Sous-Séquence maximale [cx02] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Université Haute-Alsace

Version 21 mai 2018

Table des matières

C++ - Sous-séquence maximale (Solution)

Mots-Clés Complexité des algorithmes ■
Requis Axiomatique impérative, Récursivité des actions ■
Fichiers UtilsTB, UtilsTBOpers ■
Difficulté • ○ ○ (2 h) ■

Objectif
Cet exercice étend l'étude de cas du cours @[Sous-séquence de somme maximale].

1 Énoncé

Objectif
Soit `t[1..n]` un tableau d'entiers (positifs, négatifs ou nuls). Sans utiliser de tableau auxiliaire, déterminez **la valeur et les indices** du sous-tableau `t[g..h]` donnant la somme la plus grande de tous les sous-tableaux contigus de `t`.

2 Algorithmique, Programmation

2.1 Algorithme naïf

L'**algorithme naïf** consiste à :

1. Énumérer tous les couples `(ix,jx)` qui délimitent une partie du tableau.
2. Calculer la somme des éléments pour cette partie.
3. Calculer le maximum de toutes ces sommes.

Définissez la constante `TMAX=1000` (taille maximale des tableaux) et le type `Tableau` comme étant un tableau d'entiers de taille maximale `TMAX`.

Écrivez une procédure `xactualiserSi(vmax,somme,g,h,ix,jx)` qui actualise le triplet d'entiers `(vmax,g,h)` avec le triplet d'entiers de valeurs `(somme,ix,jx)` si `somme>vmax`.

Écrivez une procédure `exec_vmaxsomme1(t,n,vmax,g,h)` qui, pour `n` éléments d'un `Tableau t`, calcule la valeur de la meilleure somme dans `vmax` (entier) ainsi que les indices dans `g` (entier) et dans `h` (entier) du sous-tableau correspondant.

Validez vos procédures avec la solution.

Solution C++ @[pgvmaxsomme2.cpp]

```
/**
  Actualise le triplet (vmax,g,h) avec (somme,ix,jx) si somme>vmax
  @param[in,out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
  @param[in] somme - somme
  @param[out] ix - indice bas
  @param[out] jx - indice haut
*/
void xactualiserSi(int& vmax, int& g, int& h, int somme, int ix, int jx)
{
```

```
  if (somme > vmax)
  {
    vmax = somme;
    g = ix;
    h = jx;
  }
}
```

```
/**
  Algorithme naïf du calcul de la "meilleure somme"
  @param[in] t - un tableau d'entiers
  @param[in] n - nombre d'éléments
  @param[out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
*/
void exec_vmaxsomme1(const Tableau& t, int n, int& vmax, int& g, int& h)
{
  vmax = t[0];
  for (int ix = 0; ix < n; ++ix)
  {
    for (int jx = 0; jx < n; ++jx)
    {
      int somme = 0;
      for (int k = ix; k <= jx; ++k)
      {
        somme += t[k];
      }
      xactualiserSi(vmax, g, h, somme, ix, jx);
    }
  }
}
```

Quelle est la complexité (en nombre de somme) de `exec_vmaxsomme1` pour n éléments ?

Solution simple

Le nombre N de fois que l'instruction de la somme :

```
somme <- somme + t[k]
```

est effectuée vaut :

$$\begin{aligned} N &= \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 \\ &= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \end{aligned}$$

Or :

$$\begin{aligned} \sum_{j=i}^{n}(j-i+1) &= \sum_{j=1}^{n-i+1} j \\ &= \frac{1}{2}(n-i+1)(n-i+2) \end{aligned}$$

Il reste donc :

$$\begin{aligned} N &= \sum_{i=1}^{n} \frac{1}{2}(n-i+1)(n-i+2) \\ &= \frac{1}{2}\sum_{i=1}^{n} i(i+1) \end{aligned}$$

Comme (formule de FAULHABER) :

$$\sum_{i=1}^{n} i^2 = \frac{1}{6}(2n^3+3n^2+n)$$

Finalement :

$$N = \frac{1}{6}(n^3+\alpha n^2+\beta n)$$

La complexité globale de la procédure est donc en $\Theta(n^3)$.

2.2 Algorithme amélioré

Dans l'algorithme précédent, les sommes partielles sont calculées plusieurs fois :

- La boucle sur `k` évalue la somme :

$$S_j = \sum_{k=i}^{j} t[k]$$

- A l'étape suivante, cette même boucle évalue la somme :

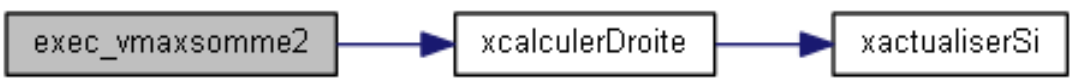
$$S_{j+1} = \sum_{k=i}^{j+1} t[k]$$

- On en déduit la relation entre S_j et S_{j+1} :

$$S_{j+1} = S_j + t[j+1]$$

Écrivez une procédure `xcalculerDroite(t,ideb,ifin,vmax,g,h)` qui calcule vers la droite le triplet d'entiers `(vmax,g,h)` du sous-tableau `t[ideb..ifin]` d'un `Tableau t`.

Écrivez une procédure `exec_vmaxsomme2(t,n,vmax,g,h)` qui calcule la valeur de la meilleure somme dans `vmax` (entier) ainsi que les indices dans `g` (entier) et dans `h` (entier) du sous-tableau correspondant pour `n` éléments d'un `Tableau t`.



Validez vos procédures avec la solution.

Solution C++ @[pgvmaxsomme2.cpp]

```
/**
  Calcule vers la droite la somme vmax et les indices de t[ideb..ifin]
  @param[in] t - un tableau d'entiers
  @param[in] ideb - indice de début
  @param[in] ifin - indice de fin
  @param[in,out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
*/
void xcalculerDroite(const Tableau& t, int ideb, int ifin, int& vmax, int& g, int& h)
{
  int somme = 0;
  for (int jx = ideb; jx <= ifin; ++jx)
  {
    somme += t[jx];
    xactualiserSi(vmax, g, h, somme, ideb, jx);
  }
}
```

```
/**
  Algorithme amélioré du calcul de la "meilleure somme"
  @param[in] t - un tableau d'entiers
  @param[in] n - nombre d'éléments
  @param[out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
*/
void exec_vmaxsomme2(const Tableau& t, int n, int& vmax, int& g, int& h)
{
  vmax = t[0];
  for (int ix = 0; ix < n; ++ix)
  {
    xcalculerDroite(t, ix, n-1, vmax, g, h);
  }
}
```

Quelle est la complexité (en terme de somme) de `exec_vmaxsomme2` pour n éléments ?

Solution simple

Le nombre N de fois que l'instruction de la somme est effectuée vaut :

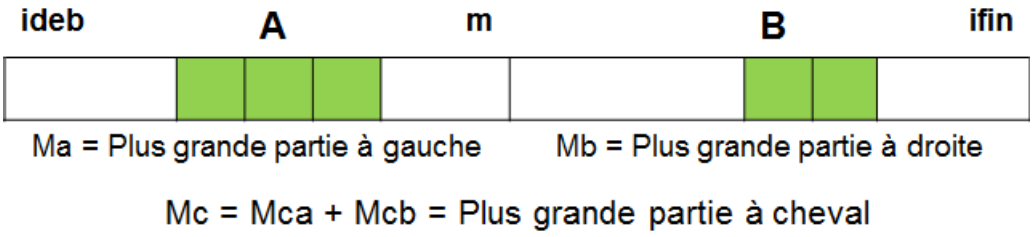
$$\begin{aligned} N &= \sum_{i=1}^{n}\sum_{j=i}^{n} 1 \\ &= \sum_{i=1}^{n}(n-i+1) \\ &= \sum_{i=1}^{n} i = \frac{1}{2}n(n+1) \end{aligned}$$

La complexité de la procédure est donc en $\Theta(n^2)$.

2.3 Une solution récursive : diviser pour résoudre

Dans la stratégie Diviser-pour-régner, le tableau initial est scindé en deux parties de tailles quasi-égales (selon que n est pair ou impair) :

- Récursivement on calcule les solutions de chaque sous-tableau :
  on obtient `Ma` et `Mb`.
- Disposant de ces informations, le problème est presque résolu. En effet, pour de nombreux tableaux, le sous-tableau désiré `t[g..h]` se trouve soit dans `A` ou soit dans `B`. Si ce n'est pas le cas, le sous-tableau `t[g..h]` que nous notons `C` est « à cheval » sur les deux parties `A` et `B`.
- Notons `Mc` la somme associée à `C`.
  La solution est donnée par la plus grande valeur entre `Ma`, `Mb` et `Mc`.



Calcul de Mc

Il reste à montrer comment calculer `Mc`.

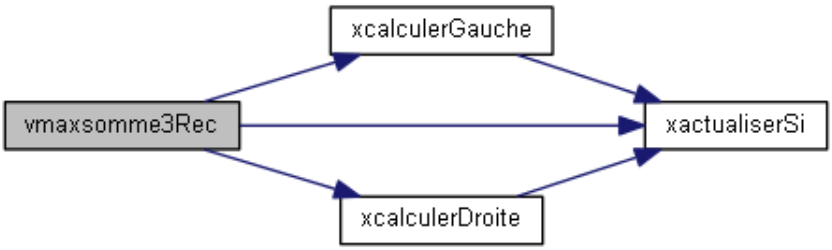
- Il suffit de noter que `Mc| s'écrit \lstinline`Mc=Mca+Mcb@ où `Mca` et `Mcb` désignent respectivement les sommes des éléments faisant partie de `A` et de `B`.
- Pour calculer `Mca` et `Mcb` on se ramène à un cas particulier du problème initial : lorsque l'un des indices `g` ou `h` est connu, la résolution est immédiate : « balayer » le tableau à partir de la position `g` ou `h` et retenir les éléments tant que leurs valeurs améliorent la somme.

Comment sortir des appels récursifs ?

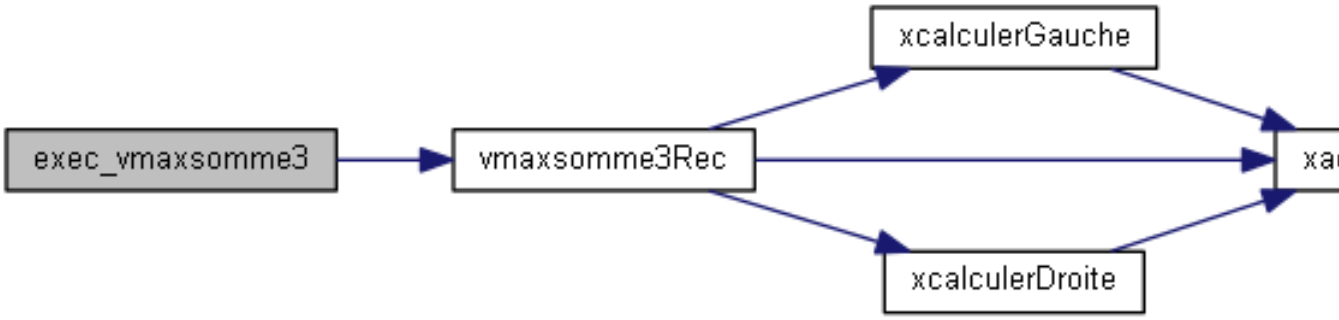
Il est nécessaire de rencontrer un « couple de données-paramètres » (transmis à l'appel) dont la solution est triviale. C'est le cas si le tableau est composé d'au plus un élément.

Écrivez une procédure `xcalculerGauche(t,ideb,ifin,vmax,g,h)` qui calcule vers la gauche le triplet d'entiers `(vmax,g,h)` du sous-tableau d'entiers `t[ideb..ifin]` de taille maximale `NMAX`.

Écrivez une procédure récursive `vmaxsomme3Rec(t,ideb,ifin,vmax,g,h)` qui calcule la valeur de la meilleure somme dans `vmax` (entier) ainsi que les indices dans `g` (entier) et dans `h` (entier) du sous-tableau d'entiers `t[ideb..ifin]` de taille maximale `NMAX`.



Écrivez une procédure `exec_vmaxsomme3(t,n,vmax,g,h)` qui se contente d'appeler la procédure récursive `vmaxsomme3Rec` pour les `n` éléments d'un `Tableau t`.



Validez vos procédures avec la solution.

Solution C++ @[pgvmaxsomme2.cpp]

```
/**
  Calcule vers la gauche la somme vmax et les indices de t[ideb..ifin]
  @param[in] t - un tableau d'entiers
  @param[in] ideb - indice de début
  @param[in] ifin - indice de fin
  @param[in,out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
*/
void xcalculerGauche(const Tableau& t, int ideb, int ifin, int& vmax, int& g, int& h)
{
  int somme = 0;
  for (int jx = ifin; jx >= ideb; --jx)
  {
    somme += t[jx];
    xactualiserSi(vmax, g, h, somme, jx, ifin);
  }
}
```

```
/**
  Algorithme récursif du calcul de la "meilleure somme"
  @param[in] t - un tableau d'entiers
  @param[in] ideb - indice de début
  @param[in] ifin - indice de fin
  @param[out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
```

```
*/
void vmaxsomme3Rec(const Tableau& t, int ideb, int ifin, int& vmax, int& g, int& h)
{
  if (ideb == ifin)
  {
    vmax = t[ideb];
    g = ideb;
    h = ifin;
  }
  else
  {
    int x;
    int milieu = (ideb + ifin)/2;
    int mgmax = t[milieu];
    xcalculerGauche(t, ideb, milieu, mgmax, g, x);
    int mdmax = t[milieu+1];
    xcalculerDroite(t, milieu+1, ifin, mdmax, x, h);
    vmax = mgmax + mdmax;
    int gmax, g1, h1;
    vmaxsomme3Rec(t, ideb, milieu, gmax, g1, h1);
    int dmax, g2, h2;
    vmaxsomme3Rec(t, milieu+1, ifin, dmax, g2, h2);
    xactualiserSi(vmax, g, h, gmax, g1, h1);
    xactualiserSi(vmax, g, h, dmax, g2, h2);
  }
}
```

```
/**
  Procédure maître : Algorithme récursif du calcul de la "meilleure somme"
  @param[in] t - un tableau d'entiers
  @param[in] n - nombre d'éléments
  @param[out] vmax - valeur maximale
  @param[out] g - indice bas
  @param[out] h - indice haut
*/
void exec_vmaxsomme3(const Tableau& t, int n, int& vmax, int& g, int& h)
{
  vmaxsomme3Rec(t, 0, n-1, vmax, g, h);
}
```

Montrez que la complexité de la procédure est un $\Theta(n \lg n)$.

Solution simple

L'équation de récurrence sous-jacente est :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$$

(Le $\Theta(n)$ est le temps nécessaire au calcul de Mc). Supposons que n soit une puissance de 2 : $n = 2^k$ (d'où $k = \lg n$). On a alors :

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2(2T(n/4) + \Theta(n/2)) + \Theta(n) \\ &\dots \\ &= 2^k T(n/2^k) + k(\Theta(n)) \end{aligned}$$

Puisque $T(1)$ est égale à une constante a, nous obtenons :

$$T(n) = n \cdot a + k \cdot \Theta(n)$$

Une complexité « en temps linéaire $\Theta(n)$ » s'exprime sous forme de fonction mathématique par la relation $T(n) = \alpha n + \beta$.

Conclusion $T(n) = \gamma n \lg n + \delta n$. La solution est donc en $\Theta(n \lg n)$.

2.4 Une solution en temps linéaire

La meilleure réponse que l'on puisse espérer pour ce problème est une solution de complexité linéaire. En effet, il est impossible de réaliser moins de $\Theta(n)$ opérations si toutes les n valeurs doivent être traitées. La solution linéaire est donc optimale.

Si cette solution existe, l'algorithme associé procèdera alors en un balayage des n éléments en mettant à jour à chaque étape les informations recherchées (somme et/ou indices). Or la valeur `t[j]| modifie la situation atteinte en \lstinline`j-1| que si elle contribue à l'amélioration de la somme déjà obtenue. En effet, quand la somme devient négative, elle ne peut pas pas préfixer une plus grande somme : il suffirait de l'ôter à la somme pour que cette somme devienne plus grande. Dans ce cas une plus grande somme ne pourra commencer que en `j`.

Écrivez une procédure `exec_vmaxsomme4(t,n,vmax,g,h)` qui calcule en temps linéaire la valeur de la meilleure somme dans `vmax` (entier) ainsi que les indices dans `g` (entier) et dans `h` (entier) du sous-tableau correspondant pour `n` éléments d'un `Tableau t`.

Validez vos procédures avec la solution.

Solution C++ @[pgvmaxsomme2.cpp]

```
/**
  Algorithme optimal du calcul de la "meilleure somme"
  @param[in] t - un tableau d'entiers
  @param[in] n - nombre d'éléments
  @param[out] vmax - valeur maximale
  @param[out] g - indice bas
```

```
  @param[out] h - indice haut
*/
void exec_vmaxsomme4(const Tableau& t, int n, int& vmax, int& g, int& h)
{
  int ix = 0;
  vmax = t[ix];
  g = 0;
  h = 0;
  int somme = 0;
  for (int jx = 0; jx < n; ++jx)
  {
    if (somme < 0)
    {
      somme = 0;
      ix = jx;
    }
    somme += t[jx];
    xactualiserSi(vmax, g, h, somme, ix, jx);
  }
}
```

Montrez que la complexité de la procédure est un $\Theta(n)$.

Solution simple
Immédiat : unique boucle `Pour` traversant les `n` éléments.

2.5 Tests et performances

Ce problème vérifie l'exactitude des résultats sur des tableaux de petites tailles puis il vérifie le comportement des algorithmes étudiés précédemment.

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsTB.cpp]

Copiez/collez ensuite les lignes suivantes :

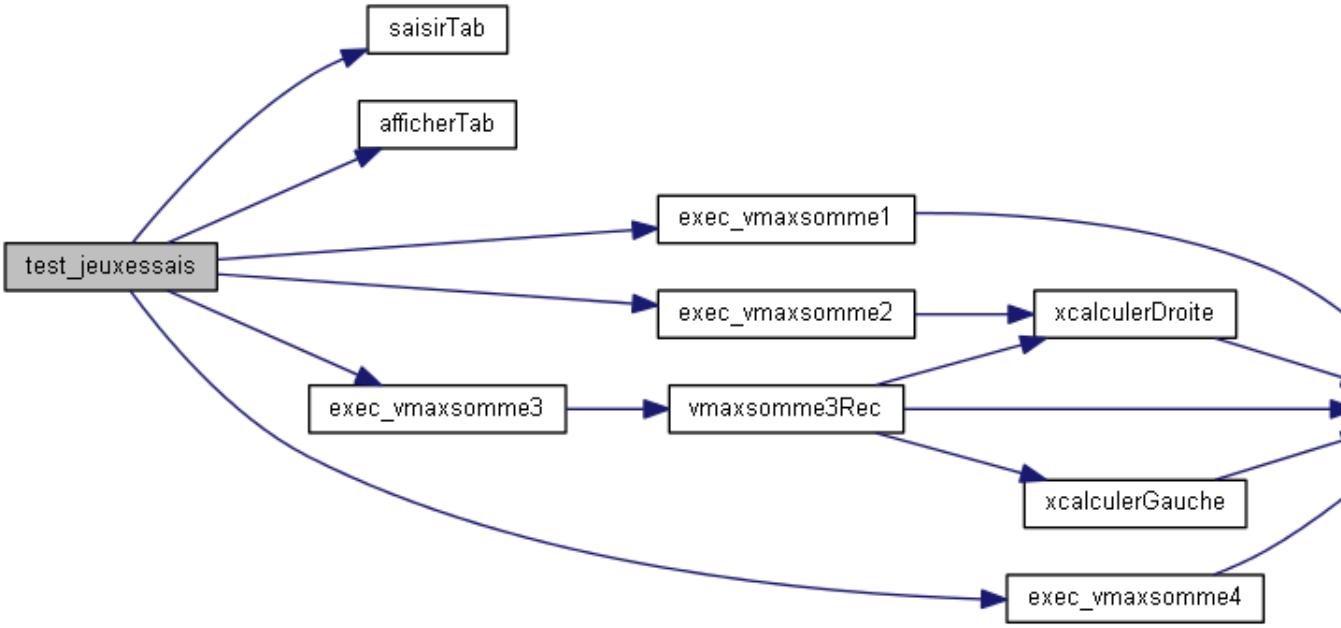
C++ **Au début** de votre programme :

```
#include "UtilsTB.cpp"
using namespace UtilsTB;
```

Soit la fonction `saisirTab(t)` qui effectue la saisie contrôlée du nombre de valeurs (entier compris entre 1 et `TMAX`), saisit les valeurs entières dans un `ITableau t` puis renvoie l'entier du nombre de valeurs saisies.

C++ @[saisirTab] (dans UtilsTB)

Écrivez une procédure `test_jeuxessais` qui déclare un entier et un `Tableau`, saisit les valeurs puis appelle les procédures `exec_vmaxsomme` et affiche les résultats pour chaque appel.



Testez avec les jeux d'essais suivants (`nelems` vaut 10).

```
t=[1 -6 4 -7 9 -5 11 -10 3 -2] M=15 g=5 h=7
t=[-12 10 15 -50 2 75 -2 4 10 5] M=94 g=5 h=10
t=[-12 10 15 -50 2 75 -2 4 -10 5] M=79 g=5 h=8
```

Validez votre procédure avec la solution.

Solution C++ @[pgvmaxsomme2.cpp]

```
void test_jeuxessais()
{
  Tableau tab;
  int nelems = saisirTab(tab);
  afficherTab(tab, nelems);
  int vmax, g, h;
  exec_vmaxsomme1(tab, nelems, vmax, g, h);
  cout<<"==> maxsomme1 = "<<vmax<<" "<<g<<" "<<h<<endl;
  exec_vmaxsomme2(tab, nelems, vmax, g, h);
  cout<<"==> maxsomme2 = "<<vmax<<" "<<g<<" "<<h<<endl;
  exec_vmaxsomme3(tab, nelems, vmax, g, h);
  cout<<"==> maxsomme3 = "<<vmax<<" "<<g<<" "<<h<<endl;
  exec_vmaxsomme4(tab, nelems, vmax, g, h);
  cout<<"==> maxsomme4 = "<<vmax<<" "<<g<<" "<<h<<endl;
}
```

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsTR.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsTR.cpp"
using namespace UtilsTR;
```

Soit la procédure `aleatoireTri(t,n,vmax)`H qui initialise les `n` premiers éléments d'un `ITableau t` en utilisant `vmax` comme valeur maximale pour la fonction de génération d'un entier pseudo-aléatoire.

C++ @[aleatoireTri] (dans UtilsTR.cpp)

Écrivez une procédure `test_performances` qui chronomètre le temps d'exécution des procédures pour des tableaux de taille `n=100,200,...,1000` tirés aléatoirement sur `]-n..n[`.

3 Preuve de l'algorithme linéaire

La solution en temps linéaire applique la démarche du raisonnement par récurrence (DRR). Cette section établit la preuve de l'algorithme linéaire. Elle ne comporte pas de question.

Principe

Soit $P(j-1)$ une assertion décrivant la partie de travail réalisée par l'algorithme jusqu'à l'étape $j-1$. Considérons alors la valeur `t[j]` de l'étape suivante j. Nous devons répondre aux questions suivantes :

- Comment traiter `t[j]` ?
- Comment calculer la somme `M` et les indices ?
- Quelle assertion obtient-on alors ?

Hypothèse de récurrence

Pour déterminer un invariant de boucle on part en général de l'hypothèse classique qui consiste à supposer « qu'on a déjà fait une partie du travail ». Autrement dit, « on a parcouru et traité les éléments de rang 1 à $j-1$ du tableau t ». Par conséquence le problème est résolu pour $t[1..j-1]$: on a trouvé une somme maximale qui commence en g et qui s'achève en h.

Sous forme d'assertion, nous avons :

- Condition d'arrêt : l'itération s'arrête lorsque $j > n$
- Pas d'itération : l'exécution des instructions du pas de boucle à l'issue d'un pas de l'itération doivent entrainer (i) la convergence de l'algorithme (avancer vers la solution) et (ii) la conservation de l'hypothèse de travail. Analysons cela.

A l'étape j on considère l'élément $t[j]$. Comment traiter $t[j]$? Ou encore, comment étendre la solution obtenue pour $t[1..j-1]$ à une solution pour $t[1..j]$? Pour cela, il faut vérifier de quelle manière $t[j]$ peut-il modifier l'état du tableau décrit ci-dessus.

La valeur $t[j]$ modifie la situation atteinte en $j-1$ que si elle contribue à l'amélioration de la somme déjà obtenue. Dans ce cas, le meilleur sous-tableau se termine en j ($h=j$). Pour avoir la valeur de g, il suffit de mémoriser l'indice de début de la meilleure somme terminant en $j-1$.

Résumé

Une étape d'itération est basée sur les actions suivantes :

- Mettre à jour l'indice k et la valeur de la meilleure somme $S[j-1]$ terminant en $j-1$
- Ajouter $t[j]$ à $S[j-1]$
- Si on obtient une meilleure valeur alors mettre à jour M, g et d
- Passer à l'étape suivante j

Initialisation

Initialement on a : $(k, j, M, S) \leftarrow (1, 1, \infty, \infty)$

Preuve de l'algorithme

Il est clair que l'hypothèse de travail P décrite précédemment est un invariant de boucle. Pour le prouver, on peut calculer les post-assertions conséquentes aux différentes actions (initialisation et corps de boucle) :

- initialisation : $P(1)$ est vraie car $j-1=0$: le sous-tableau examiné est vide
- on entre dans la boucle avec $P(j-1)$
- « tout juste » avant la dernière instruction du corps de boucle : la propriété P reste la même mais elle s'applique à j au lieu de $j-1$ (on a traité $t[j]$)
- après incrémentation `j<-j+1` : on retrouve l'assertion initiale

Convergence de l'algorithme

A chaque étape on traite $t[j]$ puis on passe à l'élément suivant $t[j+1]$. Le tableau est de dimension finie. Nécessairement, le critère d'arrêt sera atteint.

4 Conclusion

Les « passionnés » pourront s'intéresser au « casse-tête » suivant qui est une généralisation du problème étudié ici : déterminer le sous-tableau rectangulaire `t[r1..r2,c1..c2]` de somme optimale parmi tous les sous-tableaux possibles d'une matrice à deux dimensions `t[1..n,1..p]`.

5 Références générales

Ce problème a été présenté par GRENANDER (voir [Grenander-R1]) comme une alternative au problème similaire mais beaucoup plus difficile traitant des matrices rectangulaires de format $n \times p$, problème dont la réponse est liée à certaines méthodes et stratégies de codage, de digitalisation d'images graphiques et autres.

Grenander-R1, pp 865-871 @ArticleGrenander-R1, author = Grenander, title = , journal = Communication of the ACM, year = 1984,