Conception algorithmiques

TD *Programmation dynamique*

1 Pour débuter

Exercice 1.1. Sous-séquence de plus grande somme.
On dispose d'un tableau d'entiers *relatifs* de taille n. On cherche à déterminer la suite d'entrées consécutives du tableau dont la somme est maximale. Par exemple, pour le tableau $T = [5, 15, -30, 10, -5, 40, 10]$, la somme maximale est 55 (somme des éléments $[10, -5, 40, 10]$).

1. Dans un premier temps, on s'intéresse uniquement à la valeur de la somme de cette sous-séquence.

a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. En déduire la valeur de la somme maximale.

d. Écrire un algorithme de programmation dynamique pour calculer cette somme.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les indices de début et de fin de la somme. Sa complexité est-elle modifiée ?

Solution:

1. a. $S(i)$: + grande somme se terminant exactement à la position i.

b. $S(i) = \max(S(i-1) + T[i], T[i])$ $\qquad$ $S(1) = T[1]$.

c. $S = \max_i(S(i))$

d.
```
SSPGS_ite(T)
    n <- lg(T);
    S <- créerTab([1..n]);
    S[1] <- T[1];
    max <- T[1];

    pour i de 2 à n faire
      S[i] <- max(T[i],T[i]+S[i-1]);
      si S[i]>=max alors
        max <- S[i];
      finsi
    fin pour

    retourner max;
  fin

  SSPGS_rec(T)
    n <- lg(T);
    S <- créerTab([1..n],-∞);
```

```
    S[1] <- T[1];

    procédure aux(i)
      si S[i]=-∞ alors
        S[i] <- max(T[i],T[i]+aux[i-1]);
      finsi
      retourner S[i];
    fin

    aux(n);

    max <- S[1];
    pour i de 2 à n faire
      si S[i]>=max alors
        max <- S[i];
      finsi
    retourner max;
  fin
```

e. La complexité est en $\Theta(n)$.

2.
```
indices_SSPGS_ite(T)
  n <- lg(T);
  S <- créerTab([1..n]);
  ind <- créerTab([1..n]);
  S[1] <- 1;
  ind[1] <- 1;
  max <- T[1];
  indices <- (1,1);

  pour i de 2 à n faire
    si T[i]>=T[i]+S[i-1] alors
      S[i] <- T[i];
      ind[i] <- i;
    sinon
      S[i] <- T[i]+S[i-1];
      ind[i] <- ind[i-1];
    finsi
    si S[i]>=max alors
      max <- S[i];
      indices <- (ind[i],i);
    finsi
  fin pour

  retourner indices;
fin
```

La complexité est toujours en $\Theta(n)$.

Exercice 1.2. Plus grande sous-suite croissante.
Soit une séquence de nombre $s = x_1, ..., x_n$. Une sous-suite est un sous-ensemble de ces nombres pris dans l'ordre, c'est à dire une séquence de la forme : $x_{i_1}, x_{i_2}, ..., x_{i_k}$ telle que $1 \leq i_1 < i_2 < ... < i_k \leq n$. Une sous-suite est dite croissante si les x_i sont croissants.
On cherche à détérminer la plus longue sous-suite croissante de s. Par exemple, pour (2,1,7,5,2,4,8,6), une solution est (1,2,4,6).

1. Dans un premier temps, on s'intéresse uniquement à la longueur de cette sous-suite.
   a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?
   b. Caractériser (par une équation) cette sous-structure optimale.
   c. En déduire la longueur de la sous-suite croissante maximale.
   d. Écrire un algorithme de programmation dynamique pour calculer longueur.
   e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les indices des éléments de la suite. Sa complexité est-elle modifiée ?

Solution:

1. a. $L(i)$: longueur de la + longue sous-suite s'arrêtant à x_i.

b. $L(i) = 1 + \underset{x_j \leq x_i}{\text{Max}}(L(j)) \qquad L(1) = 1.$

c. $L = \underset{i}{\text{Max}}(L(i))$

d.
```
PLSSC_ite(x)
    n <- lg(x);
    L <- créerTab([1..n]);
    L[1] <- 1;
    max <- 1;
    pour i de 2 à n faire
      max_i <- 1;
      pour j de 1 à i-1 faire
        si x_j ≤ x_i alors
          si 1+L[j]>max_i alors
            max_i <- L[j]+1;
          fin si
        fin si
      fin pour
      L[i] <- max_i;

      si L[i]>max alors
        max <- L[i];
      finsi
    fin pour
    retourner max;
  fin
```

```
PLSSC_rec(x)
  n <- lg(x);
  L <- créerTab([1..n],-∞);
  L[1] <- 1;

  procédure aux(i)
    si L[i]=-∞ alors
      max_i <- 1;
      pour j de 1 à i-1 faire
        si x_j ≤ x_i alors
          si 1+aux(j)>max_i alors
            max_i <- aux(j)+1;
          fin si
        fin si
      fin pour
      L[i] <- max_i;
    fin si
    retourner L[i];
  fin

  aux(n);

  max <- 1;
  pour i de 2 à n faire
    si L[i]>max alors
      max <- L[i];
    finsi
  fin pour
  retourner max;
fin
```

e. La complexité est en $\Theta(n^2)$

2.

```
indices_PLSSC_ite(x)
  n <- lg(x);
  L <- créerTab([1..n]);
  ind <- créerTab([0..n]);
  L[1] <- 1;
  ind[0] <- listeVide;
  ind[1] <- créerListe(1);
  max <- 1;
  indices <- ind[1];
  pour i de 2 à n faire
    max_i <- 1;
    ind_i <- 0;
    pour j de 1 à i-1 faire
      si x_j ≤ x_i alors
        si 1+L[j]>max_i alors
          max_i <- L[j]+1;
```

```
            ind_i <- j;
          fin si
        fin si
      fin pour
      L[i] <- max_i;
      ind[i] <- ajouteListe(i,ind[ind_i]);

      si L[i]>max alors
        max <- L[i];
        indices <- ind[i];
      finsi
    fin pour
    retourner indices;
  fin
```

La complexité est toujours en $\Theta(n^2)$.

Exercice 1.3. Plus grande sous-chaîne commune.
Étant données deux chaînes de caractères $x = x_1x_2...x_n$ et $y = y_1y_2...y_m$, on cherche à déterminer la plus longue sous-chaîne commune (lettres contigües). Cela revient à dire que l'on cherche des indices i, j et k tels $x_ix_{i+1}...x_{i+k-1} = y_jy_{j+1}...y_{j+k-1}$ avec k maximal.

1. Dans un premier temps, on s'intéresse uniquement à la longueur de cette sous-chaîne commune.

a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. En déduire la longueur maximale d'une sous-chaîne commune.

d. Écrire un algorithme de programmation dynamique pour calculer cette longueur maximale.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également i,j et k. Sa complexité est-elle modifiée ?

Solution:

1. a. $L(i,j)$: la longueur de la + longue sous-chaîne s'arrêtant en x_i et y_j.

b. $L(i,j) = \begin{cases} 0 & \text{si } x_i \neq y_j \\ 1 + L(i-1,j-1) & \text{sinon} \end{cases}$

c. $L = \max_{i,j}(L(i,j))$

d.
```
PGSCC_ite(x,y)        \\les indices de x et y commencent à 1.
    n_x <- lg(x);
    n_y <- lg(y);
    L <- créerTab([0..n_x][0..n_y],0);
    max <- 0;

    pour i de 1 à n_x faire
```

```
      pour j de 1 à n_y faire
        si x_i = x_j alors
          L[i,j] <- L[i-1,j-1]+1;
        fin si

        si L[i,j]>max alors
          max <- L[i,j];
        finsi
      fin pour
    fin pour

    retourner max;
  fin

  PGSCC_rec(x,y)      \\les indices de x et y commencent à 1.
    n_x <- lg(x);
    n_y <- lg(y);
    L <- créerTab([0..n_x][0..n_y],-∞);

    procédure aux(i,j)
      si L[i,j]=-∞ alors
        si x_i = x_j alors
          L[i,j] <- aux(i-1,j-1)+1;
        sinon
          L[i,j] <- 0;
        fin si
      finsi
      retourner L[i,j];
    fin

    max <- 0;
    pour i de 1 à n_x faire
      pour j de 1 à n_y faire
        si aux(i,j)>max
          max <- L[i,j];
        fin si
      fin pour
    fin pour

    retourner max;
  fin
```

e. La complexité est en $\Theta(mn)$

2.
```
indices_PGSCC_ite(x,y) \\les indices de x et y commencent à 1.
Entrée: ...
Sortie: (i,j,k);
  n_x <- lg(x);
  n_y <- lg(y);
```

```
L <- créerTab([0..n_x][0..n_y],0);
L[0,0] <- 0;
max <- 0;
indices <-(0,0);

pour i de 1 à n_x faire
  pour j de 1 à n_y faire
    si x_i = x_j alors
      L[i,j] <- L[i-1,j-1]+1;
    fin si

    si L[i,j]>max alors
      max <- L[i,j];
      indices <- (i,j);
    finsi
  fin pour
fin pour

retourner (i-max+1,j-max+1,max);
fin
```

La complexité est toujours en $\Theta(mn)$

Exercice 1.4. Sac-à-dos, avec répétitions.
Lors du cambriolage d'une bijouterie, le voleur s'aperçoit qu'il ne peut pas tout emporter dans son sac-à-dos... Son sac peut supporter, au maximum, P kilos de marchandise (on supposera P entier). Or, dans cette bijouterie, se trouvent n objets différents, chacun en quantité illimitée. Chaque objet x_i à un poids p_i et une valeur v_i.
Quelle combinaison d'objets, transportable dans le sac, sera la plus rentable pour notre voleur ?
Exemple : pour $P = 10$ et les objets suivants :

objet	poids	valeur
x_1	6	30 €
x_2	3	14 €
x_3	4	16 €
x_4	2	9 €

le meilleur choix est de prendre un x_1 et deux x_4, ce qui fait un sac à 48 €.

1. Dans un premier temps, on s'intéresse uniquement à la valeur maximale que pourra emporter le voleur.
   - **a.** Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?
   - **b.** Caractériser (par une équation) cette sous-structure optimale.
   - **c.** En déduire le montant maximal des objets volés.
   - **d.** Écrire un algorithme de programmation dynamique pour calculer cette valeur maximale.
   - **e.** Quelle est la complexité de cet algorithme ?
2. Modifier l'algorithme pour qu'il renvoie également la combinaison d'objets volés. Sa complexité est-elle modifiée ?

Solution:

1. a. $S(p)$: la valeur maximum atteinte avec un sac de capacité p.

b. $S(p) = \underset{i|p_i \leq p}{\text{Max}}(S(p - p_i) + v_i)$

c. $S = S(P)$

d.
```
SADR_ite(P,poids,valeur)
entrées: la capacité P du sac, les tableaux poids et valeur.
  n <- lg(poids);
  S <- créerTab([0..P],0);

  pour p de 1 à P faire
    max_p <- 0;
    pour i de 1 à n faire
      si poids[i]<=p alors
        si S[p-poids[i]]+valeur[i]>max_p alors
          max_p <- S[p-poids[i]]+valeur[i];
        finsi
      fin si
    fin pour
    S[p] <- max_p;
  fin pour

  retourner S[P];
fin
```

e. La complexité est en $\Theta(P \times n)$

2.
```
indices_SADR_ite(P,poids,valeur)
entrées: la capacité P du sac, les tableaux poids et valeur.
  n <- lg(poids);
  S <- créerTab([0..P],0);
  ind <- créerTab([0..P],0);
  indices <- listeVide;

  pour p de 1 à P faire
    max_p <- 0;
    pour i de 1 à n faire
      si poids[i]<=p alors
        si S[p-poids[i]]+valeur[i]>max_p alors
          max_p <- S[p-poids[i]]+valeur[i];
          ind[p] <- i;
        finsi
      fin si
    fin pour
    S[p] <- max_p;
  fin pour

  p <- P;
```

```
    tantque p≠0 faire
      indices <- ajouteListe(ind[p], indices);
          p <- p-poids[ind[p]];
    fin tantque

    retourner indices;
  fin
```

Exercice 1.5. Ensembles indépendants dans des arbres.
Un sous-ensemble de noeuds dans un graphe $G = (V, E)$ est dit *indépendant* s'il n'existe pas d'arrête entre eux. (exemple) Trouver le plus grand ensemble indépendant dans un graphe est un problème difficile. Par contre, si le graphe est un arbre, c'est un sous-problème plus facile. On cherche donc l'ensemble indépendant de plus grande taille dans un arbre A.
On note $r(A)$, la racine de A et $fils(A)$, la liste des fils de A. On pourra églement noter $A(s)$, le sous-arbre de A dont la racine est s.

1. Dans un premier temps, on s'intéresse uniquement à la taille maximale d'un sous-ensemble.
   a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?
   b. Caractériser (par une équation) cette sous-structure optimale.
   c. En déduire la taille maximale d'un ensemble indépendant de A.
   d. Écrire un algorithme de programmation dynamique pour calculer cette taille maximale.
   e. Quelle est la complexité de cet algorithme ?
2. Modifier l'algorithme pour qu'il renvoie également la liste des noeuds. Sa complexité est-elle modifiée ?

Solution:

1. a. $T(s)$: la taille maximale d'un ensemble indépendant du sous-arbre $A(s)$.
   b. $T(s) = \max(\sum_{f \in fils(A(s))} T(f), \sum_{pf \in fils(fils(A(s)))} T(pf) + 1\ )$
   c. $T = T(r(A))$
   d. On suppose que les noeuds et feuilles de l'arbre sont numerotés de 1 à n (avec n la taille de l'arbre) et que fils(A) renvoie liste(0) si A est une feuille.

```
PGSEI_rec(A)
    n <- taille(A);
  T <- créerTab([0..n],-∞);
    T[0] <- 0;

    procédure aux(s)
        si T[s]=-∞ alors
            s1 <- 0; s2 <- 0;
            pour f dans fils(A(s)) faire
                s1 <- s1+aux(f);
            fin pour
```

```
                pour pf dans fils(fils(A(s))) faire
                    s2 <- s2+aux(pf);
                fin pour
                T[s] <- max(s1,s2+1);
            finsi
            retourner T[s]
        fin

        retourner aux(r(A))
    fin
```

e. La complexité est en $\Theta(|V| + |E|)$, i.e.,$\Theta(n)$ pour un arbre de taille n.

2.

```
indices_PGSEI_rec(A)
    n <- taille(A);
  T <- créerTab([0..n],-∞);
  ind <- créerTab([0..n],"");
    T[0] <- 0;

    procédure aux(s)
        si T[s]=-∞ alors
            s1 <- 0; s2 <- 0;
            pour f dans fils(A(s)) faire
                s1 <- s1+aux(f);
            fin pour
            pour pf dans fils(fils(A(s))) faire
                s2 <- s2+aux(pf);
            fin pour
            si s1>s2+1 alors
                T[s] <- s1;
                ind[s] <- "f";
            sinon
                T[s] <- s2+1;
                ind[s] <- "pf";
            finsi
        finsi
        retourner T[s]
    fin

    procédure listeNoeuds(s)
        si s=0 alors
            retourner listeVide
        sinon
            l <- listeVide
            si ind[s]="f" alors
                pour f dans fils(A(s)) faire
                    l <- concatListe(listeNoeuds(f),l);
                fin pour
            sinon
```

```
                pour pf dans fils(fils(A(s))) faire
                    l <- ajouteListe(pf,concatListe(listeNoeuds(pf),l));
                fin pour
            finsi
            retourner l
        finsi
    fin

    aux(r(A))
    retourner listeNoeuds(r(A))
fin
```

Exercice 1.6. Texte corrompu.
Soit $s[1...n]$ un chaîne de caractères sans aucun espace (ex : Ilétaitunefoisuneprincesse...). On cherche à savoir si cette chaîne correspond à un texte lisible dont on aurait effacé les espaces et le cas échéant, on veut pouvoir reconstituer ce texte. Pour cela, on dispose d'un dictionnaire un peu particulier : il s'agit d'une fonction qui, étant donné une chaîne de caractères t quelconque, renvoie VRAI si t représente un mot correct :

$$dict(t) = \begin{cases} VRAI & \text{si } t \text{ est un mot correct} \\ \text{FAUX} & \text{sinon} \end{cases}$$

1. Dans un premier temps, on cherche uniquement à savoir si s correspond à un texte valide.
   a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?
   b. Caractériser (par une équation) cette sous-structure optimale.
   c. Comment détermine-t-on si s correspond à un texte correct ?
   d. Écrire un algorithme de programmation dynamique pour le faire.
   e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les positions des espaces. Sa complexité est-elle modifiée ?

Solution:

1. a. $B(i)$: VRAI si la chaîne $[1...i]$ correspond à un texte correct.

b. $B(i) = \underset{j=1..i-1}{\text{OR}}(B(j) \text{ AND } dict(s[j+1...i]))$

c. $B(n)$

d.
```
TC_ite(s)
    n <- lg(s);
    B <- créerTab([0..n]);
      B[0] <- VRAI;

      pour i de 2 à n faire
          b <- FAUX;
          j <- 0;
```

```
            tantque j<i et non(b) faire
                b <- B[j] et dict(s[j+1..i]);
                j <- j+1;
            fin tantque
            B[i] <- b;
        fin pour

        retourner B[n];
    fin
```

e. La complexité est en $\Theta(n^2)$

2. On suppose qu'il existe une unique façon de reconstruire le texte.

```
indices_TC_ite(s)
    n <- lg(s);
    B <- créerTab([0..n]);
    ind <- créerTab([0..n]);
    B[0] <- VRAI;
    ind[0] <- 0;

    pour i de 1 à n faire
        b <- FAUX;
        j <- 0;
        tantque j<i et non(b) faire
            b <- B[j] et dict(s[j+1..i]);
            j <- j+1;
        fin tantque
        B[i] <- b;
        ind[i] <- j-1;
    fin pour

    indices <- listeVide;
    i <- n;
    tantque i≠0 faire
        indices <- ajouteListe(i,indices);
        i <- ind[i];
    fin tantque;

    retourner indices;
fin
```

2 Ordonnancement

Exercice 2.1. Le but de cet exercice est de trouver un ordonnancement optimal de chaînes de montage. Considèrons un atelier de production comportant deux chaînes de montage. Chaque chaîne comporte n postes, notées $S_{i,j}$ ($i = 1, 2$, $j = 1, \ldots, n$). Les postes $S_{1,j}$ et $S_{2,j}$ font le même travail, mais les temps de montage peuvent varier. Le temps de montage au poste $S_{i,j}$ est $a_{i,j}$.

Une pièce brute arrive sur une chaîne, passe par les postes 1 à n où elle subit un traitement, puis sort par l'autre extrémité de la chaîne de montage. Le temps de passage d'un poste à l'autre est négligeable sur une même chaîne, mais une pièce peut être transférée du poste $S_{i,j}$ au poste $S_{3-i,j+1}$ moyennant un délai $t_{i,j}$. Une pièce met un temps e_i à arriver sur le premier poste de la chaîne i, et x_i à sortir de la chaîne i. Le problème de l'ordonnancement de chaînes de montage

FIGURE 1 – Ordonnancement de chaînes de montage

consiste à déterminer les postes à sélectionner sur les chaînes 1 et 2 pour minimiser le délai de transit d'une pièce à travers tout l'atelier. La complexité des algorithmes sera comptée en nombre d'additions.

1. Structure du chemin optimal
Soit $C_j^1 = [S_{i_1,1}, S_{i_2,2}, \ldots, S_{i_{j-1},j-1}, S_{1,j}]$ un chemin optimal vers le poste $S_{1,j}$, caractérisez les sous-chemins $C_{j-1} = [S_{i_1,1}, S_{i_2,2}, \ldots, S_{i_{j-1},j-1}]$ possibles. Caractérisez de même les chemins optimaux vers le poste $S_{2,j}$.

2. Solution récursive
Notons $f_{i,j}$ le délai le plus court possible avec lequel une pièce sort du poste $S_{i,j}$, et f^* le plus court délai pour qu'une pièce traverse tout l'atelier. Exprimez f^* en fonction des $f_{i,j}$. Ecrivez une relation de récurrence vérifiée par $f_{i,j}$.

3. Calcul du temps optimal
Quelle serait la complexité d'un algorithme récursif calculant f^* à partir des relations de récurrence précédentes ? Donnez un algorithme utilisant de la programmation dynamique pour calculer f^*. Quelle est sa complexité dans le pire cas ?

4. Construction du chemin optimal
Donnez un algorithme renvoyant la séquence des postes utilisés par un chemin optimal traversant l'atelier. Quelle est sa complexitè dans le pire cas ?

Solution:

1. On note $C_j^1 = [S_{i_1,1}, S_{i_2,2}, \ldots, S_{i_{j-1},j-1}, S_{1,j}]$ le chemin optimal vers $S_{1,j}$ et $C_j^2 = [S_{i_1,1}, S_{i_2,2}, \ldots, S_{i_{j-1},j-1}, S_{2,j}]$ et le chemin optimal vers $S_{2,j}$. Pour C_j^1, les sous-chemins possibles sont soit C_{j-1}^1, soit C_{j-1}^2, les sous-chemins optimaux vers $S_{1,j-1}$ et $S_{2,j-1}$ (si les sous-chemins ne sont pas optimaux, ca veut dire qu'il y a une facon plus rapide d'arriver en $j-1$ et donc le chemin C_j^1 ne pourra pas être optimal).
idem pour C_j^2.

2.

$$\begin{aligned} f_{1,1} &= e_1 + a_{1,1} \\ f_{1,j} &= min(\ f_{1,j-1} + a_{1,j}\ ,\ f_{2,j-1} + t_{2,j-1} + a_{1,j}\ ) \\ f_{2,1} &= e_2 + a_{2,1} \\ f_{2,j} &= min(\ f_{2,j-1} + a_{2,j}\ ,\ f_{1,j-1} + t_{1,j-1} + a_{2,j}\ ) \\ f^* &= min(\ f_{1,n} + x_1\ ,\ f_{2,n} + x_2\ ) \end{aligned}$$

3. En dessinant l'arbre des appels récursifs en partant de f^*, et en considérant que le coût à chaque noeud est $O(1)$, on voit qu'on a une complexité en $O(2^{n+1})$. Voici 2 algorithmes de programmation dynamique, l'un itératif, l'autre récursif, de complexité linéaire.

```
CoutOptimalIte(a1,a2,t1,t2,e1,e2,x1,x2,n)
Entrée: ...
Sortie: f*

    f1 ← CreerTableau [1..n];
    f2 ← CreerTableau [1..n];
    f1[1] ← e1 + a1[1];
    f2[1] ← e2 + a2[1];
    pour j de 2 à n faire
        f1[j] ← min( f1[j − 1] + a1[j] , f2[j − 1] + t2[j − 1] + a1[j] );
        f2[j] ← min( f2[j − 1] + a2[j] , f1[j − 1] + t1[j − 1] + a2[j] );
    finpour
    retourne min( f1[n] + x1 , f2[n] + x2 )
fin

CoutOptimalRec(a1,a2,t1,t2,e1,e2,x1,x2,n)
Entrée: ...
Sortie: f*

    f1 ← CreerTableau [1..n];
    f2 ← CreerTableau [1..n];
    f1[1] ← e1 + a1[1];
    f2[1] ← e2 + a2[1];

    Procedure F1(z)
        si z=1 alors retourne f1[1] finsi
        si f1[z − 1]=nil alors f1[z − 1] ← F1(z − 1) finsi
        si f2[z − 1]=nil alors f2[z − 1] ← F2(z − 1) finsi
        f1[z] ← min( f1[z − 1] + a1[z] , f2[z − 1] + t2[z − 1] + a1[z] );
        retourne f1[z]
    Fin

    Procedure F2(z)
        si z=1 alors retourne f2[1] finsi
        si f1[z − 1]=nil alors f1[z − 1] ← F1(z − 1) finsi
        si f2[z − 1]=nil alors f2[z − 1] ← F2(z − 1) finsi
        f2[z] ← min( f2[z − 1] + a2[z] , f1[z − 1] + t1[z − 1] + a2[z] );
        retourne f2[z]
    Fin

    retourne min( F1[n] + x1 , F2[n] + x2 )
fin
```

4. Cet algorithme à une complexité linéaire.

```
CheminOptimalIte(a1,a2,t1,t2,e1,e2,x1,x2,n)
Entrée: ...
Sortie: seq, un tableau contenant les numeros des chaînes des
        postes de la séquence optimale.
```

```
    f1 ← CreerTableau [1..n];
    f2 ← CreerTableau [1..n];
    p1 ← CreerTableau [2..n];
    p2 ← CreerTableau [2..n];
    f1[1] ← e1 + a1[1];
    f2[1] ← e2 + a2[1];
    pour j de 2 à n faire
        si f1[j − 1] + a1[j] < f2[j − 1] + t2[j − 1] + a1[j] alors
            f1[j] ← f1[j − 1] + a1[j];
            p1[j] ← 1;
        sinon
            f1[j] ← f2[j − 1] + t2[j − 1] + a1[j];
            p1[j] ← 2;
        finsi
        si f2[j − 1] + a2[j] < f1[j − 1] + t1[j − 1] + a2[j] alors
            f2[j] ← f2[j − 1] + a2[j];
            p2[j] ← 1;
        sinon
            f2[j] ← f1[j − 1] + t1[j − 1] + a2[j];
            p2[j] ← 2;
        finsi
    finpour
    si f1[n] + x1 < f2[n] + x2 alors
        seq[n] ← 1;
    sinon
        seq[n] ← 2;
    finsi
    pour i de n-1 à 1 faire
        si seq[i+1]=1 alors
            seq[i] ← p1[i+1];
        sinon
            seq[i] ← p2[i+1];
        finsi
    finpour
    retourne seq;
fin
```

3 Géométrie algorithmique

Exercice 3.1. Le but de cet exercice est de trouver une triangulation de polygones optimale pour une fonction de pondération donnée Etant donnés $n \geq 3$ points du plan $v_1, \ldots, v_n$, le *polygone* $v_1 \cdots v_n$ est la figure constituée des n segments $[v_1v_2]$, $[v_2v_3]$, $\ldots$, $[v_{n-1}v_n]$, $[v_nv_1]$ (voir les exemples de la figure 3.1). Les v_i sont les *sommets* du polygone, et les segments $[v_iv_{i+1}]$ sont les *côtés* du polygone. Les segments $[v_iv_k]$ qui ne sont pas des côtés sont appelés des *cordes* du polygone. Un polygone est *simple* si ses côtés ne se coupent pas, et croisé sinon.

FIGURE 2 – Polygones à 7 sommets. (a) croisé (b) simple (c) convexe

Un polygone simple découpe le plan en deux parties : l'intérieur et l'extérieur du polygone. Un polygone simple est *convexe* si toutes ses cordes sont à l'intérieur du polygone.

Dans cet exercice nous ne considérons que des polygones convexes. Une triangulation d'un polygone (convexe) est un ensemble T de cordes qui divisent le polygone en triangles disjoints et ne se coupent pas. La figure suivante montre deux triangulations possibles d'un polygone à 7 côtés.

1. Montrer qu'une triangulation d'un polygone à n côtés a $(n-3)$ cordes et $(n-2)$ triangles.

2. Structure du chemin optimal, Solution récursive

Soit $P = v_1 \cdots v_n$ un polygone convexe, et w une fonction de pondération définie sur les triangles formés par les côtés et les cordes de P. Un exemple naturel de fonction de pondération est le périmètre du triangle :

$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

où $|v_i v_j|$ est la distance euclidienne entre les sommets v_i et v_j.
À toute triangulation T de P on peut donc associer une pondération totale, qui est égale à la somme des pondérations des triangles de T. Le problème de la triangulation *optimale* du polygone P pour la pondération w est de trouver une triangulation de pondération totale minimale.
Notons $t[i, j]$ la pondération totale d'une triangulation optimale du polygone $v_i \cdots v_j$, pour $1 \leq i < j \leq n$, et posons $t[i, i+1] = 0$ pour tout $1 \leq i \leq n$. La pondération totale d'une triangulation optimale de P vaut $t[1, n]$.

Soit T une triangulation optimale de P, et soit v_k, $2 \leq k \leq n-1$ le troisième sommet du triangle contenant v_1 et v_n. Donnez une formule reliant la pondération totale de T à la pondération $w(v_1, v_k, v_n)$ et aux pondérations des deux sous-polygones $v_1 \cdots v_k$ et $v_k \cdots v_n$ pour la triangulation T. Déduisez-en une relation de récurrence vérifiée par la fonction t.

3. Calcul du temps optimal, Construction d'une triangulation optimale
Donnez un algorithme calculant la pondération totale d'une triangulation optimale d'un polygone P pour une fonction de pondération w. Quelle est sa complexité dans le pire cas en nombre d'appels à la fonction w ? Donnez un algorithme retournant effectivement une triangulation optimale.

4. Si la fonction de poids est quelconque, combien faut-il de valeurs pour la définir sur tout triangle du polygone ? Comparez avec la complexité obtenue.

5. Si le poids d'un triangle est égal à son aire, que pensez-vous de l'algorithme que vous avez proposé ?

Solution:

1. *Remarque* : Si on coupe un polygone convexe en 2, il reste convexe.
Par recurrence :
- *cas de base* : $n = 3$, on a bien 0 corde et 1 triangle.
- *cas général* : $n \geq 4$,
  hypothèse de récurrence : on suppose la propriété vraie pour tout $n_0 < n$.
  Soit G un polygone convexe à n points. On découpe G par un corde c quelconque. On obtient 2 polygones P et Q à p (resp. q) points tels que $p+q = n+2$. Une triangulation de G est alors obtenue comme la réunion des triangulations de P et Q. Par hypothèse de récurrence, cette triangulation a donc $p-3+q-3+1 = n-3$ cordes (le $+1$ pour c) et $p-2+q-2 = n-2$ triangles. □

2.

$$\begin{aligned} T = t[1,k] &= w(v_1, v_k, v_n) + t[1,k] + t[k+1,n] \\ \underset{1 \leq i < j \leq n}{t[i,j]} &= \min_{k \in]i,j[} (w(v_i, v_k, v_j) + t[i,k] + t[k,j]) \end{aligned}$$

3. Complexité : $O(n^3)$ en temps et $O(n^2)$ en espace.

```
PonderationOptimale(P)
Entrée: tableau contenant les coordonnées des points de P.
Sortie: la valeur de la pondération optimale.

    n ← longueur(P);
    A ← CreerTableau [1..n];
    pour i de 1  n
        A[i][i+1] ← 0;
    finpour
    pour j de 3 à n faire
        pour i de j−2 à 1 pas -1 faire
            m ← ∞;
            pour k de i+1 à j−1
                tmp ← w(P[i], P[k], P[j]) + A[k][j] + A[i][k];
                si tmp < m alors
                    m ← tmp;
                finsi
            finpour
            A[i][j] ← m;
        finpour
    finpour
    retourne A[1][n];
fin
```

```
TriangulationOptimale(P)
Entrée: tableau contenant les coordonnées des points de P.
Sortie: liste des cordes de la triangulation optimale.

    n ← longueur(P);
    A ← CreerTableau [2..n];
    tk ← CreerTableau [2..n];
    pour i de 2 à n faire
        A[i][i] ← 0;
    finpour
    pour j de 3 à n faire
        pour i de j−1 à 2 pas -1 faire
            m ← ∞;
            pour k de i à j−1 faire
                tmp ← w(P[i−1], P[k], P[j]) + A[k+1][j] + A[i][k];
                si {tmp < m}
                    m ← tmp;
                    tk[i][j] ← k;
                finsi
            finpour
            A[i][j] ← m;
        finpour
    finpour
```

```
Procedure Cordes(i,j)
    si |j − i| < 2 alors
        retourne ∅
    sinon
        k ← tk[i][j];
        Liste((P[i − 1], P[k]), (P[k], P[j]), Cordes(i, k), Cordes(k + 1, j));
    finsi
fin
retourne Cordes(2,n);
fin
```

4. Il y a $n(n-1)(n-2) \sim n^3$ triangles dans un polygone à n points. Il faut donc $\sim n^3$ valeurs pour définir la fonction de poids... alors que l'algorithme est en $O(n^2)$ en espace.

5. Toutes les triangulations ont la même pondération...

4 Problème NP-complet

Exercice 4.1. Problème euclidien du voyageur de commerce
Étant donnés n points du plan, le problème du voyageur de commerce consiste à trouver une *tournée* (i.e. un chemin reliant tous les points, et ne passant qu'une seule fois par chaque point) qui minimise la distance totale parcourue. Ce problème est un problème difficile en général (il est NP-complet).

1. J. L. Bentley (1962) à suggéré de se restreindre aux tournées *bitoniques* : ces tournées partent du point le plus à gauche, continuent strictement de gauche à droite vers le point le plus à droite, puis retournent vers le point de départ en se déplaçant strictement de droite à gauche (on suppose que deux points n'ont pas la même abscisse). Remarquons qu'une tournée bitonique optimale n'est pas nécessairement une tournée optimale.

Décrire un algorithme utilisant de la programmation dynamique qui détermine une tournée bitonique optimale (on balayera le plan de gauche à droite). On pourra utiliser la primitive DISTANCE(p_1, p_2 : **point**) : **réel** qui calcule la distance entre deux points. Quelle est sa complexité dans le pire cas en nombre d'appels à la fonction DISTANCE ?

2. On considère maintenant le problème général du voyageur de commerce. Quel est le nombre de tournées possibles ?

3. Structure du chemin optimal, solution récursive Soit $S \subset \{1, \ldots, n\}$ et $k \in S$. On note $C(S, k)$ la distance minimale pour aller de p_1 à p_k en passant une fois et une seule par tous les points de S. Exprimez $C(S, k)$ en fonction des $C(S - \{k\}, l)$, pour $l \in S - \{k\}$. Si C^* est le coût d'un chemin optimal, exprimez C^* en fonction des $C(S, k)$, pour $S \subset \{1, \ldots, n\}$ et $k \in S$.

4. Décrivez un algorithme de programmation dynamique utilisant cette relation de récurrence pour résoudre le problème du voyageur de commerce. Quel est le nombre d'étapes de calcul par cette méthode ? Comparez avec la question 3 Peut-on espérer faire mieux ?

5. Quelle est la place mémoire utilisée par l'algorithme précédent ? Peut-on faire mieux ?

Solution:

1.
```
Bitonic_TSP(T)
  Entrée: tableau contenant les coordonnées des points triés
          par abscisses croissantes.
  Sortie: tableau [1..n] de prédécesseurs (n a 2 prédécesseurs:
          celui du tableau et n-1)
      n ← longueur(T);
      A ← CreerTableau [1..n][1..n];
      B ← CreerTableau [1..n];
      P ← CreerTableau [1..n];

      B[2] ← 2 * Dist(T[1],T[2]);
      P[2] ← 1;

      pour i de 3 à n faire
          A[i][1] ← Σ_{k=1}^{i-1} Dist(T[k],T[k+1]) + Dist(T[1],T[i]);
      finpour

      pour j de 2 à n-1 faire
          m ← ∞;
          pour k de 1 à j+1-2 faire
              si A[j+1][k] < m alors
                  m ← A[j+1][k];
                  P[j+1] ← k;
              finsi
              finpour
          B[j+1] ← m;

          pour i de j+2 à n faire
              A[i][j] ← B[j+1] − Dist(T[j],T[j+1]) + Dist(T[j],T[i])+
                        Σ_{k=j+1}^{i-1} Dist(T[k],T[k+1]);
          finpour
      finpour

      x ← n;
      tantque x ≠ 2 faire
          pour i de x-1 à P[x]+1 pas -1 faire
              P[i] ← i-1;
          finpour
              x ← P[x]+1;
      fintantque
      retourne P;
  fin

  Bitonic_TSP(T)
  Entrée: tableau contenant les coordonnées des points triés
          par abscisses croissantes.
```

```
Sortie: tableau [1..n] de prédécesseurs (n a 2 prédécesseurs:
        celui du tableau et n-1)
    n ← longueur(T);
    A ← CreerTableau [1..n][1..n];
    B ← CreerTableau [1..n];
    D ← CreerTableau [1..n][1..n];
    P ← CreerTableau [1..n];

    B[2] ← 2 * Dist(T[1], T[2]);
    P[2] ← 1;

    pour i de 1 à n faire
        D[i][i] ← 0;
    finpour

    pour i de 3 à n faire
        D[1][i-1] ← D[1][i-2] + Dist(T[i-2], T[i-1])
        A[i][1] ← D[1][i-1] + Dist(T[1], T[i]);
    finpour

    pour j de 2 à n-1 faire
        m ← ∞;
        pour k de 1 à j+1-2 faire
            si A[j+1][k] < m alors
                m ← A[j+1][k];
                P[j+1] ← k;
            finsi
        finpour
        B[j+1] ← m;

        pour i de j+2 à n faire
            si j+1 ≠ i-1 alors
                D[j+1][i-1] ← D[j+1][i-2] + Dist(T[i-2], T[i-1])
            finsi
            A[i][j] ← B[j+1] - Dist(T[j], T[j+1]) + Dist(T[j], T[i]) + D[j+1][i-1]
        finpour
    finpour

    x ← n;
    tantque x ≠ 2 faire
        pour {i de x-1 à P[x]+1 pas -1 faire
            P[i] ← i-1;
        finpour
        x ← P[x]+1;
    fintantque
    retourne P;
fin
```

2.
3.
4.
5.

5 Bio-informatique

Exercice 5.1. Alignement de séquences génétiques
Un brin d'ADN (acide désoxyribonucléique) est caractérisé par la suite de molécules, appelées *bases*, qui la composent. Il existe quatre bases différentes : deux purines (l'adénine A, la guanine G), ainsi que deux pyrimidines (la thymine T et la cytosine C). Une séquence d'ADN peut donc être modélisée par un mot sur un alphabet à quatre lettres ($AGTC$).

Pouvoir comparer l'ADN de deux organismes est un problème fondamental en biologie. L'*alignement* entre deux séquences ADN similaires permet d'observer leur degré d'apparentée et d'estimer si elles semblent ou non homologues : c'est-à-dire si elles descendent ou non d'une même séquence ancestrale commune ayant divergé au cours de l'évolution. Un alignement met en évidence les :

- identités entre les 2 séquences,
- insertions ou délétions survenues dans l'une ou l'autre des séquences.

			identité				*insertion*					
X	G		A	T	C	G		G	C	A	T	
Y	C	A	A	T		G	T	G	A	A	T	C
pondération	-1	-2	+1	+1	-2	+1	-2	+1	-1	+1	+1	-2

FIGURE 3 – Alignement de deux séquences ADN.

Notons X et Y deux séquences ADN. Pour aligner ces deux séquences, on insère des espaces entre deux bases de chaque séquence, à des emplacements arbitraires (extrémités comprises), de sorte que les deux séquences résultantes (notées X' et Y') aient la même longueur (mais à un emplacement donné, il ne peut pas y avoir un espace pour chacune des deux séquences). Nous considérons le modèle d'évolution simplifié suivant : à chaque emplacement j on attribue une *pondération*

- $+1$ si $X'[j] = Y'[j]$ (alors aucun des deux n'est un espace),
- -1 si $X'[j] \neq Y'[j]$ et aucun des deux n'est un espace,
- -2 si $X'[j]$ ou $Y'[j]$ est un espace.

Le poids de l'alignement est la somme des poids des emplacements.

Donnez un algorithme de programmation dynamique déterminant l'alignement optimal (i.e. de poids maximal) de deux séquences ADN. Déterminez sa complexité dans le pire cas.

Notation : $pa(i,j)$ pondération optimale d'un alignement de $X[1..i]$ et $Y[1..j]$.

$$pa(i,j) = \begin{cases} \max(pa(i\text{-}1,j\text{-}1)+1, pa(i\text{-}1,j)\text{-}2, pa(i,j\text{-}1)\text{-}2) \text{ si } X[i]=Y[j] \\ \max(pa(i\text{-}1,j\text{-}1)\text{-}1, pa(i\text{-}1,j)\text{-}2, pa(i,j\text{-}1)\text{-}2) \text{ sinon} \end{cases} \quad (1)$$

$$pa(i,0) = pa(0,j) = 0 \quad (2)$$

$$P_{opt} = pa(n_X, n_Y) \quad (3)$$

```
Align_opt}(X,Y)
Entrée: tableaux contenant les séquences.
Sortie: liste de couples représentant l'alignement
    A ← CreerTableau [0..n_X][0..n_Y];
    P ← CreerTableau [0..n_X][0..n_Y];

    pour i de 0 à n_X faire
        A[i][0] ← −2 ∗ i;
        P[i][0] ← (i-1, 0);
    finpour
    pour j de 1 à n_Y faire
        A[0][j] ← −2 ∗ j;
        P[0][j] ← (0, j-1);
    finpour

    pour j de 1 à n_Y faire
        pour {i de 1 à n_X faire
            si X[i] = Y[j] alors
                d = A[i-1][j-1] + 1
            sinon
                d = A[i-1][j-1] − 1
            finsi
            A[i][j] ← max(d, A[i-1][j] − 2, A[i][j-1] − 2)
            si A[i][j] = d alors
                P[i][j] ← (i-1, j-1);
            finsi
            si A[i][j] = A[i-1][j] − 2 alors
                P[i][j] ← (i-1, j);
            finsi
            si A[i][j] = A[i][j-1] − 2 alors
                P[i][j] ← (i, j-1);
            finsi
        finpour
    finpour

    i = n_X; j = n_Y: L ← (X[i], Y[j])
    tantque i ≠ 0 and j ≠ 0
        (i, j) ← P[i][j];
        L ← (X[i], Y[j]).L
    fintantque
    retourne L;
fin
```