Le problème du manteau

On vous donne un morceau de tissu de dimensions entières $X \times Y$. Vous avez un ensemble de n patrons, chacun nécessitant un morceau de tissu de dimensions entières $x_i \times y_i$ pour être reproduit. La production d'une copie du patron i vous rapportera c_i. Il est possible de vendre autant de copies d'un même patron que souhaité. On dispose d'une machine qui peut couper tout morceau de tissu en deux morceaux, soit horizontalement soit verticalement. On supposera que $x_i \leq y_i$ pour tous les objets et que pour tout morceau de tissu, à tout instant, $X \leq Y$. On notera $P(x, y)$ le profit optimal d'un morceau de taille x par y. On cherche un algorithme qui explique comment maximiser notre profit.

1. En supposant d'abord que les dimensions du morceau à couper sont strictement plus grandes que le plus large des objets, de sorte qu'on doive nécessairement faire au moins une coupe, trouver une relation de programmation dynamique convenable pour la première coupe.
2. Revenons maintenant sur notre hypothèse sur les dimensions du morceau à couper. Définir une matrice de sorte d'initialisation pour résoudre le problème.
3. Implémenter l'algorithme dans le langage de votre choix (Python, C, OCamL)

Solution: Le problème du manteau

1. On suppose ici que les dimensions du morceau à couper sont strictement plus grandes que celles du plus large des objets. On a donc nécessairement une coupe à effectuer. Deux situations peuvent se produire.
   (a) Ou bien la solution optimale produit une coupe selon l'axe X, à k unités de distance d'un bord : on se retrouve avec deux morceaux de tissu A et B de dimension X par k et X par $Y-k$.
   (b) Ou bien la solution optimale demande une coupe selon l'axe Y, à k unités de distance d'un bord : on se retrouve avec deux morceaux de tissu A et B de dimension k par Y et $X-k$ par Y.

   Le profit qu'on fait dans chaque cas est $P(A)+P(B)$ ou et $P(A)$ et $P(B)$ doivent correspondre à des solutions optimales puisque sinon on aurait pas une solution optimale au total. Comme on ne sait pas pour l'instant dans quel direction couper, on considère toutes les possibilités et donc on écrit :

$$P_x(x,y) = \max_{1\leq k\leq \left\lfloor \frac{x}{2}\right\rfloor} P\left(x-k,y\right)+P(k,y))$$

$$P_y(x,y) = \max_{1\leq k\leq \left\lfloor \frac{y}{2}\right\rfloor} P\left(x,y-k\right)+P(x,k))$$

   pour représenter le profit optimal qu'on obtient en coupant selon X et Y respectivement. On n'est intéressés qu'en le maximum de ces deux profits.
2. En revenant maintenant sur notre hypothèse sur la taille du tissu, c'est à dire en supposant que la solution optimale n'implique pas de coupe, on a deux situations possibles :
   (a) Ou bien le tissu est exactement de la taille du patron i et le gain est c_i
   (b) Ou bien le tissu n'est de la taille d'aucun patron et le gain est 0

   On peut donc définir une matrice P_0 de sorte qu'on ait toujours :

$$P(x,y) = \max\left(P_x(x,y),P_y(x,y),P_0(x,y)\right)$$

   On a alors :

$$P_0(x,y) = \begin{cases} c_i & \text{si } x=x_i \wedge y=y_i \\ 0 & \text{sinon} \end{cases}$$

Révisions de Sup
Matthieu Boyer
Planche B

Khôlle MPI
16/09

Fibonacci, mais vite.

Dans cet exercice on va chercher une méthode efficace pour calculer F_n.

1. Montrer que si $n \geq 1$:
$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$
2. Montrer que $\mathcal{O}(\log n)$ multiplications matricielles suffisent à calculer F_n
3. Montrer que chaque calcul implique $\mathcal{O}(n)$ bits.
4. Conclure que le temps de calcul de la procédure est en $\mathcal{O}\left(n^2 \log(n)\right)$.
5. En supposant que 2 entiers sur n bits peuvent être multipliés $\mathcal{O}(M(n))$, peut on prouver que la procédure nécessite un temps en $\mathcal{O}(M(n))$?

Solution: Fibonacci, mais vite.

1. Classique, on sait que $F_{n+1} = F_n + F_{n-1}$ pour $n \geq 1$. On vérifie aisément que :

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = M \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

   On conclut par récurrence.

2. En utilisant l'algorithme d'exponentiation rapide, on a le résultat.

3. Par la relation de récurrence définissant la suite de Fibonacci, celle-ci étant croissante :

$$F_{n+1} \leq 2F_n$$

   Par récurrence on a donc

$$F_n \leq 2^n$$

   et donc F_n s'écrit sur au plus n bits. En particulier, chaque terme de M^n s'écrit sur au plus n bits. Donc chaque opération implique au plus $n+1$ bits (puisque le produit de deux nombres sur n bits s'écrit sur au plus $n+1$ bits).

4. On sait que naïvement le produit de deux entiers sur n bits se calcule en $\mathcal{O}(n^2)$ et que la somme de deux tels nombres se calcule en $\mathcal{O}(n)$ bits. Puisqu'on a toujours un nombre fini constant d'opérations (4 produits et 2 additions) par produit matriciel, on a bien un temps de calcul général en $\mathcal{O}(n^2 \log(n))$ au plus.

5. On ne peut pour l'instant pas conclure sur le résultat puisque cela dépend de $M(n)$, et on peut s'attendre à ce que ça soit quadratique mais plus que linéaire. Supposons que $M(n) \leq \alpha n^c$ pour $c > 1$ une constante. Alors :

$$\begin{aligned} T(n) =& T\left(\frac{n}{2}\right) + \alpha n^c \\ =& T\left(\frac{n}{4}\right) + \alpha\left(n^c + \left(\frac{n}{2}\right)^c\right) \\ =& \dots \\ =& T(1) + \alpha n^c \left(\sum_{i=0}^{\lfloor \log_2(n) \rfloor - 1} \frac{1}{2^i}\right)^c \end{aligned}$$

   La dernière sommation peut contenir un nombre arbitraire de termes mais ne peut jamais atteindre la valeur de 2. Donc $T(n) \leq T(1) + 2^c \alpha n^c$, ce qui montre que $T(n) = \mathcal{O}\left(M(n)\right)$ pour tout $c > 1$.

Algorithme de vote de Boyer-Moore

Supposons qu'on se trouve dans un centre de convention rempli de votants, chacun portant une une pancarte avec le nom de son candidat. On va chercher le candidat majoritaire le plus vite possible en gardant ça un espace constant.

1. Proposez un algorithme naïf en $\mathcal{O}(n \times \log n)$.

On va chercher un algorithme plus efficace en $\mathcal{O}(n)$.

2. Proposer un algorithme qui résoud le problème en $\mathcal{O}(n)$. On effectuera deux passes : une pour trouver un candidat possible et une pour vérifier que le candidat convient. On pourra par exemple utiliser un compteur associé à une valeur qui augmente en voyant cette valeur et décroit en en voyant une autre.

Solution: Algorithme de vote de Boyer-Moore

Pour l'algorithme naïf, il est clair qu'un tri On va prouver la correction de l'algorithme suivant :

Algorithme 1 Algorithme de Vote de Boyer-Moore

```
function BMV(T)
    compteur = 0
    candidat = T[0]
    for i = 0; i++; i < |T| do
        if T[i] == candidat then
            compteur++
        else
            compteur--
        end if
        if c == 0 then
            candidat = T[i]
            compteur = 1
        end if
    end for
    compteur = 0
    for i = 0; i++; i < |T| do
        compteur+ = T[i] == candidat
    end for
    if compteur > |T| /2 then return candidat
    else return none
    end if
end function
```

Tout d'abord, si la liste est de longueur 1, on renvoie le seul élément dans la liste et le cas de base est bon.

Ensuite, supposons que l'algorithme fonctionne correctement pour des listes de taille n. Considérons une liste de taille $n+1$.

Pour notre premier cas, faisons comme si le premier élément n'était pas un élément majoritaire. Alors, à un moment donné dans cette liste, le compte tombera à 0 et un nouveau candidat sera sélectionné. À ce stade, notre algorithme ne peut plus être différencié du même algorithme commencé à cet index dans le tableau. De plus, l'élément majoritaire dans le reste de notre liste est évidemment le même que l'élément majoritaire dans la liste d'origine. En effet, si notre élément majoritaire apparaissait k fois à l'origine dans une liste de n éléments, alors dans notre sous-liste de n - i éléments, notre élément majoritaire apparaît k - i fois. Évidemment, $k > \frac{n}{2} \implies k - i > \frac{n-i}{2}$. Ainsi, par notre hypothèse inductive, nous trouverons l'élément majoritaire.

Pour le deuxième cas, où le premier élément est notre élément majoritaire, si la liste entière est traitée sans réinitialisation, alors évidemment l'élément majoritaire a été renvoyé. S'il a été réinitialisé, nous pouvons alors utiliser exactement le même raisonnement que précédemment pour montrer que la majorité dans la sous-liste à traiter est la même que la majorité d'origine, et nous renverrons donc toujours l'élément correct.