Looking for World's best universal sorting algorithm

And also few points on writing fast and efficient code

By Atanas Rusev

www.atanasrusev.com

First Draft, 08 February 2021, All Rights Reserved.

Hello Everyone,

Before we start – few years ago my target was initially to understand in complete detail how some sorting algorithms work. Once I got deeper I easily noticed a potential point of improvement in particular for the Quicksort algorithm. Of course hundreds of other people did notice it as well – it is after all one of the most famous around. After this the task of being faster and faster obsessed me a bit.

Here I will present basics of sorting, my endeavor and research, but also what points one shall consider when thinking on a universal generic sorting algorithm. Some part of the text is related in general to the point on how one can write faster and more efficient code.

We will shortly look into points like CPU optimizations and architecture, speed of basic operations, stability, O() notation and deviations from it, few famous algorithms, and others. Many of the subjects are a lot deeper than the short description given here. As I cannot cover them all in detail – should you need to – you will have to search and read on them more by yourself. Some links will be given here by me as well.

In addition I have developed a bigger test framework written in C++, which works so far only for int numbers. It also includes a set of algorithms uploaded on GitHub, so that we can compare results and make real conclusions. At the end I really want either to write the best current universal algorithm, or define one already existing. Once this is clear I will be really happy if it becomes the de-facto standard for GCC, Clang, MSVC, XCode and any other big and small compiler around.

Let's look to the plan, as this is the base of all structured activity in life:

Contents

1. Basics of sorting

So – let's start with what sorting is – considering a bunch of data of the same type (e.g. numbers, but can be as well strings, complex objects, pieces of memory, etc.) – when we want to order them by some quality we call this sorting.

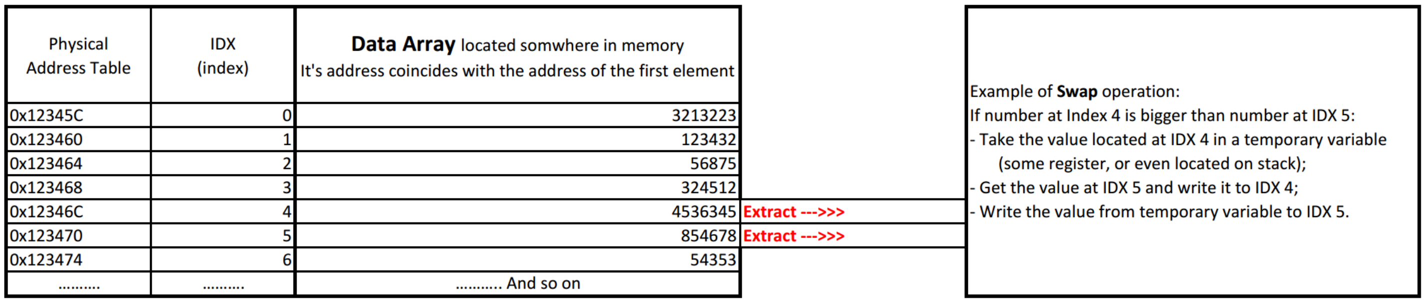
Sorting generally consists of two operations – comparisons and swaps. We also consider the intermediate data – when we compare we do this between two particular values, for which we have to extract the two pieces from the common container (e.g. RAM) and perform the comparison outside of it. The swap is as well a binary operation, which changes two elements located in memory in addresses A and B.

Each operation takes some time for the CPU.

Comparison of numbers is one of the fastest operations for a CPU.

Comparison of strings however – if it is by length – would be slower, as we need to extract first the length of the first string, then the length of the second, and then apply the real numbers comparison. If we compare the alphabetical order of strings – we shall extract from each string the first one or few letters of it, also extract the first one or few letters from the second string, and then apply the operation of comparison of e.g. the ASCII codes of their numbers.

If we compare e.g. the size of objects – for each comparison operation we have to extract this size, and then check it. So one can understand now - it might take sometimes a lot longer, when we don't just compare some integers like 20 and 27, but instead specific quality which at the end translates to a number.



Physical Address Table	IDX (index)	**Data Array** located somwhere in memory It's address coincides with the address of the first element	
0x12345C	0	3213223	
0x123460	1	123432	
0x123464	2	56875	
0x123468	3	324512	
0x12346C	4	4536345	Extract --->>>
0x123470	5	854678	Extract --->>>
0x123474	6	54353	
....	....	.... And so on	

Example of **Swap** operation:
If number at Index 4 is bigger than number at IDX 5:
- Take the value located at IDX 4 in a temporary variable (some register, or even located on stack);
- Get the value at IDX 5 and write it to IDX 4;
- Write the value from temporary variable to IDX 5.

Swap operations by the original understanding mean e.g. we have one address in the memory with the number 20, another with the number 27. We take the first in a temporary buffer, then write the second in the address of the first, finally write the temporary saved number 20 in the address of the second. If we compare this series of operations with the simple comparison of the two numbers – usually it takes a bit more time then comparison.

If we have to swap big objects, each occupying e.g. 50 bytes in the memory – then we have a problem – we have to extract 50 bytes in a temporary buffer, then copy 50 bytes from address B to address A, then copy the temporary data to address B. This would take a lot longer, thus many algorithms might work here with a linked list, and would simply change the order of elements in it. This is so because a linked list value re-order requires to change only 3 or 4 address values.

For simplest example let's consider the bubble sort – it is one of the simplest and also one of the slowest famous sorting algorithms in the world. Let's consider we have the 10 number array 3,9,8,1,5,4,7,6,2,0 and we want to sort it in normal rising order.

Then only for the first number – i.e. to put the 0 in the beginning we will do 9 swaps and 9 comparisons. At each step we check whether it is smaller than the previous number – if yes – we swap. Just like the bubbles in water go in upward direction.

Operation	Numbers array									
1	3	9	8	1	5	4	7	6	2	**0**
2	3	9	8	1	5	4	7	6	**0**	2
3	3	9	8	1	5	4	7	**0**	6	2
4	3	9	8	1	5	4	**0**	7	6	2
5	3	9	8	1	5	**0**	4	7	6	2
6	3	9	8	1	**0**	5	4	7	6	2
7	3	9	8	**0**	1	5	4	7	6	2
8	3	9	**0**	8	1	5	4	7	6	2
9	3	**0**	9	8	1	5	4	7	6	2
0	**0**	3	9	8	1	5	4	7	6	2

Then we have to do the same for the second number – it will need 8 comparisons and 8 swaps. If we don't do some optimization – 2 will also be compared with 0, but no swap will take place. Then we go on with 6 and so on. The worst case for this algorithm is completely reversed order of the numbers. It would require for 10 numbers total 9+8+7+6+5+4+3+2+1 swaps and the same number of comparisons, totaling at 45 swaps and 45 comparisons. Here is a potential implementation which presents this kind of slow algorithm:

```
void bubble(int* arr, unsigned int size) {
    for (unsigned int ext = size - 1; ext != 0; ext--) {
        for (unsigned int idx = size - 1; idx != 0; idx--) {
            if (arr[idx - 1] > arr[idx])
                swap(arr[idx - 1], arr[idx]);
            printArray(arr, size);
        }
    }
}
```

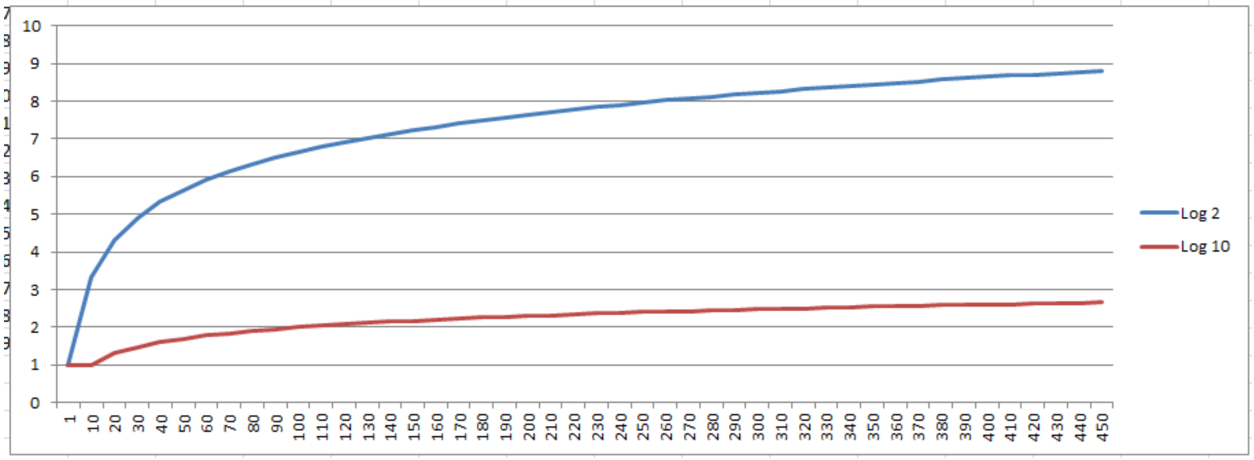
In order to improve this algorithm – we shall reduce the number of swaps and / or comparisons. We shall also break somehow the extra cycles. The code has in addition a printArray() function which shows how many times the main cycle is executed and prints the intermediate result. For 10 numbers we will have 9*9=81 calls, for 20 – 19x19=361. For more information check e.g. this Wikipedia page.

2. An algorithm complexity and the O notation

The purpose of the O(n) notation is to define (based on analysis) how fast an algorithm is. It does so by assessing the number of operations that are necessary in dependence from the input number of elements to be processed, and considers the internal cycles. It is simple mathematics and hence the single best way to define in general terms how fast an algorithm is. So – for sorting – if we have e.g. 100 elements, let's consider 1 is one basic operation unit, then the meaning of O() for different complexities will be:

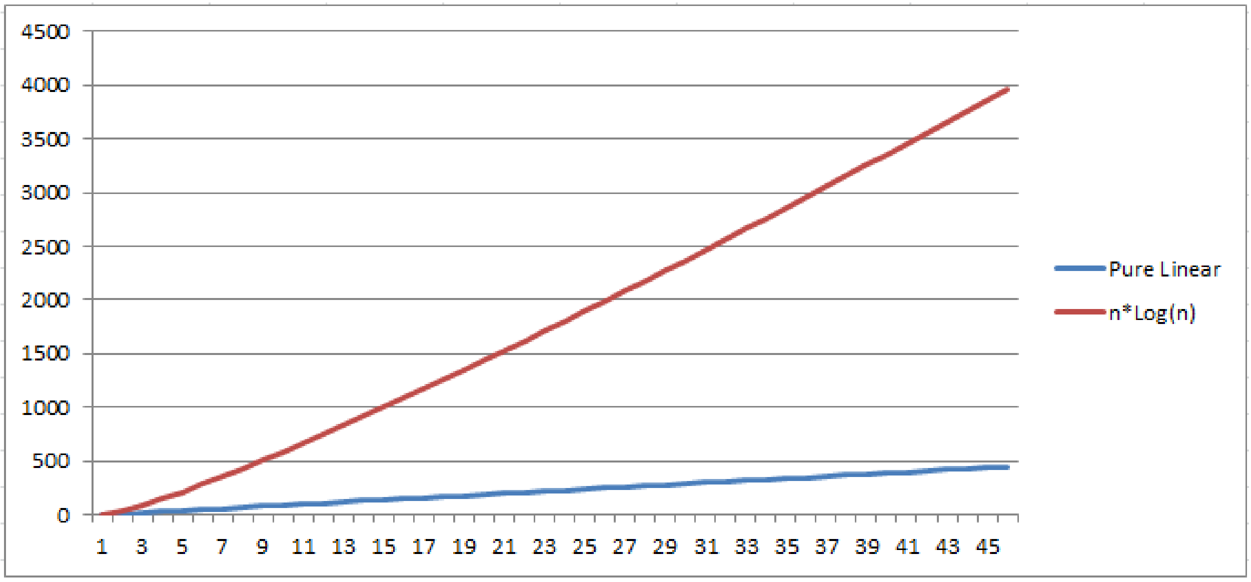
O(1) – Constant time - whatever the number of input elements – we need always the same amount of time to complete the task and let's call it for convenience one basic cycle. This is valid only for some specific HW piped operations (for example the way how a basic Multiplier works inside a CPU ALU (Arithmetical Logical Unit), or FPU, etc.). In other words this is never achieved for complex algorithms, as the CPU works sequentially.

O(log *n*) – the dependence is logarithmic – with the rise of amount of elements in input the required time rises like the log function. Like shown below – on the horizontal axis we have number of elements, on the vertical the required amount of basic cycles. So for 450 if we have Log2 – we need 8.813 basic cycles. This means – the required number of cycles rises quite slower than the number of input elements – this is extremely good efficiency. For 64000 elements Log2 gives ~16 cycles required. But this is as well hardly possible for complex algorithms. Log10 is given only to show you what it means as a mathematical dependence, practically we care about $\log_2(x)$



O(n) = O(100) - we need constant amount of time for each element sorting, each time we add more to the size of the task the time rises in linear fashion. Say we need 1 basic cycle for one element, then for 450 elements we need 450 cycles. For 64000 we need 64000 cycles. This is really good, but also not possible for a sorting algorithm.

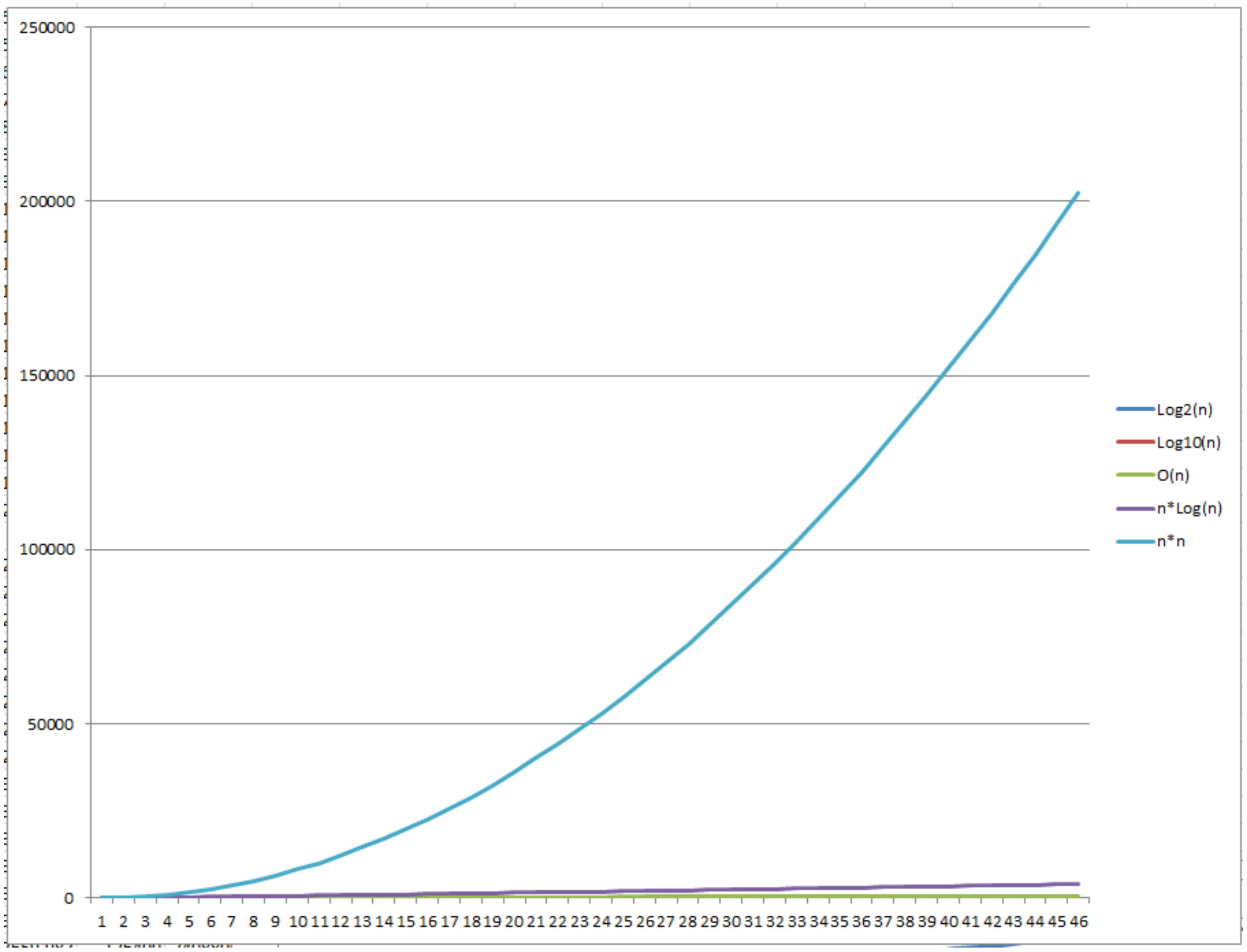
O(*n* log *n*) – this is a multiplication of the previous two. **This is as well quite good in relation to sorting – many good universal algorithms achieve it in the average case, but also in the worst case scenario!** So if the Log is Log 2, and we have 1 cycle for the linear part – for 450 elements we need 3966 cycles; for 64000 elements we need 1021810 basic cycles. This and previous algorithm together look like this on a graph:



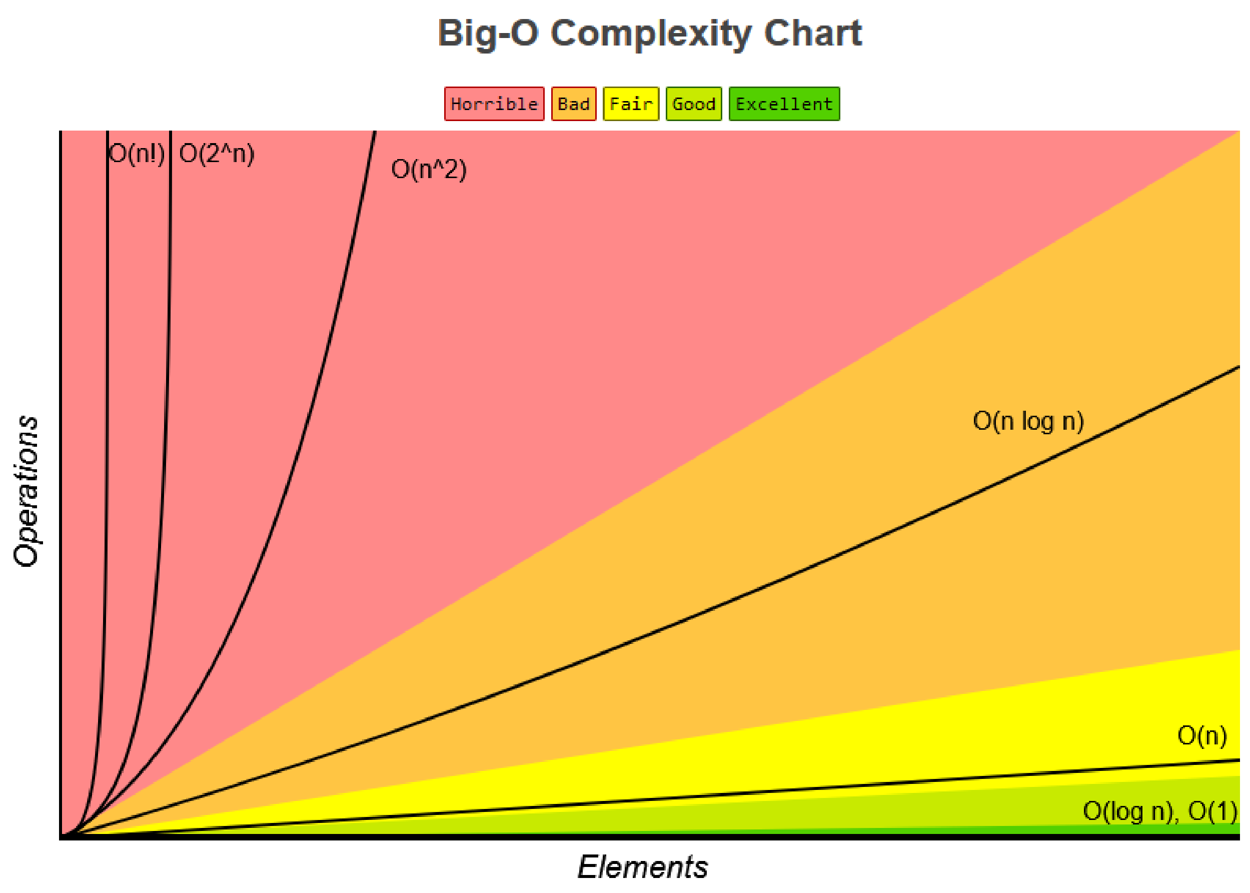
O(n^k) – the problem comes if we have exponential characteristic of the complexity of the algorithm. Here n is again the number of input elements, and k is the power. For k=2 and 450 elements – we need 450*450 = 202,500 cycles. For 64000 and k=2 – we need 4,096,000,000 cycles (4 billion). The previous mathematical dependence reqired only ~1 million. If k goes bigger – we get even worse. The worst case of the bubble sort algorithm is an O(n^2) execution.

Theoretically there are algorithms which can reach O(*n*!) cycles, but such are never used in common practice.

All basic cases presented above and together on a graph look like shown below. As you can see – the blue line rising in the skies is the O(n^2). All others are located at the bottom, hence cannot be seen:



A clearer and a bit manipulated graph will look like this presented at [Big O Cheatsheet](), take a look at the web page. Here is the graph from them:



And here we have a table showing with some hypothetical numbers the results in required cycles for different O() complexities:

Number of elements to sort	Log10(n)	Log2(n)	Linear	n*log(n)	n^2	n^3
1	1	1	1	1	1	1
10	1	3	10	33	100	1000
100	2	7	100	664	10000	1000000
1000	3	10	1000	9966	1000000	1000000000
2000	3	11	2000	21932	4000000	8000000000
5000	4	12	5000	61439	25000000	125000000000
10000	4	13	10000	132877	100000000	1000000000000
64000	5	16	64000	1021810	4096000000	262144000000000

As you've probably already guessed – everybody wants to achieve complexity not bigger than n*Log(n). Many times – when an algorithm reaches only worst case O(n^2), but the common is approximately n*Log(n) – this is considered acceptable.

To conclude take a look at one of my favorite articles on the subject by Shen Huang.

Nowadays however this assessment can be applied only on the theoretical analysis of algorithms, where it is clear what the basic operations are, what time hypothetically takes each of them, and how the total number of operations is calculated for a cycle. As you will learn later – a PC or any other complex computer system holds parts which remove or add productivity to the whole process in different fashion, leading to **significant additions** to the basic O() notation predictions.

To conclude – we need the O() for the basic evaluation of our code and here are few simple examples to illustrate this:

Logarithmic Execution: Binary search – with each step we divide the array two times (you should know it works only for sorted arrays):

```
int binarySearch(int * arr, int leftLimit, int rightLimit, int x)
{
    while (leftLimit <= rightLimit) {
        int idxToCheck = leftLimit + (rightLimit - leftLimit) / 2;

        if (arr[idxToCheck] == x)
            return idxToCheck;

        if (arr[idxToCheck] < x)
            leftLimit = idxToCheck + 1;

        else
            rightLimit = idxToCheck - 1;
    }
    return -1;
}
```

Linear Execution: When we perform some specific function once per element in an array – this is an algorithm with Linear Complexity. We can do whatever we want – it can be a slow or fast operation – but the O() here depends directly from the amount of input elements.

```
void myFunc(int* arr, unsigned int size) {
    for (unsigned int idx = size - 1; idx != 0; idx--) {
        doSomeWork(arr[idx - 1], arr[idx]);
    }
}
```

----

nLogn Execution: When we have a combination of the linear dependence and some logarithmic operation. Let's say we have an array or arrays

```
void myFunc(int** arr, unsigned int size) {
    for (unsigned int idx = size - 1; idx != 0; idx--) {
        binarySearch(array, 0, 19, 5)
    }
}
```

Quadratic Execution: Of course the bubble sort:

```
void bubble(int* arr, unsigned int size) {
    for (unsigned int ext = size - 1; ext != 0; ext--) {
        for (unsigned int idx = size - 1; idx != 0; idx--) {
            if (arr[idx - 1] > arr[idx])
                swap(arr[idx - 1], arr[idx]);
            printArray(arr, size);
        }
    }
}
```

Cubic Execution: If we take the bubble sort and simply add one more **for()** cycle – we have it :).

```
void bubble(int* arr, unsigned int size) {

    for (unsigned int prep = size - 1; prep != 0; prep--) {
       // do some work for preparation of each Bubble sort execution.
       additionalWork(arr, size);
       for (unsigned int ext = size - 1; ext != 0; ext--) {
           for (unsigned int idx = size - 1; idx != 0; idx--) {
               if (arr[idx - 1] > arr[idx])
                   swap(arr[idx - 1], arr[idx]);
               printArray(arr, size);
           }
       }
    }
}
```

3. Stability and cyclic actions count

a. Stability

Now that we have the basics – what is **Stability**? This is the quality of an algorithm to preserve the internal order of similar elements inside the sorted result. So if we have the sequence:

0,5,6,1,2,9,5,6,7,0,1 – sorted it would look like this: 0,1,1,2,5,5,6,6,7,9.

As the numbers 1, 5 and 6 are found twice – if in the result sorted array we keep the leftmost 5 on the left, and the rightmost 5 on the right, and the same for all other pairs, i.e. we keep the original order of their rising positions – the algorithm is **Stable**. When can this be helpful? If we sort a big list of e.g. 50000 names, which were already sorted by last name, and we want them after this sorted by first name – if the algorithm is stable at the end – when we have e.g. 40 George SomeFamily guys – the algorithm will keep the order of the Georges also by last name after a sort by first. So our list will be in general sorted, but the smaller group of elements inside it will preserve their original order. If we know this for sure – then we avoid some or a lot extra work in the future.

b. *Cyclic Actions Count* and how it relates to the current SW/HW environment

As we have now seen – sorting is "a cyclic operation going over an array of input data". With "Cyclic Actions Count" I mean that we have to consider how many times in one iteration of our algorithm we:

- **Compare** (of course)
- **Extract** next value in an array; extract several values and what operations we do with them
- Do "**Long Jumps**" (get some value from position in memory e.g. 5, then get the next from position 55000 – this is quite far away from 5)

- **Call a sub-function**
- **Swap** two numbers close in memory
- **Swap** two numbers **at distant locations in memory**
- Do **assignments**
- Fill **temporary variables**
- Fill **temporary arrays**
- Do **merges**
- Finally – write **too much extra logic in one cycle**

Why not mentioning this in the **O()** notation explanation? From our high-level human perspective sorting is only a series of comparisons and swaps, and the **O()** notation will help us for the theoretical analysis of the algorithm. But in order to write really efficient code one shall consider all sub-operations. Example – even if for 500 numbers we do only 500 comparisons and 200 swaps – if we do this with the price of 100 temporary arrays, 50 temporary variables, extra operations with the temporary data, etc. – our algorithm might get slower even than the bubble sort!

So here are my generic advices to increase the efficiency of your cyclic algorithm related to the upper points:

- **Extract next value in an array; extract several values and what operations we do with them**
  Whenever you extract a value – try to write the code in such a way that you do all possible operations with it and after this write it back to your desired locations. Use temp variables which normally go to the register values – your purpose is to do all the work in the registers.

- **Do "Long Jumps"** (get some value from position in memory e.g. 5, then get the next from position 55000 – this is quite far away from 5)

  NEVER do many long jumps. Avoid it completely. This is like going to the library and trying to read from row A, position 1, then row Z, position 10, then K6, then W10, then B5 – this means we make the CPU run like a mad man around in memory. Your solution – work on smaller chunks – first work on all in section A, then on all in section B, etc.

  Of course sometimes Long Jumps **are** required. That's normal, as long as we don't use them for our essential basic operations.

- **Call a sub-function**

  If you want to process some data cyclically – and you call your cycle e.g. 20000 times - tell me what will happen if 5 times in each cycle your CPU shall go to the library with functions and execute some code located somewhere else? You guessed it – we get a lot slower. If the function is located in other .so or .dll – if the OS/CPU doesn't have some optimizations this can get tragically slow (of course figuratively speaking). So – for simple operations the best would be to use macros, for anything even a bit more complex – inline and / or inline template based functions. Both are normally optimized quite good by the modern compilers, leading to complete elimination of the sub-function call process. And the code still looks structured and normal.

- **Swap two numbers close in memory**

This is the best as long as we don't do it for 200 values x-thousand times. Example for such case – bubble sort is one of the slowest algorithms as it does hundreds of extra unnecessary swaps.

- **Swap two numbers at distant locations in memory**

Just like the long jumps – shall always be avoided.

- **Do assignments**

If you make an assignment – this is writing somewhere – either in memory, or in temporary register variable. The Swap requires an assignment. The compare doesn't (in general and on some CPUs). Writing in registers is super-fast. Writing in memory – if in L1 cache – it is relatively fast, if in L3 cache, i.e. somewhere on a distant location in RAM – it is slow. The point is too generic for some meaningful advice, but you have to consider it when willing to write super-efficient code.

- **Fill temporary variables**

Using temporary variables is the best when we want to use a given value for several operations. This forces the compiler to put the variable in registers, so it is accessible for operations normally at the price of 1 CPU cycle. E.g. some theoretical values for accessing an L1, L2 and L3 caches values can be 10, 35 and 120 cycles. In other words – if we have to work frequently with some values they better be extracted once, processed a lot, and then written back to desired location in memory one time.

- **Fill temporary arrays**

This can be both an improvement, but as well degradation. Consider the amount of time required to extract the data, the amount of data, what work we will do with it. Sometimes this can lead to 100x times improvement, but some other times we might get 0,1 degradation. You have to do like me - profile and measure it – to be sure what will be the result of specific design.

- **Do merges**
  As you will see later dividing a bigger array to smaller parts is essential to achieve significant improvement in speed. The merge of two already sorted pieces is really fast. But – if you have to replace 500 cycles with e.g. 50 merges – you might get no improvement at all. Again you have to profile and measure the results of your design.

- **Finally – write too much extra logic in one cycle**
  What do I mean here? If we have an O(n^2) algorithm, and we want to avoid some part of the extra calls of the main cycle we can put some breaks to cut the extra calls of the algorithm which are not necessary. But if we do so much extra logic (resulting in e.g. 20 times reduction of the cycle calls), but this is with the price of inefficient super-duper-extra logic – we might again not increase the efficiency of our algorithm. So – as you will see later – one shall be careful with such approaches, and again profile and measure.

With all points from above – here come as well the compiler of choice and the optimization level. Whatever our language is – I agree 100% with Linus Torvalds – C is the single language closest to the machine language (it was designed with this purpose); its code instructions correspond directly to the basic instructions of CPUs. As it is the best for this – many higher level languages use basic libraries written really efficiently in C, which are called from the code of this same high-level language (the most famous are Python and Java). Those languages also use mostly the same syntax for the basic program actions. In addition – C++ is the only language we can count on as universal and since few years already as fast as C. There are some measurements that even for some specific use cases and high-level programming it is faster than C (example for one of several reasons – templates meta programming, doing a lot of work "in advance", during compilation). This is the reason why many high-level-languages have now basic libraries written also in C++.

In the last ten years we have as well crucial development of the compilers – with each year they produce faster and better code. Their optimizations get better and better. The programming languages develop as well. And so the compilers have to develop equally. Clang/LLVM was the strongest reason for pushing also GCC to improve itself in terms of efficiency. Briefly - the team behind Clang has put themselves the high target of developing better and more efficient compiler, producing also faster code, so GCC had to catch up in this chase (and adopt many of their ideas).

In addition to the previous paragraph – the CPUs also develop. As we have already reached the top gate speed of current transistor-based CPUs (the GHz) – they developed improvements in:

- number of cores and their cooperation;
- interpretation and execution of the instructions;
- multi-instructions pipelines;
- improvements in cache access and usage
  - Because memory speed and operations with cache and RAM are still quite slower than the CPU core. And here we don't talk about frequency – the problem is the RAM and cache latencies when accessing portions of data which are located in distant different places of RAM.
  - Each piece of data requires loading it to cache so that the CPU core has access to perform the work. Each cache memory data page load requires time and reserves the given cache page for certain pieces of data, blocking it for others.
  - If we need some other pieces not located in cache – the cache memory management unit (cache MMU) has to define the least used data, unload it back to RAM, then load the new data to process.
  - In general RAM access and data management is a slow process and many improvements have been (and continue to be) developed to make the process faster and more efficient.
  - In addition not all CPUs possess such rich features as they are expensive – so despite that we have magnificent development in this area still some "lower class" CPUs experience real trouble with Cache/RAM access speed.

In conclusion – writing universal code, in universal language (C++) would require:

- the cleanest instructions;
- check on different compilers;
- check on different machines and even OSes – as this changes the environment
- checks with different optimization levels.

Reason for this – once we write our magnificent code it gets translated into CPU instructions. Which exact instructions will be chosen and in what sequence is quite crucial – as if we have to repeat certain piece of code 50000 times it **will** make a **big** difference. Also – as some CPUs have extra-run-time-optimization features – whether the compiler would make use of them also counts. And each of those versions shall be tested on different CPUs to confirm the final result. In order to get an idea of what differences can have a family of CPUs – check this Wikipedia page about the ARM-Cortex-M family.

As you can see – simply doing the O() analysis without considering all other points would be quite insufficient.

SERIOUS ADVICE – if not limited by some very special reason – always use the latest versions of the compiler of choice. Only by this way you will get those new fancy optimizations, or make use of the new CPU fancy features. And take if possible a short look on the release notes when switching to a newer version.

4. Speed of basic operations / Instruction Latency, Instruction Cache / Pipeline, internal CPU interpreter, Memory Cache, etc.

This chapter is here to give you some basic ideas of the HW we write our code for. It is good to know at least the basics, and critical to have an idea of Instruction Latency and memory cache in particular.

a. Speed of operations / Instruction Latency - add, subtract, compare, divide, modulo, cache access, read / write, etc.

When we want to compare basic CPU operations we do this by the **amount of CPU clock cycles**. The CPU clock is the heartbeat of the CPU (this number for you CPU like here at the end: “ARM Cortex A9/ Intel core i7/ xxx running at **2.6 GHz**”).

So – if we talk about division – it may take e.g. from 15 to 150 (for very old or weak MCUs) clock cycles on different CPUs, it will be different for floats and integers, many times depending also on their size (8,16, 32, 64 or 128 bits), depending a lot on the type of ALU and the CPU core architecture.

Addition may take e.g. exactly 1 cycle on some CPU with specific Mathematical Co-processor and optimized data pipes, it may take 3 cycles on one which is “weaker”.

All operations have different times, and all those numbers are different for different CPUs. They are sometimes different also between **CPUs from the same series**. And again – we don’t talk about the frequency of the CPU, i.e. we don’t care whether our CPU works at 500MHz, 1.5GHz, or 2.8GHz. We count **the basic clock cycles**.

We call this **instruction latency**. The instruction latency table might be available for your CPU, though it is hard to find for some modern processors, and also sometimes hard to read. One of the best resources on this topic is this website: https://www.agner.org/optimize/. For ARM cores you can search at https://developer.arm.com, here is an example for the Cortex A9 FPU, here generic for Cortex A7.

In the past I could locate some easier to read tables, but nowadays it is hard to find digested information. Reason – some companies hide their data, in addition – simply going through instruction latencies would not be enough, as many times the current OS, usage of memory and running concurrent processes can change the results and the calculation of those latencies depend on parameters. We can accept in general this relative table (**again emphasizing that it may differ in serious proportions for some CPUs, but the general point will still be the same; I think as well that for the near future it will remain like this – until we change drastically to other than current transistor based CPUs and/or their current architectures**):

Instruction	Relative speed in cycles
Increment, Decrement	1
Bitwise Shift left / right	1 to 3
Bitwise operations - AND, OR, XOR	1 to 3
Compare	1 to 2
Add	1 to 3
Subtract	1 to 3
Multiplication	1 to 3
Modulo ("%" operator) (division remainder)	5 to 1XX
Divide	5 to 1XX
Assign (use the equal "=" operator) for Register variables	1
Assign (use the equal "=" operator) for cell located in L1 Cache	1-3
Assign (use the equal "=" operator) for cell located in L2 Cache	10-30
Assign (use the equal "=" operator) for cell located in L3 Cache	25-120
Assign (use the equal "=" operator) for cell located in some RAM location	50-200
Extract value (to compare or apply **some** instruction) for Register variables	1 + instruction time
Extract value (to compare or apply **some** instruction) for cell located in L1 Cache	1 to 3 + instruction time
Extract value (to compare or apply **some** instruction) cell located in L2 Cache	10 to 30 + instruction time
Extract value (to compare or apply **some** instruction) for cell located in L3 Cache	25 to 120 + instruction time
Extract value (to compare or apply **some** instruction) for cell located in some RAM location	40 to 200 + instruction time

If you are interested to get some real data look into these resources:

https://www.agner.org/optimize/instruction_tables.pdf

https://uops.info/background.html

https://uops.info/table.html

http://instlatx64.atw.hu/

To have some basic idea of how the CPU work – apart from searching through the internet you can take a look at this Wiki page.

The main conclusions:

- Working extensively with division and modulo is in general **SLOW**.
- Working extensively for whatever operations with data located far away in RAM is always **SLOW.**
- **One more point** – working with floating point numbers on most common CPUs is normally **slow**, though lately the FPUs also get better and better.

- The reason why we have the so called fixed point arithmetic, a.k.a. transferring all work from floating point to scaled integer domain is because it is faster. This was frequently common practice in the past when the usage of floating points was even more expensive.

Counting on mathematical co-processor and good powerful CPUs means bad design; it is acceptable and might even be required – but only for exact specific use cases, **never** for anything claiming to be **universal**.

To finalize – of course the basic CPU clock counts – if we run at 1.0 GHz – we multiply the amount of cycles by 1 ns – so some operation sequence requiring 50 cycles will need minimum 50 ns for itself. If we work on the exactly same CPU at 2.0 GHz – the total time will be 25 ns. This is clear, but it is not the point here ;).

b. Instruction Cache and pipeline, hidden internal core

When a modern CPU has to perform a series of operations it does so on a set of several or more operations. The most powerful CPUs have really big instruction caches and optimized pipeline.

This Wiki page explains some basics about the CPU Cache and this about instruction pipelining. This subject in general is not something to consider while coding, it is just good to have an idea about it.

Now **the hidden core** is another story. What some CPU makers like Intel have understood, is that sometimes they might know how to reorder some instructions to improve efficiency as no matter how good a compiler is – it is limited by generating generic code to work on many CPUs. This means – if I compile a program for Linux and x64 architecture – I don't care whether I ran it on AMD, Intel XEON, core i5, i7 or whatever. So they have added a **"middle-man"**, which takes the next set of instructions, checks whether it can improve their sequence or execution efficiency and finally sends them to the real core a bit modified. Very simple description of this is available here. This is one more reason why for some CPUs we cannot even count that our instructions will take some exact amount of time. They can be optimized and be faster, or suffer some interrupt, cache wait event, etc. and finally be executed slower than expected.

Of course if our algorithm is slow by design – no CPU, compiler or other optimization can help us at all.

c. CPU Architecture

To get an idea of how complex the architectures have become take a look at this list:

https://www.researchgate.net/figure/Intel64-64-bit-execution-environment-for-Nehalem-processor_fig1_235960679

https://en.wikipedia.org/wiki/Comparison_of_ARM_cores

https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

https://en.wikipedia.org/wiki/Apple%E2%80%93Intel_architecture

https://en.wikipedia.org/wiki/List_of_Intel_processors

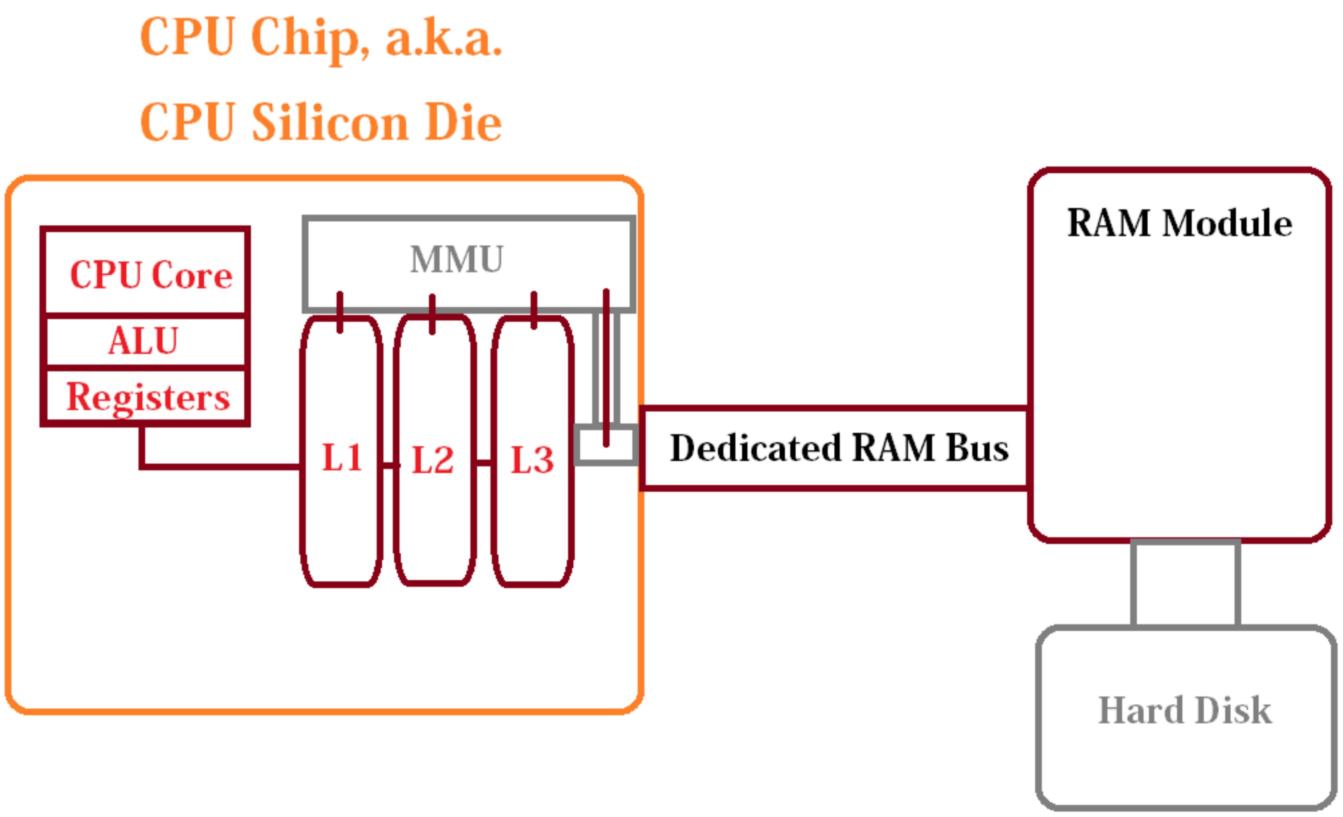
What we can say shortly is again that when we want to make a universal algorithm we shall stick first to the analysis of the O() notation, then to the simplest and cleanest instructions, and finally pay attention to all points mentioned in **Chapter 3.b** for **Cyclic actions count**. Then in order to know that our algorithm will be really fast compared to others – we shall test them all on several different platforms, so that we know that no matter the compiler or optimizations the efficiency is always as expected (compared e.g. to certain time limits or against other standard and non-standard algorithms).

d. Memory Cache and it's speed – literally crucial on almost all modern systems

Why is this so – before we go on further – if you search on the internet for CPU against memory speed development in the last 30 years you will notice how the **CPU cores increase efficiency brutally**, while **RAM** and **cache** gain **improvements** quite **slower**.

Why do we need memory cache? – Because it is the only memory directly available to the CPU core. It is on the CPU silicon die, inside the CPU chip, while RAM is a completely separate chip or even whole module connected to the CPU via special bus. The CPU core rarely works directly with RAM – most of the common purpose modern processors have at least small amount of cache – e.g. 32KB up to 30 MB – to which the data is transferred. In comparison the RAM is from e.g. 64 MB – 64 GB. For most processors only after data is transferred from RAM to Cache it is processed, then unloaded back to RAM, then the next piece is loaded from RAM to cache and processed, and so on. The reason is that RAM is fast compared to opening a file from a hard drive, but it is slow when accessed from the CPU. That's why most high-end / more powerful processors have normally 3 levels of cache, usually total up to several MB on each level.

The "closest" cache to the CPU core is labeled commonly L1, then we have optional L2 – a bit slower to access, and optional L3 – significantly slower. If L1 is almost like working with direct registers, L2 takes hypothetically e.g. 2-5 cycles to access, while L3 would take more. You can imagine it like this (the picture is generic, but presents the idea):



The architectures, their ALUs, Cores, Registers, and finally their MMUs (Memory Management Units) are so different, that one cannot count on doing very special optimizations for this or that CPU, except in custom specific scenarios. There is something we can do though, but first take a look to this table:

Type of memory access of a theoretical "good PC" with powerful 3 GHz x4 cores CPU, 16 GB RAM, 256 GB SSD, 1 TB HDD	Required Time
1 Clock cycle of a 3 GHz CPU	0.33 ns
Register Access	0.33 ns
L1 Cache Access	1 ns
L2 Cache Access	~3 ns
L3 Cache Access	~10-20 ns
RAM Access	~50-150 ns
SSD Access	~5-200 microseconds
Regular HDD Access	1-10 ms

Again you can search for more information on the subject, starting e.g. from these articles from AnandTech: 6th Gen Skylake Review – Core i5-6600k and Core i7-6700k, one ARM server review – look in particular the third part about **Memory Subsystem**.

So – what can we do? We shall first split our data to smaller chunks. Depending on the data those can be several hundred bytes or several kilobytes. Then we shall sequentially process each chunk, doing all the work we want with it. By this way all the current data is loaded into Cache. So we work with the fastest memory.

In other words – we shall avoid making "Long Jumps" in memory too frequently.

To illustrate **the bad case** with some theoretical case – let's say we have a 500MB array of data, and we jump between 10 locations each separated from the other from 50 MB of data. This means that for each jump we have to load data from RAM to Cache, do what we need to, unload, load the next data piece, and so on. If we do this sequentially it might happen that each execution of one micro-step in our algorithm can take e.g. 300 ns. Let's imagine that we need to repeat this 1,000,000 times – this would take 0.3 seconds.

If we find a way to group our data in such a way, that those 1,000,000 times do work on 10 locations which are located into one piece of data of size 1 MB (by using smarter structures and memory allocation), then we might increase the overall efficiency e.g. with 40-65% only because we don't go around in RAM like a mad man. By doing some other tricks and optimizations we might have at the end execution time reduced from 300 ms to e.g. less than 5 ms.

You will see some examples with the sorting algorithms I investigate, and despite that those will illustrate also other characteristic those will present what happens when we process the data in smaller pieces.

To finalize this chapter:

1. There are a number of lections and publications on the internet for writing a "**Cache Friendly C++**" like this, that and this one. Though the subject is not widely discussed, there are some resources, look into them.
2. If you want to understand more – research the subject of **Cache misses and hits**, or **Cache Hit Ratio**. Again not common subject, but there is **some information**. Shortly put – when the MMU of

your CPU tries to load some portion of data into cache – if by some reason this fails – it will try again after some timeout and / or additional events. If your code is written in such way, that we have a lot of cache misses – many times your program will stay waiting for several hundred nanoseconds / or several microseconds waiting for data. If this happens a lot – you end up with slow program due to bad design.

3. **The cache is not only for us.** Usually cache-rich CPUs are for higher level CPUs and we don't know how many and what other processes and threads are running on the system while our program is running. If the system is smaller, with only a little bit of cache – we again have an issue with the available amount. Cache works on pages, i.e. small portions, and those are not thousands, instead they are **only a few**, up to **several tens**.

e. Operating system

What is important for an OS is that it:

- Runs hundreds and sometimes even thousands of threads;
- Gives each thread some time slots, then gives to the next one, puts some to sleep (or they go to sleep if idle), in general the OS manages when our program gets access to the CPU;
- Many times manages the memory assignment, talks directly with HW, e.g. interacts a lot with MMU, also with RAM, which might influence our work;
- If we have more cores – normally it manages the task allocations as well;
- It might run additional tasks which require heavy CPU, RAM or GPU resources – we cannot know;
- It might be Low Level RTOS (several hundred available), custom OS, Windows (enough versions), Android (enough versions), OS X (enough versions), Linux (there are thousands of distros, enough of them make a lot of changes in the way the OS works), Embedded Linux, etc.

Take a look at Linux – more and more features, more and more optimizations and customizations – so that now it can be running from the super Limited RTOS-like-version on a device with 100MHz CPU, to the fastest server with 2x12 cores CPUs.

Actually each of the mentioned OSes gets better and faster with each year.

So – considering all this – it is our duty to always try to write our algorithms as efficient as possible. Only by this way we will be useful to the maximum amount of users. Only by this way we will be less dependent from the environment and the required resources, from which time is many times the most precious. And this is **no matter the language, no matter the platform.**

5. My test framework.

Once I decided to write a better algorithm I had to devise a testing method. Hence I wrote a generator of numbers arrays, and a test framework. Following is a description of the structure and some basic points on why/how I designed it. **Important Note**: this description is related to the first version of the test framework, in the future slight or significant changes may appear!

We have the following points:

1. Time measurement
2. Checks before and after sort
3. Short, normal and long test
4. Alignment
5. Test Data types
6. Single Test Data Container
7. Test Data Set structure and allocation
8. Main testing class and Copying of data for each intermediate run
9. Algorithms
10. Platform information
11. Executing the test, Multiple runs and statistics
12. Common functions, command line prints, txt file output

a. Time measurement

The time measurement shall be accurate. Hence I use the C++11 provided accuracy of nanoseconds, via the **std::chrono::high_resolution_clock::now()** function.

I have first compiled the project with QtCreator and MinGW (on Windows 8), but when running it I got accuracy of 100 ns max. Then I transferred the code to MSVC 2019 and there I got the real expected accuracy of 1 ns.

The reason for using the highest resolution is that many of the small arrays sorting are performed in a matter of few thousand nanoseconds and there ±100 ns count. On top of that I do multiple runs, and if we have an accumulative error of ~100 ns – we might get significant deviations.

The function I use is:

```
nanoseconds getTime_Since_Epoch_ns();
```

defined in Tools\Time_Measure_Chrono_ns.cpp /.h.

There is also a simple function in the header file for converting the nanoseconds string to be formatted with commas at before each third sign so that e.g. the time in nanoseconds 3691012918 is printed formatted like this: 3,691,012,918 ns. That's simply easier to read – we know that going from right to left we have nanoseconds, then microseconds, then milliseconds and finally seconds :). Here is the prototype:

```
inline auto convLng_ns(unsigned long long par) -> std::string
```

The execution time of each sort is measured exactly from the moment before entering the sort and until the moment that we return from the function. In particular the code is:

```
        // record start time:
        auto startTimeStamp = getTime_Since_Epoch_ns();

        // call the function
        sortFunc(dataNotAligned->arr, input->size);

        // record end time, calculate execution time, write result:
        execTime = getTime_Since_Epoch_ns() - startTimeStamp;
```

b. Checks before and after sort

Trust me data generation – which is done in its own project – can be an issue. Considering that you generate over 150 files with data – how big do you think is the chance for error :)? So I have written few very simple functions to check whether a given array is sorted, reverse sorted or shuffled. Those are called in the appropriate places before and after sorting, also to check whether the input is as planned to be sorted, reverse or not sorted. The functions are located in the files Tools\Arrays_Tools.h and cpp and have the following prototypes:

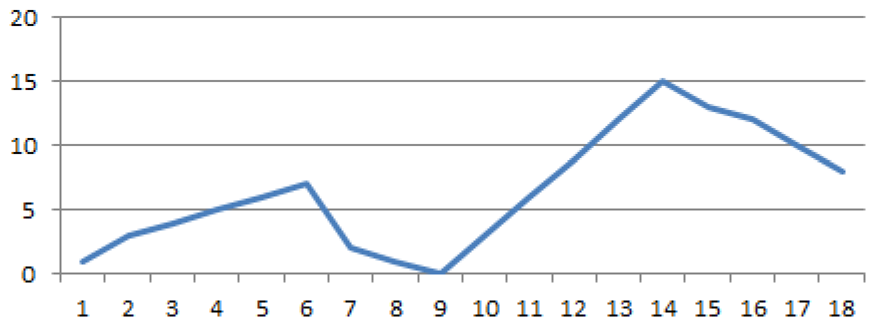
```
unsigned long counterOfPeaks(int arr[], unsigned long size);
// returns 0 if array sorted, otherwise the index of first inversion and breaks further check
unsigned long checkIfArrSorted(int arr[], unsigned long size, bool rev = false);

TestDataType_ERROR checkArrayInputBefore(SingleTestContainer* testCtr);
TestDataType_ERROR checkArrayInputBeforeForMultiTest(int* dataArr, SingleTestContainer* testCtr);
```

The counterOfPeaks() idea is that if we have e.g. this array of 18 numbers:

1,3,4,5,6,7,2,1,0,3,6,9,12,15,13,12,10,8

we have two peaks like on this chart:



I use it only to check whether in given array we have some or no peaks at all.

c. Short, normal and long test

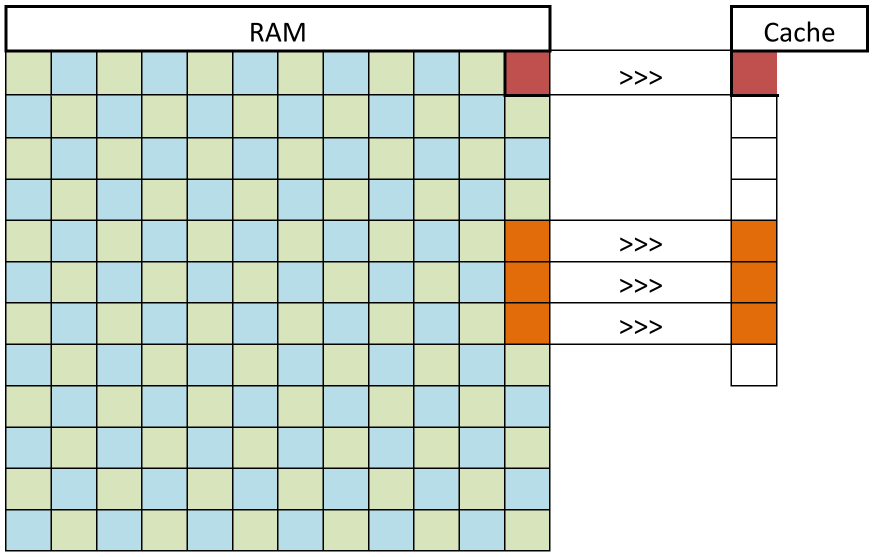
Once I started testing I noticed that we have **very slow**, slow, **fast**, and **super-fast** algorithms. Some of my runs required over 40 minutes for only one algorithm for only one run over all test arrays. Currently on my Xeon with Windows 8 many of them perform in a matter of few seconds (or less than a second for shorter tests). So we have the following test execution lengths:

```
enum class TestExecutionLength {
    SHORT,
    NORMAL,
    LONG
};
```

d. Alignment

Another important point related to data management and CPU memory cache is whether the data is **aligned**. This is so, because the Cache works on sets of data, and when those are aligned with some memory region the copying of data is faster. This is not valid for all CPUs as some have optimizations, but on some the difference will be significant.

To understand the problem – take a look to this picture where each square represents a micro-page of 4 KB of data:



Imagine we have some single cache of 8 pages x 4 KB for our CPU (total 32 KB). So – if our data is packed and aligned in such a way that it is e.g. 3,5KB located in one single block starting at address 0xABCD1000 – then the MMU will have to load one single block and the CPU will work on it. If the same data is split into several locations in one big block of 10 KB, which is spreading over 3 cells – the MMU will have to load several pages – this not only requires more time, but if those were occupied before this – the MMU will first have to unload them to RAM to make space, then load 3 pages instead of only 1, and after this whole process is finished we can start processing the data with the CPU core.

This is again only a theoretical picture. Some high-end CPUs have optimizations so that they don't have delays when loading data from not aligned addresses, for the rest – each jump to next cell is a separate process and starts some complex MMU state machine.

The process, MMU and CPU architectures and optimizations are so diverse, that this example depicts the issue only in some hypothetical generic form.

For you it is important that you can use the dedicated **alignas specifier** **(since C++11)**. Even the example code on **cppreference** mentions such a potential use to align a char array:

```
// the array "cacheline" will be aligned to 128-byte boundary
alignas(128) char cacheline[128];
```

As a result of the upper code the address in RAM will not be 0xABCD1234, but instead (as decimal 128 is hex 0x80) – the address might be either 0xABCD1200 or 0xABCD1280.

This is the reason why all my test arrays have two versions – one which is aligned, and one designed to be **not-aligned**. How do I do this – by specifying explicitly alignas(4096) for the aligned version of the array, and specifying alignas(4) for the not aligned version. Unfortunately I don't know whether the Linker would not optimize the data (if I trigger some optimization feature which might override my alignas instruction, r anything else happens during compilation) – so for each not-aligned version I add in front first a 92 bytes offset structure, and then the data array which will be sorted:

```
typedef struct arrayNonRepeatSorted_NotAligned_128_s{
    Offset_Struct_92Bytes_t offStruct;
    int arr[SIZE_arrayNonRepeatSorted128];
} arrayNonRepeatSorted_NotAligned_128_t;
```

```
alignas(4) arrayNonRepeatSorted_NotAligned_128_t arrayNonRepeatSorted_NotAligned_128 = {
offset_alignment, {-2147483648,-2147424631,-2147420383,-2147418675,-2147311553,-2147196086,… -
… …. … …. …. ……. … .., 2138934737,-2138808867,-2138680798,-2138679331,-2138654184,}};

alignas(4096) int arrayNonRepeatSorted_Aligned_128 [128] = {-2147483648,-2147424631,-2147420383,-
2147418675,-2147311553 ……………..
```

Pay attention to one lesson here – data alignment is not the cure for inefficient code. Aligning all your structures is not what will help you write faster programs. **Proper data structure** combined with **proper alignment of some of the structures** however may increase the speed of processing your data immensely. Some interesting points are covered here by Ivaylo - a colleague from Sofia (Bulgaria).

e. Test Data types

I check each algorithm for these types of arrays:

- **Random** numbers with repetition – means that many of those might repeat themselves.
- **Random** numbers without repetition – artificially shuffled.
- **Sorted** numbers – so that we check how our array behaves for one worst case for some algorithms.
- **Reverse sorted** numbers – so that we check another classical worst case.
- **Sorted** numbers **with some zeroes** inserted – this is what happens if we have part of the work done – are we better, or we perform the same like if everything is shuffled.
- Data which is **only zeroes** – because it is possible that some implementations have issues with the 0 (though this shall never happen). This is again practically sorted array.
- Data which is **only one number** – just to confirm that we have no issues here either

The sizes of the arrays are **small** (20-2050 elements), **big** (32768-524288 elements), and **super big** (2096128 elements). As you will see the differences can give us some interesting conclusions.

Those classifications are defined in "Tools\TestDescriptor.h":

```
enum class TestDataType {
    RAND_with_Repeats,
    REV_SORTED,
    RAND_NO_REPEATS_SHUFFLED,
    SORTED,
    SORTED_WITH_SOME_ZEROES_INSERTED,
    ONE_NUMBER,
    ONLY_ZEROES,
    First=RAND_with_Repeats,
    Last=ONLY_ZEROES
};

enum class TestDataSizeType {
    SMALL_32_2050,
    BIG_32768_524288,
    SUPER_BIG_2096128
};
```

All generated files are located in the folder "Test Data", and then a subfolder with corresponding name for their `TestDataType`.

Each set of aligned and not aligned version of the given array (for the aligned I explain later) is located in a separate cpp file, so that it is compiled separately. The corresponding header only "externs" the data. Extern-ing is acceptable here, we do know what we will do, in addition no array and no header or source has duplication in names – they always contain the `TestDataType` and the exact number of ints in their names.

f. Single Test Data Container

The container struct for all test data is one single and located in the file "**Tools/TestDescriptor.h**". It has a pointer to the array together with all basic parameters. The constructor simply initializes all internal parameters.

```
struct SingleTestContainer {
    int * array;
    unsigned long size;
    unsigned long long byteSize;
    // classification
    TestDataType dataType;
    TestDataSizeType sizeClass;
    // aligned
    TestDataAlignment align;
    std::string info;
    unsigned long long execTime_ns = 0;
    TestDataType_ERROR err;

    SingleTestContainer(int * ptr,
                        unsigned long sz,
                        TestDataType tp,
                        TestDataSizeType szCls,
                        TestDataAlignment ali,
                        unsigned long long btSz,
                        std::string descr) : array(ptr), size(sz), dataType(tp), sizeClass(szCls),
align(ali), byteSize(btSz), info(descr), err(TestDataType_ERROR::ARRAY_OK){
    };
};
```

This type is used in each generated array cpp file to initialize all parameters. All of them are filled during generation. The example below is from the generated file

Test Data\NonRepeat_Reverse_Sorted\GeneratedNonRepeatREVERSE_Sorted_37_numbers.cpp :

```
SingleTestContainer
arrayNonRepeat_REVERSE_Sorted_NotAligned_37_c(arrayNonRepeat_REVERSE_Sorted_NotAligned_37.arr,
(unsigned long)37 , TestDataType::REV_SORTED, TestDataSizeType::SMALL_32_2050,
TestDataAlignment::NOT_ALIGNED, sizeof(arrayNonRepeat_REVERSE_Sorted_NotAligned_37),
"arrayNonRepeat_REVERSE_Sorted_NotAligned_37");
SingleTestContainer
arrayNonRepeat_REVERSE_Sorted_Aligned_37_c(arrayNonRepeat_REVERSE_Sorted_Aligned_37, (unsigned long)37
, TestDataType::REV_SORTED, TestDataSizeType::SMALL_32_2050, TestDataAlignment::ALIGNED,
sizeof(arrayNonRepeat_REVERSE_Sorted_Aligned_37), "arrayNonRepeat_REVERSE_Sorted_Aligned_37");
```

g. Test Data Set structure and allocation

The container struct for all test data is one single and located in the file "**Tools/TestDescriptor.h**". Its definition looks like this and the interesting points here are the **constructor** and the **initAllTests** functions:

```
struct TestData {
    std::vector<SingleTestContainer*> testVect; // a vector of pointers to all NOT aligned arrays to
sort
    std::vector<SingleTestContainer*> testVectAligned; // a vector of pointers to all ALIGNED arrays to
sort
```

```
    std::vector<SingleTestContainer*> testVect_all; // a vector of pointers to all NOT aligned arrays
to sort
    std::vector<SingleTestContainer*> testVectAligned_all; // a vector of pointers to all ALIGNED
arrays to sort

    targetHwListing hw;
    targetMachineListing machine;
    targetCompilerListing compiler;
    targetOsListing os;

    TestData(targetHwListing hw_, targetMachineListing mach_, targetCompilerListing cmpl_,
targetOsListing os_, TestSelection sel, TestExecutionLength length);

    void initAllTests(TestExecutionLength length);
};
```

The purpose of the first four parameters of the constructor is self-explanatory. Now the initAllTests() simply reserves enough space for the vectors of all tests and push_back() all SingleTestContainer data generations. This is done via a inclusion from a generated header file (from the Test Data Generation project). It's easier this way – whenever I want addition of data I would not have to manually recode and e.g. add TestData vectors. Just check the code.

initAllTests() is called in the beginning of the **constructor**. Then – if we want ALL tests (the common case) – we can choose whether we want only all SHORT (the Short execution), or all SHORT + NORMAL (the Normal execution), or really all tests (the Long execution).

The other cases are interesting only when we do specific test for some algorithm – so it is up to you whether you will modify them, add some, etc. You can use them as an example on how to add your own specific test set. Here is the enum class for reference:

```
enum class TestSelection {
    ALL,
    ONLY_EQUALS_SHORT,
    ONLY_SORTED_SHORT,
    ONLY_REV_SORTED_SHORT,
    ONLY_DIVISIBLE_BY_4_NORMAL,
    REV_AND_EQUALS_SORTED_SHORT,
    SPECIAL_SHORT_SELECTED_512
};
```

h. Main testing class and Copying of data for each intermediate run

Initially I wrote the test framework to test my own sorting algorithm. Once I have added more I wanted to be able to take the same input again and again for each next run of the test set. So I had to prepare additional copying of the data for each current run.

The whole testing is performed from the class TestExecutor, located in the main.cpp. The containers for aligned and not aligned data are defined exactly before the class as follows:

```
alignas(4) mtDataNotAligned_t dataNotAligned_b = {};
alignas(4096) int dataArrayAligned_b[DATA_CONTAINER__8MB] = {};
```

The class has a simple constructor without parameters initializing the test length and selection, the description of the current HW for which it is built, and the pointers to the data containers for multiple runs.

The only 3 functions are: for performing a **single** test – practically the initial prototype and currently never called, the **multiTestDynamicData** and the **saveToCsvFile** for the results:

```
void performSingleTestStaticData(AlgorithmID algoID, std::string algoName, std::string genInf,
sortFuncPtr sortFunc);

void multiTestDynamicData(AlgorithmID algoID, std::string algoName, std::string genInf, sortFuncPtr
sortFunc, bool testAligned = false);

    // then all the results shall be saved in a csv file
void saveToCsvFile();
```

i. Algorithms

Each algorithm consists simply of one header and one source file. I did this on purpose to have maximal simplicity. All the algorithms are located in the "**Algorithms**" folder. If any of them requires some additional headers – those will always be located in an additional sub-directory in this directory. Only the main header file is included in the main.cpp and called for a set of tests.

j. Platform information

The purpose is to test all the algorithms also on different platforms so that one can compare the efficiency of the given coding when the amount of RAM, type of CPU core, cache, motherboard architecture, MMU, ALU, OS and so on are different. This is the only way to confirm which the fastest algorithm is.

The information is located in the file Tools\TestDescriptor.h.

k. Executing the test, Multiple runs and statistics

Once I started testing I've noticed that especially for short arrays we have significant differences for the different runs. We are talking here about 5-15% (the last in worst case). I thought about it and I realized the cache issues were the main reason for this. In addition we might have other influences related to RAM usage, threads, OS, etc. So I've understood I need multiple runs and average results for each type of data to be sure that one or another algorithm is really always performing better. Thus each algorithm test on the chosen set is performed like this:

```
TestExecutor testObj;

#define RUNS_PER_ALGO 10

    for (uint8_t run = 0; run < RUNS_PER_ALGO; run++) {
        testObj.multiTestDynamicData(AlgorithmID::RusevMergeSortBasedOn2_32StaticStacksOPTIMIZED_Dyn,
"RusevMergeSortBasedOn2_32StaticStacksOPTIMIZED_Dyn", "",
RusevMergeSortBasedOn2_32StaticStacksOPTIMIZED_Dyn);
    }

  for (uint8_t run = 0; run < RUNS_PER_ALGO; run++) {
        testObj.multiTestDynamicData(AlgorithmID::NEXT_ALGORITHM, " NEXT_ALGORITHM ", "",
NEXT_ALGORITHM);
    }

…..

testObj.saveToCsvFile();
```

The RUNS_PER_ALGO macro can be changed manually and the separate results for each run are printed in the csv results file on a separate line. Yes this means after this you have to manually create the statistics, sorry. I didn't have the time yet to improve this.

The multiTestDynamicData(…) consists of 6 parts:

1. `TEST FOR NOT ALIGNED DATA`
2. `TEST FOR THE ALIGNED DATA`
3. `OUTPUTTING ALL RESULT VECTORS`
4. `Statistics preparation`
5. `Final Results Printing`
6. `Checking final errors`

The last part is important as we have a lot of tests and statistics – so I check whether the averages and the Total Time match. The final SUCCESS status of the current test requires no deviations in average times, no inpout or output wrong arrays.

I. Common functions, command line prints, txt file output

Any common functions are / will be located in files under the directory Tools. You can find there the ArrayTools.cpp, TestDataAllocator.cpp, TestDataContainer.h, GenericMergeFunctions etc.

Regarding the Output – in multiple occasions I needed to print debugs for different functions and algorithms. For this I use this kind of common way to define them:

```
#define DEBUG_PRINTS_RUNTIME_LISTING_ARRAYS_OFF
#ifdef DEBUG_PRINTS_RUNTIME_LISTING_ARRAYS_ON
#define lout cout
#else
#define lout 0 && cout
#endif
```

In case the corresponding print is needed – simply change the suffix from _OFF to _ON and you get cout for the given messages, otherwise it is practically a zeroed instruction.

For the prints simply call instead of e.g. "cout << …." :

```
cout << " \n Testing Not Aligned ADDRESS: " << input->array << "  INFO: " << input->info << endl;
```

```
lout << " \n Testing Not Aligned ADDRESS: " << input->array << "  INFO: " << input->info << endl;
```

A text file output exists for each algorithm execution for each run. It is with prefix "RESULTS " and then the algorithm function name. It gives a listing of all arrays used for the current session of testing and at the end a summary for the total execution together with average results for small, big and super big tests. Currently it overrides the existing file completely, so if you use a loop of 10 test runs – you will have the execution results only from the last run. I left it like this as I don't want to have a test log 20000 lines per algorithm. I also used it in the past for debugging purposes.

If there is an error in the array check – ti is printed both on command line and in the report. In addition – the report summary tells you at the end the total number of wrong arrays. Their results are not considered in the final summary with average numbers.

----

m. Considerations when testing

6. Dual Pivot Quicksort and TmiSort. Divide and conquer - issues with and without merge - the exponential rise of execution time.

7. Radix Sort

8. Other sorts

Comparison of many famous algorithms (definitely not all implementations, a bit generic, but magnificent to get a good overview) is available in Wikipedia. Also a good place from GeeksForGeeks. There are as well other locations, check them out if you want.

9. Templated versions of the algorithms

Merging and stacks – those shall be dynamic, obligatory. No library function which allocates by default is acceptable.

int64_t and int32_t – why we need to test them both? Theoretically we have to have simply the same results with some relative increase. This means – the better algorithms remain better, the slower remain slower, but both of them have some increase the execution time. BUT – considering that the modern 64 bit architectures (and they are 64 bit since a lot of time) are optimized for 64 bit – will this be slower at all?? Alignment issues may come out as well.

10. Testing and checks – why it is obligatory. Size of arrays, difference of results depending on size.

In addition – the time measurement shall be exactly from before calling the given sort function, to exactly after we exit from it.

Data – sorted, not sorted, reverse sorted, small, average and big.

Smallest – with 20 and with 10 elements, just to check that the algorithm would not fail with only a few!

Why testing also with sorted or reverse sorted data?

1. The cost is low, the code exits at first peak. Have to check what happens with plateaus of data 13,14,14,14,14,13
2. The full pass over sorted data gives us the time of single pass over the array on the given platform.
3. SORTING a REVERSED array gives us the worst case time!
4. One single number – gives us worst case to check a complete array, also shows issues in implementation
5. Only zeroes – gives us specific case for algorithms, which in their implementation use the comparison with 0, so it is one potential corner case.

Statistics based on multiple runs. From my practical experience – from run to run we might have at least slight differences in the results. Those differences come mostly from the cache pages load/reload of L1, L2 and L3 caches, and the corresponding memory latency. Those are also dependent from other running SW. So it turned out I have to make multiple runs of my multi-test cases algorithm to be able to take a real average.

Checks before and after sorting. These are obligatory as with the necessity to generate e.g. 144 files with input data – one can easily make some mistake. Those checks got enough times error in the input, and few times also in the output. Of course they are not considered in the time measurements.

11.Testing and checks – the different platforms to test on

Now you know a bit more about how different each platform can be. The features like caches, internal CPU commands interpreter, speed of CPU and / or RAM architecture, OS – all those get better and better, faster, and can be quite different.

So – when we compare the results we cannot compare between a Xeon from 2013 with Windows 8 and a Core i7 from 2018 with Windows 10, not to mention something else like Linux or MAC OS. What we always look at is which is faster on the given platform. And we care about defining which is the fastest algorithm on more platforms. For this I will be grateful if some of you compile and test it on a modern MAC. Also if we have more AMD tests. You can either fork my project, or simply write me, so that we can get in contact, you provide me the results with your name and platform details in them and I can add them to the final results.

12.Conclusions and final results