Is there any O(2 n) algorithm your OS can run in almost no time ?

Daniel Cosmin Porumbel

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*
>
> La vérité n'est pas une fille qui saute au cou de [celui] qui ne la désire pas ; c'est plutôt une fière beauté, à qui l'on peut tout sacrifier, sans être assuré pour cela de la moindre faveur.
>
> *Arthur Schopenhauer, 1844*

Le piège

Si vous pensez que la réponse à la question du titre doit être « Non, c'est complètement impossible ; ça serait une hérésie contre le bon sens de l'informatique fondamentale ! », pensez aussi à ne pas oublier la devise de cette communication. D'ailleurs, les deux citations signifient la même chose : la réalité et la pratique peuvent être parfois si complexes qu'elles échappent aux systèmes de valeurs et aux axiomatisations avec lesquelles la théorie nous a habitué.

Les deux programmes ci-après exécutent les mêmes instructions (pour faire de la programmation dynamique). Le code a été aligné pour voir clairement la correspondance ligne par ligne.

```
#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
}
```

Les deux programmes ont été compilé avec `g++` sans option d'optimisation. Le code à gauche prend environ 0.75 secondes pour $n = 28$, 1.5 secondes pour $n = 29$ et 3 secondes pour $n = 30$. Cela suit donc une loi exponentielle qui correspond à la complexité attendue de $\mathcal{O}(2^n)$. Mais – croyez le, croyez le pas – le programme à droite prends environ 0.001 secondes pour $n = 30$ sur ma machine. Le cas $n \geq 31$ échoue dans les deux cas ; cela est discuté au point 2) ci-dessous.

Le goulet d'étranglement vient de l'initialisation du tableau `tab` de $2^n + 1$ cases. On doit aborder ce « détail d'implémentation » pour répondre à la question (théorique) tu titre. Le programme à droite n'a **pas vraiment** initialisé la zone de mémoire associée au tableau `tab` (environ 4GB) en une milliseconde. Il peut avoir plusieurs explications :

1) Le système d'exploitation (OS) a pu faire l'initialisation mémoire en préalable. Certains OS utilisent leur « idle time » pour remplir toute la mémoire disponible avec des zéros. Cela sert à garantir que les données d'un processus terminé ne peuvent pas être récupérées plus tard par un autre processus. Après investigation, mon OS n'a pas fait cela, mais cette fonctionnalité existe dans d'autres OS. Cela nous amène à une question que la théorie seule ne pourra jamais résoudre à mon avis : *quelle est la complexité d'un algorithme si on sait qu'une partie de son code est (anticipé par l'OS et) exécuté en préalable avant son lancement ?*

2) L'OS a pu faire une initialisation qu'on pourrait qualifier de « lazy ». C'est connu que certains processus demandent de la mémoire qu'ils n'utilisent pas toute de suite. Ainsi, l'OS a donné juste une forme de promesse de mémoire initialisée. Dès que le processus a réellement besoin d'utiliser une case mémoire, le processeur déclenche un « page fault » et le noyau intervient pour attribuer de la vraie RAM. [1] Si l'appel `calloc(...)` échoue pour $n \geq 31$, c'est parce que l'OS ne veut pas s'engager sur une allocation de mémoire qu'il ne peut pas assurer. *Mais il est possible de concevoir un système de gestion de mémoire qui pourrait donner de la mémoire fictive même pour $n \geq 31$, étant donné qu'on sait que le programme va toucher au maximum $\mathcal{O}(n)$ cases. On obtient un nombre de "page faults" à gérer de maximum $\mathcal{O}(n)$.*

3) Le compilateur a pu être très astucieux, car on ne retrouve pas la même vitesse si on remplace `g++` avec `gcc`. La performance se dégrade aussi si on remplace `calloc` avec la méthode C++ d'initialisation, c.à.d., avec `new int[z]()`. La situation se complexifie si on commence à utiliser des options d'optimisation. Comme toute vérité, il est possible d'éplucher cette question comme un chou avec milles feuilles. Mais je peux pas aller plus loin dans un résumé de 2 pages.

Nonobstant, pour moi il est clair que je ne peux par répondre à la question du titre par un simple « Oui » ou par un simple « Non ». C'est plus complexe et nuancé que ça.

Conclusion et perspectives

Prenons une perspective plus haute pour saisir le problème plus général : *beaucoup d'aspects de la vie réelle échappent trop facilement à ceux qui raisonnent que sur la base de la théorie ou que sur la base d'informations synthétiques, y compris à certains rapporteurs (des revues, jurys de thèse, etc).* J'ai déposé à `youtu.be/YuKnE6zH-J8` une démonstration d'un programme qui peut devenir 7-8 fois plus rapide si on ajoute un simple appel `cplex.end()` à la fin d'une fonction. Même en supposant un monde que avec des gens très très honnêtes, il est facile de se tromper soi même dans un tel cas et de penser que c'est une contribution théorique (ex, une nouvelle inégalité valide) qui a rendu l'algorithme 7-8 fois plus rapide – alors que cela ne vient que d'un simple `cplex.end()` oublié par un collaborateur.

J'ai déjà abordé ce sujet dans une suite de travaux que je présente à la Roadef depuis 3 ans. Le premier papier (Roadef 2019) commence avec cette question de vitesse d'exécution.

> La vitesse d'exécution de tout algorithme pratique dépend d'un facteur constant de complexité γ étroitement lié à la qualité de l'implémentation. Je présente des communications à la Roadef depuis dix ans et je viens seulement de réaliser que tous mes exposés se terminent invariablement avec une phrase comme « Pour finir, je vais vous épargner les détails d'implémentation parce que je préfère me focaliser sur l'essentiel et sur les concepts de haut niveau ». Malgré ses mérites, cette phrase est trop simpliste. Elle peut détourner le lecteur d'attention moyenne de certaines vérités. Comme une porte dérobée dans un logiciel, une conséquence inavouée est la suivante : le cout en temps de calcul est très peu impacté par (le facteur constant de complexité associé à) la qualité de l'implémentation et on peut l'ignorer. Ce facteur constant γ est aussi invisible lorsqu'on calcule des complexités théoriques comme $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, etc. Mais imaginez un vendeur qui dirait que le coût à payer serait de 1000€ multiplié par un facteur constant $\gamma = 3$ ou $\gamma = 6$ qui n'a aucune importance dans l'absolu. Si cela puisse être vrai pour certains, je suis heureux que mon destin m'a épargné le vide d'une existence dans un luxe si démesuré.

Le papier 2021 se conclue avec une citation de 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.

Ce résumé s'inscrit donc dans un travail plus conséquent, composé de plusieurs papiers (Roadef ou autre) déposés à `cedric.cnam.fr/~porumbed/soft/`. Si j'ai décidé de prendre la plume pour y écrire plusieurs pages, ce n'est pas pour renforcer des idées bien connues en utilisant une nouvelle phraséologie ou des arguments plus originaux. C'est plutôt pour remettre en question quelques présupposés très fallacieux et très répandus en RO et informatique.

1. Voir la réponse de Dietrich Epp sur le forum de questions `stackoverflow.com/questions/2688466/why-mallocmemset-is-slower-than-calloc` ou aussi `cs.stackexchange.com/questions/83834/what-is-the-time-complexity-of-memory-allocation-assumed-to-be`