ÉCOLE POLYTECHNIQUE - MP/PC - ÉPREUVE FACULTATIVE D'INFORMATIQUE 2006

Corrigé rédigé par Alain Schauber - alain.schauber@prepas.org

Version de MAPLE : Classic Worksheet Maple 10.

Partie I

A. Coût d'une séquence de déplacements

Question 1

Il suffit de tenir à jour les positions des deux têtes de lecteur et de faire une lecture simultanée des tableaux **r** et **d** en ajoutant au coût courant **c** le coût du déplacement de la tête concernée (lue dans **d**), calculé à l'aide de sa position courante et de sa position cible lue dans **r**. Puis il convient de mettre à jour la position de la tête déplacée.

```
> coutDe:=proc(r,d) local c,pos1,pos2,k;
    c:=0; #initialisation du coût
    pos1:=0; pos2:=0; #initialisations des positions des lecteurs
    for k to op(2,op(2,eval(r))) #longueur commune des tableaux r et d
    do
      if d[k] = 1 then c:=c+abs(r[k]-pos1); pos1:=r[k]
      else c:=c+abs(r[k]-pos2); pos2:=r[k]
      fi
    od;
    c
  end;
```

Question 2

L'ensemble des coûts de toutes les séquences possibles pour un tableau de requêtes donné est un ensemble d'entiers, qui admet donc un plus petit élément. Soit **d** une séquence de déplacements correspondant à ce coût minimal. La symétrie des rôles des deux têtes dans la lecture entraîne qu'en permutant 1 et 2 dans **d** on ne change pas le coût de la séquence. Il en résulte que le déplacement obtenu après permutation est aussi de coût minimal. Et l'une des deux commence bien par 1.

Question 3

Soit **n** le nombre de requêtes du bloc **r**. Le tableau **d** contient **n** cellules, dont les valeurs sont choisies indépendamment les unes des autres parmi les 2 valeurs 1 et 2.
Il y a donc 2^**n** déplacements satisfaisant une requête donnée.

B. Coût optimal pour deux requêtes

Question 4

On ne peut agir que sur le choix de la tête qui satisfait la 2ème requête. On compare donc le choix 1 avec le choix 2.
Résultats : pour le bloc <10,3> on trouve la séquence <1,2> avec le coût 13; pour <3,10> on trouve la séquence <1,1> de coût 10.

Question 5
Il s'agit de programmer la comparaison évoquée ci-dessus. Celle-ci va dépendre de la meilleure proximité de **r2** par rapport à 0 (position de la tête 2) et **r1** (position de la tête 1), ce qui revient à comparer **r2** avec **r1**/2.

```
> coutOpt2:=proc(r1,r2) local d;
    d:=array(1..2); d[1]:=1; #Initialisation avant comparaison
    if r2 < r1/2 then d[2]:=2 else d[2]:=1 fi;
    eval(d)
  end;
```

Partie II

A. Coût optimal pour trois requêtes

Question 6
Avec **r** = <20,9,1>, la tête 1 (par convention) se déplace en 20. C'est donc la tête 2 qui est alors la plus proche de 9. Enfin, il est clair que pour aller en 1, le déplacement de la tête 2 (de coût 8) est moins onéreux que celui de la tête 1 (de coût 19). On trouve donc la séquence <1,2,2>.

Question 7
Du point de vue du coût, on peut d'après la question 2 se limiter aux 4 stratégies commençant par 1. En voici l'énumération:
<1,1,1> : coût = 39 ; <1,1,2> : coût = 32 ; <1,2,1> : coût = 48 ; <1,2,2> : coût = 37.
On vérifie donc bien que l'approche de la question 6 ne fournit pas la solution de coût minimal car 32 < 37.

Question 8
Il s'agit d'énumérer les différentes possibilités de séquences **d**, en calculant leur coût **c** et en tenant à jour une variable **cmin** contenant le meilleur coût courant.
A chaque amélioration de **cmin**, on copie la séquence **d** correspondante dans une séquence **dmin**, qu'on retourne à l'issue de l'énumération.

```
> coutOpt3:=proc(r1,r2,r3) local r,d,c,cmin,dmin;
    r:=array(1..3,[r1,r2,r3]); d:=array(1..3,[1,1,1]);
    cmin:=coutDe(r,d); dmin:=copy(d);
    d[3]:=2; c:=coutDe(r,d); if c < cmin then dmin:=copy(d); cmin:=c fi;
    d[2]:=2; c:=coutDe(r,d); if c < cmin then dmin:=copy(d); cmin:=c fi;
    d[3]:=1; c:=coutDe(r,d); if c < cmin then dmin:=copy(d); cmin:=c fi;
    eval(dmin)
  end;
```

B. Coût optimal pour n requêtes

Question 9
Si **r** = <r(1),...,r(n)>, alors après la n-ième requête on aura 2 possibilités de satisfaction de la requête r(n) : soit elle aura été satisfaite par la tête 1, soit par la tête 2. En vertu de la symétrie des têtes évoquée dans la question 2, on peut supposer que la requête r(n) est satisfaite par la tête 1, ce qui implique que dans la matrice **cout(n)** on trouvera le coût minimal de cette requête dans la ligne n° n (si on numérote les lignes et les colonnes de 0 à n). En revanche, la position de la tête 2 dépend du numéro de la dernière requête qu'elle a satisfaite, qui est quelconque parmi les valeurs 0,1,..,n-1.
Il conviendra de comparer entre eux tous les termes de la dernière ligne (à l'exception du plus à droite, qui correspond à une configuration impossible) pour trouver le coût minimal absolu de **r**.

Question 10
1) La seule configuration possible avant lecture de la requête, c'est-à-dire conventionnellement au moment de la lecture de la requête r(0), est i=0 et j=0, pour laquelle le coût est nul (on n'a encore rien déplacé). Par suite, dans la matrice **cout(0)**, le terme d'indice (0,0) est nul et les autres à l'infini.

2) La valeur du terme d'indice (i,k) de la matrice **cout(k)** est le coût minimal obtenu après satisfaction des requêtes r(p), pour 1<= p <= k, avec l'hypothèse que la tête 2 a satisfait la k-ième requête et que la la tête 1 a satisfait la i-ème requête. Cela signifie que cette configuration est atteinte à partir d'une situation précédente telle qu'après lecture de la (k-1)-ème requête, la dernière requête satisfaite par la tête 1 est la requête r(i). Dans cette situation, la tête 2 avait satisfait en dernier une requête j, avec 0 <= j <= k-1, ce qui correspond à un coût minimal lisible dans la matrice **cout(n-1)** au niveau du terme d'indice (i,j). Comme c'est la tête 2 qui se déplace pour satisfaire la k-ième requête et qu'elle effectue un déplacement |r(k)-r(j)|, il est clair qu'on calculera le coût minimal pour atteindre la configuration (i,k) après satisfaction de k requêtes en calculant le minimum de |r(k)-r(j)| + **cout(n-1)**(i,j), pour 0 <= j <= k-1.
Remarquons que ce résultat est plus précis que celui de l'énoncé, qui n'est pas faux puisque pour j > k-1 on a **count(n-1)** (i,j)=infini (voir item 4).

3) La matrice **cout(k)** est symétrique en vertu de la symétrie des rôles des têtes, déjà mise en évidence dans la question 9, où on aurait aussi bien pu raisonner sur la dernière colonne.

4) Si i <> k et j <> k, la configuration (i,j) n'est pas atteignable après la k-ième requête, puisque celle-ci a été satisfaite soit par la tête 1 (on est alors dans la ligne k de la matrice **cout(k)**) soit par la tête 2 (on est alors dans la colonne k).

Remarque : comme mentionné plus bas dans l'énoncé, la matrice **cout(k)** est très creuse... On peut ajouter que sa diagonale aussi est "infinie" (voir remarque en fin de question 9), ainsi que les lignes ou les colonnes d'indice > k . Il n'y a donc que k éléments "intéressants" dans cette matrice, les éléments d'indice (k,j), pour 0 <= j <= k-1.

Question 11
On utilise les propriétés de la question 10. On met à jour la ligne (et par symétrie la colonne) n° k à l'aide de la propriété 2, puis on remet à "l'infini" les éléments non "infinis" de la matrice initiale dans les ligne et colonne n° k-1.

```
> mettreAJour:=proc(cout,r,k) local n,i,j,cmin;
    n:=op(2,op(2,eval(r)));
    for i from 0 to n do
      cmin:=r[k]+cout[i,0]; #r[0]=0, on traite ce cas à part
      for j from 1 to n do cmin:=min(cmin,abs(r[k]-r[j])+cout[i,j]) od;
      cout[i,k]:=cmin; cout[k,i]:=cmin
    od;
    for i from 0 to k-1 do cout[i,k-1]:=infinity; cout[k-1,i]:=infinity od;
    RETURN()
  end;
```

Question 12
Après initialisation de la matrice **cout**, il suffit de faire une boucle de mises à jour à l'aide de la fonction **mettreAJour** en faisant varier **k** de 1 à **n**.
Enfin, on calcule le coût optimal en utilisant le procédé décrit à la question 9.
Calculons le temps d'exécution. L'initialisation se fait en O(**n**^{2}). **mettreAJour** est constituée de deux boucles imbriquées de longueur **n**, et d'une boucle simple de longueur **n**. Son temps d'exécution est donc de l'ordre de O(**n**^{2})+O(**n**) = O(**n**^{2}). Comme cette fonction est itérée dans une boucle (indexée par **i**) de longueur **n**, la boucle en question s'exécute en O(**n**^3). Enfin la boucle indexée par **j** est en O(**n**). On a donc O(**n**^{2}) + O(**n**^3) + O(**n**) + O(1) (pour les instructions élémentaires extérieures aux boucles). En conclusion, le temps d'exécution de **coutOpt** en fonction de **n** est un O(**n**^3).

```
> coutOpt:=proc(r) local n,cout,i,j,k,cmin;
    n:=op(2,op(2,eval(r)));
    cout:=array(0..n,0..n);
    for i from 0 to n do for j from 0 to n do cout[i,j]:=infinity od od;
    cout[0,0]:=0;
    for k from 1 to n do mettreAJour(cout,r,k) od;
    cmin:= infinity;
    for j from 0 to n do cmin:=min(cmin,cout[n,j]) od;
    cmin
  end;
```

Question 13

On fait peu ou prou la même chose que sur la matrice **cout**, en programmant une boucle indexée par **k**, mais en ne conservant que les valeurs de la ligne **k** dans le tableau **t** de taille **n**+1, c'est-à-dire qu'on s'intéresse à l'évolution de la tête 2. Le tableau va se remplir vers la droite au fur et à mesure du déroulement de la boucle, en utilisant les valeurs déjà calculées et en les remplaçant.

La difficulté ici consiste à ne pas écraser des valeurs avant qu'elles aient été exploitées. Pour ce faire, on met à profit la remarque de l'énoncé (qui reprend la propriété 2 démontrée à la question 10) et on calcule d'abord la **k**-ième valeur, c'est-à-dire la nouvelle cellule à droite : c'est le rôle de la boucle indexée par **j** qui met à jour une variable **cmin** selon la méthode issue de la propriété 2 de la question 10. Ensuite seulement on écrase les valeurs de 0 à **k**-1 en utilisant la remarque de l'énoncé à la question 13.

Une fois le processus terminé, on calcule le minimum des **n** coûts présents dans le tableau **t**.

On remarquera que seules les **n** premières cellules du tableau sont utilisées, on peut donc travailler avec un tableau de taille **n**. Il ne s'agit pas là d'un souci (futile) d'optimisation : cela provient de l'initialisation **t[0]:=r[1]** qui représente à elle seule l'itération pour **k**=1, et permet d'éviter l'appel (ce qui créerait une erreur) dans la boucle générale à r[0], qui est nul par convention, mais n'existe pas dans **r**.

Cette nouvelle fonction **coutOpt** comporte une boucle principale indexée par **k** et de longueur **n**, contenant une boucle imbriquée (indexée par **j**) de longueur **n** elle aussi. Il y a également une boucle (indexée par **i**) de longueur **n**. Le temps d'exécution est donc en $\mathbf{n}O(\mathbf{n})+O(\mathbf{n})+O(1) = O(\mathbf{n}^2)$.

On a donc affaire à une fonction de complexité quadratique, ce qui représente une amélioration par rapport à la version de la question12.

```
> coutOpt:=proc(r) local n,t,k,cmin,j,i;
    n:=op(2,op(2,eval(r)));
    t:=array(0..n);
    t[0]:=r[1]; #première requête = premier coût
    for k from 2 to n do
      #on calcule d'abord l'élément le plus à droite
      cmin:=r[k]+t[0];
      for j from 1 to k-2 do cmin:=min(cmin,abs(r[k]-r[j])+t[j]) od;
      t[k-1]:=cmin;
      #puis les autres
      for i from 0 to k-2 do t[i]:=abs(r[k]-r[k-1])+t[i] od
    od;
    #calcul final du minimum des coûts obtenus
    cmin:= t[0];
    for i from 1 to n-1 do cmin:=min(cmin,t[i]) od;
    cmin
  end;
```