Time complexity in sorting algorithms in C

Sam Serbouti serbouti@kth.se

September 18, 2024

In this report, I will explore the time complexity of various sort algorithms implemented in C, and compare their efficiency. For better comparison between algorithms, I will perform warm-ups operations of `benchmark(99000)` (to fully optimize the cache) for each test.
To compare time complexities, I measure the time it takes to sort the array of size n. But I will do so 5 times, and extract a minimum, maximum and average values of these 5 experiments. They loosely correspond to best, worst and average cases. I then plot this as n grows.
5 experiment per array size is **not** ideal: sometimes the average runtime is greater than the worst case or lower than the best case. However, given the time I had to wait for each experiment, it was the best compromise I could find.
For most algorithms, I will use this swap procedure:

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

1 Selection sort runtime

Selection sort selects the smallest element of the sorted part of the array and swaps it with the first element of the unsorted part of the array. It thus does two sequential passes through the array (which is not ideal for big datasets).

```c
for (int i = 0; i < n - 1; i++) {
    int candidate = i; //candidate for smallest element
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[candidate]) {candidate = j;}
    }
    if (candidate != i) {swap(&arr[i], &arr[candidate]);}
}
```

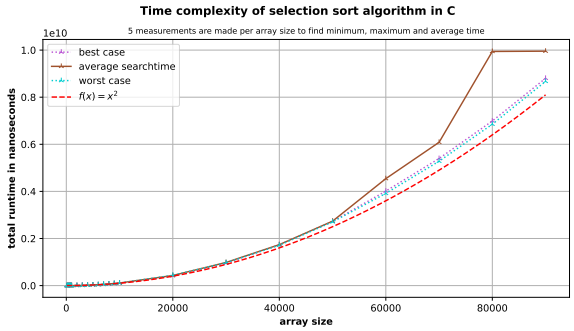

Figure 1: The time complexity for selection sort follows $\mathcal{O}(n^2)$

2 Insertion sort runtime

Insertion sort splits the array into two groups and builds a sorted array one item at a time. This algorithm operates *in-place*, so it only requires a constant amount of additional memory.

```
for (int i = 1; i < n; i++) {
    for (int j = i; j > 0 && arr[j] < arr[j - 1]; j--)
        {swap(&arr[j], &arr[j - 1]);}
}
```

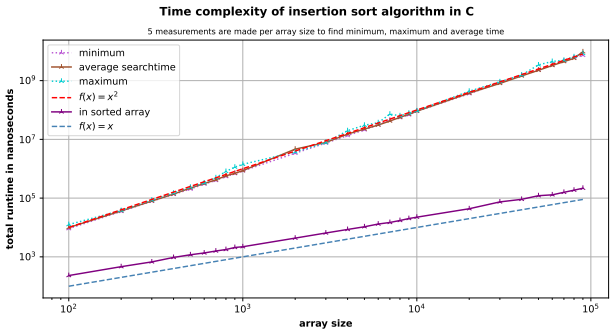

Figure 2: The time complexity for insertion sort also loosely follows $\mathcal{O}(n^2)$

If I plot *(on a log scale)* my measurements for a random array and a sorted array (best case), I notice that $t_{average}(n) \hookrightarrow \mathcal{O}(n^2)$ and $t_{best}(n) \hookrightarrow \mathcal{O}(n)$.

3 Merge sort runtime

A merge sort algorithm has a *divide and conquer* approach to the problem : it recursively divides the array into two halves until each sub-array has one element (this is handled by `sort()`) and then merges the two sorted sub-arrays into one sorted array by using a temporary auxiliary array to help with merging the elements (done by `merge()`).

```c
void merge(int *org, int *aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++) {aux[k] = org[k];} //copy
    int i = lo;         // start index for left subarray
    int j = mid + 1;    // start index for right subarray
    for (int k = lo; k <= hi; k++) {
        if (i > mid) {org[k] = aux[j++];}
        else if (j > hi) {org[k] = aux[i++];}
        else if (aux[i] <= aux[j]) {org[k] = aux[i++];}
        else {org[k] = aux[j++];}
    }
}
void sort(int *org, int *aux, int lo, int hi) {
    if (lo >= hi) {return;}  //array has 1 or 0 elements
    int mid = (lo + hi) / 2; //midpoint
    sort(org, aux, lo, mid);      // sort left half
    sort(org, aux, mid + 1, hi);  // sort right half
    merge(org, aux, lo, mid, hi);
}
void mergeSort(int *org, int n) {
    int *aux = (int*)malloc(n * sizeof(int));
    sort(org, aux, 0, n - 1);
}
```

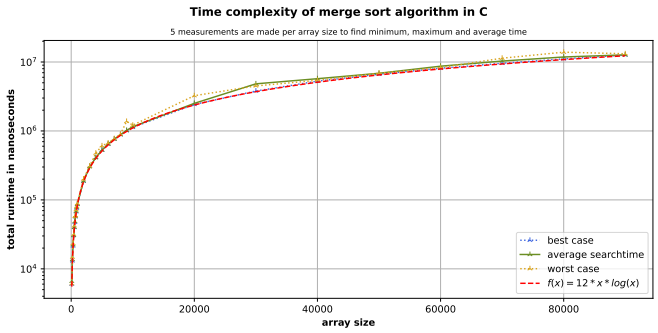

Figure 3: The time complexity for merge sort follows $\mathcal{O}(n \cdot log(n))$

We get $t(n) \hookrightarrow \mathcal{O}(n \cdot log(n))$ which is an ideal complexity. However, merge sort requires additional space for the temporary array used during merging.

4 Quick sort runtime

This algorithm chooses the last element of the array as a pivot, partitions the array around it, checks that all elements at its left are smaller and repeats in the two parts thus created until there is only one or no elements left to split.

```c
int partition(int array[], int low, int high) {
    int pivot = array[high]; //last element as pivot
    int i = low - 1; //idx of smaller element
```

```
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++; //move index of the smaller element
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]); //replace pivot
    return (i + 1); //index of pivot
}
void quickSort(int array[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(array, low, high);
        quickSort(array, low, pivotIndex - 1);   //sort the left part
        quickSort(array, pivotIndex + 1, high);  //sort the right part
    }
}
```

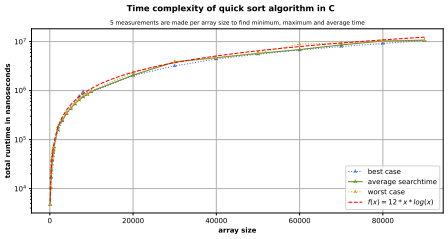

Figure 4: We obtain similar results for a quick sort algorithm

5 Conclusion

	selection	inssertion	merge	quick
time	n^2	n^2	$nlog(n)$	$nlog(n)$
space	1	1	n