

SORTING

Unit 7

Introduction

- Sorting refers to arranging a set of data in some logical order
- For ex. A telephone directory can be considered as a list where each record has three fields - name, address and phone number.
- Being unique, phone number can work as a key to locate any record in the list.

Introduction

- Sorting is among the most basic problems in algorithm design.
- We are given a sequence of items, each associated with a given key value. And the problem is to rearrange the items so that they are in an increasing(or decreasing) order by key.
- The methods of sorting can be divided into two categories:
 - **Internal** Sorting
 - **External** Sorting

- **Internal Sorting**

- ✓ If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used

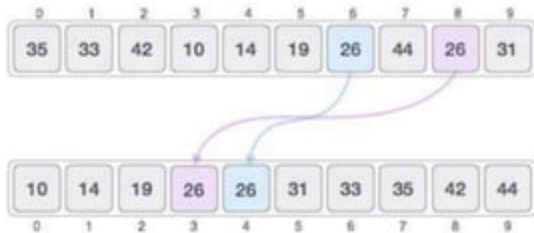
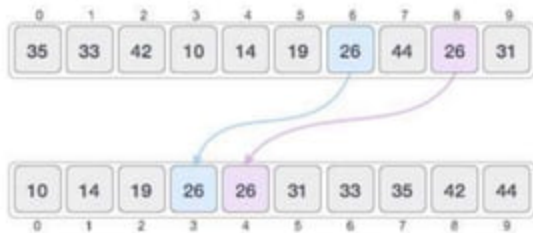
- **External Sorting**

- ✓ When the data to be sorted can't be accommodated in the memory at the same time and some has to be kept in auxiliary memory, then external sorting methods are used.

❖ **NOTE:** We will only consider **internal sorting**

Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.

Efficiency of Sorting Algorithm

- The complexity of a sorting algorithm measures the running time of a function in which n number of items are to be sorted.
- The choice of sorting method depends on efficiency considerations for different problems.
- Three most important of these considerations are:
 - The length of time spent by programmer in coding a particular sorting program
 - Amount of machine time necessary for running the program
 - The amount of memory necessary for running the program

Efficiency of Sorting Algorithm

- Various sorting methods are analyzed in the cases like – **best case**, **worst case** or **average case**.
- Most of the sort methods we consider have requirements that range from $O(n \log n)$ to $O(n^2)$.
- A sort should not be selected only because its sorting time is $O(n \log n)$; the relation of the file size n and the other factors affecting the actual sorting time must be considered

Efficiency of Sorting Algorithm

- Determining the time requirement of sorting technique is to actually run the program and measure its efficiency.
- Once a particular sorting technique is selected the need is to make the program as efficient as possible.
- Any improvement in sorting time significantly affect the overall efficiency and saves a great deal of computer time.

Efficiency of Sorting Algorithm

- Space constraints are usually less important than time considerations.
- The reason for this can be, as for most sorting programs, the amount of space needed is closer to $O(n)$ than to $O(n^2)$
- The second reason is that, if more space is required, it can almost always be found in auxiliary storage.

Things to remember

- An ideal sort is an **in-place sort** where the algorithm uses no additional array storage, and hence it is possible to sort very large lists without the need to allocate additional working storage.
- A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.
- Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

Things to remember

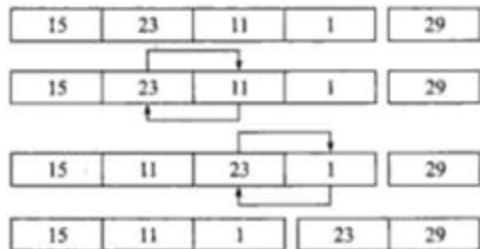
- Sorting can be performed in many ways.
- Over a time several methods (or algorithms) are being developed to sort data(s).
 - Bubble sort, Selection sort, Quick sort, Merge sort, Insertion sort are the few sorting techniques discussed in this chapter.
- It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances.

BUBBLE SORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0th element and compare it with the 1st element.
- If it is found to be greater than the 1st element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.



First iteration



Second iteration



Third iteration



Fourth iteration

Fig. 11.1 Bubble Sort

Algorithm

```
/* Bubble sort*/
for (i = 0; i < n-1 ; i++)
{
    xchanges=0;
    for (j = 0; j <n-1-i; j++)
    {
        if (arr[j] > arr[j+1])
        {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            xchanges++;
        }
        /*End of if*/
    }
    /*End of inner for loop*/
    if (xchanges==0) /*If list is sorted*/
        break;
    printf("\nAfter Pass %d elements are : ",i+1);
    for (k = 0; k < n; k++)
        printf("%d ", arr[k]);
    printf("\n");
}
/*End of outer for loop*/
```

TIME COMPLEXITY

- The time complexity for bubble sort is calculated in terms of the number of comparisons $f(n)$ (or of number of loops)
- Here two loops(outer loop and inner loop) iterates(or repeated) the comparison.
- The inner loop is iterated one less than the number of elements in the list (i.e., $n-1$ times) and is reiterated upon every iteration of the outer loop

$$\begin{aligned}f(n) &= (n-1) + (n-2) + \dots + 2 + 1 \\&= n(n-1) = O(n^2).\end{aligned}$$

TIME COMPLEXITY

- Best Case

- sorting a sorted array by bubble sort algorithm
- In best case outer loop will terminate after one iteration, i.e it involves performing one pass which requires $n-1$ comparison

$$f(n) = O(n^2)$$

- Worst Case

- Suppose an array [5,4,3,2,1], we need to move first element to end of an array
- $n-1$ times the swapping procedure is to be called

$$f(n) = O(n^2)$$

- Average Case

- Difficult to analyse than the other cases
- Random inputs, so in general

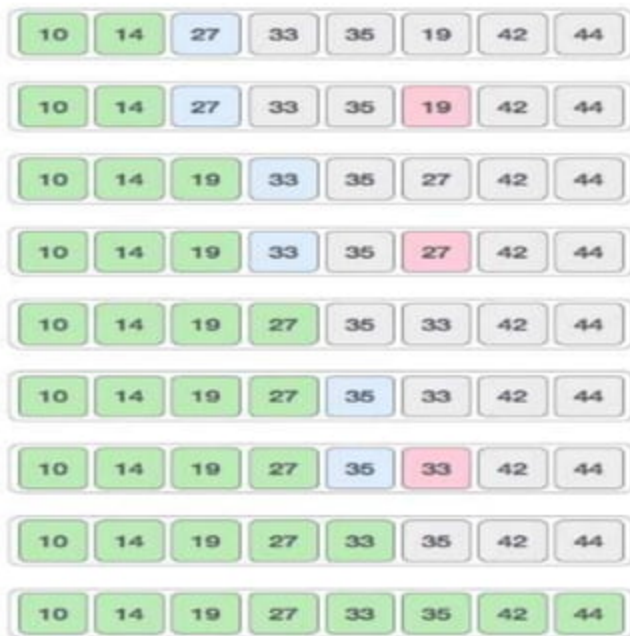
$$f(n) = O(n^2)$$

- Space Complexity

- $O(n)$

SELECTION SORT

- Find the least(or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.
- Let $a[n]$ be a linear array of n elements. The selection sort works as follows:
- Pass 1: Find the location **loc** of the smallest element in the list of n elements $a[0]$, $a[1]$, $a[2]$, $a[3]$,, $a[n-1]$ and then interchange $a[loc]$ and $a[0]$.
- Pass 2: Find the location **loc** of the smallest element in the sub-list of $n-1$ elements $a[1]$, $a[2]$, $a[3]$,, $a[n-1]$ and then interchange $a[loc]$ and $a[1]$ such that $a[0]$, $a[1]$ are sorted.
- Then we will get the sorted list
 $a[0] \leq a[1] \leq a[2] \leq a[3] \dots \leq a[n-1]$



Algorithm:

SelectionSort(A)

```
{  
    for( i = 0; i < n ; i++)  
    {  
        least=A[i];  
        p=i;  
        for ( j = i + 1; j < n ; j++)  
        {  
            if (A[j] < A[i])  
                least= A[j]; p=j;  
        }  
        swap(A[i],A[p]);  
    }  
}
```

Time Complexity

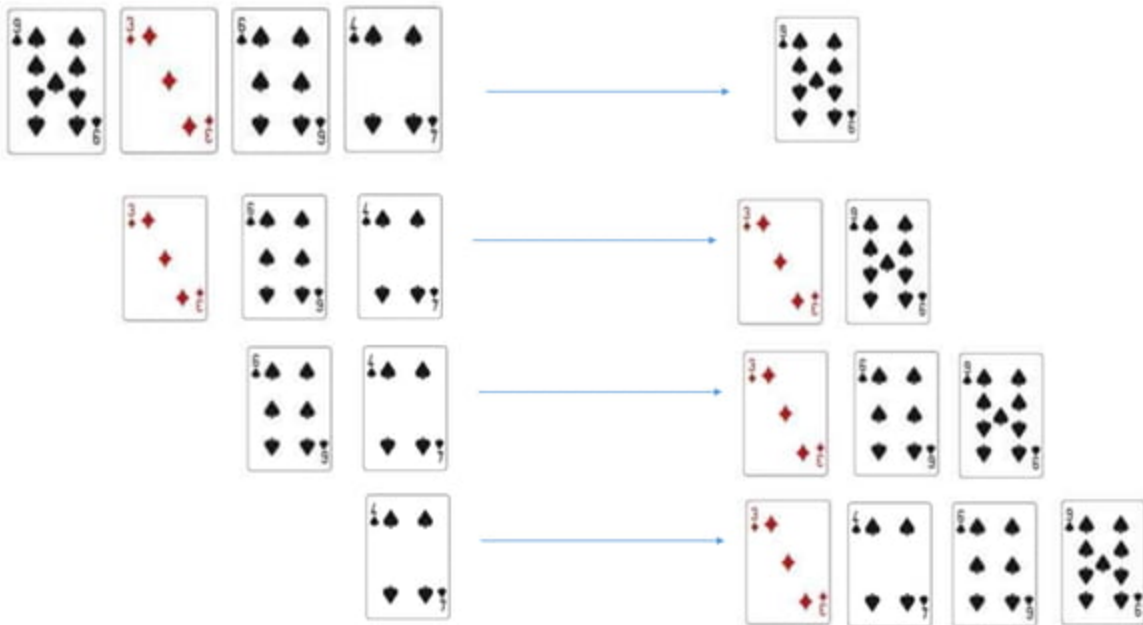
- Inner loop executes $(n-1)$ times when $i=0$, $(n-2)$ times when $i=1$ and so on:
- Time complexity = $(n-1) + (n-2) + (n-3) + \dots + 2 + 1$
 $= O(n^2)$

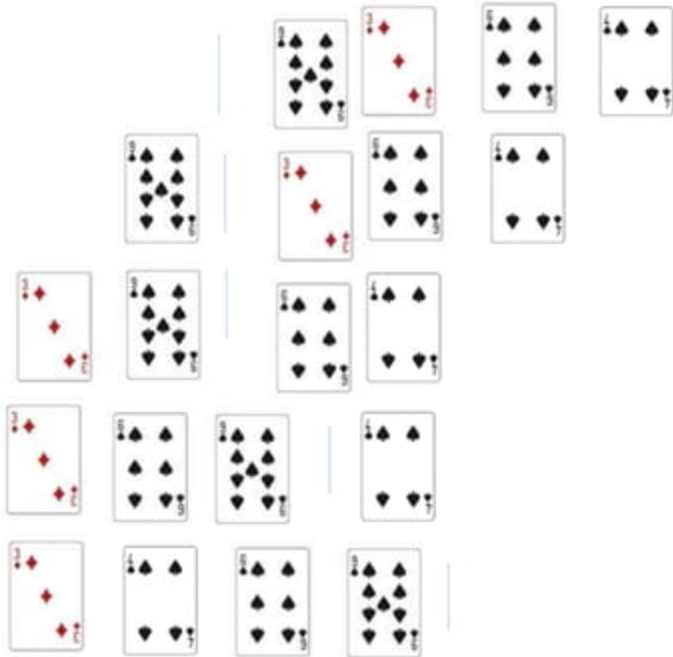
Space Complexity

- Since no extra space beside n variables is needed for sorting so
- $O(n)$

Insertion Sort

- Like sorting a hand of playing cards start with an empty hand and the cards facing down the table.
- Pick 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

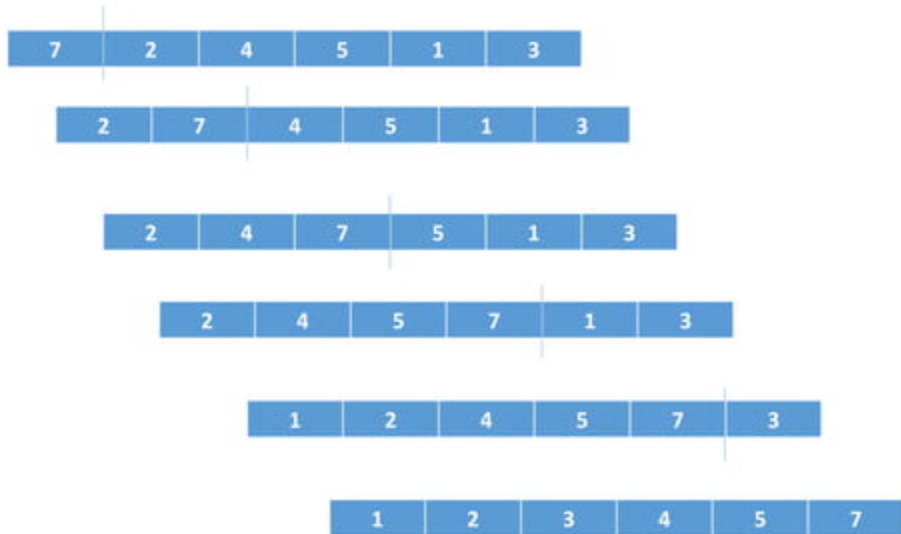
- Suppose an array $a[n]$ with n elements. The insertion sort works as follows:

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.



Algorithm

7	2	4	1	5	3
---	---	---	---	---	---

1st Pass

7	7	4	1	5	3
---	---	---	---	---	---

2	7	4	1	5	3
---	---	---	---	---	---

i	value	hole
1	2	1
1	2	0

```
InsertionSort(){  
    for (i=1;i<n;i++){  
        value=C[ i ];  
        hole= i ;  
        while(hole>0 && C[hole-1]>value){  
            C[hole]=C[hole-1];  
            hole=hole-1;  
        }  
        C[hole]=value;  
    }  
}
```

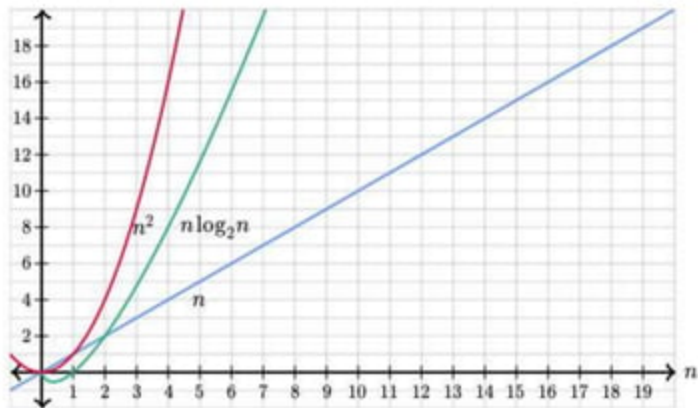
Time Complexity

- Best Case:
 - If the array is all but sorted then
 - Inner Loop won't execute so only some constant time the statements will run
 - So Time complexity = $O(n)$
- Worst Case:
 - Array element in reverse sorted order
 - Time complexity = $O(n^2)$
- Space Complexity
 - Since no extra space beside n variables is needed for sorting so
 - Space Complexity = $O(n)$

Divide and conquer algorithms

- The sorting algorithms we've seen so far have worst-case running times of $O(n^2)$
- When the size of the input array is large, these algorithms can take a long time to run.
- Now we will discuss two sorting algorithms whose running times are better
 - Merge Sort
 - Quick Sort

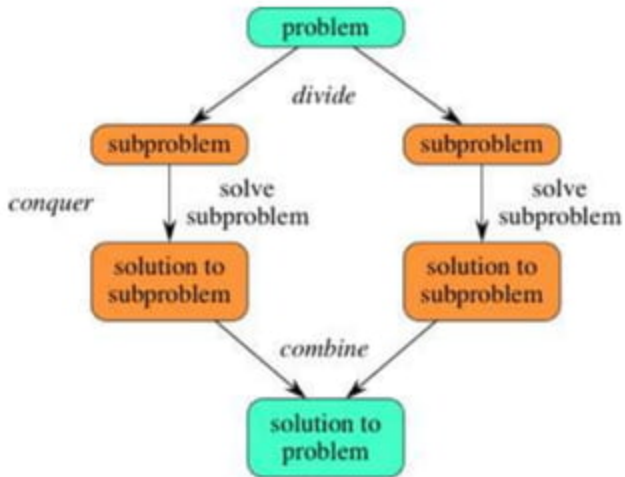
Algorithm	Worst-case running time	Best-case running time	Average-case running time
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$



Divide-and-conquer

- **Divide-and-conquer**, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem.
- Think of a divide-and-conquer algorithm as having three parts:
 - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
 - **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
 - **Combine** the solutions to the subproblems into the solution for the original problem.

Divide-and-conquer

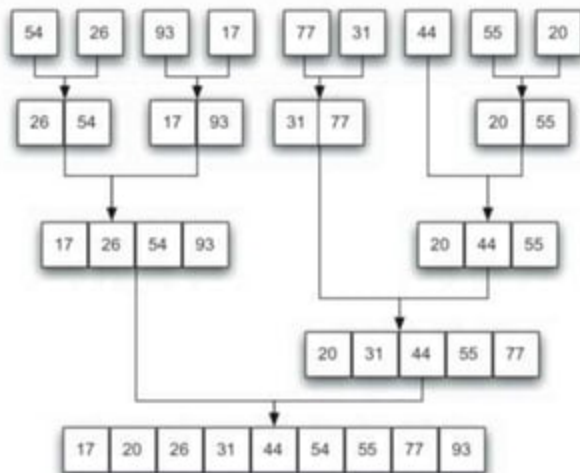
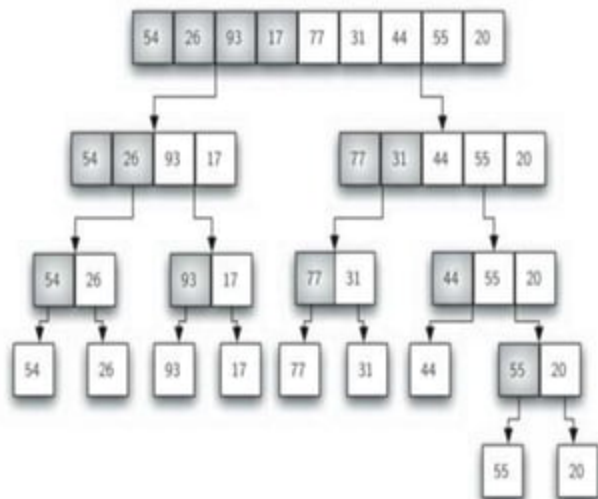


Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.
- With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

Merge Sort

- Because we're using divide-and-conquer to sort, we need to decide what our sub problems are going to be.
- Full Problem: Sort an entire Array
- Sub Problem: Sort a sub array
- Lets assume **array[p..r]** denotes this subarray of array.
- For an array of n elements, we say the original problem is to sort **array[0..n-1]**



Merge Sort

- Here's how merge sort uses divide and conquer
 1. **Divide** by finding the number **q** of the position midway between **p** and **r**. Do this step the same way we found the midpoint in binary search: add **p** and **r**, divide by 2, and round down.
 2. **Conquer** by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray **array[p..q]** and recursively sort the subarray **array[q+1..r]**.
 3. **Combine** by merging the two sorted subarrays back into the single sorted subarray **array[p..r]**.

Merge Sort

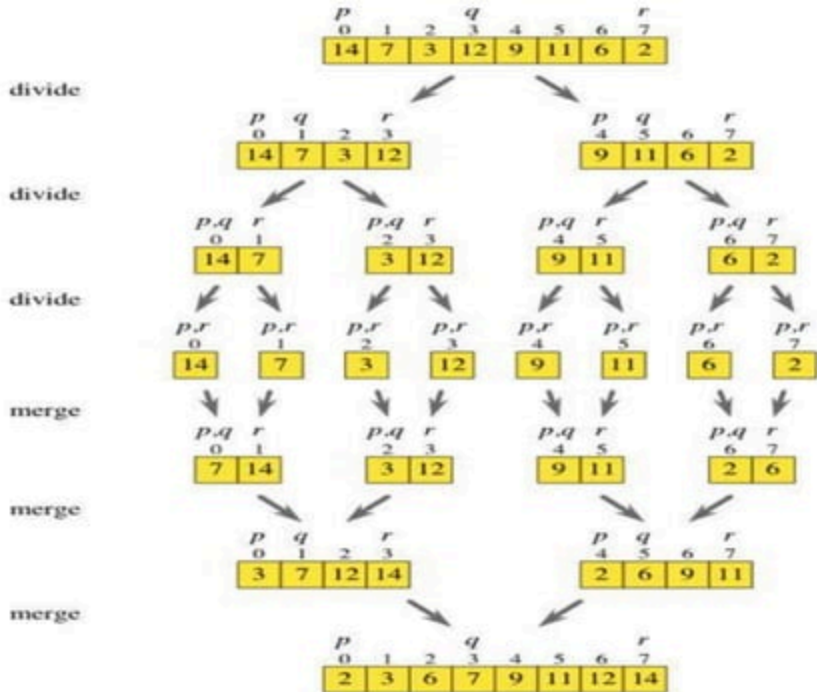
- Let's start with array holding [14,7,3,12,9,11,6,2]
- We can say that **array[0..7]** where $p=0$ and $r=7$
- In the **divide** step we compute $q=3$
- The **conquer** step has us sort the two subarrays
 - $\text{array}[0..3] = [14,7,3,12]$
 - $\text{array}[4..7] = [9,11,6,2]$
 - When we come back from the conquer step, each of the two subarrays is sorted i.e.
 - $\text{array}[0..3] = [3,7,12,14]$
 - $\text{array}[4..7] = [2,6,9,11]$
- Finally, the **combine** step merges the two sorted subarrays in first half and the second half, producing the final sorted array [2,3, 6,7,9, 11, 12,14]

How did the subarray array[0..3] become sorted?

- It has more than two element so it's not a base case.
- So with $p=0$ and $r=3$, compute $q=1$, recursively sort array[0..1] and array[2..3], resulting in array[0..3] containing [7,14,3,12] and merge the first half with the second half, producing [3,7,12,14]

How did the subarray array[0..1] become sorted?

- With $p=0$ and $r=1$, compute $q=0$, recursively sort array[0..0] ([14]) and array[1..1] ([7]), resulting in array[0..1] still containing [14, 7], and merge the first half with the second half, producing [7, 14].



Analysis of merge Sort

- We can view merge sort as creating a tree of calls, where each level of recursion is a level in the tree.
- Since number of elements is divided in half each time, the tree is balanced binary tree.
- The height of such a tree tend to be **$\log n$**

Analysis of merge Sort

- Divide and conquer
- Recursive
- Stable
- Not In-place
- $O(n)$ space complexity
- $O(n \log n)$ time complexity

Recursive function calls and finally merge

```
void MergeSort(int *A,int n) {  
    int mid,i, *L, *R;  
    if(n < 2) return;  
    mid = n/2;  
    L = (int*)malloc(mid*sizeof(int));  
    R = (int*)malloc((n- mid)*sizeof(int));  
  
    for(i = 0;i<mid;i++)  
        L[i] = A[i]; // creating left subarray  
    for(i = mid;i<n;i++)  
        R[i-mid] = A[i]; // creating right subarray  
  
    MergeSort(L,mid);  
    MergeSort(R,n-mid);  
    Merge(A,L,mid,R,n-mid);  
    free(L);  
    free(R);  
}
```

Merge function merges the whole block of array

```
void Merge(int *A,int *L,int leftCount,int *R,int  
rightCount) {  
    int i,j,k;  
    i = 0; j = 0; k = 0;  
    while(i<leftCount && j< rightCount) {  
        if(L[i] < R[j]) A[k++] = L[i++];  
        else A[k++] = R[j++];  
    }  
    while(i < leftCount)  
        A[k++] = L[i++];  
    while(j < rightCount)  
        A[k++] = R[j++];  
}
```

Analysis of merge Sort

- The time to merge sort n numbers is equal to the time to do two recursive merge sorts of size $n/2$ plus the time to merge, which is linear.
- We can express the number of operations involved using the following recurrence relations

$$T(1)=1$$

$$T(n) = 2T(n/2) + n$$

- Going further down using the same logic

$$T(n/2) = 2T(n/4) + n/2$$

- Continuing in this manner, we can write

$$T(n) = nT(1) + n \log n$$

$$= n + n \log n$$

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

Analysis of merge Sort

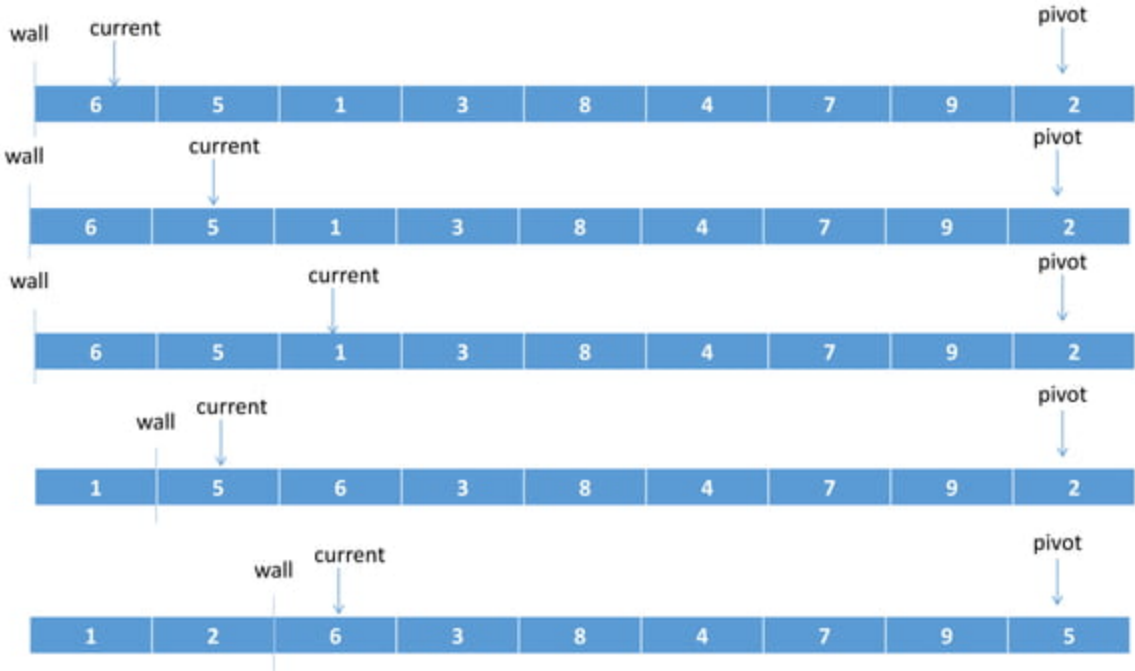
- Although merge sort's running time is very attractive it is not preferred for sorting data in main memory.
- Main problem arises when it uses linear extra memory as we need to copy the original array into two arrays of half the size and the additional work spent on copying to the temporary array and back
- This might considerably slow down the time to sort
- However merge sort can work very well with linked list as it doesn't require additional space. Since we only need to change pointer links rather than shifting the element.

Quick Sort

- Quick sort is one of the most popular sorting techniques.
- As the name suggests the quick sort is the fastest known sorting algorithm in practice.
- It has the best average time performance.
- It works by partitioning the array to be sorted and each partition in turn sorted recursively. Hence also called **partition exchange sort**.

Quick Sort

- In partition one of the array elements is chosen as a pivot element
- Choose an element $\text{pivot} = a[n-1]$. Suppose that elements of an array a are partitioned so that pivot is placed into position i and the following condition hold:
 - Each element in position 0 through $i-1$ is less than or equal to pivot
 - Each of the elements in position $i+1$ through $n-1$ is greater than or equal to key
- The pivot remains at the i^{th} position when the array is completely sorted. Continuously repeating this process will eventually sort an array.



Anyone problem with Quick Sort???

- Check this

<https://code.snipcademy.com/tutorials/algorithms/sorting/quick>

- Code on Github where pivot as the first index

https://github.com/ashim888/dataStructureAndAlgorithm/blob/dev/codes/sorting/quick_sort.c

Algorithm

- Choosing a pivot
 - To partition the list we first choose a pivot element
- Partitioning
 - Then we partition the elements so that all those with values less than pivot are placed on the left side and the higher value on the right
 - Check if the current element is less than the pivot.
 - If lesser replace it with the current element and move the wall up one position
 - else move the pivot element to current element and vice versa
- Recur
 - Repeat the same partitioning step unless all elements are sorted

Analysis of Quick Sort

- Best case
 - The best case analysis assumes that the pivot is always in the middle
 - To simplify the math, we assume that the two sublists are each exactly half the size of the original $T(N) = T(N/2) + T(N/2) + \dots + 1$ leads to $T(N) = O(n \log n)$
- Average case
 - $T(N) = O(n \log n)$
- Worst case
 - When we pick minimum or maximum as pivot then we have to go through each and every element so
 - $T(N) = O(n^2)$

Heap

- *A heap is defined as an almost complete binary tree of nodes n such that value of each node is less than or equal to the value of the father OR greater than or equal to the value of the father.*
 - ***Descending heap*** is an almost complete binary tree in which the value of each node is smaller or equal to the value of its father
 - ***Ascending heap*** is an almost complete binary tree in which the value of each node is greater or equal to the value of its father

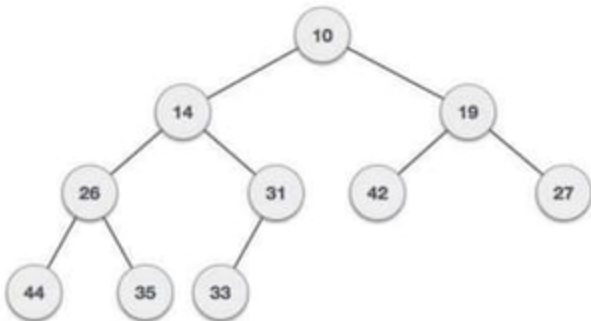
Heap

- Heap is a special case of balanced binary tree data structure where root-node key is compared with its children and arranged accordingly.
- If α has child node β then –
 - $\text{Key}(\alpha) \geq \text{key}(\beta)$
- As the value of the parent is greater than that of child, this property generates MAX heap. Based on this criteria a heap can be of two types
 - Min - Heap
 - Max - Heap

Heap

- Min-Heap

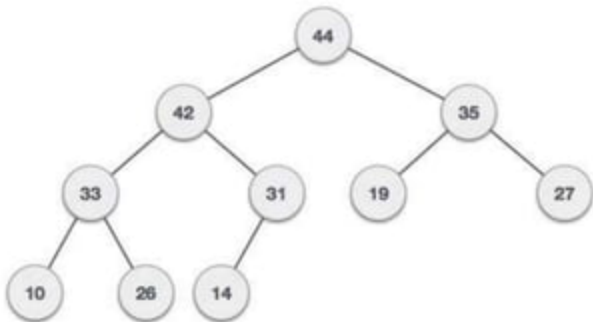
- Where the value of the root node is less than or equal to either of its children
- For input 35 33 42 10 14 19 27 44 26 31



Heap

- **Max-Heap –**

- where the value of root node is greater than or equal to either of its children.
- For input **35 33 42 10 14 19 27 44 26 31**



Max Heap Construction Algorithm

Step 1 – Create a new node at the end of heap.

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Input 35 33 42 10 14 19 27 44 26 31

Max Heap Deletion Algorithm

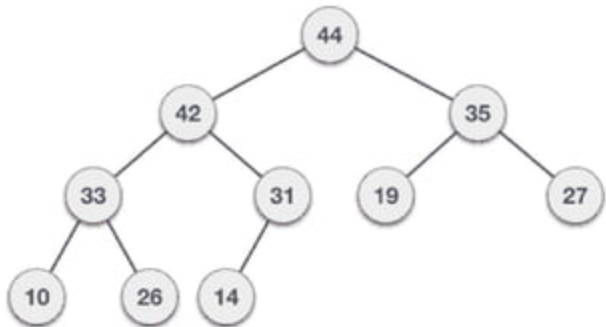
Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.



Analysis of Heap Sort

- The worst case complexity of the heap sort is $O(n \log n)$, therefore, Heap sort is superior to quick sort in the worst case
- Heap sort is not very efficient for small array because of the overhead of initial heap creation and computation.
- The space requirement for the heap sort is only one additional record to hold the temporary value.

Radix Sort

- The idea behind radix sort is slightly more complex than that of bucket sort.
- Algorithm:
 - Take the least significant digit of each element in the unsorted array
 - Perform a stable sort based on that key
 - Repeat the process sequentially with each more significant digit

Radix Sort

- For example suppose we want to sort the list
849,770,67,347,201,618,66,495,13,45
- Sorting by least significant digit(i.e. ones digit)
770,201,13,495,45,66,67,347,618,849
- Stability is maintained. Now sort by more significant tens digit,
210,13,618,45,347,849,66,67,770,495
- We now consider the hundreds digit to be zero
12,45,66,67,201,347,495,618,770,849

Assignments

- Slides at myblog

<http://www.ashimlamichhane.com.np/2016/08/unit-7-sorting/>

- Assignments at github

https://github.com/ashim888/dataStructureAndAlgorithm/tree/dev/Assignments/assignment_8

Reference

- <https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>
- http://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
- <http://bigocheatsheet.com/>
- <http://stackoverflow.com/questions/5222730/why-is-merge-sort-preferred-over-quick-sort-for-sorting-linked-lists>