EPFL - Information, Calcul et Communication - Partie théorique

Série 4: Solutions

1 Une meilleure version du tri par fusion?

En pratique, il peut effectivement être une bonne idée de trier une liste au moyen de l'algorithme de tri par fusion en divisant la liste en plus de deux parties à chaque fois. Mais en théorie, on ne gagne pas grand chose: le nombre de niveaux à "descendre" pour atteindre des listes de taille 1 est donné ici par $\log_4(n)$, qui est effectivement plus petit que $\log_2(n)$, mais pas de beaucoup! En effet, $\log_4(n) = \log_2(n)/\log_2(4) = 0.5 \cdot \log_2(n)$; la complexité temporelle de l'algorithme ne s'améliore donc pas et vaut toujours $\mathcal{O}(n \cdot \log_2(n))$.

Notez aussi qu'en pratique, on obtient certes un algorithme avec moins de niveaux, mais l'algorithme de fusion de 4 listes est plus complexe que celui de la fusion de 2 listes (même s'ils ont la même complexité temporelle en théorie pour des listes avec un même nombre total d'éléments).

2 Nombres de Fibonacci

a)

Fibonacci

entrée : *nombre entier positif* n
sortie : *nombre entier positif* $F(n)$

Si $n = 1$ *ou* $n = 2$
  Sortir: 1

Sortir: **Fibonacci**$(n-1) +$ **Fibonacci**$(n-2)$

b) Cet algorithme, aussi simple et élégant soit-il, est malheureusement horriblement chronophage! On peut voir par exemple que pour calculer $F(n)$, cet algorithme recalcule (inutilement) au moins 2 fois $F(n-2)$, donc au moins 4 fois $F(n-4)$, et ainsi de suite de 2 en 2 jusqu'à $F(2)$ ou $F(1)$. Le nombre d'opérations effectuées est donc supérieur ou égal à $2^{n/2} = (\sqrt{2})^n \simeq 1.41^n$, et il en va de même de la complexité temporelle de l'algorithme. De l'autre côté, on peut voir de manière similaire que l'algorithme ne calcule pas plus de 2 fois $F(n-1)$, pas plus de 4 fois $F(n-2)$, et ainsi de suite, d'où on déduit que la complexité temporelle de l'algorithme est entre $\Theta(1.41^n)$ et $\Theta(2^n)$.

Note: De manière plus précise, on peut voir que le nombre d'opérations effectuées par l'algorithme pour calculer $F(n)$ vaut... également $F(n)$! Et une analyse plus poussée montre que lorsque n est grand, $F(n) = \Theta(\phi^n)$, où ϕ est le nombre d'or ($\simeq$1.618).

c) Pour éviter que l'algorithme effectue tous ces calculs inutiles, on peut utiliser le principe de mémoïsation vu au cours. Mais ici, on peut aussi écrire directement une version itérative et beaucoup plus efficace de l'algorithme:

Fibonacci itératif

entrée : *nombre entier positif* n
sortie : *nombre entier positif* $F(n)$

$x \longleftarrow 1$
$y \longleftarrow 1$
Pour i *allant de* 1 *à* $n-2$
  $z \longleftarrow x + y$
  $y \longleftarrow x$
  $x \longleftarrow z$
Sortir : x

3 Rendu de pièces de monnaie

Avec P_1, le rendu glouton n'atteint pas le montant exact et rend 1 fr. + 1 fr. + 40 cts, alors qu'un rendu exact serait obtenu avec 70 cts + 70 cts + 40 cts + 40 cts + 40 cts.

Avec P_2, le rendu glouton n'atteint pas le montant exact et rend 1 fr. + 1 fr. + 50 cts, alors qu'un rendu exact serait obtenu avec 70 cts + 70 cts + 70 cts + 50 cts.

Avec P_3, le rendu glouton atteint le montant exact et rend 1 fr. + 1 fr. + 40 cts + 10 cts + 10 cts, alors qu'un rendu optimal (avec moins de pièces) serait obtenu avec 1 fr. + 1fr. + 30 cts + 30 cts.

Avec P_4, le rendu glouton atteint le montant exact avec un nombre minimal de pièces et rend 1 fr. + 1 fr. + 40 cts + 20 cts.

4 Un autre algorithme mystère

Tout d'abord, il convient d'observer que cet algorithme se termine pour toute valeur d'entrée $n > 100$ (la sortie valant simplement $n - 10$ dans ce cas, donc 91 si $n = 101$, 92 si $n = 102$, etc.). Voilà déjà une condition de terminaison un peu étrange.

Mais le plus étrange est que pour *tout* nombre n entre 1 et 100 en entrée, la sortie de cet algorithme est le nombre 91 (ce qui lui vaut le surnom de "fonction 91")! Voici quelques exemples pour vous en convaincre:

Si $n = 100$, alors $\mathbf{algo}(100) = \mathbf{algo}(\mathbf{algo}(111)) = \mathbf{algo}(101) = 91$.

Si $n = 99$, alors $\mathbf{algo}(99) = \mathbf{algo}(\mathbf{algo}(110)) = \mathbf{algo}(100) = 91$, par ce qu'on vient de calculer ci-dessus.

Si $n = 98$, alors $\mathbf{algo}(98) = \mathbf{algo}(\mathbf{algo}(109)) = \mathbf{algo}(99) = 91$, encore une fois, par ce qu'on vient de calculer ci-dessus.

Et ainsi de suite jusqu'à $n = 1$!

5 Pour le plaisir: deviner une date d'anniversaire* (©FSJM, 2017)

L'algorithme pour résoudre ce problème est le suivant: il y a dix possibilités au départ:

15, 16 et 19 mai; 17 et 18 juin; 14 et 16 juillet; 14, 15 et 17 août

et le but est de réduire ce nombre à une en utilisant les informations qui sont fournies au fur et à mesure (qui sont en effet des informations précieuses, mêmes si elles n'en ont pas l'air de prime abord).

Première information: Manon, qui connaît le mois, dit à Julie, qui connaît le jour: "Je ne sais pas quelle est la date, mais je sais que tu ne le sais pas non plus".

De là, on déduit que le mois de la naissance d'Anne ne peut pas contenir des jours qui à eux seuls permettraient à Julie de déterminer la date: ces jours sont le 18 et 19, donc les seuls mois possibles sont juillet ou août.

Deuxième information: Julie répond à Manon: "Je ne savais pas quelle était la date, mais maintenant je le sais".

Donc Julie, avec l'information additionnelle que le mois est juillet ou août, peut déduire quelle est la date: ceci exclut que le jour soit le 14, sinon elle ne ne pourrait pas faire cette déduction. Le jour est donc le 15, le 16 ou le 17.

Troisième information: Manon conclut: "Alors je sais aussi quelle est la date".

Manon, qui n'a a priori que l'information du mois (et l'information ci-dessus que le jour est le 15, le 16 ou le 17), ne peut conclure que si il n'y a qu'une date possible dans le mois en question: elle conclut donc (ainsi que tout le monde autour d'elle) que la date est le 16 juillet.