

Report on Sorting Algorithms

December 7, 2014

Submitted to:

Dr. Md. Sanaullah Chowdhury
Associate Professor
Department of Computer Science
Engineering
University of Chittagong

Submitted by:

Kavi Zahan Sultana
ID: 13701017
Department of Computer Science
Engineering
University of Chittagong
Date of Submission
08/12/2014

Contents

0.1	Introduction	4
0.2	Insertion Sort	4
0.2.1	Algorithm	4
0.2.2	Time Complexity of Insertion Sort	4
0.2.3	Graph of Time Complexity of In- sertion Sort	7
0.3	Merge Sort	8
0.3.1	Algorithm	9
0.3.2	Time Complexity of MERGE-SORT	10
0.3.3	Graph of Time Complexity of MERGE- SORT	11
0.4	Bubble Sort	11
0.4.1	Algorithm of BUBBLE SORT . .	11
0.4.2	Time Complexity of BUBBLE SORT	12
0.4.3	Time Complexity Graph of BUBBLE- SORT	12
0.5	Quick Sort	13
0.5.1	Algorithm of Quick Sort	13
0.5.2	Time Complexity of Quick Sort .	14
0.5.3	Time Complexity Graph of Quick Sort	15
0.6	Heap Sort	15

0.6.1	Algorithm of Heap Sort	15
0.6.2	Time Complexity of Heap Sort .	16
0.6.3	Time Complexity Graph	17

0.1 Introduction

This report provides the introduction, algorithm, running time and graphs of sorting algorithms.

0.2 Insertion Sort

Insertion sort is an efficient algorithm for sorting a small number of elements. We present the pseudocode called INSERTION-SORT. It takes as a parameter an array $A[1..n]$ containing a sequence of length n that is to be sorted.

0.2.1 Algorithm

INSERTION – SORT(A)

```
1 for  $j = 2$  to  $A.length$ 
2    $key = A[j]$ 
3    $i = j - 1$ 
4   while  $i > 0$  and  $A[i] > key$ 
5      $A[i + 1] = A[i]$ 
6      $i = i - 1$ 
7    $A[i + 1] = key$ 
```

0.2.2 Time Complexity of Insertion Sort

INSERTION – SORT(A) *cost* *times*

```
1 for  $j = 2$  to  $A.length$                        $c_1$      $n$ 
```

2	$key = A[j]$	c_2	$n - 1$
3	$i = j - 1$	c_3	$n - 1$
4	while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^n t_j$
5	$A[i + 1] = A[i]$	c_5	$\sum_{j=2}^n (t_j - 1)$
6	$i = i - 1$	c_6	$\sum_{j=2}^n (t_j - 1)$
7	$A[i + 1] = key$	c_7	$n - 1$

The running time of the algorithm is the sum of running times for each statement executed. To compute $T(n)$, the running time of *INSERTION-SORT* on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n - 1)$$

1. Best case:

In *INSERTION-SORT*, the best case occurs if the array is already sorted. For each $j = 2, 3, \dots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 2, 3, \dots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

$$= an + b$$

where, $a = (c_1 + c_2 + c_3 + c_4 + c_7)$ and $b = -(c_2 + c_3 + c_4 + c_7)$, which depend on the statement costs c_i ; it is thus a ***linear function*** of n .

2. Worst case:

If the array is in reverse sorted order-that is, in decreasing order- the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j - 1]$, and so $t_j = j$ for $j = 2, 3, \dots, n$. Noting that

$$\sum_{j=2}^n j = n(n + 1)/2 - 1$$

and

$$\sum_{j=2}^n (j - 1) = n(n - 1)/2$$

Therefore, in worst case, the running time of *INSERTION-SORT* is

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_3(n-1) + c_4(n(n+1)/2 - 1) + c_5(n(n-1)/2) \\ &= (c_4/2 + c_5/2 + c_6/2)n^2 + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_8)n \\ &= an^2 + bn + c \end{aligned}$$

which is a ***quadratic equation***. Here, $an^2 + bn + c$ for constants a, b and c that again depend on the statement costs c_i ; it is thus a ***quadratic function*** of n .

3.Average case:

The **average case** is often roughly as bad as the worst case. On average half the elements in $A[1..j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1..j-1]$, and so t_j is about $j/2$. The resulting average case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Here is the chart of time complexity of insertion sort.

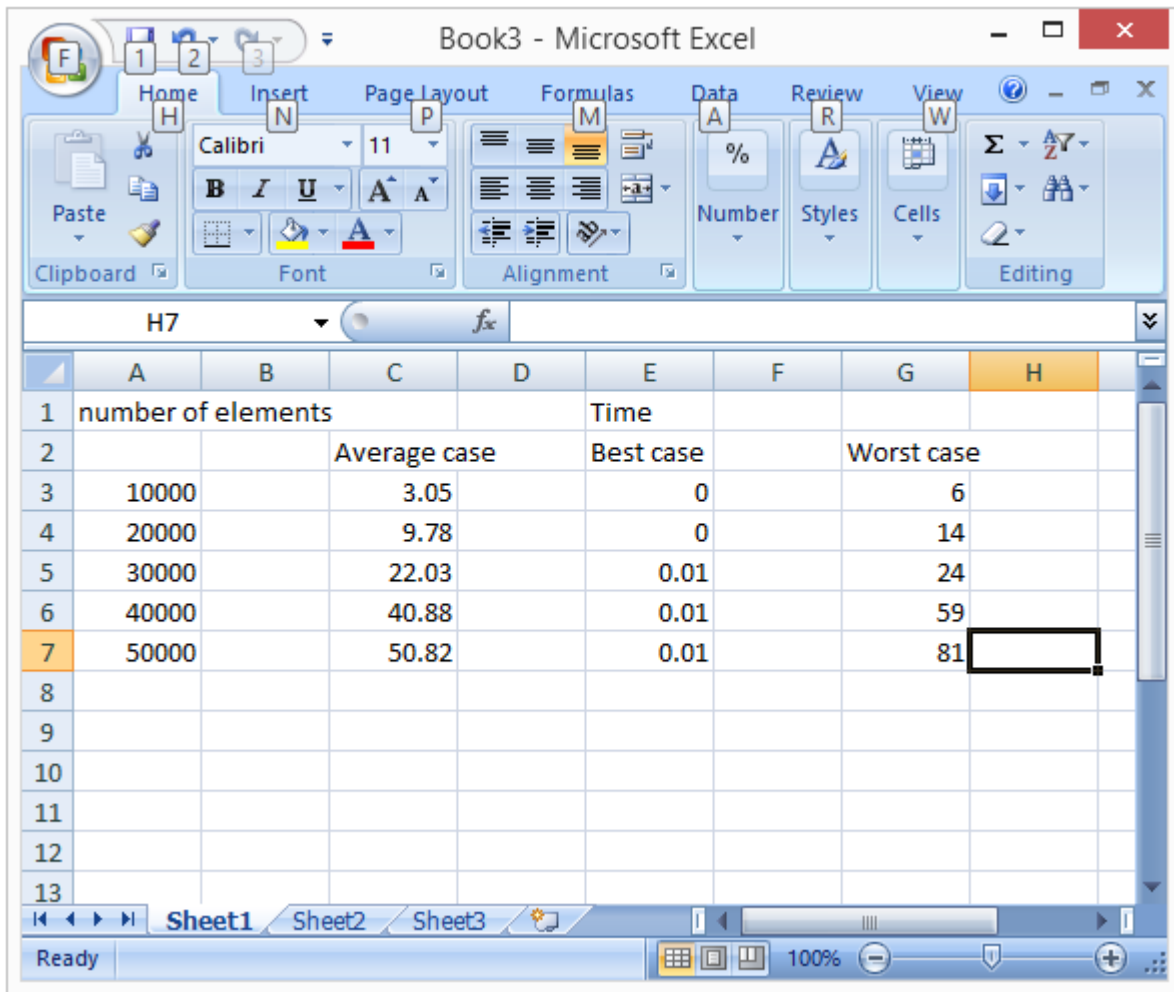


fig: Chart of Insertion Sort.

0.2.3 Graph of Time Complexity of Insertion Sort

Here is the graph of comparison of running times of average case, best case and worst case.

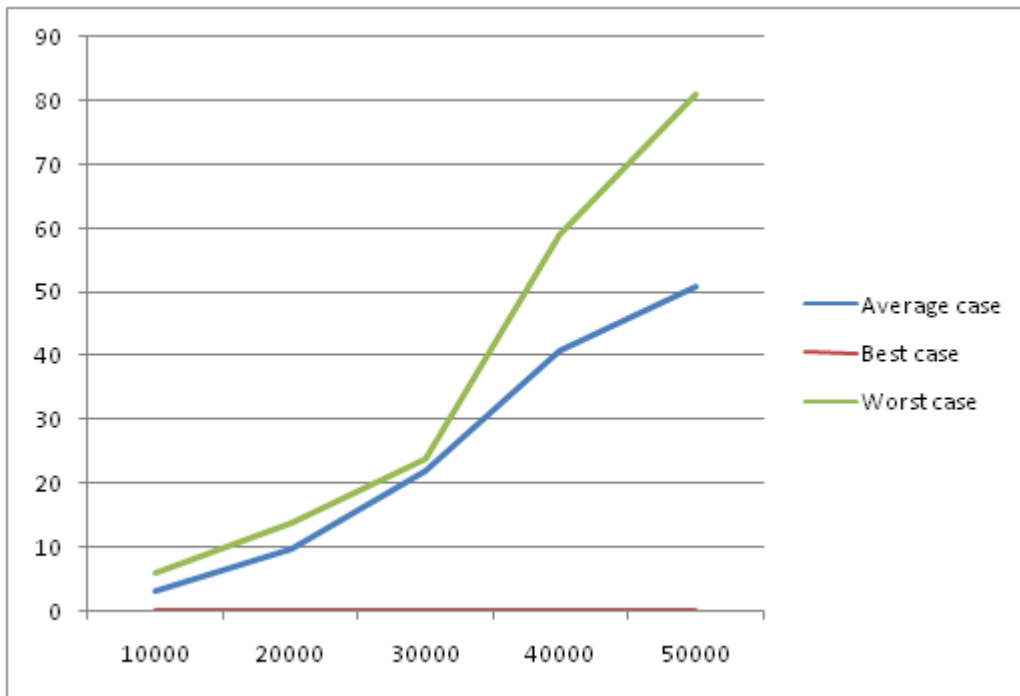


fig: time vs n graph for insertion sort.

0.3 Merge Sort

The ***Merge Sort*** algorithm closely follows the divide and conquer paradigm. Intuitively, it operates as follows,

Divide:

Divide the n - element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer:

Sort the two subsequences recursively using merge sort.

Combine:

Merge the two sorted subsequences to produce the sorted answer.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the **combine** step. We merge by calling an auxiliary procedure $MERGE(A, p, q, r)$, where A is an array and p, q and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarray $A[p..q]$ and $A[q + 1..r]$ are in sorted order. It **merges** them to form a single sorted subarray that replaces the current subarray $A[p..r]$.

Our $MERGE$ procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged.

0.3.1 Algorithm

The following pseudocode implements $MERGE(A, p, q, r)$ procedure, which merges the two subsequences. And another pseudocode $MERGE - SORT(A, p, r)$ sorts the elements in the subarray $A[p..r]$.

```
 $MERGE(A, p, q, r)$   
1   $n_1 = q - p + 1$   
2   $n_2 = r - q$   
3  //create arrays  $L[1...n_1 + 1]$  and  $R[1...n_2 + 1]$ .  
4  for  $i = 1$  to  $n_1$   
5    do  $L[i] = A[p + i - 1]$ 
```

```

6  for  $j = 1$  to  $n_2$ 
7    do  $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else  $A[k] = R[j]$ 
17      $j = j + 1$ 

```

MERGE – SORT(A, p, r)

```

1  if  $p < r$ 
2     $q = \lfloor (p + r) / 2 \rfloor$ 
3    MERGE – SORT( $A, p, q$ )
4    MERGE – SORT( $A, q + 1, r$ )
5    MERGE( $A, p, q, r$ )

```

0.3.2 Time Complexity of MERGE-SORT

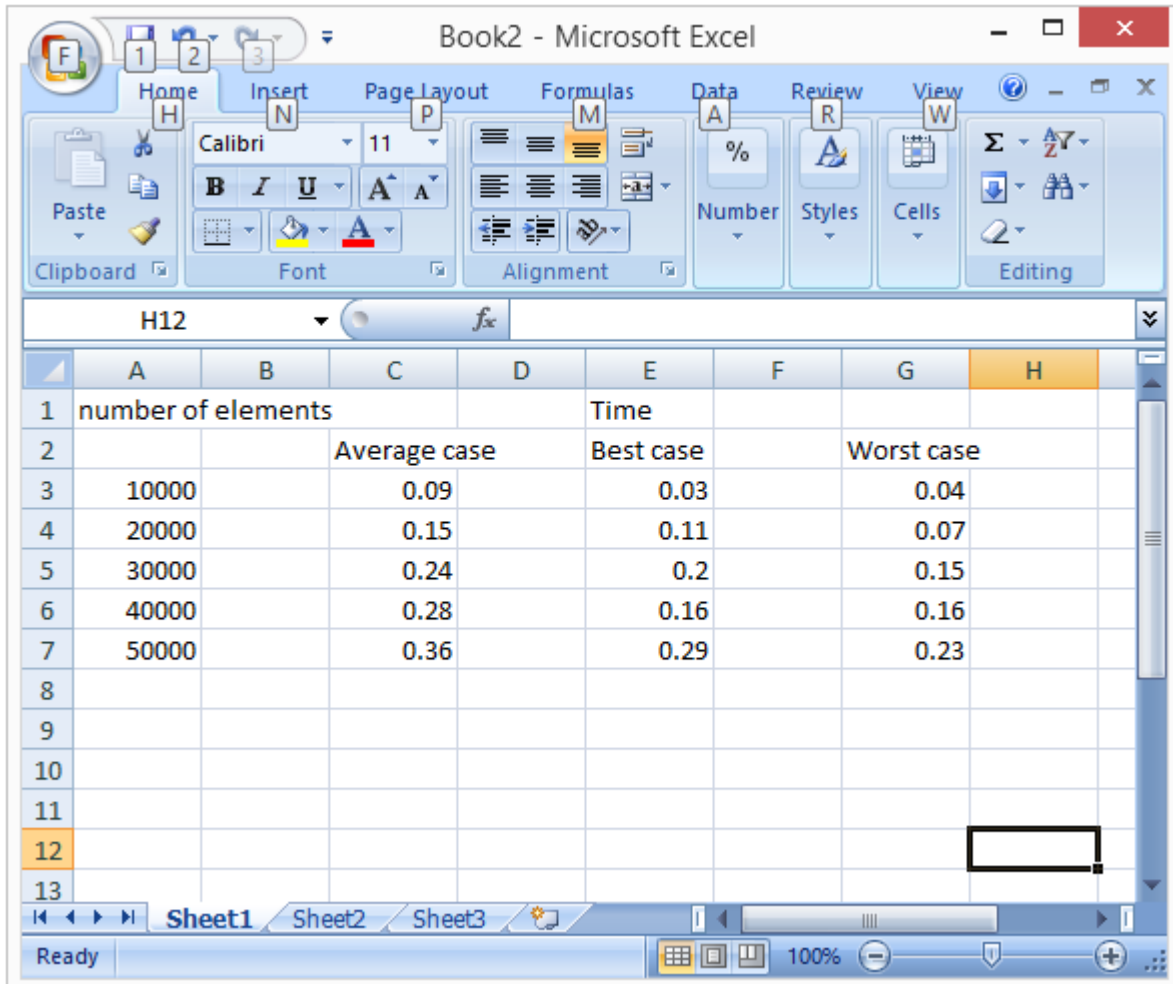
There are no such differences among the best, worst and average case of time complexity in merge sort.

Best case: In best case, the running time of merge sort is, $T(n) = O(n \log(n))$.

Average case: In average case, the running time of merge sort is, $T(n) = O(n \log(n))$.

Worst case: In worst case, the running time of merge sort is, $T(n) = O(n \log(n))$.

Here, is the chart of time complexity in merge sort-



	A	B	C	D	E	F	G	H
1	number of elements				Time			
2			Average case		Best case		Worst case	
3	10000		0.09		0.03		0.04	
4	20000		0.15		0.11		0.07	
5	30000		0.24		0.2		0.15	
6	40000		0.28		0.16		0.16	
7	50000		0.36		0.29		0.23	
8								
9								
10								
11								
12								
13								

fig: Time Complexity chart.

0.3.3 Graph of Time Complexity of MERGE-SORT

We can represent time complexity of *MERGE-SORT* by the following graph.

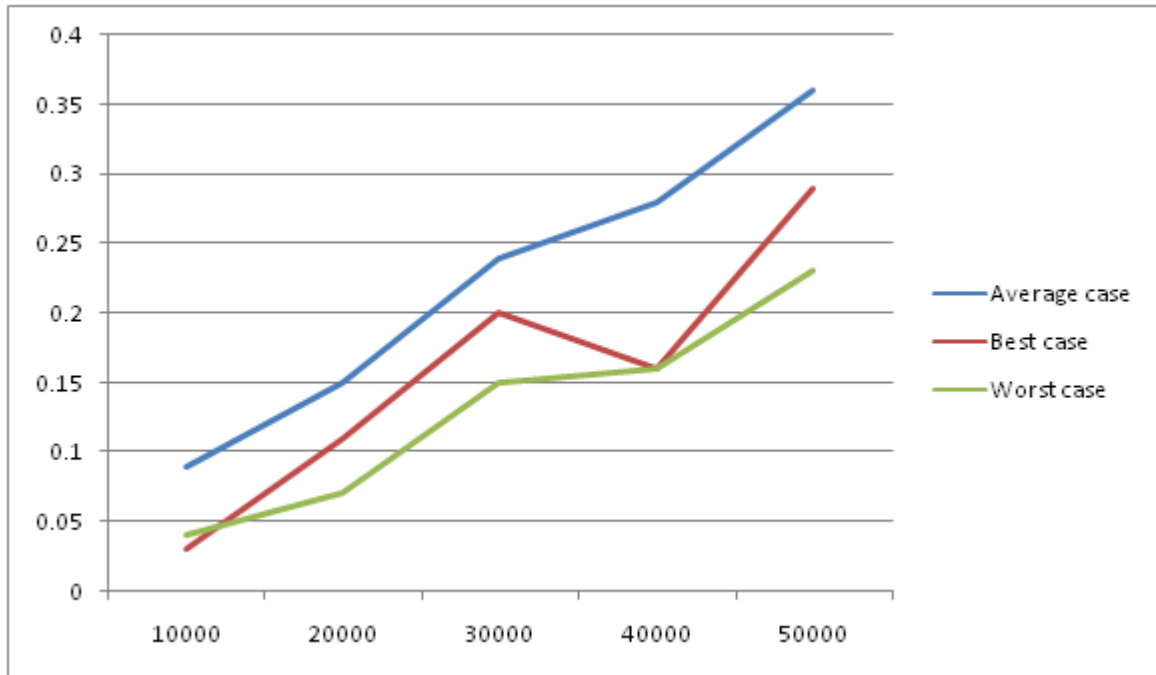


fig: Time complexity graph of merge sort.

0.4 Bubble Sort

Bubble Sort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

0.4.1 Algorithm of BUBBLE SORT

BUBBLE – SORT(A)

```
1  for  $i = 1$  to  $length[A]$ 
2    do for  $j = length[A]$  downto  $i + 1$ 
3      do if  $A[j] < A[j - 1]$ 
4        then exchange  $A[j] < - > A[j - 1]$ 
```

0.4.2 Time Complexity of BUBBLE SORT

In average and worst case, the running time of BUBBLE-SORT are same, $T(n) = O(n^2)$. But in best case, the running time is $O(n)$.

Here is the table of time complexity-

	A	B	C	D	E	F	G	H
1	number of elements				Time			
2			Average case		Best case		Worst case	
3	10000		12.34		5.44		12.84	
4	20000		25.77		21.23		50.97	
5	30000		103.58		49.95		114.41	
6	40000		204.22		94.2		197.68	
7	50000		184.41		153.89		257.57	
8								
9								
10								
11								
12								
13								

fig: Table of Time Complexity of Bubble Sort.

0.4.3 Time Complexity Graph of BUBBLE-SORT

The following graph represents time complexity of BUBBLE-SORT.

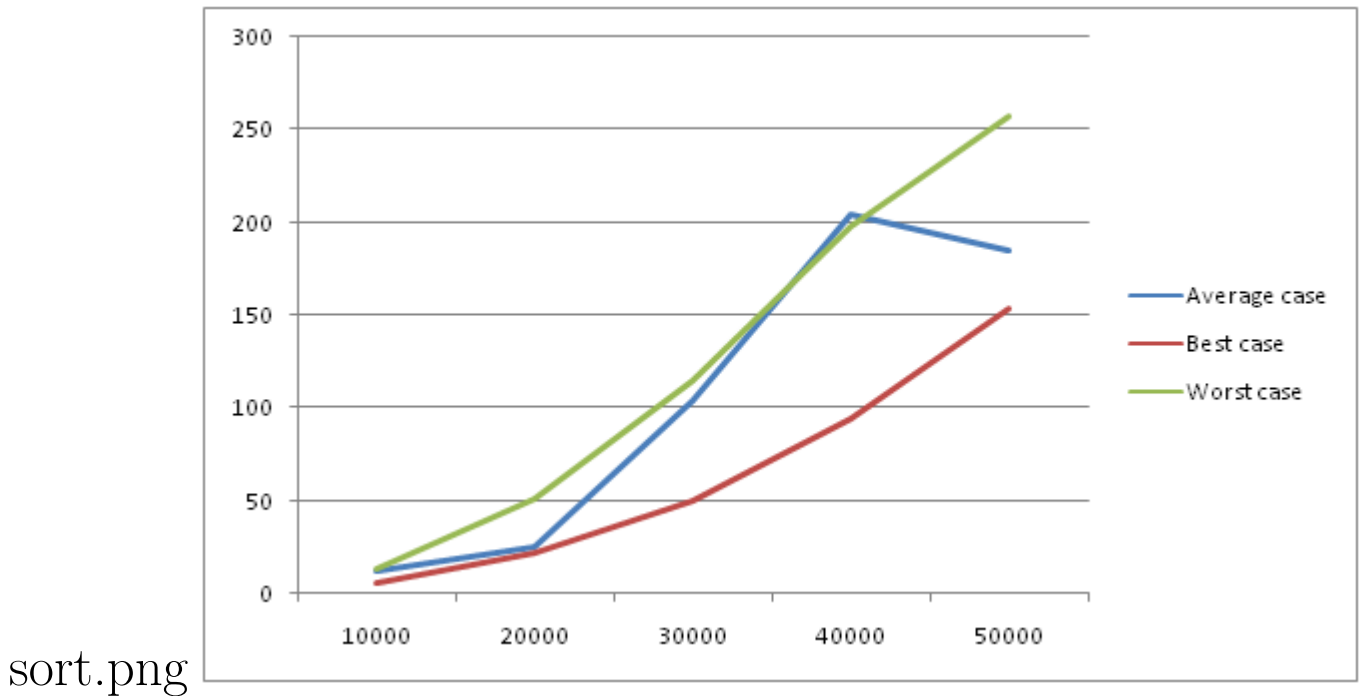


fig: Time complexity of BUBBLE-SORT.

0.5 Quick Sort

Quick Sort, like Merge Sort, applies the divide and conquer paradigm. Here is the three-step divide and conquer process for sorting a typical subarray $A[p..r]$.

Divide: Partition the array $A[p..r]$ into two (possible empty) subarrays $A[p..q - 1]$ and $A[q + 1..r]$ such that each element of $A[p..q - 1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q + 1..r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p..q - 1]$ and $A[q + 1..r]$ by recursive calls to quick sort.

Combine: Because the subarrays are already sorted,

no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

0.5.1 Algorithm of Quick Sort

The following procedure implements quicksort:

```
QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2     $q = \textit{PARTITION}(A, p, r)$ 
3    QUICKSORT( $A, p, q - 1$ )
4    QUICKSORT( $A, q + 1, r$ )
```

Partitioning the array

The key to the algorithm is the *PARTITIONING* procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION( $A, p, r$ )
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4    if  $A[j] \leq x$ 
5       $i = i + 1$ 
6      exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

0.5.2 Time Complexity of Quick Sort

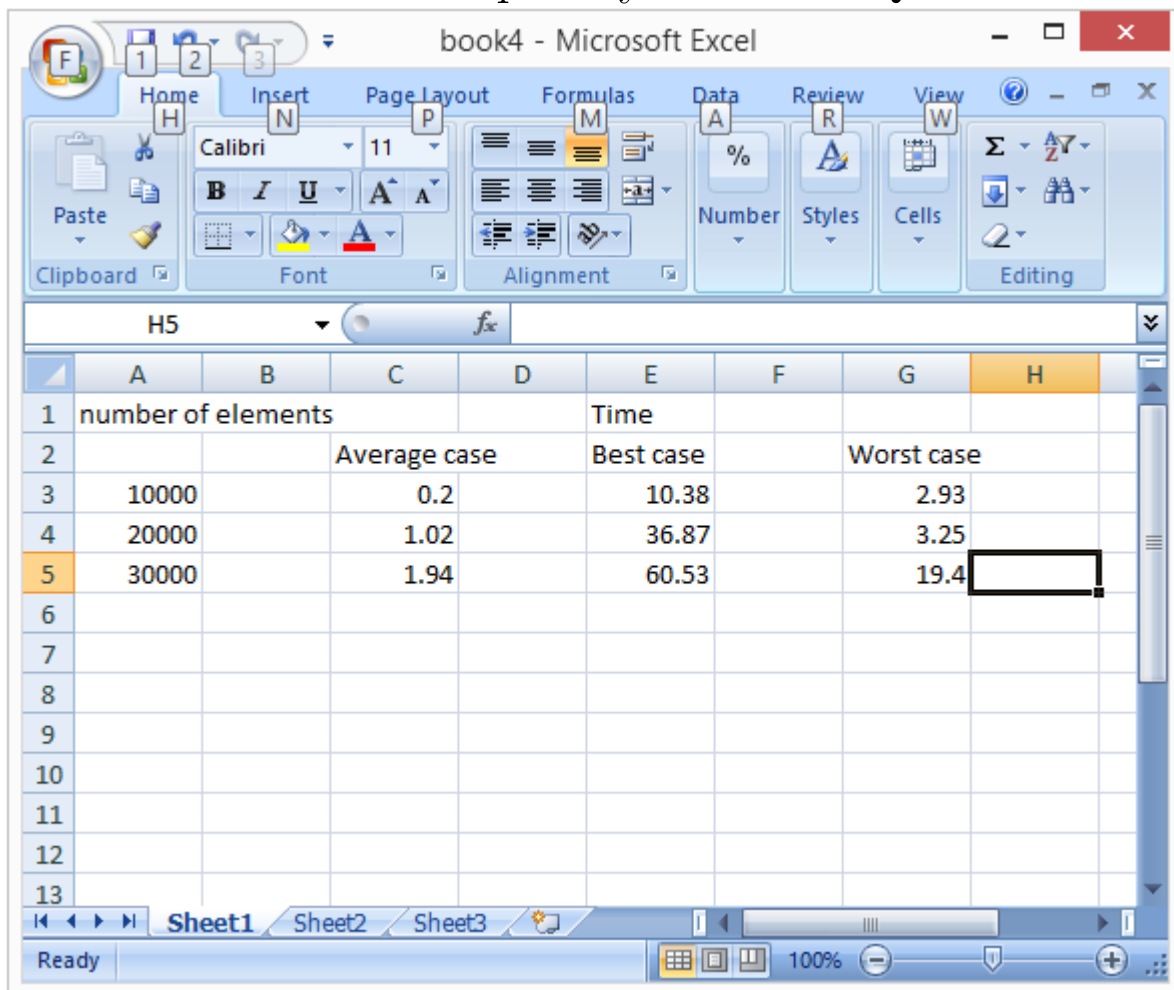
The running time of quick sort is similar at best and average case, but it is larger at worst case.

Average case: The running time is $O(n \lg n)$.

Best case: The running time is $O(n \lg n)$.

Worst case: The running time is $O(n^2)$.

Here is the Time Complexity Chart of Quick Sort:



	A	B	C	D	E	F	G	H
1	number of elements				Time			
2			Average case		Best case		Worst case	
3	10000		0.2		10.38		2.93	
4	20000		1.02		36.87		3.25	
5	30000		1.94		60.53		19.4	
6								
7								
8								
9								
10								
11								
12								
13								

fig: Time Complexity chart of Quick Sort.

0.5.3 Time Complexity Graph of Quick Sort

From chart, we draw a graph that represents the efficiency of Quick Sort algorithm.

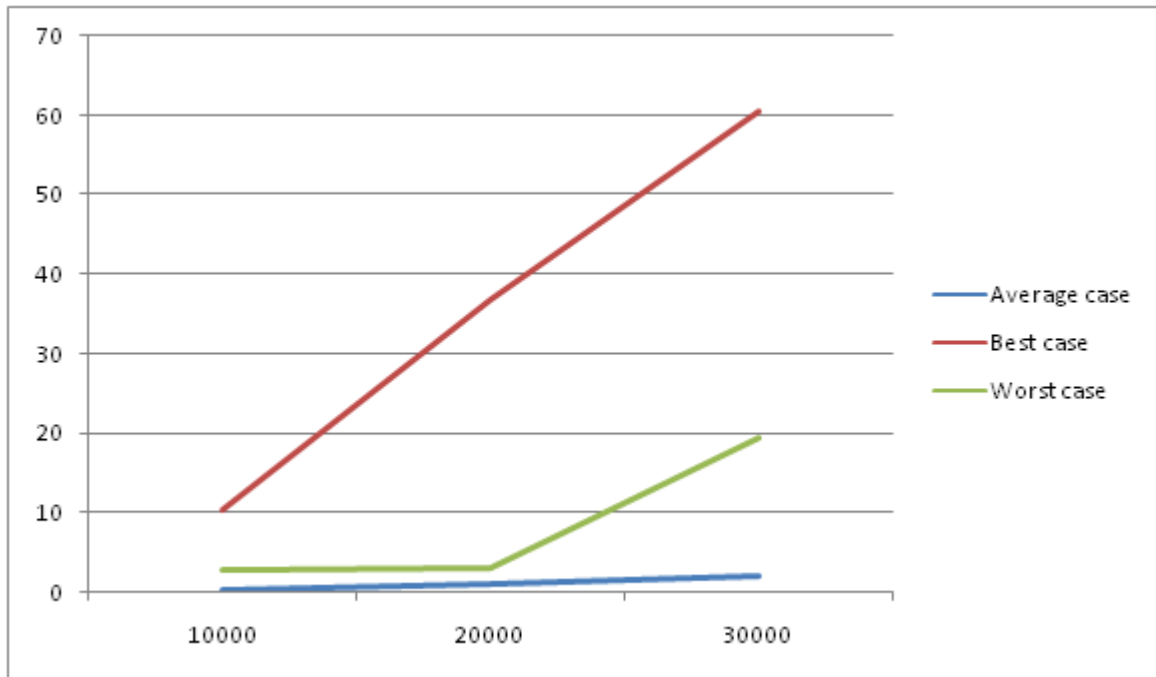


fig: Time Complexity graph of quick sort.

0.6 Heap Sort

Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a

heap, to manage information.

0.6.1 Algorithm of Heap Sort

MAX – HEAPIFY(A, i)

```
1   $l = LEFT(i)$ 
2   $r = RIGHT(i)$ 
3  if  $l \leq A.heap - size$  and  $A[l] > A[i]$ 
4     $largest = l$ 
5  else  $largest = i$ 
6  if  $r \leq A.heap - size$  and  $A[r] > A[largest]$ 
7     $largest = r$ 
8  if  $largest \neq i$ 
9    exchange  $A[i]$  with  $A[largest]$ 
10   MAX – HEAPIFY( $A, largest$ )
```

BUILD – MAX – HEAP(A)

```
1   $A.heap-size = A.length$ 
2  for  $i = [A.length/2]$  downto 1
3    MAX – HEAPIFY( $A, i$ )
```

HEAPSORT(A)

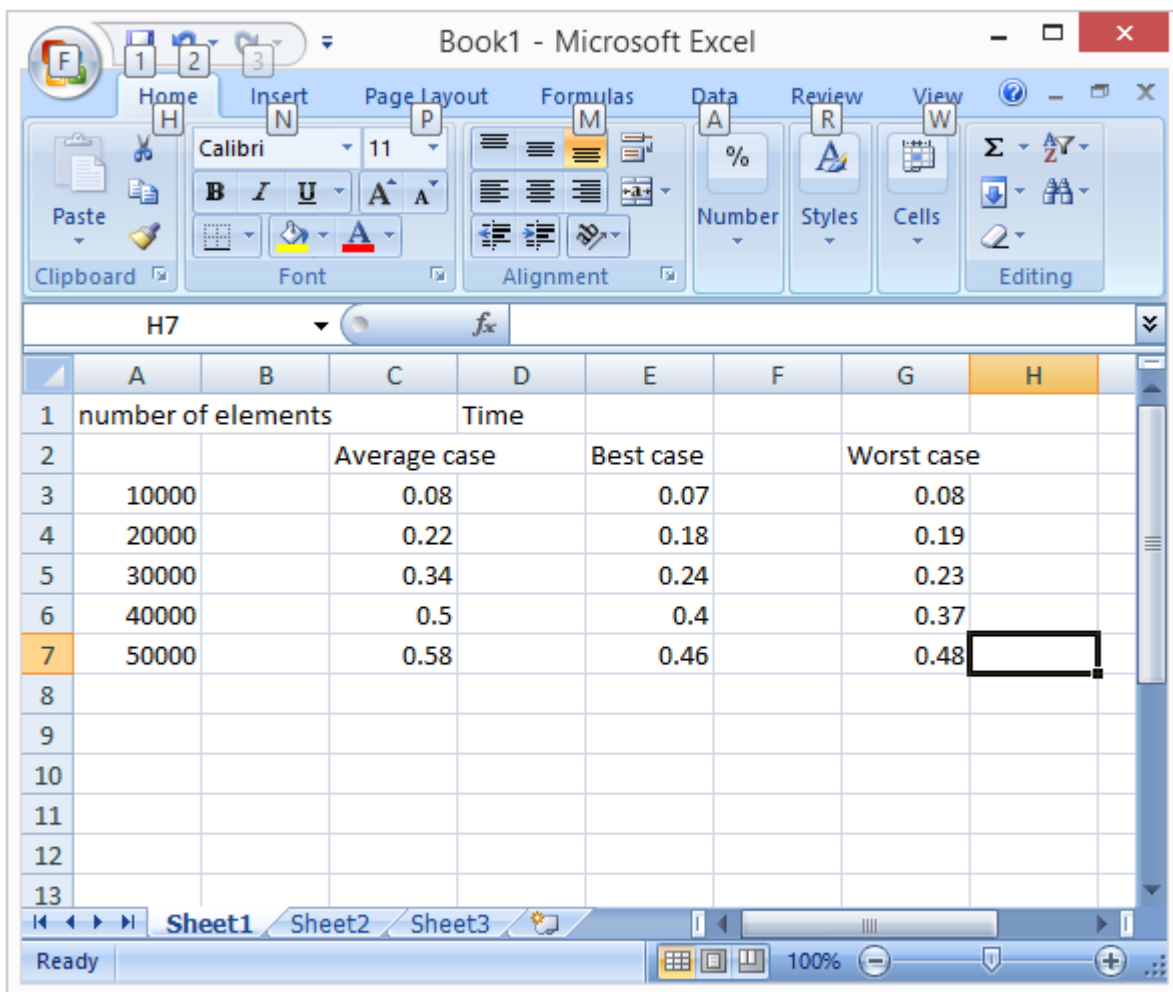
```
1  BUILD – MAX – HEAP( $A$ )
2  for  $i = A.length$  downto 2
3    exchange  $A[1]$  with  $A[i]$ 
4     $A.heap-size = A.heap-size - 1$ 
5    MAX – HEAPIFY( $A, 1$ )
```

0.6.2 Time Complexity of Heap Sort

The Time complexity of Heap sort is not so complex. The running time of Heao sort is $O(nlgn)$. This is same for the best, average and worst case.

Chart for Time Complexity

Here is the table that contains the data of running time and number of elements.



	A	B	C	D	E	F	G	H
1	number of elements			Time				
2			Average case		Best case		Worst case	
3	10000		0.08		0.07		0.08	
4	20000		0.22		0.18		0.19	
5	30000		0.34		0.24		0.23	
6	40000		0.5		0.4		0.37	
7	50000		0.58		0.46		0.48	
8								
9								
10								
11								
12								
13								

fig: Time Complexity chart.

0.6.3 Time Complexity Graph

Here is the graph that represents the time complexity graph in Heap-Sort.

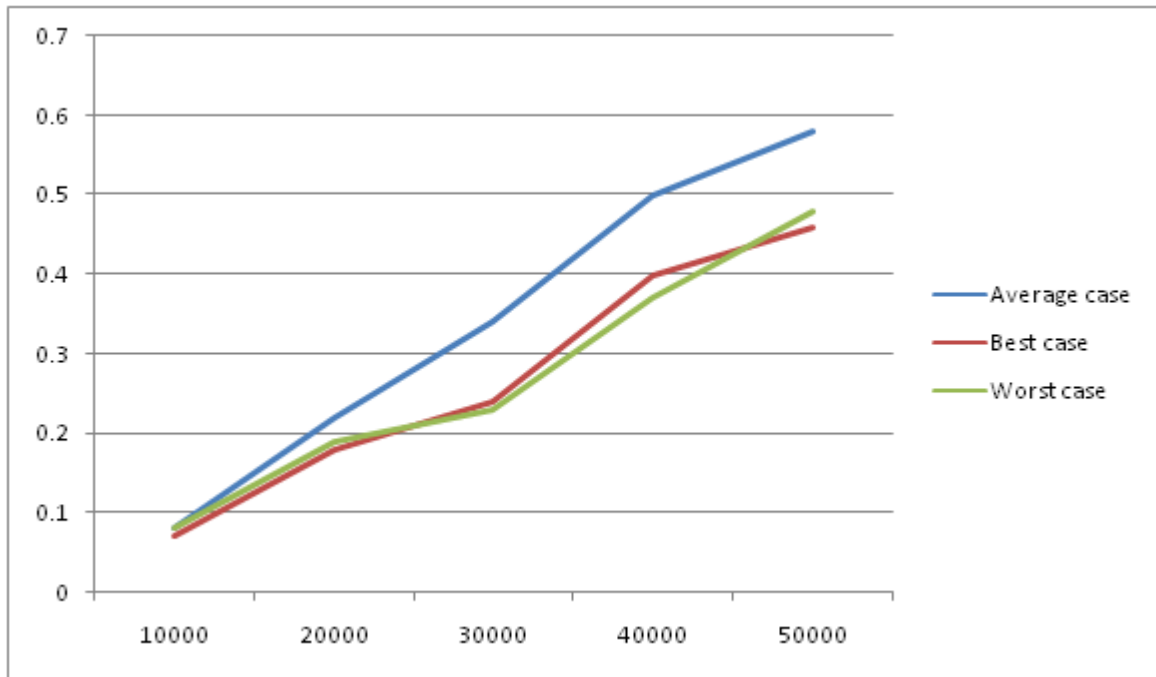


fig: Time Complexity graph of Heap-Sort.