COMPARATIVE STUDY
OF
SORTING ALGORITHMS

BY :

ABHILASH MISHRA

SUBMITTED TO:

Dr. RAKESH MOHANTY,

Associate Professor

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Assistant Professor Dr. Rakesh Mohanty for giving me the opportunity to carry out the mini-project, and for guiding me and motivating me through this wonderful journey . I would also like to thank my parents, who stood like a pillar behind me through the ups and downs I faced during this period, for providing me assistance whenever needed, regardless of the hours on the clock. I feel privileged to be blessed with such comforting hands around me, which are, but, available only to a privileged few in this world.

Abstract

Sorting is amongst the most common operations performed in computer applications. Sorting helps to organise the data so that search operations can be performed faster through search techniques (such as binary search, interpolation search ,Fibonacci search,etc) . There are many sorting algorithms, and their sorting techniques have different complexities. Also, each sort has different performance with different data size. Theoretical and practical performance of sorts tend to differ too. Under all these circumstances, the study of sorting algorithm becomes essential.

In this project, we investigate the performance of four sorting techniques, bubble sort, insertion sort, merge sort and a modified version of insertion sort and compare their performances.

The results of the project would contribute immensely to designing better performing sorting algorithms, which in turn, would make searching on the data faster.

Contents

Review on sorting algorithms

(Bubble sort, insertion sort ,merge sort ,modified insertion sort)

Bubble sort

Bubble sort is the simplest of all sorts. It works by repeatedly swapping the adjacent elements if they are placed in the wrong order. This continues till all elements are sorted. In the 1st swap, the largest element is put in place, followed by the 2nd element ,and so on.

Bubble sort is quite easy to implement, but the efficiency of the sort is low. It is not used for practical applications.

Pseudo code

```
func bubblesort ( var a as array )
 for i from 1 to N
 for j from 0 to N - 1
 if a[j] > a[j + 1]
 swap( a[j], a[j + 1] )
end func
```

Advantages of bubble sort:

1. It is a very simple algorithm to implement and works well when the data size is small.
2. This algorithm takes O(1)extra space.

Disadvantages of bubble sort:

This sort has a complexity $O(n^2)$ and is poor for large volumes of data.

Insertion sort

Insertion sort is a better performing sort than bubble sort. It works by inserting each element into its correct position in the array. It is done in the same way as card players arranging their cards. When they receive a card, they place it in its correct position.

Unlike its counterparts like bubble sort and selection sort, complexity of insertion sort depends on the ordering of the data. It works well for shorter and partially sorted data sets. The efficient inner loop takes the advantage of the partial ordering of data.

Psuedo code :

INSERTION-SORT(*A*)
1 **for** *j* ← 2 **to** *length*[*A*]
2 **do** *key* ← *A*[*j*]
3 ▹ Insert *A*[*j*] into the sorted sequence *A*[1 _ *j* - 1].
4 *i* ← *j* - 1
5 **while** *i* > 0 and *A*[*i*] > *key*
6 **do** *A*[*i* + 1] ← *A*[*i*]
7 *i* ← *i* - 1
8 *A*[*i* + 1] ← *key*

Advantages:

1. Though the complexity of insertion sort is $O(n^2)$,it takes O(n) time for a sorted array.
2. It is almost twice as fast as bubble sort. So, it is used in combination with merge sort and quick sort for greater efficiency.
3. It takes O(1) space for sorting.

Disadvantages:

1. This sort is not efficient for larger data sets.

Merge sort

Merge sort is a very efficient sort for sorting large sets of data. The sort uses divide and conquer paradigm for sorting the data. The input list is split into smaller lists recursively till the size of each sub list is of 1.It is then sorted by comparing two sub lists at a time , till the final sorted list is constructed.

Merge sort has a recursive as well as a non-recursive version .But the non-recursive version is inefficient and is thus not generally used.

Psuedo code:

```
MERGE-SORT(A, p, r)
1 if p < r
2 then q ← ⌊(p + r)/2⌋
3 MERGE-SORT(A, p, q)
4 MERGE-SORT(A, q + 1, r)
        5 MERGE(A, p, q, r)

MERGE(A, p, q, r)
1       n1 ← q - p + 1
2       n2 ← r - q
3       create arrays L[1 _ n1 + 1] and R[1 _ n2 + 1]
4       for i ← 1 to n1
5       do L[i] ← A[p + i - 1]
6       for j ← 1 to n2
7       do R[j] ← A[q + j]
8       L[n1 + 1] ← ∞
9       R[n2 + 1] ← ∞
10      i ← 1
11      j ← 1
12      for k ← p to r
```

13 **do if** $L[i] \leq R[j]$

14 **then** $A[k] \leftarrow L[i]$

15 $i \leftarrow i + 1$

16 **else** $A[k] \leftarrow R[j]$

17 $j \leftarrow j + 1$

Advantages:

1. The complexity of merge sort is O(n log n).So,the performance of the algorithm is very high.
2. The worst case running time of this algorithm is O(n),which is a rare case.

Disadvantages:

1. Merge sort requires a large amount of memory to perform sorting.So,it is not advisable for devices with lower memory capabilities.
2. It is not recommended for smaller arrays, as it requires O(n)auxiliary space

Modified insertion sort

This is an attempt to improve the performance of insertion sort.In normal insertion sort,sorting is done from the front end of the array , arranging the elements in ascending order as it moves forward.

The performance of the sort may be improved by performing the sort from both ends of the array by sorting the elements in ascending order in the 1st half of the array,sorting the elements in descending order in the 2nd half of the array in parallel and merging them.

Psuedo code:

INSERTION-SORT(*A,C*)

1 **for** $j \leftarrow 2$ **to** *length*[*A*]/2

2 **do** $key \leftarrow A[j]$

3 ▹ Insert $A[j]$ into the sorted sequence $A[1 : j - 1]$.

4 $i \leftarrow j - 1$

5 **while** $i > 0$ and $A[i] > key$

6 **do** $A[i + 1] \leftarrow A[i]$

7 $i \leftarrow i - 1$

8 $A[i + 1] \leftarrow key$

for $j \leftarrow length[A]/2 + 2$ **to** $length[A]$

2 **do** $key \leftarrow A[j]$

3 ▹ Insert $A[j]$ into the sorted sequence $A[n/2 : j - 1]$.

4 $i \leftarrow j - 1$

5 **while** $i > n/2 + 1$ and $A[i] > key$

6 **do** $A[i + 1] \leftarrow A[i]$

7 $i \leftarrow i - 1$

8 $A[i + 1] \leftarrow key$

9 i=1, j=n/2+1

10 while(i <= n/2 && j<=n)

11 **do** if(A[i]<A[j])

12 **do** C[k++]=A[i++].

13 else if(A[i]>A[j])

14 **do** C[k++]=A[j++].

15 if(i<n/2)

16 **do** C[k++]=A[i++]

17 else if(j < n)

18 do C[k++]=A[j++]

19 print C.

Example :

For example: consider the data set with 8 elements and with the worst permutation of input.

INPUT, A =[8 7 6 5 4 3 2 1]

For performing insertion sort, the number of comparisons required is

comparisons =(n -1)n/2

In the modified insertion sort, starting from 7 in the left hand side and from 2

in the right hand side, we get the following steps:

A = [7 8 6 5 4 3 1 2]

comparisons = (1 + 1)

In the next step, we take 6 from left and 3 from right.

A=[6 7 8 5 4 1 2 3]

comparisons = (1 + 1) + (2 + 2)

Next, we take 5 from left and 4 from right.

A=[5 6 7 8 1 2 3 4]

comparisons = (1 + 1) + (2 + 2) + (3 + 3)

Now, we have two sets of sorted data from 1 to 4 and from 5 to 8.

Now merging the two sorted ordered data into another data set, we get

A1=[5 6 7 8]

and

A2=[1 2 3 4]

Merging the two sorted data lists,

Final sorted list =

OUTPUT=[1 2 3 4 5 6 7 8]

comparisons = (1 + 1) + (2 + 2) + (3 + 3) + 4=16

Therefore, total number of comparisons is 16.

But in insertion sort, number of comparisons = 28. (By applying n(n-1)/2 formula)

The difference in number of comparisons comes to 12

.

Using induction,it can be proved that :

For a permutation of worst case input with n elements,

The number of comparisons is equal to

comparisons =$(n^2)/4$

COMPARISONS ON SORTING ALGORITHMS

In the previous section, we studied various sorting and searching algorithms. In this section, we perform experiments on the algorithms and compare their behaviour based on the outcomes. The inputs used for comparisons were generated randomly by rand() function from <stdlib.h> library file. The inputs are same for all the sorts.

Test-1

Number of comparisons of different sorts

SI NO	Number of elements	Bubble sort	Insertion sort	Merge sort	Modified insertion sort
1.	30	435	240	108	158

2.	100	4950	2706	545	1472
3.	2000	1999000	947056	19392	489686
4.	300000	44999850000	22260723921	5070049	11155738192

TABLE-1

From the table, we observe that as the number of elements increase,the number of comparisons increase considerably for bubble sort, insertion sort and modified insertion sort. The number of comparisons in merge sort is much smaller compared to its counterparts. Also, the modified version of insertion sort performs better than the orthodox insertion sort.

Test-2

Time taken by different sorts

SI NO	Data set	Bubble sort	Insertion sort	Merge sort	Modified Insertion sort
1.	30	0.125s	0.14s	0.094s	0.187s
2.	100	0.125s	0.14s	0.094	0.236
3.	2000	0.265s	0.125s	0.156s	0.156s
4.	300000	698.785s	335.619s	0.562s	231.052s

TABLE-2

From the table, we can observe how $O(n^2)$ algorithms slow down drastically when the number of elements are increased. Bubble sort turns out to be the slowest amongst all sorts. Modified insertion sort has a slight better performance than the orthodox insertion sort. Merge sort was the fastest amongst all.

Test-3

In this test,we take various permutations of different elements(3 distinct elements, 4 distinct elements, 5 distinct elements ,6 distinct elements ,7 distinct elements) and compare the performance of the sorts.

(The data set is present in the excel file “input permutations.xls” attached in the mail or in the folder containing this document)

From the spreadsheet ,we can conclude that ::

1. Sorting depends on the ordering of the input data. When we compare merge sort and insertion sort, we find that for data sets where the elements are almost sorted, merge sort performs poorly. In cases where the elements are reversely sorted, merge sort performs the best.

2. In case of modified insertion sort, like merge sort, performance of the sort is best when the data set was reverse sorted and poor when the data elements were partially sorted.
3. Insertion sort performs the best when the data set is sorted , and performs worst when the data set is reversely sorted.

CONCLUSION

The comparison study shows us that the sorting performance is strongly correlated with the permutation of the data set. This can be a factor while selecting a sorting algorithm to sort a set to data.

We found out that merge sort, having a complexity of O(n log n) surpasses other $O(n^2)$ algorithms in performance. But it crumbles when the data set is small.

Insertion sort and bubble sort are good at sorting small data sets, but are poor at sorting larger data sets.

Modified insertion sort performs the best when the data set is in reverse sorted order, and worst when it is in sorted form.

The study performed can be used to design better sorting algorithms, improving the performance of the present algorithms.

==========

Bibliography

1. Wikipedia.com
2. Data structure book: Tanenbaum
3. Algorithm design(Corman)
4. Geeksforgeeks.com

==========