Is there any $\mathcal{O}(2^n)$ algorithm your OS can run in almost no time ?

Daniel Porumbel[1]

CEDRIC, CNAM, Paris, France daniel.porumbel@cnam.fr

If you think it's simple, then you have misunderstood the problem.

Bjarne Stroustrup (concepteur C++), 1997

Truth is no harlot who throws her arms round the neck of him who does not desire her ; on the contrary, she is so coy a beauty that even the man who sacrifices everything to her can still not be certain of her favors.

Arthur Schopenhauer, 1844

The misleading appearances

If you think that the answer to the question from the title should be "No, it is completely impossible ; that would be heresy against the common sense of fundamental Computer Science !", remember the motto of this paper. By the way, both quotes mean the same thing : reality and practice can sometimes be so complex that they evade the belief system and the axiomatizations that theory has accustomed us with.

The two programs below execute the same C/C++ instructions. The code was aligned to easily see the correspondence line by line.

```
#include<iostream>
using namespace std;
int main(){
    int n = 30;
    int z = 1<<n;           //2^n
    z++;
    int*tab=(int*)malloc(z*sizeof(int));
    for(int i=0;i<z;i++)
        tab[i] = 0;
    for(int i=0;i<n;i++)
        if(tab[1<<(i+1)] < 7 + tab[1<<i])
            tab[1<<(i+1)] = 7 + tab[1<<i];
    cout<<tab[z-1]<<endl; //always 7*n
}
```

```
#include<iostream>
using namespace std;
int main(){
    int n = 30;
    int z = 1<<n;           //2^n
    z++;
    int*tab=(int*)calloc(z,sizeof(int));
    //recall calloc returns by default
    //a memory block filled with zeros
    for(int i=0;i<n;i++)
        if(tab[1<<(i+1)] < 7 + tab[1<<i])
            tab[1<<(i+1)] = 7 + tab[1<<i];
    cout<<tab[z-1]<<endl; //always 7*n
}
```

The two programs were compiled using `g++` with no optimisation option. The left program needs about 0.75 seconds for $n = 28$, 1.5 seconds for $n = 29$ and 3 seconds for $n = 30$. This follows an exponential law that corresponds to the expected $\mathcal{O}(2^n)$ complexity. But – believe it or not – the right program needs about 0.001 seconds for $n = 30$ on my machine. I uploaded a video showing this at `youtu.be/9NEOrdLM3Zg`. This behaviour was confirmed by a dozen of my colleagues on their computers (far from using the same OS or the same compiler). The $n = 31$ case fails in both cases ; this will be discussed at point "2)" below.

The computational bottleneck comes from initializing the array `tab`, filling its $2^n + 1$ cells with zeros. We need to dive into this "implementation detail" to answer the (theoretical) question from the title. The right program has **not really** initialized the memory area associated with the array `tab` (about 4GB) in one millisecond. There may be several explanations :

1) The Operating System (OS) initialized the memory beforehand. Some operating systems use their idle time to fill in all available memory with zeros. The goal is to be sure that the data of a terminated process cannot be accessed later by another process. After investigation, it turned out that my OS does not have this feature. However, this feature does exist in other operating systems. This brings us to a question that I don't think theory alone will ever solve :

What is the complexity of an algorithm if we know that a part of its code is (anticipated by the OS and) executed before the algorithm starts ?

2) The OS might have performed a kind of lazy initialisation. It is well-known that many processes can request a memory area without needing it right away. The OS has given just a form of promise that it can provide initialized memory. When the process really needs to use a memory cell, the processor triggers a page fault and the kernel steps in to allocate real RAM. [1]

For $n \geq 31$, my OS does not want to commit to an impossible memory allocation. But one can implement a different memory management system that may return a larger virtual memory block for $n \geq 31$, knowing that the program will never used more than $\mathcal{O}(n)$ cells. This gives $\mathcal{O}(n)$ page faults to handle.

3) The compiler might have been very clever, because I can not find the same speed if I replace `g++` with `gcc`. The performance also degrades if we replace `calloc` with the C++ method of initialization, i.e., with `new int[z]()`. The situation gets even more complex if we start using compiler optimization options. Like any truth, it is possible to go deeper and deeper and find thousands layers. But this is outside the scope of this 2 pages long communication.

However, it is clear to me that I cannot answer the question in the title with a simple "Yes" or "No". It's more complex and nuanced than that.

Conclusion and prospects

Let's take a higher standpoint : the problem is that many aspects of real life evade the understanding of those who do not go to the root of most questions, only reasoning on the basis of theory or only on the basis of summarized information. This may sometimes happen even for some referees (for journals, PhD thesis committees, etc.). I have uploded at `youtu.be/YuKnE6zH-J8` a demonstration of a program that can become 7-8 times faster if one adds a simple call `cplex.end()` at the end of a function. Even by assuming the world is filled only with honest people, it is easy to deceive oneself in such a case and to think that it is a theoretical contribution (e.g., a new valid inequality) that has made the algorithm 7-8 times faster – when it is only a simple `cplex.end()` forgotten by a collaborator.

I have already covered this subject in a series of talks I present at Roadef since 2019. The first paper starts with this very connection between theoretical and real life execution speed :

> *Whether we like it or not, the speed of any algorithm that actually runs on a physical machine depends on a constant complexity factor K closely related to the quality of the implementation. After ten years of talks at various CS conferences, I finally realized that I always finish all my speeches with a line like : "And finally, I will spare you the implementation details because I prefer to focus on the essentials and on the high level concepts". Despite some merits, this approach is far too simplistic. It is likely to divert from the truth many readers of average attention. Like a backdoor in a software program, an unspoken consequence is that the (constant complexity factor associated with) the quality of the implementation has no relevant impact on the total computational cost and can be ignored. This constant complexity factor K is also invisible when calculating theoretical complexities such as $O(n^2)$, $O(n^2 \log n)$, etc. But imagine a salesman saying that the cost to pay is 1000€ multiplied by a some constant factor K=3 or K=6 which is not relevant in absolute terms. With all due respect to all rich people for whom 1000€ or 3000€ is more or less the same, I would not let myself be convinced so easily. I am even very grateful to my fate that it spared me the emptiness of living a life in such a great luxury.*

The 2021 paper concludes by citing Christopher Strachey (Oxford, vers 1970) :

> *It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing is unsound and clumsy because the people who do it have not any clear understanding ofthe fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen.*

This communication is just another brick inside a larger work, composed of several papers (Roadef or other) uploaded at `cedric.cnam.fr/~porumbed/soft/`. If I took the pain to write a dozen of pages, I did not do it simply to reinforce well known ideas using more original arguments. The goal is to challenge several fallacious presuppositions/prejudices that have had a very negative effect on Computer Science, by pushing people to disconnect theory from practice.

1. See the answer of Dietrich Epp on the question `stackoverflow.com/questions/2688466/why-mallocmemset-is-slower-than-calloc` or also `cs.stackexchange.com/questions/83834/what-is-the-time-complexity-of-memory-allocation-assumed-to-be`