m * lo + c$ —— ②

$x = m * pos + c$ —— ③

after solving these three equation we get,

- $pos = lo + (x - arr[lo]) * (hi - lo) / (arr[hi] - arr[lo])$

Program:

```
int interpolation_search (int arr[], int n, int x){
        int lo=0, hi=n-1;
        while(lo<=hi && x>=arr[lo] && x<=arr[hi]){
            if (lo==hi){
                if(arr[lo]==x) return lo;
                return -1;
            }
            int pos=lo+(((double)(hi-lo)/(arr[hi]-arr[lo]))
                        *(x-arr[lo]));
            if (arr[pos]==x) return pos;
            if(arr[pos]<x) lo=pos+1;
            else hi=pos-1;
        }
        return -1;
}
```

(5) Exponential search:

- Given a sorted array and an element x tobe searched
- Ex:- arr[] = {10, 20, 40, 45, 55} ; X = 45

  O/P:- 3

- Approach:

① Find Range where element is present

② Do Binary search in above found Range.

Find range:-
- The idea is to start with subarray size 1, compare its last element to x, then try size 2, then 4 and so.. on. until last element of subarray is not greater.
- once we find an index i (after repeated doubling of i), then we know that element must be present b/w i/2 and i

Program:-

```
int exponential_search (int arr[], int n, int x){
        if (arr[0] == x) return 0;
        int i = 1;
        while (i<n && arr[i] <= x)
                i = i*2;
        return binary_search (arr, i/2, min(i, n-1), x);
}
```

Time Complexity: O(log n)

Space: O(1)

Application :-
- It is particularly useful for unbounded searches, where size of array is infinite.
- It work better than binary search for bounded arrays, and also when the element to be searched closer to first element.

Sorting

(6) Selection Sort:
- The selection sort algorithm sorts an array by repeatedly finding the minimum element (ascending order) from unsorted array part and putting it at the beginning.
- On every iteration, selection sort select the min^m element & put it into the sorted part of array

Ex:- arr[] = 64 25 12 22 11
min element = 11 → 1st iteration
11 25 12 22 64
2nd min element = 12 → 2nd iteration
11 12 25 22 64
11 12 22 25 64
11 12 22 25 64 → sorted.

Program:

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

Time: $O(n^2)$ | Space: $O(1)$

Note:- the good thing is selection not makes more than $O(n)$ swaps and can be useful when memory write is costly.

*Stability:

- The selection sort is not stable
- A sorting algorithm is said to be stable if two objects with equal or same keys appear in the same order in sorted output.

ex:- I/P: 4_A 5 3 2 4_B 1
O/P: 1 2 3 4_A 4_B 5 → stable

- selection sort can be made stable if instead of swapping the minm element, push the every element from their position of inserting to that element one step ahead

ex:- 1 2 5 6 7 3 4

(11)

(7) Bubble Sort:-

- It work on the approach that peak a greatest element from array & put into the end.
- It is done by repeatedly swapping the adjacent element if they are in wrong order.

ex:-

First Pass

(5 1 4 2 8) → (1 5 4 2 8), compare 1 & 5 ∵ 5>1 swap.

(1 5 4 2 8) → (1 4 5 2 8), compare 5 & 4 ∵ 5>4 swap.

(1 4 5 2 8) → (1 4 2 5 8), compare 5 & 2 ∵ 5>2 swap.

(1 4 2 5 8) → (1 4 2 5 8), compare 5 & 8 ∵ 5<8 don't swap

- In First pass the largest element is on the end.
- Now we will do 2nd pass in which we compare till end−1.

2nd pass

(1 4 2 5 8) → (1 4 2 5 8), ∵ 1<4 don't swap

(1 4 2 5 8) → (1 2 4 5 8) ∵ 4>2 swap

(1 2 4 5 8) → (1 2 4 5 8) ∵ 4<5 don't swap.

Now in second pass 2nd greatest element is at idx 2 position from last.

Note:- If we will do 3rd pass then there is no any swap so we can break the loop bcz array is sorted now.

Program:-

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

- Worst & Average Case Time: O(n*n). Worst case occur when array is reverse sorted.
- Best case time: O(n) : Best case occur when array is sorted
- Space : O(1)
- Stable : yes
- In place sorting : yes

* Application :. In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost sorted array and fix it with linear time (2n). (Polygon Filling Algo)

(8) Insertion Sort :-

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

* Approach:

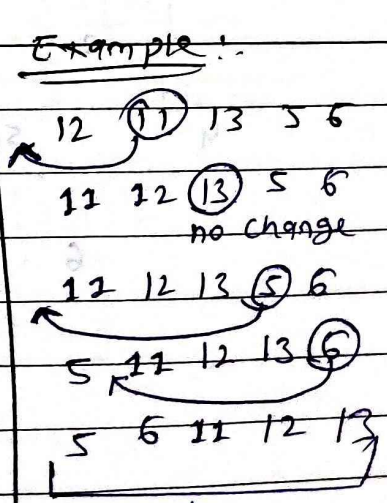
- pick a element and compare it to predecessor, if it is smaller than predecessor. move the greater elements one up and compare with again predecessor. untill the exact position found. (by swapping)

* Code

```
void insertion_sort(int arr[], int n){
    int i, key, j;
    for(i=1; i<n; i++){
        key = arr[i];
        j = i-1;
        while(j>=0 && arr[j]>key){
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Time : O(n^2)
Space : O(1)

Example :-



Boundary cases: It takes maxm time if element sorted in reverse
. It takes min time O(n) when element are already sorted
sorting in place: Yes
Stable: yes
Uses:- when input array is almost sorted, only few element are misplaced in complete big array.

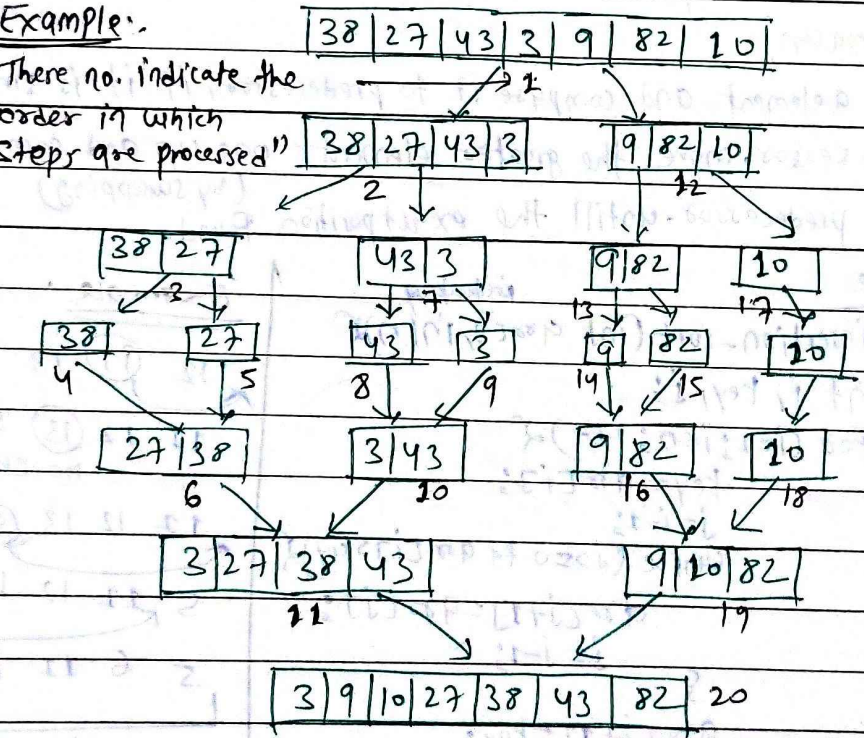
Binary insertion sort

- we can use Binary search to find the proper location to insert the selected item at each iteration.
- But as a whole, still has a running worst case $O(n^2)$ bcz of the series of swaps required for each iteration.

9. Merge Sort:-

- It work on Divide and conquer Algorithm. it divides the input array into two halves, call itself for the two halves, and then merge the two sorted halves.

Example:-



Note:- the array is recursively divided into two halves till the size becomes 1. once the size becomes 1, the merge processes come and start merging back into sorted order till complete array is mersed.

Program:

```
void merge (int *Arr, int start, int mid, int end) {
        int temp[end-start+1];
        int i=start, j=mid+1, k=0;
        while (i<=mid && j<=end) {
                if (Arr[i] <= Arr[j]) {
                        temp[k] = Arr[i];
                        i+=1;
                }
                else {
                        temp[k] = Arr[j];
                        j+=1;
                }
                k+=1
        }
        while (i<=mid) {
                temp[k] = Arr[i]
                k+=1; i+=1;
        }
        while (j<=end) {
                temp[k] = Arr[j];
                k+=1; j+=1;
        }
        //copy temp to original Array interval
        for (i=start; i<=end; i+=1) {
                Arr[i] = temp[i-start];
        }
}

start → void mergsort (int * Arr, int start, int end) {
        if (start < end) {
                int mid = (start+end)/2;
                mergeSort (Arr, start, mid);
                mergesort (Arr, mid+1, end);
                merge (Arr, start, mid, end);
        }
}
```

Time complexity: $\theta(n \log n)$:

- Recurrence relation $\Rightarrow T(n) = 2T(n/2) + \theta(n)$
- In All cases (worst, average and best) time is $\theta(n \log n)$ bcz it always divide the array into two halves and take linear time to merge two halves.

Application:

- merge sort is useful for sorting linked list in $O(n \log n)$.
- Inversion count problem
- Used in External sorting (a class of sorting algorithm that can handle massive amount of data). when the data being sorted is not fit into the memory.

Drawbacks of mergesort:

- Slower compare to other sorting algorithm for smaller tasks.
- Requires an additional memory space of $O(n)$
- It goes through the whole process even if array is sorted.

(20) Quick Sort:

- It is divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. after partition the pivot element is at their correct position in sorted array and all element smaller than pivot is before pivot (left side) and larger than pivot is after pivot (right side)

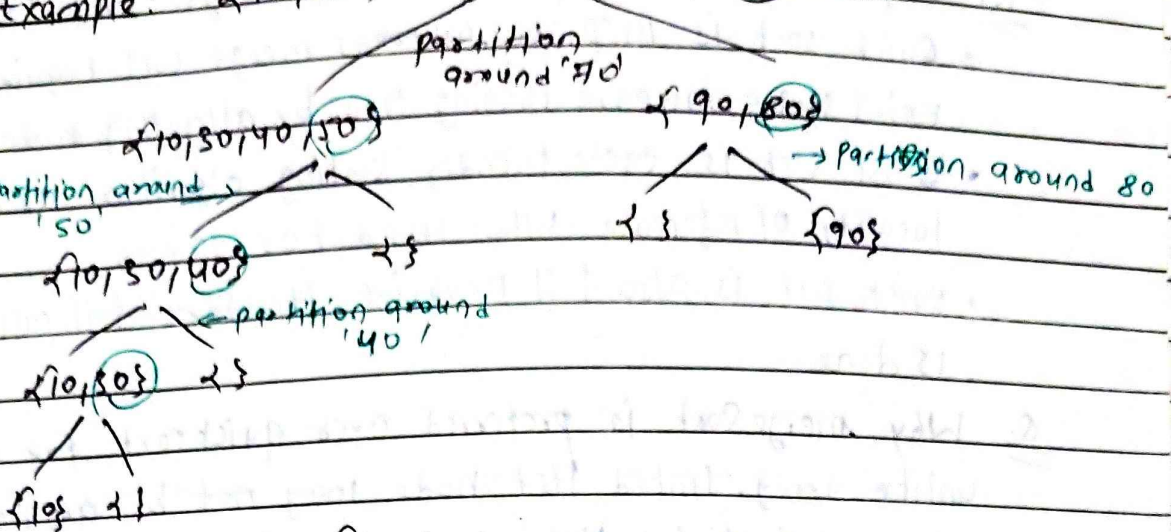
ex:- 10 80 30 90 40 50 (70) ← pick as pivot

↳ { 10, 30, 40, (50), 90, 80 } after partition

smaller greater

- Different ways to pick pivot
  1. always pick first element as pivot
  2. Always pick last element as pivot
  3. pick a random element as pivot
  4. pick median as pivot.

Example: {10, 80, 30, 90, 40, 50, 70}



result array = {10, 30, 40, 50, 70, 80, 90}

Program:-

```
int partition (int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high-1; j++) {
                if (arr[j] < 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 < high) {
                int pi = partition (arr, low, high);
                quicksort (arr, low, pi-1);
                quicksort (arr, pi+1, high);
        }
}
```

Time complexity

- Best case: $\Theta(n \log n)$:- when pivot is middle element always
- Average case: $O(n \log n)$:- when put $O(n/9)$ in one partition and $O(9n/10)$ in other.
- Worst case: $\Theta(n^2)$:- when pivot is always greatest or smallest element

Stable: NO

Q. Why Quick Sort is preferred over merge sort for arrays.

- Quick sort is in place. whereas merge sort requires O(n) extra space, increase running time by allocating & deallocating
- Quick sort is cache friendly sorting algorithm as it good locality of reference when used for array
- Quick sort is also tail recursive, therefore tail call optimization is done

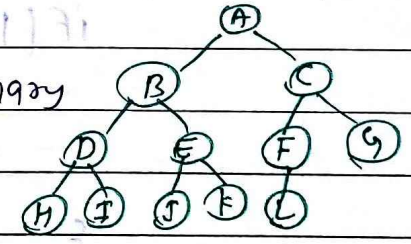
Q. Why mergeSort is preferred over quicksort for linked list?
Unlike array, linked list node may not be adjacent in memory. in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge sort can be implemented without extra space.

11. Heap Sort:-

- Heap sort is a comparison-based sorting technique based on Binary heap data structure. It is similar to selection sort, where we first find the minimum element and place the minm element at beginning. repeat same process for remaining element.

Binary Heap:

- A complete binary tree is a binary tree in which every level, except last possibly is completely filled, and all nodes are as far left as possible.
- A Binary Heap is a complete Binary tree where items are stored in a special order such that the value in a parent node is greater (or smaller) than the nodes in its two children nodes. (max heap, min heap). The heap can be represented by a binary tree or array.



Array based representation for binary Heap
if parent node is at index i, then left = 2*i+1 index and Right = 2*i+2 index

Heap sort algorithm for sorting in increasing order:-

① Build a max heap from the input data

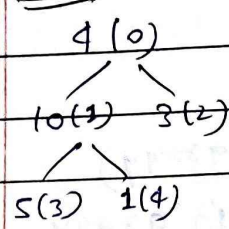
② At this point, the largest item is stored at the root. Replace it with the last item of the heap and reduce size by 1.
- Finally heapify the root of the tree

③ Repeat Step2 while size of heap is greater than 1.

Build the Heap

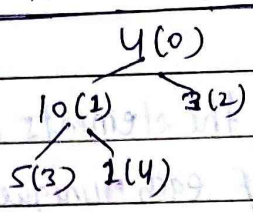
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

ex:- IP: 4, 10, 3, 5, 1

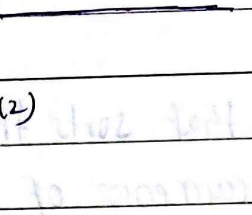


[Number in bracket are array index]

- Apply heapify to index 1.



- Apply heapify to index 0.



Note :- The heapify procedure calls itself recursively to build heap in top down manner.

program :-

```
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--) // Build heap rearrange
        heapify(arr,n,i);       // array
    for(int i=n-1; i>0; i--){ // one by one extract
        swap(arr[0], arr[i]); // element from heap
        heapify(arr,i,0);
    }
}
```

Time complexity: O(n log n)

O(log n) → for heapify & O(n) → for create & build heap()

In-place: yes

Stable: NO

Application:

- sort a nearly sorted (or k sorted) array)
- k largest (or smallest) elements in array.

(12) Counting Sort:

It is a sorting algorithm that sorts the elements of an array by counting th no. of occurrence 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.

Ex:- arr = [4|2|2|8|3|3|1]

① Find out maximum element in the array i.e 8

② initialize an array of length max+1 with all element '0'

	0	1	2	3	4	5	6	7	8
count =	0	0	0	0	0	0	0	0	0

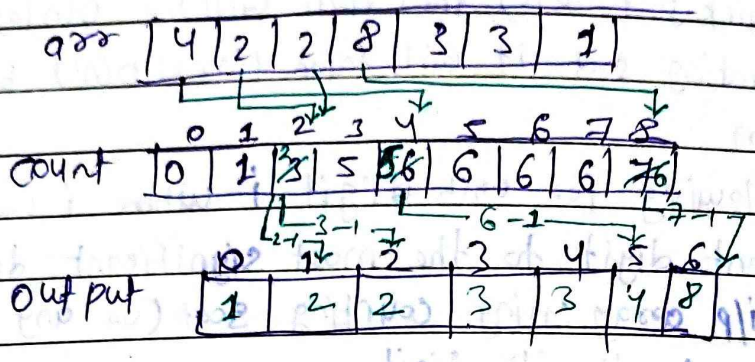
③ Store the count of each element at their respective index in new array

	0	1	2	3	4	5	6	7	8
	0	1	2	2	1	0	0	0	1

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

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

(5) Find the index of each element of the original array in the count array. and decrease the count by one.



Program:

```
Void countSort(int arr[], int size){
    int output[size];
    int max = arr[0];
    for(int i=1; i<size; i++)      // Find max element
        if(arr[i] > max) max = arr[i];
    int count[max+1];  // create count with max+1
    for(int i=0; i<=max; ++i) // Initialize count to zero
        count[i] = 0;
    for(int i=0; i<size; i++)  // store count of each element
        count[arr[i]]++;
    for(int i=1; i<=max; i++) // store cumulative count
        count[i] += count[i-1];
    for(int i=size-1; i>=0; i--){
        output[count[arr[i]]-1] = arr[i];
        count[arr[i]]--;
    }
    for(int i=0; i<size; i++){ arr[i] = output[i]; }
}
```

Time: Best = Worst = average = $O(n+k)$ (no of element, maxm value, range of I/P)

Space: O(max)

Stable: yes

Use when there are small values with multiple count

(13) Radix Sort:

- If the element are in the range from 1 to n^2. then we use radix sort. we can sort such an array in linear time.
- If we will use comparison based sorting algorithm (merge, Heap, Quick-sort.. etc) then time will be $O(n \log n)$
- with counting sort it will take ~~linear~~ $O(n^2)$ bcz range is 'n^2'

Algorithm

1. Do following for each digit i where i varies from least significant digit to the most significant digit.
   - sort i/p array using counting sort (or any stable sort) according to the ith digit.

ex:- 170, 45, 75, 90, 802 24, 2, 66

- sorting by least significant digit (1s place) gives:
  170, 90, 802, 2, 24, 45, 75, 66
- Sorting by next digit (10s place) gives.
  802, 2, 24, 45, 66, 170, 75, 90
- sorting by the most significant digit (100s place) gives
  2, 24, 66, 75, 90, 170, 802

Time: $O(d * (n+b))$, b=base for representing numbers.
d = digit in input integers. (of maxm no.)

- if k is maxm value then $d = \log_b(k)$ i.e total time $= O(\log_b k * (n+b))$
- so to make this linear we set b as 'n'. i.e if the no. are represent in base n (or every digit take $\log_2(n)$ bits).

Program:

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

```
for (i=1; i<10; i++)
    count[i] += count[i-1];
for (i=n-1; i>=0; i--) {
    output[count[(arr[i]/exp)%10]-1] = arr[i];
    count[(arr[i]/exp)%10]--;
}
for (i=0; i<n; i++) arr[i] = output[i];
}
void radixsort(int arr[], int n){
    int maxm = arr[0];
    for (int i=1; i<n; i++)
        if (arr[i] > maxm) maxm = arr[i];
    for (int exp=1; maxm/exp; exp*=10)
        countSort(arr, n, exp);
}
```

(14) Bucket Sort:

- Bucket sort is mainly useful when input is uniformaly distributed over a range. for ex:-
- sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformaly distributed across the range.
- Here we cannot use counting sort bcz index is integer in counting sort.

***Algorithm:**

1. create n empty buckets (or lists).
2. Do following for every element arr[i].
   a. Insert arr[i] into bucket [n*arr[i]]
3. Sort individual bucket using insertion sort
4. concatenate all sorted buckets.

	0	1	2	3	4	5	6	7	8	9
Input	0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

	0	1	2	3	4	5	6	7	8	9
Bucket	/	0.12 0.17	0.4 0.23 0.26	0.39	/	/	0.68	0.72 0.78	/	0.94

	0	1	2	3	4	5	6	7	8	9
Output	0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94

* Program

```
void bucketSort(float arr[], int n){
    vector<float> b[n];
    for(int i=0; i<n; i++){
        int bi = n*arr[i];
        b[bi].push_back(arr[i]);
    }
    for(int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    int index=0;
    for(int i=0; i<n; i++)
        for(int j=0; j<b[i].size(); j++)
            arr[index++] = b[i][j];
}
```

Time complexity: if we assume that insertion in bucket takes O(1) time then steps 1 and 2 of the algo clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (but in our program we use vector for simplicity) step 4 also takes O(n) time as there will be 'n' item in all bucket. The main step is 3. this step also takes O(n) time on average if all numbers are uniformaly distributed.

* Bucket sort if number having integer part.

1. Find maxm and minm element of array
2. Calculate the range of each bucket
   range = (max-min)/n (n is no of buckets)
3. Create n buckets of calculated range.
4. Scatter the array elements to there buckets
   BucketIndex = (arr[i]-min)/range
5. Now sort each bucket individually.
6. Gather the sorted element from bucket to original array

Date

Page No. 141.

Ex:- Input

9.8	0.6	10.1	1.9	3.07	3.04	5.0	8.0	4.8	7.68

↓ scatter in bucket

unsorted bucket

0.6	1.9	3.07	3.04	5.0	4.8	8.0	7.68	9.8	10.1

↓ sort bucket

sorted bucket

0.6	1.9	3.04	3.07	4.8	5.0	7.68	8.0	9.8	10.1

output Array

0.6	1.9	3.04	3.07	4.8	5.0	7.68	8.0	9.8	10.1