Why Sorting is So Important in Data Structures

Er. Rahul Kaushal
Assistant Professor
Department of Computer Science & Engineering
Panipat Institute of Engineering & Technology

Abstract— One of the basic areas of the computer science is Data Structure. Sorting is an important issue in Data Structure which creates the sequence of the list of items. Although numbers of sorting algorithms are available, it is all the more necessary to select the best sorting algorithm. Therefore, sorting problem has attracted a great deal of research as sorting technique is very often used in a large variety of important applications so as to arrange the data in ascending or descending order. This paper presents that sorting is an important area of study in computer science. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed.

Key words: Algorithm, Bubble Sort, Selection Sort, Quick Sort and Insertion Sort

I. INTRODUCTION

A. Sorting:

Sorting is important in programming for the same reason it is important in everyday life. It is easier and faster to locate items in a sorted list than unsorted. Sorting algorithms can be used in a program to sort an array for later searching or writing out to an ordered file or report.

In computer science, arranging in an ordered sequence is called "sorting". Sorting is a common operation in many applications, and efficient algorithms to perform it have been developed. The most common uses of sorted sequences are: making lookup or search efficient; making merging of sequences efficient.

In short, algorithms are computer programming. They are the same thing. ... Most of the time computer programmers create their own algorithms for the particular problem they are trying to solve, or sometimes they borrow a successful solution from someone who has solved it before.

1) Use of algorithms in programming:

A programming algorithm is a computer procedure that is a lot like a recipe (called a procedure) and tells your computer precisely what steps to take to solve a problem or reach a goal. The ingredients are called inputs, while the results are called the outputs.

2) Searching in computer science:

In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure. ... Searching also encompasses algorithms that query the data structure, such as the SQL SELECT command. Search algorithms can be classified based on their mechanism of searching.

B. Bubble sort:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

C. Performance

Bubble sort has worst-case and average complexity both $O(n^2)$, where nis the number of items being sorted. There exist many sorting algorithms, such as merge sort with substantially better worst-case or average complexity of O(n log n). Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only O(n). By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.

D. Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

1) First Pass

(5 1 4 2 8) (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 5 4 2 8) (1 4 5 2 8), Swap since 5 > 4

(1 4 5 2 8) (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

2) Second Pass

(1 4 2 5 8) (1 4 2 5 8)

(1 4 2 5 8) (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

3) Third Pass

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

E. Optimizing bubble sort:

The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n − 1 items when running for the n-th time:

1) procedure bubbleSort(A : list of sortable items)

```
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        swapped = true
      end if
    end for
    n = n - 1
  until not swapped
end procedure
```

More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over a lot of the elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable:

To accomplish this in pseudocode we write the following:

2) procedure bubbleSort(A : list of sortable items)

```
  n = length(A)
  repeat
    newn = 0
    for i = 1 to n-1 inclusive do
      if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        newn = i
      end if
    end for
    n = newn
  until n = 0
end procedure
```

F. Radix sort:

In computer science, radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers. Radix sort dates back as far as 1887 to the work of Herman Hollerithon tabulating machines.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. MSD radix sorts work the other way around.

LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j". If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

Original, unsorted list:
170, 45, 75, 90, 802, 2, 24, 66
Sorting by least significant digit (1s place) gives:
170, 90, 802, 2, 24, 45, 75, 66
Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.
Sorting by next digit (10s place) gives:
802, 2, 24, 45, 66, 170, 75, 90
Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.
Sorting by most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The

number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:
170, 045, 075, 090, 002, 024, 802, 066
The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:
2 (bucket size for digits of 0: 170, 090)
2 (bucket size for digits of 2: 002, 802)
1 (bucket size for digits of 4: 024)
2 (bucket size for digits of 5: 045, 075)
1 (bucket size for digits of 6: 066)
A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:
2 (bucket size for digits of 0: 002, 802)
1 (bucket size for digits of 2: 024)
1 (bucket size for digits of 4: 045)
1 (bucket size for digits of 6: 066)
2 (bucket size for digits of 7: 170, 075)
1 (bucket size for digits of 9: 090)
A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:
6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)
1 (bucket size for digits of 1: 170)
1 (bucket size for digits of 8: 802)

At least one LSD radix sort implementation now counts the number of times that each digit occurs in each column for all columns in a single counting pass. (See the external links section.) Other LSD radix sort implementations allocate space for buckets dynamically as the space is needed.

G. Insertion sort:

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:Array prior to the insertion of x becomesArray after the insertion of xwith each element greater than x copied to the right as it is compared against x.The most common variant of insertion sort, which operates on arrays, can be described as follows:Suppose there exists a function called Insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.To perform an insertion sort, begin at the left-most element of the array and invoke Insert to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.Pseudocode of the complete algorithm follows, where the arrays are zero-based

```
i ← 1
while i < length(A)
   j ← i
   while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j ← j - 1
   end while
   i ← i + 1
end while
```

H. Quick sort:

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
The steps are:

1) Pick an element, called a pivot, from the array.
2) Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3) Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance

```
algorithm quicksort(A, lo, hi) is
 if lo < hi then
 p := partition(A, lo, hi)
  quicksort(A, lo, p)
   quicksort(A, p + 1, hi)
algorithm partition(A, lo, hi) is
  pivot := A[lo]
  i := lo – 1
 j := hi + 1
 loop forever
    do
     i := i + 1
  while A[i] < pivot
  do
  j := j – 1
   while A[j] > pivot
  if i >= j then
    return j
     swap A[i] with A[j]
```

The entire array is sorted by quicksort(A, 0, length(A)-1).

II. CONCLUSION & FUTURE SCOPE

In this research paper we have studied about different sorting algorithms along with their examples. Every sorting algorithm has advantage and disadvantage.. Various Sorting algorithms have been compared on the basis of different factors like complexity, number of passes, number of comparison etc. My first target is to remove the demerits of various sorting algorithms. It is also seen that many algorithms are problem oriented so we will try to make it global oriented. Hence we can say that there are many future works which are as follows.

1) Remove disadvantage of various fundamental sorting and advance sorting.
2) Make problem oriented sorting to global oriented.

In the end we would like to say that there is huge scope of the sorting algorithm in the near future, and to find optimum-sorting algorithm, the work on sorting algorithm will go on forever.

REFERENCES

[1] Y.Han "Deterministic sorting in O(nloglogn) time and linear space", Proceeding of the thirty-fourth annual ACM symposium on theory of computing, Monteral Quebec, Canada
[2] Y.Han, M.Thorup, "Integer Sorting in O(nloglogn) time and linear space" proceesings of the 43rd symposium on foundations of Computer Science
[3] J. Alnihoud and R. Mansi, "An Enhancement of Major Sorting Algorithms", International Arab Journal of Information Technology, vol. 7, no. 1
[4] SultanullahJadoon et al., "Design and Analysis of Optimized Selection Sort Algorithm", International Journal of Electric & Computer Sciences IJECS-IJENS, Vol: 11 No: 01.
[5] Savina&SurmeetKaur, "Study of Sorting Algorithm to Optimize Search Results", International Journal of Emerging Trends & Technology in Computer Science, Volume 2, Issue 1.
[6] Md. Khairullah, "Enhancing Worst Sorting Algorithms", International Journal of Advanced Science and Technology,Vol. 56

