Scientific Journal of Impact Factor(SJIF): 3.134

e-ISSN(O): 2348-4470
p-ISSN(P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 2, Issue 6, June -2015

A Comparison Based Analysis of Different Types of Sorting and Searching Algorithms in Data Structures

Patel Pooja

Department of Computer Science & Engineering.
Saffrony Institute of Technology, Mahesana, Gujarat, India.

Abstract: *Sorting and Searching is an important data structure operation, which makes easy searching, arranging and locating the information. I have discussed about various sorting and searching algorithms with their comparison to each other. This paper presents the different types of sorting algorithms of data structure like quick, insertion and also gives their performance analysis with respect to time complexity and searching algorithms of data structure like linear search, binary search.*

Keywords: *Sorting, Quick Sort, Insertion Sort, Bubble Sort, Searching, Linear Search, and Binary Search.*

I. INTRODUCTION:

Sorting and Searching are two fundamental operations in a computer science. Sorting means arranging on data in given order such that increment or decrement. Searching means find out location or find out an element of a given item in a collection of item. Many data structures are used to store information but Arrays, linked lists and tree are basic data structures used to sorting and searching operation.

Sorting data has been developed to arrange the array values in various ways for a database. For instance, sorting will order an array of numbers from lowest to highest or from highest to lowest, or arrange an array of strings into alphabetical order. Most simple sorting algorithms involve two steps which are compare two items and swap two items or copy one item.

Searching element is any type of a numerical data, alphabet, String, character data. we define to which type of searching algorithm used to which type of Problem and we compare to different type of searching algorithm in a different important parameter like time complexity, space complexity.

Each record is associated to a key and this key is separated to different record. If key are store on start of a record so this type of key is called to internal key. In other case key are store on separate table including pointer to the record so this type of key is called external key. Every file or tables have a two set of key. First set are define a uniquely data this key is a called primary key and this set have an internal and external key. Second set are define none uniquely data this key is a called secondary key.

There exist different types of algorithms that solve the Sorting and Searching method .
Sorting Algorithm:-1. Quick Sort
2. Insertion Sort
3. Merge sort

Searching Algorithm:-1. Liner Search
2. Binary Search

II. SEARCHING ALGORITHMS:

1. Sequential Searching:-

Simplest form of searching technique is a sequence search or linear search. This search is applicable to a small size of array or linked list data structure. Find out on searching element in sequential manner on unordered list or ordered list. In this method searching process are started at begins to end, scans the elements one by one of the list from left to right until the searching record/element is found. If we taken on ordered list or table then time it given to fast searching and good efficient as a comparison on unordered list.

Algorithm:

Here L is a location variable, I is a variable which value is a element position and A is Array/List [0-N]. SE is search element.

LINEAR _SEARCH (A, I, SE)

1. Initialize L=0
2. For I=0 to length [*A*] //scan on element start to end
3. If (SE = =A [*I*]) //check on searching element
4. L=L+1 // increment on location variable
5. Return A [I] //print on finding element
6. END If
7. END For LOOP

If L=0 then return "not find out element"
Exit

Advantages:

For smaller lists linear search may be faster because the speed of the simple increment.
Linear search very simplicity, resource efficient and memory efficient.
It can operate sorted and unsorted array and link list.

Disadvantages:

Linear search technique is low efficient and slower
than other searching algorithm.
Not suitable for large data set.

Example

Unordered Array A[1-6]	**A[0]**	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**Output**
	33	**22**	**55**	**11**	**33**	**44**	

Step:-1 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	**Return A[0],L=1**
Step:-2 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	
Step:-3 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	
Step:-4 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	
Step:-5 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	**Return A[0],L=2**
Step:-6 SE=33	**33**	**22**	**55**	**11**	**33**	**44**	

Example of Linear Search

1. Binary Searching:-

Another simplest form of searching technique and best efficient method is a binary search. It can be used if the table is stored as an array and mid size of array. If we use to linked list and insert/delete on data in linked list during searching time then problem are generated. This algorithm is also based on Divide-and-Conquer algorithm.

Divide: In sorted list is divided into two parts.

Conquer: We first compare to search/input element with the mid element of the list. If do not match then we check search element and mid element if search element is less then to mid element then go to first part/left part otherwise go to second/right part.

Combine: we apply to divide and conquer part whenever our search element is not found. When our search element is found then list are automatic combined and return.

Algorithm: Here L is a location variable, LOW is start index and HIGHT is last index element position of Sorted Array/List A[0-N]. SE is search element

BINARY_SEARCH (A,LOW,HIGHT,MID,SE)

1. Initialize LOW=0, HIGHT=N,L=0
2. While(LOW<=HIGHT) //scan on element start to end
3. MID=(LOW+HIGHT)/2 //divide on list
4. IF (SE==A[MID]) // check on searching element
5. L=L+1 // increment on location variable
6. Return A[MID] //print on finding element
7. Else If(SE<A[MID])
8. HIGHT=MID-1 // go to Left/first part
9. Else
10. LOW=MID+1 // go to right/second part
11. END If Statement
12. END While LOO
13. If L=0 then return “not find out element”
14. Exit

Advantage:

The general moral is that for large lists binary is very much faster than Linear search.
This search technique given a searching approach on binary search tree and other tree.

Disadvantage:

Binary search is not appropriate for linked list structures (no random access for the middle term)
Apply searching before we needed searching algo.
Not suitable for inserted/deleted a data in a searching time as a compare to other searching algorithm.

Example

Step=1, LOW=0, HEIGHT=5, MID= (0+5)/2-2, A [MID] =33, SE=33

Check SE== A[MID]	11	22	33	33	44	55	

Step=2, LOW=MID+1, 2+1=3, HEIGHT=5, MID= (3+5)/2-4, A [MID] =44, SE=33

Check SE== A[MID]	11	22	33	33	44	55	

Step=3, LOW=3, HEIGHT-MID-1, 4-1=3, MID= (3+3)/2-3, A [MID] =33, SE=33

Check SE== A[MID]	11	22	33	33	44	55	

Example Of Binary Search

III. SORTING ALGORITHMS:

1. Insertion Sort [Best: O(N), Worst: O(N^2)]

Start with a sorted list of 1 element on the left, and N-1 unsorted items on the right. Take the first unsorted item and insert it into the sorted list, moving elements as necessary. We now have a sorted list of size 2, and N -2 unsorted elements. Repeat for all elements

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.It has one of the simplest implementation It is efficient for smaller data sets, but very inefficient for larger lists.

Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency. It is Stable, as it does not change the relative order of elements with equal keys. It is better than Selection Sort and Bubble Sort algorithms.

Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.

Algorithm:

```
/* a[ ] is the array, p is starting index, that is 0, And  r is the last index of array. */
void quick sort(int a[], int p, int r)
{
if(p < r)
{
int q;
q = partition(a, p, r);
quick sort(a, p, q);
quick sort(a, q+1, r);
}
}
int partition(int a[], int p, int r)
{
```

Advantage: Relative simple and easy to implement.
Twice faster than bubble sort.

Disadvantage: Inefficient for large lists.

Complexity of Insertion Sort

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate

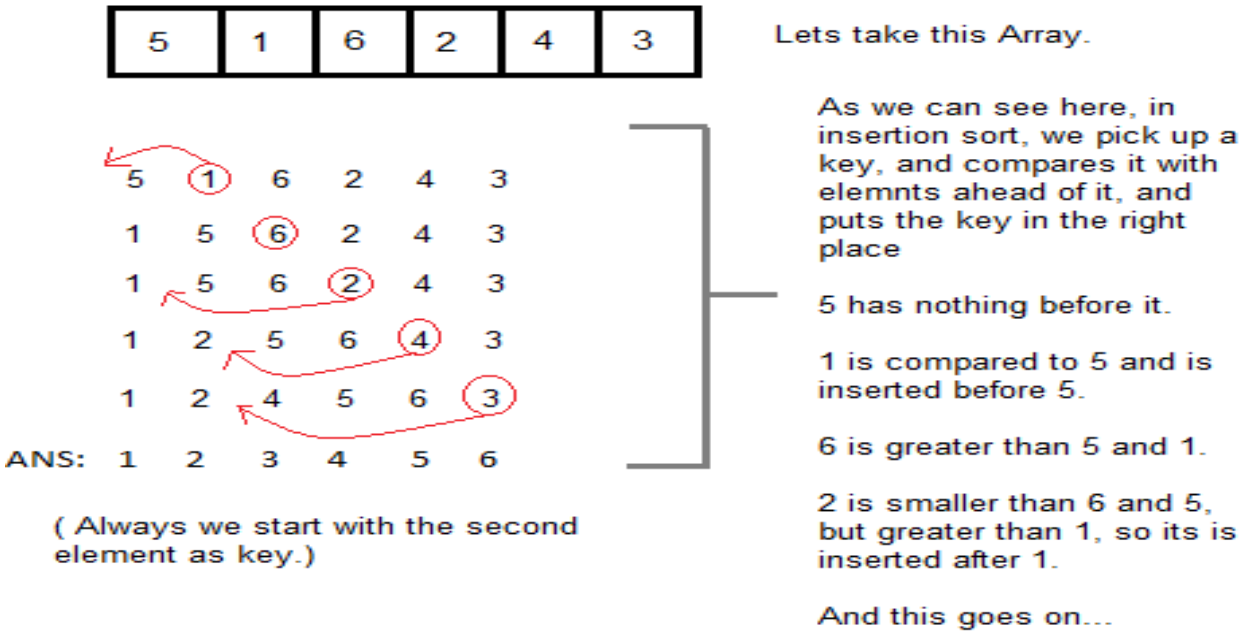
the number of comparisons of an array of N elements

We need 0 comparisons to insert the first element
We need 1 comparison to insert the second element
We need 2 comparisons to insert the third element
...
We need (N-1) comparisons (at most) to insert the last element
Totally,

1 + 2 + 3 + ... + (N-1) = O(n2)

Worst Case Time Complexity: O(n2)
Best Case Time Complexity: O(n)
Average Time Complexity: O(n2)

Insertion Sorting Work Example:



2. Quick sort [Best: O(N lg N), Avg: O(N lg N), Worst:(N^2)]

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition exchange sort).

This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

Algorithm:

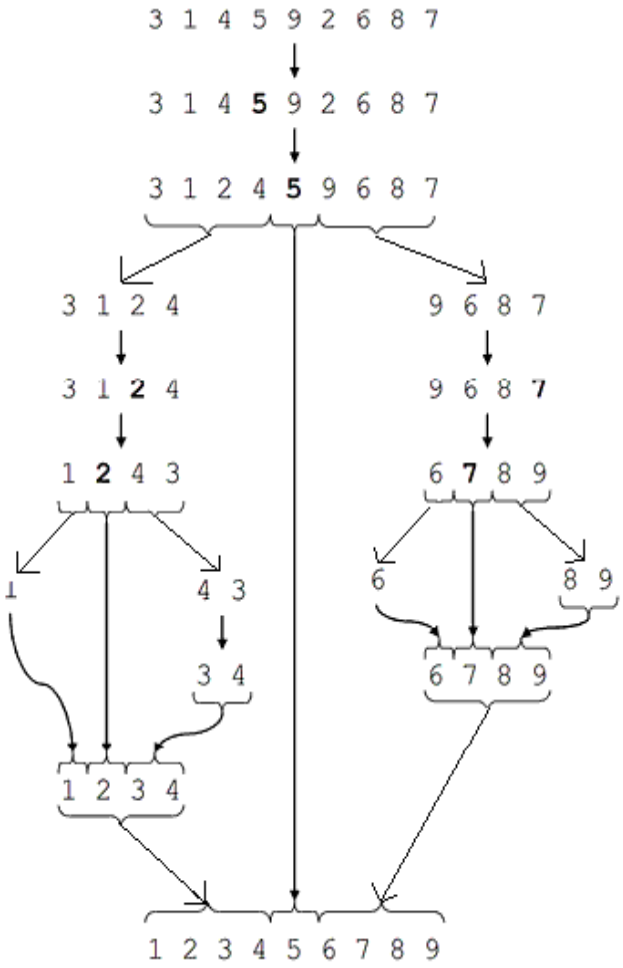

```
/* a[ ] is the array, p is starting index, that is 0, And  r is the last index of array. */

void quick sort(int a[], int p, int r)
{
if(p < r)
        {
        int q;
        q = partition(a, p, r);
        quick sort(a, p, q);
        quick sort(a, q+1, r);
        }
        }
        int partition(int a[], int p, int r)
        {
        int i, j, pivot, temp;
        pivot = a[p];
        i = p;
        j = r;
        while(1)
        {
        while(a[i] < pivot && a[i] != pivot)
                                        i++;
        while(a[j] > pivot && a[j] != pivot)
                                        j-;
                if(i < j)
                {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                }
                else
                {
                        return j;
                }
                }
        }
```

Advantage: Fast and efficient
Disadvantage: Show horrible result if list is already sorted.

Quick Sorting Work Example:

Ex. 3, 1, 2, 4, 5, 9, 6, 8, 7

The pivot element here is 5.



Complexity Analysis of Quick Sorting

1. **Best Case:-**

The best case for divide and conquer algorithms comes when input is divided as evenly as possible, i.e. each sub problem is of size n/2. The recurrence relation for best case is:

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

The partition step on each sub problem is linear in its size as the partitioning step consists of at most n swaps. Thus the total effort in partitioning the 2k problems of size n/2k

The recurrence relation reduces to:
$T(n)=T(n/2)+T(n/2)+ \theta(n)$
$T(n)=2T(n/2)+ \theta(n)$ …… (i)

Equation (i) can also be written as

$T(n)=2T(n/2)+ cn$ for some constant c.

The recursion tree for the best case looks like this.

Therefore, the running time for the best case of quick sort is:$T(n)=\theta(n \lg n)$
The best case is same as a average case.

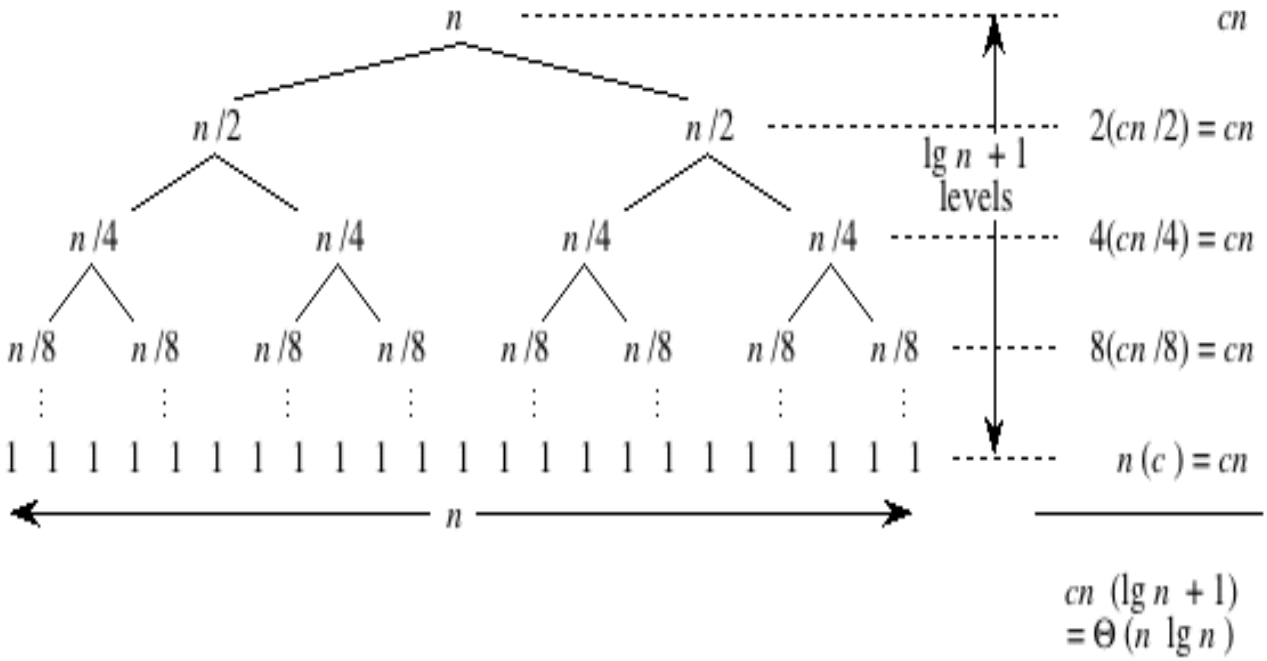


Fig 2: Recursion tree for best case of Sequential quick sort algorithm

2. Worst case:

Let the pivot element splits the array as unequally as possible. Thus instead of n/2 elements in the smaller half we get zero i.e. the pivot element is the biggest or smallest element in the array. This unbalanced partitioning arises in each recursive call. The recurrence relation for best case is:

$T(n)=T(n-1)+T(0)+f(n)$

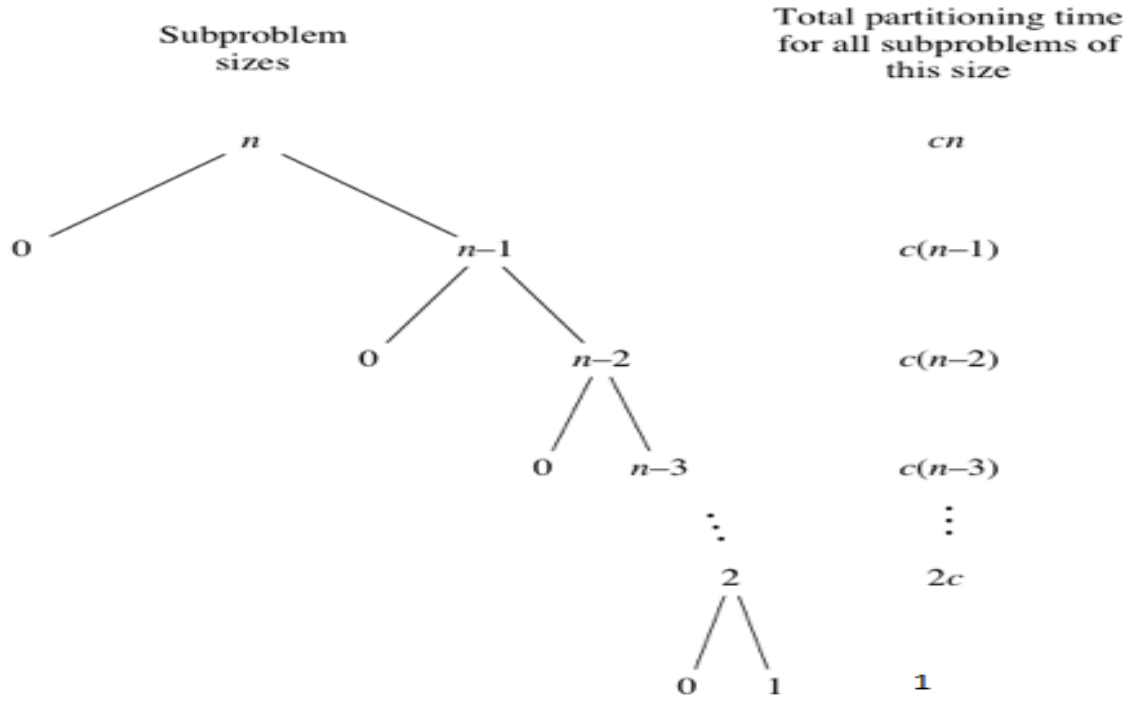
$T(n)=T(n-1)+T(0)+\theta(n)$ …… .(ii)

Equation (ii) can also be written as $T(n)=T(n-1)+T(0)+cn$ for some constant c.

The recursion tree for the worst case looks like this
Therefore, the running time is :

$T(n) = \theta(n2)$.

Worst Case Time Complexity: O(n2)
Best Case Time Complexity: O(n log n) Average Time Complexity: O(n log n)



T(n)=⁰(n2)

Fig 3: Recursion tree for worst case of quick sort algorithm

3. Merge Sort [Best: O(N lg N), Avg: O(N lg N), Worst:O(N lgN)]

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub array A[*p* .. *r*]. Initially, *p* = 1 and *r* = *n*, but these values change as we recurse through sub problems [8]. To sort A[*p* .. *r*]:

Divide Step

If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split A[*p* .. *r*] into two sub arrays A[*p* .. *q*] and A[*q* + 1 .. *r*], each containing about half of the elements of A[*p* .. *r*]. That is, *q* is the halfway point of A[*p* .. *r*].

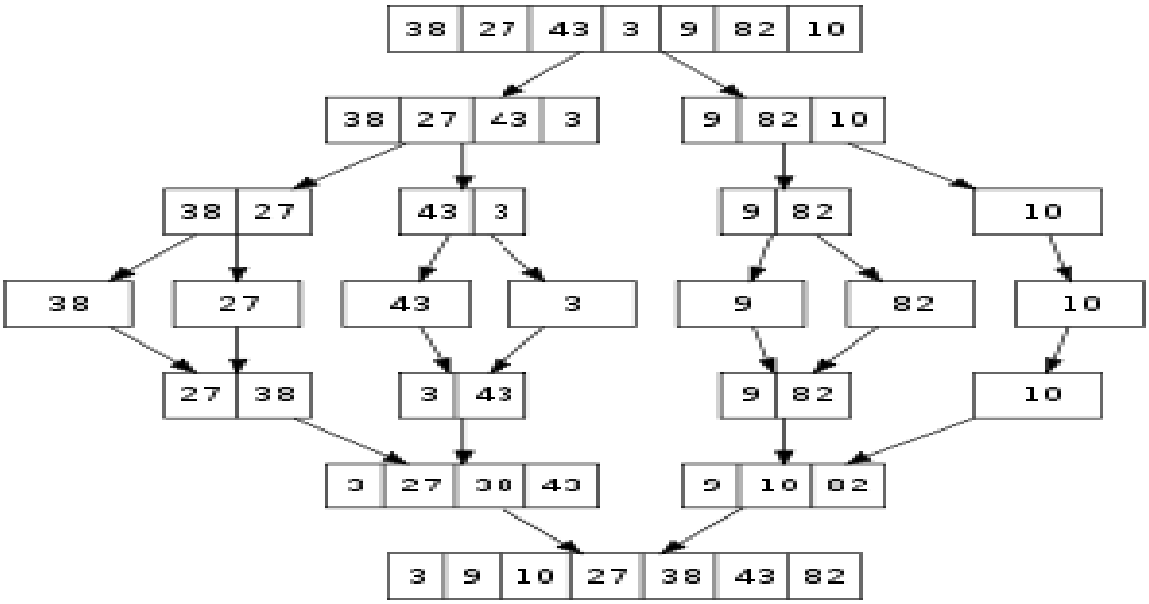
Conquer Step

Conquer by recursively sorting the two sub arrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].

Combine Step

Combine the elements back in *A*[*p* .. *r*] by merging the two sorted sub arrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence.

Merge Sorting Work Example:



To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

Algorithm:

```
/* a[] is the array, p is starting index, that is 0,and r is the last index of array. */
void merge sort(int a[], int p, int r)
{
        int q;
        if(p < r)
        {
        q = floor( (p+r) / 2);
        mergesort(a, p, q);
        mergesort(a, q+1, r);
        merge(a, p, q, r);
        }
}

void merge(int a[], int p, int q, int r)
{
        int b[5]; //same size of a[]

        int i, j, k;
        k = 0;
        i = p;
        j = q+1;
        while(i <= q && j <= r)
        {
        if(a[i] < a[j])
        {
        b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;
```

```
}
else
{
b[k++] = a[j++];
}
}
while(i <= q)
{
b[k++] = a[i++];
}
while(j <= r)
{
b[k++] = a[j++];
}
for(i=r; i >= p; i--)
{
a[i] = b[--k]; // copying back the sorted list to a[]
}
}
```

Advantage*:* Well suited for large data set.
Disadvantage*:* At least twice the memory requirements than other sorts

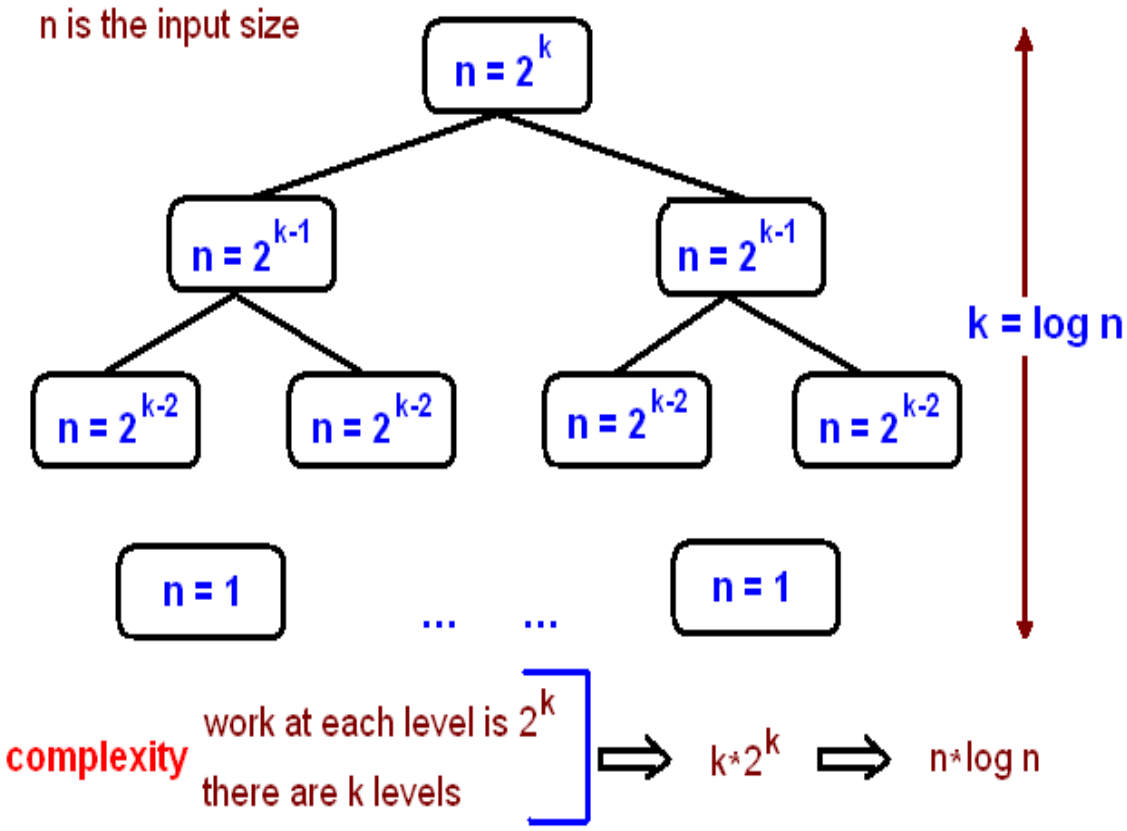
Complexity of Merge sort

Suppose T(n) is the number of comparisons needed to sort an array of n elements by the Merge Sort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation

T(n) = 2*T(n/2) + n - 1

The solution to this equation is beyond the scope of this course. However I will give you a resoning using a binary tree. We visualize the merge sort dividing process as a tree.

Worst Case Time Complexity: O(n log n)
Best Case Time Complexity: O(n log n)
Average Time Complexity: O(n log n)



Conclusion: Here we describe three sorting algorithm and two searching algorithm. Their best case, average case and worst case. Sorting Algorithm

Sorting

	Time Complexity					
Sort	Avg.	Best	Worst	Space	Stability	Remarks
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Merge Sort	O(n*logn)	O(n*logn)	O(n*logn)	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quick Sort	O(n*logn)	O(n*logn)	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.

Searching

Parameter	**Sequential Search**	**Binary Search**
Total no of comparison	(N+1)/2 for successful, N for unsuccessful	Each comparison reduces the number of possible candidates by a factor of 2. approximately comparison is 2*log2N
Time Complexity 1.Best Case 2.Average Case 3.Worst Case	 O(1) O(N) O(N)	 O(1) O(log N) O(log N)
Space Complexity	O(N)	O(N)
Associated key	Internal key	Internal key
Searching type	Internal Searches	Internal Searches
Sorting	Do not needed, depending on user	Yes , use to any type of Sorting method
Data Structure	Array, Linked List	Array
Simplicity	Easy	Average
Efficient	Low	Medium
Algorithm	Straightforward algorithm	Divide-and-Conquer
Implementation on programming	Easy	Average
Strategy	Scan on list start to end and match search element one by one	Divide on list match on mid element and search element. If the search element is less than the middle element, do again searching on the left half; otherwise, search the right half.
Table and file are	Unordered/Ordered	Ordered

Reference

[1] Ms. Nidhi Chhajed, Mr. Imran Uddin, Mr. Simarjeet Singh Bhatia," International Journal of Advanced Research in Computer Science and Software Engineering " Volume 3, Issue 2, February 2013 .

[2]Mr. Pankaj Sareen ," International Journal of Advanced Research in Computer Science and Software Engineering " Volume 3, Issue 3, March 2013

[3]Mr.Kamlesh Kumar Pandey , "International Journal of Computer Science and Mobile Computing" Voumel.3 Issue.7, July- 2014, pg. 751-758

[4] Ms.Ishwari Singh Rajput , Mr.Bhawnesh Kumar , Ms.Tinku Singh," Department of Computer Science & Engineering J.P Institute of Engineering & Technology, Meerut U.P., India "*Volume 57– No.9, November 2012*

[5]Mr.Gaurav Kocher1,Mr. Nikita Agrawal2," International Journal of Scientific Engineering and Research" Volume 2 Issue 3, March 2014

