



Unit 6 Sorting

Hours 5

Marks: 7

Sorting

- consider a list of values:

2 ,4 ,6 ,8 ,9 ,1 ,22 ,4 ,77 ,8 ,9

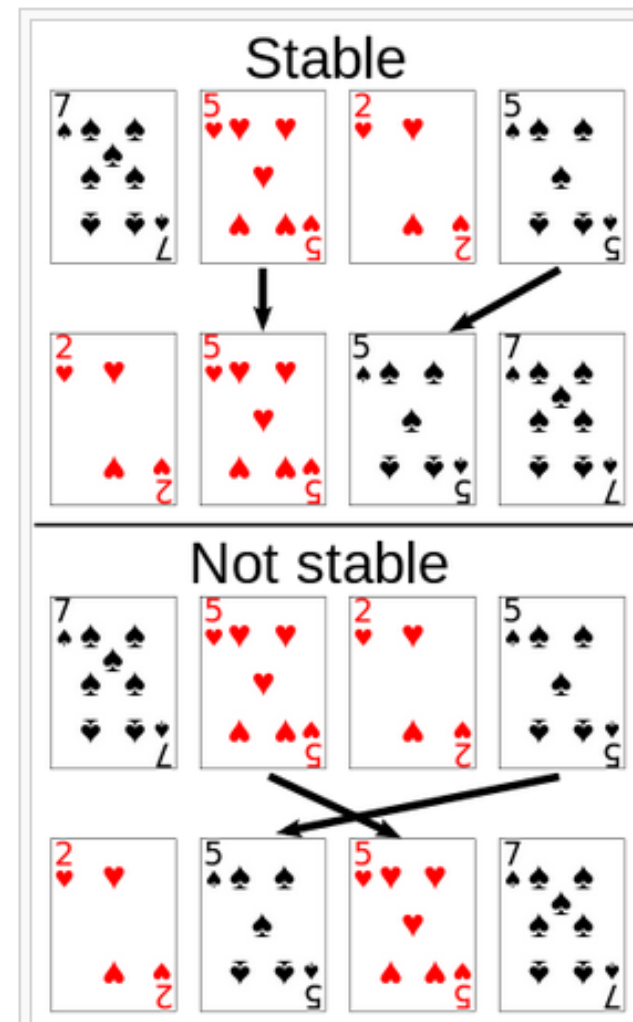
- After sorting the values:

1, 2, 4, 4, 6, 8, 8,9 , 9 , 22, 77

- In brief the **sorting** is a **process of arranging the items in a list in some order** that is either ascending or descending order.

Classification of Sorting

- Internal:
 - If the records that it is sorting are in main memory.
- External:
 - If the records are in auxiliary storage.
- Stable:
 - A sorting algorithm is stable if two elements that are equal **remain in the same relative position** after sorting is completed.
 - A stable sort keeps records with the same key in the same relative order that they were in before the sort.



An example of stable sort on playing cards. When the cards are sorted by rank with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.

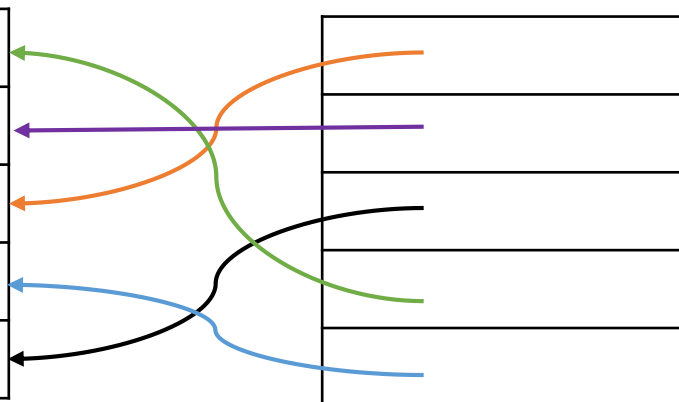
Sorting Techniques

- Actual Record Sorting
 - Actual data will be sorted.
- Sorting by Address
 - Actual data are prohibited to move from their location.
 - Auxiliary Pointer table is maintained for sorting accordingly.

Original Pointer Table

	Record 1	4	DDD
	Record 2	2	BBB
	Record 3	1	AAA
	Record 4	5	EEE
	Record 5	3	CCC

Sorted Pointer Table

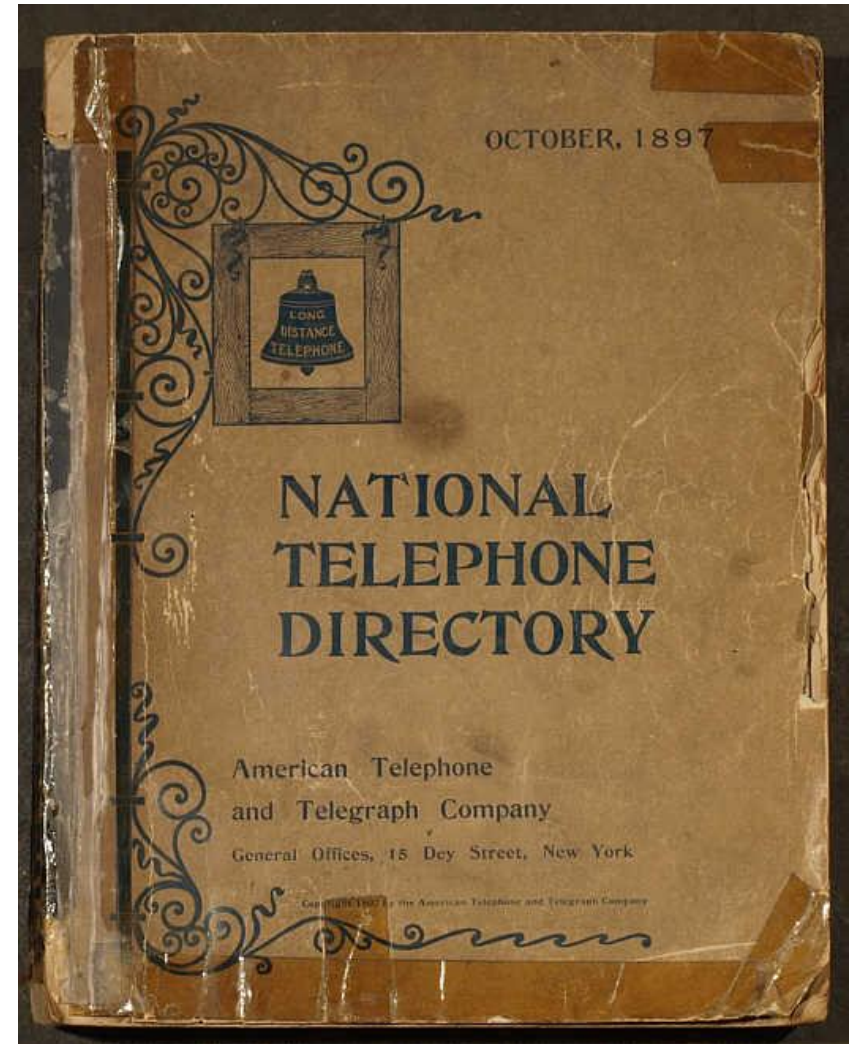


Sorting types:

- Adaptive Search:
 - takes advantage of the existing order of the input to try to achieve better times, so that the time taken by the algorithm to sort is a smoothly growing function of the size of the sequence *and* the disorder in the sequence.
 - In other words, the more presorted the input is, the faster it should be sorted.
- In-place Sort:
 - In-place algorithm updates input sequence only through replacement or swapping of elements. The input is usually overwritten by the output as the algorithm executes. However a small amount of extra storage space is allowed for auxiliary variables.

Sorting Algorithms

- Bubble Sort
- Insertion
- Selection
- Quick
- Merge
- Comparison and Efficiency of sorting



Bubble Sort

- The basic idea of this sort is to pass through the array sequentially several times.
- Each pass consists of comparing each element in the array with its successor

(for example $a[i]$ with $a[i + 1]$) and
interchanging the two elements if they are not in the proper order

Bubble Sort

5	1	6	2	4	3
---	---	---	---	---	---


Lets take this Array.

<u>5</u>	<u>1</u>	6	2	4	3
1	5	<u>6</u>	<u>2</u>	4	3
1	5	2	<u>6</u>	<u>4</u>	3
1	5	2	4	<u>6</u>	<u>3</u>
1	5	2	4	3	6

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

First pass

									
54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Algorithm

```
BubbleSort(A, n)
{
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

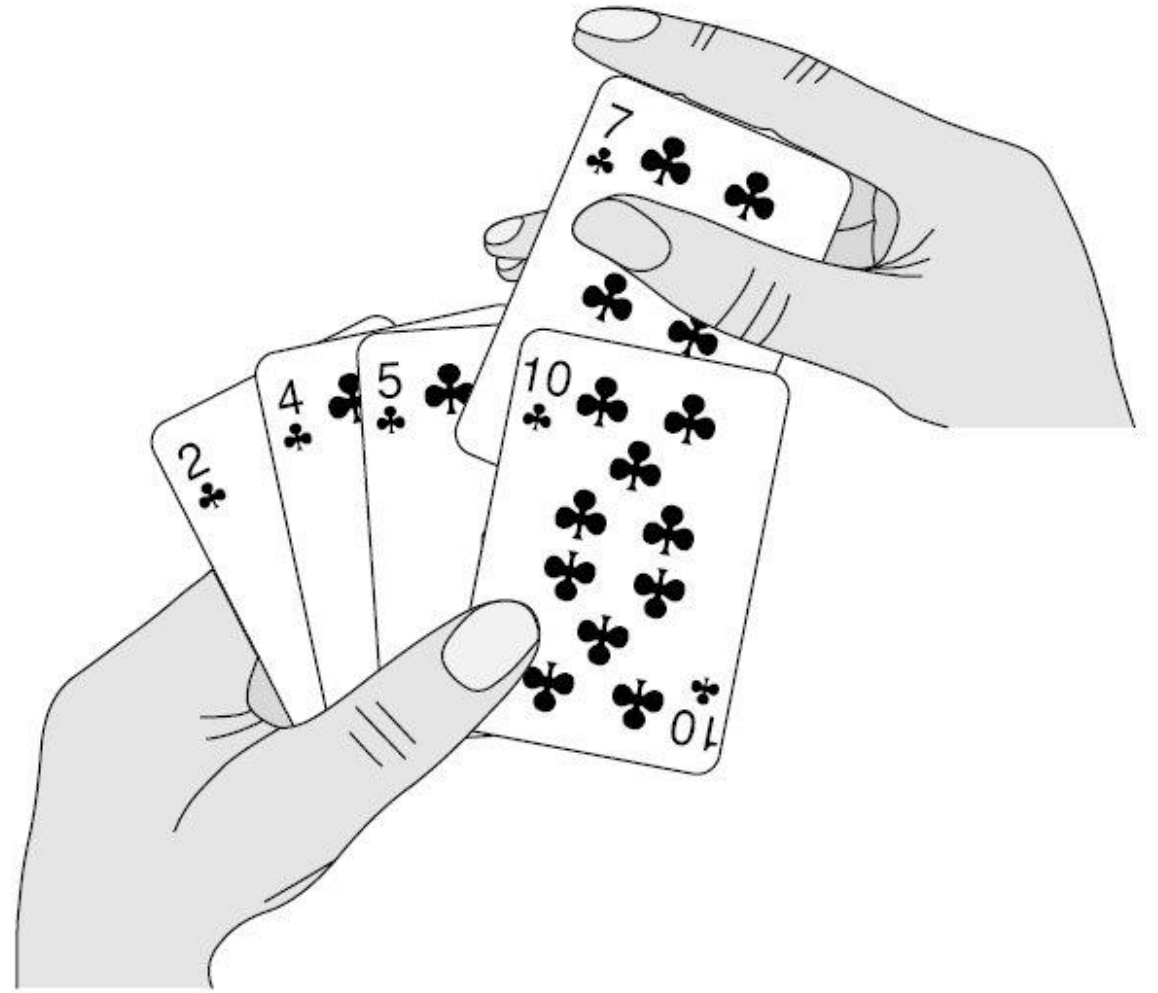
Time Complexity:

- Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:
- Time complexity = $(n-1) + (n-2) + (n-3) + \dots + 2 + 1 = O(n \cdot n) = O(n^2)$
 - We have formula for arithmetic series as
$$1+2+3+4+5+\dots+(n-2)+(n-1)+n$$
$$\frac{n}{2}\{2a+(n-1)d\} = \frac{n(n-1)}{2} = \frac{(n^2 - n)}{2}$$
$$a = \text{starting element}$$
$$d = \text{common difference}$$

Space Complexity:

- Since no extra space besides 3 variables is needed for sorting
- Space complexity = $O(n)$

Insertion Sort



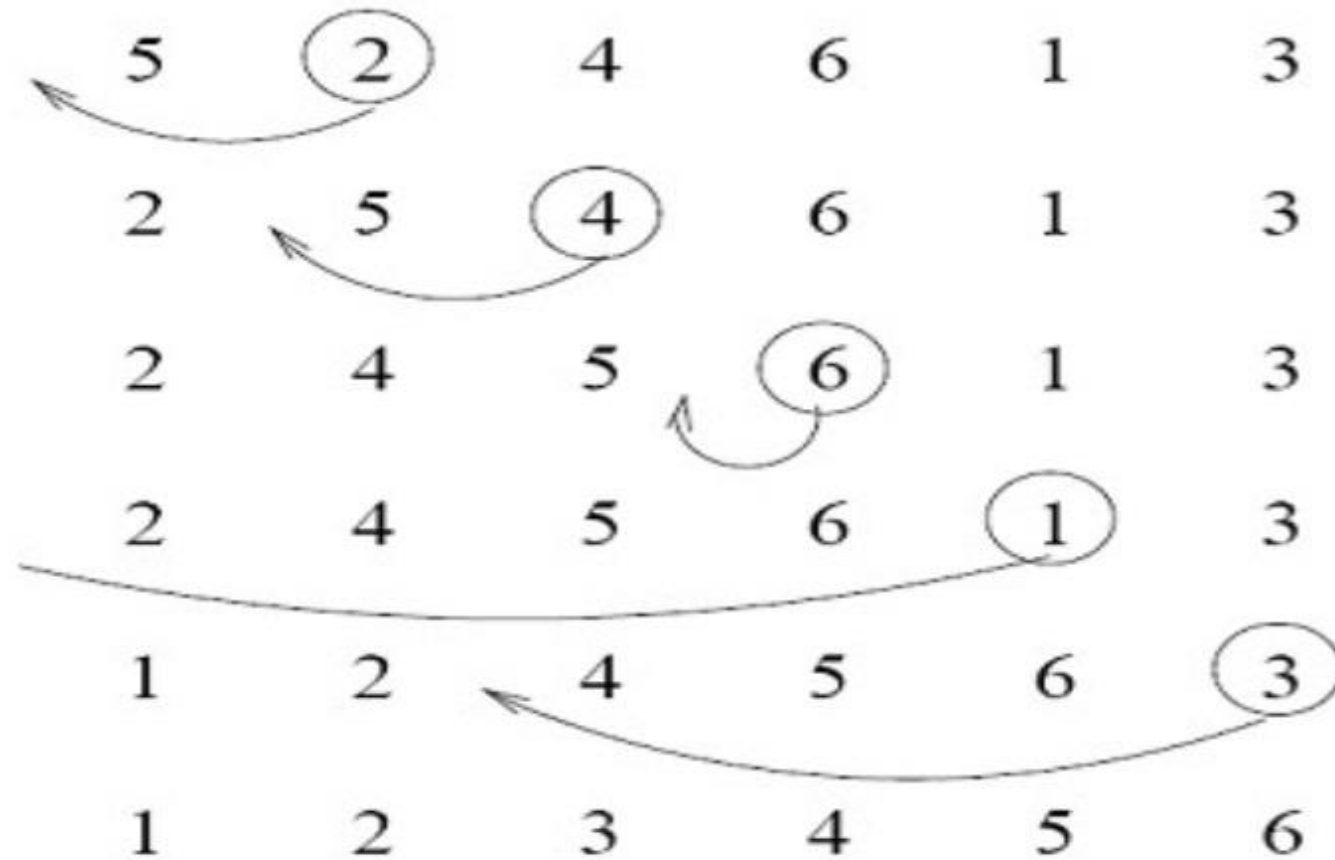
Like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted.

Insertion sort

- Suppose an array $a[n]$ with n elements. The insertion sort works as follows:
- pass 1: $a[0]$ by itself is trivially sorted.
- Pass 2: $a[1]$ is inserted either before or after $a[0]$ so that $a[0], a[1]$ is sorted.
- Pass 3: $a[2]$ is inserted into its proper place in $a[0], a[1]$ that is before $a[0]$, between $a[0]$ and $a[1]$, or after $a[1]$ so that $a[0], a[1], a[2]$ is sorted.
-
- pass N : $a[n-1]$ is inserted into its proper place in $a[0], a[1], a[2], \dots, a[n-2]$ so that
 $a[0], a[1], a[2], \dots, a[n-1]$ is sorted with n elements.

Insertion sort

Example:



Algorithm example of Insertion Sort

```
Void Insetsort(int x[], int n){  
    int i, k, y;  
    for(k = 1; k < n; k++) {  
        y = x[k];  
        for(i = k-1; i >= 0 && y < x[i]; i--){  
            x[i+1] = x[i];  
        }  
        x[i+1] = y;  
    }  
}
```

Time Complexity:

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when $i=1$, 2 times when $i=2$... and $n-1$ times when $i=n-1$:

$$\begin{aligned}\text{Time complexity} &= 1 + 2 + 3 + \dots + (n-2) + (n-1) \\ &= O(n^2)\end{aligned}$$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when $i=1$, 1 times when $i=2$... and 1 times when $i=n-1$:

$$\begin{aligned}\text{Time complexity} &= 1 + 2 + 3 + \dots + 1 + 1 \\ &= O(n)\end{aligned}$$

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

$$\text{Space complexity} = O(1)$$

Selection Sort

- The algorithm divides the input list into two parts:
 - 1. The sublist of items **already sorted**,
 - which is built up from left to right at the front (left) of the list, and
 - 2. Sublist of items **remaining to be sorted** that occupy the rest of the list.
- Initially,
 - the sorted sublist is empty
 - the unsorted sublist is the entire input list.
- The algorithm **proceeds by finding the smallest** (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Selection Sort

- 64 25 12 22 11 // this is the initial, starting state of the array
//Find smallest item from unsorted list and add to sorted list
- **11** 25 12 22 64 // sorted sublist = {11}
- **11 12** 25 22 64 // sorted sublist = {11, 12}
- **11 12 22** 25 64 // sorted sublist = {11, 12, 22}
- **11 12 22 25** 64 // sorted sublist = {11, 12, 22, 25}
- **11 12 22 25 64** // sorted sublist = {11, 12, 22, 25, 64}

```
/* a[0] to a[n-1] is the array to sort */  
int i, j;  
for (j = 0; j < n-1; j++) {  
    /* assume the min is the first element */  
    int iMin = j;  
    for ( i = j+1; i < n; i++) {  
        /* if this element is less, then it is the new minimum */  
        if (a[i] < a[iMin]) {  
            /* found new minimum; remember its index */  
            iMin = i;  
        }  
    }  
    if(iMin != j) {  
        swap(a[j], a[iMin]);  
    }  
}
```

Performance Analysis

- Selecting the lowest element requires scanning all n elements
- (this takes $n - 1$ comparisons) and then swapping it into the first position.
- Finding the next lowest element requires scanning the remaining $n - 1$ elements and swapping and so on,
- ie $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 \in O(n^2)$ comparisons
 - Each of these scans requires one swap for $n - 1$ elements (the final element is already in place).

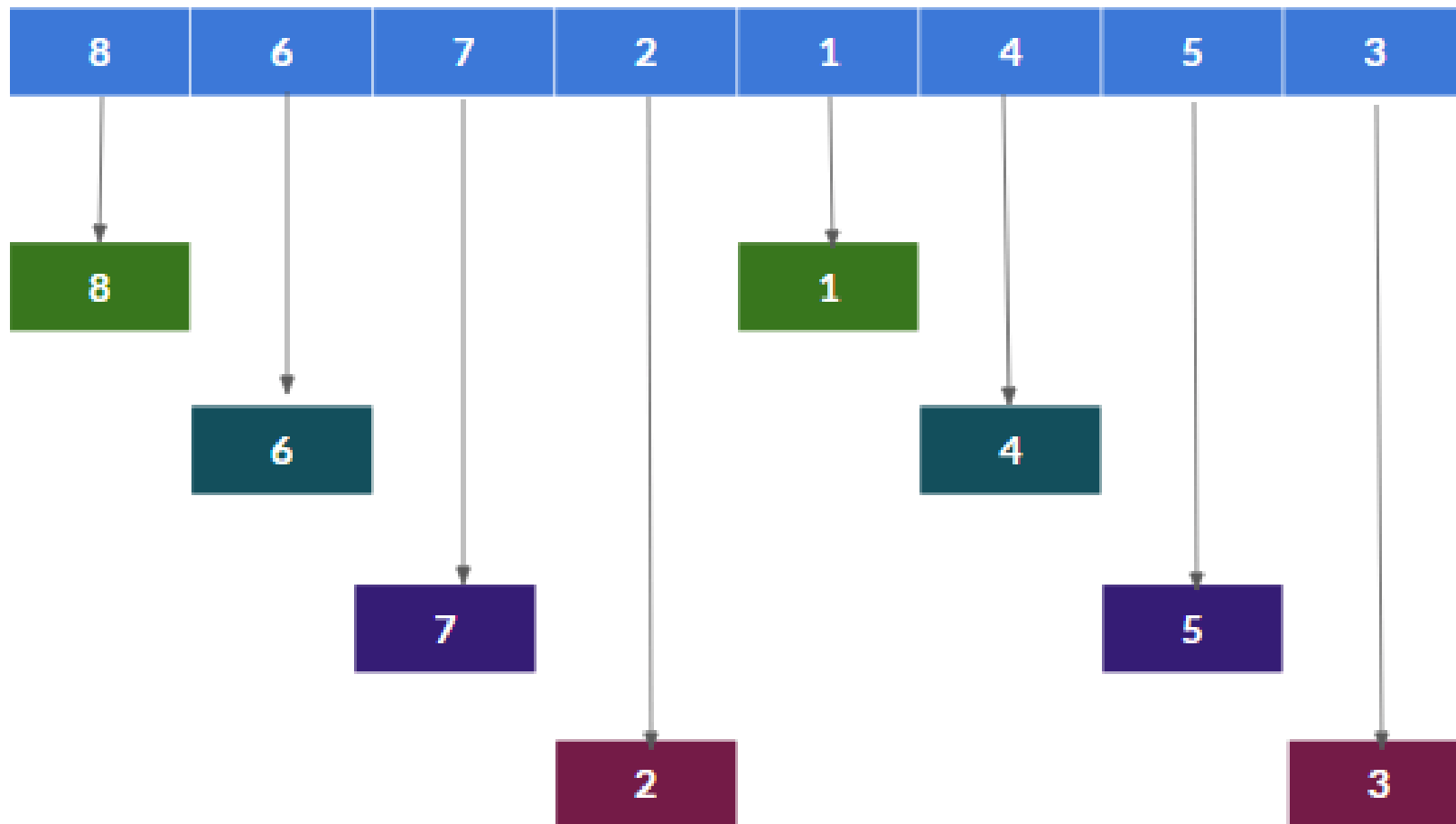
Shell Sort

- [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 the exchange of far items.
- In Shell sort, 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 are sorted.

Shell Sort Algorithm

- **Step 1)** Initialize the interval value, $h = n/2$. (n is the size of the array)
- **Step 2)** Put all the elements within a distance of the interval h in a sub-list.
- **Step 3)** Sort those sub-lists using insertion sort.
- **Step 4)** Set new interval, $h=h/2$.
- **Step 5)** If $h>0$, go to step 2. Else go to step 6.
- **Step 6)** The resultant output will be the sorted array.

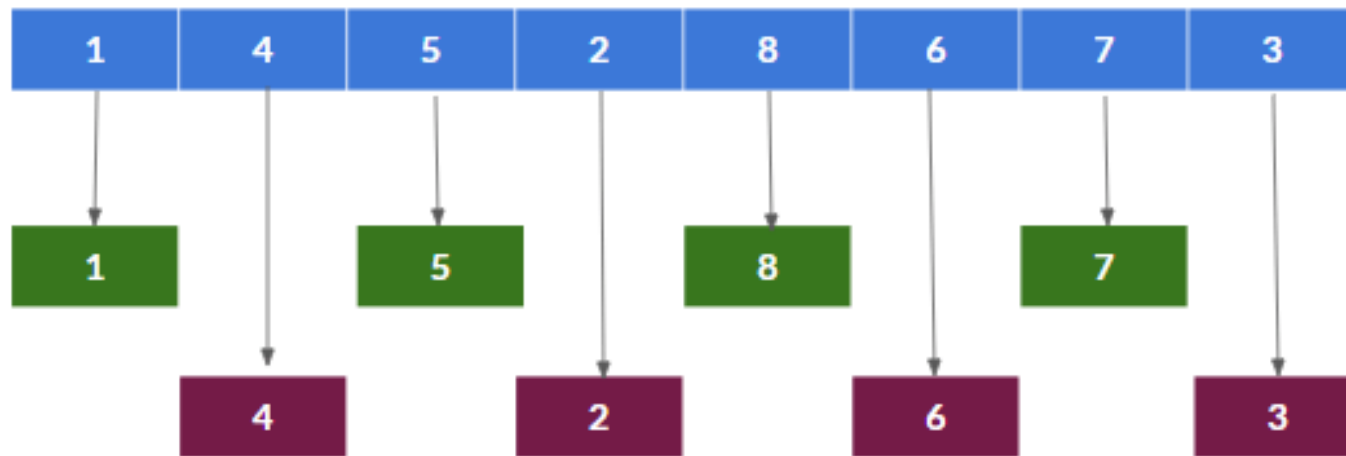
- **Step 1)** Here, the array size is 8. Thus, the interval value will be $h = 8/2$ or 4.
- **Step 2)** Now, we will store all the elements at a distance of 4. For our case, the sublists are- {8, 1}, {6, 4}, {7, 5}, {2, 3}.



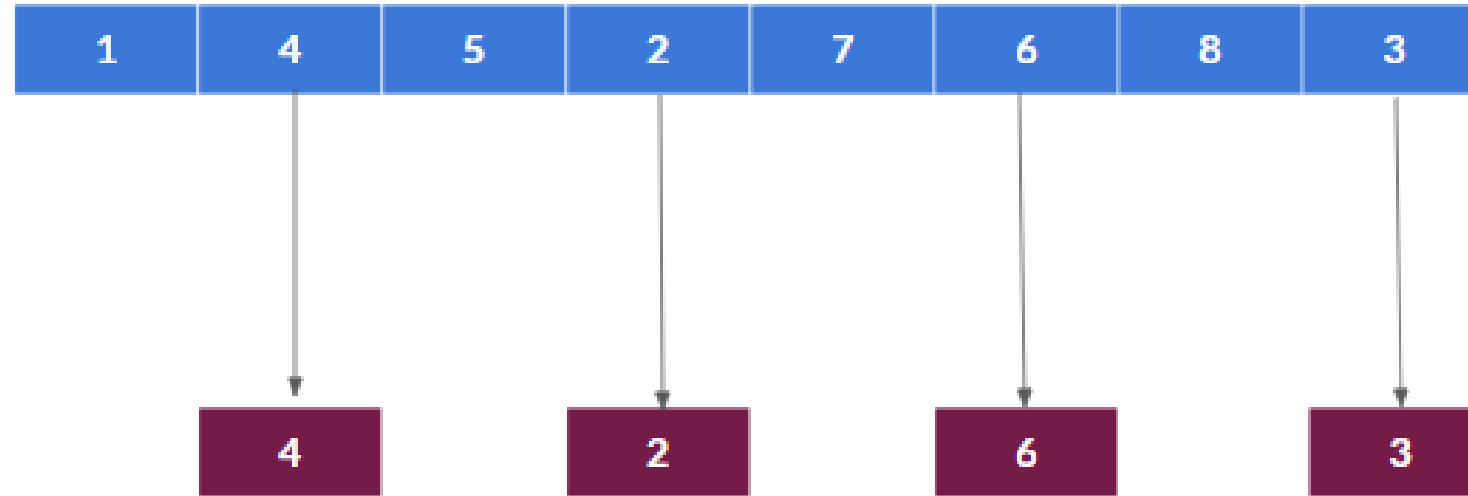
- **Step 3)** Now, those sublists will be sorted using insertion sort, where a temporary variable is used to compare both numbers and sort accordingly.
- The array would be like the following after swapping operations-

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

- **Step 4)** Now, we will decrease the initial value of the interval. The new interval will be $h=h/2$ or $4/2$ or 2 .
- **Step 5)** As $2>0$, we will go to step 2 to store all the elements at a distance of 2. For this time, the sublists are-
- $\{1, 5, 8, 7\}, \{4, 2, 6, 3\}$



- Then these sublists will be sorted using insertion sort. After comparing and swapping the first sublist, the array would be the following.



- After sorting the second sublist, the original array will be:



- Then again, the interval will be decreased. Now the interval will be $h=h/2$ or 2/1 or 1. Hence we will use the insertion sort algorithm to sort the modified array.
- Following is the step-by-step procedural depiction of insertion sort.



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

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

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

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

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

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

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

- The interval is again divided by 2. By this time, the interval becomes 0. So we go to step 6.
- **Step 6)** As the interval is 0, the array is sorted by this time. The sorted array is-

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

Source

- <https://www.guru99.com/shell-sort-algorithm.html>

Quick Sort or partition-exchange sort

- In practice this is the **fastest sorting method**. It possesses very good average case complexity among all the sorting algorithms.
- Quicksort is a divide and conquer algorithm.
 - Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
- The steps are:
 - **Pick an element, called a *pivot***, from the array.
 - *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the ***partition operation***.
 - Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

- **Pivot**
- 54 Left 26 93 17 77 31 44 20 Rt 56 26 < 54:true move left
- 54 26 93 17 77 31 44 20 56 93 < 54:false, Check right..56 >54: T move right
- 54 26 93 17 77 31 44 20 56 20 > 54:f swap 93 and 20
- 54 26 20 17 77 31 44 93 56 continue moving left and right
- 54 26 20 17 77 31 44 93 56 swap 77 and 44
- 54 26 20 17 44 31 77 93 56 move left and right
- 54 26 20 17 44 31 77 93 56 exchange 54 and 31
- 31 26 20 17 44 54 77 93 56
- Now we have found actual position of 54. We repeat recursively same process for two sub list in left side and right side of 54.

Quicksort

- To sort $a[\text{left} \dots \text{right}]$:
 1. if $\text{left} < \text{right}$:
 - 1.1. Partition $a[\text{left} \dots \text{right}]$ such that:
 - all $a[\text{left} \dots p-1]$ are less than $a[p]$, and
 - all $a[p+1 \dots \text{right}]$ are $\geq a[p]$
 - 1.2. Quicksort $a[\text{left} \dots p-1]$
 - 1.3. Quicksort $a[p+1 \dots \text{right}]$
 2. Terminate

```
Partition(A,l,r)
{
    x =l;
    y =r ;
    p = A[l];
    while(x<y)
    {
        while(A[x] <= p)
            x++;
        while(A[y] >p)
            y--;
        if(x<y)
            swap(A[x],A[y]);
    }
    A[l] = A[y];
    A[y] = p;
    return y; //return position of pivot
}
```

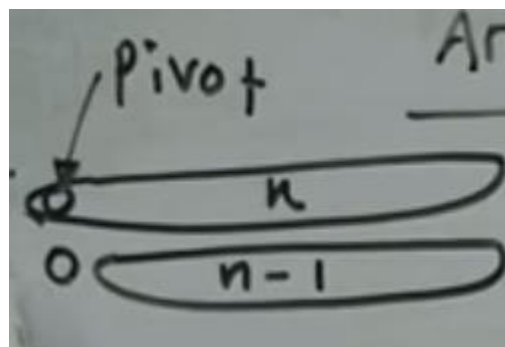
Algorithm:

QuickSort(A,l,r)

```
{  
    if(l<r)  
    {  
        p = Partition(A,l,r);  
        QuickSort(A,l,p-1);  
        QuickSort(A,p+1,r);  
    }  
}
```

Properties:

- Not -stable Sorting
- In-place sorting ($O(\log n)$ use extra space for recursion)
- Not Adaptive
- $O(n^2)$ time, typically $O(n \log n)$



Analysis of Quick Sort (worst case)

$$T(n) = T(n-1) + n \quad \text{--- (1)}$$

substitute $(n-1)$ in place of n in eqⁿ (1)

$$\therefore T(n-1) = T(n-2) + (n-1) \quad \text{--- (2)}$$

Now substitute eqⁿ (2) in eqⁿ (1)

$$\therefore T(n) = T(n-2) + (n-1) + n \quad \text{--- (3)}$$

Derive the value of $T(n-2)$ from eqⁿ (1)

$$T(n-2) = T(n-3) + (n-2) \quad \text{--- (4)}$$

substitute eqⁿ (4) in eqⁿ (3)

$$\therefore T(n) = T(n-3) + (n-2) + (n-1) + n$$

$$T(n) = T(1) + 2 + 3 + 4 + \dots + (n-1) + n$$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$$

Best case

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \\ T\left(\frac{n}{2}\right) &\Rightarrow 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \\ \Rightarrow T(n) &= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\ &\Rightarrow 2^2 T\left(\frac{n}{2^2}\right) + 2n \\ &\Rightarrow 2^2 \left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n \\ &\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + n + 2n \\ &\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + 3n \end{aligned}$$

$T(1) = 1$

$$\begin{aligned} T(n) &= 2^K T\left(\frac{n}{2^K}\right) + Kn \\ \Rightarrow \text{let } 2^K &= n \\ \text{hence } K \log_2 2 &= \log_2 n \\ \Rightarrow K &= \log n \\ T(n) &\Rightarrow n T\left(\frac{n}{n}\right) + n \times \log n \\ &\Rightarrow n T(1) + n \log n \\ &\Rightarrow n + n \log n \\ &\Rightarrow O(n \log n) \end{aligned}$$

Analysis of Quick Sort

- **Best Case:**

Divides the array into two partitions of equal size, therefore

$T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,

$T(n) = O(n \log n)$

- **Worst case:**

when one partition contains the $n-1$ elements and another partition contains only one element. Therefore its recurrence relation is:

$T(n) = T(n-1) + O(n)$, Solving this recurrence we get, $T(n) = O(n^2)$

- **Average case:**

Good and bad splits are randomly distributed across throughout the tree. Analysis of all the permutations which distribute pivot evenly to places in the list results in complexity of

$T(n) = O(n \log n)$

Merge Sort

- Divide and Conquer
- Merge sort is a technique which divides the array into subarrays of size 2
- We then have $n/2$ arrays of size 2.
- We then sort the elements and merge
 - Repeat this process until there is only one array remaining of size n .

Algorithm

MergeSort(arr[], l, r)

If $r > l$

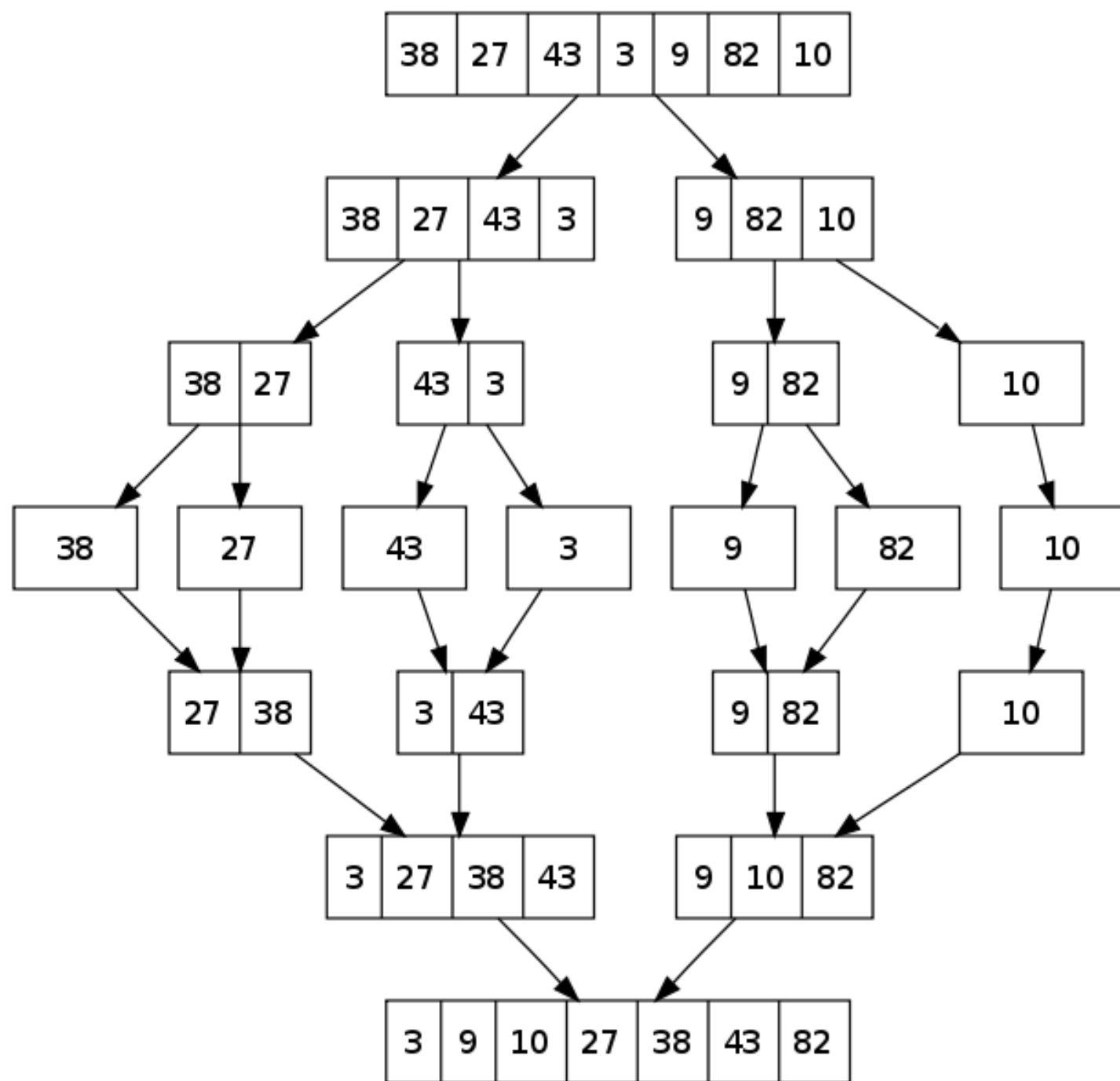
1. Find the middle point to divide the array into two halves:

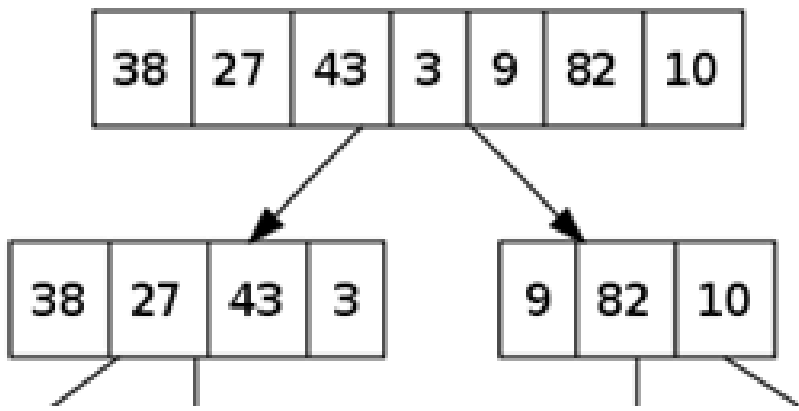
$$\text{middle } m = (l+r)/2$$

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)





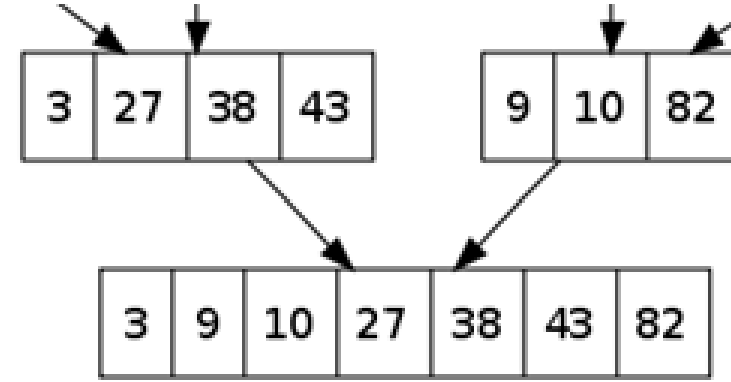
Algorithm:

MergeSort(A, l, r)

```

{
  If(l < r)
  {
    m = ⌊(l+r)/2⌋
    MergeSort(A, l, m)
    MergeSort(A, m+1, r)
    Merge(A, l, m+1, r)
  }
}

```



Merge(A, B, l, m, r)

```

{
  x = l, y = m, k = 1;
  While(x < m && y < r)
  {
    if(A[x] < A[y])
    {
      B[k] = A[x];
      k++; x++;
    }
    else
    {
      B[k] = A[y];
      k++; y++;
    }
  }
  while(x < m)
  {
    B[k] = A[x];
    k++; x++;
  }
  while(y < r)
  {
    B[k] = A[y];
    k++; y++;
  }
  For(i = 1; i <= r; i++)
  {
    A[i] = B[i];
  }
}

```

Time Complexity:

Recurrence Relation for Merge sort:

$$T(n) = 1 \text{ if } n=1$$

$$T(n) = 2 T(n/2) + O(n) \text{ if } n>1$$

Solving this recurrence we get

$$T(n) = O(n \log n)$$

Space Complexity:

It uses one extra array and some extra variables during sorting,

therefore,

$$\text{Space Complexity} = 2n + c = O(n)$$

Heap Sort

<https://www.programiz.com/dsa/heap-sort>

Algorithm	Worse case	Average case
Bubble sort	$\frac{n * (n - 1)}{2} = O(n^2)$	$\frac{n * (n - 1)}{2} = O(n^2)$
Quick sort	$\frac{n * (n + 3)}{2} = O(n^2)$	$1.4 n \log n = O(n \cdot \log n)$
Insertion sort	$\frac{n * (n - 1)}{2} = O(n^2)$	$\frac{n * (n - 1)}{4} = O(n^2)$
Selection sort	$\frac{n * (n - 1)}{2} = O(n^2)$	$\frac{n * (n - 1)}{2} = O(n^2)$
Merge sort	$n \cdot \log n = O(n \cdot \log n)$	$n \cdot \log n = O(n \cdot \log n)$
Heap sort	$n \cdot \log n = O(n \cdot \log n)$	$n \cdot \log n = O(n \cdot \log n)$

Algorithm	Comments
Bubble sort	Good for small n ($n < 10$)
Quick sort	Excellent for virtual memory environment
Insertion sort	Good for almost sorted data
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 the average case and far superior to quick sort in the worst case

Thank You