Sorting

The concept of an ordered set of elements is one that has considerable impact in our daily lives. So sorting is one of the most common ingredients of programming systems. The process of rearranging the items in a list according to some linear order is termed as sorting.

– The process of finding a telephone number in a telephone directory is simplified considerably by the fact that the names are listed in alphabetical order.
– In a library, books are shelved in a specific order, each book is assigned a specific position relative to the others and can be retrieved in a reasonable amount of time.

Types of sorting
Internal sorting : Records to be sorted are in main memory.
External sorting: Records to be sorted, or some of them are kept in auxiliary storage(disk/tape).

Stable Sort

- It is possible for two records in a list to have the same key
- A sorting algorithm is ***stable*** if for all records ***i*** and ***j*** such that ***k*[*i*]** equals ***k*[*j*]**, if ***r*[*i*]** preceded ***r*[*j*]** in the original list, ***r*[*i*]** precedes ***r*[*j*]** in the sorted list
    - i.e, a stable sort keeps records with same key in the same relative order that they were in before the sort

Original Table	
Name	**Address**
AAB	**BBC**
CDI	LKK
AAB	**BBA**
KKA	KSO
IEH	IEU

Sorted Table	
Name	**Address**
AAB	**BBC**
AAB	**BBA**
CDI	LKK
IEH	IEU
KKA	KSO

This sort is known as stable sort. In normal considerations, the data is sorted like;
AAB BBA
AAB BBC
But the stable sort keeps track of original pattern unless specified.

Sorting algorithm:
Exchange sort:
Comparison based. The basic idea is to compare two elements; if out of order, swap them or move one of the elements. Example: Bubble sort, Quick sort.

Selection sort:
An element is selected and is placed in its correct position. Example: Selection sort, Heap sort.

Insertion sort:
Sorts by inserting an element into a sorted list. Example : insertion sort, merge sort.

Bubble sort:

Basic Idea: Pass through the list sequentially several times. At each pass, each element in the list is compared to its successor and they are interchanged if they are not in proper order.

At each pass, one element will be in its proper positions. In general, A[n-i] will be in its place after pass i. Since each pass place a new element in its proper position, a total of N-1 passes are required for a list of N elements. Also, all the elements in positions greater than or equal to N-i are already in proper position after pass i, so they need not be considered in succeeding passes.

Procedure:

```
void bubblesort(int a[], int N)
{
        int pass, j;
        for(pass=0; pass<N-1; pass++)
        {       for(j=0; j<N-pass-1; j++)
                        if(a[j]>a[j+1]
                                swap(&a[j], &a[j+1]);
        }
}
```

Tracing example, total no. of items N = 8

Original list **25, 57, 48, 37, 12, 92, 86, 33**.

25	**57**	**48**	**37**	**12**	**92**	**86**	**33**	**Interchange**	
25	**57**	**48**	**37**	**12**	**92**	**86**	**33**	**No**	
25	**57**	**48**	**37**	**12**	**92**	**86**	**33**	**Yes**	
25	**48**	**57**	**37**	**12**	**92**	**86**	**33**	**Yes**	
25	**48**	**37**	**57**	**12**	**92**	**86**	**33**	**Yes**	1st Pass
25	**48**	**37**	**12**	**57**	**92**	**86**	**33**	**No**	
25	**48**	**37**	**12**	**57**	**92**	**86**	**33**	**Yes**	
25	**48**	**37**	**12**	**57**	**86**	**92**	**33**	**Yes**	
25	**48**	**37**	**12**	**57**	**86**	**33**	**92**		

25	**48**	**37**	**12**	**57**	**86**	**33**	**|92**	**No**	
25	**48**	**37**	**12**	**57**	**86**	**33**	**|92**	**Yes**	
25	**37**	**48**	**12**	**57**	**86**	**33**	**|92**	**Yes**	
25	**37**	**12**	**48**	**57**	**86**	**33**	**|92**	**No**	2nd Pass
25	**37**	**12**	**48**	**57**	**86**	**33**	**|92**	**No**	
25	**37**	**12**	**48**	**57**	**86**	**33**	**|92**	**Yes**	
25	**37**	**12**	**48**	**57**	**33**	**86**	**|92**		

25	**37**	**12**	**48**	**57**	**33**	**86**	**92**	**Interchange**	
25	**37**	**12**	**48**	**57**	**33**	**|86**	**92**	**No**	
25	**37**	**12**	**48**	**57**	**33**	**|86**	**92**	**Yes**	
25	**12**	**37**	**48**	**57**	**33**	**|86**	**92**	**No**	3^{rd} Pass
25	**12**	**37**	**48**	**57**	**33**	**|86**	**92**	**No**	
25	**12**	**37**	**48**	**57**	**33**	**|86**	**92**	**Yes**	
25	**12**	**37**	**48**	**33**	**57**	**|86**	**92**		
25	**12**	**37**	**48**	**33**	**|57**	**86**	**92**	**Interchange**	
25	**12**	**37**	**48**	**33**	**|57**	**86**	**92**	**Yes**	
12	**25**	**37**	**48**	**33**	**|57**	**86**	**92**	**No**	4^{th} Pass
12	**25**	**37**	**48**	**33**	**|57**	**86**	**92**	**No**	
12	**25**	**37**	**48**	**33**	**|57**	**86**	**92**	**Yes**	
12	**25**	**37**	**33**	**48**	**|57**	**86**	**92**		
12	**25**	**37**	**33**	**|48**	**57**	**86**	**92**	**Interchange**	
12	**25**	**37**	**33**	**|48**	**57**	**86**	**92**	**No**	
12	**25**	**37**	**33**	**|48**	**57**	**86**	**92**	**No**	5^{th} Pass
12	**25**	**37**	**33**	**|48**	**57**	**86**	**92**	**Yes**	
12	**25**	**33**	**37**	**|48**	**57**	**86**	**92**		
12	**25**	**33**	**|37**	**48**	**57**	**86**	**92**	**Interchange**	
12	**25**	**33**	**|37**	**48**	**57**	**86**	**92**	**No**	6^{th} Pass
12	**25**	**33**	**|37**	**48**	**57**	**86**	**92**	**No**	
12	**25**	**33**	**|37**	**48**	**57**	**86**	**92**		
12	**25**	**33**	**37**	**48**	**57**	**86**	**92**	**Interchange**	
12	**25**	**|33**	**37**	**48**	**57**	**86**	**92**	**No**	7^{th} Pass
12	**25**	**|33**	**37**	**48**	**57**	**86**	**92**		

Algorithm:

- Given a list A of size N, the following algorithm uses bubble sort to sort the list
  - For ***pass* = 0** To ***N* – 2**
    - For ***j* = 0** To ***N* – *pass* – 2**
      - If ***A*[*j*] > *A*[*j* + 1]**

        Swap the elements ***A*[*j*]** and ***A*[*j* + 1]**

        End If
    - End For
  - End For

Efficiency:

This algorithm is good for small n usually less than 100 elements.

No. of comparisons = (n-1) + (n-2) + … +2 + 1

$$= (n-1)(n-1 + 1)/2$$
$$= n(n-1)/2$$
$$= O(n^2)$$

No. of Interchanges:
- This cannot be greater than no. of comparisons
- In the best case, there are no interchanges
- In the worst case, this equals no of comparisons

The average and worse case running time of bubble sort is $\boldsymbol{O(n^2)}$.
It is actually the no of interchanges which takes up most time of the program's execution than the no of comparisons.
When elements are large and interchange operation is expensive, it is better to maintain an array of pointers to the elements. One can then interchange pointers rather then the elements itself.

Insertion Sort
Basic idea: Sorts a list of record by inserting new element into an existing sorted list. An initial list with only one item is considered to be sorted list. For a list of size N, N-1 passes are made, and for each pass the elements from a[0] through a[i-1] are sorted.
Take the element a[i], find the proper place to insert a[i] within 0, 1, …, i-1 and insert a[i] at that place.
To insert new item into the list
– Search the position in the sorted sublist from last toward first
– While searching, move elements one position right to make a room to insert a[i]
– Place a[i] in its proper place

Initially 25 57 48 37 12 92 86 33

Pass 1	**<u>25</u>**	**<u>57</u>**	**48**	**37**	**12**	**92**	**86**	**33**	***Insert 57***
Pass 2	**<u>25</u>**	**<u>48</u>**	**<u>57</u>**	**37**	**12**	**92**	**86**	**33**	***Insert 48***
Pass 3	**<u>25</u>**	**<u>37</u>**	**<u>48</u>**	**<u>57</u>**	**12**	**92**	**86**	**33**	***Insert 37***
Pass 4	**<u>12</u>**	**<u>25</u>**	**<u>37</u>**	**<u>48</u>**	**<u>57</u>**	**92**	**86**	**33**	***Insert 12***
Pass 5	**<u>12</u>**	**<u>25</u>**	**<u>37</u>**	**<u>48</u>**	**<u>57</u>**	**<u>92</u>**	**86**	**33**	***Insert 92***
Pass 6	**<u>12</u>**	**<u>25</u>**	**<u>37</u>**	**<u>48</u>**	**<u>57</u>**	**<u>86</u>**	**<u>92</u>**	**33**	***Insert 86***
Pass 7	**<u>12</u>**	**<u>25</u>**	**<u>33</u>**	**<u>37</u>**	**<u>48</u>**	**<u>57</u>**	**<u>86</u>**	**<u>92</u>**	***Insert 33***

C-Procedure

```
void InsertionSort(int a[], int N)
{
        int i, j;
        int hold; /* the current element to insert */

        for (i = 1; i < N; i++)  // Insert a[i] into the sorted list
        {
                hold = a[i]; /* hold the element to be inserted */
                for (j = i-1; j >= 0 && a[j] > hold; j--)        //Move right 1 position all
                        a[j+1] = a[j];                           //elements greater than hold

                a[j+1] = hold; /* Place hold in its proper place */
        }
}
```

Efficiency:

No of comparisons:

Best case: $\boldsymbol{n - 1}$

Worst case: $\boldsymbol{n^2/2 + O(n)}$

Average case: $\boldsymbol{n^2/4 + O(n)}$

No of assignments (movements)

Best case: $\boldsymbol{2*(n-1)}$ // moving from a[i] to hold and back

Worst case: $\boldsymbol{n^2/2 + O(n)}$

Average case: $\boldsymbol{n^2/4 + O(n)}$

Hence running time of insertion sort is $\mathbf{O(n^2)}$ in worst and average case and $\mathbf{O(n)}$ in best case and space requirement is $\mathbf{O(1)}$**.**

Advantages:

It is an excellent method whenever a list is nearly in the correct order and few items are removed from their correct locations

Since there is no swapping, it is twice as faster than bubble sort

Disadvantage:

It makes a large amount of shifting of sorted elements when inserting later elements.

Selection Sort:

The selection sort algorithm sorts a list by selecting successive elements in order and placing into their proper sorted positions.

A list of size N require N-1 passes:

For each pass I,

- Find the position of i^{th} largest (or smallest) element.
- To place the i^{th} largest (of smallest) in its proper position, swap this element with the element currently in the position of its largest (or smallest) element.

C – Procedure

```
void SelectionSort(int a[], int N)
{
        int i, j;
        int maxpos;
        for (i = N-1; i > 0; i--)        //Find the position of largest element from 0 to i
        {
                maxpos = 0;
                for (j = 1; j <= i; j++)
                        if (a[j] > a[maxpos])
                                maxpos = j;
                if(maxpos != i)
                        swap(&a[maxpos], &a[i]);        //Place the ith largest element
        }                                               // in its place
}
```

Tracing: Initially 25 57 48 37 12 92 86 33

Find largest between a[0] and a[7] -> 92, swap 92 with the last element 33

Pass 1 25 57 48 37 12 <u>33</u> 86 <u>92</u>

Find largest between a[0] and a[6] -> 86, since 86 is in 6th position and so is i, no interchange.

Pass 2 25 57 48 37 12 33 <u>86</u> 92

Find largest between a[0] and a[5] -> 57, swap with 33

Pass 3 25 <u>33</u> 48 37 12 <u>57</u> 86 92

Find largest between a[0] and a[4] -> 48, swap 48 with 12

Pass 4 25 33 <u>12</u> 37 <u>48</u> 57 86 92

Find largest between a[0] and a[3] -> 37, No swap since i = maxpos

Pass 5 25 33 12 <u>37</u> 48 57 86 92

Find largest between a[0] and a[2] -> 33, swap 33 with 12

Pass 6 25 <u>12</u> <u>33</u> 37 48 57 86 92

Find largest between a[0] and a[1] -> 25, swap 12 with 25

Pass 7 <u>12</u> <u>25</u> 33 37 48 57 86 92

Finally the list is sorted.

Efficiency:

No of comparisons:

Best, average and worst case: $n(n-1)/2$

No of assignments (movements)

Best, average and worst case: $3(n-1)$, (total $n-1$ swaps)

If we include a test, to prevent interchanging an element with itself, the number of interchanges in the best case would be 0.

Hence running time of selection sort is $O(n^2)$ and additional space requirements is $O(1)$.

- **Advantages:**
  - It is the best algorithm in regard to data movement
  - An element that is in its correct final position will never be moved and only one swap is needed to place an element in its proper position
- **Disadvantages**
  - In case of number of comparisons, it pays no attention to the original ordering of the list. For a list that is nearly correct to begin with, selection sort is slower than insertion sort

Divide and Conquer Sorting Algorithms

- The idea of dividing a problem into smaller but similar subproblems is called *divide and conquer*
- Divide and Conquer Sorting
  - Procedure Sort(list)

```
if (list has length greater than 1)
        Partition the list into two sublists lowlist, highlist
        Sort(lowlist)
        Sort(highlist)
        Combine (lowlist, highlist)
End If
End Procedure
```

Quick Sort:

It is the fastest known sorting algorithms used in practice.

Basic idea

Divide the list into two sublists such that all elements in the first list is less than some pivot key and all elements in the second list is greater than the pivot key,
and finally sort the sublists independently and combine them.

Algorithm:

If size of list *A* is greater than 1

- Pick any element *v* from *A*. This is called the *pivot*
- Partition the list *A* by placing *v* in some position *j*, such that
  - all elements before position *j* are less than or equal to *v*
  - all elements after position *j* are greater than or equal to *v*
- Recursively sort the sublists *A*[0] through *A*[*j*-1] and *A*[*j*+1] through *A*[*N*-1]
- Return *A*[0] through *A*[*j*-1] followed by *A*[*j*] (the pivot) followed by *A*[*j*+1] through *A*[*N*-1]

25 57 48 37 12 92 86 33

Choose the first element 25 as the pivot and partition the array

(12) 25 (57 48 37 92 86 33)

The first subarray is automatically sorted

12 25 (57 48 37 92 86 33)

Choose the first element 57 of the second subarray as the pivot and partition the subarray

12 25 (48 37 33) 57 (92 86)

12 25 (48 37 33) 57 (92 86)
12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

Quick Sort Code:

```
void QuickSort(int A[], int N)
{
        QSort(A, 0, N - 1);
}

void QSort(ItemType A[], int low, int high)
{
        int pivotloc;
        if (low < high)
        {
                pivotloc = partition(A, low, high);
                QSort(A, low, pivotloc - 1);
                QSort(A, pivotloc + 1, high);
        }
}
```

int partition(int A[], int low, int high)

```
{
        int down, up;
        int pivot;
        pivot = A[low];   /* choose first element as the pivot
        down = low;              /* Initialize pointers */
        up = high;

        while (down < up)
        {
                while (A[down] <= pivot && down < high)     /* move right */
                        down++;
                while (A[up] > pivot)                       /* move left */
                        up--;
                if (down < up)                 /* exchange element at up and down */
                        swap (&A[down], &A[up]);
        }
        swap (&A[low], &A[up]);              /* Place pivot at its proper position */
  return up;                                 /* return the pivot location */
}
```

Description of partition procedure

- Choose any element as the pivot, here we choose the first element as the first item in the list.
- Initialize two pointers, ***up*** and ***down*** to the upper bound (***low***) and lower bound (***high***) of the array
- While ***down*** is left of ***up***, repeat these steps
  - Move ***down*** right, skipping over elements that are smaller than or equal to pivot.
  - Move ***up*** left, skipping over elements that are larger than the pivot
  - When ***down*** and ***up*** have stopped, ***down*** is pointing at a large element and ***up*** is pointing at a small element
  - If ***down*** is left of ***up***, swap the elements at ***down*** and ***up*** (The effect is to push a large element to the right and a small element to the left)
- Swap the first element (***pivot***) with the element at ***up***
- Return ***up*** as the pivot location

Tracing Example:

Sorting the following data using Quick Sort.

25 57 48 37 12 92 86 33

25	57	48	37	12	92	86	33		
Choose Pivot 25									
Down							up		
25	57	48	37	12	92	86	33		
	down						up		
25	57	48	37	12	92	86	33		
	down			Up					
25	57	48	37	12	92	86	33		
down < up, so swap up and down and increment down									
		down		Up					
25	12	48	37	57	92	86	33	down > 25 so stop	
	up	down							
25	12	48	37	57	92	86	33	down>=up, swap with up	
(12)	25	(48	37	57	92	86	33)		
List is divided into two lists, pivot is at its proper position. First list contains only one element so automatically sorted. Now choose 48 as pivot element for second list.									
Choose pivot 48		down					up		
12	25	48	37	57	92	86	33		
				down			up		
12	25	48	37	57	92	86	33	Down<up so, swap	
				down			up		
12	25	48	37	33	92	86	57		
					down		up	Move down	
12	25	48	37	33	92	86	57		
				Up	down		up	Move up	
12	25	48	37	33	92	86	57		
12	25	(33	37)	48	(92	86	57)	down>=up so, swap	
Sublist further got divided into two sublists. Pivot 48 is at its proper position									
12	25	(33	37)	48	(92	86	57)		
		down	up						
12	25	(33	37)	48	(92	86	57)	Choose 33 as pivot	
			down up					Move down	
12	25	(33	37)	48	(92	86	57)	Move up	
		up	down						
12	25	(33	37)	48	(92	86	57)	down>=up, so swap	
12	25	33	(37)	48	(92	86	57)	37 is single list so automatically sorted	
Pivot 92					down		up		
12	25	33	37	48	92	86	57	Move down	
							down up		
12	25	33	37	48	92	86	57	down>=up	

12	25	33	37	48	(57	86)	92	
Pivot 57					down	up		Move down
12	25	33	37	48	57	86	92	
						down up		
12	25	33	37	48	57	86	92	Move up
					up	down		
12	25	33	37	48	57	86	92	Down>=up, so swap
12	25	33	37	48	57	(86)	92	
Sublist 86 automatically is sorted.								

Method for choosing pivot:
First element: Choose the first item in the list.
Random element: Choose any item form the list. Swap it with the first item to apply the algorithm.
Median: Pick three elements randomly and use their median as pivot.

Efficiency:
No. of comparisons:

Average case: $\boldsymbol{O(n \log n)}$
Worst case: $\boldsymbol{O(n^2)}$

No of interchanges (swaps)

Average case: $\boldsymbol{O(n \log n)}$
Worst case: $\boldsymbol{O(n^2)}$

Hence, the time complexity of QuickSort is $\boldsymbol{O(n \log n)}$ for average case and $\boldsymbol{O(n^2)}$ for worst case

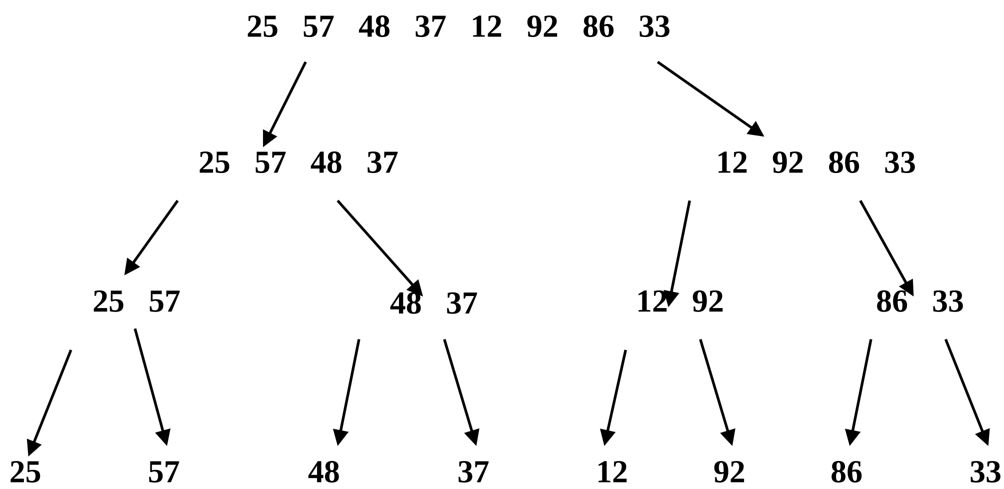
Merge Sort:
The merge sort also uses divide and conquer approach. It divides the list into sub lists. Then merge two sorted into a single list.

Algorithm outline
If size of list is greater than 1

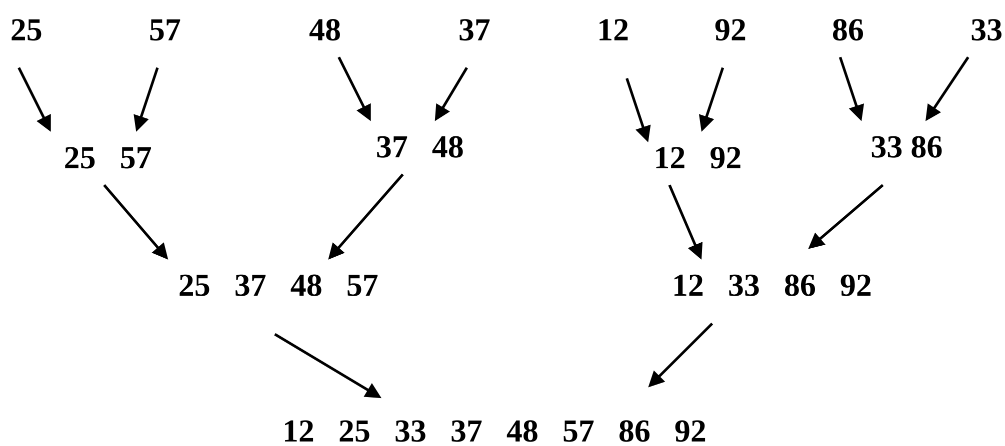
- divide the list into two sublists of sizes nearly equal as possible
- recursively sort the sublist separately.
- Merge the two sorted sublists into a single sorted list

End if

- **First Phase:** Partition the list in two equal halves, until the list size is 1



- **Second Phase:** Merge the sorted sublists



Merging description:

To simply, the algorithm merge to sorted sublist into a third list.
When finished, we copy back the third list in the original sorted halves to get the sorted list.
The basic merging algorithm takes two input sorted arrays A and B, and an output array C. We initialize the ponters Aptr, Bptr, and Cptr to point the beginning of their respective arrays. The smaller of A[Aptr] and B[Bptr] is copied to the next entry in C and appropriate pointers are advanced. When any one of the list has been finished, the remainder of the other list is copied to C.

Aptr

25	37	48	57

Bptr

12	33	86	92

Cptr

12			

Compare 25 and 12 and insert minimum(12) into array C and move Bptr and Cptr

Aptr

25	37	48	57

Bptr

12	33	86	92

Cptr

12				

Compare 25 and 33 and insert minimum(25) into array C and move Aptr and Cptr

Aptr

25	37	48	57

Bptr

12	33	86	92

Cptr

12	25			

Compare 37 and 33 and insert minimum (33) into array C and move Bptr and Cptr.
In this way the two lists are merged.

C-Code:

```
void main()
{
  clrscr();
  int n,i;
  int x[N];
  int temp[N];
  printf("\nEnter no. of elements to sort: ");
  scanf("%d",&n);
  printf("\nEnter elements to sort:\n");
  for (i = 0; i < n; i++)
  scanf("%d",&x[i]);
  //perform merge sort on array
  msort(x,temp,0,n-1);
  printf("Sorted List \n");
  for (i = 0; i < n; i++)
  printf("%d\n", x[i]);
getch();
}
```

```
void msort(int x[], int temp[], int left, int
right)
{
  int mid;
  if(left<right)
   {
    mid = (right + left) / 2;
    msort(x, temp, left, mid);
    msort(x, temp, mid+1, right);

    merge(x, temp, left, mid+1, right);
   }
}
```

```
void merge(int x[], int temp[], int left, int
mid, int right)
{
  int i, lend, no_element, tmpos;

  lend = mid - 1;
  tmpos = left;
  no_element = right - left + 1;

  while ((left <= lend) && (mid <= right))
  {
    if (x[left] <= x[mid])
    {
            temp[tmpos] = x[left];
            tmpos = tmpos + 1;
            left = left +1;
    }
    else
    {
            temp[tmpos] = x[mid];
            tmpos = tmpos + 1;
            mid = mid + 1;
    }
  }

  while (left <= lend)
  {
    temp[tmpos] = x[left];
    left = left + 1;
    tmpos = tmpos + 1;
  }
  while (mid <= right)
  {
    temp[tmpos] = x[mid];
    mid = mid + 1;
    tmpos = tmpos + 1;
  }

  for (i=0; i <= no_element; i++)
  {
    x[right] = temp[right];
    right = right - 1;
  }
}
```

Efficiency:
No. of Comparisons:
For all cases the number of comparisons is to be $O(n*\log n)$, the constant term is different for different cases. On average, it requires fewer than $n* \log n - n + 1$ comparisons.

No. of assignments:
For our implementation, it is twice the no of comparisons, merging in the temporary array and copying back to the original array which is still $O(n*\log n)$

Space Complexity:
In contrast to other sorting algorithms, we have studied, Merge Sort requires $O(n)$ extra space for the temporary memory used while merging. Algorithm has been developed for performing in-place merge in $O(n)$ time, but this would increase the no of assignments
If recursive version is used, additional space is required for the implicit stack, which is $O(\log n)$. Hence, Space complexity for Merge Sort is $O(n)$

Notes on Merge sort:
Even though the worst-case running-time of Merge Sort is $\boldsymbol{O(n\log n)}$, it is **not** an algorithm of choice for sorting contiguous lists. Merge Sort can prove superior over other sorting algorithms when used with linked lists.
Binary Tree Sort:
General idea is to create a binary search tree and access the elements either in LVR and RVL for ascending and descending order.

But, in case of imbalanced tree (right skewed and left skewed), the search time goes approximately n2. Therefore, to minimize the search time, AVL trees are maintained. This will increase performance up to nlogn. Still, BST requires some time to search and retrieve the data. After deletion of elements, there are some burden to maintain the BST property. It mean, the tree is accessed 2 times. To minimize the time for retrieval, heap is created. In heap sort, the heap creation takes time, but the retrieval takes no time.

Heap Sort

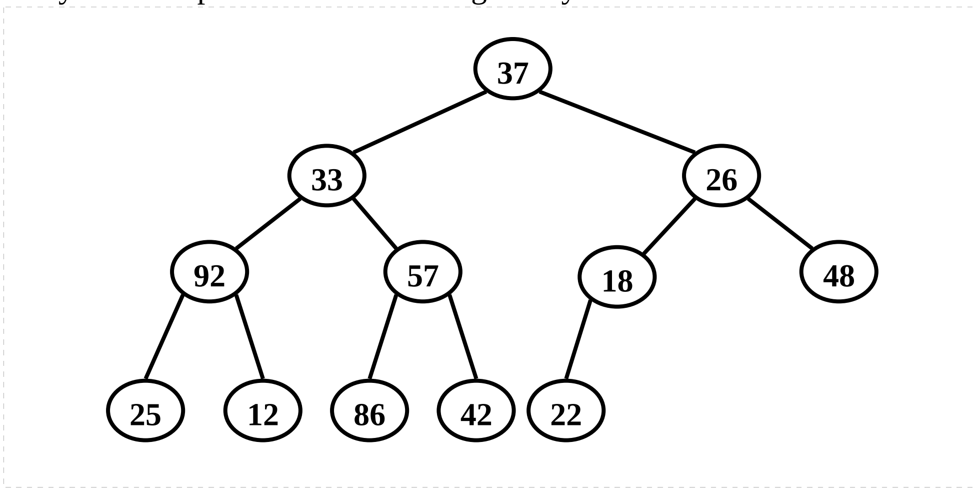
- The heap sort algorithm sorts by representing its input as a heap in the array
- Two phases in sorting
  1. Converts the array representation of the tree into a heap
  2. Repeatedly moves the largest element to the last position by swapping the first element with the last element and adjusts the heap property at each stage in the remaining elements

First Phase:
1. The entries in the array being sorted are interpreted as a binary tree in array implementation
2. Tree with only one node automatically satisfies the heap property. So, we don't need to worry about any of the leaves.

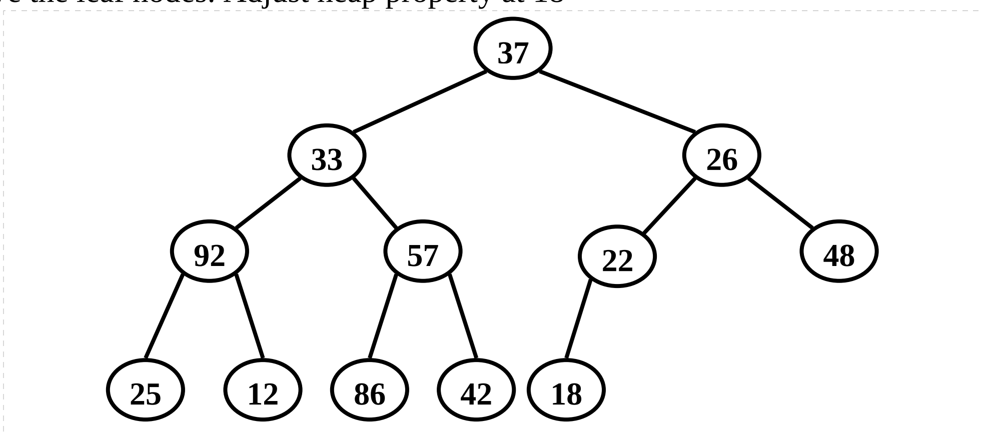
3. Start from the level above the leaf nodes, and work out backward towards the root.
Lets take an example for tracing following elements.

37	**33**	**26**	**92**	**57**	**18**	**48**	**25**	**12**	**86**	**42**	**22**

This array can be represented as following binary tree:

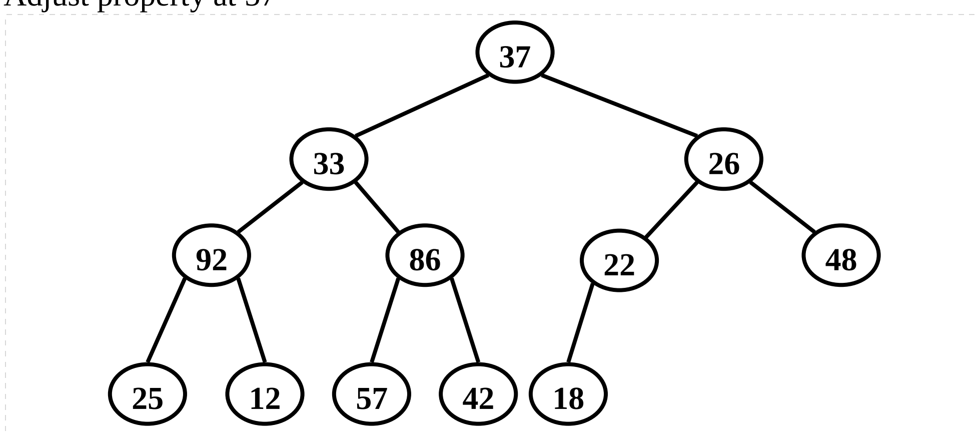


This is not a heap. So, adjust to convert it to max heap as follows:
Leave the leaf nodes. Adjust heap property at 18



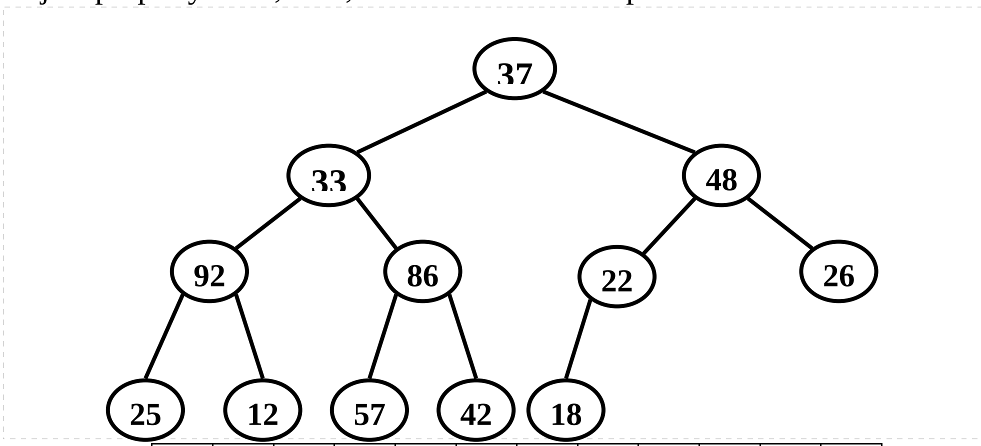
37	33	26	92	57	**22**	48	25	12	86	42	**18**

Adjust property at 57



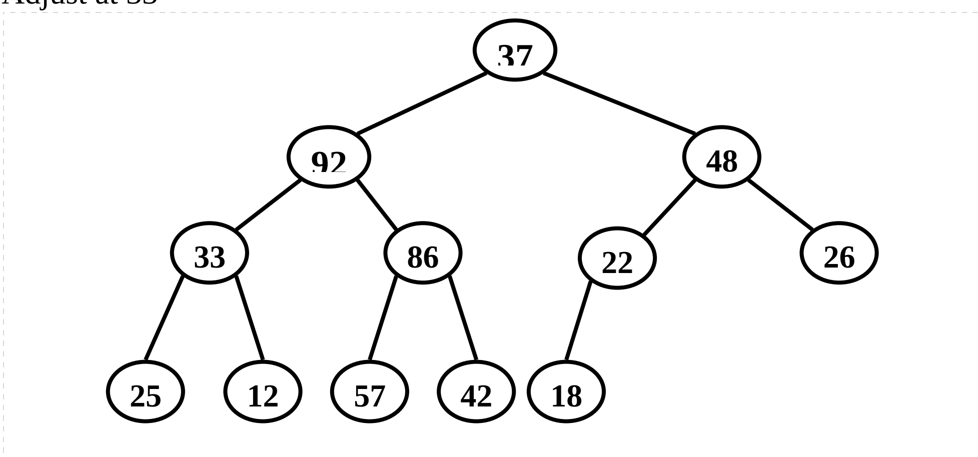
37	33	26	92	**86**	22	48	25	12	**57**	42	18

At 92 the subtree already satifies the heap property, so it will remain as it is.
Adjust property at 26, here, 48 should be come up.



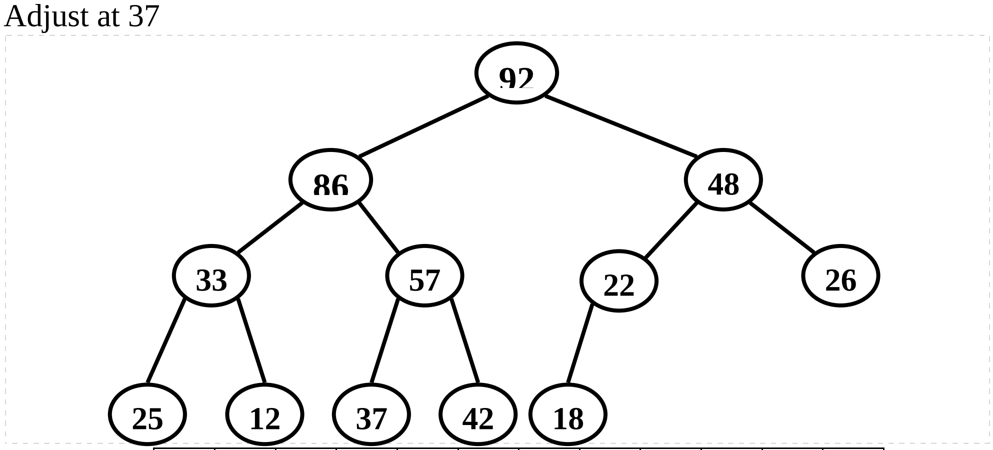
37	33	**48**	92	86	22	**26**	25	12	57	42	18

Adjust at 33



37	**92**	48	**33**	86	22	26	25	12	57	42	18

Adjust at 37



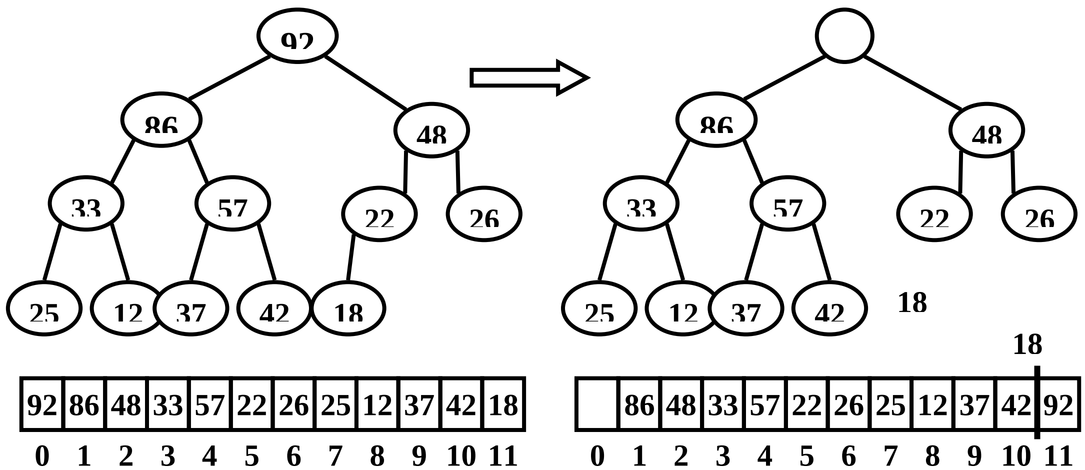
92	86	48	33	57	22	26	25	12	**37**	42	18

Second Phase:

- Note that the root (the first element of the array) has the largest key
- Repeat these steps until the size of heap becomes 1
    1. Move the largest key at root to the last position of the heap, replacing an entry *x* currently at the last position
    2. Decrease a counter *i* that keeps track of the size of the heap, thereby excluding the largest key from further sorting
    3. The element x may not belong to the root of the heap, so insert x into the proper position to restore the heap property

Move the largest element, 92, to the last position of the heap, thus replacing 18

Now we must find a way to put 18 in its proper position, thus restoring the heap property

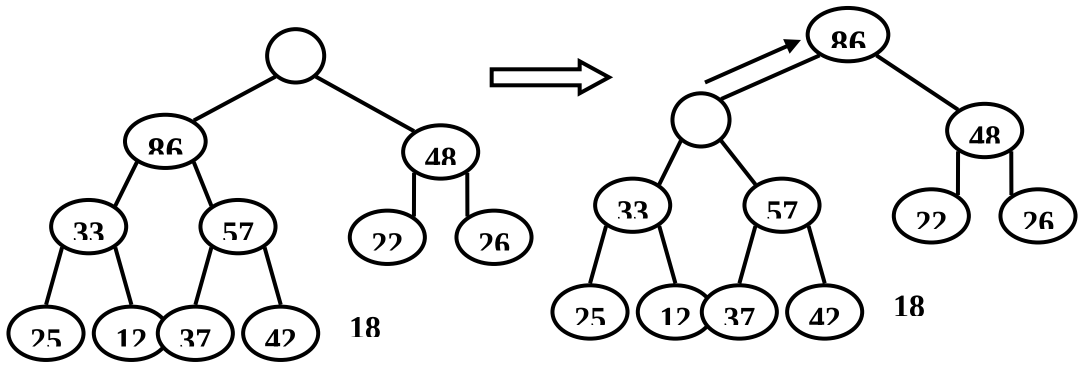


Adjusting the Heap Properties:

- When the last element, ***x***, is replaced by the largest element at the root, a hole is created at the root and the heap size becomes smaller by **1**
- We must move ***x*** somewhere to restore the heap property

1. First we look if ***x*** can be placed in the hole, by looking at the two children of that hole
2. If ***x*** belongs to the hole, then we put ***x*** there and we are done
3. Else we slide the larger of the two children to the hole, thus pushing the hole down one level
4. We repeat this process on the subtree until ***x*** can be placed in the hole or there are no children
5. Thus, our action is to place ***x*** in its correct spot along a path from the root containing minimum children.

Delete 92

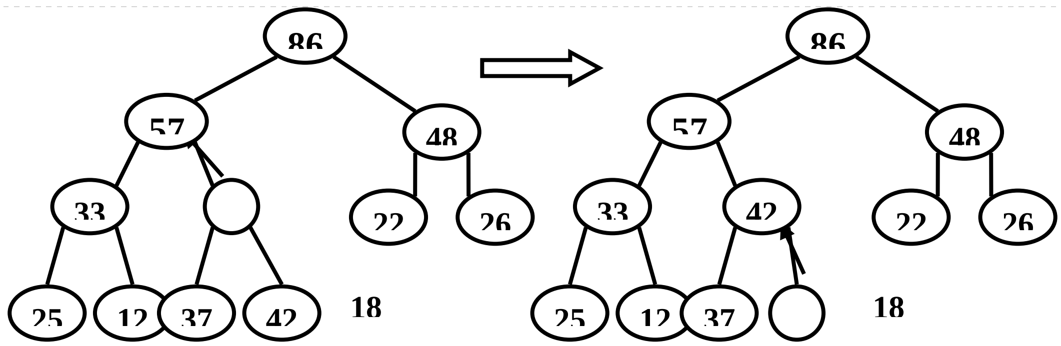


18

	86	48	33	57	22	26	25	12	37	42	92
0	1	2	3	4	5	6	7	8	9	10	11

18

86		48	33	57	22	26	25	12	37	42	92
0	1	2	3	4	5	6	7	8	9	10	11



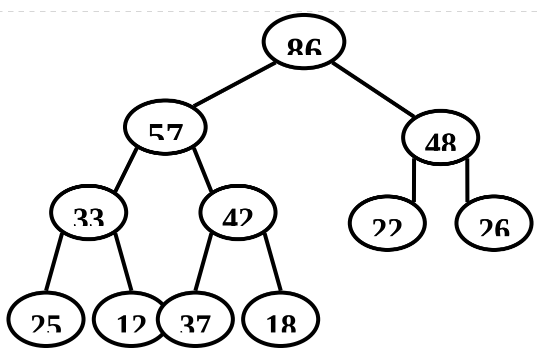
18

86	57	48	33		22	26	25	12	37	42	92
0	1	2	3	4	5	6	7	8	9	10	11

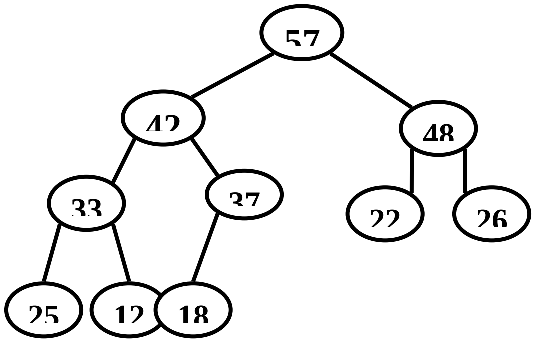
18

86	57	48	33	42	22	26	25	12	37		92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 86

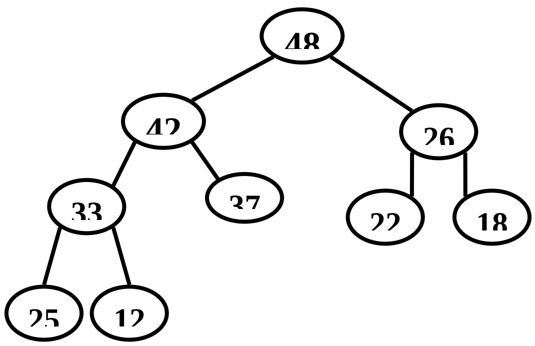


86	57	48	33	42	22	26	25	12	37	18	92
0	1	2	3	4	5	6	7	8	9	10	11



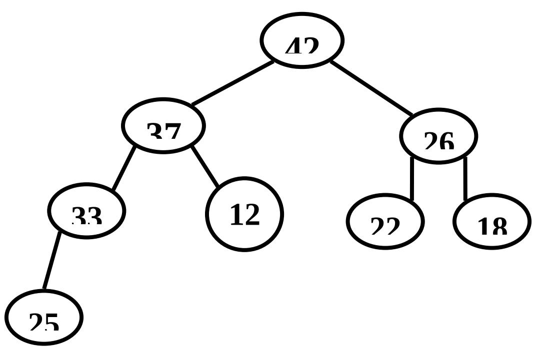
57	42	48	33	37	22	26	25	12	18	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 57



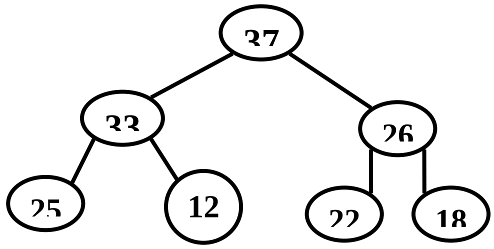
48	42	26	33	37	22	18	25	12	57	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 48



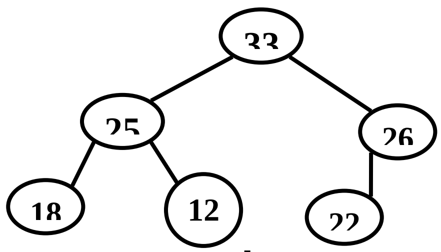
42	37	26	33	12	22	18	25	48	57	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 42



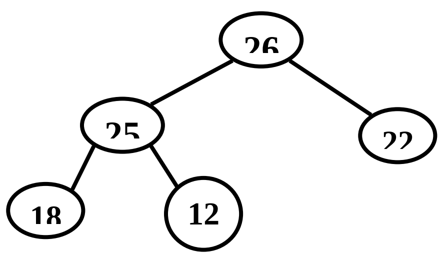
37	33	26	25	12	22	18	42	48	57	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 37



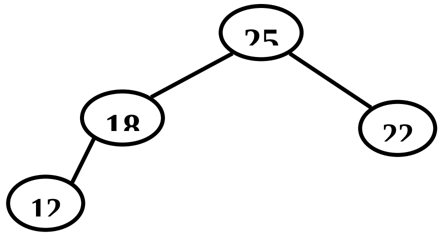
33	25	26	18	12	22	37	42	48	57	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 33



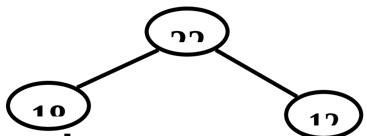
26	25	22	18	12	33	37	42	48	57	86	92
0	1	2	3	4	5	6	7	8	9	10	11

Delete 26



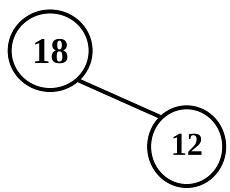
25	18	22	12	26	33	37	42	48	57	86	92
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Delete 25



22	18	12	25	26	33	37	42	48	57	86	92
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Delete 22



18	12	22	25	26	33	37	42	48	57	86	92
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Delete 18

12	18	22	25	26	33	37	42	48	57	86	92
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Delete 12

12	18	22	25	26	33	37	42	48	57	86	92
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Finally, the data are sorted

Heap Sort Efficiency

- No of comparisons and assignments
  - Worst-case: $O(n\log n)$
  - Average case: $O(n\log n)$

- Hence time complexity of heap sort is $O(n\log n)$ for both worst case and average case and space complexity is $O(1)$. In average case, it is not as efficient as quick sort, however, it is far superior to quick sort in worse case. Generally, heap sort is used for large amount of data.

Heap as Priority Queue:

- A ***priority queue*** is a data structure with the following primitive operations
  - Insert an item
  - Remove the item having the largest (or smallest) key
- Implementations
  - Use a sorted contiguous list, removal takes $O(1)$ but insertion takes $O(n)$
  - Use an unsorted list, insertion takes $O(1)$ but removal takes $O(n)$

Efficiency Priority Queue:

- Consider the properties of heap:
  - The item with largest key is on the top and can be removed immediately. However it will take time $O(\log n)$ to restore the heap property for remaining keys
  - For insertion we shift the new item from down to up which also takes $O(\log n)$

- Hence, implementation of a priority queue as a heap proves advantageous for large *n*
  - It efficiently represents in contiguous storage and is guaranteed to require only logarithmic time for both insertions and deletions

Shell Sort

Significant improvement on simple insertion sort can be achieved by using shell sort (or diminishing increment sort). This method separates original file into subfiles. These subfiles contain every k^{th} element of the original file. The value of k is called an increment. Eg. If k=5, then subfile consists of x[0], x[5], x[10],… is first sorted.

After the first k subfiles are sorted (usually by simply insertion), a new smaller value of k is chosen and the file is again partitioned into a new set of subfiles. Each of these larger subfiles is sorted and the process is repeated yet again, until eventually the value of the k is set to 1.

The decreasing sequence of increments can
Either be fixed at the start of the entire process. The last value must be 1

Or take the first increment to hk = floor(N/2) and hk = floor(hk / 2) until hk = 1. hk = subsequent increment.
Tracing for following numbers:

81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15

Let hk = 13 / 2 = 6, so here increment is 6, The shell sort will be sub divided into 6 sub files.

Sub files			**Sorted sub files**		
81	17	15	15	17	81
94	95		94	95	
11	28		11	28	
96	58		58	96	
12	41		12	41	
35	75		35	75	

After 1st iteration, the list will look like:
15 94 11 58 12 35 17 95 28 96 41 75 81

Now, in second iteration hk = floor(hk/2) = floor(6/2) = 3

Subfiles					**Sorted subfiles**				
15	58	17	96	81	15	17	58	81	96
94	12	95	41		12	41	94	95	
11	35	28	75		11	28	35	75	

After 2nd iteration, the list will be as follows:
15 12 11 17 41 28 58 94 35 81 95 75 96

Now, hk = floor(hk/2) = 3 / 2 = 1
15 12 11 17 41 28 58 94 35 81 95 75 96

Now, using simple insertion sort we get,

<u>15 12</u> 11 17 41 28 58 94 35 81 95 75 96

<u>12 15 11</u> 17 41 28 58 94 35 81 95 75 96

<u>11 12 15 17</u> 41 28 58 94 35 81 95 75 96

<u>11 12 15 17 41</u> 28 58 94 35 81 95 75 96

11	12	15	17	41	28	58	94	35	81	95	75	96
11	12	15	17	28	41	58	94	35	81	95	75	96
11	12	15	17	28	41	58	94	35	81	95	75	96
11	12	15	17	28	41	58	94	35	81	95	75	96
11	12	15	17	35	28	41	58	94	81	95	75	96
11	12	15	17	35	28	41	58	81	94	95	75	96
11	12	15	17	35	28	41	58	81	94	95	75	96
11	12	15	17	35	28	41	58	75	81	94	95	96
11	12	15	17	35	28	41	58	75	81	94	95	96

Hence, the list is finally sorted.

Efficiency
Worse case : $O(n^2)$
Average case : $O(n(\log n)^2)$ (if appropriate increment sequent is used)

Radix Sort
The sorting is based on the values of the actual digits in the positional representations of the numbers being sorted.

Process
Beginning with the least-significant digit and ending with the most-significant digit, perform the following action,
Take each number in the order in which it appears in the file and place it into one of the ten queues, depending on the value of the digit currently being processed.

Then restore each queue to the original file starting with the queue of numbers with a 0 digit and ending with the queue of numbers with a 9 digit.

When these actions have been performed for each digit, starting with the least significant digit and ending with the most significant, the file is sorted.

Tracing example:
We have **64, 8, 216, 512, 27, 729, 0, 1, 343, 125**
First Pass

	0	**1**	**512**	**343**	**64**	**125**	**216**	**27**	**8**	**729**
no%10	0	1	2	3	4	5	6	7	8	9

Second Pass

	8		**729**							
	1	**216**	**27**							
	0	**512**	**125**		**343**		**64**			
(no/10)%10	0	1	2	3	4	5	6	7	8	9

Third Pass

	64									
	27									
	8									
	1									
	0	**125**	**216**	**343**		**512**		**729**		
(no/100)%10	0	1	2	3	4	5	6	7	8	9

Finally, we have, **0, 1, 8, 27, 34, 125, 216, 343, 512, 729**
Here, the no. of passes equals maximum number of digits in the given numbers to be sorted.
Efficiency : **O(n.logn)**

Comparison table:

Algorithms	**Worse Case**	**Average Case**
Bubble Sort	**$O(n^2)$**	**$O(n^2)$**
Quick Sort	**$O(n^2)$**	**O(n.logn)**
Insertion Sort	**$O(n^2)$**	**$O(n^2)$**
Selection Sort	**$O(n^2)$**	**$O(n^2)$**
Merge Sort	**O(n.logn)**	**O(n.logn)**
Heap Sort	**O(n.logn)**	**O(n.logn)**
Radis Sort	**O(n.logn)**	**O(n.logn)**

Selecting a sort algorithm:

Algorithms	**Comments**
Bubble Sort	**Good for small n usually less than 10**
Quick Sort	**Excellent for virtual memory environment**
Insertion Sort	**Good for almost sorted records**
Selection Sort	**Good for partially sorted data and small ‘n’**
Merge Sort	**Good for external file sorting**
Heap Sort	**As efficient as quick sort in average case and far superior to quick sort in the worse case**
Radix Sort	**Good when number of digits(letters) are less**