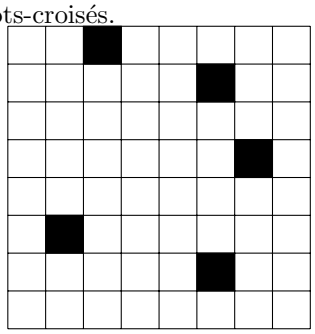
Université Savoie Mont Blanc, UFR SceM
INFO504, L3 INFO

October 24, 2023

Problème 2, Plus grand rectangle

(Tableaux bi-dimensionnels, boucles, structures, optimisation)

On cherche un algorithme pour trouver le plus grand rectangle "blanc" que l'on peut trouver dans une grille type mots-croisés.



Les tableaux à deux dimensions (ou plus) sont possibles en C, avec une syntaxe `T[i][j]` pour accéder à la i-ème ligne et j-ème colonne. Attention cependant, en mémoire les cases mémoires sont placées de manière contigüe, comme un tableau mono-dimensionnel. De plus le nombre de colonnes doit être connu dès la compilation. Ainsi, les indices de lignes sautent en mémoire toutes les cases d'une ligne d'un coup. On peut le vérifier sur l'exemple suivant:

```c
// 8 lignes et 16 colonnes
#define M 8
#define N 16
typedef char Grille[M][N]; // chaque ligne fait 16 octets
Grille T;                  // 8*16 variables de type char créées.
printf("%p\n", &T[0][0] ); // une adresse A
printf("%p\n", &T[0][1] ); // l'adresse A+1
printf("%p\n", &T[0][2] ); // l'adresse A+2
printf("%p\n", &T[1][0] ); // l'adresse A+16
printf("%p\n", &T[1][1] ); // l'adresse A+17
printf("%p\n", &T[1][2] ); // l'adresse A+18
printf("%p\n", &T[2][0] ); // l'adresse A+32
printf("%p\n", &T[1][N] ); // l'adresse A+32 aussi !!
```

Par convention, on peut mettre 0 pour vide et 1 pour plein.

Question 1. Rectangle vide

Ecrire une fonction qui retourne vrai si le rectangle spécifié est vide:

```c
// 0 <= i1 <= i2 < M,  0 <= j1 <= j2 < N,
int est_vide( Grille T, int i1, int j1, int i2, int j2 );
```

```c
int est_vide( Grille G, int i1, int j1, int i2, int j2 )
{
  for ( int i = i1; i <= i2; i++ )
    for ( int j = j1; j <= j2; j++ )
      if ( G[ i ][ j ] != 0 ) return 0;
  return 1;
}
```

Question 2. Type Rectangle

C'est pénible de toujours manipuler 4 paramètres. On propose donc de faire une structure `Rectangle` ainsi:

```c
typedef struct {
  int i1, j1, i2, j2;
} Rectangle;
```

Réécrire la fonction précédente avec maintenant le prototype:

```c
int est_vide( Grille T, Rectangle* R );
```

Faut-il passer le rectangle par valeur ou par adresse ?

Ecrivez aussi une fonction qui calcule l'aire d'un rectangle.

```c
int aire( Rectangle* R );
```

Même si on ne va pas modifier les données du `Rectangle`, on passe les structures par adresse pour économiser une copie de toutes les données dans une variable locale (ici 16 octets). Un autre exemple est la grille, qui est elle aussi passée par adresse car c'est un tableau.

```
int est_vide( Grille G, Rectangle* R )
{
  for ( int i = R->i1; i <= R->i2; i++ )
    for ( int j = R->j1; j <= R->j2; j++ )
      if ( G[ i ][ j ] != 0 ) return 0;
  return 1;
}
int aire( Rectangle* R )
{
  return ( R->i2 - R->i1 + 1 ) * ( R->j2 - R->j1 + 1 );
}
```

Question 3. Plus grand rectangle à une position donnée

Proposez maintenant une fonction qui retourne un rectangle de coin inférieur (i, j) de plus grande aire et qui ne contient aucune case noire.

```
Rectangle plus_grand_rectangle_ij( Grille T, int i, int j );
```

```
Rectangle plus_grand_rectangle_ij_v1( Grille T, int i, int j )
{
  Rectangle Rmax = { i, j, i, j };
  Rectangle Rcur = { i, j, i, j };
  int       jMax = N-1;
  for ( int i2 = i; i2 < M; ++i2 ) {
    Rcur.i2 = i2;
    for ( int j2 = j; j2 < N; ++j2 ) {
      Rcur.j2 = j2;
      if ( est_vide( T, &Rcur ) ) {
        if ( aire( &Rcur ) > aire(&Rmax ) ) Rmax = Rcur;
      }
    }
  }
  return Rmax;
}
```

Question 4. Plus grand rectangle

Ecrire enfin une fonction qui retourne un rectangle de plus grande aire et qui ne contient aucune case noire.

```
Rectangle plus_grand_rectangle( Grille T );
```

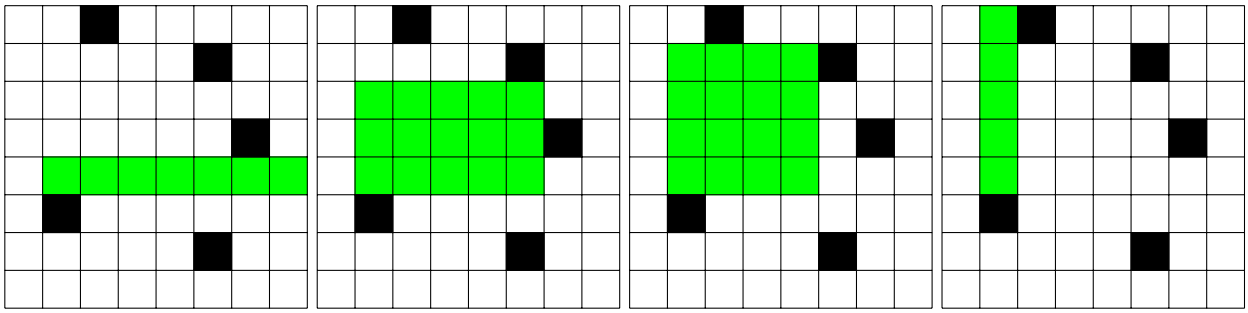
Remarques : Est-ce une fonction efficace ? Si on double M et N, est-ce que le programme s'exécutera 1x, 2x, 4x, 8x, 16x, ..., plus lentement ?

```
Rectangle plus_grand_rectangle_v1( Grille T )
{
  Rectangle Rmax = { 0, 0, 0, 0 };
  Rectangle Rcur;
  for ( int i = 0; i < M; ++i ) {
    for ( int j = 0; j < N; ++j ) {
      Rcur = plus_grand_rectangle_ij_v1( T, i, j );
      if ( aire( &Rcur ) > aire(&Rmax ) ) Rmax = Rcur;
    }
  }
  return Rmax;
}
```

La fonction `plus_grand_rectangle_v1` fait une double boucle et effectue MN appels à `plus_grand_rectangle_ij_v1`. Celle-ci fait à chaque fois entre 1 et MN itérations, chacune ayant un coût proportionnel à la taille du rectangle testé. On arrive à une complexité en temps de l'ordre de M^3N^3. Si la grille a très peu de cases occupées, le temps d'exécution peut être multiplié par 64 si on double M et N !

En pratique on observe un temps d'exécution multiplié par 6-7.

Question 5. Plus grand rectangle à une position donnée, avec décroissance



On constate que, à (i, j) fixé, plus il y a de lignes dans le rectangle, moins il peut y avoir de colonnes. On peut donc réécrire la fonction `plus_grand_rectangle_ij` en utilisant la colonne la plus lointaine trouvée à la ligne précédente.

```
Rectangle plus_grand_rectangle_ij_v2( Grille T, int i, int j );
```

```
Rectangle plus_grand_rectangle_ij_v2( Grille T, int i, int j )
{
  Rectangle Rmax = { i, j, i, j };
  Rectangle Rcur = { i, j, i, j };
  int       jMax = N-1;
  for ( int i2 = i; i2 < M; ++i2 ) {
    Rcur.i2 = i2;
    for ( int j2 = j; j2 <= jMax; ++j2 ) {
      Rcur.j2 = j2;
      if ( est_vide( T, &Rcur ) ) {
        if ( aire( &Rcur ) > aire( &Rmax ) ) Rmax = Rcur;
      } else { // plus besoin d'aller voir aussi à droite
        jMax = j2-1;
        break;
      }
    }
    if ( jMax < j ) break;
  }
  return Rmax;
}
```

Il est plus compliqué d'estimer le gain de temps d'exécution, car il dépend du nombre de cases occupées (plus il est important, plus cette approche est meilleure que la première méthode naïve). En pratique on observe un temps d'exécution divisé par 3 ou 4.

Question 6. Test d'un rectangle vide

Chaque test de rectangle vide est assez cher (temps proportionnel à la taille du rectangle testé, dans le pire cas). Pour être plus efficace, on propose de faire un calcul plus compliqué a priori. On va plutôt compter le nombre de cases occupées dans le rectangle spécifié ! Le rectangle est donc vide si ce nombre est zéro.

Si on voit la grille G comme une fonction du plan, on est en train de calculer la somme des valeurs de G sur le rectangle choisi. Autrement dit c'est un calcul d'intégrale. Or il existe des formules pour transformer une intégrale de domaine en intégrale sur le bord du dommaine, notamment la formule de Green.

C'est encore plus simple dans le cas d'une grille et d'une somme sur un rectangle. La formule de Green se traduit par un précalcul par ligne sur G. Posez-vous la question sur une ligne. Comment compter efficacement le nombre de cases occupées entre 2 bornes `j1` et `j2`, en utilisant un précalcul qui prend en mémoire $N+1$ données ? Cela peut se faire en temps constant !

Pour la grille entière, ce précalcul construit un tableau `A` de taille $M*(N+1)$. Ensuite chaque appel à `est_vide_green` prendra un temps proportionnel au nombre de lignes du rectangle seulement.

L'idée est de calculer la somme `S` des valeurs de `G` jusqu'à la case (exclue). Ensuite la différence entre `S[j2+1]` et `S[j1]` donne le résultat:

```
        i1                    j1
G: 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0
S: 0 0 0 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 5 5 5
S[ j1 ] - S[ i1 ] = 3 - 1 = 2 // ok
```

```
void Fx( Area Ax, Grille G )
{
  for ( int i = 0; i < M; i++ )
    Ax[ i ][ 0 ] = 0;
  for ( int i = 0; i < M; i++ )
    for ( int j = 0; j < N; j++ )
      Ax[ i ][ j+1 ] = Ax[ i ][ j ] + ( G[ i ][ j ] != ' ' ? 1 : 0 );
}
```

```
int est_vide_Ax( Grille G, Area Ax, Rectangle* R )
{
  int nb = 0;
  for ( int i = R->i1; i <= R->i2; i++ )
    nb += Ax[ i ][ R->j2+1 ] - Ax[ i ][ R->j1 ];
  return nb == 0;
}
```

Il suffit alors de faire le précalcul par un appel à `Fx` puis d'appeler `est_vide_green` au lieu de `est_vide`. On vérifie alors que le temps d'exécution n'est que multiplié par 8 environ lorsqu'on double M et N.

On peut faire encore mieux ! Comme notre domaine est rectangulaire, on peut aussi intégrer/sommer l'information de `A` par colonne. Ensuite l'accès à seulement 4 valeurs du tableau permet de déterminer combien de cases pleines a un rectangle. La fonction `est_vide` prend alors un temps constant très court, indépendant de la taille de la grille ou de son occupation !

```
                                                 j1      j2+1
    j1    j2                                 0 0 0 0 0 0 0 0 0
  0 1 0 0 1 0 0 0   0 0 1 1 1 2 2 2 2   i1 0 0{1}1 1 2(2)2 2
i1 1 0 0 0 1 1 0 0   0 1 1 1 1 2 3 3 3      0 1 2 2 2 4 5 5 5
  0 0 0 0 0 1 0 0   0 0 0 0 0 0 1 1 1      0 1 2 2 2 4 6 6 6
  0 1 0 0 0 0 1 0   0 0 1 1 1 1 1 2 2      0 1 3 3 3 5 7 8 8
i2 0 0 1 0 0 1 0 0   0 0 0 1 1 1 2 2 2 i2+1 0 1(3)4 4 6{9}A A
    Grille G            Area Ax               Area Axy (hexa)

=> Nb_cases_pleines( i1, j1, i2, j2 ) = {9} - (2) - (3) + {1} = 5
```

```
void Fxy( Area Axy, Grille G )
{
  for ( int i = 0; i <= M; i++ )
    Axy[ i ][ 0 ] = 0;
  for ( int j = 0; j <= N; j++ )
    Axy[ 0 ][ j ] = 0;
  for ( int i = 0; i < M; i++ )
    for ( int j = 0; j < N; j++ )
      Axy[ i+1 ][ j+1 ] = Axy[ i+1 ][ j ] + ( G[ i ][ j ] != ' ' ? 1 : 0 );
  for ( int j = 0; j <= N; j++ )
```

```
    for ( int i = 0; i < M; i++ )
      Axy[ i+1 ][ j ] += Axy[ i ][ j ];
}
int est_vide_Axy( Grille G, Area Axy, Rectangle* R )
{
  int nb = Axy[ R->i2 + 1 ][ R->j2 + 1 ]
    - Axy[ R->i1 ][ R->j2 + 1 ]
    - Axy[ R->i2 + 1 ][ R->j1 ]
    + Axy[ R->i1 ][ R->j1 ];
  return nb == 0; // aisément modifiable en nb <= cste
}
```

Question 7. Meilleur algorithme possible

Réfléchissez (ou cherchez sur le net) le meilleur algorithme possible pour calculer le plus grand rectangle.

La méthode proposée au-dessus n'est toujours pas optimale pour ce problème donné. On peut trouver une approche en $O(MN)$ (contre $O(M^2N)$ plutôt ici). Elle se fait en précalculant des informations par case: nombre de cases libres à droite et en haut entre autres.

Néanmoins notre approche est plus générale, car elle peut traiter un problème du genre: chaque case a un coût (un nombre ≥ 0) et on cherche le plus grand rectangle tel que son coût total ne dépasse pas un coût x donné. Par exemple, on veut installer un stade de foot dans une ville et on doit acheter/raser des parcelles à un certain coût. Cet algorithme vous donne une solution à ce problème.