Complexity Analysis

Jonatan Lind
Charlie Habolin

Karlstad University

November 23, 2020

Table of Contents

Introduction

The current algorithms (bubble sort and linear search) are causing slowdowns for some of our clients. We need a more efficient sorting- and search-algorithms to eliminate slowdown. To find out which algorithm is the best for our use case we need to write a program to measure the time each algorithm takes. We need to test the best, worst, and average case for each algorithm and for different number of elements to see how it effects the run-time. We are also going to analyze the time complexity of each algorithm to find out how the time spent grows in respect to the number of elements.

Algorithms

Bubble sort

The bubble sort algorithm is a compact algorithm consisting only of two loops and a conditional swap. While sorting a vector, the sorting makes larger value to "bubble up" by iterative swap every value larger than the current value. This forces the algorithm to on average do n² comparisons, making it a rather unused algorithm in real application, due to a comparatively poor performance.

Best case

Comparisons:	$O(n)$
Swaps:	$O(1)$

In the best case, the vector is already sorted in the direction the implementation sorts, usually ascending. In this case, no swaps takes place, only n comparisons are done for verification.

Average and worst case

Comparisons:	$O(n^2)$
Swaps:	$O(n^2)$

The average and worst case are the same, all other vector populations except for the sorted one requires n² comparisons. To force the average/worst case for benchmark, the vector is sorted randomly before using bubble sort.

Insertion sort

Insertion sort traverses every element in the vector, and checks if the value is smaller than the previous element. If so they are swapped and the previous-previous value are compared until a value is not smaller than the compared one. The next element are then evaluated.

The real-world use case is for insertions into a already sorted vector. Using this algorithm for insertions gives it a complexity close to O(n), due to only needing to do comparison and a insert (swap) of the new element into the vector.

Best case

Comparisons:	$O(n)$
Swaps:	$O(1)$

In the best case, the vector is already sorted in the direction the implementation sorts, usually ascending. In this case, no swaps takes place, only n comparisons are done for verification.

Average and worst case

Comparisons: $O(n^2)$
Swaps: $O(n^2)$

The average and worst case are the same, all other vector populations except for the sorted one requires n^2 comparisons and swaps. To force the average/worst case for benchmark, the vector is sorted in reverse order before using insertion sort.

Quick sort

Quick sort operates by selecting a element as a *pivot* point. All numbers larger and smaller are then partitioned into two children. Recursively this are done to the two children again. When the leaf of the binary tree are reached, the tree are collapsed *in-order* to produce a sorted vector.

Best case and average case

Time complexity: $O(n \cdot \log(n))$

Best and average case share the same time complexity. This make quick sort desirable for real-world applications for use on randomly populated vectors. To force the best/average case for benchmark, the vector is sorted randomly before using quick sort.

Worst case

Time complexity: $O(n^2)$

The worst complexity is in the case the vector is already sorted in the direction the implementation sorts. To force the worst case for benchmark, the vector is sorted in forward order before using quick sort.

Linear search

Linear search is the most simple for of search, the algorithm is just the linear traversal of an vector from start to end, stopping if the searched value is found in the array.

Best case

Time complexity: $O(1)$

In a best case, the first value is the searched value. To force the best case for benchmark, the value to search for is the first element in the vector, in our case *0* with a predefined vector where every element value is the elements position in the vector.

Average case

Time complexity: $O(\frac{n}{2})$

If the vector is populated randomly, the average search time is traversing half the vector. To force the average case for benchmark, the value to search for is random, and the vector is randomly populated.

Worst case

Time complexity: $O(n)$

The worst case is if the value is the last in the vector, or non-existent. Then all the elements in the array have to be traversed. To force the worst case for benchmark, the value to search for is -1, and the vector is randomly populated with positive integers.

Binary search

Binary search is an algorithm where by a vector is split in two, a check if that middle value is the searched for. If not, the larger or lower split (depending on the searched value) is discarded and the remaining part is again split in two, and the algorithm repeating until the final value is reached.

A requirement for using binary search is that the vector needs to be sorted beforehand.

Best case

Time complexity: $O(1)$

In the best case, the vectors median value is the searched value. To force the best case for benchmark, the value to search for is median value, in our case *array length / 2* with a predefined vector where every element value is the elements position in the vector.

Average case

Time complexity: $O(\log(n))$

If the vector is populated randomly, the average search time is log(n). To force the average case for benchmark, the value to search for is random, and the vector is sorted.

Design and implementation

This benchmark project aims to be be efficient and clearly written. While no real complete road-map of all parts and stages where made beforehand, care was taken to discuss and reason if a feature was optimal and would integrate well with the rest of the project.

Different functions in the project are clearly separated into different {.h, .c} pairs, with only necessary functions exposed in the header file. This create a modularity between parts, preventing wrong use of a interface.

This project utilize function-pointer in many parts of the code to prevent a need of code-duplication due to C not having support for declaration of generic functions (templated functions).

Compiling the project with all possible optimizations enabled, allows the compiler to improve the efficiency of the code. Also all warnings are treated as errors. Using const wherever possible prevents unintentional modifications of arguments and variables. This also allows the compiler to assert values and pave way for heavier optimizations.

By only utilizing the stack and not allocating anything on the heap, we prevent any memory leak from occurring.

Result

Every algorithm are timed for six different vector lengths doubling each step, sorts starting at 1k elements and search starting at 10k elements. For every vector lengths, sorts are done 15 times to get a average value without extreme point measurements. For sorting, 5k iterations are done. To benchmark *best*, *average* and *worst* characteristics of every algorithm, the vectors are setup to facilitate this. Every characteristics are then timed according to rules above.

Result from the benchmark are in the appendix of this report.

Problems

One of the problems during the development was that one of the computers was misbehaving. The benchmark was not measuring the time correctly. It was inconsistent and often had the same time between many tests even when the number of elements was doubled each pass. On some of the tests the measured time was recorded as zero. This resulted in the big-O calculations on the machine to be useless. Even when the same benchmark was run several times, the inconsistency made it so no conclusion could be made about our predicted time complexity.

```
QuickSort:

Best:
Size(N)     Time(T)              T/N                 T/NlogN             T/N²
1000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
2000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
4000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
8000  0.0010416666667 | 1.3020833333e-07  3.3360320609e-08  1.6276041667e-11
16000 0.0010416666667 | 6.5104166667e-08  1.5485801294e-08  4.0690104167e-12

Average:
Size(N)     Time(T)              T/N                 T/NlogN             T/N²
1000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
2000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
4000  0.0010416666667 | 2.6041666667e-07  7.2296593420e-08  6.5104166667e-11
8000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
16000 0.0010416666667 | 6.5104166667e-08  1.5485801294e-08  4.0690104167e-12

Worst:
Size(N)     Time(T)              T/NlogN             T/N²                T/N³
1000  0.0000000000000 | 0.0000000000e+00  0.0000000000e+00  0.0000000000e+00
2000  0.0020833333333 | 3.1555807370e-07  5.2083333333e-10  2.6041666667e-13
4000  0.0093750000000 | 6.5066934078e-07  5.8593750000e-10  1.4648437500e-13
8000  0.0333333333333 | 1.0675302595e-06  5.2083333333e-10  6.5104166667e-14
16000 0.1312500000000 | 1.9512109630e-06  5.1269531250e-10  3.2043457031e-14
```

Example of bogus measurement values.

The machine was running the program through “Windows subsystem for Linux” which could be the source of the problem. To try this theory, the program was instead compiled for Windows through PowerShell. The program was run but the time was still inconsistent and had times that were zero. The problem was the machine and not the program. The solution was to run the program on another machine that was running Linux natively. This solved the problems with the measurements of time.

Conclusion

Just because an algorithm has a better time complexity does not mean it is always better. There are several things to consider when choosing the best algorithm for each situation. Time taken is just one of the things that makes an algorithm the best fit. For example, in some cases when n is small, Insertion sort is sometimes quicker than quick-sort, even though Insertion sort has a greater time complexity.

Big O time complexity is an approximation of the time and is most accurate when n is large. In our program when testing with smaller number of elements the big O was often inconsistent, but when the number of elements tested increased the big O time became more consistent and more in line with the predicted behavior for the algorithm.

Time complexity is a convenient way of comparing algorithms with each other, without needing to go into each algorithms source code.

Appendix

Output of all sort and search algorithms.

```
Bubble Sort:

Best:
Size(N)     Time(T)            T/(N/2)            T/N                T/NlogN
1000  0.0000073333333 | 1.4666666667e-08  7.3333333333e-09  2.4444444444e-09
2000  0.0000022666667 | 2.2666666667e-09  1.1333333333e-09  3.4332718419e-10
4000  0.0000039333333 | 1.9666666667e-09  9.8333333333e-10  2.7299193675e-10
8000  0.0000073333333 | 1.8333333333e-09  9.1666666667e-10  2.3485665709e-10
16000 0.0000140666667 | 1.7583333333e-09  8.7916666667e-10  2.0912026067e-10

Average:
Size(N)     Time(T)            T/NlogN            T/N²               T/N³
1000  0.0025803333333 | 8.6011111111e-07  2.5803333333e-09  2.5803333333e-12
2000  0.0030592666667 | 4.6338062221e-07  7.6481666667e-10  3.8240833333e-13
4000  0.0145073333333 | 1.0068775484e-06  9.0670833333e-10  2.2667708333e-13
8000  0.0778887333333 | 2.4944573912e-06  1.2170114583e-09  1.5212643229e-13
16000 0.3634466000000 | 5.4031313554e-06  1.4197132812e-09  8.8732080078e-14

Worst:
Size(N)     Time(T)            T/NlogN            T/N²               T/N³
1000  0.0011764000000 | 3.9213333333e-07  1.1764000000e-09  1.1764000000e-12
2000  0.0038879333333 | 5.8889700161e-07  9.7198333333e-10  4.8599166667e-13
4000  0.0159728000000 | 1.1085878663e-06  9.9830000000e-10  2.4957500000e-13
8000  0.0626120000000 | 2.0052061382e-06  9.7831250000e-10  1.2228906250e-13
16000 0.2507077333333 | 3.7271137356e-06  9.7932708333e-10  6.1207942708e-14
```

```
Insertion Sort:

Best:
Size(N)     Time(T)            T/(N/2)            T/N                T/NlogN
1000  0.0000042000000 | 8.4000000000e-09  4.2000000000e-09  1.4000000000e-09
2000  0.0000147333333 | 1.4733333333e-08  7.3666666667e-09  2.2316266972e-09
4000  0.0000041333333 | 2.0666666667e-09  1.0333333333e-09  2.8687288269e-10
8000  0.0000080000000 | 2.0000000000e-09  1.0000000000e-09  2.5620726228e-10
16000 0.0000151333333 | 1.8916666667e-09  9.4583333333e-10  2.2497772120e-10

Average:
Size(N)     Time(T)            T/NlogN            T/N²               T/N³
1000  0.0003960666667 | 1.3202222222e-07  3.9606666667e-10  3.9606666667e-13
2000  0.0013124666667 | 1.9879653750e-07  3.2811666667e-10  1.6405833333e-13
4000  0.0017096666667 | 1.1865895285e-07  1.0685416667e-10  2.6713541667e-14
8000  0.0068086000000 | 2.1805159575e-07  1.0638437500e-10  1.3298046875e-14
16000 0.0275569333333 | 4.0967154611e-07  1.0764427083e-10  6.7277669271e-15

Worst:
Size(N)     Time(T)            T/NlogN            T/N²               T/N³
1000  0.0007964000000 | 2.6546666667e-07  7.9640000000e-10  7.9640000000e-13
2000  0.0011332666667 | 1.7165349424e-07  2.8331666667e-10  1.4165833333e-13
4000  0.0033753333333 | 2.3426409759e-07  2.1095833333e-10  5.2739583333e-14
8000  0.0137245333333 | 4.3954063893e-07  2.1444583333e-10  2.6805729167e-14
16000 0.0543223333333 | 8.0757586542e-07  2.1219661458e-10  1.3262288411e-14
```

```
Quick Sort:

Best:
Size(N)     Time(T)             T/N                 T/NlogN             T/N²
1000  0.0001440000000 | 1.4400000000e-07  4.8000000000e-08  1.4400000000e-10
2000  0.0001903333333 | 9.5166666667e-08  2.8829385613e-08  4.7583333333e-11
4000  0.0001746000000 | 4.3650000000e-08  1.2118065803e-08  1.0912500000e-11
8000  0.0003748666667 | 4.6858333333e-08  1.2005445298e-08  5.8572916667e-12
16000 0.0008059333333 | 5.0370833333e-08  1.1981302518e-08  3.1481770833e-12

Average:
Size(N)     Time(T)             T/N                 T/NlogN             T/N²
1000  0.0001413333333 | 1.4133333333e-07  4.7111111111e-08  1.4133333333e-10
2000  0.0001549333333 | 7.7466666667e-08  2.3467422825e-08  3.8733333333e-11
4000  0.0001743333333 | 4.3583333333e-08  1.2099557875e-08  1.0895833333e-11
8000  0.0003743333333 | 4.6791666667e-08  1.1988364814e-08  5.8489583333e-12
16000 0.0008056666667 | 5.0354166667e-08  1.1977338153e-08  3.1471354167e-12

Worst:
Size(N)     Time(T)             T/NlogN             T/N²                T/N³
1000  0.0014226666667 | 4.7422222222e-07  1.4226666667e-09  1.4226666667e-12
2000  0.0017459333333 | 2.6445281255e-07  4.3648333333e-10  2.1824166667e-13
4000  0.0056886000000 | 3.9481574527e-07  3.5553750000e-10  8.8884375000e-14
8000  0.0238275333333 | 7.6309838528e-07  3.7230520833e-10  4.6538151042e-14
16000 0.0928808666667 | 1.3808012594e-06  3.6281588542e-10  2.2675992839e-14
```

```
Linear Search:

Best:
Size(N)     Time(T)             T/1                 T/logN              T/(N/2)
10000  0.0000018160000 |1.8160000000e-06  4.5400000000e-07  3.6320000000e-10
20000  0.0000005350000 |5.3500000000e-07  1.2438880932e-07  5.3500000000e-11
40000  0.0000005440000 |5.4400000000e-07  1.1820793319e-07  2.7200000000e-11
80000  0.0000006186000 |6.1860000000e-07  1.2616533689e-07  1.5465000000e-11
160000 0.0000006484000 |6.4840000000e-07  1.2459359165e-07  8.1050000000e-12

Average:
Size(N)     Time(T)             T/logN              T/(N/2)             T/N
10000  0.0000035486000 |8.8715000000e-07  7.0972000000e-10  3.5486000000e-10
20000  0.0000064292000 |1.4948047343e-06  6.4292000000e-10  3.2146000000e-10
40000  0.0000119074000 |2.5874065141e-06  5.9537000000e-10  2.9768500000e-10
80000  0.0000237766000 |4.8493093260e-06  5.9441500000e-10  2.9720750000e-10
160000 0.0000471272000 |9.0557481682e-06  5.8909000000e-10  2.9454500000e-10

Worst:
Size(N)     Time(T)             T/(N/2)             T/N                 T/NlogN
10000  0.0000037972000 |7.5944000000e-10  3.7972000000e-10  9.4930000000e-11
20000  0.0000062288000 |6.2288000000e-10  3.1144000000e-10  7.2410562194e-11
40000  0.0000119214000 |5.9607000000e-10  2.9803500000e-10  6.4761215752e-11
80000  0.0000238834000 |5.9708500000e-10  2.9854250000e-10  6.0888643854e-11
160000 0.0000473324000 |5.9165500000e-10  2.9582750000e-10  5.6844865412e-11
```

```
Binary Search:

Best:
Size(N)     Time(T)            T/1                T/logN             T/(N/2)
10000  0.0000016976000 |1.6976000000e-06  4.2440000000e-07  3.3952000000e-10
20000  0.0000005644000 |5.6440000000e-07  1.3122438127e-07  5.6440000000e-11
40000  0.0000005620000 |5.6200000000e-07  1.2211922510e-07  2.8100000000e-11
80000  0.0000006560000 |6.5600000000e-07  1.3379317976e-07  1.6400000000e-11
160000 0.0000007440000 |7.4400000000e-07  1.4296365235e-07  9.3000000000e-12

Average:
Size(N)     Time(T)            T/1                T/logN             T/(N/2)
10000  0.0000009594000 |9.5940000000e-07  2.3985000000e-07  1.9188000000e-10
20000  0.0000005556000 |5.5560000000e-07  1.2917835973e-07  5.5560000000e-11
40000  0.0000005602000 |5.6020000000e-07  1.2172809591e-07  2.8010000000e-11
80000  0.0000006424000 |6.4240000000e-07  1.3101941871e-07  1.6060000000e-11
160000 0.0000006986000 |6.9860000000e-07  1.3423979507e-07  8.7325000000e-12

Worst:
Size(N)     Time(T)            T/1                T/logN             T/(N/2)
10000  0.0000005742000 |5.7420000000e-07  1.4355000000e-07  1.1484000000e-10
20000  0.0000005554000 |5.5540000000e-07  1.2913185924e-07  5.5540000000e-11
40000  0.0000005984000 |5.9840000000e-07  1.3002872651e-07  2.9920000000e-11
80000  0.0000006372000 |6.3720000000e-07  1.2995886302e-07  1.5930000000e-11
160000 0.0000006710000 |6.7100000000e-07  1.2893630474e-07  8.3875000000e-12
```