

Algorithmique — M1

Partiel du 17 novembre 2009 - corrigé

On applique les cours

Exercice 1 – Récurrence

Étant donné que

$$T(n) = 2T(\lfloor n/4 \rfloor) + 3 \log n$$

trouvez le comportement asymptotique de $T(n)$. Justifiez votre réponse.

Correction. On utilisera le Master Theorem avec les coefficients $a = 2$, $b = 4$ et la perturbation $f(n) = \log n$. On trouve d'abord l'exposant :

$$k = \log_4 2 = 1/2.$$

On voit que la perturbation est petite $\log n = o(n^{k-\varepsilon})$, en choisissant, par exemple $\varepsilon = 0,1$. Effectivement $\log n = o(n^{0.4})$ puisque le logarithme croît moins vite que n'importe quelle puissance positive.

Par Master Theorem on obtient donc :

$$T(n) = \Theta(n^{1/2}) = \Theta(\sqrt{n}).$$

Exercice 2 – Puissance

On veut calculer x^{13} en faisant aussi peu de multiplications que possible.

1. Expliquez l'algorithme de cours pour ce problème.
2. Trouvez une chaîne de multiplications pour calculer x^{13} . Combien de multiplications faut-il ?
3. Est-il possible d'obtenir x^{13} en moins de multiplications ?

Correction. L'algorithme diviser-pour-régner de cours calcule x^n comme suit :

– si $n = 0, 1, 2$ on renvoie $1, x, x * x$ respectivement ;

– sinon si n est pair alors on calcule d'abord $y = x^{n/2}$, et ensuite $x^n = y * y$;

– sinon (si n est impair) alors on calcule d'abord $y = x^{\lfloor n/2 \rfloor}$, et ensuite $x^n = y * y * x$;

Pour $n = 13$ ça donne : $x^{13} = y * y * x$ avec $y = x^6$. A son tour $y = z * z$ avec $z = x^3$. Finalement $z = x * x * x$. En tout on a 5 multiplications.

J'ai trouvé plusieurs solutions différentes de celle-là mais toujours en 5 opérations. Exemple :

$$y = x^2 = x * x; z = x^3 = y * x; v = x^5 = y * z; x^{13} = v * v * z.$$

Exercice 3 – Antarctide

L'explorateur dispose pour le chauffage de 20 kg de bois, 10 kg d'essence, 12 kg de charbon, et 300 kg d'éthanol. Un kg de bois produit 15 Mj de chaleur, d'essence 47, de charbon 35, et d'éthanol 30. L'explorateur peut porter jusqu'à 35kg de combustible. Quelle quantité de chaque produit doit-il prendre pour se procurer le maximum de chaleur ?

1. C'est une instance d'un problème classique vu en cours. Lequel ?
2. Expliquez l'algorithme de cours pour ce problème. (inutile de prouver sa correction)
3. Appliquez cet algorithme et trouvez la solution.

Correction. C'est une instance de sac-à-dos fractionnaire (avec le pouvoir calorifique qui remplace le coût dans le problème). L'algo glouton de cours est comme ceci :

- Classer les produits dans l'ordre décroissant de coûts par kilo (ici pouvoir calorifique).
- Prendre autant de premier produit que possible (jusqu'au remplissage de sac ou épuisement de stock).
- Pareil pour le deuxième produit etc. . .

Cela donne les produits classés comme ceci : essence, charbon, ethanol, bois.

Et le contenu de sac : essence - tous les 10 kg ; charbon - tous les 12 kg ; ethanol - 13 kg. Le sac est plein

On invente des algorithmes

Exercice 4 – Méthode imposée

On cherche la somme d'un tableau B de n éléments entiers.

1. Écrivez un algorithme de type diviser-pour-régner qui résout ce problème.
2. Analysez sa complexité.
3. Comparez-la avec celle de l'algorithme naïf vu en L1.

Correction. On définit la fonction Sum(B,i,j) qui est la somme des éléments de B entre les positions i et j. Ainsi la valeur recherchée est Sum(B,1,n). On va calculer Sum(B,i,j) en découpant le tableau B[i..j] en deux moitiés ; calculant les sommes pour chaque moitié ; et additionnant ces deux sommes. Ceci donne le code suivant :

```
fonction Sum(B,i,j)
  si i=j
    retourner B[i]
  m= (i+j) div 2
  gauche=Sum(B,i,m)
  droite=Sum(B,m+1,j)
  retourner (gauche+droite)
```

On a la récurrence suivante sur la complexité de cet algo :

$$T(n) = 2 * T(n/2) + O(1)$$

En appliquant le Master Theorem on voit que l'exposant est $k = \log_2 2 = 1$, que la perturbation $O(1) = O(n^0)$ est petite, et donc

$$T(n) = \Theta(n).$$

L'algo naïf de L1 est bien sûr le suivant :

```
fonction SumL1(B,n)
  s=0
  pour i de 1 à n
    s=s+B[i]
  retourner s
```

Sa complexité est $O(n)$ parce qu'il y a juste une boucle "pour" répétée n fois. On constate que l'algo diviser-pour-régner n'est pas plus efficace que l'algo naïf pour ce problème.

Exercice 5 – Deux couleurs

Etant donné un graphe non-orienté et connexe $G = (V, E)$, on cherche à le colorier en noir et blanc (de manière que deux sommets adjacents ne soient jamais d'une même couleur).

1. Écrivez un algorithme glouton qui résout ce problème.
2. Analysez sa complexité.
3. Justifiez sa correction.
4. Pourquoi on ne fait pas d'algorithme glouton quand il y a plus de couleurs ?

Correction. L'observation principale est la suivante : si un sommet est noir, alors tous ses voisins doivent être blancs ; et vice versa. Ainsi on fixe la couleur d'un sommet (par exemple noir) ; on colorie tous ses voisins en blanc ; tous leurs voisins en noir etc (en fait c'est un parcours en largeur). Soit on arrive à colorier ainsi tout le graphe, soit on découvre une anomalie, et dans ce cas-là le 2-coloriage est impossible.

Voici le pseudo-code (en supposant que le graphe est représenté par les listes d'adjacence, est $Adj(v)$ est la liste des voisins du sommet v). Tous les sommets sont gris au départ, et on les colorie en parcourant le graphe. W est la file d'attente.

```

pour tout v
    v.couleur=gris
s= un sommet quelconque
W= file vide de sommets
s.couleur=noir
W.enfiler(s)
tant que W non vide
    v= W.défiler
    maCouleur=v.couleur
    autreCouleur=inverser(maCouleur)
    pour tout u dans Adj(v)
        si u.couleur =maCouleur
            Imprimer("coloriage impossible !")
            exit
        si u.couleur = gris
            u.couleur=autreCouleur
            W.enfiler(u)
Imprimer("coloriage trouvé !")

```

La complexité est la même que pour le parcours en profondeur $O(|V| + |E|)$.

Si l'algorithme dit "coloriage trouvé", alors tous les sommets sont coloriés (il ne reste plus de sommets gris), parce que le BFS (parcours en profondeur) dans un graphe connexe non-orienté visite tous les sommets. Les voisins de chaque sommet blanc sont noirs et vice versa grâce à la boucle principale (s'il y avait deux voisins de même couleur, ça serait détecté au moment où le deuxième voisin sort de la file). On a donc démontré que si l'algo dit "coloriage trouvé", alors c'est vraiment le cas.

Pour la réciproque, il reste à prouver, que si le coloriage est possible l'algorithme le trouvera. Effectivement, si le coloriage est possible, alors il existe un coloriage où le sommet s est noir (sinon on inverse la couleur de tous les sommets). Dans ce coloriage tous les voisins de s doivent être blanc, tous leurs voisins noirs etc (c-à-d tous les sommets à une distance paire de s sont noirs, tandis que les sommets à une distance impaire sont blanc). Mais c'est exactement le coloriage calculé par l'algorithme !

Malheureusement cette méthode ne marche pas pour 3 couleurs (ou plus) puisque la couleur d'un sommet ne détermine pas les couleurs de ses voisins comme c'était le cas pour 2 couleurs.

Exercice 6 – Un problème NP-complet

Dans un graphe on appelle un sous-ensemble S de sommets *stable* s'il n'existe pas d'arête du graphe qui relie deux sommets de S .

Étant donné un graphe non-orienté $G = (V, E)$ (de N sommets) représenté par une matrice d'adjacence $M[i, j]$ (de taille $N \times N$) et un entier M on cherche un ensemble stable de C sommets. Dans cet exercice il faut trouver un algorithme retour-arrière qui résout ce problème. On va appeler une solution partielle un tableau d'entiers $[i_1, \dots, i_k]$ dans l'ordre croissant tel que les sommets $[v_{i_1}, \dots, v_{i_k}]$ forment un stable.

1. Écrivez une fonction booléenne $test(B, k, M, N)$ qui teste est-ce que le tableau $B[1..k]$ est une solution partielle pour un graphe représenté par un tableau (matrice d'adjacence) $M[N, N]$.
2. Comment trouver une solution partielle de taille 0 ? Comment à partir d'une solution partielle de taille k passer à ses extensions de taille $k + 1$? Comment dire est-ce qu'on a déjà trouvé le stable de taille C ?
3. Écrivez un algorithme retour-arrière de recherche d'un stable de taille C .

4. Estimez la complexité de votre algorithme.

Correction. Pour le test on suppose que $B[k]$ est un tableau croissant de k entiers entre 0 et N . Il reste à tester l'absence d'arêtes. Dans la fonction ci-dessus je suppose que B, M, N sont des variables globales.

```
Boolean fonction test(k)
    pour i de 1 à k-1
        pour j de i+1 à k
            si  $M[B[i], B[j]] = 1$ 
                retourner faux
    retourner vrai
```

–La seule solution partielle de taille 0 est le tableau vide.

–Si on a une solution partielle $B[1..k]$ pour l'étendre il faut choisir un nouveau sommet v supérieur à $B[k]$ (attention, pour $k=0$ cette contrainte disparaît) ; et l'ajouter à la place $B[k+1]$. Il faut que le B ainsi obtenu passe le test ci-dessus.

–Une solution partielle $B[1..k]$ est un stable recherché si $k = C$.

Ceci mène à l'algorithme retour-arrière suivant

```
fonction TrouverStable(k)
    si  $k=C$ 
        imprimer  $B$  ; arrêter
    si  $k=0$ 
        alors début = 1
        sinon début =  $B[k]+1$ 
    pour  $v$  de début à  $N$ 
         $B[k+1] = v$ 
        si test( $k+1$ )
            TrouverStable( $k+1$ )
```

Le programme principal appelle

TrouverStable(k)

Au pire cas et algorithme essaye tous les $2^{|V|}$ sous-ensembles des sommets, chaque essai prend $k^2 = O(|V|^2)$ opération (à cause du test). Ce qui donne une estimation de complexité $O(|V|^2 2^{|V|})$ qui est exponentielle. On peut gagner un peu sur le test, mais tous les algorithmes connus pour ce problème sont exponentiels.