Alvin Larsson boalvin@hotmail.se

Kasper Barzenji kasperbarzenji@hotmail.se

Introduction

A local it-consulting company needs assistance with a client that has recently launched the beta-version of a new service. The program has in several instances encountered severe performance issues, largely caused by ineffective sorting and searching algorithms for large workloads. Most likely, these have to be replaced to mitigate afromentioned scaling issues.

The purpose of this lab is to document the best, worst and average performane of the two main algorithms currently employed(linear search and bubble sort), and several alternative algorithms(binary search, insertion sort, and quicksort).

In order to evaluate and demonstrate these different approaches a specialized graphical user interface is utilized. This interface needs to rigidly conform to common design principles, mainly a clear distinction between front-end user interaction and back-end computations.

Background

The searching algorithm currently employed by the service, bubble sort, effectively sorts elements by comparing each element *n* in numerical array *arr* with element *n+1*, ie: the next element in *arr*. If the next element *n+1* contains a lesser value than *n*, the previous element, then the two elements are switched. The higher value in *n* is thereby "bubbled" forward through the sequence to its sorted position. Since this procedure may only move elements one position or index at a time, it needs to be repeatedly executed in *arr* in proportion to the the length of the array.

The best case for bubble sort is defined as an array that is already sorted in ascending order , because no elements have to be switched. The worst case is defined as a situation in which all elements *n* are in an unsorted state, which could imply that they are "sorted" by value in descending/reverse order. Thus all elements need to be switched by the algorithm. The average case is equivalent to the average number of operations that need to be performed on a given array. This can be emulated by randomly distributing values in an array before passing it to the algorithm, then proceeding to repeat the process a set amount of times. The average case can then be determined by observing the amount of operations performed on the array in each iteration.

An alternative sorting algorithm, insertion sort, works by selecting each element *n* in sequential order, for each element *n* the element left of *n* is compared to the value of *n*, if n is lesser than the given element then the two elements switch position. This procedure is repeated until element *n* is placed in its correct position.

The best case for the insertion sort algorithm is identical to the best case of bubble sort. Since all elements are already sorted, no switching of elements need to take place. Likewise, the worst case scenario is one in which all elements occupy an unsorted position, therefore the maximum amount of operations has to be performed on the array. The average case can also be determined in the same way, by randomizing the value of all elements in the array and performing the necessary operations, and subsequently repeating the process a set number of times in order to obtain an average.

The last sorting algorithm, quicksort, is an complex procedure that consists of two main components. First, an element *n* in a given array is selected, based on variable criteria. This element is referred to as a *pivot*, and is subsequently moved to either the left or right-most position in the array. The first element to the left of the pivot in the array that is *larger* than the pivot is referred to as itemFromLeft, the first element to the right of the pivot that is *smaller* than the pivot is known as itemFromRight. These two elements are then swapped with eachother and the procedure is repeated until the index of itemFromLeft exceeds that of itemFromRight. Lastly, itemFromLeft is swapped with the pivot. From this point all elements to the left of the pivot are less than the pivot, and all elements to the right are higher then the pivot. The array is now divided along two partitions, with each partition consisting of elements to the right and left of the pivot. The entire process is then recursively applied to each of these partitions, who are then divided into even more partitions, until the list is considered sorted.

The best case, as in bubble and insert sort, is defined as an array that is already sorted. The same applies to the worst and average cases respectively.

Linear search is the main searching algorithm employed by the system. It functions by searching for a certain element *n* in array *arr* starting from the first index and ending at the last index, this is described as being in *linear* fashion. This process continues until the element is found. The best case can therefore be obtained if element *n* is present in the very first index of the array. Since no elements have to been searched further. Likewise, the worst case can be determined in the opposite context, in which the element occupies the last position, or index, of the array. This case is therefore variable in proportion in the length of the array. The average case can be determined by randomizing values in an array, then proceeding to apply the algorithm. Repeating this process several times. By analyzing each iteration the average case in the array can be determined.

Binary search is an alternative searching algorithm that works by taking the left-most and right-most indexes of a sorted array *arr,* then calculating the average of these, rounding down if necessary. The result is known as *mid*, the equivalent index being the middle value of the array. If element *n* at index *mid* is higher than target element *e,* the element is being searched for, then element at index *mid* along with the elements to the right of mid are eliminated from the array. If element *n* at index *mid* is less than the target element, then the element at index mid along with elements to the left of mid are eliminated. This procedure continues until element at index *mid* is equivalent to the target element.

The best case for this algorithm is if the target element is present at index mid, which is equivalent to the middle value of the sorted array, the first element to be searched. The worst case

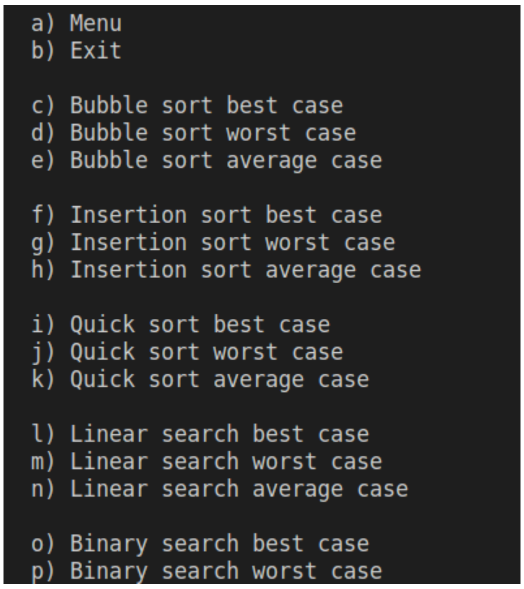
Design/Implementation

The interface which is utilized to both demonstrate and emulate these different sorting and searching algorithms is designed with intitution in mind, separating user interaction from the backend.

The program allows the user to methodically simulate the main and alternative searching and sorting algorithms, by computing the necessary information and presenting it in a clear and consice data table in the interface. This makes it easier to compare different results and visualizes important information in an intuitive format. This also allows the user to identify time complexity for different algorithms with large datasets.

The program is compromised of five c-files. The user interface, or backend, is managed by IO.c and UI.c. The former contains functions that facilitate user input in the terminal, which handles input and output. The latter manages the user interface through different functions. The user interface is initialized from the main class, main.c, by calling ui_run() from UI.c. The run function is itself composed of other functions. UI_menu() and UI_menu_options() are utilized to display a set of options in a menu-format, while ui_get_choice() is used to read user input. This is connected to a switch loop reads the input and starts the benchmarking procedure for the respective algorithm, while also selecting the appropriate case.

The menu is organized in alphabetic order, sorted by algorithm and their respective cases. This allows the user to easily discern different options while keeping input length to a minimum.


```
a) Menu
b) Exit

c) Bubble sort best case
d) Bubble sort worst case
e) Bubble sort average case

f) Insertion sort best case
g) Insertion sort worst case
h) Insertion sort average case

i) Quick sort best case
j) Quick sort worst case
k) Quick sort average case

l) Linear search best case
m) Linear search worst case
n) Linear search average case

o) Binary search best case
p) Binary search worst case
```

The backend is composed of analyze.c and algorithm.c. The latter contains the implementation of the different sort and searching algorithms while analyze acts an interface that communicates with the front-end to produce appropriate results by utilizing the benchmarking function. This function takes several arguments, including an enum which designates the algorithm to test, a enum which designates the case to test(best, worst, average), a special array called result_t into which the results of the procudure are written, and an integer which informs the function how many different array sizes to test. The default array size is 512, informed by a constant. For each iteration of the procedure the size of the array is doubled. This array is filled with different values depending on the case and algorithm currently employed by the function. The dataset is tested by using the function calculateResult(), which uses the algorithm to sort or search the array, measures execution speed in seconds, then writes the result into result_t. The time it takes to execute the algorithm is measured using the POSIX clock_gettime function. The resulting data is then collected by the calling function by reading the result_t array, then formated into a data table displayed in the interface.

Bubble sort is implemented using a simple nested for-loop. Where the outer loop traverses each item in the array, while an inner loop executes the same procudure for each iteration, swapping adjecent onsorted items along the way. The outer loop breaks after a single iteration if the array is already sorted.

Insert sort is implemented using an inner while loop and an outer for loop. The inner loop checks if an item at index of the current iteration is less than its adjecent item to the left, if true, the item adjcent to the left and the current item are switched. If element is already sorted then this procedure is immediately evaluated to be false, the sorting process therefore becomes much quicker.

The main sorting algorithm, linear search, is implemented using a simple for loop that checks if target value exists at current index in the array. Returns false if target value is not found.

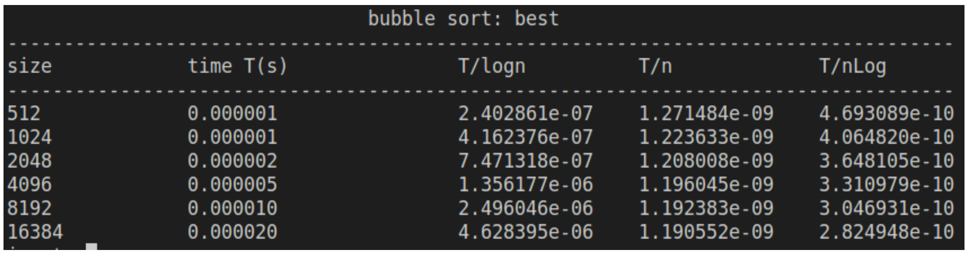
An alternative sorting algorithm, binary search, is implemented by sorting an array in ascending order, then initializing variables *left* and *mid* to 0, while the variable *right* is equal to the size of the array. Variable *mid* is then set to the middle index of the array by dividing *left + right* with 2. If the element at this index equals the target value, the function returns true, since the element has been found. If the element is **less** than the target value, then the left half of the array(from mid-1) is eliminated and *left* is set to the new "left" boundary of the array, at *mid+1*. The procedure, beginning with initializing mid to the new middle element of the array, is then repeated for the left half of the array, as long as the left and right-most boundary are not equal, which is controlled by a while-loop. Likewise, if the middle element of the array is **higher** than the target value, then the right half of the array is eliminated by changing the right-most boundary of the array to the boundary of the left half, which is done by changing the corresponding variable *right* to the new boundary, at *mid-1.* This process is then repeated until the element is found or the left and right-most boundaries are equal.

//qsort

Results + Analysis

The respective time-complexity for each algorithm, or how the time taken the run the algorithm increases in relation to the size of the dataset being handled, can be determined by analyzing the data produced by simulations in the program. This data is structured into a table, in which the numerical properties of certain algorithms can be observed, including the algorithm currently used, the case(best, worst, average), the size of the array being tested, the run-time of the algorithm, and the various calculations concerning the relationship between size and run-time.

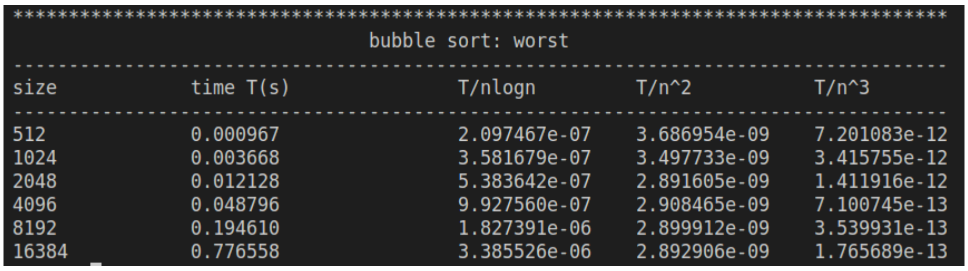
Beginning with the first sorting algorithm, bubble sort, this process can be demonstrated.



```
                                bubble sort: best
--------------------------------------------------------------------------------------
size            time T(s)               T/logn          T/n             T/nLog
--------------------------------------------------------------------------------------
512             0.000001                2.402861e-07    1.271484e-09    4.693089e-10
1024            0.000001                4.162376e-07    1.223633e-09    4.064820e-10
2048            0.000002                7.471318e-07    1.208008e-09    3.648105e-10
4096            0.000005                1.356177e-06    1.196045e-09    3.310979e-10
8192            0.000010                2.496046e-06    1.192383e-09    3.046931e-10
16384           0.000020                4.628395e-06    1.190552e-09    2.824948e-10
```

In this simulation, the first array of 512 elements is sorted in approximately 0.000001 seconds, with each subsequent array doubling in size. The input array already being sorted in ascending order, the best case.

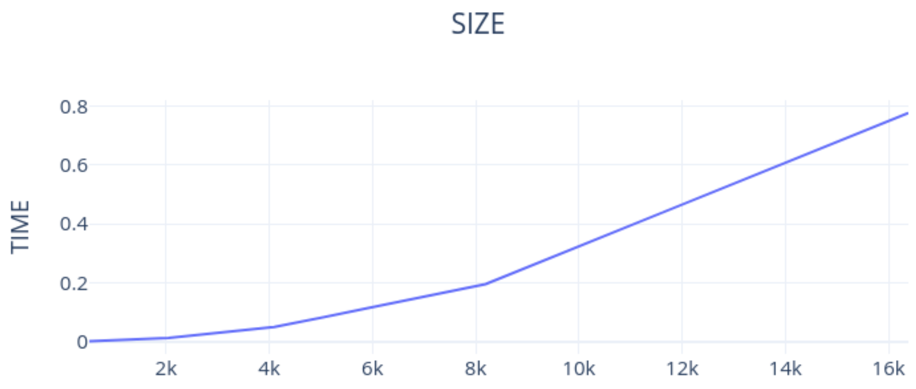
On a technical level this algorithm is implemented using an inner and outer for-loop, each iterating through the length of the array. If no unsorted elements are found during the inner-loop, the outer loop breaks and the function terminates, hence the total number of iterations and comparisons is equal to the inner for-loop, which is dependent on array size. Execution time therefore grows in linear proportion to the size of the array, the time complexity in turn being described as O(n). In the data-table this relationship is proven by observing that run-time T divided by size N converges to a constant, which indicates a reliable correlation.



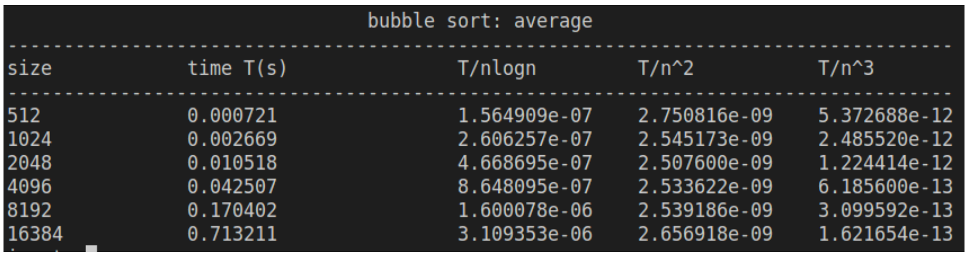
```
*********************************************************************************
                              bubble sort: worst
---------------------------------------------------------------------------------
size            time T(s)              T/nlogn         T/n^2           T/n^3
---------------------------------------------------------------------------------
512             0.000967               2.097467e-07    3.686954e-09    7.201083e-12
1024            0.003668               3.581679e-07    3.497733e-09    3.415755e-12
2048            0.012128               5.383642e-07    2.891605e-09    1.411916e-12
4096            0.048796               9.927560e-07    2.908465e-09    7.100745e-13
8192            0.194610               1.827391e-06    2.899912e-09    3.539931e-13
16384           0.776558               3.385526e-06    2.892906e-09    1.765689e-13
```

In the worst case, the outer and inner for-loop need to run through their whole length, since every element in the array is unsorted. Hence, the number of comparisons and iterations is equal to the size of the array to the power of two, the time complexity therefore being O(n^2). Again, this pattern can be observed in the table, with T/n^2 converging to constant.

This relationship can also be visualized by using a simple graph.



Likewise, the time complexity for the average case, which is an array in random order, can be determined by observing the data.



```
                              bubble sort: average
---------------------------------------------------------------------------------
size            time T(s)              T/nlogn         T/n^2           T/n^3
---------------------------------------------------------------------------------
512             0.000721               1.564909e-07    2.750816e-09    5.372688e-12
1024            0.002669               2.606257e-07    2.545173e-09    2.485520e-12
2048            0.010518               4.668695e-07    2.507600e-09    1.224414e-12
4096            0.042507               8.648095e-07    2.533622e-09    6.185600e-13
8192            0.170402               1.600078e-06    2.539186e-09    3.099592e-13
16384           0.713211               3.109353e-06    2.656918e-09    1.621654e-13
```

This the same time complexity as the worst case, O(n^2). From a mathematical perspective this can be explained using probabilities.

Assuming an randomly ordered array of given size n, the probability of that array already being sorted is equal to the probability of a single element being in its sorted position multiplied by itself in proportion to the length of the array.

In other words, the probability of an array of size 512 already being sorted after randomization:

p = (probability of an item occupying its sorted position)

n = (the size of the array)

P = 1/n = 1/512 = 0.00195 = 0.01%

P^n = 0.00195^512 < 0.1e20 (almost infinitely small)

An alternative method:

Given 512 elements with each being a value between 0 and 512, duplicate values being allowed, the number of permutations can be determined:

512 * 512 * 512 * 512 …

Or

512^512 = (almost infinite)

Since only one of these permutations is perfectly sorted, the likelyhood of the algorithm minimizing its run-time(by breaking out of the outer for-loop) is 1/(512^512), which is next to impossible.

Likewise, the probability of *half* of the array being in a sorted state, or rather the algorithm effectively cutting its run-time in half, is vanishingly small.

P = 1/n = 1/512 = 0.00195 = 0.01%

P^(n/2) = 0.00195^256 < 0.1e20 (almost infinitely small)

Probability of almost a *tenth* of the array being in a sorted state:

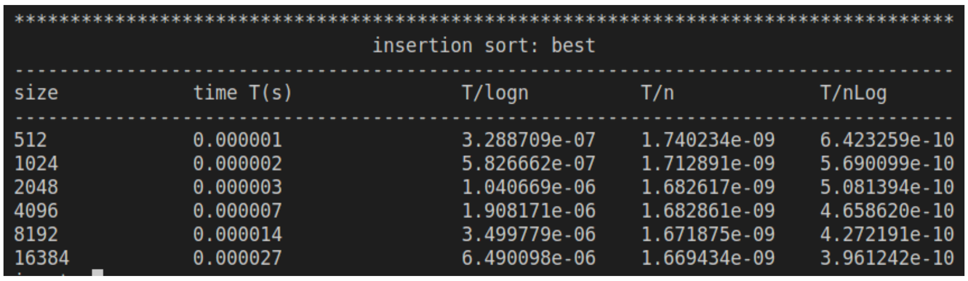
P = 1/n = 1/512 = 0.00195 = 0.01%

P^(n/10) = 0.00195^51 = 6.191 * 10^(-139)

And so on...

In a relatively large dataset, the algorithms average run-time, and there it’s time complexity, is therefore likely going to be similiar to it’s worst.

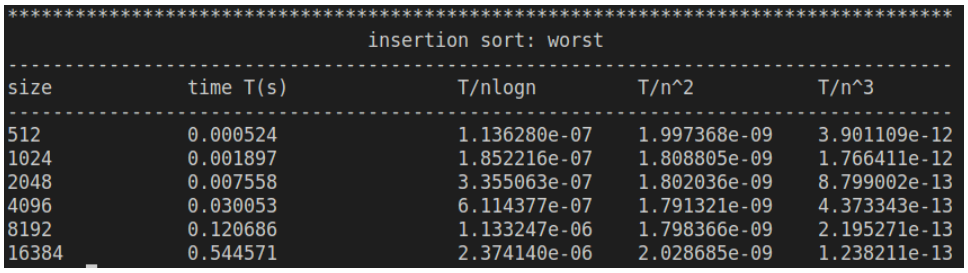
The second sorting algorithm, insertion sort, following the implementation described in this document, with the most optimal input.



```
**********************************************************************************
                              insertion sort: best
----------------------------------------------------------------------------------
size           time T(s)              T/logn         T/n            T/nLog
----------------------------------------------------------------------------------
512            0.000001               3.288709e-07   1.740234e-09   6.423259e-10
1024           0.000002               5.826662e-07   1.712891e-09   5.690099e-10
2048           0.000003               1.040669e-06   1.682617e-09   5.081394e-10
4096           0.000007               1.908171e-06   1.682861e-09   4.658620e-10
8192           0.000014               3.499779e-06   1.671875e-09   4.272191e-10
16384          0.000027               6.490098e-06   1.669434e-09   3.961242e-10
```

Since no elements to the left of an iteration need to be compared further, the outer for-loop continues in its entire length, which is equal to the size of the array. The best case is therefore T/n or O(n).

The same algorithm using the worst possible input, or worst case.



```
********************************************************************************
                              insertion sort: worst
--------------------------------------------------------------------------------
size            time T(s)               T/nlogn         T/n^2           T/n^3
--------------------------------------------------------------------------------
512             0.000524                1.136280e-07    1.997368e-09    3.901109e-12
1024            0.001897                1.852216e-07    1.808805e-09    1.766411e-12
2048            0.007558                3.355063e-07    1.802036e-09    8.799002e-13
4096            0.030053                6.114377e-07    1.791321e-09    4.373343e-13
8192            0.120686                1.133247e-06    1.798366e-09    2.195271e-13
16384           0.544571                2.374140e-06    2.028685e-09    1.238211e-13
```

Since all elements in the array are unsorted, the maximum amount of comparisons are performed.

The first element in the array compares itself to the elements left of it, which are none. The second element does the same, with one element being to the left of it, the third element, with two elements to the left, and so on...

This can be described mathematically:

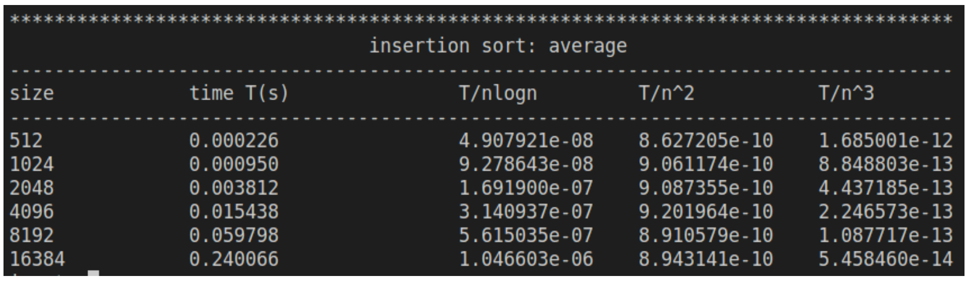
1+2+3+4+(N-1)…

The total amount of comparisons, assuming the sequence, can therefore be calculated using gauss's trick, N(N+1) / 2. But since the sequence begins at 0, with zero comparisons, the formula needs to be adjusted accordingly, N(N-1) / 2.

Since time complexity is defined in terms of an *upper bound* or an asymptotic function, constants in the expression are dropped. This means that the function is redefined as N(N-1). Simplifying the expression, N^2 – N.

In big-O notation the lower term is dropped in favour of the larger term, therefore the expression can be redefined to N^2. The time complexity for the worst case is therefore O(N^2).

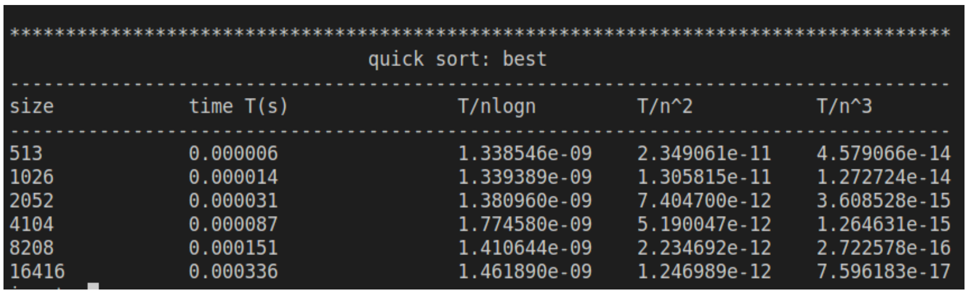
Using the average case, or a randomly assigned array.



```
********************************************************************************
                              insertion sort: average
--------------------------------------------------------------------------------
size            time T(s)               T/nlogn         T/n^2           T/n^3
--------------------------------------------------------------------------------
512             0.000226                4.907921e-08    8.627205e-10    1.685001e-12
1024            0.000950                9.278643e-08    9.061174e-10    8.848803e-13
2048            0.003812                1.691900e-07    9.087355e-10    4.437185e-13
4096            0.015438                3.140937e-07    9.201964e-10    2.246573e-13
8192            0.059798                5.615035e-07    8.910579e-10    1.087717e-13
16384           0.240066                1.046603e-06    8.943141e-10    5.458460e-14
```

As with bubble sort, the average case is roughly equal to the worst case. The same mathematical statements apply here aswell. In an relatively large dataset, random numerical permutations which prove beneficial to the algorithm in terms of performance(values already being in a sorted position) become vanishly rare. Therefore the algorithms average performance on larger arrays is similiar to it's worst, hence the time complexity being roughly equal.

The last sorting algorithm, quicksort, with the most optimal input.



```
*********************************************************************************
                              quick sort: best
---------------------------------------------------------------------------------
size            time T(s)               T/nlogn         T/n^2           T/n^3
---------------------------------------------------------------------------------
513             0.000006                1.338546e-09    2.349061e-11    4.579066e-14
1026            0.000014                1.339389e-09    1.305815e-11    1.272724e-14
2052            0.000031                1.380960e-09    7.404700e-12    3.608528e-15
4104            0.000087                1.774580e-09    5.190047e-12    1.264631e-15
8208            0.000151                1.410644e-09    2.234692e-12    2.722578e-16
16416           0.000336                1.461890e-09    1.246989e-12    7.596183e-17
```

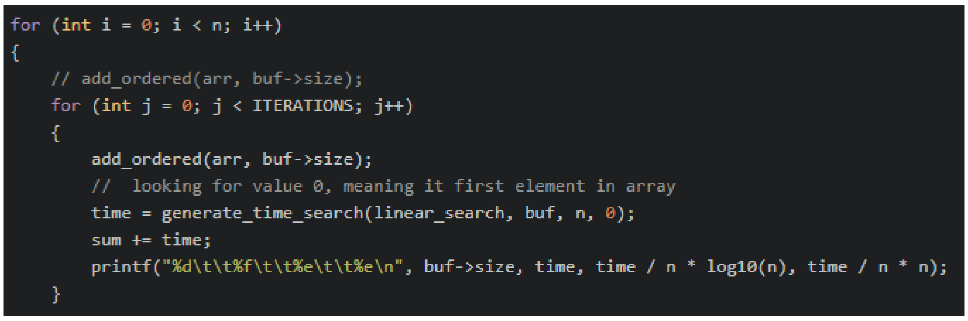
Assuming the technical implementation as described, this algorithm repeatedly divides an array into different partitions by placing the chosen pivot element in it's sorted position. And for each "division", or partition split, the algorithm performs a number of comparisons. Thus the best case is when the least amount of splits are performed on the array, which is when the array is perfectly split in the middle, and the left and right partitions are already fully sorted.

In order to achieve this behaviour, the algorithm needs to always pick the middle element as the pivot point, by combining this with an ascending already sorted array, of odd length, one can mathematically define how many times the array will be split using a formula.

N(size) / 2^x, x defining how many times to split by two. Splits stop when there is only one element remaining, or

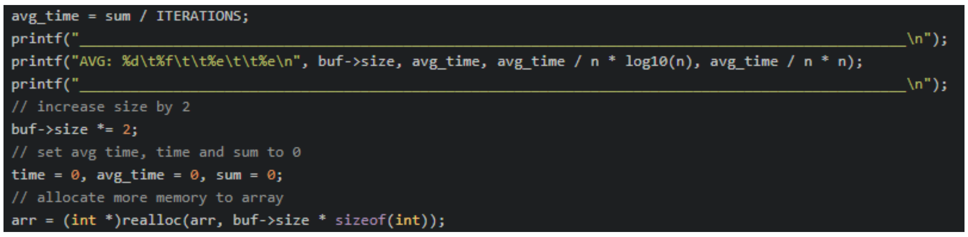
I also took the time to benchmark both linear and binary search, but before I show the numbers I would like to go through how I got the numbers displayed. I made a function called `generate_time_search` that returns a time of type double data type, the function parameters takes in a function which is going to be either linear_search function or binary_search, and those in return also have three parameters (const int *arr, int n, int v). With use of the `clock_gettime` I am able to time the function and in return get the time back in nanoseconds.

For both best cases, I am filling up an array of buf->size n times (depending of how many rows) and after each iteration I am doubling the array size. As dispalyed here:



```c
for (int i = 0; i < n; i++)
{
    // add_ordered(arr, buf->size);
    for (int j = 0; j < ITERATIONS; j++)
    {
        add_ordered(arr, buf->size);
        //  looking for value 0, meaning it first element in array
        time = generate_time_search(linear_search, buf, n, 0);
        sum += time;
        printf("%d\t\t%f\t\t%e\t\t%e\n", buf->size, time, time / n * log10(n), time / n * n);
    }
```

I am then depending of how many iterations we will run which is defaulted to 4, adding the time into sum variable so that I then can get the average depending on how many iterations we have run. As shown in this picture here:



```c
avg_time = sum / ITERATIONS;
printf("________________________________________________________________________________\n");
printf("AVG: %d\t%f\t\t%e\t\t%e\n", buf->size, avg_time, avg_time / n * log10(n), avg_time / n * n);
printf("________________________________________________________________________________\n");
// increase size by 2
buf->size *= 2;
// set avg time, time and sum to 0
time = 0, avg_time = 0, sum = 0;
// allocate more memory to array
arr = (int *)realloc(arr, buf->size * sizeof(int));
```

Which avg_time is the sum divided by number of iterations.

For worst case and also average case, these will be similar like best case, only difference is for the worst case, we reverse the order on the array, so instead of filling the array starting from index 0 and adding by 1, so 0, 1, 2, 3 etc we reverse the order and start from the end and then work our way back to 0. For the average case, we emulate average by filling the array with random numbers, likely how it would work in practice.

For binary search however, the best case scenario is if the key we are looking for is in the middle of the array, worst case is when the key we are looking for is the first index or last index in the array. I simulated the average case by filling the array with random numbers and the key that we are searching for is a random number depending on array size.

Discussion/Conclusion