ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 5, Issue 5, May 2016

A COMPARATIVE STUDY AND ANALYSIS OF SEARCHING AND SORTING ALGORITHMS

W.SARADA, DR.P.V.KUMAR

***Abstract*— Sorting is one of the most fundamental problems in computer science, as it is used in most software applications. Data-driven algorithms especially use sorting to gain an efficient access to data. Many sorting algorithms with distinct properties for different architectures have been developed. Searching for items and sorting through items are tasks that we do every day. Searching and sorting are also common tasks in computer programs. We search for all occurrences of a word in a file in order to replace it with another word. We sort the items on a list into alphabetical or numerical order. Because searching and sorting are common computer tasks, we have well-known algorithms, for doing searching and sorting. We'll look at two searching algorithms and four sorting algorithms here in detail, and go through examples of each algorithm and determine the performance of each algorithm, in terms of how "quickly" each algorithm completes its task.**

***Index Terms*— Sorting, Algorithms, complexity, time, space, performance**

.

I. INTRODUCTION

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types; Internal Sorts:- This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts they can only process relatively small list due to memory constraints. Algorithms can b implemented by different program ms in different programming languages. Different algorithms for solving the same problem are more efficient if they require fewer basic operationsmay solvedifferent instances of the different instances of the same problem more or less efficiently. Sorting is equal to Creating a sequence of items in a Set S, such that a binary relation (ex., ≤) holds for pairs of consecutive elements.An algorithm for sorting a set of elem ents different algorithms may be better (more efficient) for sorting under different circumstances. Complexities of algorithms are judged by the Time (how fast) and space (how much memory).

II. PERFORMANCE MEASUREMENT

We can compare the speed of two programs without using time by counting the number of instructions or operations in the two programs. Typically, the faster program has fewer operations. Often, the number of operations is directly proportional to the number of data items that the program operates on.

When comparing the performance of two search algorithms or two sorting algorithms, we concentrate on two types of operations: data movements, or swaps, and comparisons. Data movements occur when we replace one item in a list with another item in the list. Data comparisons occur when we compare one item in a list with either another item in the list, or an item outside the list. Additionally, it's often sufficient to determine the approximate number of swaps and comparisons, rather than the exact number of swaps and comparisons, for a search or sorting algorithm.

III. SEARCH ALGORITHMS

There are two types of search algorithms: algorithms that don't make any assumptions about the order of the list, and algorithms that assume the list is already in order. We'll look at the former first, derive the number of comparisons required for this algorithm, and then look at an example of the latter. In the discussion that follows, we use the term search term to indicate the item for which we are searching. We assume the list to search is an array of integers, although these algorithms will work just as well on any other primitive data type (doubles, characters, etc.). We refer to the array elements as items and the array as a list

A. Linear search

The simplest search algorithm is linear search. In linear search, we look at each item in the list in turn, quitting once we find an item that matches the search term or once we've reached the end of the list. Our "return value" is the index at which the search term was found, or some indicator that the search term was not found in the list.

a. *Performance of linear search*

When comparing search algorithms, we have to see number of comparisons required, since we don't swap any values while searching. Often, when comparing performance, we look at three cases:

• Best case: To find an item, may be a few number of comparisons are necessary

• Worst case: To find an item, may be many number of comparisons are necessary

• Average case: On average, to find an item in the list, number of comparisons required.

For linear search, our cases look like this:

• Best case: when searching, the number of comparisons in this case is 1 .

• Worst case: when searching, it takes N comparisons in the worst case, and is equal to the size of the array.

• Average case: On average, the search term will be somewhere in the middle of the array.

B. *Binary search*

We use the binary search algorithm to find data stored in an array. This method is very effective, as each iteration reduced the number of items to search by one-half. However, since data was stored in an array, insertions and deletions were not efficient. Binary search trees store data in nodes that are linked in a tree-like fashion. For randomly inserted data, search time is O(lg n). Worst-case behavior occurs when ordered data is inserted. In this case the search time is O(n). Binary search exploits the ordering of a list. The idea behind binary search is that each time we make a comparison, we eliminate half of the list, until we either find the search term or determine that the term is not in the list. We do this by looking at the middle item in the list, and determining if our search term is higher or lower than the middle item. If it's lower, we eliminate the upper half of the list and repeat our search starting at the point halfway between the first item and the middle item. If it's higher, we eliminate the lower half of the list and repeat our search starting at the point halfway between the middle item and the last item.

a. *Performance of binary search*

The best case for binary search still occurs when we find the search term on the first try. In this case, the search term would be in the middle of the list. As with linear search, the best case for binary search is O(1), since it takes exactly one comparison to find the search term in the list.
The worst case for binary search occurs when the search term is not in the list, or when the search term is one item away from the middle of the list, or when the search term is the first or last item in the list.
The average case occurs when the search term is anywhere else in the list. The number of comparisons is roughly the same as for the worst case, so it also is O(logN). In general, anytime an algorithm involves dividing a list in half, the number of operations is O(log N).

III. SORTING ALGORITHMS

To determine the performance of Selection sort, Bubble sort, Quick sort, Merge sort, each algorithm in terms of the number of data comparisons and the number of times we swap list items.

A. Selection sort

The idea behind selection sort is that we put a list in order by placing each item in turn. In other words, we put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

Selection sort

```
void selectionSort(int arr[]){
int i, j, temp, pos_greatest;
 for( i = arr.length-1; i > 0; i--){
pos_greatest = 0; for(j = 0; j <= i; j++){
 if( arr[j] > arr[pos_greatest]) pos_greatest = j;
}//end inner for loop
 temp = arr[i]; arr[i] = arr[pos_greatest];
arr[pos_greatest] = temp;
}//end outer for loop
}//end selection sort swap the largest element to the end of
range compare the current element to the largest seen so far;
if it is larger, remember its index
```

a.*Complexity of selection sort*

• Same number of iterations
• Same number of comparisons in the worst case
• fewer swaps (one for each outer loop = n-1)
• also O(n2)

b.*Selection sort on linked lists*

• Implementation similar to bubble sort; also O(n2)
• Instead of pos_greatest, have a variable Node largest which keeps the reference of the node with the largest element we have seen so far
• Swap elements once in every iteration through the unsorted part of the list:

```
E element v = current.element;
 current.element =largest.element;
 largest.element= v;
```

c. *Performance of selection sort*

The best case for selection sort occurs when the list is already sorted, in the best case, selection sort requires O(N2) comparisons, worst case when the first item in the list is the

largest, and the rest of the list is in order, average case requires the same number of comparisons, O(N2), and roughly N/2 swaps. Thus, the number of swaps in the average case is O(N).

B. *Bubble sort*

Bubble sort performs more swaps in each pass and will finish sorting the list sooner than selection sort will. Like selection sort, bubble sort works by comparing two items in the list at a time. Unlike selection sort, bubble sort will always compare two consecutive items in the list, and swap them if they are out of order.

```
Bubble sort void bubbleSort(int arr[])
{
int i; int j; int temp;
for(i = arr.length-1; i > 0; i--)
{
 for(j = 0; j < i; j++)
{
if(arr[j] > arr[j+1])
{
temp = arr[j]; arr[j] = arr[j+1];
arr[j+1] = temp;
}//
}// end inner loop
 }//end outer loop
}// end bubble sort swap adjacent elements, if in the wrong
order
```

a. *Complexity of bubble sort*

• For an array of size n, in the worst case: 1st passage through the inner loop: n-1 comparisons and n-1 swaps

• (n-1)st passage through the inner loop: one comparison and one swap.

• All together: c ((n-1) + (n-2) + ... + 1), where c is the time required to do one comparison, one swap, check the inner loop condition and increment j.

• We also spend constant time k declaring i,j,temp and initialising i. Outer loop is executed n-1 times, suppose the cost of checking the loop condition and decrementing i is c1.

b. *Bubble sort of lists*

• Bubble sort is just as efficient (or rather inefficient) on linked lists.

• We can easily bubble sort even a singly linked list.

• Assume we have a class Node with fields: element of type E and next of type Node.

• (Strictly speaking getter/setter methods would be better, but this is just for the sake of brevity…)

• The List class just has head field.

Bubble sort of a linked list

```
Node border = null; // first node in the sorted part
while (border != head) { Node current = head; // start from
the first node
 while (current.next != border) {
 if (current.element > current.next.element) {
E element v = current.element; current.element =
current.next.element; current.next.element = v;
} current = current.next;
} border = current; // the sorted part increases by one
}
```

c.*Complexity of bubble sort on lists*

• Same complexity as for arrays O(n2):

• First time we iterate until we see a null (swapping elements)

• Second time we iterate until we see the last node;

• … each time the border is one link closer to the head of the list

• until border == head.

d. *Time Complexity of Bubble Sort :*

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is: n (n – 1) / 2 ≈ n 2 – n
Best case : O (n2)
Average case : O (n2)
Worst case : O (n2)

e. *Performance of bubble sort*

The best case for bubble sort occurs when the list is already sorted and makes one pass through the list, performing no swaps and N-1 comparisons, worst case occurs when the list is in reverse order. The number of swaps in the worst case is greater than in selection sort: each comparison results in a swap, so there are O(N2) swaps in bubble sort! The average case looks like the worst case: O(N2) comparisons and O(N2) swaps. The tradeoff is that we may be able to do half as much iteration, on average, as we do in selection sort. In terms of the number of comparisons and the number of swaps

both the Selection sort and bubble sort had many similarities, the same thing for quick sort and merge sort.

C. Quick sort

Quick sort is an example of a divide and conquer algorithm which sorts a list effectively by dividing the list into smaller and smaller lists, and sorting the smaller lists in turn also known as partition exchange sort invented by CAR Hoare. It is based on partition. The basic concept of quick sort process is pick one element from an array and rearranges the remaining elements around it. This element divides the main list into two sub lists. This chosen element is called pivot. Once pivot is chosen, then it shifts all the elements less than pivot to left of value pivot and all the elements greater than pivot are shifted to the right side. This procedure of choosing pivot and partition the list is applied recursively until sub-lists consisting of only one element.

a. *Time Complexity of Quick sort:*

Best case : O (n log n)
Average case : O (n log n)
Worst case : O (n2)

b. *Advantages of quick sort:*

1. This is faster sorting method among all.
2. Its efficiency is also relatively good.
3. It requires relatively small amount of memory.

c. *Disadvantages of quick sort:*

1. It is complex method of sorting so, it is little hard to implement than other sorting methods.

d. *Performance of quicksort*

The best case of quick sort occurs, obviously, when the list is already sorted. For this algorithm, the best case resembles the average case in terms of performance. The average case occurs when the pivot splits the list in half or nearly in half on each pass. The worst case for quick sort occurs when the pivot is always the largest or smallest item on each pass through the list. In this instance, we do not split the list in half or nearly in half, so we do N comparisons over roughly N passes, which means that in the worst case quick sort is closer to O(N2) in performance. For the same reason, the number of swaps in the worst case can be as high as O(N2).

D. Merge sort

Merge sort requires very few comparisons and swaps; it instead relies on a "divide and conquer" strategy that's slightly different from the one that quick sort uses. The algorithm repeats until all of these "sublists" have exactly one element and each sublist is sorted. In the next phase of the algorithm, the sublists are gradually merged back together as sorted.

a. *Performance of merge sort*

The number of comparisons per pass is (first+last)/2 in the best case, where the list is sorted, for the worst and average cases, the number of comparisons and number of swaps for merge sort is O(NlogN). Merge sort is as fast as quick sort but in practice, however, merge sort performs a bit more slowly, because of all the data copying the algorithm requires.

Algorithm	Avg time	Best Time	Worst Time	space	Stability
Bubble	n2	n2	n2	constant	Stable
Selection	n2	n2	n2	constant	stable
Quick	nlogn	nlogn	n2	constant	stable
Merge	nlogn	nlogn	nlogn	depends	Stable

IV. Conclusion

Each sorting algorithm has its own advantages and disadvantages. Selection sort and bubble sort are the two O(N2) sorts and quick sort and merge sort are the two O(NlogN) sorts.

• The advantage of Selection sort is that the number of swaps is O(N), and the disadvantage is that it does not stop early if the list is sorted and it looks at every element in the list in turn. **It** is suitable in cases where you need to select max element in the list. With time complexity same as Bubble Sort o(n^2), its performance however is better than Bubble Sort.

• The advantage of Bubble sort is that it stops once it determines that the list is sorted and its disadvantage is that it is O(N2) in both swaps and comparisons and is actually the least efficient sorting method, but **it is** primarily used in case elements are integer format with time complexity of o(n^2)

• The advantage of Quick sort is that it is the fastest sort and is O(NlogN) in both the number of comparisons and the number of swaps and the disadvantage is that the algorithm is a bit tricky to understand.

• The advantage of Merge sort is, it is as fast as quick sort and is O(NlogN) in both swaps and comparisons and the disadvantage is that it requires more copying of data from temporary lists back to the "full" list, which slows down the algorithm a bit. This technique is used for large sets of data that do not fit completely in fast memory.

REFERENCES

[1] Cederman D, Tsigas P. GPU-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics Jan 2010; 14:4:1.4–4:1.24, doi:10.1145/1498698.1564500.

[2] Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, IEEE Computer Society: Washington, DC, USA, 2009; 1–10, doi:10.1109/IPDPS.2009.5161005.

[3] Knuth, Donald E. [1998]. The Art of Computer Programming, Volume 3, Sorting and Searching. Addison-Wesley, Reading, Massachusetts.

[4] Batcher KE. Sorting networks and their applications. Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), ACM: New York, NY, USA, 1968; 307–314, doi:10.1145/1468075

[5] Singleton R C. Algorithm 347: An efficient algorithm for sorting with minimal storage [m1]. Commun. ACM Mar 1969; 12(3):185–186, doi:10.1145/362875.362901

W.Sarada is a research scholar at Rayalaseema University, Kurnool, and Andhra Pradesh, India. She is working as an Assistant Professor in the Department of Computer Science at RBVRR Women's College .Her areas of interests include Data Mining, Software Engineering, Digital Image Processing, Computer Net Works.

Dr.P.V.Kumar, Professor, College of Computer Science and Engineering, Osmania University, Hyderabad .He is a research supervisor for M.tech,, M.Phil and PhD students.
.