Efficiency of Different Sorting Algorithms in Java

Alexander Alvarez

Fall 2024

Introduction

The time efficicency of different sorting algorithms on an array with varying sizes are to be measured. The cost of sorting and the use cases of certain sorting algorithms are discussed as well. The sorting algorithms in question are namely:

- selection sort
- insertion sort
- merge sort
- quick sort

The general behaviour of these algorithms are described as well as how they were created and implemented for testing.

Method

The algorithms will be tested using arrays of sizes `n`. I chose ten sizes starting from 10 elements and doubling up to 5120 (`n =` {10, 20,..., 2540, 5120}).

I created a `loop` amount of unsorted arrays with the same size. The average time for an algorithm to sort one array is the quotient of the total time to sort a `loop` amount of arrays divided by `loop`. I chose this `loop` value to be `1000`. Due to consistency, only the minimum time out of ten runs for each `n` will be measured.

Each algorithm has a method with two functions (this is the method that's run ten times for each `n`). The first is creating a two dimensional array with a `loop` amount of unsorted one dimensional arrays with the size `n`. Below is the code used for that function:

```
int[][] array = new int[loop][n]; // creates 2D array
for(int i = 0; i < loop; i++) // for each 1D array
  for(int j = 0; j < n; j++) // for each index of the 1D array
    array[i][j] = rnd.nextInt(n*10); // randomizes a value
```

The second function measures the time for an algorithm to sort a `loop` amount of arrays. The sorting algorithm runs in a `for` loop that sorts a different array for each loop iteration. The time is taken right before and after this `for` loop for which the method returns the time difference.

```
long t0 = System.nanoTime(); // time 1
for(int i = 0; i < loop; i++) // cycles through each 1D array
  selection_sort(array[i]); // sorts that 1D array
long t1 = System.nanoTime(); // time 2
return (t1 - t0); // returns difference between time 2 and 1
```

A `swap` method is a simple method that swaps the elements in an array at two specified indices. The `swap` method is used in most of the sorting algorithms discussed in this report.

The sorting algorithms will also be run for a sufficient time before the actual benchmark to enable the just-in-time compiler for time optimizations.

selection sort

Selection sort selects the first element in the array and marks it as the minimum value (`min`) before searching through the whole array after the selected element for an even smaller value. Elements with an even smaller value is marked as the new `min`. The algorithm swaps the first element and `min` if both elements are at a different index from eachother. The algorithm then repeats this on every element in the array after the first.

```
public static void selection_sort(int[] array){
  for(int i = 0; i < array.length - 1; i++){ // for each element
    int min = i; // mark the min element
    for(int j = i + 1; j < array.length; j++) // begin search
      if(array[j] < array[min]) min = j; // mark new min
    if(!(min == i)) swap(array, min, i); // swap if different
  }
}
```

The code for the algorithm is quite simple and easy to implement. Running the code for selection sort yields the following graph below.

Selction sort is generally inefficient on greater array sizes due to the algorithm having to search through the whole array regardless of how unsorted

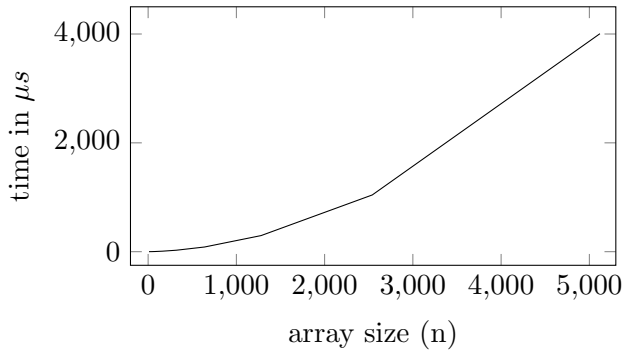

Figure 1: Selection sort time

the array is. The best, average and worst case scenarios for the time complexity are therefore the same which is $O(n^2)$. Selction sort is thus more useful on smaller arrays.

Another propery of selection sort is that it is an unstable sorting algorithm, i.e. the order of elements with the same value in the unsorted array may not be the same as in the sorted array. This property can be ignored if the order doesn't matter. Unstable sorting algorithms can even be faster than stable sorting algorithms if they don't need to keep track of the order.

insertion sort

Insertion sort works by going through each element in the array until it finds an element that is smaller than the previous element. The algorithm then swaps the smaller element with every previous element until it reaches a previous element that is no longer greater than the smaller element. The algorithm then continues through the array after sorting the smaller element.

```
public static void insertion_sort(int[] array){
  for(int i = 0; i < array.length; i++) // goes through the array
    for(int j = i; j > 0 && array[j] < array[j - 1]; j--) // checks elements
      swap(array, j, j - 1); // swap if element is smaller
}
```

Inseriton sort uses more write operations than selection sort which only swapped the smallest element instead of every element in between. The code for insertion sort is also quite simple and easy to implement. Running the code for insertion sort yields the following graph below.

Insertion sort is much more efficient compared to selection sort by around half the sorting time for most array sizes. The more sorted an array is, the more efficient insertion sort becomes. The best case scenario for insertion sort is on an already sorted array which gives the time complexity $O(n)$.

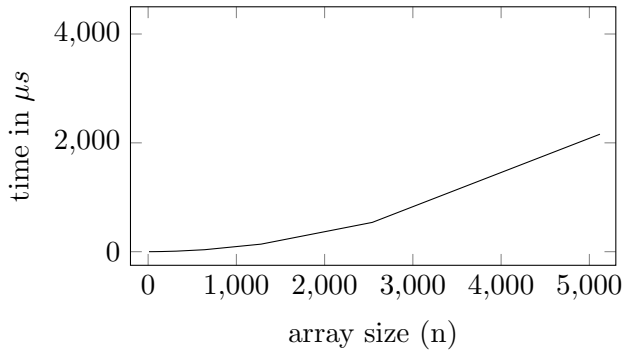

Figure 2: Insertion sort time (unsorted array)

However, due to how the algorithm works, insertion sort has the time complexity $O(n^2)$ in the average scenario which is seen in the graph above. The worst case scenario would be a sorted array that is in reverse order. The graph for insertion sort in the worst case scenario is displayed below:

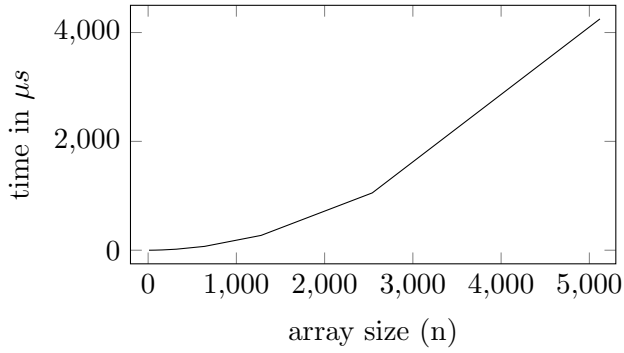

Figure 3: Insertion sort time (sorted array in reverse order)

Insertion sort performs slightly worse than selection sort in the worst case. This makes insertion sort better than selection sort for use on smaller and nearly sorted arrays.

Inseriton sort is a stable sorting algorithm unlike selection sort. Inseriton sort is thus applicable in cases where the order of elements with equal value matters.

merge sort

Merge sort works by recursively dividing an unsorted array into smaller subarrays before merging the subarrays back together in a sorted sequence.

A `sub` array is first created with the size `n`. We perform the sorting between two specified indices of the `sub` array rather than actually creating several subarrays. These indices will be the "lower" (`lo`) and "upper" (`up`)

indices of the `array`. The `array` is split in the "middle" (`mid`) where the lower and upper half is recursively sorted. The sorted halves is lastly merged back together.

```
private static void merge_sort(int[] array, int[] sub, int lo, int up){
  if(lo != up){ // stop if sub is too small
    int mid = lo + (up - lo)/2; // calculates mid
    merge_sort(array, sub, lo, mid); // sort lower half
    merge_sort(array, sub, mid+1, up); // sort upper half
    merge(array, sub, lo, mid, up); // merge halves
  }
}
```

Elements between the indices `lo` and `up` from `array` are copied onto the same indices in `sub`. These elements are already sorted in two halves seperated by the `mid` index. The first indices of the lower and upper half is then stored in two respective variables `i` and `j`.

A `for` loop then goes from `lo` to `up` where the smaller value of either half is inserted from `sub` to `array`. The index with the smaller value, `i` or `j`, is then incremented by 1. If one half is emptied before the other, i.e. `i` or `j` increments above their respective upper index `mid` or `up`, then the rest of the elements in the non-empty half is sequentially inserted.

```
private static void merge(int[] array, int[] sub, int lo, int mid, int up){
  for(int i = lo; i <= up; i++) sub[i] = array[i]; // copy to sub
  int i = lo; // first index of lower half
  int j = mid + 1; // first index of upper half
  for(int k = lo; k <= up; k++){ // for every element from lo to up
    if(i > mid) array[k] = sub[j++]; // if lower half is empty
    else if(j > up) array[k] = sub[i++]; // if upper half is empty
    else if(sub[i] < sub[j]) array[k] = sub[i++]; // insert from lower half
    else array[k] = sub[j++]; // insert from upper half
  }
}
```

The code for the merge sort algorithm is more complicated compared to the previous algorithms. Merge sort uses an additional array to sort which requires more memory and can be unfavourable if memory usage matters. Running the code for the merge sort algorithm yields the following graph below.

Merge sort is much faster than the other algorithms. There is however a minor optimization that can be made to the code. We can decrease the overall amount of copying by first copying all the elements from `array` to `sub` before calling the `merge_sort` method. Then we switch the arguments

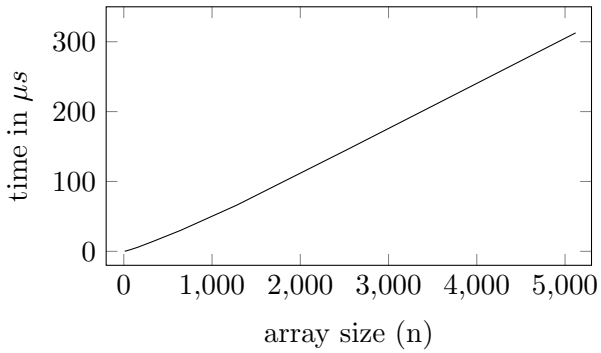

Figure 4: Merge sort time before optimization

`array` and `sub` with eachother in the recursive call methods whithin the `merge_sort` method. This allows the removal of the `for` loop that does the copying in the `merge` method. Running the code with these optimizations implemented yields the following graph:

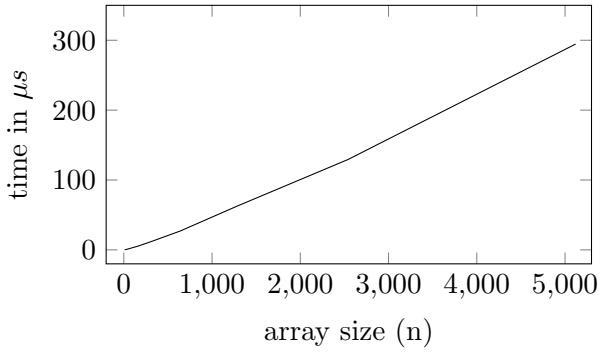

Figure 5: Merge sort time after optimization

The improvements are minor but can be seen in the graph. Merge sort divides an array into $log(n)$ subarrays which takes a time of $O(n)$ to merge back into one array. This has a time complexity of $O(n * log(n))$ which is difficult to conclude using the graph only. The time complexity is the same for every scenario since the algorithm will perform the same operation regardless of how sorted the arrays is. Overall, merge sort shows a huge increase in performance compared to the previous algorithms. As a stable sorting algorithm, merge sort is very useful on bigger array sizes where order matters and memory is of no concern.

quick sort

Quick sort works by recursively sorting around a pivot point. The algorithm uses the lower and upper indices of the `array` to partition the `array` into two

parts. The `partition` method gives a pivot point that is used to get new lower and upper indices for the two parts. This algorithm is run recursively until the array is sorted.

```
private static void quick_sort(int[] array, int lo, int up){
  if(lo < up){ // stops if part is too small
    int pi = partition(array, lo, up); // partitions and gives pivot
    quick_sort(array, lo, pi-1); // sorts lower part
    quick_sort(array, pi+1, up); // sorts upper part
  }
}
```

The sorting takes place in the `partition` method. The value of an arbitrary element is first chosen as the pivot. The algorithm then places all elements with smaller and greater values than the pivot in each side of the pivot. The index of the pivot is then returned.

```
private static int partition(int[] array, int lo, int up){
  int pivot = array[up]; // chose pivot value
  int j = (lo - 1); // smaller value part index
  for (int i = lo; i <= up - 1; i++) // for all elements from lo to up
    if (array[i] < pivot) swap(array, i, ++j); // swaps smaller value
  swap(array, j+1, up); // corrects index of pivot element
  return (j + 1); // returns pivot index
}
```

The result is an array with two parts of unsorted elements where one part consists of values smaller than the pivot, and the other part consisting of values greater than the pivot. The array is ultimately sorted when the algorithm is run recursively. Running the code for quick sort yields the following graph:

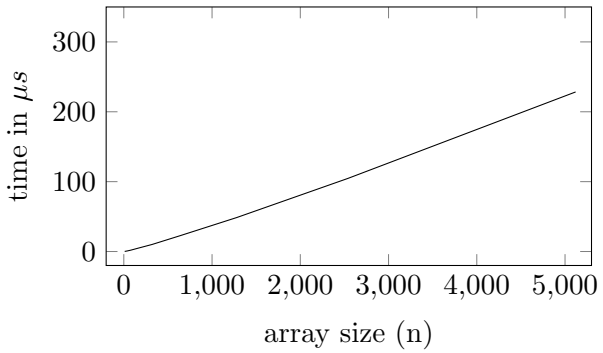

Figure 6: Quick sort time

Quick sort is slightly faster than merge sort and has the time complexity $O(n * log(n))$ for the best and average case scenarios. Quick sort is the fastest compared to previous algorithms but can have the time complexity $O(n^2)$ in the worst case.

Conclusion

Selection sort is the most inefficient out of the four algorithms. With a time complexity of $O(n^2)$ it's best used on smaller arrays.

Insertion sort is more efficient than selection sort but has the same time complexity of $O(n^2)$. As a stable sorting algorithm, insertion sort is best used on smaller arrays that are nearly sorted and where order of equal valued elements matters.

Merge sort is another stable sorting algorithm that is far more efficicent than both selection and insertion sort. With a time complexity of $O(n * log(n))$ it's best used on bigger arrays where order matters.

Quick sort is generally more efficient than merge sort and has a time complexity of $O(n * log(n))$ in most cases. Quick sort can though be slower than merge sort in worst case scenarios which I predict is rare. Quick sort is an unstable sorting algorithm which, due to its generally faster speed, is used more than merge sort in cases where order does not matter.