Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)

(Religious Jain Minority)

Department of Information Technology

Chapter-5
Sorting and Searching

Introduction to Sorting: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Shell Sort, Radix sort. Analysis of Sorting Techniques. Comparison of sorting Techniques Introduction to Searching: Linear search, Binary search, Hashing Techniques, Different Hash functions, Collision& Collision resolution techniques, Analysis of searching Techniques.

Unit 5
Sorting and Searching

Introduction to Sorting Techniques

A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array. There are many factors to consider when choosing a sorting algorithm to use.

1. Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

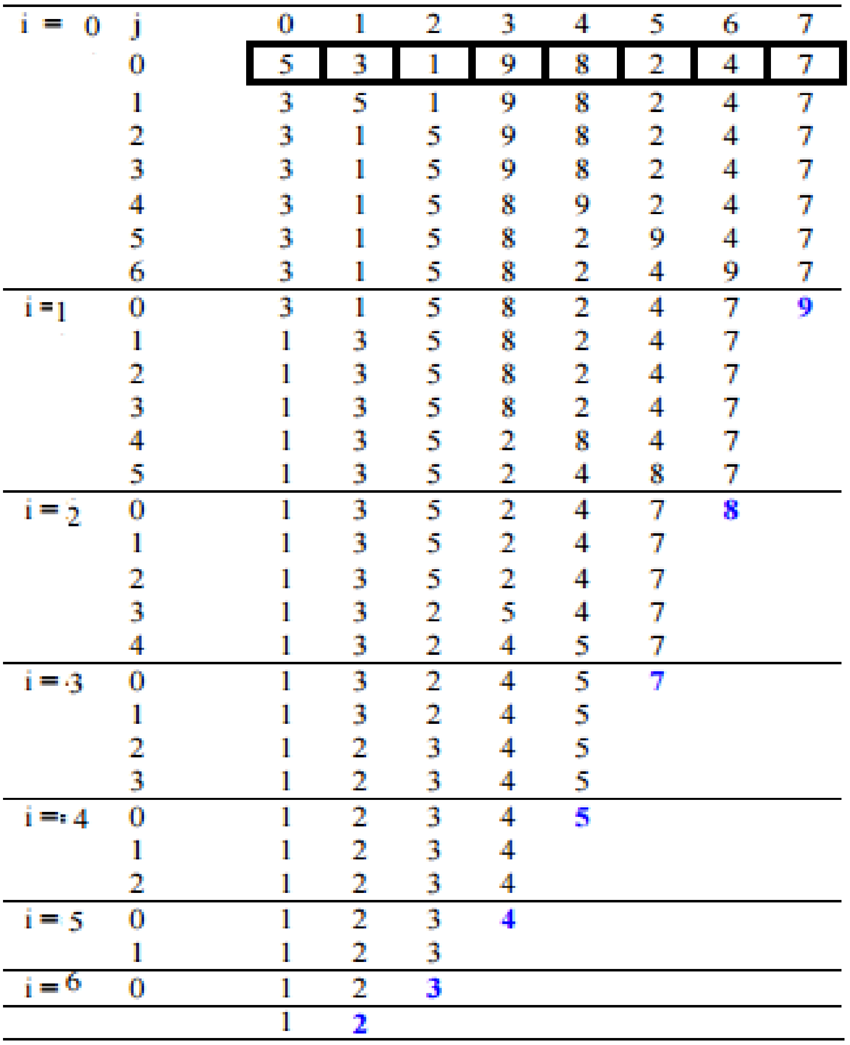
(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

```
// C program for implementation of Bubble sort
#include <stdio.h>

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

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

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

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
Sorted array:
11 12 22 25 34 64 90
```



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

2. Selection Sort

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

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

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

Following example explains the above steps:

```
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
```

```
// C program for implementation of selection sort
#include <stdio.h>

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++)
        printf("%d ", arr[i]);
    printf("\n");
}

// 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);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
Sorted array:
11 12 22 25 64
```

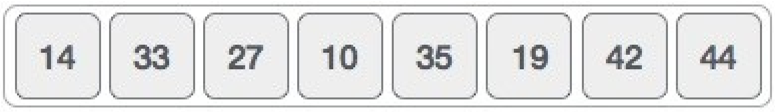
Time Complexity: O(n^2) as there are two nested loops.

<u>3. Insertion Sort</u>

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

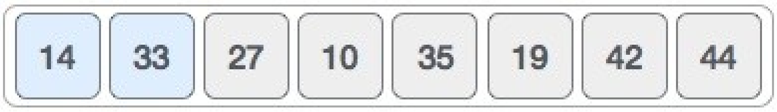
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

We take an unsorted array for our example.



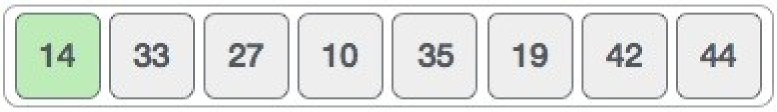
14	33	27	10	35	19	42	44

Insertion sort compares the first two elements.



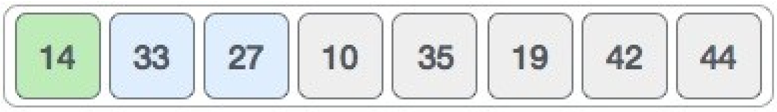
14	33	27	10	35	19	42	44

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



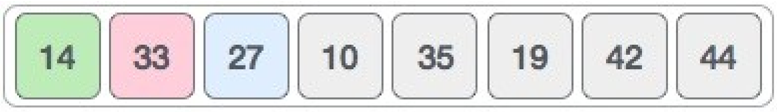
14	33	27	10	35	19	42	44

Insertion sort moves ahead and compares 33 with 27.



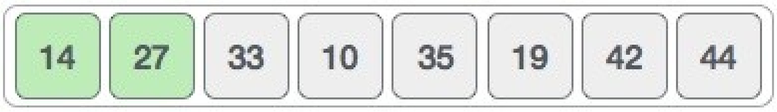
14	33	27	10	35	19	42	44

And finds that 33 is not in the correct position.



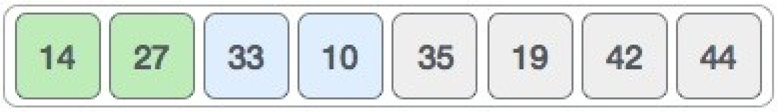
14	33	27	10	35	19	42	44

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



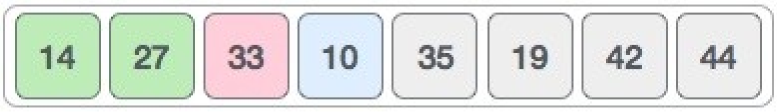
14	27	33	10	35	19	42	44

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

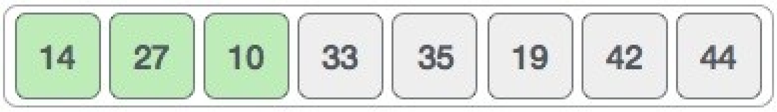


14	27	33	10	35	19	42	44

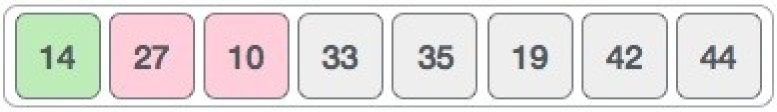
These values are not in a sorted order.



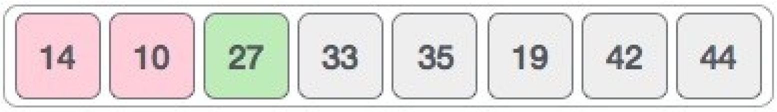
So we swap them.



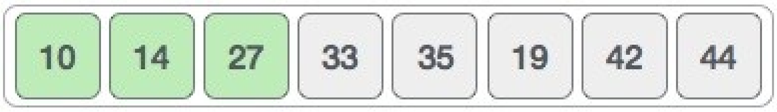
However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Pseudocode

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Time Complexity: O(n*n)

4. Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

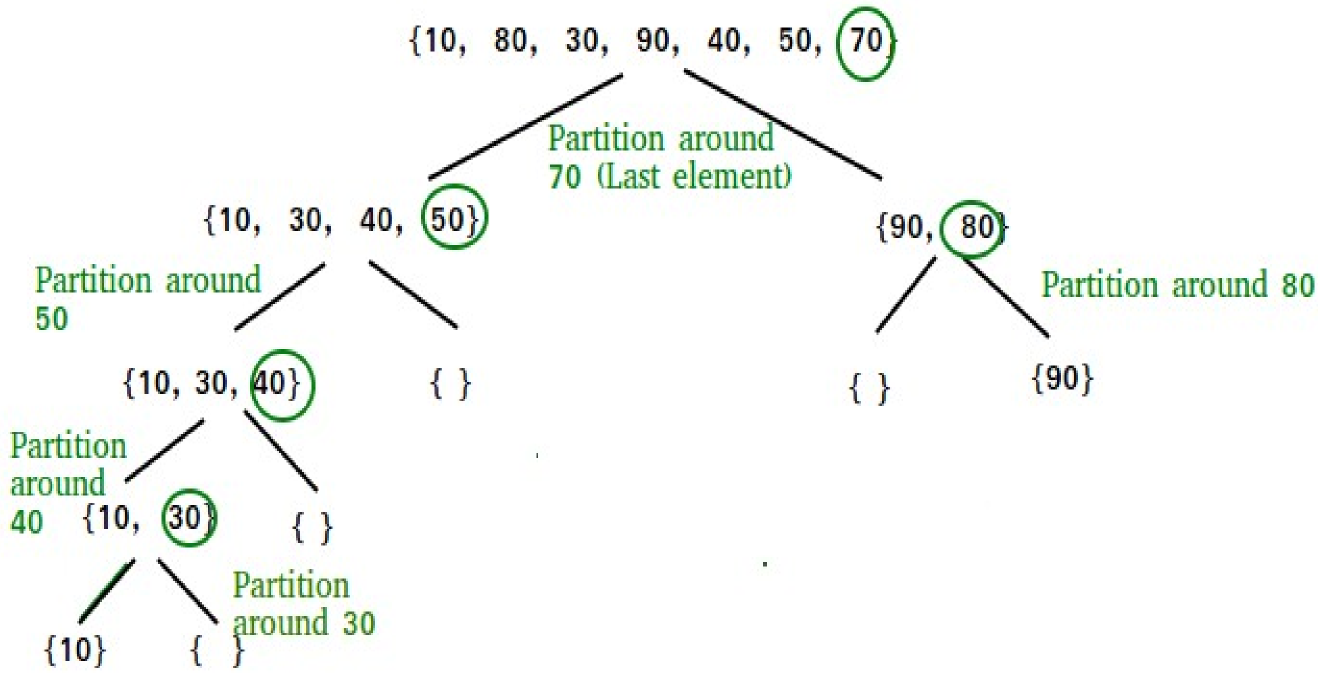
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
```

```
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
```

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

<u>//Implementation of Quick Sort</u>

```
#include<stdio.h>

#include<conio.h>

void quick_sort(int[],int,int);

int partition(int[],int,int);



void quick_sort(int a[100],int l,int u)

{

int j;

if(l<u)

{

j=partition(a,l,u);//l=0,u=5

quick_sort(a,l,j-1);//recursive call for elements less than pivot

quick_sort(a,j+1,u);

}

}

int partition(int a[100],int l,int u)

{

int v,i,j,temp;

//select pivot (element to be placed at right position)

v=a[l];

i=l;//lower bound

j=u+1;//upper bound

do
```

```
{
do
{
i++;
}while(a[i]<v&&i<=u);//find the element greater than pivot
do
{
j--;
}while(a[j]>v);//find the element smaller than pivot
if(i<j)//swap the greater element and smaller element
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}while(i<j);
a[l]=a[j];//finally swap data of a[j] and pivot i.e. a[l]
a[j]=v;
return(j);
}

void main()
{
int a[100],n,i,op;

    printf("QUICK SORT\n");
    printf("Enter no of elements\n");
    scanf("%d",&n);//n=6
```

```
        printf("Enter the nos\n");
        for(i=0;i<=n-1;i++)
        scanf("%d",&a[i]);//{14,12,16,13,11,15}
        quick_sort(a,0,n-1);//n-1=5
        printf("sorted array is\n");
        for(i=0;i<=n-1;i++)
        printf("%d\n",a[i]);
}
```

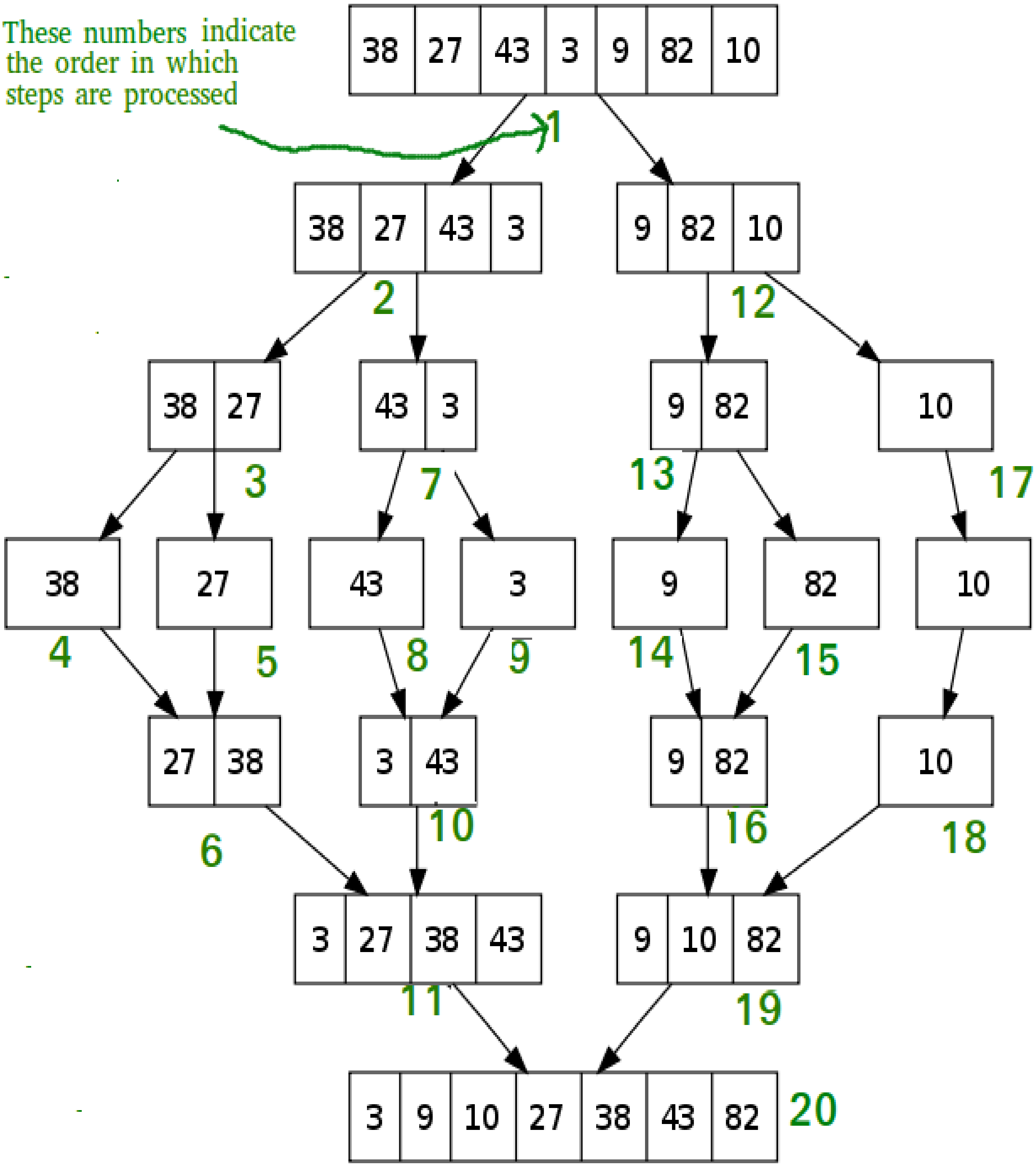
Time Complexity: O(nlogn)

5. Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

//Implementation of Merge Sort

```
#include<stdio.h>
#include<conio.h>
void mergesort(int[],int,int);
void merge(int[],int,int,int,int);
void mergesort(int a[100],int i, int j)//divide the elements by recursive calls to mergesort()
{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid);
mergesort(a,mid+1,j);
merge(a,i,mid,mid+1,j);
}
}
void merge(int a[100],int i1,int j1, int i2,int j2)
{
int temp[100];
int i,j,k;
i=i1;//lower bound of array 0
j=i2;//j=mid +1
k=0;
while(i<=j1&&j<=j2)
{
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1)//copy remaining elements from first half
temp[k++]=a[i++];
while(j<=j2)//copy remaining elements from second half
temp[k++]=a[j++];
for(i=i1,j=0;i<=j2;i++,j++)//copy sorted array from temp to a
a[i]=temp[j];
}
void main()
{
int a[100],n,i,op;
    printf("MERGE SORT\n");
    printf("Enter no of elements\n");
    scanf("%d",&n);//let consider n=8
    printf("Enter the nos\n");
    for(i=0;i<=n-1;i++)
    scanf("%d",&a[i]);//2,4,1,6,8,5,3,9
    mergesort(a,0,n-1);//here n-1 is 7
    printf("sorted array is\n");
    for(i=0;i<=n-1;i++)
    printf("%d\n",a[i]);
}
```

Time Complexity: O(nlogn)

6. 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 maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, 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 value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

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 of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.)

1. Build the heap

Heap	newly added element	swap elements
null	6	
6	5	
6, 5	3	
6, 5, 3	1	
6, 5, 3, 1	8	
6, **5**, 3, 1, **8**		5, 8
6, **8**, 3, 1, 5		6, 8
8, 6, 3, 1, 5	7	
8, 6, **3**, 1, 5, **7**		3, 7

8, 6, 7, 1, 5, 3, 2	4	
8, 6, 7, **1**, 5, 3, 2, **4**		1, 4
8, 6, 7, 4, 5, 3, 2, 1		

2. Sorting.

Heap	swap elements	delete element	sorted array	details
8, 6, 7, 4, 5, 3, 2, **1**	8, 1			swap 8 and 1 in order to delete 8 from heap
1, 6, 7, 4, 5, 3, 2, **8**		8		delete 8 from heap and add to sorted array
1, 6, **7**, 4, 5, 3, 2	1, 7		8	swap 1 and 7 as they are not in order in the heap
7, 6, **1**, 4, 5, **3**, 2	1, 3		8	swap 1 and 3 as they are not in order in the heap
7, 6, 3, 4, 5, 1, **2**	7, 2		8	swap 7 and 2 in order to delete 7 from heap
2, 6, 3, 4, 5, 1, **7**		7	8	delete 7 from heap and add to sorted array
2, **6**, 3, 4, 5, 1	2, 6		7, 8	swap 2 and 6 as they are not in order in the heap
6, **2**, 3, 4, **5**, 1	2, 5		7, 8	swap 2 and 5 as they are not in order in the heap
6, 5, 3, 4, 2, **1**	6, 1		7, 8	swap 6 and 1 in order to delete 6 from heap
1, 5, 3, 4, 2, **6**		6	7, 8	delete 6 from heap and add to sorted array
1, **5**, 3, 4, 2	1, 5		6, 7, 8	swap 1 and 5 as they are not in order in the heap
5, **1**, 3, **4**, 2	1, 4		6, 7, 8	swap 1 and 4 as they are not in order in the heap
5, 4, 3, 1, **2**	5, 2		6, 7, 8	swap 5 and 2 in order to delete 5 from heap
2, 4, 3, 1, **5**		5	6, 7, 8	delete 5 from heap and add to sorted array
2, **4**, 3, 1	2, 4		5, 6, 7, 8	swap 2 and 4 as they are not in order in the heap
4, 2, 3, **1**	4, 1		5, 6, 7, 8	swap 4 and 1 in order to delete 4 from heap
1, 2, 3, **4**		4	5, 6, 7, 8	delete 4 from heap and add to sorted

				array
1, 2, **3**	1, 3		4, 5, 6, 7, 8	swap 1 and 3 as they are not in order in the heap
3, 2, **1**	3, 1		4, 5, 6, 7, 8	swap 3 and 1 in order to delete 3 from heap
1, 2, **3**		3	4, 5, 6, 7, 8	delete 3 from heap and add to sorted array
1, **2**	1, 2		3, 4, 5, 6, 7, 8	swap 1 and 2 as they are not in order in the heap
2, **1**	2, 1		3, 4, 5, 6, 7, 8	swap 2 and 1 in order to delete 2 from heap
1, **2**		2	3, 4, 5, 6, 7, 8	delete 2 from heap and add to sorted array
1		1	2, 3, 4, 5, 6, 7, 8	delete 1 from heap and add to sorted array
			1, 2, 3, 4, 5, 6, 7, 8	completed

//Implememtation of Heap Sort

```
#include <stdio.h>
void main()
{
int heap[10],no,i,j,c,root,temp;
printf("\n Enter no of elements :");
scanf("%d",&no);
printf("\n Enter the nos : ");
for (i = 0;i < no;i++)
scanf("%d",&heap[i]);
for (i = 1;i<no;i++)
{
c=i;
do
{
root = (c - 1)/2;
```

```
if (heap[root] < heap[c])
{
temp = heap[root];
heap[root] = heap[c];
heap[c] = temp;
}
c = root;
} while (c != 0);
}
/* to create MAX heap array */
printf("Heap array : ");
for (i=0;i < no;i++)
printf("%d\t ", heap[i]);
for (j=no-1;j>=0;j--)
{
temp=heap[0];
heap[0]=heap[j];
heap[j]=temp;
root=0;
do
{
c=2*root+1;
/* swap max element with rightmost leaf element */
/* left node of root element */
if ((heap[c]<heap[c+1])&&c<j-1)
c++;
if (heap[root]<heap[c]&&c<j)
{
```

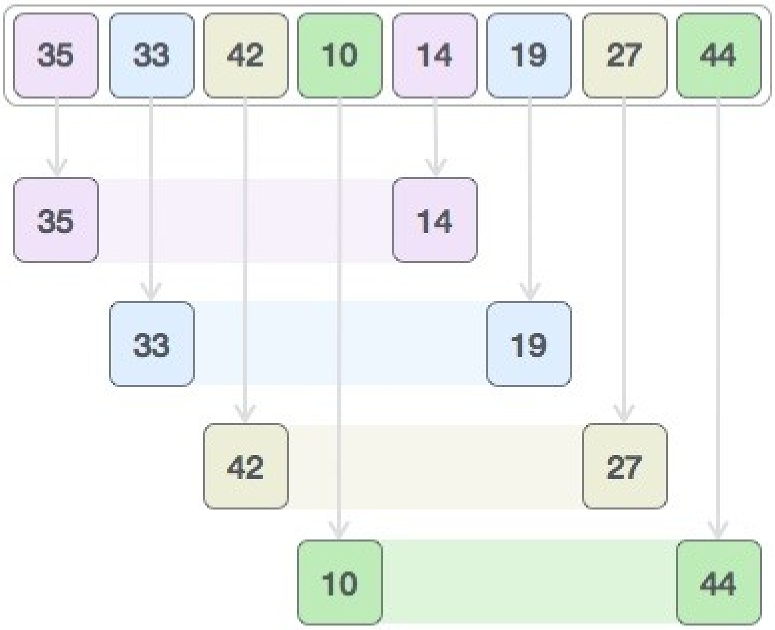
```
temp=heap[root];
heap[root]=heap[c];
heap[c]=temp;
}
root=c;
/* again rearrange to max heap array */
} while(c<j);
}
printf("\n The sorted array is : ");
for (i=0;i<no;i++)
printf("\t%d",heap[i]);
printf("\n");
//printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn) \n");
}
```

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

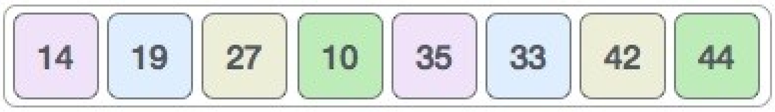
7. Shell sort

Shell Sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

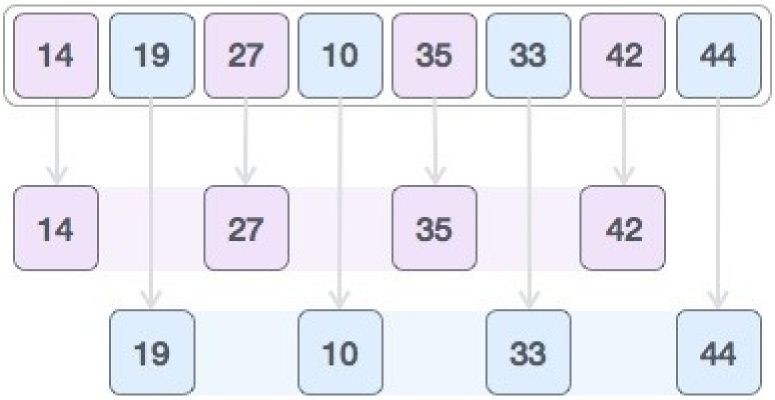
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



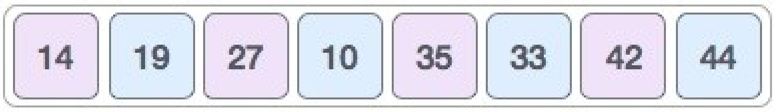
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}

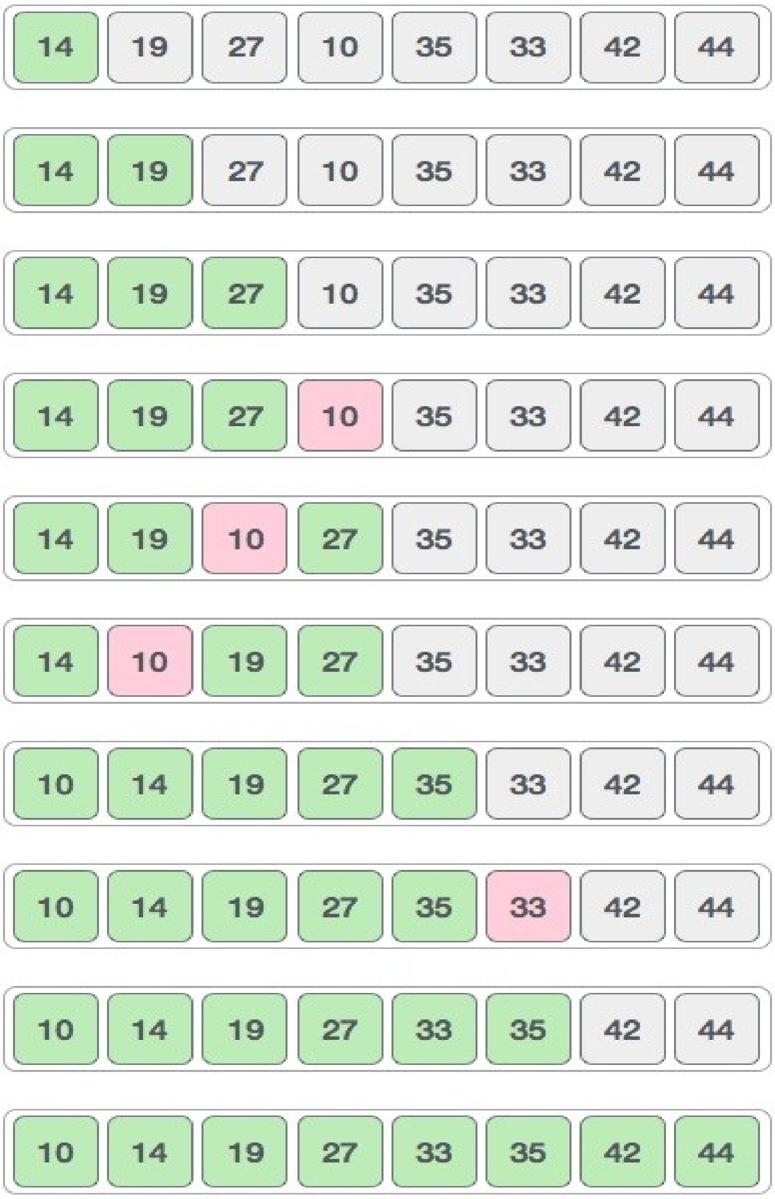


We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 10 27 35 33 42 44

14 10 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

Algorithm

Following is the algorithm for shell sort.

```
Step 1 − Initialize the value of h
Step 2 − Divide the list into smaller sub-list of equal interval h
Step 3 − Sort these sub-lists using insertion sort
Step 3 − Repeat until complete list is sorted
```

Time Complexity: Time complexity of above implementation of shellsort is O(n^2). In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity.

8. Radix Sort

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.

Steps:

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process. In other words:

1. Take the least significant digit of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

Step-by-step example:

Original, unsorted list:

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 most significant digit (100s place) gives:

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

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075,090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)
2 (bucket size for digits of 2: 002, 802)
1 (bucket size for digits of 4: 024)
2 (bucket size for digits of 5: 045, 075)
1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)
1 (bucket size for digits of 2: 024)
1 (bucket size for digits of 4: 045)
1 (bucket size for digits of 6: 066)
2 (bucket size for digits of 7: 170, 075)
1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0:002,024, 045, 066, 075, 090)
1 (bucket size for digits of 1: 170)
1 (bucket size for digits of 8: 802)

```
Algorithm: Radix-Sort (list, n)
shift = 1
for loop = 1 to keysize do
   for entry = 1 to n do
      bucketnumber = (list[entry].key / shift) mod 10
      append (bucket[bucketnumber], list[entry])
   list = combinebuckets()
   shift = shift * 10
```

Analysis

Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n** keys, radix sort has order **O(m.n)**.

However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records.

Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	**1st Pass**	**2nd Pass**	**3rd Pass**
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

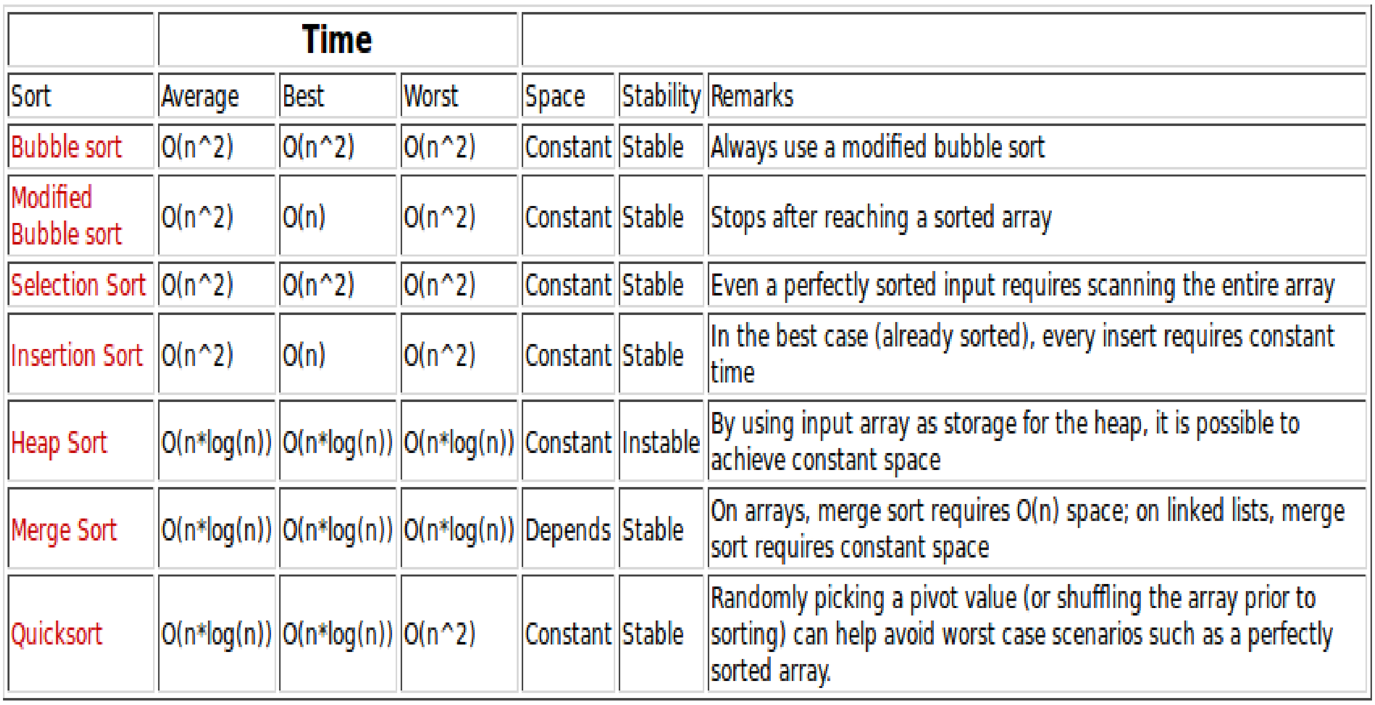
In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n*** elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.

Analysis of Sorting Algorithms

Comparison of Sorting algorithms

In this table, *n* is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. The run times and the memory requirements listed below should be understood to be inside big O notation, hence the base of the logarithms does not matter; the notation $\log^2 n$ means $(\log n)^2$.

These are all comparison sorts, and so cannot perform better than O(*n* log *n*) in the average or worst case.



	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	Stops after reaching a sorted array
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

Introduction to Searching Techniques

Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Example:

```
// Linearly search x in arr[].  If x is present then return its
// location,  otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
        if (arr[i] == x)
         return i;
    return -1;
}
```

The time complexity of above algorithm is O(n).

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

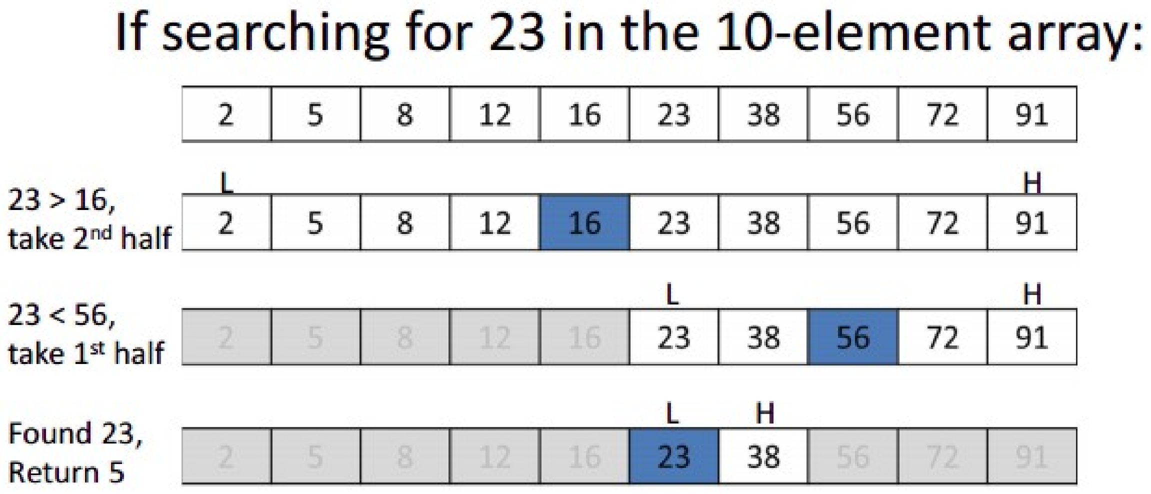
Binary Search

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

A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search: Search 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 item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Logn).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Time Complexity:
The time complexity of Binary Search can be written as

```
T(n) = T(n/2) + c
```

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is $Theta(Logn)$.

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Algorithmic Paradigm: Divide and Conquer

Hashing Techniques

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

   hash = hashfunc(key)
   index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

   Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

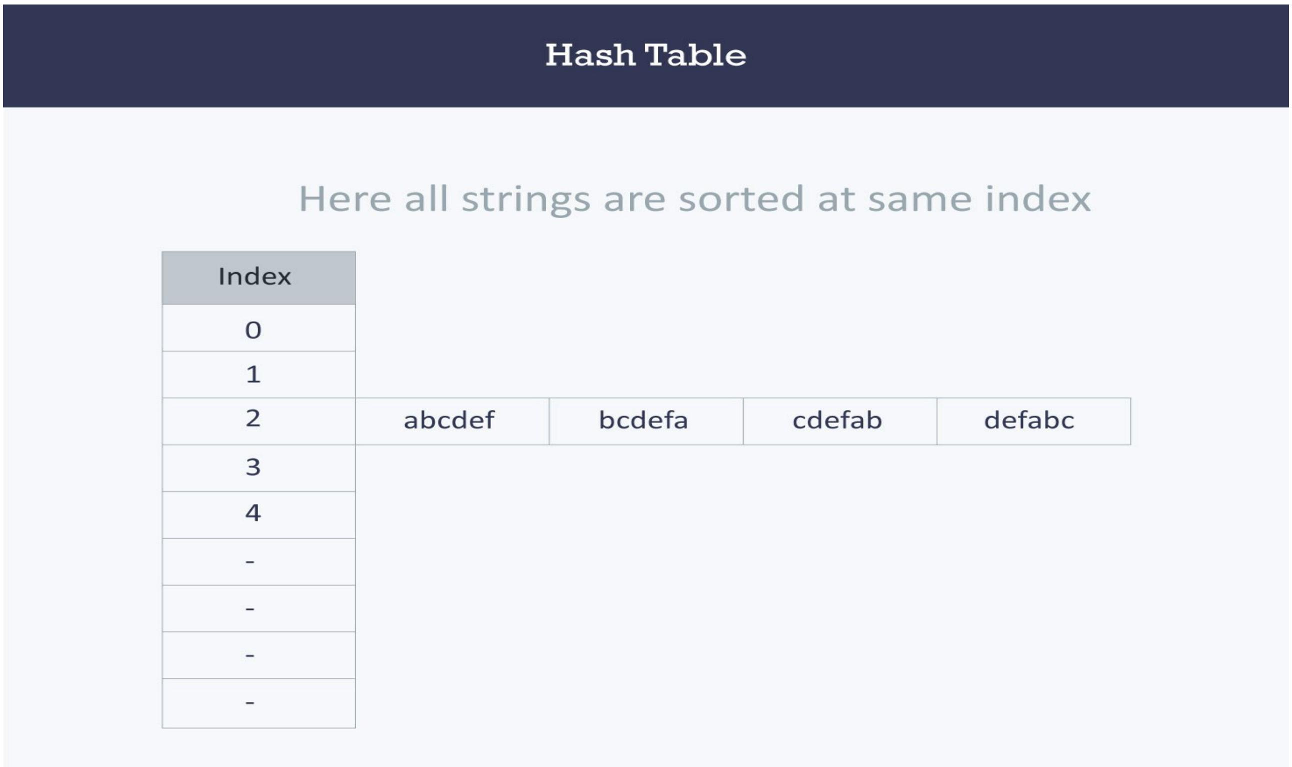
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



Hash Table

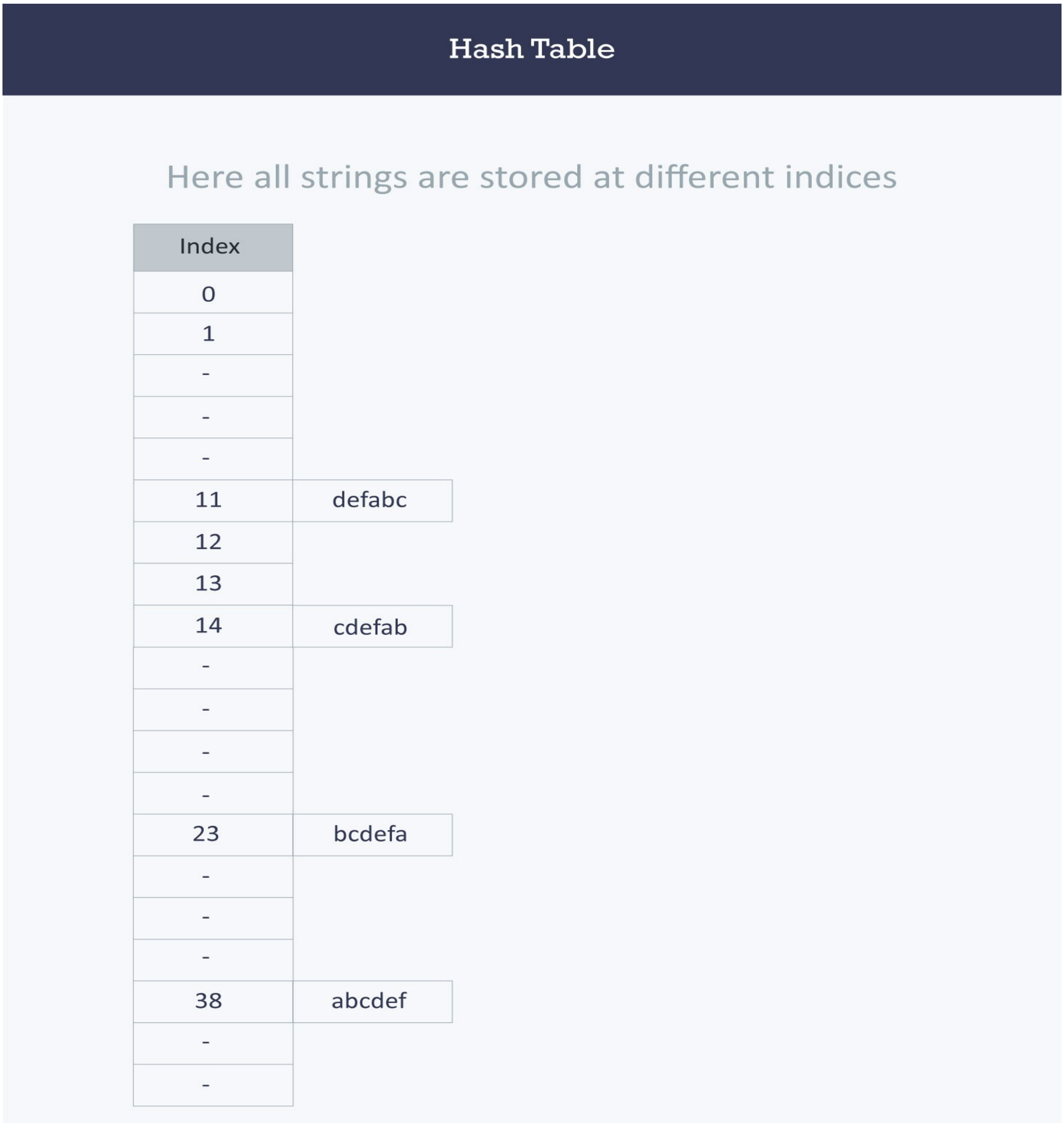
Here all strings are sorted at same index

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

Here, it will take O(n) time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.

Let's try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11



Hash Table

Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Collision

When two different keys produce the same address, there is a collision. The keys involved are called synonyms.Coming up with a hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them. The possible solution, can be:

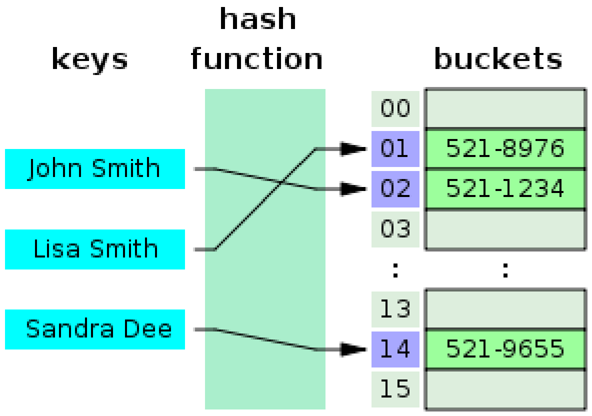
- Spread out the records
- Use extra memory
- Put more than one record at a single address.

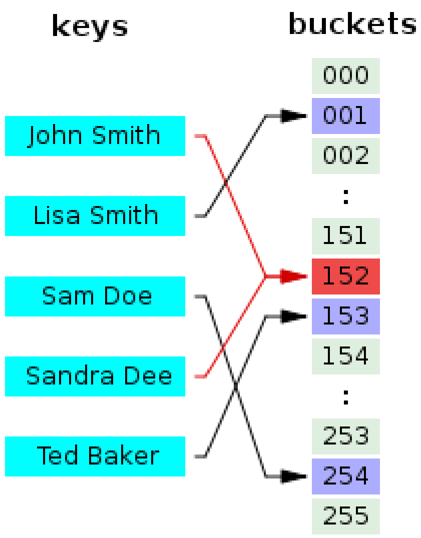
An example of Collision

Hash table size: 11
Hash function: key mod hash size
So, the new positions in the hash table are:

Key	23	18	29	28	39	13	16	42	17
Position	1	7	7	6	6	2	5	9	6

Another example (in a phonebook record):





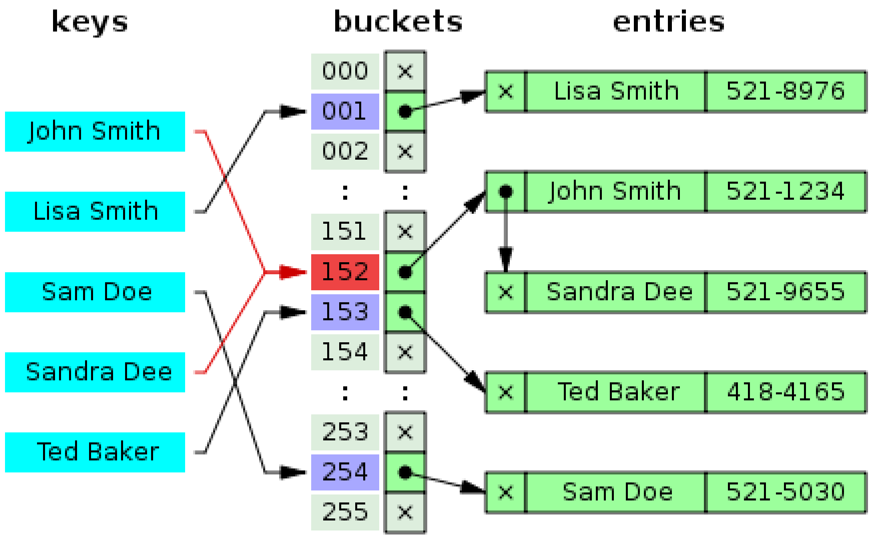
Here, the buckets for keys 'John Smith' and 'Sandra Dee' are the same. So, its a collision case.

Collision Resolution

Collision occurs when h(k1) = h(k2), i.e. the hash function gives the same result for more than one key. The strategies used for collision resolution are:

- **Chaining**
  - Store colliding keys in a linked list at the same hash table index
- **Open Addressing**
  - Store colliding keys elsewhere in the table

Chaining



Strategy:

Maintains a linked list at every hash index for collided elements.

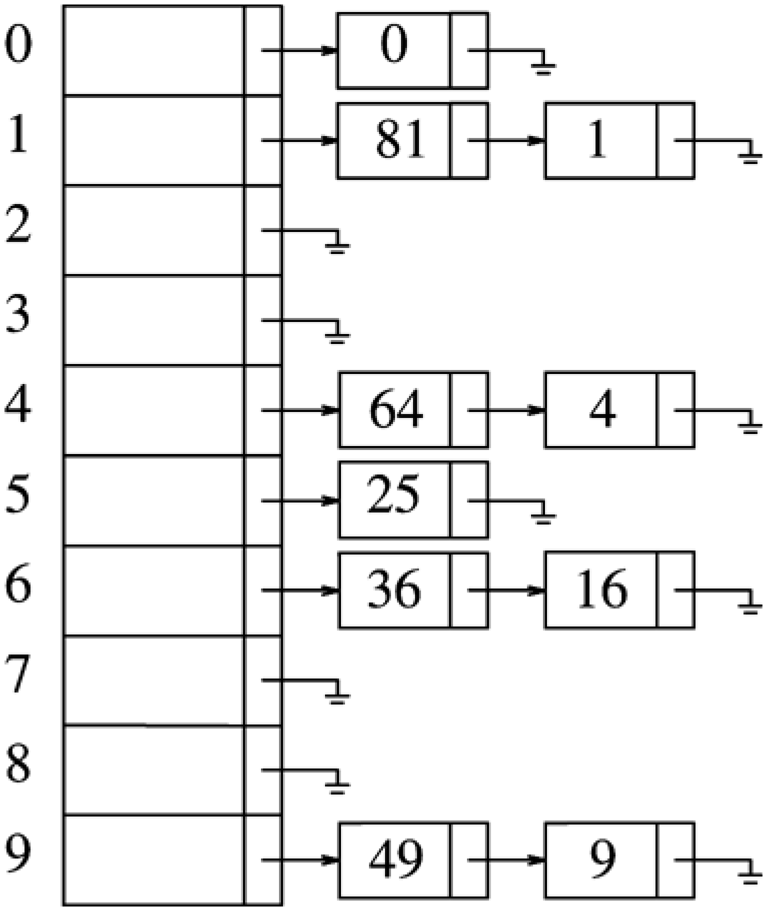
Lets take the example of an insertion sequence: {0 1 4 9 16 25 36 49 64 81}

Here, h(k) = k mod tablesize = k mod 10 (tablesize = 10)

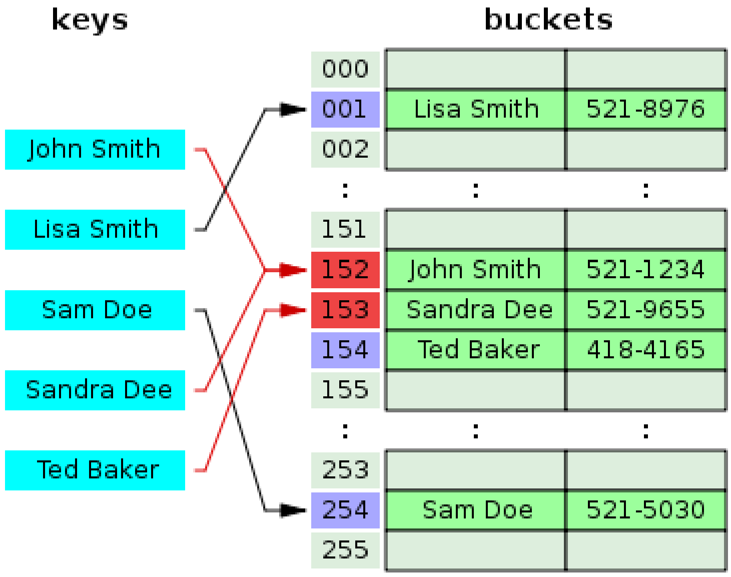
- Hash table T is a vector of linked lists
  Insert element at the head (as shown here) or at the tail
- Key k is stored in list at T[h(k)]

So, the problem is like: "Insert the first 10 perfece squares in a hash table of size 10"

The hash table looks like:



Open Addressing



As shown in the above figure, in open addressing, when collision is encountered, the next key is inserted in the empty slot of the table. So, it is an 'inplace' approach.

Probing

The next slot for the collided key is found in this method by using a technique called **"Probing".** It generates a probe sequence of slots in the hash table and we need to chose the proper slot for the key 'x'.

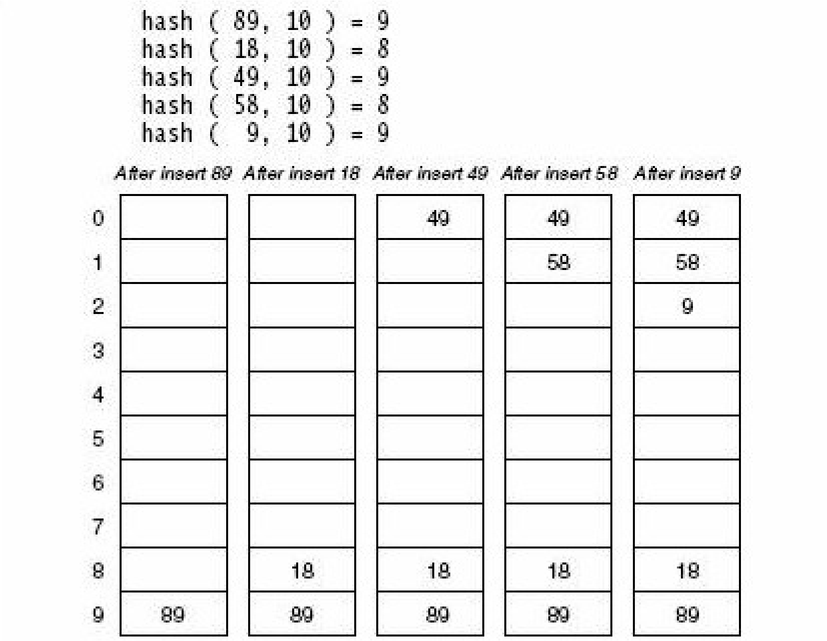
- h0(x), h1(x), h2(x), …
- Needs to visit each slot exactly once
- Needs to be repeatable (so we can find/delete what we've inserted)

- *Hash function*
    - hi(x) = (h(x) + f(i)) mod TableSize
    - f(0) = 0 ==> position for the 0th probe
    - f(i) is "the distance to be traveled relative to the 0th probe position, during the ith probe".

Some of the common methods of probing are:

1. Linear Probing:

Suppose that a key hashes into a position that has been already occupied. The simplest strategy is to look for the next available position to place the item. Suppose we have a set of hash codes consisting of {89, 18, 49, 58, 9} and we need to place them into a table of size 10. The following table demonstrates this process.



```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

The first collision occurs when 49 hashes to the same location with index 9. Since 89 occupies the A[9], we need to place 49 to the next available position. Considering the array as circular, the next available position is 0. That is (9+1) mod 10. So we place 49 in A[0].

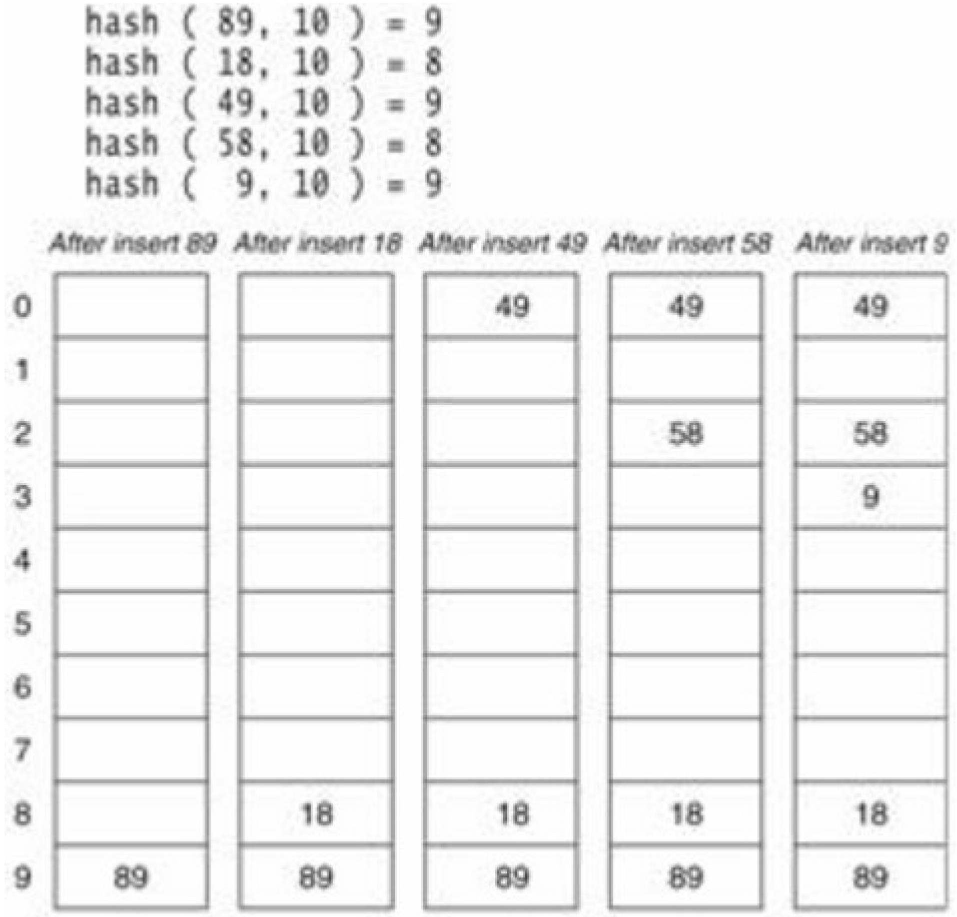
Several more collisions occur in this simple example and in each case we keep looking to find the next available location in the array to place the element. Now if we need to find the element, say for example, 49, we first compute the hash code (9), and look in A[9]. Since we do not find it there, we look in A[(9+1) % 10] = A[0], we find it there and we are done.

So what if we are looking for 79? First we compute hashcode of 79 = 9. We probe in A[9], A[(9+1)]=A[0], A[(9+2)]=A[1], A[(9+3)]=A[2], A[(9+4)]=A[3] etc. Since A[3] = null, we do know that 79 could not exists in the set.

2. Quadratic Probing:

Although linear probing is a simple process where it is easy to compute the next available location, linear probing also leads to some clustering when keys are computed to closer values. Therefore we define a new process of Quadratic probing that provides a better distribution of keys when collisions occur. In quadratic probing, if the hash value is K , then the next location is computed using the sequence K + 1, K + 4, K + 9 etc..

The following table shows the collision resolution using quadratic probing.



```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

- Avoids primary clustering
- f(i) is quadratic in i: eg: $f(i) = i^2$
- $h_i(x) = (h(x) + i^2) \bmod \text{tablesize}$

3. Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
Hash$_1$(key) = key % 10
Hash$_2$(key) = 7 – (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Analysis of Searching Techniques

Consider a list of n elements or can represent a file of n records, where each element is a key / number. The task is to find a particular key in the list in the shortest possible time. If you know you are going to search for an item in a set, you will need to think carefully about what type of data structure you will use for that set. At low level, the only searches that get mentioned are for sorted and unsorted arrays. However, these are not the only data types that are useful for searching.

1. Linear search

Start at the beginning of the list and check every element of the list. Very slow [order O(n)] but works on an unsorted list.

2. Binary Search

This is used for searching in a sorted array. Test the middle element of the array. If it is too big. Repeat the process in the left half of the array, and the right half if it’s too small. In this way, the amount of space that needs to be searched is halved every time, so the time is O(log n).

3. Hash Search

Searching a hash table is easy and extremely fast, just find the hash value for the item you’re looking for then go to that index and start searching the array until you find what you are looking for or you hit a blank spot. The order is pretty close to o(1), depending on how full your hash table is.