Université de Lille

FST- Département Informatique

ACT
Durée 3h
Tous documents autorisés

2017–2018
Master1 Informatique
14 juin 2018

Les algorithmes seront écrits en pseudo-langage ou dans un langage de votre choix. La clarté de vos réponses sera prise en compte. Le barème est donné à titre indicatif et est un barème "minimal".

Exercice 1 Connaissances de base - *6 pts*

Q 1. Pour chacun des algorithmes suivants, dire si sa complexité (temporelle) dans le pire des cas est en $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^2)$ ou $\Theta(n^3)$, en justifiant très brièvement (en supposant qu'on est dans le cadre du coût uniforme, i.e. en ne prenant pas en compte la taille des opérandes pour les opérations élémentaires) :

```
int T1(int n){
  int c=0;
  for (int i=1; i < n ; i++)
    for (int j=n+i; j >i ; j--)
    c++;
  return c;}
```

$\Theta(n^2)$

```
int T3(int n){
  if (n <= 0) return 1;
  else return 1+ T3(n-5);}
```

$\Theta(n)$

```
int T2(int n){
  int c=0;
  int b= n*n*n;
  for (int i=1;i<b;i=3*i) c++;
  return c;}
```

$\Theta(\log n)$

```
int T4(int n){
  if (n <= 0) return 1;
  else return T4(n/3)+T4(n/3)+T4(n/3);}
```

$\Theta(n)$

Le Master Theorem -version allégée- donne $O(n)$. L'arbre des appels est un arbre ternaire complet de profondeur $\log_3 n$, donc il y a de l'ordre de $\Theta(n)$ nøeuds et la complexité associée à chaque noeud est en temps constant.

Donc, la complexité est $\Theta(n)$.

Q 2. On veut trier des fonctions par "ordre" asymptotique de grandeur : si $f = o(g)$, f sera "avant" g pour cet ordre. On notera $f(n) \ll g(n)$ si $f = o(g)$. Trier les fonctions suivantes :

$$n \quad 4^n + n^2 \quad n \lg n + n^2 \quad \lg n \quad n^3 \quad n\sqrt{n} \quad 5^n$$

$$\lg n \ll n \ll n\sqrt{n} \ll n \lg n + n^2 \ll n^3 \ll 4^n + n^2 \ll 5^n$$

Q 3. Soit la suite $T(n)$ définie par $T(2) = T(1) = T(0) = 1$ et :

$$T(n) = T(n-1) + 2 * T(n-2) + 3 * T(n-3), n > 2$$

On considère le problème de calculer $T(n)$ pour l'entrée n. Soit l'algorithme naïf récursif associé :

```
int  T(int n){
  if (n<=2)
    return 1;
  else
    return T(n-1)+2*T(n-2)+3*T(n-3);
}
```

Q 3.1. Soit $A(n)$ le nombre d'appels récursifs à T effectués lors de l'exécution de la function $T(n)$. Exprimez $A(n)$ en fonction de $A(n-1), A(n-2), A(n-3)$. Qu'en déduire sur la complexité de l'algorithme naïf ?

$A(n) = 1$ si $n \leq 2$
$A(n) = A(n-1) + A(n-2) + A(n-3) + 1$ sinon
$A(n) > 3 * A(n-3)$ donc $A(n) > 3^{n/3} A(n \mod 3)$ donc $A(n) > 3^{n/3}$.

Q 3.2. Proposez un algorithme en $O(n)$ (en supposant qu'on est dans le cadre du coût uniforme, i.e. en ne prenant pas en compte la taille des opérandes) pour calculer $T(n)$. Peut-on dire qu'il est polynomial ?

```
int[] T =new int [n+1];
T[0]=1;
T[1]=1;
T[2]=1;
for (int i=>3;  i<=n;i++) T[i]=T[i-1]+2*T[i-<2]+3*T[n-3];
return T[n];
```

Q 4. Soit A, B, C, D quatre problèmes de décision. A est NP-dur, B est P, A se réduit polynomialement en D, C se réduit polynomialement en B, B se réduit en D.

Q 4.1. Parmi A, B, C, D quels sont les problèmes dont on peut affirmer qu'ils sont P ?

B et C

Q 4.2. Parmi A, B, C, D quels sont les problèmes dont on peut affirmer qu'ils sont NP-durs ?

A, D

Exercice 2 La toile - *5 pts*

Soient les trois problèmes de décision suivants :

SansLien

Donnée : n pages web avec les hyperliens entre ces pages, un entier k

Sortie : Oui, si on peut trouver un ensemble de k pages telle qu'aucune ne contienne un hyperlien vers une des $k-1$ autres pages. Non, sinon.

Communauté

Donnée : n pages web avec les hyperliens entre ces pages, un entier k

Sortie : Oui, si on peut trouver un ensemble de k pages telle que chacune contienne un hyperlien vers chacune des $k-1$ autres pages. Non, sinon.

Référence

Donnée : n pages web avec les hyperliens entre ces pages, un entier k

Sortie : Oui, si on peut trouver un ensemble de k pages et parmi elles une page telle que chacune des $k-1$ autres pages contienne un hyperlien vers elle. Non, sinon.

Q 1. Pour chacun de ces trois problèmes, dire si il est P, NP, NP-dur. Justifier.

SansLien : NP pareil qu'Independant, donc NP-dur
Communaute : NP pareil que Clique, donc NP-dur
Référence : une page avec k prédéceseurs : P, donc NP

Exercice 3 Choisir ses missions - *5 pts*

Un certain nombre de missions nous sont proposées. Pour chacune d'elles, on connaît la date de début, la date de fin et la valeur de la mission. On suppose que les missions sont **triées par début croissant**. On ne peut effectuer qu'une mission à la fois (on peut commencer une mission dès que l'autre termine). On cherche à sélectionner les missions à effectuer de façon à maximiser la valeur des missions effectuées.

Par exemple, pour les 5 missions suivantes,

mission 1 : [3, 7], valeur 8 mission 2 : [5, 10], valeur 12 mission 3 : [7, 8], valeur 2
mission 4 : [9, 11], valeur 2 mission 5 : [10, 12], valeur 4

il sera optimal d'effectuer les missions 2 et 5 pour une valeur totale 16.

Le problème est donc :

Entrée :
n le nombre de missions
$[d_i, f_i], v_i, 1 \leq i \leq n$ les débuts, fins et valeurs des missions, les d_i étant croissants.
Sortie : un choix $I \subset [1..n]$ optimal de missions compatibles - i.e. tel que, si $i \in I, j \in I$ avec $i \neq j$, alors $f_i \leq d_j$ ou $f_j \leq d_i$ et $\sum_{i \in I} v_i$ soit maximal.

Q 1. Quelles missions choisirez-vous d'effectuer pour optimiser la valeur, si les missions sont :
mission 1 : [3, 9], valeur 8 mission 2 : [5, 6], valeur 6 mission 3 : [7, 11], valeur 10
mission 4 : [8, 10], valeur 12 mission 5 : [11, 12], valeur 7

2,4,5 pour 25

Q 2. Proposez un algorithme qui étant donné le numéro i d'une mission retourne $Next(i)$, le numéro de la première mission dont le début est supérieur ou égal à la fin de la mission i. Si une telle mission n'existe pas, on pourra retourner une valeur sentinelle comme -1.

Pour le premier exemple, on aurait donc $Next(1) = 3, Next(2) = 5, Next(3) = 4, Next(4) = Next(5) = -1$.

```
for (int j=i; j<n; j++) if (fin[i] <=deb[j]) return j;
return n;
```

n valeur sentinelle (plus pratique que -1 pour la suite).
L'algorithme ci-dessus est donc en $\Theta(n)$ dans le pire des cas. On peut améliorer par une recherche dichotomique et obtenir un algorithme en $O(\log n)$.

Q 3. Proposez un algorithme polynomial pour le problème. Analysez sa complexité.
Meil(i)=max (Meil(i+1), val[i]+Meil(next(i))

```
Meil[n]=0;
for (int i=n-1; i>=0; i--) Meil[i]=max(Meil[i+1],val[i]+Meil[next(i)]);
//remontée
i=0;
while(i <n) {
  if Meil[i]>Meil[i+1] {sortir(i);i=next(i);}
  else i++;}
```

En n^2 ou $n \log n$ selon implémentation de next.

Exercice 4 Ordonner des tâches - *5 pts*

Les utilisateurs d'une machine soumettent des *tâches* qui doivent s'exécuter sur celle-ci. Une instance du problème est :
— n le nombre de tâches
— $a_0, a_1, \ldots, a_{n-1}$: pour chaque tâche, sa date d'arrivée (en secondes)
— $t_0, t_1, \ldots, t_{n-1}$: pour chaque tâche, sa durée (en secondes)

On cherche un ordonnancement qui minimise le temps de complétion des tâches, c'est à dire la date à laquelle toutes les tâches sont terminées. Une tâche ne peut être interrompue si elle est en cours d'exécution. Une solution peut être vue comme une permutation des n tâches indiquant l'ordre dans lequel elles s'exécutent.

Par exemple, soit $n = 5$, $a_0 = 10, t_0 = 3$, $a_1 = 0, t_1 = 2$, $a_2 = 4, t_2 = 1$, $a_3 = 9, t_3 = 2$, $a_4 = 2, t_4 = 4$. Une permutation possible est $1, 4, 2, 0, 3$. La tâche 1 serait lancée au temps 0, la tâche 4 au temps 2, la tâche 2 au temps 6, la tâche 0 au temps 10, la tâche 3 au temps 13, avec un temps de complétion égal à 15.

Q 1. Dans l'exemple précédent, l'ordre d'exécution proposé est-il optimal ? Justifier.
Non, $1, 4, 2, 3, 0$ a un temps de complétion égal à 14, donc inférieur.

Q 2. Proposez un algorithme en $\Theta(n)$ qui étant donné l'instance du problème et un ordonnancement - une permutation des n tâches-, calcule la date à laquelle toutes les tâches seront terminées. Par exemple, pour l'instance ci-dessus et l'ordonnancement $1, 4, 2, 0, 3$, la sortie attendue est 15. Vous pouvez supposer que l'instance et l'ordonnancement sont représentés par la structure de données de votre choix.

```
int completionTime(int[] a, int[] t, int[] ord){
int tc=0;
for (int i=0; i<a.length;i++)
tc=Math.max(tc,a[ord[i]])+t[ord[i]];
return tc;
}
```

Q 3. Le professeur Ordonnix propose d'utiliser un critère glouton pour trier les tâches afin de trouver une permutation optimale, c'est à dire qui minimise le temps de complétion. Il propose pour cela trois critères de tri :

— par t_i croissant, i.e. par durée croissante
— par a_i croissant, i.e. par date d'arrivée croissante
— par $a_i + t_i$ croissant.

Lequel de ces trois critères donne toujours la solution optimale ? Justifiez votre réponse, par une preuve pour le critère qui garantit l'optimalité, par un contre-exemple pour chacun des x autres critères.

— par t_i croissant, i.e. par durée croissante Non optimal, par exemple $a_0 = 0, d_0 = 3, a_1 = 1, d_0 = 1$. Par ce critère donnerait 5 comme complétion au lieu de 4.
— par a_i croissant, i.e. par date d'arrivée croissante Optimal.
Supposons qu'un ordonnancement ne soit pas trié par a_i croissant : deux éléments consécutifs ne respectent pas le critère, ie a[ord[i]]>a[ord[i+1]] ; Soit ord' défini par ord'[j]=ord[j] si $j \neq i$ et $j \neq i+1$, ord'[i]=ord[i+1], ord'[i+1]=ord[i].
Notons d_0(resp. d_1) la date de lancement de ord[i] (resp ord[i+1]) dans ord et d_2 la date à laquelle la tâche ord[i+1] est terminée.
On peut remarquer que $d_1 = d_0 + T[ord[i]]$ puisque la date d'arrivée de ord[i+1] est inférieure à celle de ord[i] : elle peut être lancée dès que ord[i] est finie.
Les tâches de numéro < ord[i] s'exécutent comme avant.
La tâche ord'[i] sera lancée au plus tard à d_0 : en effet à d_0 les autres tâches sont terminées et a[ord[i]]>a[ord'[i]]
la tâche ord[i+1] sera donc lancée au plus tard à $d_0 + T[ord[i+1]]$, puisque sa date d'arrivée est au plus d_0 et que ord'[i] est finie à $d_0 + T[ord[i+1]]$ et donc terminée au plus tard à d_2.
Donc les tâches suivantes seront lancées et donc terminées au plus tard comme dans ord.
ord' est donc aussi bon que ord.
donc la solution triée par date d'arrivée croissante est optimale.
Donc si on part d'une optimale, on la trie en éliminant les inversions comme ci-dessus : on obtient une solution triée au moins aussi bonne donc optimale.
Ou par l'absurde, on suppose qu'il existe des solution non triées par a_i croissant qui soient optimales et strictement meilleures que la triée. On prend parmi celles-ci celle qui a le minimum d'inversions, ie de couples (i,j), i<j tels que a[ord[i]]>a[ord[j]]). On fait la transformation ci-dessus : elle fait diminuer de 1 le nombre d'inversions et est aussi bonne.
Donc il existe une solution optimale triée par a_i croissant. De plus, si on a des tâches qui ont même date d'arrivée, on peut remarquer que l'ordre dans lequel elles sont exécutées - à condition que l'ordonnancement soit trié par a_i croissant, n'a pas d'importance pour la date de complétion.
Donc toute solution triée par a_i croissants, et donc la gloutonne, est optimale.
— par $a_i + t_i$ croissant. Non optimal, même exemple que ci-dessus.