A Comparison Based Analysis of Four Different Types of Sorting Algorithms in Data Structures with Their Performances
Nidhi Imran Simarjeet
*International Journal of Advanced Research in Computer Science and Software Engineering*
*Feb 2013 Vol. 3, Issue 2*

Aman Kumar

Department of Computer Science & Engineering, IIT Kanpur
E-mail: amanku@iitk.ac.in

3 November, 2014
Contents

1 Introduction

2 Sorting Algorithms
   - Quicksort
   - Heapsort
   - Insertion Sort
   - Mergesort

3 Experiment and Result

4 Conclusion
Introduction

Which Algorithm to use and when?
Which Algorithm to use and when?

- Quick Sort
Introduction

Which Algorithm to use and when?

- Quick Sort
- Heap Sort
Introduction

Which Algorithm to use and when?

- Quick Sort
- Heap Sort
- Insertion Sort
Introduction

Which Algorithm to use and when?

- Quick Sort
- Heap Sort
- Insertion Sort
- Merge Sort
Quicksort

This sorting algorithm is based on Divide-and-Conquer paradigm that is the problem of sorting a set is reduced to the problem of sorting two smaller sets.
Quicksort

This sorting algorithm is based on Divide-and-Conquer paradigm that is the problem of sorting a set is reduced to the problem of sorting two smaller sets.
Quicksort

This sorting algorithm is based on Divide-and-Conquer paradigm that is the problem of sorting a set is reduced to the problem of sorting two smaller sets.

![Quicksort Algorithm Diagram](image)
Time Complexity of Quick Sort

\[ T(N) = 2T(N/2) + cN \]

**Best Case Analysis:**
- \( O(n\log n) \)

**Average Case Analysis:**
- \( O(n\log n) \)

**Worst Case Analysis:**
- \( O(n^2) \)
Time Complexity of Quick Sort

- \( T(N) = 2T(N/2) + cN \)
Time Complexity of Quick Sort

- \( T(N) = 2T(N/2) + cN \)
- **Best Case Analysis:** \( O(n\log n) \)
Time Complexity of Quick Sort

- $T(N) = 2T(N/2) + cN$
- **Best Case Analysis:** $O(n\log n)$
- **Average Case Analysis:** $O(n\log n)$
Time Complexity of Quick Sort

- \( T(N) = 2T(N/2) + cN \)
- **Best Case Analysis:** \( O(n\log n) \)
- **Average Case Analysis:** \( O(n\log n) \)
- **Worst Case Analysis:** \( O(n^2) \)
Time Complexity of Quick Sort

- $T(N) = 2T(N/2) + cN$
- **Best Case Analysis:** $O(n\log n)$
- **Average Case Analysis:** $O(n\log n)$
- **Worst Case Analysis:** $O(n^2)$
Heapsort

The heap (binary) data structure is an array object that can be viewed as a complete binary tree as shown in figure.

Time Complexity:
O(nlogn)
Heapsort

The heap(binary) data structure is an array object that can be viewed as a complete binary tree as shown in figure:
Heapsort

The heap(binary) data structure is an array object that can be viewed as a complete binary tree as shown in figure:

![Binary Heap Diagram]

- **Time Complexity:** $O(n \log n)$
Insertion Sort

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm. It builds the sorted sequence one element at a time.

Best Case time Complexity: \(O(n)\)

Worst Case time Complexity: \(O(n^2)\)
Insertion Sort

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm. It builds the sorted sequence one element at a time.
Insertion Sort

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm. It builds the sorted sequence one element at a time.

![Insertion Sort Diagram](image)
Insertion Sort

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm. It builds the sorted sequence one element at a time.

- **Best Case time Complexity:** $O(n)$
Insertion Sort

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm. It builds the sorted sequence one element at a time.

- **Best Case time Complexity:** $O(n)$
- **Worst Case time Complexity:** $O(n^2)$
Mergesort

This algorithm is also based on Divide-and-Conquer approach. The general idea is to imagine them split into two sets and each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of \( n \) elements.

**Time Complexity:**

\[ O(n \log n) \]
Mergesort

This algorithm is also based on Divide-and-Conquer approach. The general idea is to imagine them split into two sets and each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements.
Mergesort

This algorithm is also based on Divide-and-Conquer approach. The general idea is to imagine them split into two sets and each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements.
Mergesort

This algorithm is also based on Divide-and-Conquer approach. The general idea is to imagine them split into two sets and each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements.

- **Time Complexity:** $O(n\log n)$
Experiment

In this experiment, Turbo C++ 3.0 compiler is used in which the data set contains random numbers. The initial range of the data set starts from 50 to 10000 elements with an increment of 100 elements and later the size of elements increased and reached to 30000 with an interval of 1000 elements.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Quick Sort</th>
<th>Heap Sort</th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Nil</td>
<td>Nil</td>
<td>1</td>
</tr>
<tr>
<td>1500</td>
<td>Nil</td>
<td>Nil</td>
<td>3</td>
</tr>
<tr>
<td>2000</td>
<td>Nil</td>
<td>Nil</td>
<td>5</td>
</tr>
<tr>
<td>2500</td>
<td>Nil</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>3000</td>
<td>Nil</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Table: Shows the number of clock ticks taken by the four algorithms for sorting different data sizes.
In this experiment Turbo C++ 3.0 compiler is used in which the data set contains random numbers. The initial range of data set starts from 50 to 10000 elements with increment of 100 elements and later the size of elements increased and reached to 30000 with the interval of 1000 elements.
Experiment

In this experiment Turbo C++ 3.0 compiler is used in which the data set contains random numbers. The initial range of data set starts from 50 to 10000 elements with increment of 100 elements and later the size of elements increased and reached to 30000 with the interval of 1000 elements.

<table>
<thead>
<tr>
<th>No. of clock ticks</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
<th>25000</th>
<th>300000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick sort</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Nil</td>
<td>Nil</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table : Shows the number of clock ticks taken by the four algorithms for sorting
## Experiment and Result

<table>
<thead>
<tr>
<th>No. of clock ticks</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
<th>25000</th>
<th>30000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick sort</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>0.164835</td>
<td>0.164835</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>0.054945</td>
<td>0.164835</td>
<td>0.274725</td>
<td>0.384615</td>
<td>0.549451</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Nil</td>
<td>Nil</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table:** Shows time taken (in seconds) by the four algorithms for sorting
## Experiment and Result

<table>
<thead>
<tr>
<th></th>
<th>Quick</th>
<th>Heap</th>
<th>Insertion</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time Complexity</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Average</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \cdot n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Worst</td>
<td>$O(n \cdot n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \cdot n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Space Complexity</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$N$</td>
</tr>
<tr>
<td><strong>Comparison based</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Inplace</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Internal</td>
<td>Internal</td>
<td>Internal</td>
<td>Can be both Internal and external</td>
</tr>
<tr>
<td><strong>Stable</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Table:** shows comparison of the three sorting techniques on various parameters
From the above analysis it can be said that in a list of random numbers from 10000 to 30000, insertion sort takes more time to sort as compare to heap, quick and merge sorting techniques. If we take worst case complexity of all the four sorting techniques then insertion sort and quick sort technique gives the result of the order of N square, but here if one needs to sort a list in this range then quick sorting technique will be more helpful than the other techniques.
References I

- Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, “Computer Algorithms”
Thanks!