International Journal of Advance Research and Innovation
Vol. 2(2), Apr-Jun 2014, pp. 29-42
Doi: 10.51976/ijari.221405
www.gla.ac.in/journals/ijari

Article Info
Received: 05 Mar 2014 | Revised Submission: 20 Apr 2014 | Accepted: 28 May 2014 | Available Online: 15 Jun 2014

Advanced Optimization of Fundamental Searching and Sorting Algorithms

*Jasrat Singh**

ABSTRACT

Searching and Sorting are the most important data structure operations, which make easy searching, arranging and locating the information. One of the basic problems of computer science is sorting a list of items and searching elements in the array. There are a number of solutions to these problems, known as sorting algorithm, and searching algorithm. Some searching algorithms are simple and spontaneous, such as the Linear Search, Binary Search. All searching algorithms are problem specific meaning they work well on some specific problem and do not work well for all the problems, therefore, appropriate for specific kinds of problems. Some searching algorithm works on less number of elements. There are some fundamental sorting algorithms as Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Bubble Sort etc. After studying various sorting and searching algorithms we found that the algorithms can be optimised. This paper presents a new searching algorithms named as "OBSwQS" is designed to perform searching quickly and more effectively as compared to the existing version of searching algorithm in which there are used the concepts of Binary Search and Quick Sort algorithms, and the optimised sorting algorithms as "DNSS", "DNBS", "Insertion Sort within the Merge Sort" and etc. are the optimised sorting algorithm which are the optimization of Fundamental Selection Sort, Bubble Sort, Merge Sort and etc. The introduction of Dual Nature Selection Sort and Dual Nature Bubble Sort version of selection sort algorithm and Bubble Sort for sorting the data stored in database instead of existing selection sort algorithm and Bubble Sort algorithm will provide an opportunity to the users to save almost 50% of their operation time with almost 100% accuracy. In "Insertion Sort within the Merge Sort algorithm" there is found out the value of maximum value of n (n is number of items to be sorted) so that the insertion sort beats the merge sort. Hence the "ISwMS" is optimization of Fundamental Merge Sort algorithm.

Keywords: *Dual Nature Selection Sort; Dual Nature Bubble Sort; Insertion Sort within the Merge Sort Algorithm; Optimized Binary Search with Quick Sort.*

1.0 Introduction

One of the basic problems of computer science is sorting a list of items and searching elements in the array. There are a number of solutions to this problem, known as sorting algorithms, and searching.. Some searching algorithms are simple and spontaneous, such as the Linear Search, Binary Search. All searching and sorting algorithms are problem specific meaning they work well on some specific problem and do not work well for all the problems, therefore, appropriate for specific kinds of problems.

There are several elementary and advance sorting algorithms. All sorting algorithm are problem specific meaning they work well on some specific problem and some are suitable for floating point numbers, some are good for specific range, some sorting algorithms are used for huge number of data, and some are used if the list has repeated values. We sort data either in statistical order or lexicographical, sorting numerical value either in increasing order or decreasing order and alphabetical value like addressee key.

Everything in this world has some advantage and disadvantage, sorting is a data structure operation, which is used for making easy searching and arranging of element or record.

There are many fundamental and advance sorting algorithms. Sorting algorithm help searching data quickly and in this way it saves time. Arranging the record or element in

**Department of Computer Science and Engineering, TMU, Moradabad, Uttar Pradesh, India*
(E-mail: Jassy790@gmail.com)

some mathematical or logical order is known as sorting. Mainly sorting may be either numerical or in alphabetical. In numerical sorting we will sort numeric value either in increasing order or decreasing order and in alphabetical sort we will sort the alphabetical value e.g. Name key. I grew up with the bubble sort in common; I am sure with many colleagues having learnt one sorting algorithm, there seemed little point in learning than any others. It was hardly an exciting area of study. The efficiency with which sorting is carried out will often have a significant impact on the overall efficiency of a program. Consequently there has been much research and it is interesting to see the range of alternative algorithms that have been developed. It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted; In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to\ the data being sorted. Four common algorithms are the exchange or bubble sort, the selection sort, the insertion sort and the quick sort. The selection sort is a good. It is intuitive and very simple to program. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons.

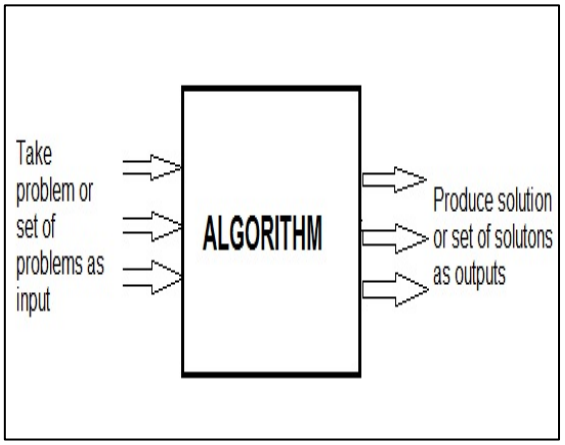
1.1 Algorithm

In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input. In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s) for the target machines to produce output from given *input* (perhaps null)

Algorithm is the process in which there is taken problem or set of problems as input and produce solution or set of solutions as outputs.

Fig 1: Algorithm



1.2 Complexity analysis of algorithms

In computer science, the analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

1.3 Efficiency of an algorithm can be measured in terms of/type of complexity

a. Execution time (Time Complexity).
b. The amount of memory required (Space Complexity).

1.3.1 Time complexity

In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms.

1.3.1.1 Space complexity

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input

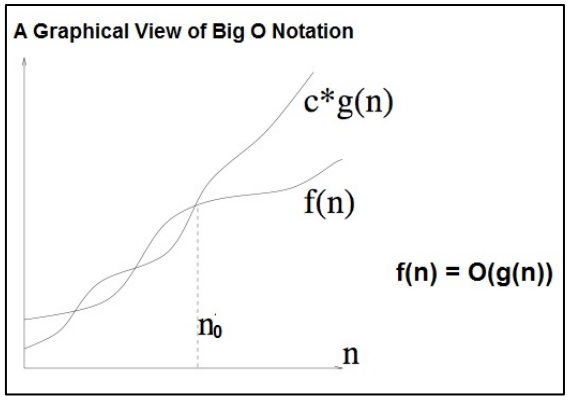
1.4 Asymptotic notation for complexity analysis of algorithm

In computational complexity theory, asymptotic computational complexity is the usage of the asymptotic analysis for the estimation of computational complexity of algorithms and computational problems, commonly associated with the usage of the big O notation.

There are used five asymptotic notations

1.4.1 Big o notation

Fig 2: Big O Notation



A function f(n) is of O(f(n)) iff there exist positive constants c and n0 such that

|f(n)| <= c*|g(n)| for all n>= n0.

f(n) = O(g(n))

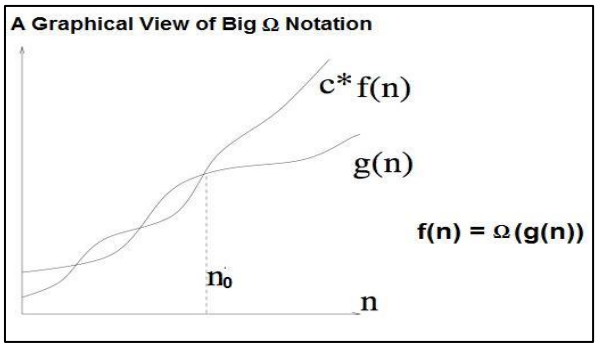
This is the upper case of complexity analysis and this is known as worst case complexity of the algorithm.

For example worst case complexity of the Quick Sort algorithm is

T(n) = O(n2)

1.4.2 Big Ω notation

Fig 3: Big Omega Notation



A function f(n) is of Ω(g(n)) iff there exist positive constants k and n0 such that |

f(n)| >=c*|g(n)| for all n >=n0.

f(n) = Ω(g(n))

This is used for Best Case Complexity of the algorithms. Hence this is the important notation for the analysis of the algorithms,

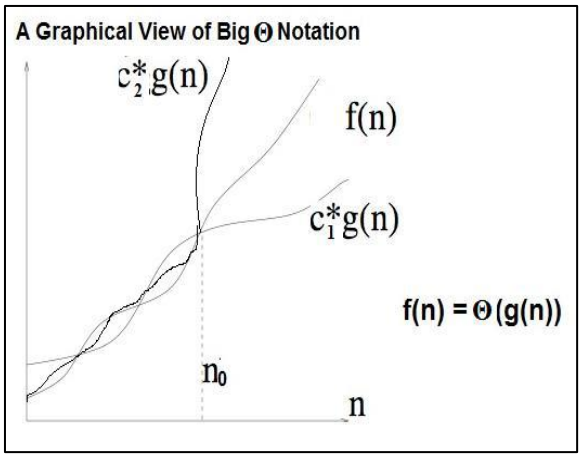
For example the Best case complexity of the Quick Sort algorithm is

T(n) = f(nlgn)

1.4.3 Theta Θ notation

A function f(n) is of Θ(f(n)) iff it is of O(f(n)) and it is of Ω(f(n)).

Fig 4: Theta Notation



C1*g(n) < = f(n) < = C2*g(n)

F(n) = Θ(g(n))

Some of the hierarchy of complexities is as follows

O(1) < O(log n) < O(n) < O(n log n) < O(n2) < O(n3) < O(2n) < O(n!)

This is used for the analysis of average case complexity of the algorithm.

1.4.4 Small oh (o) notation

This is another notation of complexity analysis and it is similar to big O, but difference is that there is not a equality condition. This also known as the asymptotic tight bound.

f(n) = o(g(n)) iff f(n) < C(g(n)) for n0 > n

1.4.5 Small omega (ƓƆ) notation

Small omega notation (ω) is similar of big omega notation but there is a difference that there exist inequalities between the relations.

f(n) = ƓƆ(g(n)) iff f(n) > C(g(n)) for n0 > n

2.0 Optimization of Fundamental Algorithms

2.1 Dual nature selection sort

As the selection sort has three main characteristic, the first that it is In-place algorithm and second it is a stable algorithm, and third it is simple while all other algorithm of order O(n2) do not have all these characteristics. So it is felt that why its time complexity should not improve. In this scenario, the author capture the idea from old selection sort that if we sort two data elements (smallest as well as largest) of the given data in single iteration, then its time can be reduced.

Therefore our DNSSA sorts the data from front and rear ends of the array and finishes the execution of outer loop when it reaches at the middle of the array. In its first iteration it finds the smallest and largest data elements of array and place those in their desired locations, then it finds the next smallest and largest data elements from the remaining array and sorts those in their next respective locations in the array.

In this way it executes half the iteration of the outer loop, while old SS only finds either smallest or largest (but not both) element of array and requires the full iteration of outer loop. Although DNSSA is still O(n2), but In this way its performance level has very much improved as compared to other sorting algorithm of said order i.e Bubble sort, insertion sort etc.

Although DNSSA is still O(n2), but In this way its performance level has very much improved as compared to other sorting algorithm of said order i.e Bubble sort, insertion sort etc.

2.2 Fundamental selection sort algorithm

This is a very easy sorting algorithm to understand and is very useful when dealing with small amounts of data. However, as with Bubble sorting, a lot of data really slows it down. Selection sort does have one advantage over other sort techniques. Although it does many comparisons, it does the least amount of data moving. Thus, if your data has small keys but large data area, then selection sorting may be the quickest.

The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.

The selection sort has a complexity of O(n2) . The Selection sort is the unwanted stepchild of the n2 sorts. It yields a better performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there isn't really any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

2.2.1 Algorithm

```
FSSA (A, n)
1. fori ← n – 1 to 0
1.1. IndexOfLarge ← 0
1.2. for j←1 to i
1.2.1 if (A[j]>A[IndexOfLarge])
1.2.1.1 indexOfLarge ← j
1.3. Large ←A[IndexOfLarge]
1.4. A[IndexOfLarge] ← A[i]
1.5 A[i] ← Large
```

Table 1: Complexity Analysis of Selection Sort Complexity

Pseudo Code Instruction	Instruction Exe Time	How Many times the instruction is exe. By CPU
1. for i ← n – 1 to 0	C_1	n.
1.1. IndexOfLarge ← 0	C_2	n-1
1.2. for j←1 to i	C_3	$\sum_{i=1}^{n-1}(i+1)$
1.2.1 if (X[j]>X[IndexOfLarge))	C_4	$\sum_{i=1}^{n-1}(i)$
1.2.1.1 indexOfLarge ← j	C_5	$\sum_{i=1}^{n-1} i(t_{ij})$
1.3. Large ←X[IndexOfLarge]	C_6	n-1
1.4. X[IndexOfLarge] ← X[i]	C_7	n-1
1.5 X[i] ← Large	C_8	n-1

In order to evaluate the execution time complexity of the given data of n elements. First we simplify the execution time of some inner loop statements in above algorithm.

Note that

- t_{ij}= 1 when the if statement is true, 0 otherwise
- $C_3\sum_{i=1}^{n-1}(i+1) = C_3\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1$

$= C_3 n(n-1)/2 + C_3(n-1).$

- $C_4\sum_{i=1}^{n-1} i = C_4\, n(n-1)/2$
- $C_5\sum_{i=1}^{n-1} i = C_5\, n(n-1)/2$

Best-Case Time Complexity of FSSA

Then for the best-case scenario we have that all tij = 0 so we get

$T(n) = C_1 n + C_2 n - C_2 + C_3 n(n-1)/2 + C_3 (n-1) + C_4 n(n-1)/2 + C_6 n - C_6 + C_7 n - C_7 + C_8 n - C_8$

$T(n) = n^2 (C_3/2 + C_4/2) + n (C_1 + C_2 + C_3 - C_3/2 - C_4/2 + C_6 + C_7 + C_8) - (C_2 + C_3 + C_7 + C_8)$

Let

$a = (C_3/2 + C_4/2).$

$b = (C_1 + C_2 + C_3 - C_3/2 - C_4/2 + C_6 + C_7 + C_8)$ and

$c = -(C_2 + C_3 + C_7 + C_8)$

Then T(n) becomes-

$$T(n) = a n^2 + bn + c$$

Thus here in best-case, the complexity of execution time of an algorithm shows the lower bound and is asymptotically denoted with Ω. Therefore by ignoring the constant a, b, c and the lower terms of n, and taking only the dominant term i.e. n2, then the asymptotic running time of selection sort will be Ω(n2) and will lie in of set of asymptotic function i.e. Θ(n2). Hence we can say that the asymptotic running time of SS will be:

$$T(n) = \Theta(n^2)$$

Worst -Case Time Complexity of FSSA

Now for the worst-case scenario we have that all tij = 1 so we have

$T(n) = C_1 n + C_2 n - C_2 n + C_3 n(n-1)/2 + C_3 (n-1) + C_4 n(n-1)/2 + C_5 n(n-1)/2 + C_6 n - C_6 + C_7 n - C_7 + C_8 n - C_8$

$T(n) = n^2 (C_3/2 + C_4/2 + C_5/2) + n (C_1 + C_2 + C_3 - C_3/2 - C_4/2 + C_5/2 + C_6 + C_7 + C_8) - (C_2 + C_3 + C_6 + C_7 + C_8)$

.....2.2.1.a

Taking highest order- $T(n) = n^2 (C_3/2 + C_4/2 + C_5/2)$ 2.2.1.b

Let

$a = (C_3/2 + C_4/2 + C_5/2)$

$b = (C_1 + C_2 + C_3 - C_3/2 - C_4/2 + C_6 + C_7 + C_8)$ and

$c = -(C_2 + C_3 + C_6 + C_7 + C_8)$

then T(n) becomes

$$T(n) = a n^2 + bn + c$$

Thus here in worst-case, the complexity of execution time of an algorithm shows the upper bound and is asymptotically denoted with Big-O. Therefore by ignoring the constant a, b,c and the lower terms of n, and taking only the dominant term i.e. n2 , then the asymptotic running time of selection sort will be of the order of O(n2) and will lie in of set of asymptotic function i.e Θ(n2). Hence we can say that the asymptotic running time of old SS will be:

$T(n) = \Theta(n2)$

It means that the best and worst case asymptotic running time of selection sort is same in the term of order of asymptotic notation i.e.

T(n)=Θ(n2), however there may be little difference in actual running time, which will be very less and hence ignored

2.2.2 Dual nature selection sort algorithm

DNSSA (A,n)

```
k ← 0;
fori ← n – 1 to k;
IndexOfLarge←IndexOfSmall←k;
forj←k + 1 to i
{
if (X[j] > X(IndexOfLarge])
IndexOfLarge ← j;
if (X[j] < X[IndexOfSmall])
IndexOfSmall← j;
}
Large ← X[IndexOfLarge];
Small ← X[IndexOfSmall];
X[IndexOfLarge] ← X[i];
X[IndexOfSmall] ← X[k];
if (IndexOfLarg==k)
  {          Temp=X[i];
X[i]=Large;
X[IndexOfSmall]=Temp;
}
Else X[i] ← Large
if (IndexOfLarg == k)
  {   Temp = X[k];
X[k] = Small;
X[IndexOfLarge] = Temp;
}
Else X[k] ← Small;
k ← k + 1:
```

Here we evaluate the Total Execution Time of FSSA for the given data of n elements. Firstly we simplify the execution time of some statements in terms of dependent variable n.

Table 2: Complexity of DNSSA

Pseudo Code Instruction	Instruction Exe Time	How Many times the instruction is exe. By CPU
k ← 0;	C_1	1
for i ← n – 1 to k;	C_2	n/2+1
IndexOfLarge←IndexOfSmall←k;	C_3	n/2
for j←k + 1 to i	C_4	$\sum_{i=1}^{n/2}(i+1)$
{	C_5	$\sum_{i=1}^{n/2} i$
if (X[j] > X(IndexOfLarge])	C_6	$\sum_{i=1}^{n/2} i(t_{ij})$
IndexOfLarge ← j;	C_7	$\sum_{i=1}^{n/2} i$
if (X[j] < X[IndexOfSmall])	C_8	$\sum_{i=1}^{n/2} i(t_{ij})$
IndexOfSmall← j;	C_9	n/2
}	C_{10}	n/2
Large ← X[IndexOfLarge];	C_{11}	n/2
Small ← X[IndexOfSmall];	C_{12}	n/2
X[IndexOfLarge] ← X[i];	C_{13}	n/2
X[IndexOfSmall] ← X[k];		
if (IndexOfLarg==k)		
{ Temp=X[i];		
X[i]=Large;		
X[IndexOfSmall]=Temp;		
}		
Else X[i] ← Large	C_{14}	
if (IndexOfLarg == k)		
{		n/2
Temp = X[k];		
X[k] = Small;		
X[IndexOfLarge] = Temp;		
}		
Else X[k] ← Small;	C_{15}	
k ← k + 1:		n/2

Note that

- $t_{i,j} = 1$ when the if statement is true, 0 otherwise
- $C_4\sum_{i=1}^{n/2}(i+1) = C_4\sum_{i=1}^{n/2} i + C_4\sum_{i=1}^{n/2} 1$

$= C_4 n/2(n/2+1)/2 + C_4(n/2).$

- $C_5\sum_{i=1}^{n/2} i = C_5\, n/2(n/2+1)/2 = C_5 n^2/8 + C_5\, n/4$

When $t_{ij}=1$ & 0 otherwise.

$C_6\sum_{i=1}^{n/2} i = C_6\, n/2(n-1)/2 = C_6 n^2/8 + C_6 n/4$

More over the instructions enclosed in dashed and dotted box of Pseudo code of DNSSA will rarely be executed, i.e chance of their execution is less, mostly the Else section of each If statement will execute. So we are considering the total execution of If statement in case of either true or false logical condition.

Best Case Time Complexity

Then for the best-case scenario we have that all tij = 0 so we have

$T(n) = C_1 + C_2 n/2 + C_{2+} C_3 n/2 + C_4 n/2(n/2+1)/2 + C_4(n/2) + C_5 n^2/8 + C_5 n/4 + C_7 n^2/8 + C_7 n/4 + C_9/9 + C_9 n/2 + C_{10} n/2 + C_{11} n/2 + C_{12} n/2 + C_{13} n/2 + C_{14} n/2 + C_{15} n/2$

$T(n) = n^2(C_4 8 + C_5/8 + C_7/8) + n(C_2/2 + C_3/2 + C_4/4 + C_4/2 + C_5/4 + C_7/4 + C_9/2 + C_{10}/2 + C_{11}/2 + C_{12}/2 + C_{13}/2 + C_{14}/2 + C1_5/2) - (C_2 + C_3 + C_7 + C_8)$

Let

$a = (C_4 8 + C_5/8 + C_7/8).$

$b = (C_2/2 + C_3/2 + C_4/4 + C_4/2 + C_5/4 + C_7/4 + C_9/2 + C_{10}/2 + C_{11}/2 + C_{12}/2 + C_{13}/2 + C_{14}/2 + C1_5/2)$

$c = C_1 + C_2$

Then T(n) becomes

$$\mathbf{T(n) = a n^2 + bn + c}$$

Thus here in best-case, the complexity of execution time of an algorithm shows the lower bound and is asymptotically denoted with Ω. Therefore by ignoring the constant a, b, c and the lower terms of n, and taking only the dominant term i.e. n2, then the asymptotic running time of selection sort will be Ω(n2) and will lie in of set of asymptotic function i.e. Θ(n2). Hence we can say that the asymptotic running time of Optimized Selection Sort Algorithm (DNSSA) will be:

$$\mathbf{T(n) = \Theta(n2)}$$

Worst Case Time Complexity

Then for the best-case scenario we have that all tij = 1 so we get

$T(n) = C_1 + C_2 n/2 + C_{2+} C_3 n/2 + C_4 n/2(n/2+1)/2 + C_4(n/2) + C_5 n^2/8 + C_5 n/4 + C_7 n^2/8 + C_7 n/4 + C_9/9 + C_9 n/2 + C_{10} n/2 + C_{11} n/2 + C_{12} n/2 + C_{13} n/2 + C_{14} n/2 + C_{15} n/2$

$T(n) = n^2(C_4/8 + C_5/8 + C_6/8 + C_7/8 + C_8/8) + n(C_2/2 + C_3/2 + C_4/4 + C_4/2 + C_5/4 + C_6/4 + C_7/4 + C_8/4 + C_9/2 + C_{10}/2 + C_{11}/2 + C_{12}/2 + C_{13}/2 + C_{14}/2 + C_{15}/2) - (C_2 + C_3 + C_7 + C_8)$

……..2.2.2.a

Let

$a = (C_4/8 + C_5/8 + C_6/8 + C_7/8 + C_8/8).$

$b = (C_2/2 + C_3/2 + C_4/4 + C_4/2 + C_5/4 + C_6/4 + C_7/4 + C_8/4 + C_9/2 + C_{10}/2 + C_{11}/2 + C_{12}/2 + C_{13}/2 + C_{14}/2 + C_{15}/2)$

$c = C_1 + C_2$

Then T(n) becomes-

$$T(n) = a n^2 + bn + c$$

Thus here in worst-case, the complexity of execution time of an algorithm shows the upper bound and is asymptotically denoted with Big-O. Therefore by ignoring the constant a, b, c and the lower terms of n, and taking only the dominant term i.e. n2, then the asymptotic running time of selection sort will be of the order of O(n2) and will lie in of set of asymptotic function i.e Θ(n2). Hence we can say that the asymptotic running time of Optimized Selection Sort Algorithm (OSSA) will be:

$T(n) = C4/8 + C5/8 + C6/8 + C7/8 + C8/8).)n2$

Assume that

$C_1 = C_2 = C_3 = C_4 = C_5 = C_6 = C_7 = C_8 = C$…… (A)

Hence-

$T(n) = 5C/8(n^2)$ ……..2.2.2.b

Hence the Complexity of DNSSA

$$\mathbf{T(n) = \Theta(n^2)}$$

Thus here in worst-case, the complexity of execution time of an algorithm shows the upper bound and is asymptotically denoted with Big-O. Therefore by ignoring the constant a, b,c and the lower terms of n, and taking only the dominant term i.e. n2 , then the asymptotic running time of selection sort will be of the order of O(n2) and will lie in of set of asymptotic function i.e Θ(n2). Hence we can say that the asymptotic running time of old SS will be:

$$\mathbf{T(n) = \Theta(n2)}$$

It means that the best and worst case asymptotic running time of selection sort is same i.eT(n)=Θ(n2), however there may be little difference in actual running time, which will be very less and hence ignored.

Comparison of Complexity of Fundamental Selection Sort and DNSSA

From equation.2.2.1.a and (A)

Complexity of FSS

T(n)= 3C/2(n2)

And Complexity of DNSSA (from equation 2.2.2.b)

T(n) =5C/8(n2)

$$\frac{\text{Complexity of FSSA}}{\text{Complexity of DNSS}} = \frac{3C/2(n^2)}{5C/8(n^2)} = 0.41$$

Hence DNSS algorithm same almost half time in comparison of FSS that is DNSS algorithm is better than the FSS algorithm, And it is the optimization of Selection Sort.

For better understand the improvement in complexity of DNSSA in comparison of Selection Sort Algorithm-

Suppose that-

Suppose that-

$C_1 = C_2 = C_3 = C_4 = C_5 = C_6 = C_7 = C_8 = C_9 = C_{10} = C_{11} = C_{12} = C_{13} = C_{14} = C_{15} = C_{16} = 1$ ..... (B)

Then Complexity of FSSA- from the equation..(2.1.1.a) and (B)

$T(n) = 3/2(n^2) + 9/2(n) + 5.$

Hence $T(100) = 15445$

Complexity of DNSSA- from equation.. (6.1.a)

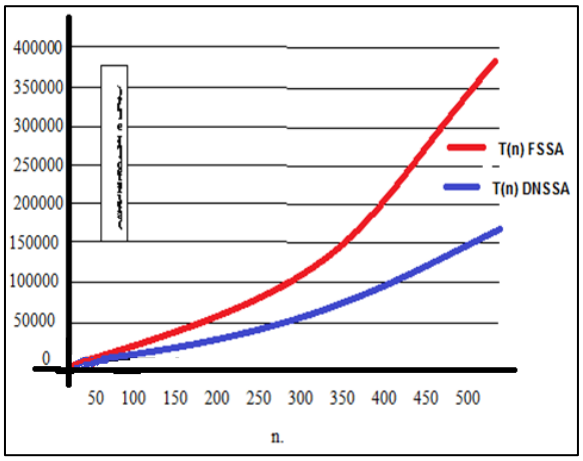
$T(n) = 5/8(n^2) + 25/4(n) + 2.$

Hence $T(100) = 6877$

Below is the table representing the calculated time, when multiple values of n are used.

n	Calculated Time of FSSA	Calculated Time of DNSSA	Percentage Improvement
n	T(n)	T(n)	
50	3970	1877	52.7%
100	15445	6877	55.5%
150	34420	15002	56.4%
200	60895	26252	56.9%
250	94870	40627	57.2%
300	136345	58127	57.4%
350	185320	78752	57.5%
400	241795	102502	57.6%
450	305770	129377	57.7%
500	377245	159377	57.8%

Fig 4: Line Graph Plotted in Response of Data



From above graph, one can easily observe that with the increase in number of data elements, the DNSSA takes less time as compared to the Old SS and it will be true for any number of data.

Advantage:

1. Simple and easy to implement.
2. It is faster than the FSSA.

Disadvantage:

Inefficient for large lists, so similar to the more efficient insertion sort, the insertion sort should be used in its place.

2.3 Dual nature bubble sort algorithm

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage.

Dual Nature Bubble Sort is optimised Sorting algorithm of Fundamental Bubble Sort Algorithm, DNBSA. In DNBS algorithm the array is sorted from both side left side

Dual Nature Selection Sort

For i←1 to lowerbound(n/2)
For j← (n-i+1) to i+1
If A[j] <A[j-1]
Exchange (A[j],A[j-1])
If A[n-j+1] > A[n-j+2]
Exchange (A[n-j+1],A[n-j+2])

Complexity of DNBS Algorithm

Dual Nature Bubble Sort Cost Times

For i←1 to lowerbound(n/2) C_1 n/2
For j← (n-i+1) to i+1 C_2 (n-1)+(n-3)+(n-5)+…+1
If A[j] <A[j-1] C_3 (n-1)+(n-3)+(n-5)+…+1
Exchange(A[j],A[j-1]) C_4 (n-1)+(n-3)+(n-5)+…+1
If A[n-j+1] > A[n-j+2] C_5 (n-1)+(n-3)+(n-5)+…+1
Exchange(A[n-j+1],A[n-j+2]) C_6 (n-1)+(n-3)+(n-5)+…+1

Complexity of DNBS

$\mathbf{T(n)} = C_1(n/2)+(C_2+C_3+C_4+C_5)[n(n/2)-(n/2)^2$
$= C_1(n)+C_2[n^2/2] – C_2(n)/2+C_3[n^2/2] –$
$C_3(n)/2+c_4[n^2/2] – C_4(n)/2$
$T(n) = (C_2+C_3+ C_4+C_5+C_6)(n^2/2-n^2/4)+(C_1)n/2$
……2.3.a
Hence- $T(n) = (C_2+C_3+C_4+C_5+C_6)(n^2)/4+ C_1(n)$
Assume that $C_2=C_3=C_4= C_5=C_6= C.$
$T(n) = 5C(n^2)/4$ ……2.3.b
$\rightarrow \mathbf{T(n) = O(n^2)}$

2.3.1 Bubble sort algorithm

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. In general case its complexity is O(n2)

Algorithm

Bubble Sort (A, n)

1. Fori ← 1 to n
2. For j ← n to i+1
3. If A[j] <A[j-1]
4. Exchange(A[j],A[j-1])

Complexity Analysis of Bubble Sort Algorithm

Bubble Sort	cost	times
For i←1 to n	C_1	n
For j← n to i+1	C_2	(n-1)+(n-2)+……..+1
If A[j] <A[j-1]	C_3	(n-1)+(n-2)+……..+1
Exchange(A[j],A[j-1])	C_4	(n-1)+(n-2)+……..+1

Complexity of Bubble Sort-

$\mathbf{T(n)}= C_1(n)+C_2[n(n-1)/2]+C_3[n(n-1)/2]+C_4[n(n-1)/2] =$
$C_1(n)+C_2[n^2/2]–C_2(n)/2+C_3[n^2/2]–C_3(n)/2+C_4[n^2/2]–C_4(n)/2$
$T(n)=(C_2+C_3+C_4)(n^2)/2+(C_1-C_2/2–C_3/2–C_4/2)n$…2.3.1.a
Hence- $T(n) = (C_2+C_3+C_4)(n^2)/2$
Assume that
$C_2=C_3=C_4=C.T(n)=3C(n^2)/2$ ……2.3.1.b
$\rightarrow \mathbf{T(n) = O(n^2)}$

Comparision of Bubble Sort and Dnbsa

Complexity of Bubble Sort- $T(n) = 3C/2(n^2)$
Complexity of DNBS- $T(n) = 5C/4(n^2)$

$$\frac{\text{Complexity of BSA}}{\text{Complexity of DNBS}} = \frac{3C/2(n^2)}{5C/4(n^2)}$$
$=0.81$

The complexity of DNBS is lower than the Bubble Sort, and it save the one by fourth of Bubble Sort algorithm. Hence DNBS has the better efficiency than the Bubble Sort.

For better understand the improvement in complexity of DNBSA in comparison of Bubble Sort Algorithm-

Suppose that-

$C_1 = C_2 = C_3 = C_4 = C_5 = C_6 = C_7 = C_8 = C_9 = C_{10} = C_{11} = C_{12} = C_{13} = C_{14} = C_{15} = C_{16} = 1$

Then

Complexity of BSA- from the equation..(2.3.1.a)

$T(n) = 3/2(n^2) - (n)/2.$

Hence $T(100) = 15445$

Complexity of DNBSA- from equation.. (2.3.a)

$T(n) = 5/4(n^2) + n.$

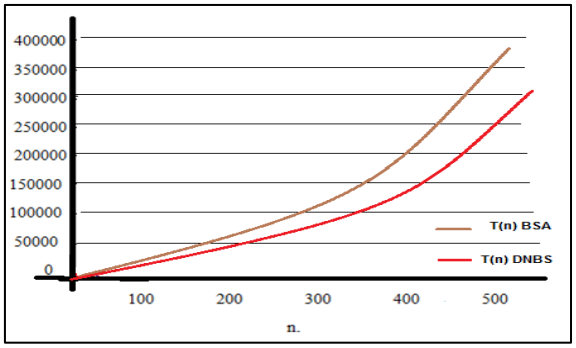
Hence $T(100) = 6877$

Below is the table representing the calculated time, when multiple values of n are used.

n.	Calculated Time of BSA	Calculated Time of DNBSA	Percentage Improvement
n.	T(n)	T(n)	
50	3725	3175	15.0%
100	14950	12600	16.0%
200	599000	50200	16.40%
300	134850	112800	16.50%
400	239800	200400	16.55%
500	374750	313000	16.60%

Below is the line graph plotted in response of data given in above table.

Fig: 5. Efficiency of DNBS



From above graph, one can easily observe that with the increase in number of data elements, the DNSA takes less time as compared to the BS and it will be true for any number of data.

Advantage:

1. Simplicity and ease of implementation.
2 It has better efficiency than Bubble Sort

2.4 Insertion sort within the merge sort

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of O(n log n).Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. There are 14 non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machine

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures – the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Like the bubble sort the insertion sort has a complexity of O(n2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.

Insertion sort is probably the first sorting algorithm that is taught in programming classes. It is far from efficient in terms of the number of comparisons, but is very compact in terms of code space required. As such, the insertion sort algorithm may be useful for sorting small lists.

Algorithm

ISwMS (A, p, r)

1. If $p < r$
2. Then $q \leftarrow$ lower bound of $(p+r)/2$
3. If$(p-q > 24)$
4. Merge Sort(A, p, q)

```
5.  Merge Sort(A,q+1, r)
    Else
6.  Insertion Sort(A, p, q)
7.  Insertion Sort(A, q+1, r)
8.  Merge(A, p, q, r)
Merge (A, p, q, r)
1.  n1 ← q-p+1
2.  n2 ← r-p
3.  For i← 1 to n1
4.  do L[i] ← A[p+i-1]
5.  For j ←1 to n2
6.  do R[j] ←A[q+j]
7.  L[n1 +1] ←infinite
8.  R[n2 +1]  ←infinite
9.  i  ← 1
10. j ← 1
11. For k←p to r
12. do if  L[i] ≤ R[j]
13. A[k] ←L[i]
14. i← i+1
15. Else A[k]←R[j]
16. j ←j+1
Insertion Sort (A, p, q)
n← q-p
for j ←  2 to n
do key ← A[j]
i← j -1
 While i> 0 && A[i] > key
do A[i+1] ← A[i]
i←i -1
A[i+1] ← key
```

Comparison of Insertion Sort and Merge Sort

From the above equation of Complexity of Insertion Sort and Merge Sort

Complexity of Merge Sort- T(n) = 8C(nlgn)

Complexity of Insertion Sort- T(n) = 3C/2(n2)

Complexity of Merge Sort >= Complexity of Insertion Sort

8C(nlgn) >= 3C/2(n2)

16lg n>= 3n

Hence n<=24. That is for n less than and or equal to 24 the Insertion Sort algorithm beats the Merge Sort algorithm. Hence the Mer Sort can be optimised by using the Insertion Sort for less than or equal to the specific range of number of elements.

2.5 Optimised binary search with quick Sort

Optimised binary search with quick sort algorithm is the new searching algorithm which is used to search the desired element in unsorted array with the concept of quick sort and binary search. So this is the optimised algorithm for searching the element.

One of the most frequent operations performed on database is searching. To perform this operation we have different kinds of searching algorithms, some of which are Binary Search, Index Sequential Access Method (ISAM), but these and all other searching algorithms work only on data, which are previously sorted. An efficient algorithm is required in order to make the searching algorithm fast and efficient. This research paper presents a new sorting algorithm named as "Optimized Binary Search With Quick Sort Algorithm, "OBSQS". Optimised Binary Search with Quick Sort is designed to perform searching quickly and more effectively as compared to the existing version of searching algorithm. The introduction of OBSQS version of searching algorithm for searching the data stored in data base.

The complexity of this algorithm is better than the complexity of existing search algorithms. In Optimised Binary Search with Quick Sort the desired element is searched without sorting the array and this algorithm use the both concepts of binary search and quick sort, quick sort is used to partition the array into two parts and part which have the value is taken and further proceed to completion the searching of the element. Hence Optimised Binary Searching Algorithm is the combination of both Binary Search Algorithm as well as the Quick Sort Algorithm. Dividation of the array into two part is done by quickpoint value which is find out with quick sort quickpoint function, and then taken a part by the binary search algorithm's concept.

In this searching algorithm"Optimised Binary Search with Quick Sort" the concept is used both binary search and quick sort algorithm. In Optimised Binary Search with Quick Sort the desired element is searched without sorting the array, and this algorithm use the both concepts of binary search and quick sort, quick sort is used to partition the array into two parts and part wich have the value is taken and further proceed to completion the searching of the element. Hence Optimised Binary Searching Algorithm is the combination of both Binary Search Algorithm as well as the Quick Sort Algorithm. Dividation of the array. The complexity is better than other searching algorithm, and the main advantages of this algorithm are that it is used for unsorted array. In this algorithm the array is divided into almost equally two part after the Checking of quickpoint value and after that our is divided into almost half of the of starting array and so on until the element is not fount or the array is searched completely.

Algorithm

Optimised binary search with quick sort algorithm is the searching algorithm which is used to search the desired element in unsorted array with the concept of quick sort and binary search. So this is the optimised algorithm for searching the element. To

perform this operation we have different kinds of searching algorithms, some of which are Binary Search, Index Sequential Access Method (ISAM), but these and all other searching algorithms work only on data, which are previously sorted. An efficient algorithm is required in order to make the searching algorithm fast and efficient of OBSQS version of searching algorithm for searching the data stored in data base.

This algorithm is perform the Divide and conquer approach as in the quick sort is used but the major difference is that it takes only one pat after checking the desired item with quickpoint value and hence the complexity is better than quick sort and existing searching algorithm-

Divide- Divide the array almost into two part and take the part which has the desired element.

Conquer- Perform dividing and checking with item till the desired value found or the array is completely searched. **Combine**- Nothing.

Hence the optimised Search with quick sort algorithm is the optimised searching algorithm which is optimised with use of concept of binary search and quick sort algorithm. Hence Optimised Binary Searching Algorithm is the combination of both Binary Search Algorithm as well as the Quick Sort Algorithm. Dividation of the array. The complexity is better than other searching algorithm, and the main advantages of this algorithm is that it is used for unsorted array. In this algorithm the array is divided into almost equally two part after the Checking of quickpoint value and after that our is divided into almost half of the of starting array and so on until the element is not fount or the array is searched completely. So that the complexity of searching algorithm become better and efficient good and so that the efficiency become better that given algorithm.

Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment.

Many of these issues are best dealt with at the algorithmic level, rather than by tweaking the code.

OBSWQS Search (A, p, r)

```
If  p< r
      {
If  A[r]==key
then  Item is found.
        Else
         {
              q←quickpoint(A, p, r)
             If A[i+1]>key
            Then Search(A,p,q-1)
            Else Search(A,q+1,r)
         }
  }
 Else   Item not found.
quickpoint(A, p, r)
   X←A[r]
i←p-1
for j←p to r-1
if A[j]<=x
i←i+1
exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1.
```

Performance of Optimised Binary Search with Quick Sort

The running time of optimised binary search with quick sort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge. Complexity of this is become better than binary search in the sense it use unsorted array instead of sorted array and it is better than linear search in the sense of the complexity, it is more efficient than linear search algorithm.

1. **By the Master Theorem.**

$T(n) = T(n/2) + O(1)$.

Best Case Complexity.

$T(n) = \Omega(1)$.

Average Case Complexity

$T(n) = \Theta(\lg n)$

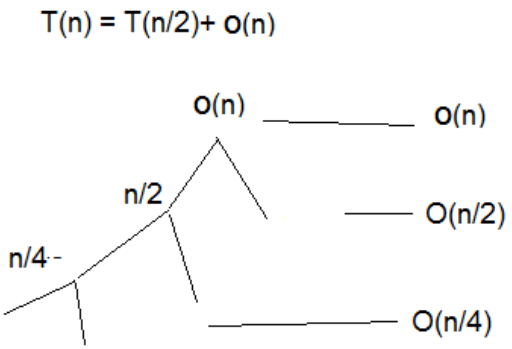
Worst Case Complexity.

$T(n) = O(n)$

Hence the complexity of this searching algorithm can be expressed in all three asymptotic notation of the complexity analysis.

Complexity Analysis

Fig 6: QSwMS Tree



Complexity= Complexity of Checking+ Complexity of Breacking

=O(n)+O(n/2)+O(n/4)+….til 2i

Now calculate i.

2i =n. i=lgn.

That is Comlexity in order form and eliminating the minimum term.

T(n)=O(n-2)

Advantages

1. It is fast and efficient.
2. It has better complexity than other searching algorithms.
3. It is used for the unsorted array.

Disadvantages

1. There is used the partition concept of the quick sort and hence it increases the complexity of the algorithm.

It has the more complexity than binary search algorithm in the sense of using the quick sort's partition concept.

References

[1] Y. Han, Deterministic sorting in O(nlog log n) time and linear space, Proceedings ofthe thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec Canada, 2002, 602-608

[2] M. Thorup, Randomized Sorting in O (n log log n) Time and Linear Space UsingAddition, Shift, and Bit-wise Boolean Operations, Journal of Algorithms, 42(2), 2002, 205-230

[3] Y. Han, M. Thorup, Integer Sorting in O(n (log log n) Time and Linear Space, Proceedings of the 43rd Symposium on Foundations of Computer Science, 2002, 135-144

[4] P. M. McIlroy, K. Bostic, M. D. McIlroy, Engineering radix sort, Computing Systems, 2004, 224-230

[5] M. D. McIlroy, A killer adversary for quick sort, Software--Practice andExperience,1999, 123-145

[6] I. Flores, Analysis of Internal Computer Sorting, J.ACM 7, 4, 1960, 389- 409

[7] G. Franceschini, V. Geffert, An In-Place Sorting with O (n log n) Comparisons and O (n) Moves, In Proc. 44th Annual IEEE Symposium on Foundations of Computer Science, 242-250, 2003

[8] D. Knuth, The Art of Computer programming Sorting and Searching, 2nd edition, 3, Addison- Wesley, 1998

[9] C. A. R. Hoare, Algorithm 64: Quick sort, Comm. ACM 4, 7, 1961, 321

[10] Soubhik Chakraborty, Mausumi Bose, and Kumar Sushant, A Research thesis, On Why Parameters of Input Distributions need be taken into Account for a More Precise Evaluation of Complexity for Certain Algorithms

[11] D.S. Malik, C++ Programming: Program Design Including Data Structures, Course Technology (Thomson Learning), 2002, www.course.com

[12] D. Jim´enez-Gonz´alez, J. Navarro, and J. Larriba-Pey. CC-Radix: A Cache Conscious Sorting Based on Radix Sort, In Euromicro

Conference on Parallel Distributed and Network based Processing, 101-108, 2003

[13] J. L. Bentley, R. Sedgewick, Fast Algorithms for Sorting and Searching Strings", ACM-SIAM SODA, 97, 360-369, 1997

[14] I. Flores, Analysis of Internal Computer Sorting, J.ACM 8, 1961, 41-80

[15] J. W. J. Williams, Algorithm 232: Heap sort". Comm. ACM 7, 6, 1964, 347-348

[16] A. Andersson, S. Nilsson, 1994, A New Efficient Radix Sort". In the Proceeding of the 35 Annual IEEE Symposium on Foundation of Computer Science, 1994, 714-721

[17] I. J. DAVIS, A Fast Radix Sort, The computer journal 35, 6, 636-642, 1992

[18] V. Estivill-Castro, D. Wood, A Survey of Adaptive Sorting Algorithms, Computing Surveys, 24:441-476, 1992

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2, 2001