Lecture-30

Radix Sort

1) Do following for each digit i where i varies from least significant digit to the most significant digit.
…………**a)** Sort input array using counting sort (or any stable sort) according to the i'th digit.

Example:
Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

What is the running time of Radix Sort?
Let there be d digits in input integers. Radix Sort takes $O(d*(n+b))$ time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let $k <= n^c$ where c is a constant. In that case, the complexity becomes $O(n\text{Log}_b(n))$. But it still doesn't beat comparison based sorting algorithms.

Linear Search

Linear search is to check each element one by one in sequence. The following method linearSearch() searches a target in an array and returns the index of the target; if not found, it returns -1, which indicates an invalid index.

```
int linearSearch(int arr[], int target)
{
   for (int i = 0; i < arr.length; i++)
   {
      if (arr[i] == target)
         return i;
   }
   return -1;
}
```

Linear search loops through each element in the array; each loop body takes constant time. Therefore, it runs in linear time O(n).

<u>Lecture-31</u>

Binary Search

For sorted arrays, *binary search* is more efficient than linear search. The process starts from the middle of the input array:

- If the target equals the element in the middle, return its index.
- If the target is larger than the element in the middle, search the right half.
- If the target is smaller, search the left half.

In the following binarySearch() method, the two index variables first and last indicates the searching boundary at each round.

```
int binarySearch(int arr[], int target)
{
   int first = 0, last = arr.length - 1;

   while (first <= last)
   {
      int mid = (first + last) / 2;
      if (target == arr[mid])
         return mid;
      if (target > arr[mid])
         first = mid + 1;
      else
         last = mid - 1;
   }
   return -1;
}
```

```
arr: {3, 9, 10, 27, 38, 43, 82}

target: 10
first: 0, last: 6, mid: 3, arr[mid]: 27   --  go left
first: 0, last: 2, mid: 1, arr[mid]: 9    --  go right
first: 2, last: 2, mid: 2, arr[mid]: 10   --  found

target: 40
first: 0, last: 6, mid: 3, arr[mid]: 27   --  go right
first: 4, last: 6, mid: 5, arr[mid]: 43   --  go left
first: 4, last: 4, mid: 4, arr[mid]: 38   --  go right
first: 5, last: 4                         --  not found
```

Binary search divides the array in the middle at each round of the loop. Suppose the array has length n and the loop runs in t rounds, then we have n * (1/2)^t = 1 since at each round the array length is divided by 2. Thus t = log(n). At each round, the loop body takes constant time. Therefore, binary search runs in logarithmic time O(log n).

The following code implements binary search using recursion. To call the method, we need provide with the boundary indexes, for example, binarySearch(arr, 0, arr.length - 1, target);

```

binarySearch(int arr[], int first, int last, int target)
{
    if (first > last)
        return -1;

    int mid = (first + last) / 2;

    if (target == arr[mid])
        return mid;
    if (target > arr[mid])
        return binarySearch(arr, mid + 1, last, target);
    // target < arr[mid]
    return binarySearch(arr, first, mid - 1, target);
}
```