A Pragmatic Approach to Compare Various Sorting Techniques

Tanu Babbar\textsuperscript{1} and Mandeep Juneja\textsuperscript{2}

\textsuperscript{1}Computer Department, Kalpana Chawla Institute of Engg. & Technology, Hisar, Haryana-125001, India
babbar.tanu@gmail.com
tanu_babbar@rediffmail.com

\textsuperscript{2}Computer Department, Modern College of Technical Education, Hisar, Haryana-125001, India
mandeep.juneja@rediffmail.com

Abstract

Sorting is an operation that segregates items into groups according to specified criterion. It means arranging the elements of the array in ascending or descending order. There are many different algorithms, each one having certain advantages and disadvantages that have to be weighed against each other in the light of the particular application. Here we mainly focus on comparisons of various sorting techniques on the basis of their computational complexity, memory usage, time required, stability and their adaptability. We also explain some more factors of sorting like how each algorithm operates, that there is no best sorting algorithm. The advantages and disadvantages of each algorithm, worse-case asymptotic behavior is not always the deciding factor in choosing an algorithm and the initial condition (input order and key distribution) affects performance as much as the algorithm choice.

Keywords: Computational complexity, stability, asymptotic behavior, performance.

1. Introduction

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed fundamental activity. Objects are sorted in telephone books, in income tax files, in Tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic. Sorting may based on criteria like numerical, chronological, alphabetical, hierarchical etc. In Computer Science, due to obvious reasons, Sorting (of data) is of immense importance and is one of the most extensively researched subjects. It is one of the most fundamental algorithmic problems. So much so that it is also fundamental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc. It is estimated that around 25\% of all CPU cycles are used to sort data. Hence, sorting is a relevant and essential activity, particularly in data processing. The basic premise behind sorting an array is that its elements start out in some (presumably) random order and need to be arranged from lowest to highest. If the number of items to be sorted is small, a human reader may be able to tell at a glance what the correct order ought to be. If there are a large number of items, a more systematic approach is required. The ideal sorting algorithm would have the following properties:

- Stable: Equal keys aren't reordered.
- Operates in place, requiring O(1) extra space.
- Worst-case O(n·lg(n)) key comparisons.
- Worst-case O(n) swaps.
- Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.
- Finiteness-No matter what the input, algorithm must terminate after a finite no of steps.
- Clear and unambiguous.
- Effectiveness.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application. Each algorithm will have three sections-an introduction which will give pseudo code and a basic explanation of the algorithm and finally investigations in the run-time analysis of the algorithm.
2. Sorting Techniques

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.

2.1 Bubble Sort

It is probably one of the oldest, most easiest, straight forward, inefficient sorting algorithms. It is the algorithm introduced as a sorting routine in most introductory courses on Algorithms. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list.

Procedure:

```
BubbleSort( int a[], int n)
Begin
  for i = 1 to n-1
    sorted = true
    for j = 0 to n-1-i
      if a[j] > a[j+1]
        temp = a[j]
        a[j] = a[j+1]
        a[j+1] = temp
        sorted = false
    end for
  if sorted
    break from i loop
  end for
End
```

Examine the following :- (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8  6  10 3  1  2  5  4   } pass 0
6  8  3  1  2  5  4  10  } pass 1
6  3  1  2  5  4  8  10  } pass 2
3  1  2  5  4  6  8  10  } pass 3
1  2  3  4  5  6  8  10  } pass 4
1  2  3  4  5  6  8  10  } pass 5
1  2  3  4  5  6  8  10  } pass 6
1  2  3  4  5  6  8  10  } pass 7
```

Fig 1-Bubble sort

The above tabulated clearly depicts how each bubble sort works. Note that each pass results in one number being bubbled to the end of the list.

2.2 Selection Sort

The idea of Selection Sort is rather simple. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where it’s supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-Ith index of the list.

Procedure:-

```java
void SelectionSort(int a[], int array_size)
{
  int i;
  for (i = 0; i < array_size - 1; ++i)
  {
    int j, min, temp;
    min = i;
    for (j = i+1; j < array_size; ++j)
    {
      if (a[j] < a[min])
        min = j;
    }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
  }
}
```

Consider the following :-

```
8  6  10 3  1  2  5  4   } pass 0
1  6  10 3  8  2  5  4   } pass 1
1  2  10 3  8  6  5  4   } pass 2
1  2  3  10 8  6  5  4   } pass 3
1  2  3  4  8  6  5  10  } pass 4
1  2  3  4  5  6  8  10  } pass 5
1  2  3  4  5  6  8  10  } pass 6
1  2  3  4  5  6  8  10  } pass 7
```

Fig 2- selection sort

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sub list is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).
2.3 Insertion Sort

The Insertion Sort algorithm is used when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead). It is also stable and has the recursive base case. It is implemented by inserting a particular element at the appropriate position. The first pass starts with comparison of 1st element with the 0th element. It means in every pass an element is compared with all elements before it.

**Procedure:**

```c
void insertionSort(int a[], int array_size) {
    int i, j, index;
    for (i = 1; i < array_size; ++i) {
        index = a[i];
        for (j = i; j > 0 && a[j - 1] > index; j--)
            a[j] = a[j - 1];
        a[j] = index;
    }
}
```

Examine the following:

- 8 6 10 3 1 2 5 4 } pass 0
- 6 8 10 3 1 2 5 4 } pass 1
- 6 8 10 3 1 2 5 4 } pass 2
- 3 6 8 10 1 2 5 4 } pass 3
- 1 3 6 8 10 2 5 4 } pass 4
- 1 2 3 6 8 10 5 4 } pass 5
- 1 2 3 5 6 8 10 4 } pass 6
- 1 2 3 4 5 6 8 10 } pass 7

**Fig 3:** Insertion Sort

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. We start from 8 and the next element 6. Then remove 6 from its current position and "insert" it back to the top. That constituted pass 1. Repeat the same process and do the same thing for 3 which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its < (6,8,10) but > 1. As we carry on till we reach the end of the list we'll find that the list has been sorted.

2.4 Quick sort

It is the most popular sorting algorithm. Its virtue is that it sorts in-place (even though it's a recursive algorithm) and that it usually is very fast. One reason for its speed is that its inner loop is very short and can be optimized very well. The fourth number distribution (array is sorted, except for the last 256 items which are random) is very expectedly a pathological case for the vanilla quick sort and thus was skipped with the larger arrays. Very unexpectedly this distribution was pathological for the optimized quick sort implementation too, with larger arrays, and thus I also skipped it in the worst cases (because else it would have affected negatively the scale of the bar charts). I don't have any explanation of why this happened.

**Procedure:**

```c
int functionPartition (Array A, int Lb, int Ub);
begin
    select a pivot from A[Lb]...A[Ub];
    reorder A[Lb]...A[Ub] such that:
    all values to the left of the pivot are <= pivot
    all values to the right of the pivot are >= pivot
    return pivot position;
end;

procedure QuickSort (Array A, int Lb, int Ub);
begin
    if Lb < Ub then
        M = Partition (A, Lb, Ub);
        QuickSort (A, Lb, M - 1);
        QuickSort (A, M, Ub);
    End;

Take, for example, the list

- 7 9 4 5 1 3 8 3 } low = 7; High = 3; middle = 1
- 3 1 3 4 5 9 7 8 } lows = 3;4; highs = 1;8; middle = 3;9
- 1 3 3 457 8 9 } low = 4, high = 7, middle = 5
- 1 3 3 4 5 7 8 9

**2.5 Shell sort**

Named after its inventor Donald Shell, Shell sort is an improved version of the insertion algorithm. It bases itself in that insertion sort is most efficient when the list is mostly organized. In a way, Shell sort is an insertion sort in which the number of ‘jumps’ from value to value (which in insertion sort is 1) starts large and ends small. This can be visualized by arranging the list into a 2-dimensional array and arranging each column by insertion sort, and sequentially repeating this with a decreasing number of columns.

```c
void sort(int data[])
{
    for(int gap = data.len/2; gap>0; gap/=2)
        for(i=0;i<n-d;i++)
            for(int j=i+gap;j<n;j+=gap)
                if(data[j] > data[j+gap])
                    swap(data[j],data[j+gap]);
}
```

if (a[i]>a[i+d])
{
    temp=a[i];
a[i]=a[i+d];
a[i+d]=temp;
}

if(d==1)
    return;
d=d/2.0+0.5;

Take, for example, the list
Here's an example list -
7 9 4 5 1 3 8 3
To begin, we divide the list into two:
7 9 4 5   1 3 8 3
We then swap every corresponding value from each side, putting the smaller values on the left and bigger values on the right:
1 3 4 3   7 9 8 5
We then split each group in two:
1 3 4 3   7 9 8 5
And then swap values from each group, again:
1 3 4 3  7 5 8 9
It is pretty close to being sorted, so we now do the whole list:
1 3 3 4 5 7 8 9

3. Asymptotic Notation

A problem may have numerous algorithmic solutions. In order to choose the best algorithm for a particular task, there is a need to judge how long a particular solution will take to run. Or, more accurately, we need to be able to judge how long two solutions will take to run, and choose the better of the two.

3.1 Asymptotic complexity

It is a way of expressing the main component of the cost of an algorithm, using idealized units of computational work.

3.2 Big O

This notation is used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. In computer science, big O notation is used to classify algorithms by how they respond (e.g. in their processing time or working space requirements) to changes in input size. Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols $o$, $\Omega$, $\omega$, and $\Theta$, to describe other kinds of bounds on asymptotic growth rates.

3.2.1 Properties of Big O

The definition of big O is pretty ugly to have to work with all the time, kind of like the "limit" definition of a derivative in Calculus. Here are some helpful theorems you can use to simplify big O calculations:

- Any $k$th degree polynomial is $O(n^k)$.
- $a n^k = O(n^k)$ for any $a > 0$.
- Big O is transitive. That is, if $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$.
- $\log_a n = O(\log_b n)$ for any $a, b > 1$. This practically means that we don’t care, asymptotically, what base we take our logarithms to. (I said asymptotically. In a few cases, it does matter.)
- Big O of a sum of functions is big O of the largest function. How do you know which one is the largest? The one that all the others are big O of.
- Big O of a sum of functions is big O of the largest function. How do you know which one is the largest? The one that all the others are big O of.

$$f(n) = O(g(n))$$ is true if $$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$ is a constant.

3.3 Complexity of various sorting techniques

<table>
<thead>
<tr>
<th>Sort</th>
<th>Average</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Sort</td>
<td>Average</td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>Quick sort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n \log(n))$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table 1 - complexity comparison

4. Comparison
4.1 Stability of the algorithms
- Insertion sort is stable because the algorithm preserves the original order of equal elements.
- Bubble and selection Sorts are stable ones, whereas Shell Sorts are unstable but it will decided by the way they have been implemented.
- Quick sort is also not stable. For example, if all of the elements in the list are the same, the first and last elements exchange in the first split process. It will also not preserves the original order later on.

4.2 Advantages of the algorithms
- Insertion sort is the easiest algorithm to program and debug. Insertion sort for small size inputs is efficient. It is an in-place sorting algorithm.
- Selection sort is preferable to insertion sort in terms of number of writes ($\Theta(n)$ swaps versus $O(n^2)$ swaps).
- The primary advantage of the bubble sort is that it is popular and easy to implement. Furthermore, in the bubble sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.
- Quick sort’s running time for an average case is $O(n \log n)$, which is also faster than $O(n^2)$. For a large number of input sizes, the performance is better than that of the Insertion sort but randomized Quick sort can makes the worst case.
- The Shell sort is simple to understand, easy to implement, and has good performance for small to medium arrays.

4.3 Strategy of application of the sorting algorithms
- For small number of data sizes, we can use Insertion sort or bubble sort. It is easy to program and run faster for small size input.
- Quick sort is a preferred sorting method in applications that require a sorting algorithm that is usually very fast, but on occasion can have a longer running time.
- The basic Shell Sort algorithm is good for numbers, bad for strings.

4.4 CPU time issues
CPU time gives general criteria to compare the sorting algorithms. It tells which sorting algorithm is fast. But we cannot tell exactly how much CPU time used by the measured sorting algorithms. The reason can be described in several aspects. The first is the extra time from loading arrays. The second is the extra time from calculating the two global variables. As far as those extra times are very smaller than the time used in comparisons and assignments, the CPU time is correct.

The usage of different data type of the two global variables, i.e., the number of comparisons and the number of assignments, also affects our calculations significantly. When we use int type of integers or float type variables, those number are out of range if the data size is over 100000. When we use double long type of integers, our calculation will take about 60 CPU hours for Insertion sort if the data size is 200000.
Graph 1: Total time taken for an algorithm to sort 1000 random lists.

Graph 2: Average time taken for an algorithm to sort one random list

5. Limitations of pragmatic comparison

Although an effective way to compare how different algorithms perform on a system, the main disadvantage of pragmatic data is that it is entirely dependent on the computer it has been obtained on. Very different results can arise from running algorithms on systems as dissimilar as a mainframe and a cell phone. Different variables, such as memory, processor, operating system, and currently running programs can affect the runtime of the algorithms. This was kept constant in this investigation by always running all the algorithms on the same system. Nonetheless, this type of study does not come to any system-independent conclusions.

Similarly to the previous statement, programming language and compiler/interpreter used can also affect the speed of programs. If the algorithms were implemented in a language such as C, the result would have probably been much faster sorting due to C’s compiled nature also if the language the algorithms were implemented with was particularly efficient at, for example, recursion. Quick sort might have gotten a much smaller runtime. This leads to another limitation of this sort of investigation: it does not conclude anything about code-independent algorithms.

The solution to this drawback would be to use a system-independent method of analyzing algorithms. By obtaining relevant data from the analysis of algorithms, a concrete comparison regarding their speed can be used to obtain system and programming language independent results. Apart from only random and mostly sorted lists, arrays sorted in reverse order and arrays with duplicate elements could also have been investigated to lead to more thorough conclusions on the best algorithm.

6. Conclusions
The data obtained reveals that the speed of each algorithm, from slowest to fastest for a sufficiently large list, ranks as follows:

1. Quicksort
2. Shellsort
3. Selectionsort
4. Insertionsort
5. Bubble sort

There is a large difference in the time taken to sort very large lists between the fastest two and the slowest three. This is due to the efficiency Quick sort and Shell sort have over the others when the list sorted is sufficiently large. Also, the results show that for a very small list size, only selection sort and insertion sort are faster than Quick and Shell sort, and by a very small amount. It should also be noted that Quick sort is eventually faster than Shell sort, though it is slower for small lists. In a practical sense, the difference between the speeds of Quick sort and Shell sort are not noticeable unless the list is very large (has over 1000 items). For very small lists, the difference between all the algorithms is too small to be noticeable. However, Shell sort is much less system-intensive than Quick sort because it is not recursive.

For lists expected to be very large (for example, the articles in a news website’s archive, or the names in a phonebook) Quick sort should be used because, despite its larger use of space resources, it is significantly faster than any other of the algorithms in this investigation when the array is sufficiently large.

7. References