Annales d'algorithmique avancée

Jean-Michel Dischler et Frédéric Vivien

Table des matières

1 TD 1 : recherche par rang

Recherche du maximum

1. Concevez un algorithme de recherche du maximum dans un ensemble à n éléments (vous disposez en tout et pour tout d'une fonction de comparaison).

   MAXIMUM(A)
   max $\leftarrow A[1]$
   pour $i \leftarrow 2$ **à** n **faire**
   si max ¡ $A[i]$ **alors** max $\leftarrow A[i]$
   renvoyer max

2. Quelle est la complexité de votre algorithme en nombre de comparaisons ?
   Réponse : $n - 1$.

3. Montrez qu'il est optimal.

   Tout élément hormis le maximum doit avoir perdu une comparaison, sinon, on ne peut pas savoir qu'il n'est pas le maximum. Il y a $n-1$ tels éléments. Tout algorithme de recherche du maximum doit donc faire au moins $n-1$ comparaisons.

Recherche du deuxième plus grand élément

Nous supposerons ici que l'ensemble considéré ne contient pas deux fois la même valeur.

1. Proposez un algorithme simple de recherche du deuxième plus grand élément.

   DEUXIÈME-PLUS-GRAND(A)

   rang_max $\leftarrow 1$
   pour $i \leftarrow 2$ **à** n **faire si** A[rang_max] ¡ $A[i]$ **alors** rang_max $\leftarrow$ i
   si rang_max $\neq 1$ **alors** rang_second $\leftarrow 1$
   sinon rang_second $\leftarrow 2$
   pour $i \leftarrow 2$ **à** n **faire si** $i \neq$ rang_max **et** A[rang_second] ¡ $A[i]$ **alors** rang_second $\leftarrow$ i
   renvoyer A[rang_second]

2. Quel est sa complexité en nombre de comparaisons ?

   La recherche du maximum coûte $n-1$ comparaisons. La boucle qui recherche le deuxième plus grand élément une fois que le maximum a été trouvé effectue $n-2$ comparaisons. D'où un coût total de $2n-3$ comparaisons.

3. Récrivez votre algorithme de recherche du maximum sous la forme d'un tournoi (de tennis, de foot, de pétanque ou de tout autre sport). Il n'est pas nécessaire de formaliser l'algorithme ici, une figure explicative sera amplement suffisante.

   Les comparaisons sont organisées comme dans un tournoi :
   - *Dans une première phase, les valeurs sont comparées par paires. Dans chaque paire, il y a bien sûr un plus grand élément (le « vainqueur ») et un plus petit élément (le « vaincu »).*
   - *Dans la deuxième phase, les valeurs qui étaient plus grand élément de leur paire à la phase précédente sont comparées entres elles deux à deux.*
   - *On répète ce processus jusqu'au moment où il n'y a plus qu'**un** plus grand élément.*

   Ce procédé est illustré par la figure 1.

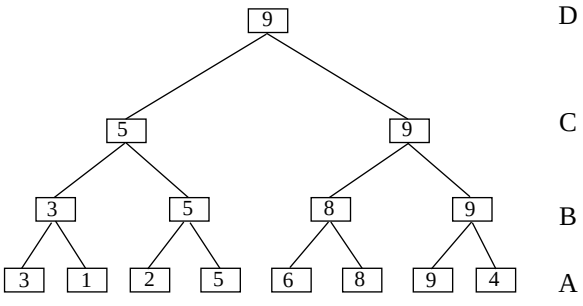

FIG. 1 – Méthode du tournoi pour la détermination du maximum : A : les éléments sont comparés par paires ; B : les plus grands éléments de la phase A sont comparés entre eux, par paires ; C : les éléments « vainqueurs » à la phase B sont comparés entre eux ; D : il ne reste plus qu'un élément, c'est l'élément maximal.

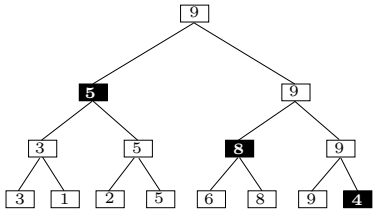

FIG. 2 – Le deuxième plus grand élément a nécessairement été battu par le plus grand élément (et que par lui). Il figure donc parmi les éléments comparés à l'élément maximal. Ces éléments apparaissent ici sur fond noir.

4. Dans combien de comparaisons, le deuxième plus grand élément de l'ensemble a-t-il été trouvé être le plus petit des deux éléments comparés ?

   Le deuxième plus grand n'est plus petit que devant le plus grand élément. Il n'a donc « perdu » que dans une comparaison, celle avec le plus grand élément.

5. Proposez un nouvel algorithme de recherche du deuxième plus grand élément.

   Le deuxième plus grand élément est donc un des éléments qui ont été battus par le plus grand élément. L'algorithme a lieu en deux phases :

   (a) *On recherche tout d'abord le plus grand élément suivant la méthode du tournoi.*

   (b) *On obtient le deuxième plus grand élément en recherchant l'élément maximal parmi ceux qui ont été éliminés du tournoi lors d'une comparaison avec l'élément maximal.*

   Voir la figure 2.

6. Quelle est sa complexité en nombre de comparaisons ?

   La recherche de l'élément maximal coûte $n-1$ comparaisons, comme d'habitude. Ensuite la recherche du deuxième plus grand élément nous coûte $m-1$ comparaisons, où m est le nombre d'éléments à qui l'élément maximal a été comparé. Dans le pire cas [1]*, m est égal à la hauteur de l'arbre moins 1 (un arbre réduit à sa racine étant de hauteur un). Or un arbre binaire presque parfait à n feuilles est de hauteur $\lceil \log_2 n \rceil$. D'où la complexité :*

$$T(n) = n + \lceil \log_2 n \rceil - 2$$

   Note : cet algorithme est optimal.

Recherche du maximum et du minimum

Nous supposerons ici que l'ensemble considéré ne contient pas deux fois la même valeur.

1. Proposez un algorithme naïf de recherche du maximum et du minimum d'un ensemble de n éléments.

```
MAXIMUM-ET-MINIMUM(A)
  max ← A[1]
  pour i ← 2 à n faire
    si max ¡ A[i] alors max ← A[i]
  min ← A[1]
  pour i ← 2 à n faire
    si min ¿ A[i] alors min ← A[i]
  renvoyer max et min
```

2. Quelle est sa complexité en nombre de comparaisons ?

   Cet algorithme effectue $2n-2$ comparaisons.

3. Proposez un algorithme plus efficace.

   Indication : dans une première phase les éléments sont comparés par paire.

   L'algorithme se décompose en trois phases :

   (a) *On compare par paire les éléments de l'ensemble. On met d'un côté les plus grands éléments (ici dans les cases paires du tableau) — c'est-à-dire les éléments qui sont sortis « vainqueurs » de leur comparaison — et de l'autre les plus petits (ici dans les cases impaires).*

   (b) *On recherche le minimum parmi tous les plus petits éléments (si on a un nombre impair d'éléments, il ne faut pas oublier le n^e élément qui n'a été comparé avec personne dans la première phase).*

   (c) *On recherche le maximum parmi tous les plus grands éléments.*

[1] Quand n n'est pas une puissance de deux, la complexité peut varier d'une comparaison suivant la place initiale dans l'arbre du maximum.

MAXIMUM-ET-MINIMUM(A)
Pour $i \leftarrow 1$ **à** $n-1$ **faire par pas de** 2
si $A[i] > A[i+1]$ **alors** échanger $A[i]$ et $A[i+1]$
min $\leftarrow A[1]$
Pour $i \leftarrow 3$ **à** n **faire par pas de** 2
si $A[i] <$ min **alors** min $\leftarrow A[i]$
max $\leftarrow A[2]$
Pour $i \leftarrow 4$ **à** n **faire par pas de** 2
si $A[i] >$ max **alors** max $\leftarrow A[i]$
si n est impair **alors si** $A[n] >$ max **alors** max $\leftarrow A[n]$
renvoyer max **et** min

4. Quelle est sa complexité en nombre de comparaisons ?
Regardons indépendamment le coût des trois phases :
(a) On peut former $\lfloor \frac{n}{2} \rfloor$ *paires, on effectue donc* $\lfloor \frac{n}{2} \rfloor$ *comparaisons.*
(b) Parmi n *éléments on a* $\lceil \frac{n}{2} \rceil$ *éléments de rangs impairs. Dans cette phase on effectue donc* $\lceil \frac{n}{2} \rceil - 1$ *comparaisons.*
(c) Ici aussi on effectue aussi $\lceil \frac{n}{2} \rceil - 1$ *comparaisons.*
D'où une complexité totale en :

$$T(n) = \left\lfloor \frac{n}{2} \right\rfloor + 2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) = n + \left\lceil \frac{n}{2} \right\rceil - 2$$

5. Montrez que cet algorithme est optimal.
Indication : on appelle *unité d'information* :
– l'information « l'élément x ne peut pas être le plus grand élément » ;
– l'information « l'élément x ne peut pas être le plus petit élément ».

(a) Quel est le nombre minimal d'unités d'information qu'un algorithme de recherche du maximum et du minimum doit produire pour nous garantir la validité de son résultat ?
Pour être sûr qu'un élément est bien le maximum (respectivement le minimum) il faut que l'on sache que les $n-1$ *autres ne peuvent pas être le maximum (respectivement le minimum) ce qui représente* $n-1$ *unités d'informations. L'algorithme doit donc produire au moins* $2n-2$ *unités d'information.*

(b) Combien d'unités d'information sont produites par la comparaison de deux éléments (distinguez des cas, suivant que l'on a ou non des unités d'informations sur ces valeurs).

i. Si on n'a d'unités d'information pour aucun des deux éléments, la comparaison nous fait gagner deux unités d'information : le plus petit des deux ne peut pas être le maximum, ni le plus grand le minimum.

ii. Si on a la même unité d'information pour les deux éléments (par exemple, aucun des deux ne peut être le plus grand), la comparaison nous procure une unité d'information (dans notre exemple, le plus grand des deux ne peut pas être le plus petit).

iii. Si on a une unité d'information pour chacun des deux éléments, mais des unités de type différent : si celui qui peut être le minimum est plus grand que celui qui peut être le maximum, on gagne deux unités d'information, et sinon zéro.

iv. Si on a une unité d'information pour un des éléments (par exemple, ne peut pas être le plus petit) et zéro pour l'autre, la comparaison peut nous donner une unité d'information (celui sans information est plus petit que l'autre dans notre exemple et il ne peut pas être le maximum) ou deux (celui sans information est plus grand, ne peut donc plus être le minimum, et l'autre ne peut plus être le maximum).

v. *Si on a deux unités d'information pour un des éléments et zéro pour l'autre, la comparaison nous donne une unité d'information (par exemple, si l'élément sans information est plus grand, il ne peut plus être le minimum).*

(c) Concluez.

Il nous faut donc $2n-2$ unités d'information pour pouvoir conclure. Les comparaisons de type 5(b)i nous donnent toujours deux unités d'information chacune, or on peut au plus effectuer $\lfloor \frac{n}{2} \rfloor$ comparaisons de ce type (autant que l'on peut former de paires). Les autres comparaisons nous donnent, dans le pire des cas, une seule unité d'information chacune. Donc il nous faudra au moins effectuer $2n-2-2\left(\lfloor \frac{n}{2} \rfloor\right) = 2\lceil \frac{n}{2} \rceil - 2$ telles comparaisons. Dans le pire des cas, il nous faudra donc effectuer au moins :

$$\left\lfloor \frac{n}{2} \right\rfloor + 2\left\lceil \frac{n}{2} \right\rceil - 2 = n + \left\lceil \frac{n}{2} \right\rceil - 2$$

comparaisons, d'où l'optimalité de notre algorithme !

2 TD 2 : récursivité

Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$\mathrm{Fib}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{sinon.} \end{cases}$$

1. Écrivez un algorithme récursif calculant $\mathrm{Fib}(n)$.

   FIBONACCI(n)
   si $n = 0$ ou $n = 1$ **alors renvoyer** 1
   sinon renvoyer FIBONACCI($n-1$) + FIBONACCI($n-2$)

2. Montrez que la complexité (en nombre d'additions) de cet algorithme est en $\Omega(2^{\frac{n}{2}})$.

   On procède par récurrence. On veut montrer qu'il existe une constante c strictement positive telle que $T(n) \geq c.2^{\frac{n}{2}}$, pour des valeurs de n supérieures à une certaine borne n_0 (à déterminer). Supposons le résultat démontré jusqu'au rang $n-1$. Alors :

   $$T(n) = T(n-1) + T(n-2) + 1 \geq c.2^{\frac{n-1}{2}} + c2^{\frac{n-2}{2}} + 1 \geq c.2^{\frac{n-2}{2}} + c.2^{\frac{n-2}{2}} + 1 \geq 2 \times c.2^{\frac{n-2}{2}} = c.2^{\frac{n}{2}}$$

   Il nous reste juste à montrer que cette équation est vraie « au départ ». Nous ne pouvons bien évidemment pas partir des cas $n = 0$ et $n = 1$, puisque pour ces valeurs $T(n) = 0$. Nous partons donc des cas $n = 2$ et $n = 3$ (la récurrence nécessite deux *valeurs de départ) :*
   - *Cas $n = 2$:* FIBONACCI*(2)* = FIBONACCI*(1)* + FIBONACCI*(0)*, *et $T(2) = 1$. Pour que la propriété désirée soit vraie, c doit donc vérifier :*

     $$1 \geq c.2^{\frac{2}{2}} = 2c \quad \Leftrightarrow \quad c \leq \frac{1}{2}$$

   - *Cas $n = 3$:* FIBONACCI*(3)* = FIBONACCI*(2)* + FIBONACCI*(1)*, *et $T(3) = 2$. Pour que la propriété désirée soit vraie, c doit donc vérifier :*

     $$2 \geq c.2^{\frac{3}{2}} = 2\sqrt{2}c \quad \Leftrightarrow c \leq \frac{\sqrt{2}}{2}$$

   Donc si $c = \frac{1}{2}$, pour $n \geq 2$, on a $T(n) \geq c2^{\frac{n}{2}}$ et donc $T(n) = \Omega(2^{\frac{n}{2}})$.

3. Écrire un algorithme récursif qui calcule, pour $n > 0$, le couple (FIBONACCI(n), FIBONACCI($n-1$)).

FIB-PAIRE(n)
  si $n = 1$ **alors renvoyer** $(1, 1)$
    sinon $(x, y) =$ FIB-PAIRE$(n-1)$
      renvoyer $(x + y, x)$

4. Utilisez l'algorithme précédent pour écrire un nouvel algorithme calculant FIBONACCI(n).

   FIBONACCI(n)
     si $n = 0$ **alors renvoyer** 1
       sinon $(x, y) =$ FIB-PAIRE(n)
         renvoyer x

5. Qu'elle est la complexité (en nombre d'additions) de cet algorithme ?
   La complexité de l'algorithme FIB-PAIRE*, en nombre d'additions, est donnée par la récurrence* $T(n) = 1 + T(n-1)$*. On a donc* $T(n) = n - 1$ *pour* FIB-PAIRE*, et par extension pour la nouvelle version de* FIBONACCI*.*

Opérations ensemblistes

Dans cette partie on considère des ensembles représentés par des tableaux, certains ensembles seront triés et d'autres pas. **Toutes les solutions proposées doivent être récursives.**

1. Nous voulons un algorithme APPARTENANCE(A, x) qui recherche si un élément x appartient à l'ensemble A. Si x appartient effectivement à A, l'algorithme renverra VRAI, et FAUX sinon.
   (a) Cas des ensembles non triés :
      i. Écrivez un tel algorithme.

         RECHERCHE(A, *rang*, x)
           si *rang* $>$ *longueur*(A) **alors renvoyer** FAUX
           si A[*rang*] $= x$ **alors renvoyer** VRAI
                sinon renvoyer RECHERCHE(A, *rang* $+ 1$, x)

         L'appel initial de l'algorithme est alors RECHERCHE(A, 1, x).

      ii. Quelle est sa complexité en nombre de comparaisons ?
         Dans le pire cas, l'élément n'appartient pas à l'ensemble et tout le tableau est parcouru. La complexité au pire est donc en $\Theta(n)$*, où* n *est la longueur du tableau (et donc la taille de l'ensemble).*

   (b) Cas des ensembles triés (dans l'ordre croissant) :
      i. Écrivez un tel algorithme.

         RECHERCHE(A, *rang*, x)
           si *rang* $>$ *longueur*(A) ou A[*rang*] $> x$
             alors renvoyer FAUX
             sinon si A[*rang*] $= x$
               alors renvoyer VRAI
               sinon renvoyer RECHERCHE(A, *rang* $+ 1$, x)

         L'appel initial de l'algorithme est alors RECHERCHE(A, 1, x).

      ii. Quelle est sa complexité en nombre de comparaisons ?
         Le pire cas est aussi en $\Theta(n)$ *: il intervient quand l'élément recherché n'appartient pas à l'ensemble mais est plus grand que tous les éléments de l'ensemble.*

      iii. Utilisez une recherche dichotomique pour améliorer votre algorithme.

RECHERCHE(A, x, *inf*, *sup*)
  $milieu \leftarrow \left\lfloor \frac{inf+sup}{2} \right\rfloor$
  si $A[milieu] = x$
    alors renvoyer VRAI
    sinon si $A[milieu] > x$ **alors renvoyer** RECHERCHE(A, x, *inf*, *milieu* - 1)
                sinon renvoyer RECHERCHE(A, x, *milieu* + 1, *sup*)

iv. Quelle est la complexité de votre nouvel algorithme ?

Posons $n = sup - inf + 1$ le nombre d'éléments dans la partie du tableau à étudier. Considérons la taille du tableau lors de l'éventuel appel récursif. Nous avons deux cas à considérer :

- *L'appel effectué est :* RECHERCHE*(A, x, inf, milieu - 1). Le nombre d'éléments concernés est alors : $milieu - 1 - inf + 1 = \left\lfloor \frac{sup-inf}{2} \right\rfloor = \left\lfloor \frac{n-1}{2} \right\rfloor \leq \frac{n}{2}$.*
- *L'appel effectué est :* RECHERCHE*(A, x, milieu + 1, sup). Le nombre d'éléments concernés est alors : $sup - (milieu + 1) + 1 = \left\lceil \frac{sup-inf}{2} \right\rceil = \left\lceil \frac{n-1}{2} \right\rceil \leq \frac{n}{2}$.*

On passe donc d'un ensemble de taille n à un ensemble de taille au plus $\frac{n}{2}$. D'où $T(n) \leq 2 \times T(\frac{n}{2})$ (la fonction $T(n)$ étant croissante, on peut se permettre l'approximation). Par conséquent : $T(n) \leq 2 \times \log_2(n) T(\frac{n}{2^{\log_2 n}})$ et $T(n) = O(\log_2 n)$.

2. Nous voulons maintenant un algorithme UNION(A, B) qui nous renvoie l'union des deux ensembles qui lui sont passés en argument.

(a) Cas des ensembles non triés :

i. Écrivez un tel algorithme.

UNION(A, B, *rang*, C)
  si RECHERCHE(A, $B[rang]$) = FAUX
    alors $longueur(C) \leftarrow longueur(C) + 1$
      $C[longueur(C)] \leftarrow B[rang]$
  UNION(A, B, $rang + 1$, C)

L'appel initial est alors UNION(A, B, 1, C) où C est un tableau de taille $longueur(A) + longueur(B)$, et dont les $longueur(A)$ premières cases contiennent les éléments de A.

ii. Quelle est sa complexité ?

La recopie de A dans C est de coût longueur(A).

L'algorithme UNION *est appelé $longueur(B)$ fois, chacun de ces appels effectuant un appel à* RECHERCHE *sur A, dont le coût au pire est en longueur(A). La complexité au pire de* UNION *est donc en $\Theta(longueur(A) \times longueur(B))$ ou $\Theta(nm)$, n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

(b) Cas des ensembles triés (dans l'ordre croissant) :

i. Écrivez un tel algorithme.

UNION(A, a, B, b, C)
  si $a > longueur(A)$ et $b > longueur(B)$ **alors renvoyer** C
  si $b > longueur(B)$ ou $B[b] > A[a]$
    alors $longueur(C) \leftarrow longueur(C) + 1$
      $C[longueur(C)] \leftarrow A[a]$
      UNION(A, $a + 1$, B, b, C)
    sinon $longueur(C) \leftarrow longueur(C) + 1$
      $C[longueur(C)] \leftarrow B[b]$
      si $a \leq longueur(A)$ et $A[a] = B[b]$ **alors** UNION(A, $a + 1$, B, $b + 1$, C)
                sinon UNION(A, a, B, $b + 1$, C)

L'appel initial est UNION(A, 1, B, 1, C).

ii. Quelle est sa complexité ?

La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ *ou* $\Theta(n+m)$ *: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.*

3. Nous voulons maintenant un algorithme INTERSECTION(A, B) qui nous renvoie l'intersection des deux ensembles qui lui sont passés en argument.

   (a) Cas des ensembles non triés :

   i. Écrivez un tel algorithme.

   INTERSECTION(A, B, *rang*, C)
   si RECHERCHE(A, B[*rang*]) = VRAI
   alors $longueur(C) \leftarrow longueur(C) + 1$
   $C[longueur(C)] \leftarrow B[rang]$
   INTERSECTION(A, B, $rang + 1$, C)

   L'appel initial est alors INTERSECTION*(A, B, 1, C) où C est un nouveau tableau, de taille* $\min(longueur(A), longueur(B))$*, et ne contenant initialement aucun élément (*$longueur(C) = 0$*).*

   ii. Quelle est sa complexité ?

   L'algorithme INTERSECTION *est appelé* $longueur(B)$ *fois, chacun de ces appels effectuant un appel à* RECHERCHE *sur A, dont le coût au pire est en longueur(A). La complexité au pire de* INTERSECTION *est donc en* $\Theta(longueur(A) \times longueur(B))$ *ou* $\Theta(nm)$*, n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

   (b) Cas des ensembles triés (dans l'ordre croissant) :

   i. Écrivez un tel algorithme.

   INTERSECTION(A, a, B, b, C)
   si $a > longueur(A)$ ou $b > longueur(B)$ **alors renvoyer** C
   si $A[a] = B[b]$ **alors** $longueur(C) \leftarrow longueur(C) + 1$
   $C[longueur(C)] \leftarrow A[a]$
   renvoyer INTERSECTION(A, $a+1$, B, $b+1$, C)
   sinon si $B[b] > A[a]$ **alors renvoyer** INTERSECTION(A, $a+1$, B, b, C)
   sinon renvoyer INTERSECTION(A, a, B, $b+1$, C)

   L'appel initial est INTERSECTION(A, 1, B, 1, C, 0).

   ii. Quelle est sa complexité ?

   La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ *ou* $\Theta(n+m)$ *: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.*

4. Nous voulons maintenant un algorithme DIFFÉRENCE(A, B) qui nous renvoie la différence des deux ensembles qui lui sont passés en argument (La différence de A et de B, notée $A \setminus B$ est l'ensemble des éléments de A n'appartenant pas à B).

   (a) Cas des ensembles non triés :

   i. Écrivez un tel algorithme.

   DIFFÉRENCE(A, *rang*, B, C)
   si RECHERCHE(B, A[*rang*]) = FAUX
   alors $longueur(C) \leftarrow longueur(C) + 1$
   $C[longueur(C)] \leftarrow A[rang]$
   DIFFÉRENCE(A, $rang + 1$, B, C)

   L'appel initial est alors DIFFÉRENCE(A, 1, B, C) où C est un tableau de taille $longueur(A)$, ne contenant initialement aucun élément ($longueur(C) = 0$).

ii. Quelle est sa complexité ?
L'algorithme Différence *est appelé longueur(A) fois, chacun de ces appels effectuant un appel à* Recherche *sur B, dont le coût au pire est en longueur(B). La complexité au pire de* Différence *est donc en* $\Theta(longueur(A) \times longueur(B))$ *ou* $\Theta(nm)$, *n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

(b) Cas des ensembles triés (dans l'ordre croissant) :

i. Écrivez un tel algorithme.

Différence(A, a, B, b, C)
si $a > longueur(A)$ **alors renvoyer** C
si $A[a] = B[b]$ **alors renvoyer** Différence(A, $a+1$, B, $b+1$, C)
sinon si $A[a] < B[b]$ **alors** $longueur(C) \leftarrow longueur(C) + 1$
$C[longueur(C)] \leftarrow A[a]$
renvoyer Différence(A, $a+1$, B, b, C)
sinon renvoyer Différence(A, a, B, $b+1$, C)

L'appel initial est Différence(A, 1, B, 1, C, 0).

ii. Quelle est sa complexité ?
La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ *ou* $\Theta(n+m)$ *: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.*

3 TD 3 : multiplications « diviser pour régner »

Multiplications « diviser pour régner »

1. Montrez comment multiplier deux polynômes linéaires $ax+b$ et $cx+d$ à l'aide de trois multiplications seulement. (*Indication* : l'une des multiplications est $(a+b)(c+d)$.)
D'une part : $(ax+b)\times(cx+d) = acx^2+(ad+bc)x+bd$. *D'autre part :* $(a+b)(c+d) = ac+ad+bc+bd = ac+bd+ad+bc$. *D'où :* $(ax+b)\times(cx+d) = acx^2+((a+b)(c+d)-ac-bd)x+bd$, *et les trois seules multiplications nécessaires sont les calculs : ac, bd et* $(a+b)(c+d)$.
2. Donnez deux algorithmes « diviser pour régner » permettant de multiplier deux polynômes de degré au plus n et s'exécutant en $\Theta(n^{\log_2 3})$.
   (a) Le premier algorithme devra couper les coefficients du polynôme d'entrée en deux moitiés, l'une supérieure et l'autre inférieure.
   Soient $P[X]$ *et* $Q[X]$ *les deux polynômes d'entrée.* $P[X] = \sum_{i=0}^{n} p_i X^i$ *et* $Q[X] = \sum_{i=0}^{n} q_i X^i$.

$$\begin{aligned} P[X] &= \sum_{i=0}^{n} p_i X^i \\ &= \left(\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_i X^i\right) + \left(\sum_{i=1+\lfloor \frac{n}{2} \rfloor}^{n} p_i X^i\right) \\ &= \left(\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_i X^i\right) + \left(X^{1+\lfloor \frac{n}{2} \rfloor} \sum_{i=0}^{n-1-\lfloor \frac{n}{2} \rfloor} p_{i+1+\lfloor \frac{n}{2} \rfloor} X^i\right). \end{aligned}$$

On pose alors $B[X] = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_i X^i$ *et* $A[X] = \sum_{i=0}^{n-1-\lfloor \frac{n}{2} \rfloor} p_{i+1+\lfloor \frac{n}{2} \rfloor} X^i$. $A[X]$ *et* $B[X]$ *sont alors deux polynômes de degré au plus* $\lfloor \frac{n}{2} \rfloor$ *et* $P[X] = A[X]X^{1+\lfloor \frac{n}{2} \rfloor} + B[X]$. *On définit de même les polynômes C et D pour Q :* $Q[X] = C[X]X^{1+\lfloor \frac{n}{2} \rfloor} + D[X]$.
Avec les notations précédemment définies, on a :

$$\begin{aligned} P[X]Q[X] = \; & A[X]C[X]X^{2+2\lfloor \frac{n}{2} \rfloor} \\ & +((A[X]+B[X])(C[X]+D[X]) - A[X]C[X] - B[X]D[X])X^{1+\lfloor \frac{n}{2} \rfloor} \\ & +C[X]D[X]. \end{aligned}$$

Par conséquent, le produit de deux polynômes de degré au plus n peut se ramener au calcul de trois produits de polynômes de degré au plus $\lfloor \frac{n}{2} \rfloor$ ($A[X]C[X]$, $B[X]D[X]$ et $(A[X]+B[X])(C[X]+D[X])$), à des additions de polynômes de degré au plus n —ce qui coûte $\Theta(n)$— et à des multiplications par un monôme X^j —ce qui est un simple décalage des indices et coûte également $\Theta(n)$. L'équation de récurrence définissant la complexité de notre algorithme est alors :

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n).$$

Nous appliquons alors le théorème vu en cours :

Théorème 1 (Résolution des récurrences « diviser pour régner »).
Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ une fonction définie pour les entiers positifs par la récurrence :

$$T(n) = aT(n/b) + f(n),$$

où l'on interprète n/b soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$.
$T(n)$ peut alors être bornée asymptotiquement comme suit :

i. Si $f(n) = O(n^{(\log_b a)-\epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.

ii. Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.

iii. Si $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ pour une certaine constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Ici $a = 3$, $b = 2$ et $f(n) = \Theta(n)$. Comme $\log_2 3 > 1$, nous nous trouvons dans le cas i) du théorème et donc

$$T(n) = \Theta\left(n^{\log_2 3}\right).$$

Pour fixer les idées, $\log_2 3 \approx 1,58$ et l'algorithme naïf de multiplications de polynômes est en $\Theta(n^2)$.

(b) Le second algorithme devra séparer les coefficients du polynôme d'entrée selon la parité de leur indice.

$$\begin{aligned} P[X] &= \sum_{i=0}^{n} p_i X^i \\ &= \sum_{i \ pair} p_i X^i + \sum_{i \ impair} p_i X^i \\ &= \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_{2i} X^{2i} + \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} p_{2i+1} X^{2i+1} \\ &= \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_{2i} X^{2i} + X \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} p_{2i+1} X^{2i} \end{aligned}$$

On pose alors $A[X] = \sum_{i=1}^{\lfloor \frac{n-1}{2} \rfloor} p_{2i+1} X^i$ et $B[X] = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_{2i} X^i$. $A[X]$ et $B[X]$ sont alors deux polynômes de degré au plus $\lfloor \frac{n}{2} \rfloor$ et $P[X] = A[X^2]X + B[X^2]$. On définit de même les polynômes C et D pour Q : $Q[X] = C[X^2]X + D[X^2]$.
Avec les notations précédemment définies on a :

$$\begin{aligned} P[X]Q[X] &= A[X^2]C[X^2]X^2 \\ &+((A[X^2] + B[X^2])(C[X^2] + D[X^2]) - A[X^2]C[X^2] - B[X^2]D[X^2])X \\ &+B[X^2]D[X^2] \end{aligned}$$

Par conséquent, le produit de deux polynômes de degré au plus n peut se ramener au calcul de trois produits de polynômes de degré au plus $\lfloor \frac{n}{2} \rfloor$ ($A[X]C[X]$, $B[X]D[X]$ et $(A[X]+B[X])(C[X]+D[X])$), à des additions de polynômes de degré au plus n —ce qui coûte $\Theta(n)$— à des multiplications par un monôme X^j —ce qui est un simple décalage des indices et coûte également $\Theta(n)$— et à des transpositions du polynôme $R[X]$ au polynôme $R[X^2]$ —ce qui est encore un simple décalage des indices et coûte également $\Theta(n)$. L'équation de récurrence définissant la complexité de notre algorithme est donc comme précédemment :

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n),$$

et la complexité est la même :

$$T(n) = \Theta\left(n^{\log_2 3}\right).$$

3. Montrez que deux entiers à n bits peuvent être multipliés en $\Theta(n^{\log_2 3})$ étapes.
   L'entier $m = \sum_{i=0} m_i 2^i$ peut être vu comme un polynôme : il nous suffit de réappliquer un des algorithmes vu à la question précédente pour obtenir le résultat escompté.

Calcul de (cos(nx), sin(nx))

Écrire un algorithme prenant en entrée un entier `n` et une paire de valeurs réelles qui sont en fait les valeurs du cosinus et du sinus d'un certain angle `x`, et renvoyant la paire `(cos(nx), sin(nx))`. Autrement dit, le deuxième argument de la fonction est une paire `(a,b)` telle que `a = cos x` et `b = sin x`. Le schéma de calcul doit être récursif (mais non « diviser pour régner »).

On pourra se servir des formules de trigonométrie suivantes :

```
cos(nx) = cos((n-1)x) cos(x) - sin((n-1)x) sin(x)
sin(nx) = sin((n-1)x) cos(x) + cos((n-1)x) sin(x)
```

TRIGO(n, a, b)
  Si $n = 0$ **alors renvoyer** $(1, 0)$
    sinon $c, d =$ TRIGO$(n-1, a, b)$
      renvoyer $(ac - bd, ad + bc)$

4 TD 4 : recherche de l'élément majoritaire

Nous nous intéressons à un tableau A de n éléments, n étant supposé être une puissance de deux. Nous supposons également que la seule opération à notre disposition nous permet de vérifier si deux éléments sont ou non égaux. Un élément x de A est dit majoritaire si et seulement si A contient strictement plus de $n/2$ occurrences de x. Nous nous intéresserons à la complexité au pire.

Algorithme naïf

1. Écrivez un algorithme qui calcule le nombre d'occurrences d'une valeur x présentes entre les indices i et j d'un tableau A.

   OCCURRENCES(x, A, i, j)
     compteur $\leftarrow 0$
     pour $k \leftarrow i$ **à** j **faire**
       si $A[k] = x$ **alors** *compteur* $\leftarrow$ *compteur* $+ 1$
     renvoyer *compteur*

2. Quelle est la complexité de cet algorithme ?
   La boucle exécute $j - i + 1$ itérations. La complexité de cet algorithme est donc en $\Theta(j - i)$.

3. Au moyen de l'algorithme précédent, écrivez un algorithme MAJORITAIRE qui vérifie si un tableau A contient un élément majoritaire.

```
MAJORITAIRE(A)
  pour i ← 1 à longueur(A)/2 faire
    si OCCURRENCES(A[i], A, i, longueur(A)) > longueur(A)/2 alors renvoyer VRAI
  renvoyer FAUX
```

4. Quelle est la complexité de cet algorithme ?

   Dans le pire cas, la boucle effectue $n/2$ itérations, chacune de ces itérations effectuant un appel à OCCURRENCES *sur un tableau de taille $n-i$ (i variant de 1 à n) donc de coût $\Theta(n-i)$. Le coût total de l'algorithme est donc en $\Theta(n^2)$.*

Premier algorithme « diviser pour régner »

1. Proposez un algorithme MAJORITAIRE construit suivant le paradigme « diviser pour régner ». Cet algorithme divisera en deux le tableau A sur lequel il travaille. Il renverra le couple (VRAI, x) si le tableau A contient un élément majoritaire (x étant cet élément) et renverra le couple (FAUX, 0) si le tableau A ne contient pas d'élément majoritaire.

```
MAJORITAIRE(A, i, j)
  si i = j alors renvoyer (VRAI, A[i])
  (r_x, x) ← MAJORITAIRE(A, i, (i+j-1)/2)
  (r_y, y) ← MAJORITAIRE(A, (i+j+1)/2, j)
  si r_x = FAUX et r_y = FAUX alors renvoyer (FAUX, 0)
  si r_x = VRAI et r_y = VRAI
    alors si x = y
      alors renvoyer (VRAI, x)
      sinon
        c_x ← OCCURRENCES(x, A, i, j)
        c_y ← OCCURRENCES(y, A, i, j)
        si c_x > (j-i+1)/2
          alors renvoyer (VRAI, x)
          sinon si c_y > (j-i+1)/2
            alors renvoyer (VRAI, y)
            sinon renvoyer (FAUX, 0)
    sinon si r_x = VRAI
      alors si OCCURRENCES(x, A, i, j) > (j-i+1)/2
        alors renvoyer (VRAI, x)
        sinon renvoyer (FAUX, 0)
      sinon si OCCURRENCES(y, A, i, j) > (j-i+1)/2
        alors renvoyer (VRAI, y)
        sinon renvoyer (FAUX, 0)
```

Justifications

Les deux seuls cas qui ne sont peut-être pas immédiats sont les suivants :

(a) $r_x =$ FAUX et $r_y =$ FAUX : dans ce cas il n'y a pas d'élément qui soit majoritaire dans la première moitié du tableau, ni d'élément qui soit majoritaire dans la deuxième moitié du tableau. Si le table contient n éléments, le nombre d'occurrences d'un élément quelconque dans la première moitié du tableau est donc inférieur ou égal à $\frac{\frac{n}{2}}{2}$ —la première moitié ayant $\frac{n}{2}$ éléments— et il en va de même pour le deuxième moitié. Donc le nombre d'occurences d'un élément quelconque dans le tableau est inférieur à $\frac{n}{2}$ et le tableau ne contient pas d'élément majoritaire.

(b) $r_x = \text{VRAI}$ *et* $r_y = \text{VRAI}$ *avec* $x = y$ *: dans ce cas* x *est présent au moins* $1 + \frac{n}{4}$ *fois dans chacune des deux parties —qui sont de taille* $\frac{n}{2}$ *— et donc au moins* $2 + \frac{n}{2}$ *fois dans le tableau.*

2. Quelle est la complexité de cet algorithme ?

   La complexité de cet algorithme est définie par la relation de récurrence :

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

   En effet, la phase de combinaison nécessite, dans le pire des cas, la recherche du nombre d'occurences de deux éléments dans le tableau, ce qui a un coût de n*, toutes les autres opérations étant de coût constant (*$\Theta(1)$*).*

   Nous avons donc ici : $a = 2$*,* $b = 2$ *et* $f(n) = \Theta(n) == \Theta(n^{\log_2 2})$*. Nous sommes donc dans le cas 2 du théorème et donc :*

$$T(n) = \Theta(n \log n).$$

Deuxième algorithme « diviser pour régner »

1. Écrivez un algorithme construit suivant le paradigme « diviser pour régner », prenant en entrée un tableau A —qu'il divisera en deux— et possédant la propriété suivante :
   - soit cet algorithme nous garantit que le tableau A ne contient pas d'élément majoritaire ;
   - soit cet algorithme nous renvoie un élément x et un entier $c_x > n/2$ tels que x apparaisse *au plus* c_x fois dans A et que tout autre élément de A apparaisse *au plus* $n - c_x$ fois dans A.

```
PseudoMajoritaire(A, i, j)
  si i = j alors renvoyer (Vrai, A[i], 1)
  (r_x, x, c_x) ← Majoritaire(A, i, (i+j-1)/2)
  (r_y, y, c_y) ← Majoritaire(A, (i+j+1)/2, j)
  si r_x = Faux et r_y = Faux alors renvoyer (Faux, 0, 0)
  si r_x = Vrai et r_y = Faux alors renvoyer (Vrai, x, c_x + (j-i+1)/4)
  si r_x = Faux et r_y = Vrai alors renvoyer (Vrai, y, c_y + (j-i+1)/4)
  si r_x = Vrai et r_y = Vrai
    alors si x = y
      alors renvoyer (Vrai, x, c_x + c_y)
      sinon si c_x = c_y
        alors renvoyer (Faux, 0, 0)
        sinon si c_x > c_y
          alors renvoyer (Vrai, x, (j-i+1)/2 + c_x - c_y)
          sinon renvoyer (Vrai, y, (j-i+1)/2 + c_y - c_x)
```

Justifications

Nous considérons un par un les différents cas de figure :

- $r_x = \text{FAUX}$ *et* $r_y = \text{FAUX}$*. Aucun élément n'apparaît strictement plus de* $\frac{n}{4}$ *fois dans la première (resp. la deuxième) moitié du tableau A. Donc un élément quelconque de A apparaît au plus* $\frac{n}{4}$ *fois dans chacune des deux moitiés, et donc* $\frac{n}{2}$ *fois en tout dans A. Donc A ne contient pas d'élément majoritaire.*
- $r_x = \text{VRAI}$ *et* $r_y = \text{FAUX}$*. Un élément quelconque de A apparaît donc* au plus $\frac{n}{4}$ *fois dans la deuxième moitié de A. Nous avons deux cas à considérer :*
  - *x apparaît donc au plus* $c_x + \frac{n}{4}$ *fois dans A.*
  - *Un élément autre que x apparaît* au plus $\frac{n}{2} - c_x$ *fois dans la première moitié de A. Par conséquent un tel élément apparaît au plus* $\left(\frac{n}{2} - c_x\right) + \frac{n}{4} = \frac{3n}{4} - c_x = n - \left(c_x + \frac{n}{4}\right)$ *fois dans A.*

  D'où le résultat.
- $r_y = \text{VRAI}$ *et* $r_x = \text{FAUX}$ *: ce cas est symétrique du précédent.*

– r_x = VRAI *et* r_y = VRAI *:*
  – $x = y$. *x est présent* au plus $c_x + c_y$ *fois dans A. De plus, tout autre élément est présent* au plus $\frac{n}{2} - c_x$ *fois dans la première moitié de A et* $\frac{n}{2} - c_y$ *fois dans la deuxième moitié, soit en tout* au plus $n - (c_x + c_y)$ *fois dans A.*
  – $x \neq y$ *et* $c_x = c_y$. *x est présent* au plus c_x *fois dans la première moitié et* $\frac{n}{2} - c_y = \frac{n}{2} - c_x$ *fois dans la deuxième moitié, soit* $\frac{n}{2}$ *fois en tout et x n'est pas un élément majoritaire de A. Symétriquement, il en va de même de y. Tout autre élément ne peut être un élément majoritaire (voir le tout premier cas).*
  – $x \neq y$ *et* $c_x > c_y$. *Alors x est présent* au plus c_x *fois dans la première moitié de A et* $\frac{n}{2} - c_y$ *fois dans la deuxième moitié, soit* au plus $\frac{n}{2} + c_x - c_y$ *fois dans A, et ce nombre est strictement supérieur à* $\frac{n}{2}$ *car* $c_x > c_y$. *y est présent au plus* $\frac{n}{2} + c_y - c_x = n - (\frac{n}{2} + c_x - c_y)$ *fois dans A. Tout autre élément est présent au plus* $\left(\frac{n}{2} - c_x\right) + \left(\frac{n}{2} - c_y\right) = n - c_x - c_y = \frac{n}{2} - c_x + \frac{n}{2} - c_y \leq \frac{n}{2} - c_x + c_y$ *(car* $c_y > \frac{n}{4}$*)* $= n - \left(\frac{n}{2} + c_x - c_y\right)$.

2. Quelle est la complexité de cet algorithme ?

   En dehors des appels récursifs, tous les traitements ont un coût constant : $\Theta(1)$. *La complexité de l'algorithme est donc donnée par la relation de récurrence :*

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1).$$

   Nous nous trouvons donc ici dans le cas 1) du théorème (avec $\epsilon = 1$*) et la complexité de l'algorithme est donc :*

$$T(n) = \Theta(n).$$

3. À partir de l'algorithme précédent, écrivez un algorithme MAJORITAIRE qui vérifie si un tableau A contient un élément majoritaire.

```
MAJORITAIRE(A)
  (réponse, x, c_x) ← PSEUDOMAJORITAIRE(A, 1, longueur(A))
  si réponse = FAUX
    alors renvoyer FAUX
    sinon si OCCURRENCES(x, A, 1, longueur(A)) > longueur(A)/2
      alors renvoyer VRAI
      sinon renvoyer FAUX
```

4. Quelle est la complexité de cet algorithme ?

   La complexité de cet algorithme est en $\Theta(n)$ *car c'est la complexité de l'appel à l'algorithme* PSEUDO-MAJORITAIRE *et celle de l'appel à l'algorithme* OCCURRENCES.

5 TD 5 : plus longue sous-séquence commune

Problématique

Une séquence est une suite finie de symboles pris dans un ensemble fini. Si $u = \langle a_1, a_2, ..., a_n \rangle$ est une séquence, où a_1, a_2, ..., a_n sont des lettres, l'entier n est la *longueur* de u. Une séquence $v = \langle b_1, b_2, ..., b_m \rangle$ est une *sous-séquence* de $u = \langle a_1, a_2, ..., a_n \rangle$ s'il existe des entiers i_1, i_2, ..., i_m ($1 \leq i_1 < i_2 < ... < i_m \leq n$) tels que $b_k = a_{i_k}$ pour $k \in [1, m]$. Par exemple, $v = \langle B, C, D, B \rangle$ est une sous-séquence de $u = \langle A, B, C, B, D, A, B \rangle$ correspondant à la suite d'indice $\langle 2, 3, 5, 7 \rangle$.

Une séquence w est une *sous-séquence commune* aux séquences u et v si w est une sous-séquence de u et de v. Une sous-séquence commune est *maximale* ou est une *plus longue sous-séquence* si elle est de longueur maximale. Par exemple : les séquences $\langle B, C, B, A \rangle$ et $\langle B, D, A, B \rangle$ sont des plus longues sous-séquences communes de $\langle A, B, C, B, D, A, B \rangle$ et de $\langle B, D, C, A, B, A \rangle$.

Résolution par programmation dynamique

1. On cherche à déterminer la longueur d'une sous-séquence commune maximale à $u = \langle a_1, a_2, ..., a_n \rangle$ et $v = \langle b_1, b_2, ..., b_m \rangle$. On note $L(i, j)$ la longueur d'une sous-séquence commune maximale à $\langle a_1, a_2, ..., a_i \rangle$ et $\langle b_1, b_2, ..., b_j \rangle$ $(0 \leq j \leq m,\ 0 \leq i \leq n)$. Donnez une récurrence définissant $L(i, j)$. *Indication* : on pourra distinguer les cas $a_i = b_j$ et $a_i \neq b_j$.

$$L(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ 1 + L(i-1, j-1) & \text{sinon et si } a_i = b_j, \\ \max(L(i, j-1), L(i-1, j)) & \text{sinon.} \end{cases} \quad (1)$$

En effet, si $a_i \neq b_j$, une plus longue sous-séquence commune de $\langle a_1, a_2, ..., a_i \rangle$ et $\langle b_1, b_2, ..., b_j \rangle$ ne peut pas se terminer par une lettre c égale à a_i et à b_j, et donc une plus longue sous-séquence commune
- *soit ne se termine pas par a_i et elle est alors une plus longue sous-séquence commune des séquences $\langle a_1, a_2, ..., a_{i-1} \rangle$ et $\langle b_1, b_2, ..., b_j \rangle$, et est de longueur $L(i-1, j)$;*
- *soit ne se termine pas par b_j et elle est alors une plus longue sous-séquence commune des séquences $\langle a_1, a_2, ..., a_i \rangle$ et $\langle b_1, b_2, ..., b_{j-1} \rangle$, et est de longueur $L(i, j-1)$.*

Si, par contre, $a_i = b_j$, alors une plus longue sous-séquence commune de $\langle a_1, a_2, ..., a_i \rangle$ et $\langle b_1, b_2, ..., b_j \rangle$ se termine par $a_i = b_j$ (sinon on peut la rallonger par a_i) et son préfixe (la sous-séquence moins son dernier élément) est une plus longue sous-séquence commune de $\langle a_1, a_2, ..., a_{i-1} \rangle$ et $\langle b_1, b_2, ..., b_{j-1} \rangle$.

2. Écrivez alors un algorithme récursif calculant la longueur de la plus longue sous-séquence commune de deux séquences.

```
PLSSC-Récursif(A, i, B, j)
  si i = 0 ou j = 0
    alors renvoyer 0
    sinon si A[i] = B[j]
      alors renvoyer 1+ PLSSC-Récursif(A, i − 1, B, j − 1)
      sinon renvoyer max(PLSSC-Récursif(A, i − 1, B, j), PLSSC-Récursif(A, i, B, j − 1))
```

3. Montrez que cet algorithme est de complexité au moins exponentielle dans le cas où les deux séquences n'ont pas d'éléments en commun.

Dans ce cas, on se trouve toujours dans le troisième cas de l'équation 1 et la complexité est définie par la récurrence :

$$T(n,m) = \begin{cases} 0 & \textit{si } n = 0 \textit{ ou } m = 0, \\ T(n, m-1) + T(n-1, m) & \textit{sinon.} \end{cases}$$

$$\begin{aligned} \textit{D'où}\quad T(n,m) &= T(n, m-1) + T(n-1, m) \\ &= (T(n, m-2) + T(n-1, m-1)) + (T(n-1, m-1) + T(n-2, m-1)) \\ &\geq 2T(n-1, m-1). \end{aligned}$$

Donc $T(n, m) = \Omega(2^{\min(m,n)})$.

4. Écrivez alors un algorithme suivant le paradigme de la programmation dynamique et calculant la longueur de la plus longue sous-séquence commune de deux séquences.

```
PLSSC-ProgDyn(A, B)
  n ← longueur(A)
  m ← longueur(B)
  pour i ← 1 à n faire l[i, 0] ← 0
  pour j ← 1 à m faire l[0, j] ← 0
  pour i ← 1 à n faire
    pour j ← 1 à m faire
      si A[i] = B[j]
        alors l[i, j] ← 1 + l[i − 1, j − 1]
        sinon l[i, j] ← max(l[i − 1, j], l[i, j − 1])
  renvoyer l[n, m]
```

5. Quelle est la complexité de cet algorithme ?
   La complexité est due aux deux boucles imbriquées : $T(n, m) = O(nm)$.
6. Modifiez l'algorithme précédent pour que l'on puisse en plus construire *une* plus longue sous-séquence commune et affichez une telle sous-séquence.

```
PLSSC-ProgDyn(A, B)
  n ← longueur(A)
  m ← longueur(B)
  pour i ← 1 à n faire l[i, 0] ← 0
  pour j ← 1 à m faire l[0, j] ← 0
  pour i ← 1 à n faire
    pour j ← 1 à m faire
      si A[i] = B[j]
        alors l[i, j] ← 1 + l[i − 1, j − 1]
              s[i, j] ← « = »
        sinon si l[i − 1, j] > l[i, j − 1])
              alors l[i, j] ← l[i − 1, j]
                    s[i, j] ← « ← »
              sinon l[i, j] ← l[i, j − 1]
                    s[i, j] ← « → »
  renvoyer l[n, m] et s
```

```
Affichage-PLSC(s, A, i, j)
  si i = 0 ou j = 0 alors renvoyer
  si s[i, j] = « = »
    alors Affichage-PLSC(s, A, i − 1, j − 1)
          affiche A[i]
    sinon si s[i, j] = « ← » alors Affichage-PLSC(s, A, i − 1, j)
          si s[i, j] = « → » alors Affichage-PLSC(s, A, i, j − 1)
```

On appelle initialement PLSSC-ProgDyn(A, B) puis Affichage-PLSC(s, A, n, m).

6 TD 6 : Algorithmes gloutons

Le coût de la non panne sèche

Le professeur Bell conduit une voiture entre Amsterdam et Lisbonne sur l'autoroute E10. Son réservoir, quand il est plein, contient assez d'essence pour faire n kilomètres, et sa carte lui donne les distances entre les stations-service sur la route.

1. Donnez une méthode efficace grâce à laquelle Joseph Bell pourra déterminer les stations-service où il peut s'arrêter, sachant qu'il souhaite faire le moins d'arrêts possible.
   La solution consiste à s'arrêter le plus tard possible, c'est-à-dire à la station la plus éloignée du dernier point d'arrêt parmi celles à moins de n kilomètres de ce même point.
2. Démontrez que votre stratégie est optimale.
   On procède de la même manière que pour prouver l'optimalité de l'algorithme glouton pour la location de voiture : on considère une solution $F = (x_1, x_2, ..., x_p)$ fournie par notre algorithme et une solution optimale $G = (y_1, y_2, ..., y_q)$, on a donc $p \geq q$ et on cherche à montrer que $p = q$ —ici les deux suites sont bien sûr des suites de stations-service, triées de celle la plus proche du point de départ à celle la plus éloignée.
   Soit k le plus petit entier tel que :

$$\forall i < k, x_i = y_i, \text{ et } x_k \neq y_k.$$

Par définition de notre algorithme, on a $x_k > y_k$. On construit à partir de la solution optimale $G = (y_1, y_2, ..., y_{k-1}, y_k, y_{k+1}, ..., y_q)$ la suite de stations-service $G' = (y_1, y_2, ..., y_{k-1}, x_k, y_{k+1}, ..., y_q)$. G' est une liste de même taille. Montrons que c'est aussi une solution —et donc une solution optimale. Il nous faut vérifier qu'il n'y a pas de risque de panne sèche c'est-à-dire que :

- *$x_k - y_{k-1} \leq n$. F est une solution, donc $x_k - x_{k-1} \leq n$. Par conséquent, $x_k - y_{k-1} = x_k - x_{k-1} \leq n$.*
- *$y_{k+1} - x_k \leq n$. G est une solution, donc $y_{k+1} - y_k \leq n$. D'où $y_{k+1} - x_k < y_{k+1} - y_k \leq n$. Si $y_{k+1} < x_k$, on peut en fait supprimer y_{k+1} de G' et obtenir une solution strictement meilleure ce qui contredirait l'optimalité de G.*

G' est donc bien une solution optimale de notre problème.

Nous avons donc construit à partir de G une solution G' qui contient un élément en commun de plus avec F. On itère jusqu'à ce que l'on ait une solution optimale contenant F. Alors $p \leq q$, ce qui prouve le résultat attendu.

Problème du sac à dos

Variante « tout ou rien »

Un voleur dévalisant un magasin trouve n objets, le i^{e} de ces objets valant v_i euros et pesant w_i kilos, v_i et w_i étant des entiers. Le voleur veut bien évidemment emporter un butin de plus grande valeur possible mais il ne peut porter que W kilos dans son sac à dos. Quels objets devra-t-il prendre ?

Variante fractionnaire

Le problème est le même que le précédent, sauf qu'ici le voleur peut ne prendre qu'une *fraction* d'un objet et n'est plus obligé de prendre l'objet tout entier comme précédemment, ou de le laisser.

1. Proposez un algorithme glouton pour la variante fractionnaire.

   On commence par trier les objets suivant leur valeur au kilo, v_i/w_i, décroissante. L'algorithme consiste alors à prendre autant que faire se peut de l'objet de plus grande valeur au kilo —sa totalité si le sac à dos peut le contenir en entier et sinon la quantité nécessaire à remplir le sac— puis recommencer avec les autres objets jusqu'à ce que le sac soit plein.

2. Montrez que cet algorithme est optimal.

   On procède comme pour le problème de la location d'une voiture ou celui de la minimisation des arrêts en station service : on compare ici la solution construite par notre algorithme avec une solution optimale. On compare ces deux solutions après avoir triés les objets qu'elles contiennent par valeur au kilo décroissante. Pour le premier objet pour lequel les quantités diffèrent, on rajoute dans la solution optimale la quantité manquante de l'objet en question en enlevant le poids équivalent en objets de valeur au kilo inférieure —ce qui est toujours possible vu l'ordre dans lequel on compare les deux solutions. On conclut comme précédemment.

3. Quelle est sa complexité ?

   La complexité du tri est en $O(n \log n)$ (voir le cours) et celle de l'algorithme proprement dit en n, où n est le nombre d'objets. La complexité de l'ensemble est donc en $O(n \log n)$.

4. Montrez au moyen d'un contre-exemple que l'algorithme glouton équivalent pour la variante « tout ou rien » n'est pas optimal.

   On considère trois objets : le premier pèse 10 kilos et vaut 60 euros (valeur de 6 euros par kilo), le deuxième pèse 20 kilos et vaut 100 euros (valeur de 5 euros par kilo) et le troisième pèse 30 kilos et vaut 120 euros (valeur de 4 euros par kilo). La stratégie gloutonne précédente prendra initialement le premier objet et ne pourra plus alors en prendre d'autre, quand la solution optimale ici aurait été de prendre les deux autres objets...

5. Proposez un algorithme de programmation dynamique résolvant la variante « tout ou rien ».

   On commence par établir une récurrence définissant la valeur du butin emporté. $SAD(i, w)$ est la valeur maximale du butin s'il n'est composé que de certains des i premiers objets (les objets sont ici numérotés mais pas ordonnés) et qu'il pèse au maximum w kilos. On a plusieurs cas à considérer :

(a) *Le i^e objet est plus lourd que la capacité du sac : on ne peut pas le prendre et le problème se ramène à la recherche du meilleur butin parmi les $i-1$ premiers objets.*

(b) *Il est possible de prendre le i^e objet :*

i. *On ne le prend pas et le problème se ramène à la recherche du meilleur butin parmi les $i-1$ premiers objets.*

ii. *On le prend et la valeur du butin est celle du i^e objet plus celle du butin que l'on peut constituer à partir des $i-1$ premiers objets compte tenu que l'on a pris le i^e objet et que le poids portable est diminué d'autant.*

De ce qui précède on tire l'équation :

$$SAD(i,w) = \begin{cases} SAD(i-1,w) & \text{si } w < w_i, \\ \max(v_i + SAD(i-1, w-w_i), SAD(i-1,w)) & \text{sinon.} \end{cases} \quad (2)$$

Il nous reste à transcrire cette définition récursive sous la forme d'un algorithme de programmation dynamique :

SacÀDos(W, w, v)
  pour $w \leftarrow 0$ **à** W **faire** $Valeur[0,w] \leftarrow 0$
  pour $i \leftarrow 1$ **à** n **faire**
    pour $w \leftarrow 0$ **à** W **faire**
      si $w_i > w$
        alors $Valeur[i,w] \leftarrow Valeur[i-1,w]$
        sinon $Valeur[i,w] \leftarrow \max(v_i + Valeur[i-1, w-w_i], Valeur[i-1,w])$
renvoyer *Valeur*

Cet algorithme nous calcule la valeur optimale du butin ($Valeur(n,W)$) mais pas sa composition, ce que fait l'algorithme suivant :

Butin(*Valeur*, i, W, w, v)
      si $w_i > W$
        alors renvoyer Butin(*Valeur*, $i-1$, W, w, v)
        sinon si $v_i + S[i-1, W-w_i] > S[i-1,W]$
          alors renvoyer $\{i\} \cup$ Butin(*Valeur*, $i-1$, $W-w_i$, w, v)
          sinon renvoyer Butin(*Valeur*, $i-1$, W, w, v)

Cet algorithme est appelé initialement : Butin*(*SacÀDos*(w, v), n, W, w, v).*

6. Quelle est sa complexité ?
   La complexité de l'algorithme SacÀDos *est en $\Theta(nW)$ et celle de l'algorithme* Butin *est en $\Theta(n)$ (à chaque étape on décrémente i de 1). La complexité totale est donc en $\Theta(nW)$.*

7 TD 7 : Algorithmes gloutons

Emploi du temps de salles

On suppose que l'on a un ensemble de n cours, c_1, ..., c_n, le cours c_i commençant à l'heure *début*(c_i) et finissant à l'heure *fin*(c_i). Écrivez un algorithme qui attribue une salle à chaque cours sachant que l'on veut minimiser le nombre de salles occupées et que l'on ne veut pas que deux cours aient lieu en même temps dans la même salle.

Salles(c)
  Trier les cours par dates de début croissantes
  $n_{salles} \leftarrow 0$
  pour $i \leftarrow 1$ **à** n **faire**
    si parmi les n_{salles} utilisées il n'existe pas de salle libre à partir de l'heure *début*(c_i)

alors $n_{salles} \leftarrow n_{salles} + 1$
affecter c_i à la salle n_{salles}
sinon affecter c_i à une des salles disponibles

Si l'algorithme augmente, à cause du cours c_i, le nombre de salles utilisées (n_{salles}) alors, par définition de l'algorithme, il y a cours à la date début(c_i) dans chacune des n_{salles} utilisées. En comptant c_i, il y a donc au moins $(1 + n_{salles})$ cours qui ont lieu à la date début(c_i) et il faut donc au moins $(1 + n_{salles})$ salles pour les abriter. L'algorithme est donc optimal.

Sous-graphes acycliques

Soit $G = (S, A)$ un graphe non orienté. On définit un ensemble I de sous-ensembles de A comme suit : F appartient à I si et seulement si (S, F) est un graphe acyclique.

1. Montrez que (A, I) est un matroïde.

   Pour montrer que (A, I) est un matroïde, nous devons montrer qu'il vérifie trois propriétés :

   A **est un ensemble fini.** *C'est une évidence.*

   I **est héréditaire.** *Soient H un élément de I ($H \in I$) et soit F un sous-ensemble de H ($F \subset H$). (S, F) est trivialement un graphe. De plus, c'est un sous-graphe de (S, H). Donc, si (S, F) contenait un cycle, il en irait de même de (S, H). (S, H) étant acyclique, (S, F) est acyclique et I contient F ($F \in I$).*

   Propriété d'échange. *Soient F et H deux éléments de I, $|F| < |H|$. On doit montrer qu'il existe au moins un élément x de $H \setminus F$ tel que $F \cup \{x\}$ soit élément de I.*

   Nous découpons le problème en cas :

   Sommets isolés différents. *Si H contient une arête x incidente à un sommet isolé dans (S, F), $(S, F \cup \{x\})$ est acyclique et $(F \cup \{x\}) \in I$.*

   Mêmes sommets isolés. *Les sommets isolés de F sont aussi isolés dans H. De deux choses l'une : soit F et H n'ont pas les mêmes composantes connexes, soit ils ont les mêmes.*

   Composantes connexes différentes. *Soit x une arête entre deux sommets u et v appartenant à des composantes connexes différentes de (S, F). Si $(S, F \cup \{x\})$ n'est pas acyclique, alors il existait déjà une chaîne dans (S, F) reliant u et v ce qui contredit l'hypothèse que u et v appartiennent à des composantes connexes différentes de (S, F). Donc $F \cup \{x\} \in I$.*

   Mêmes composantes connexes. *Les composantes connexes des deux graphes sont exactement les mêmes. Chacune de ces composantes connexes est un sous-graphe connexe acyclique de H (resp. F) et contient donc une arête de moins que de sommets (d'après le point 4 du théorème 7 de caractérisation des arbres vu en cours). Par conséquent, le nombre d'arêtes de H (resp. F) est exactement le nombre de ses sommets, $|S|$, moins le nombre de ses composantes connexes. Donc, F et H ont le même nombre d'éléments, ce qui contredit l'hypothèse : $|F| < |H|$. Ce cas est donc impossible.*

   On pourrait bien sûr se contenter ne n'étudier que les cas où les composantes connexes sont les mêmes ou sont différentes. Il m'a semblé que le cheminement suivi ici était plus naturel...

2. Proposez un algorithme pour calculer un plus grand (en nombre d'arêtes) sous-graphe acyclique de G.

   Pour utiliser les résultats vus en cours, il nous faut un matroïde pondéré. Il nous manque donc juste la fonction de pondération. Il nous suffit ici d'utiliser une fonction qui associe un poids de 1 à chaque arête !

   PLUS-GRAND-SOUS-GRAPHE-ACYCLIQUE($G = (S, A)$)
   $F \leftarrow \emptyset$
   pour x appartenant à A **faire si** $(S, F \cup \{x\})$ est acyclique **alors** $F \leftarrow F \cup \{x\}$
   renvoyer (S, F)

   Tout simplement !

Ordonnancement de tâches avec priorités

On doit réaliser un ensemble de n tâches T_1, ..., T_n sur une unique machine. Chaque tâche T_i a une durée d_i et une priorité p_i. Une réalisation des tâches T_1, ..., T_n en est une permutation T_{i_1}, T_{i_2}, ..., T_{i_n} représentant l'ordre dans lequel elles sont exécutées. La date de fin d'exécution F_i d'une tâche T_i est égale à la somme des durées des tâches qui la précèdent dans la permutation plus sa durée propre (d_i). La *pénalité* d'une exécution vaut $\sum_{i=1}^{n} p_i F_i$. On cherche un ordonnancement qui minimise cette pénalité.

1. Soit quatre tâches de durées respectives 3, 5, 7 et 4, et de priorités respectives 6, 11, 9 et 5. Des deux réalisations (T_1, T_4, T_2, T_3) et (T_4, T_1, T_3, T_2), quelle est la meilleure ?
   (a) *(T_1, T_4, T_2, T_3) : $F_1 = d_1 = 3$, $F_4 = d_1+d_4 = 7$, $F_2 = d_1+d_4+d_2 = 12$, $F_3 = d_1+d_4+d_2+d_3 = 19$. La pénalité vaut donc : $\sum_{i=1}^{4} p_i F_i = 6\times 3 + 11\times 12 + 9\times 19 + 5\times 7 = 18 + 132 + 171 + 35 = 356$.*
   (b) *(T_4, T_1, T_3, T_2) : $F_4 = d_4 = 4$, $F_1 = d_4+d_1 = 7$, $F_3 = d_4+d_1+d_3 = 14$, $F_2 = d_4+d_1+d_3+d_2 = 19$. La pénalité vaut donc : $\sum_{i=1}^{4} p_i F_i = 6\times 7 + 11\times 19 + 9\times 14 + 5\times 4 = 42 + 209 + 126 + 20 = 397$.*

   La première réalisation est donc la meilleure des deux.
2. Soient deux tâches T_i et T_j consécutives dans une réalisation telles que : $\frac{p_i}{d_i} < \frac{p_j}{d_j}$. Afin de minimiser la pénalité, laquelle doit être exécutée en premier et laquelle en second ?

   On a donc deux réalisations identiques sauf que dans la première T_i est exécutée juste avant T_j et dans la deuxième T_j est exécutée juste avant T_i. Soit d la somme de la durée des tâches exécutée dans la première (resp. deuxième) réalisation avant le début de l'exécution de d_i (resp. d_j). Les pénalités des deux réalisations ne diffèrent que pour les termes relatifs à T_i et T_j :

   Première réalisation. $(d+d_i)p_i + (d+d_i+d_j)p_j = ((d+d_i)p_i + (d+d_j)p_j) + d_i p_j$.

   Deuxième réalisation. $(d+d_j)p_j + (d+d_j+d_i)p_i = ((d+d_j)p_j + (d+d_i)p_i) + d_j p_i$.

   Comme $\frac{p_i}{d_i} < \frac{p_j}{d_j}$, $d_j p_i < p_j d_i$ et la deuxième réalisation est moins coûteuse que la première. Par conséquent, T_j doit être exécutée avant T_i.
3. En déduire un algorithme.

   Dans une solution optimale on doit donc forcément avoir $\frac{p_{i+1}}{d_{i+1}} < \frac{p_i}{d_i}$, ce qui ne nous laisse guère le choix quant à la solution.

   Ordo-Avec-Priorités(T, d, p)

   Exécuter les tâches par ratio $\frac{p_i}{d_i}$ décroissant

8 TD 8 : Dénombrement sur les arbres binaires

Dénombrement sur les arbres binaires

Dans cet exercice on notera n le nombre de nœuds d'un arbre binaire, f son nombre de feuilles et h sa hauteur. Tous les arbres considérés seront supposés non vides.

1. Quelle est la hauteur maximale d'un arbre à n nœuds ?

   Avec n nœuds on construit un arbre de hauteur maximale quand chaque nœud, excepté la feuille, a un unique fils. L'arbre n'a alors qu'une seule branche qui contient les n nœuds et qui est de longueur $n-1$. La hauteur maximale réalisable est donc $n-1$ et on a toujours : $h \leq n-1$.
2. Quel est le nombre maximal de feuilles d'un arbre de hauteur h ?

   Si $h = 0$, c'est-à-dire si l'arbre se réduit à sa racine, $f = 1$. Sinon, on raisonne par récurrence : le sous-arbre gauche (resp. droit) de la racine est un arbre de hauteur $h-1$ et il a un nombre maximal de feuilles pour un arbre de hauteur $h-1$. Si on note $f(h)$ le nombre maximal de feuilles d'un arbre de hauteur h on a donc :

$$f(h) = \begin{cases} 1 & \text{si } h = 0, \\ 2 \times f(h-1) & \text{sinon.} \end{cases}$$

   On a donc $f(h) = 2^h$. Le nombre maximal de feuilles d'un arbre de hauteur h est donc 2^h et on a toujours : $f \leq 2^h$.

3. Quel est le nombre maximal de nœuds d'un arbre de hauteur h ?

   Si $h = 0$, c'est-à-dire si l'arbre se réduit à sa racine, $n = 1$. Sinon, on raisonne par récurrence : le sous-arbre gauche (resp. droit) de la racine est un arbre de hauteur $h-1$ et il a un nombre maximal de nœuds pour un arbre de hauteur $h-1$. Si on note $n(h)$ le nombre maximal de nœuds d'un arbre de hauteur h on a donc :

$$n(h) = \begin{cases} 1 & \text{si } h = 0, \\ 1 + 2 \times n(h-1) & \text{sinon.} \end{cases}$$

   On a donc $n(h) = 2^{h+1} - 1$. Le nombre maximal de nœuds d'un arbre de hauteur h est donc $2^{h+1} - 1$ et on a toujours : $n \leq 2^{h+1} - 1$.

4. Quelle est la hauteur minimale d'un arbre de n nœuds ?

   La minoration de la hauteur est une conséquence directe du résultat de la question précédente :

$$n \leq 2^{h+1} - 1 \quad \Leftrightarrow \quad n + 1 \leq 2^{h+1} \quad \Leftrightarrow \quad \log_2(n+1) \leq h + 1 \quad \Leftrightarrow \quad h \geq \log_2(n+1) - 1$$

   D'où la minoration : $h \geq \lceil \log_2(n+1) - 1 \rceil$.

5. Montrez que le nombre de branches vides — nombre de fils gauches et de fils droits vides — d'un arbre à n nœuds est égal à $n + 1$.

   Indication : on distinguera les nœuds ayant zéro, un et deux fils.

   On note :
   - *p le nombre de nœuds ayant zéro fils ;*
   - *q le nombre de nœuds ayant un fils ;*
   - *r le nombre de nœuds ayant deux fils.*

   On a les relations suivantes :
   - *$n = p + q + r$: un nœud a soit zéro, soit un, soit deux fils.*
   - *$0 \times p + 1 \times q + 2 \times r = n - 1$: tous les nœuds, la racine exceptée, ont un père ; on compte ces $n-1$ nœuds à partir de leurs pères, $0 \times p$ étant fils d'un nœud sans fils, $1 \times q$ étant fils d'un nœud ayant un unique fils et $2 \times r$ étant fils d'un nœud ayant deux fils.*
   - *$2 \times p + 1 \times q + 0 \times r = b$: on compte les branches vides de chaque nœud.*

   En additionnant les deux dernières équations on obtient :

$$2(p + q + r) = b + n - 1 \quad \Leftrightarrow \quad 2n = b + n - 1 \quad \Leftrightarrow \quad b = n + 1$$

   en utilisant la première équation.

   ***Autre solution**. On raisonne par récurrence : un arbre réduit à un unique nœud a deux branches vides, donc quand $n = 1$, $b = n + 1$. On suppose la propriété vérifiée pour tous les arbres contenant au plus n nœuds. Soit A un arbre à $n+1$ nœuds. On supprime arbitrairement une feuille de A, ce qui nous donne un arbre B à n nœuds et $n+1$ branches vides, d'après l'hypothèse de récurrence. Pour passer de A à B on a supprimé de A une feuille, et donc deux branches vides, et on a rajouté une branche vide au père de la feuille enlevée. Donc B a une branche vide de moins que A qui en a donc $(n+1)+1$, CQFD.*

6. Montrez que le nombre de feuilles est inférieur ou égal à $\frac{n+1}{2}$ ($f \leq \frac{n+1}{2}$) et qu'il y a égalité si et seulement si chaque nœud de l'arbre est soit une feuille, soit a deux fils.

   Indication : se servir du raisonnement utilisé à la question précédente.

   On a vu précédemment que l'on avait : $2 \times p + 1 \times q + 0 \times r = b$ avec $b = n + 1$. Or $p = f$: les nœuds sans fils sont exactement les feuilles ! q étant positif, on a $2 \times f \leq n+1$ ce qui établit l'inégalité recherchée, et on a égalité quand $q = 0$, c'est-à-dire quand l'arbre ne contient pas de nœuds ayant un seul fils.

7. Montrez que le nombre de feuilles d'un arbre est égal au nombre de nœuds de degré deux, plus un.

   On se sert une fois de plus des relations entre p, q et r : $p + q + r = n$ et $q + 2r = n - 1$. En éliminant q entre les deux équations on obtient : $p = r + 1$ qui est le résultat recherché.

Complexité du tri

Les algorithmes de tris par comparaison peuvent être considérés de façon abstraite en termes d'**arbres de décision**. Un arbre de décision représente les comparaisons (à l'exclusion de toute autre opération) effectuées par un algorithme de tri lorsqu'il traite une entrée de taille donnée. La figure 3 présente l'arbre de décision correspondant à l'algorithme de tri par insertion s'exécutant sur une liste de trois éléments.

Soient $\langle a_1, a_2, ..., a_n \rangle$ les n valeurs à trier. Dans un arbre de décision chaque nœud interne est étiqueté par une expression $a_i \leq a_j$, pour certaines valeurs de i et de j, $1 \leq i, j \leq n$. L'exécution de l'algorithme de tri suit un chemin qui part de la racine de l'arbre de décision pour aboutir à une feuille. À chaque nœud interne, on effectue une comparaison $a_i \leq a_j$ et les comparaisons suivantes auront lieu dans le sous-arbre gauche si $a_i \leq a_j$ et dans le sous-arbre droit sinon. Chaque feuille est désignée par une permutation $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$: si l'algorithme de tri aboutit en la feuille $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$, les valeurs à trier vérifient : $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$.

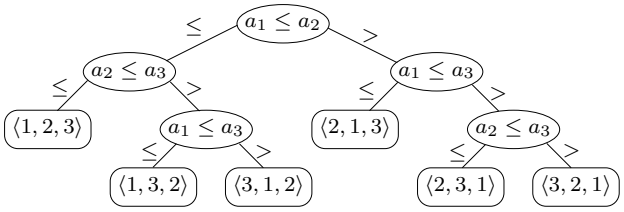

FIG. 3 – Arbre de décision correspondant au traitement de trois éléments au moyen du tri par insertion.

1. Quel est le nombre de feuilles d'un tel arbre de décisions ?
   On a $n!$ permutations possibles de n éléments, donc $n!$ feuilles possibles à notre arbre de décision.
2. En déduire une borne inférieure sur la hauteur de l'arbre de décision.
   On a vu précédemment que : $f \leq 2^h$ ce qui équivaut à $h \geq \log_2 f$ avec ici $f = n!$. Donc $h \geq \log_2(n!)$.
3. En déduire une *borne inférieure* sur la complexité du tri de n éléments.
   Indication : d'après la formule de Stirling, on a $n! > \left(\frac{n}{e}\right)^n$.
   La longueur d'un chemin de la racine à une feuille dans l'arbre de décision est égale au nombre de comparaisons nécessaires au tri pour parvenir à la réponse souhaitée. La longueur du plus long chemin de la racine à une feuille — qui est égale par définition à la hauteur de l'arbre — nous donne donc la complexité $T(n)$ de l'algorithme dans le pire cas. D'après la question précédente, la hauteur d'un tel arbre de décision, quel que soit l'algorithme de tri, est au moins de $\log_2(n!)$. Donc, en utilisant la formule de Stirling : $T(n) \geq \log_2(n!) > n \log(\frac{n}{e})$, d'où, comme $\log(ab) = \log(a) + \log(b)$:

$$T(n) = \Omega(n \log n).$$

Arbres binaires de recherche

1. Montrez que le temps de création d'un arbre binaire de recherche à partir d'une liste quelconque de n éléments est $\Omega(n \log n)$.
   Partant d'une liste quelconque de n éléments, si nous construisons un arbre binaire de recherche que nous parcourons en affichant les clés suivant un parcours (en profondeur) infixe, on obtient la liste des n éléments triés. Vu le résultat obtenu à l'exercice précédent la complexité de cette manipulation, qui est un tri, est $\Omega(n \log n)$. Le parcours avec affichage étant de complexité $\Theta(n)$, la construction de l'arbre binaire de recherche est $\Omega(n \log n)$.
2. Écrivez un algorithme qui teste si un arbre binaire est un arbre binaire de recherche.

EstArbreDeRecherche(x)
  (*booléen*, *min*, *max*) ← VérifieArbre(x)
  renvoyer *booléen*

VérifieArbre(x)
  $(b_1, min_1, max_1) \leftarrow$ VérifieArbre($gauche(x)$)
  si $b_1 =$ Faux **alors renvoyer** (Faux, 0, 0)
  $(b_2, min_2, max_2) \leftarrow$ VérifieArbre($droit(x)$)
  si $b_2 =$ Faux **alors renvoyer** (Faux, 0, 0)
  si $max_1 \leq clé(x) \leq min_2$
    alors renvoyer (Vrai, min_1, max_2)
    sinon renvoyer (Faux, 0, 0)

9 TD 9 : Tri topologique

Un **tri topologique** d'un graphe orienté acyclique $G = (S, A)$ est un ordre linéaire des sommets de G tel que si G contient l'arc (u, v), u apparaît avant v. Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale tel que tous les arcs soient orientés de gauche à droite.

Attention : le tri topologique d'un graphe orienté acyclique n'est pas forcément unique.

1. Proposez un tri topologique du graphe de la figure 1.

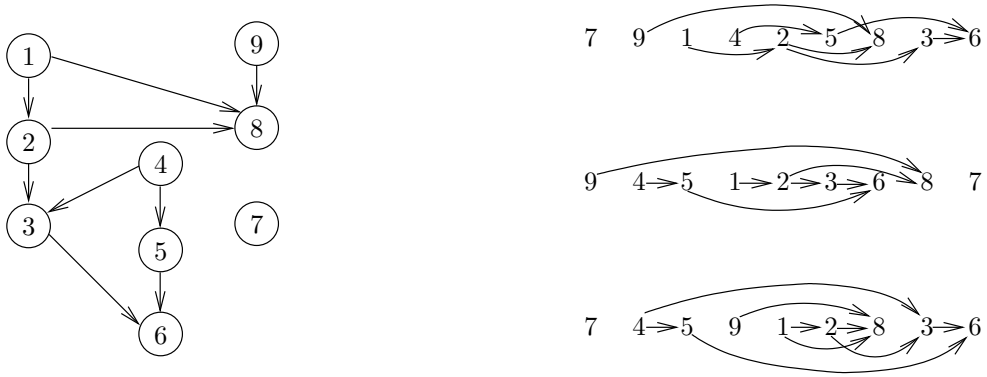

Fig. 4 – Exemple de graphe orienté acyclique avec trois de ses tris topologiques possibles.

2. Modifiez l'algorithme de parcours en profondeur vu en cours pour qu'il calcule pour chaque nœud u sa date de fin de traitement *fin*[u] (la date à laquelle lui et ses fils ont été traités).

PP(G)
  pour chaque sommet u de G **faire** *couleur*[u] ← Blanc
  pour chaque sommet u de G **faire si** *couleur*[u] = Blanc **alors** Visiter-PP(G, u, *couleur*)

Visiter-PP(G, s, *couleur*)
  couleur[s] ← Gris
  pour chaque voisin v de s **faire**
    si *couleur*[v] = Blanc **alors** Visiter-PP(G, v, *couleur*)
  couleur[s] ← Noir
  temps ← *temps* + 1
  fin[s] ← *temps*

3. Quel lien pouvez-vous faire entre les dates de fin de traitement et un tri topologique ?
   Si le graphe G contient l'arc (u, v), le traitement de u finira après *le traitement de v, et on aura donc $fin[u] > fin[v]$. Trier les sommets par date de fin de traitements décroissantes nous fournit donc un tri topologique du graphe. La figure 5 présente le graphe de la figure 1 où les nœuds ont été annotés avec des fins de date de traitement lors d'un parcours en profondeur (¡¡ racines ¿¿ successives : nœuds 1, 9, 4 et 7). On retrouve alors le troisième des tris topologiques de la figure 1.*

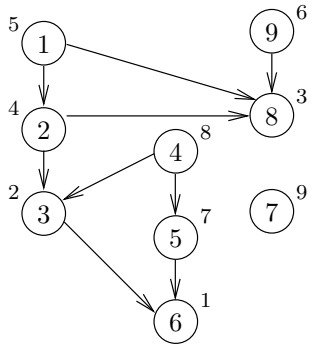

FIG. 5 – Graphe annoté par les dates de fin de traitement d'un parcours en profondeur.

4. Proposez un algorithme de tri topologique. Quel est sa complexité ?
   On commence par parcourir le graphe en profondeur avec l'algorithme de la question 2, puis on trie les sommets par date de fin décroissante. La complexité de l'ensemble est donc celle du parcours $O(|S|+|A|)$ plus celle du tri $O(|S|\log(|S|))$, soit $O(|S|\log(|S|)+|A|)$.
5. Améliorez votre algorithme pour qu'il soit de complexité linéaire.

```
PP(G)
  Soit P une pile initialement vide
  pour chaque sommet u de G faire couleur[u] ← BLANC
  pour chaque sommet u de G faire si couleur[u] = BLANC alors VISITER-PP(G, u, couleur, P)
  tant que non PILE-VIDE(P) faire
    u ← DÉPILER(P)
    AFFICHER(u)

VISITER-PP(G, s, couleur, P)
  couleur[s] ← GRIS
  pour chaque voisin v de s faire
      si couleur[v] = BLANC alors VISITER-PP(G, v, couleur, P)
  couleur[s] ← NOIR
  EMPILER(P,s)
```

   On stocke dans une pile les éléments dans l'ordre dans lequel leur traitement se termine. On retirera les éléments de la pile dans l'ordre inverse de celui dans lequel ils ont été insérés (une pile fonctionne toujours sur le mode ¡¡ premier entré, dernier sorti ¿¿). Utiliser une pile plutôt que simplement les dates de fin de traitement nous évite d'avoir à trier ces dates (ce qui nous coûterait $O(|S|\log(|S|))$. L'algorithme complet a ici un coût de $O(|S|+|A|)$ (le parcours est de coût $O(|S|+|A|)$ et la pile contient $|S|$ éléments, donc les dépiler est de coût $O(|S|)$).
6. Proposez un autre algorithme de tri topologique, basé cette fois-ci sur le fait qu'un sommet de degré entrant nul peut être placé en tête d'un tri topologique. Quelle est la complexité de cet algorithme ?
   La figure 6 présente le second algorithme de tri topologique. Un sommet qui est placé dans la file, est un sommet dont tous les prédécesseurs ont déjà été ordonnés : on peut donc le placer à son tour dans l'ordre sans risquer de violer un arc. La complexité de l'ensemble est une fois de plus en $O(|S|+|A|)$.

```
TRI-TOPOLOGIQUE(G = (S, A))
  Soit F une file initialement vide
  pour chaque sommet u de G faire
    degré(u) ← degré entrant de u
    si degré(u) = 0 alors INSERTION(F, u)
  tant que non FILE-VIDE(F) faire
    u ← SUPPRESSION(F)
    AFFICHER(u)
    pour chaque voisin v de u faire
      degré(v) ←degré(v) − 1
      si degré(v) = 0 alors INSERTION(F, v)
```

FIG. 6 – Deuxième algorithme de tri topologique.

10 TD 10 : Circuit de poids moyen minimal

Soit $G = (S, A)$ un graphe orienté, de fonction de pondération $w : A \to \mathbb{R}$, et soit $n = |S|$. On définit le **poids moyen** d'un circuit c formé des arcs $\langle e_1, e_2, ..., e_k \rangle$ ($c = \langle e_1, e_2, ..., e_k \rangle$) par :

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Soit $\mu^* = \min_{c \text{ circuit de } G} \mu(c)$. Un circuit c pour lequel $\mu(c) = \mu*$ est appelé **circuit de poids moyen minimal**. Nous étudions ici l'algorithme de Karp, qui est un algorithme efficace permettant de calculer μ^*.

Nous supposons, sans perte de généralité, que chaque sommet $v \in S$ est accessible à partir d'un sommet origine $s \in S$. Soit $\delta(s, v)$ le poids d'un plus court chemin de s vers v, et soit $\delta_k(s, v)$ le poids d'un plus court chemin de s vers v composé exactement de k arcs (si un tel chemin n'existe pas, $\delta_k(s, v) = +\infty$).

1. Montrez que le minimum μ^* est effectivement atteint, et est atteint sur un circuit élémentaire.
   Soit $\mathcal{C}$ un circuit de G. Si $\mathcal{C}$ n'est pas un circuit élémentaire, il existe un sommet u de G qui est visité au moins deux fois par $\mathcal{C}$. On décompose $\mathcal{C}$ comme suit : $\mathcal{C} = u \overset{c_1}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} u...u \overset{c_k}{\rightsquigarrow} u$ où chaque circuit c_i ne contient pas le sommet u comme sommet intermédiaire. Si un des circuits c_i n'est pas élémentaire, on le décompose comme on l'a fait pour $\mathcal{C}$. Finalement, on obtient une décomposition de $\mathcal{C}$ en somme de circuits élémentaires : $\mathcal{C} = \sum_{i=1}^{p} c_i$. $\mu(\mathcal{C}) = \frac{\sum_{i=1}^{p} w(c_i)}{\sum_{i=1}^{p} l(c_i)}$, où $l(c_i)$ est le nombre d'arcs constituant le circuit c_i. Soit $k \in [1, p]$ tel que $\frac{w(c_k)}{l(c_k)} \leq \min_{1 \leq i \leq p} \frac{w(c_i)}{l(c_i)}$. On a alors :

$$\mu(\mathcal{C}) = \frac{\sum_{i=1}^{p} w(c_i)}{\sum_{i=1}^{p} l(c_i)} \geq \frac{\sum_{i=1}^{p} w(c_k) \frac{l(c_i)}{l(c_k)}}{\sum_{i=1}^{p} l(c_i)} = \frac{w(c_k)}{l(c_k)}.$$

   Donc le poids moyen d'un circuit non élémentaire est toujours supérieur au minimum des poids moyens de ces circuits constitutifs et, sans perte de généralité, on peut calculer μ^* en ne considérant que les circuits élémentaires. Le nombre de circuits élémentaires étant bien évidemment fini (par exemple : un circuit définit un sous-ensemble des arcs, et le nombre de sous-ensembles d'un ensemble fini est fini), le minimum est atteint sur un circuit.

2. Montrez que si $\mu^* = 0$:
   (a) G ne contient aucun circuit de poids strictement négatif ;
   Si c est un circuit de poids strictement négatif, $\mu(c) < 0$ et, comme $\mu^* = 0$, $\mu(c) < \mu^*$, ce qui est absurde.
   (b) $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ pour tous les sommets $v \in S$.
   Comme G ne contient pas de circuits de poids strictement négatifs, d'après la question précédente, les plus courts chemins sont bien définis. Si d est le poids d'un plus court chemin de s à v, il existe

un chemin élémentaire de s à v de poids d : on prend un plus court chemin dont on ôte les éventuels circuits et, comme tous les circuits sont de poids positifs, on obtient un circuit encore plus court, donc de même poids. Un chemin élémentaire de G contient au plus $n-1$ arcs puisque G contient n sommets. D'où le résultat.

3. Montrez que, si $\mu^* = 0$,
$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \geq 0$$
pour tout sommet $v \in S$. (*Indication* : on utilisera les deux propriétés démontrée à la question 2).

$$\begin{array}{rl} \max_{0\leq k\leq n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} \geq 0 & \Leftrightarrow \\ \max_{0\leq k\leq n-1}(\delta_n(s,v) - \delta_k(s,v)) \geq 0 & \Leftrightarrow \\ \delta_n(s,v) - \min_{0\leq k\leq n-1} \delta_k(s,v) \geq 0 & \Leftrightarrow \\ \delta_n(s,v) - \delta(s,v) \geq 0 & \Leftrightarrow \\ \delta_n(s,v) \geq \delta(s,v) & \end{array}$$

La dernière équation étant vraie d'après les résultats de la question 2, résultats qui nous garantissent également que l'expression $\delta(s,v)$ est bien définie.

4. On suppose ici que $\mu^* = 0$. Soit c un circuit de poids nul, et soient u et v deux sommets quelconques de c. Soit x le poids du chemin de u à v le long du circuit c. Démontrez que $\delta(s,v) = \delta(s,u) + x$.
(*Indication* : quel est le poids du chemin de v à u le long du circuit c ?)
On peut construire un chemin de s à v en concaténant un plus court chemin de s à u (de poids $\delta(s,u)$) avec le chemin de u à v le long de c (chemin de poids x). Ce chemin a un poids supérieur ou égal à celui d'un plus court chemin de s à v. D'où :
$$\delta(s,u) + x \leq \delta(s,v).$$
On construit un chemin de s à u symétriquement : on prend un plus court chemin de s à v, auquel on concatène le chemin de v à u le long de c (chemin de poids $-x$ car c est de poids nul et que le chemin de u à x le long de c a pour poids x). On obtient ainsi :
$$\delta(s,v) - x \leq \delta(s,u).$$
En combinant ces deux équations on obtient le résultat escompté.

5. Montrez que, si $\mu^* = 0$, il existe un sommet v sur le circuit de poids moyen minimal tel que
$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$
(*Indication* : montrez qu'un plus court chemin vers un sommet quelconque du circuit de poids moyen minimal peut être étendu le long du circuit pour créer un plus court chemin vers le sommet suivant dans le circuit.)
D'après la question 1, le minimum μ^* est atteint sur un circuit élémentaire. Soit c un tel circuit. On a donc $w(c) = 0$. Soit u_0 un sommet quelconque sur c. Soit p un plus court chemin de s à u_0 et soit $l(p)$ la longueur de ce chemin en nombre d'arcs. Pour $1 \leq i \leq n - l(p)$ on définit u_i comme le sommet suivant u_{i-1} le long du circuit c, et c_i comme le chemin de u_0 à u_i le long du circuit c. D'après la question précédente on a : $\delta(s,u_{n-l(p)}) = \delta(s,u_0) + w(c_{n-l(p)})$. Par construction, le chemin de s à u_0 puis de u_0 à $u_{n-l(p)}$ contient exactement n arcs. On a donc $\delta(s,u_{n-l(p)}) = \delta_n(s,u_{n-l(p)})$. Or l'équation à démontrer est équivalente à : $\delta_n(s,v) = \delta(s,v)$.

6. Montrez que, si $\mu^* = 0$,
$$\min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$
Cette question est une bête combinaison des résultats des questions 3 et 5.

7. Montrez que si l'on ajoute une constante t au poids de chaque arc de G, μ^* est également augmenté de t. Servez-vous de cette propriété pour montrer que

$$\mu^* = \min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}.$$

On note w' la nouvelle fonction de pondération : $\forall a \in A, w'(a) = t + w(a)$. Soit c un circuit quelconque de G : $\mu'(c) = \frac{1}{k}\sum_{i=1}^{k} w'(e_i) = \frac{1}{k}\sum_{i=1}^{k}(t + w(e_i)) = t + \mu(c)$. D'où $\mu'^* = t + \mu^*$.
À chaque arc on rajoute un poids $t = (-\mu^*)$. On obtient alors comme poids moyen minimal : $\mu'^* = 0$ et, d'après la question précédente,

$$\min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta'_n(s,v) - \delta'_k(s,v)}{n-k} = 0.$$

Or $\delta'_n(s,v) - \delta'_k(s,v) = \delta_n(s,v) - \delta_k(s,v) + (n-k) \times (-\mu^*)$. D'où,

$$\min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v) + (n-k) \times (-\mu^*)}{n-k} = 0 \Leftrightarrow$$

$$\mu^* = \min_{v \in S} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}.$$

8. Proposez un algorithme permettant de calculer μ^*. Quelle est sa complexité ?

KARP
pour chaque sommet u de G **faire**
  $\delta(s,v) \leftarrow +\infty$
  pour $k \leftarrow 1$ **à** n **faire** $\delta_k(s,v) \leftarrow +\infty$
$\delta_0(s,s) \leftarrow 0$
pour $k \leftarrow 1$ **à** n **faire**
  pour chaque arc (u,v) de G **faire**
    si $\delta_k(s,v) > \delta_{k-1}(s,u) + w(u,v)$ **alors**
      $\delta_k(s,v) \leftarrow \delta_{k-1}(s,u) + w(u,v)$
      si $\delta_k(s,v) < \delta(s,v)$ **alors**
        $\delta(s,v) \leftarrow \delta_k(s,v)$
        $k(v) \leftarrow k$
$\mu^* \leftarrow +\infty$
pour chaque sommet u de G **faire**
  si $\mu^* > \frac{\delta_n(s,u) - \delta(s,u)}{n-k(u)}$ **alors** $\mu^* \leftarrow \frac{\delta_n(s,u) - \delta(s,u)}{n-k(u)}$

Remarque : d'après la question 2, $k(u)$ ne peut pas être égal à n et il n'y a pas de risque de division par zéro dans l'algorithme.
La complexité de l'algorithme est en $O(|S|^2 + |S|.|A|)$: $O(|S|^2)$ pour les initialisations et $O(|S|.|A|)$ pour le calcul proprement dit. On pourrait réduire le coût de l'initialisation des valeurs de $\delta_k(s,v)$ en utilisant la boucle de calcul suivante, plus efficace mais peu esthétique :

pour $k \leftarrow 1$ **à** n **faire**
  pour chaque arc (u,v) de G **faire**
    $\delta_k(s,v) \leftarrow \delta_{k-1}(s,u) + w(u,v)$
pour $k \leftarrow 1$ **à** n **faire**
  pour chaque arc (u,v) de G **faire**
    si $\delta_k(s,v) > \delta_{k-1}(s,u) + w(u,v)$ **alors**
      $\delta_k(s,v) \leftarrow \delta_{k-1}(s,u) + w(u,v)$
      si $\delta_k(s,v) < \delta(s,v)$ **alors**
        $\delta(s,v) \leftarrow \delta_k(s,v)$
        $k(v) \leftarrow k$

Ici, la première boucle fait uniquement les $O(|S|.|A|)$ initialisations nécessaires. On n'y gagne que si $|A| < |S|$, cas somme toute rarissime !

9. Pourquoi l'hypothèse « chaque sommet $v \in S$ est accessible à partir d'un sommet origine $s \in S$ » n'est-elle pas restrictive ?
   Pour contourner cette hypothèse il suffit de rajouter à G un sommet s et un arc de poids nul de s vers chacun des sommets de G pour se ramener dans les hypothèses de l'algorithme, cette construction étant moins chère que l'exécution de l'algorithme.

11 TD 11 : Plus courts chemins pour tout couple de sommets

Nous nous intéressons ici à la recherche des plus courts chemins entre tous les couples de sommets d'un graphe (typiquement on cherche à élaborer la table des distances entre tous les couples de villes d'un atlas routier). On dispose en entrée d'un graphe orienté $G = (S, A)$ et d'une fonction de pondération w. Nous supposons que le graphe G peut contenir des arcs de poids négatifs mais pas des circuits de poids strictement négatifs. On note n le nombre de sommets de G ($n = |S|$), et on note $\{1, 2, ..., n\}$ les n sommets de G.

Programmation dynamique naïve

Comme nous l'avons remarqué en cours, tout sous-chemin d'un plus court chemin est lui-même un plus court chemin et le problème des plus courts chemins vérifie bien la propriété de sous-structure optimale : nous pouvons donc essayer de le résoudre par programmation dynamique.

1. On note $d_{i,j}^{(m)}$ le poids minimal d'un chemin d'*au plus* m arcs du sommet i au sommet j. Définissez récursivement $d_{i,j}^{(m)}$ (la récursion portera ici sur le nombre d'arcs d'un plus court chemin).
   Pour $m = 0$ il existe un plus court chemin sans arc de i vers j si et seulement si $i = j$:

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ \infty & \text{sinon.} \end{cases}$$

   Pour $m \geq 1$, $d_{i,j}^{(m)}$ est la longueur du plus court chemin de i à j contenant au plus m arcs. Soit un tel plus court chemin contient exactement m arcs et il est obtenu par concaténation d'un plus court chemin d'au plus $m-1$ arcs de i à un sommet k et de l'arc de k à j, soit il n'en contient au plus que $m-1$ et sa longueur est égale à $d_{i,j}^{(m-1)}$. Par conséquent :

$$d_{i,j}^{(m)} = \min\left(d_{i,j}^{(m-1)}, \min_{1 \leq k \leq n}\left\{d_{i,k}^{(m-1)} + w_{k,j}\right\}\right) = \min_{1 \leq k \leq n}\left\{d_{i,k}^{(m-1)} + w_{k,j}\right\},$$

   la formule étant simplifiée grâce à la propriété : $w_{j,j} = 0$.

2. Construisez, au moyen de la formule de récurrence obtenue à la question précédente, un algorithme de calcul des longueurs des plus courts chemins pour tout couple de sommets, algorithme basé sur le paradigme de la programmation dynamique.
   On note $W = (w_{i,j})_{1 \leq i,j \leq n}$ la matrice des poids et $D^{(m)} = \left(d_{i,j}^{(m)}\right)_{1 \leq i,j \leq n}$ la matrice des poids des plus courts chemins contenant au plus m arcs. Le calcul de $D^{(m)}$ à partir de $D^{(m-1)}$ et de W se fait au moyen de l'algorithme ci-dessous :

```
EXTENSION-PLUS-COURTS-CHEMINS(D, W)
  n ← lignes(D)
  soit D' = (d'_{i,j})_{1≤i,j≤n} une matrice carrée de taille n
  pour i ← 1 à n faire
    pour j ← 1 à n faire
      d'_{i,j} ← +∞
      pour k ← 1 à n faire
        d'_{i,j} ← min(d'_{i,j}, d_{i,k} + w_{k,j})
  renvoyer D'
```

À partir de cet algorithme, la résolution du problème est triviale :

PLUS-COURTS-CHEMINS(W)
  $n \leftarrow$ *lignes*(W)
  $D^{(1)} \leftarrow W$
  pour $m \leftarrow 2$ **à** $n-1$ **faire**
    $D^{(m)} \leftarrow$ EXTENSION-PLUS-COURTS-CHEMINS($D^{(m-1)}, W$)
  renvoyer $D^{(n-1)}$

3. Quelle est la complexité de votre algorithme ?
   L'algorithme EXTENSION-PLUS-COURTS-CHEMINS *s'exécute en* $\Theta(n^3)$*, à cause des trois boucles imbriquées. Le coût total de résolution est donc en* $\Theta(n^4)$*.*
4. Exécutez votre algorithme sur le graphe de la figure 7.

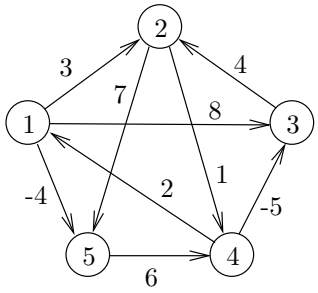

FIG. 7 – Exemple de graphe orienté.

La figure 8 présente un exemple d'exécution de cet algorithme.

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

FIG. 8 – Séquence des matrices calculées par PLUS-COURTS-CHEMINS.

Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un autre algorithme conçu suivant le principe de la programmation dynamique.

1. Ici la récursion n'a pas lieu sur le nombre d'arcs d'un plus court chemin, mais sur les sommets intermédiaires de ces chemins, un sommet intermédiaire étant un sommet autre que les extrémités du chemin. Ici, $d_{i,j}^{(k)}$ est la longueur du plus court chemin de i à j n'utilisant comme sommets intermédiaires que des sommets parmi $\{1, 2, ..., k\}$.
   Définissez $d_{i,j}^{(k)}$ de manière récursive.

De deux choses l'une, un plus court chemin de i à j n'ayant comme sommets intermédiaires que des sommets de $\{1,2,..,k\}$ contient ou ne contient pas le sommet k :

(a) Si le plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1,2,..,k\}$ a effectivement comme sommet intermédiaire k, alors p est de la forme $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ où p_1 (resp. p_2) est un plus court chemin de i à k (resp. de k à j) n'ayant comme sommets intermédiaires que des sommets de $\{1,2,...,k-1\}$.

(b) Si le plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1,2,..,k\}$ ne contient pas k, alors c'est un plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1,2,..,k-1\}$.

La structure explicitée ci-dessus nous donne directement une récursion définissant $d_{i,j}^{(k)}$:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{si } k = 0, \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) & \text{sinon.} \end{cases}$$

2. Proposez un algorithme de calcul de la longueur des plus courts chemins basé sur la récursion précédente et le paradigme de la programmation dynamique.

FLOYD-WARSHALL(W)
  $n \leftarrow$ *lignes*(W)
  $D^{(0)} \leftarrow W$
  pour $k \leftarrow 1$ **à** n **faire**
    pour $i \leftarrow 1$ **à** n **faire**
      pour $j \leftarrow 1$ **à** n **faire**
        $d_{i,j}^{(k)} \leftarrow \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$
  renvoyer $D^{(n)}$

3. Quelle est la complexité de votre algorithme ?
   On remarque aisément que l'algorithme de Floyd-Warshall est de complexité $\Theta(n^3)$.

4. Proposez un algorithme de construction effective des plus courts chemins à partir de l'algorithme précédent.
   Tout comme on a défini récursivement les longueurs des plus courts chemins, on peut définir récursivement les prédécesseurs dans les plus courts chemins : $\pi_{i,j}^{(k)}$ représente ici le prédécesseur du sommet j dans le plus court chemin de i à j n'utilisant comme sommets intermédiaires que des sommets parmi $\{1,2,...,k\}$. Pour $k = 0$, un plus court chemin ne possède aucun sommet intermédiaire, donc :

$$\pi_{i,j}^{(0)} = \begin{cases} \text{NIL} & \text{si } i = j \text{ ou } w_{i,j} = \infty, \\ i & \text{si } i \neq j \text{ et } w_{i,j} < \infty. \end{cases}$$

   Dans le cas général, si le plus court chemin est de la forme $i \rightsquigarrow k \rightsquigarrow j$ le prédécesseur de j est le même que celui du plus court chemin de k à j et n'utilisant comme sommets intermédiaires que des sommets parmi $\{1,2,...,k-1\}$. Autrement, on prend le même prédécesseur de j que celui qui se trouvait sur le plus court chemin de i à j et n'utilisant comme sommets intermédiaires que des sommets parmi $\{1,2,...,k-1\}$. Nous avons donc, dans tous les cas :

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}, \\ \pi_{k,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}. \end{cases}$$

   Ce qui nous donne l'algorithme suivant :

FLOYD-WARSHALL(W)
  $n \leftarrow$ *lignes*(W)
  $D^{(0)} \leftarrow W$

pour $i \leftarrow 1$ **à** n **faire**
  pour $j \leftarrow 1$ **à** n **faire**
    si $i = j$ ou $w_{i,j} = \infty$
      alors $\pi_{i,j}^{(0)} = \text{NIL}$
      sinon $\pi_{i,j}^{(0)} = i$
pour $k \leftarrow 1$ **à** n **faire**
  pour $i \leftarrow 1$ **à** n **faire**
    pour $j \leftarrow 1$ **à** n **faire**
      si $d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$
        alors
          $d_{i,j}^{(k)} \leftarrow d_{i,j}^{(k-1)}$
          $\pi_{i,j}^{(k)} \leftarrow \pi_{i,j}^{(k-1)}$
        sinon
          $d_{i,j}^{(k)} \leftarrow d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$
          $\pi_{i,j}^{(k)} \leftarrow \pi_{k,j}^{(k-1)}$
renvoyer $D^{(n)}$ et $\Pi^{(n)}$

5. Exécutez votre algorithme sur le graphe de la figure 7.
   La figure 9 présente le résultat de l'exécution de l'algorithme de Floyd-Warshall sur le graphe de la figure 7.

12 TD 12 : Heuristique de rangement

On a n objets. Le i^{e} objet est de taille s_i avec $0 < s_i < 1$. On souhaite ranger ces objets dans des boîtes en utilisant le minimum de boîtes possibles, sachant que chaque boîte est de taille unitaire : chaque boîte peut contenir un sous-ensemble quelconque des objets du moment que la somme des tailles de ces objets n'excède pas 1.

Ce problème est NP-complet. Pour le résoudre on utilise l'heuristique **Fischer-Price** : on prend les objets l'un après l'autre et un objet est placé dans la première boîte qui peut l'accueillir. On note $S = \sum_{i=1}^{n} s_i$.

1. Montrez que le nombre optimal de boîtes nécessaires est au moins égal à $\lceil S \rceil$.
   Les boîtes doivent contenir une taille totale S d'objets. Chaque boîte étant de taille 1, il faut au moins $\lceil S \rceil$ boîtes pour contenir les objets ($\lceil S \rceil$ est le plus petit entier supérieur ou égal à S).
2. Montrez que l'heuristique Fischer-Price laisse au plus une boîte remplie à moins de la moitié.
   On raisonne par l'absurde et on suppose qu'il existe au moins deux boîtes, notées A et B, qui sont remplies à moins de la moitié. Sans perte de généralités, on suppose que la boîte A a commencé à être remplie avant la boîte B. Soit x le premier objet ajouté à la boîte B. La boîte B étant à la fin de l'algorithme remplie à moins de la moitié et tous les poids étant positifs, x est de taille inférieure à $1/2$. A étant remplie à moins de la moitié à la fin de l'exécution de l'heuristique, et tous les poids étant positifs, A était remplie à moins de la moitié quand x a été rajouté à B. Donc l'heuristique, vu sa définition, a ajouté x à A au lieu de commencer à remplir une nouvelle boîte (B). Il y a contradiction et l'heuristique Fischer-Price laisse au plus une boîte remplie à moins de la moitié.
3. Démontrez que le nombre de boîtes utilisées par l'heuristique Fischer-Price n'est jamais strictement supérieur à $\lceil 2S \rceil$.
   Deux corrections différentes : une « brutale » par l'absurde, et une plus « subtile » et qui permet d'obtenir une borne plus fine :
   (a) *Nous venons de voir que l'heuristique Fischer-Price laisse au plus une boîte remplie à moins de la moitié. Donc toutes les boîtes sauf au maximum une sont remplies strictement à plus de la moitié. Raisonnons une fois de plus par l'absurde. Supposons que le nombre de boîtes utilisées, b, est strictement supérieur à $\lceil 2S \rceil$:*

$$b > \lceil 2S \rceil \quad \Leftrightarrow \quad b \geq 1 + \lceil 2S \rceil \quad \Leftrightarrow \quad b - 1 \geq \lceil 2S \rceil.$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

FIG. 9 – Séquence des matrice $D^{(k)}$ et $\Pi^{(k)}$ calculées par l'algorithme FLOYD-WARSHALL pour le graphe de la figure 7.

Or, d'après ce qui précède, (au moins) $b-1$ boîtes sont remplies strictement à plus de la moitié. Ces boîtes contiennent une taille totale t telle que :

$$t > \frac{1}{2}(b-1) \geq \frac{1}{2}\lceil 2S \rceil \geq \frac{1}{2}2S = S.$$

Donc $t > S$, ce qui est absurde puisque la taille totale des objets est S. Il y a donc contradiction et le nombre de boîtes utilisées par l'heuristique Fischer-Price n'est jamais strictement supérieur à $\lceil 2S \rceil$.

(b) On note de nouveau b le nombre de boîtes utilisées par l'heuristique. On note t_i la taille du contenu de la i^e boîte, c'est-à-dire la somme des tailles des objets contenus dans cette boîte. La taille totale des objets (S) est bien sûr égale à la somme des tailles contenues dans les boîtes : $S = \sum_{i=1}^{b} t_i$. Vu la question précédente, au maximum une boîte peut être remplie à moins de la moitié. Deux cas de figure se présentent :

i. Aucune boîte n'est moins qu'à moitié remplie. Par conséquent toutes les boîtes sont remplies strictement plus qu'à moitié et pour tout $i \in [1, b], t_i > \frac{1}{2}$. D'où :

$$S = \sum_{i=1}^{b} t_i > b\frac{1}{2} \quad \Rightarrow \quad b < 2S.$$

ii. Exactement une boîte est remplie à moins de la moitié. Supposons qu'il s'agit de la première : $t_1 \leq \frac{1}{2}$ et pour tout $i \in [2, b], t_i > \frac{1}{2}$. On suppose, sans perte de généralité, que la deuxième boîte est celle, parmi les $b-1$ boîtes remplies strictement plus qu'à moitié, dont le contenu est le plus petit : $t_2 = \min_{2 \leq i \leq b} t_i$. On remarque que l'on a : $t_1 + t_2 > 1$: sinon l'heuristique aurait utilisé les objets d'une des deux boîtes pour compléter l'autre au lieu d'utiliser une boîte de plus. On tire de ce qui précède :

$$S = \sum_{i=1}^{b} t_i = t_1 + \sum_{i=2}^{b} t_i \geq t_1 + (b-1)t_2 = (t_1 + t_2) + (b-2)t_2 > 1 + (b-2)\frac{1}{2} = \frac{1}{2}b.$$

On retrouve de nouveau : $b < 2S$.

Dans les deux cas on a montré l'inégalité : $b < 2S$, d'où l'on peut tirer : $b < \lceil 2S \rceil$ en majorant le membre droit par sa partie entière. On peut trouver une borne plus fine en remarquant que b est entier et que le plus grand entier inférieur ou égal à $2S$ est $\lfloor 2S \rfloor$, ce qui nous donne la borne : $b \leq \lfloor 2S \rfloor$.

4. Établir une borne égale à deux pour l'heuristique Fischer-Price.

Soit b le nombre de boîtes utilisées par l'heuristique Fischer-Price et soit b^* le nombre de boîtes d'une solution optimale. On doit donc montrer que $b \leq 2b^*$. Or on sait que $b \leq \lceil 2S \rceil$ et que $\lceil S \rceil \leq b^*$. Il nous suffit donc de montrer que $\lceil 2S \rceil \leq 2\lceil S \rceil$. Or, par définition : $S \leq \lceil S \rceil$. Or :

$$S \leq \lceil S \rceil \quad \Rightarrow \quad 2S \leq 2\lceil S \rceil \quad \Rightarrow \quad \lceil 2S \rceil \leq 2\lceil S \rceil.$$

Par conséquent :

$$b \leq \lceil 2S \rceil \leq 2\lceil S \rceil \leq 2b^*.$$

5. Donnez une implémentation de l'heuristique Fischer-Price.

FISCHER-PRICE(s)

```
b ← 0
Pour i ← 1 à n faire
  j ← 1
  tant que j ≤ b et s_i + taille[j] > 1 faire j ← j + 1
  si j ≤ b
    alors taille[j] ← taille[j] + s_i
    sinon b ← b + 1
          taille[b] ← s_i
```

6. Quelle est la complexité de votre heuristique ?
   La première boucle comporte n itérations, et la boucle ¡¡ tant que ¿¿ en compte au plus $\lceil 2S \rceil$ vu ce qui précède. D'où une complexité en $O(nS)$. (On peut aussi remarquer que $S \leq n$, car dans le pire cas il faut une boîte par objet, et obtenir une complexité en $O(n^2)$)

13 TP 1 : complexité et temps d'exécution

Recherche simultanée du minimum et du maximum

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de recherche simultanée du minimum et du maximum, et d'autre part l'algorithme optimal vu en TD (si besoin est, le corrigé du TD est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).
   Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char ** argv)
{
  int i;
  long int n,* tableau, min, max;
  long int debut, apres_premier, apres_second;

  /*Initialisations*/
  if (argc!=2){
    fprintf(stderr,"Le programme prend
       obligatoirement un argument\ n");
    return(-1);
  }
  n = atol(argv[1]);
  tableau = malloc(n*sizeof(long int));
  for(i=0; i<n; i++) tableau[i] = random();
  debut  = clock();


  /*Premier algorithme*/
  min = tableau[0];
  max = tableau[0];
  for(i=1; i<n; i++){
    if (tableau[i] > max) max=tableau[i];
    if (tableau[i] < min) min=tableau[i];
  }
  apres_premier = clock();

  /*Second algorithme*/
  if (tableau[1]>tableau[0]){
    min = tableau[0];
    max = tableau[1];
  }
  else{
    min = tableau[1];
    max = tableau[0];
  }

  for(i=2; i<n-1; i+=2){
    if (tableau[i+1]>tableau[i]){
      if (tableau[i+1]>max) max=tableau[i+1];
      if (tableau[i]  <min) min=tableau[i];
    }
    else{
      if (tableau[i]  >max) max=tableau[i];
      if (tableau[i+1]<min) min=tableau[i+1];
    }
  }
  if ((n%2)!=0){
      if (tableau[n] > max) max=tableau[n];
      if (tableau[n] < min) min=tableau[n];
  }
  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"Premier  algorithme: %.3lf\n
          Deuxieme algorithme: %.3lf\n",
          (1.0*(apres_premier-debut))
             /CLOCKS_PER_SEC,
          (1.0*(apres_second-apres_premier))
             /CLOCKS_PER_SEC);
  return 0;
}
```

   Le résultat dépend de l'ordinateur sur lequel ces fonctions sont testées, mais la version de complexité la plus faible n'est pas plus rapide. En effet, son code est plus compliqué et met en œuvre des opérations dont le coût est comparable à celui des fonctions de comparaisons (seules prises en compte lors du calcul de la complexité).
2. Récrivez le programme précédent en remplaçant les comparaisons au moyen des opérateurs $>$ et $<$ par des appels à une fonction `compare(a, b)` qui renvoie la valeur du test « `a` $>$ `b` ».

Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int compare(long int a, long int b)
{
  int i, j =0;
  for(i=0; i<1000; i++) j++;

  return (a > b);
}

int main(int argc, char ** argv)
{
  int i;
  long int n, * tableau, min, max;
  long int debut, apres_premier,
           apres_second;

  /*Initialisations*/
  if (argc!=2){
    fprintf(stderr,"Le programme prend
       obligatoirement un argument\n");
    return(-1);
  }
  n = atol(argv[1]);
  tableau = malloc(n*sizeof(long int));
  for(i=0; i<n; i++) tableau[i]=random();
  debut = clock();

  /*Premier algorithme*/
  min = tableau[0];
  max = tableau[0];
  for(i=1; i<n; i++){
    if (compare(tableau[i], max))
       max = tableau[i];
    if (compare(min, tableau[i]))
       min = tableau[i];
  }
  apres_premier = clock();

  /* Second algorithme*/
  if (compare(tableau[1], tableau[0])){
    min = tableau[0];
    max = tableau[1];
  }
  else{
    min = tableau[1];
    max = tableau[0];
  }

  for(i=2; i<n-1; i+=2){
    if(compare(tableau[i+1],tableau[i])){
      if (compare(tableau[i+1], max))
         max = tableau[i+1];
      if (compare(min, tableau[ i ]))
         min = tableau[i];
    }
    else{
      if (compare(tableau[ i ], max))
         max = tableau[i];
      if (compare(min, tableau[i+1]))
         min = tableau[i+1];
    }
  }
  if ((n%2)!=0){
      if (compare(tableau[n], max))
         max = tableau[n];
      if (compare(min, tableau[n]))
         min = tableau[n];
  }
  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);
  return 0;
}
```

Le fait de remplacer l'invocation directe de la comparaison (via les opérateurs $<$ et $>$) par un appel de fonction augmente considérablement le coût d'exécution de ces tests. Quand le coût des tests devient nettement plus important que celui des autres opérations, les programmes se comportent comme le prédit leur complexité ***en nombre de comparaisons****.*

Calcul de x^n

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de calcul de x^n et d'autre part la « méthode binaire » vue en cours (si besoin est, le cours est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).

Pour implémenter cet algorithme, vous avez besoin de pouvoir récupérer le i^{e} bit d'un entier. Pour ce faire vous pouvez utiliser les décallages : $n >> i$ est équivalent à une division par 2^i et amène donc le i^{e} bit en position de poids faible ; et l'opérateur « & » qui est le *et* logique bit à bit.

Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
  int i, rang;
  long int n, x, resultat;
  long int debut, apres_premier,
           apres_second;

  /* Initialisations*/
  if (argc!=3){
    fprintf(stderr,"Le programme prend
       obligatoirement deux arguments.\n");
  return(-1);
  }
  n = atol(argv[1]); x = atoi(argv[2]);

  /* Premier algorithme */
  debut = clock();
  resultat = x;
  for(i=2; i<=n; i++) resultat = x * resultat;
  fprintf(stderr,"resultat= %ld\n", resultat);
  apres_premier = clock();

  /* Deuxieme algorithme */
  resultat = x;
  /*Calcul du rang du bit de poids fort*/
  rang = sizeof(long int)*8-1;
  while(((n >> rang)==0)&&(rang>0)) rang--;

  for(rang = rang-1; rang >=0; rang--){
    if ((n >> rang)&1){
      resultat = resultat * resultat;
      resultat = x*resultat;
    }
    else{
      resultat = resultat * resultat;
    }
  }

  fprintf(stderr,"resultat = %ld\n",
     resultat);
  apres_second = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);
  return 0;
}
```

Les temps d'exécution reflète bien ici la complexité telle qu'elle a été calculée en cours.

2. Pour pouvoir comparer effectivement les deux algorithmes, à la question précédente, il vous a fallu utiliser des valeurs de la puissance, n, relativement élevées. De ce fait, les résultats des calculs étaient forcément faux et généraient des dépassement de capacités (*overflow*), à moins de n'essayer de ne calculer que des puissances de 0, 1 ou -1.

   Pour remédier à ce problème, récrivez votre programme en utilisant la librairie *gmp* de GNU qui permet de faire du calcul en précision arbitraire. Pour ce faire, vous avez uniquement besoin d'inclure le fichier de déclaration idoine (`gmp.h`), de lier votre application avec la librairie *ad-hoc* (`gmp`), et d'utiliser les fonctions et constructions appropriées, celles listées ci-dessous suffisant amplement :
   - `mpz_t q` : déclare l'entier `q` en précision arbitraire ;
   - `mpz_init(mpz_t q)` : effectue l'initialisation de l'entier `q`, cette initialisation est indispensable ;
   - `mpz_set_ui(mpz_t q, unsigned long x)` : affecte la valeur de `x` à `q` ;
   - `mpz_mul(mpz_t a, mpz_t b, mpz_t c)` : effectue la multiplication de `b` et de `c` et stocke le résultat dans `a`.
   - `mpz_out_str(FILE * stream, int base, mpz_t q)` : affiche sur le flot de sortie `stream` (typiquement `stderr` ou `stdout`) la valeur de l'entier `q` exprimée dans la base `base` qui doit être un entier compris entre 2 et 32.

   Comparez les temps d'exécution des deux algorithmes. Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <gmp.h>


int main(int argc, char ** argv)
{
  int i, rang;
  long int x, n;
  mpz_t resultat, mpg_x;
  long int debut, apres_premier,
           apres_second;

  /*Initialisations*/
  if (argc!=3){
    fprintf(stderr,"Le programme prend
       obligatoirement deux arguments.\n");
    return(-1);
  }
  n = atol(argv[1]);
  x = atoi(argv[2]);
  mpz_init(resultat);
  mpz_init(mpg_x);
  mpz_set_ui(mpg_x, x);

  /* Premier algorithme */
  debut  = clock();
  mpz_set_ui(resultat, x);
  for(i=2; i<=n; i++)
     mpz_mul(resultat, mpg_x, resultat);
  fprintf(stderr,"resultat = ");
  mpz_out_str(stderr, 10, resultat);
  fprintf(stderr,"\n");
  apres_premier = clock();

  /* Deuxieme algorithme */
  mpz_set_ui(resultat, x);
  /*Calcul rang du bit de poids fort*/
  rang = sizeof(long int)*8-1;
  while(((n >> rang)==0)&&(rang>0)) rang--;

  for(rang = rang-1; rang >=0; rang--){
    if ((n >> rang)&1){
      mpz_mul(resultat, resultat, resultat);
      mpz_mul(resultat, mpg_x, resultat);
    }
    else{
      mpz_mul(resultat, resultat, resultat);
    }
  }
  fprintf(stderr,"resultat = ");
  mpz_out_str(stderr, 10, resultat);
  fprintf(stderr,"\n");

  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);

return 0;
}
```

On retrouve les différences prédites par les études de complexité. Cependant, ici, le coût d'une multiplication n'est plus constant, il dépend de la taille des opérandes. De ce fait, les multiplications sont très rapidement plus coûteuses (car elles ont lieu sur les valeurs réelles, et donc sur des nombres plus grands), la différence entre les algorithmes apparaît plus rapidement, c'est-à-dire pour des valeurs de la puissance plus petite, ce qui donne l'illusion d'une différence d'efficacité moindre.

14 TP 2 : parcours d'arbres

Structure de données

Définissez une structure de données `C` qui vous permette de représenter des arbres binaires.

Parcours en profondeur

Implémentez un algorithme de parcours en profondeur des arbres binaires, et testez votre programme.

Parcours en largeur

Implémentez un algorithme de parcours en profondeur des arbres binaires, et testez votre programme. Vous utiliserez des files réalisées au moyen de tableaux dans un premier temps, et au moyen de listes chaînées

dans un deuxième temps.

Arbres n-aires

Étendez votre structure de données et vos programmes pour pouvoir traiter des arbres n-aires.

15 Devoir en temps libre

Algorithme glouton : l'art de rendre la monnaie

On considère le problème consistant à rendre n centimes (de franc ou d'euro) en monnaie, en utilisant le moins de pièces possible.

1. Décrivez un algorithme glouton permettant de rendre la monnaie en utilisant des pièces de cinquante, vingt, dix, cinq et un centime.
   On note n_{50} le nombre de pièces de 50 centimes à rendre, n_{20} celui de pièces de 20 centimes, etc.

   RENDRE-LA-MONNAIE(n)

   $reste \leftarrow n$
   $n_{50} \leftarrow \lfloor \frac{reste}{50} \rfloor$
   $reste \leftarrow reste - 50 \times n_{50}$
   $n_{20} \leftarrow \lfloor \frac{reste}{20} \rfloor$
   $reste \leftarrow reste - 20 \times n_{20}$
   $n_{10} \leftarrow \lfloor \frac{reste}{10} \rfloor$
   $reste \leftarrow reste - 10 \times n_{10}$
   $n_5 \leftarrow \lfloor \frac{reste}{5} \rfloor$
   $reste \leftarrow reste - 5 \times n_5$
   $n_1 \leftarrow reste$
   renvoyer $(n_{50}, n_{20}, n_{10}, n_5, n_1)$

2. Démontrez que votre algorithme aboutit à une solution optimale.
   On considère une instance de notre problème, c'est-à-dire une somme s à rendre, la solution de notre algorithme glouton pour cette instance, notée $(n_{50}, n_{20}, n_{10}, n_5, n_1)$, et une solution optimale notée $(m_{50}, m_{20}, m_{10}, m_5, m_1)$. On veut montrer que les deux solutions sont identiques.

   #### Première solution

   On raisonne par l'absurde et on suppose qu'elles sont différentes.
   On considère les pièces par valeur décroissante. Pour la pièce de plus grande valeur, v, pour laquelle les deux solutions diffèrent, notre solution gloutonne utilise, par construction, plus de pièces que la solution optimale (l'algorithme glouton utilise le maximum de pièces possible à chaque fois, et ici il reste la même somme à rendre pour les deux solutions). Nous avons cinq cas différents à considérer : $v = 1, 5, 10, 20$ et 50 :

   (a) *$v = 1$. Ce cas est trivialement impossible. Les deux solutions utilisent le même nombre de pièces de 50 centimes, de 20, de 10 et de 5. Chaque solution utilise alors $s - 50 \times n_{50} - 20 \times n_{20} - 10 \times n_{10} - 5 \times n_5$ pièces de 1 centime et elles ne diffèrent pas. Il y a contradiction.*

   (b) *$v = 5$. On a donc $n_{50} = m_{50}$, $n_{20} = m_{20}$, $n_{10} = m_{10}$ et $n_5 > m_5$. Comme $n_5 > m_5$ avec $5 \times n_5 + n_1 = 5 \times m_5 + m_1$, $m_1 \geq 5$ et on peut remplacer cinq pièces de 1 centime dans la solution optimale par une pièce de 5 ce qui nous donne une solution strictement meilleure, ce qui contredit l'optimalité.*

   (c) *$v = 10$. On a donc $n_{50} = m_{50}$, $n_{20} = m_{20}$, et $n_{10} > m_{10}$. Comme $n_{10} > m_{10}$ avec $10 \times n_{10} + 5 \times n_5 + n_1 = 10 \times m_{10} + 5 \times m_5 + m_1$, $5 \times m_5 + m_1 \geq 10$. Alors*

      i. *soit $m_5 \geq 2$ et on peut remplacer deux pièces de 5 centimes par une de 10 ;*

ii. *soit $m_5 \leq 1$, alors $m_1 \geq 5$ et on peut remplacer cinq pièces de 1 centime dans la solution optimale par une pièce de 5 ce qui nous donne une solution strictement meilleure, ce qui contredit l'optimalité.*

Dans tous les cas l'optimalité de la solution optimale est contredite.

(d) *$v = 20$. On a donc $n_{50} = m_{50}$ et $n_{20} > m_{20}$. Comme $n_{20} > m_{20}$ avec $20 \times n_{20} + 10 \times n_{10} + 5 \times n_5 + n_1 = 20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1$, $10 \times m_{10} + 5 \times m_5 + m_1 \geq 20$. On sait que $m_{10} \geq 1$, sinon on se trouverait dans le cas 2c que l'on a montré être impossible. Alors,*

i. *soit $m_{10} \geq 2$ et on peut remplacer deux pièces de 10 centimes par une de 20 ;*

ii. *soit $m_{10} = 1$; dans ce cas $5 \times m_5 + m_1 \geq 10$ et on se retrouve dans le cas 2c que l'on a montré être impossible.*

Dans tous les cas l'optimalité de la solution optimale est contredite.

(e) *$v = 50$. On a donc $n_{50} > m_{50}$. Comme $n_{50} > m_{50}$ avec $50 \times n_{50} + 20 \times n_{20} + 10 \times n_{10} + 5 \times n_5 + n_1 = 50 \times m_{50} + 20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1$, $20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1 \geq 50$. On sait que $m_{20} \geq 1$, sinon on se trouverait dans le cas 2c que l'on a montré être impossible. Alors,*

i. *soit $m_{20} = 1$ et alors $10 \times m_{10} + 5 \times m_5 + m_1 \geq 30$ ce qui ne peut être une solution optimale (voir l'étude du cas 2d) ;*

ii. *soit $m_{20} = 2$ et alors $10 \times m_{10} + 5 \times m_5 + m_1 \geq 10$ avec $m_{10} \geq 1$ (sinon $5 \times m_5 + m_1 \geq 10$ ce qui ne peut être optimal, comme le montre l'étude du cas 2b) et on peut remplacer les deux pièces de 20 et une de 10 par une pièce de 50 ;*

iii. *soit $m_{20} \geq 3$ et on peut remplacer trois pièces de vingt centimes par une pièce de cinquante et une de dix, ce qui contredit l'optimalité...*

Quelles que soient les configurations, on a abouti à une contradiction. Les deux solutions (la gloutonne et l'optimale) sont donc identiques.

Deuxième solution

Il est évident qu'une solution optimale contient au plus quatre pièces de 1 centime (cinq pièces de 1 centime pouvant être remplacées par une de 5 centimes), au plus une pièce de 5, au plus une pièce de 10, et au plus deux pièces de 20 (trois pièces de 20 pouvant être remplacées par une de 50 et une de 10). De plus, une solution optimale ne contient pas en même temps une pièce de 10 centimes et deux de 20. On a donc :

$$m_1 \leq 4, m_5 \leq 1, m_{10} \leq 1, m_{20} \leq 2 \text{ et } m_{10} + m_{20} \leq 2.$$

Par conséquent, dans une solution optimale, les pièces de 1, 5, 10 et 20 centimes représentent un montant d'au plus 49 centimes ($m_1 + 5 \times m_5 + 10 \times m_{10} + 20 \times m_{20} \leq 49$) et il n'y a pas de lattitude quant au choix du nombre de pièces de 50 centimes : $m_{50} = \lfloor \frac{n}{50} \rfloor = n_{50}$.

De ce qui précède on déduit que :

(a) *Le montant représenté par les pièces de 1, 5 et 10 centimes est inférieur à 19 centimes, et donc que $m_{20} = \lfloor \frac{n - 50 \times n_{50}}{20} \rfloor = n_{20}$.*

(b) *Le montant représenté par les pièces de 1 et 5 centimes est inférieur à 9 centimes, et donc que $m_{10} = \lfloor \frac{n - 50 \times n_{50} - 20 \times n_{20}}{10} \rfloor = n_{10}$.*

(c) *Le montant représenté par les pièces de 1 centime est inférieur à 4 centimes, et donc que $m_5 = \lfloor \frac{n - 50 \times n_{50} - 20 \times n_{20} - 10 \times n_{10}}{5} \rfloor = n_5$.*

(d) *Comme $m_{50} = n_{50}$, $m_{20} = n_{20}$, $m_{10} = n_{10}$ et $m_5 = n_5$, on a également $m_1 = n_1$.*

3. On suppose que les pièces disponibles ont pour valeur p^0, p^1, ..., p^k pour deux entiers $p > 1$ et $k \geq 1$ donnés. Montrez que l'algorithme glouton aboutit toujours à une solution optimale sur un tel jeu de pièces.

On considère une instance de notre problème, c'est-à-dire une somme s à rendre, la solution de notre algorithme glouton pour cette instance, notée $(n_k, n_{k-1}, ..., n_1, n_0)$ (où n_j est le nombre d'exemplaires

utilisés de la pièce de valeur p^j), et une solution optimale notée $(m_k, m_{k-1}, ..., m_1, m_0)$. On veut montrer que les deux solutions sont identiques. On raisonne par l'absurde et on suppose qu'elles sont différentes.

Par construction de l'algorithme glouton, $n_k \geq m_k$. L'algorithme glouton utilisant le maximum de pièces possible à chaque fois, il existe un entier j appartenant à l'intervalle $[0, k]$ tel que pour tout i, $j+1 \leq i \leq k$, $n_i = m_i$ et $n_j > m_j$. $s = \sum_{i=0}^{k} n_i \times p^i = \sum_{i=0}^{k} m_i \times p^i$ et $\forall i \in [j+1, k], n_i = m_i$. Donc,

$$\sum_{i=0}^{j} n_i \times p^i = \sum_{i=0}^{j} m_i \times p^i \quad \Leftrightarrow \quad \sum_{i=0}^{j-1} (m_i - n_i) \times p^i = (n_j - m_j) \times p^j.$$

En remarquant que tous les termes n_i sont positifs, puis que $n_j > m_j$ implique $n_j - m_j \geq 1$ on obtient :

$$(n_j - m_j) \times p^j \leq \sum_{i=1}^{j-1} m_i \times p^i \quad \Rightarrow \quad p^j \leq \sum_{i=0}^{j-1} m_i \times p^i.$$

Montrons alors qu'il existe une pièce (d'une valeur autre que p^k) utilisée au moins p fois dans la solution optimale, ce qui contredit l'optimalité : on peut remplacer p exemplaires de cette pièce par une pièce de valeur strictement supérieure. On raisonne par l'absurde et on suppose que toutes les pièces de valeurs strictement inférieures à p^k sont prises au plus $p-1$ fois : $\forall i \in [0, k-1], m_i \leq p-1$. On a alors :

$$p^j \leq \sum_{i=0}^{j-1} m_i \times p^i \leq \sum_{i=0}^{j-1} (p-1) \times p^i = \sum_{i=0}^{j-1} p^{i+1} - \sum_{i=0}^{j-1} p^i = \sum_{i=1}^{j} p^i - \sum_{i=0}^{j-1} p^i = p^j - 1,$$

ce qui est absurde.

4. Donnez un ensemble de valeurs de pièces pour lesquelles l'algorithme glouton ne donne pas une solution optimale.

   Si les pièces à notre disposition ont pour valeurs respectives : 7, 5 et 1 centimes, l'algorithme glouton utilisera une pièce de 7 centimes et quatre de 1 pour composer la somme de 11 centimes, quand la solution optimale utilise deux pièces de 5 et une de 1.

Diviser pour régner : intervalle de plus grande somme

Nous avons un tableau A de n entiers relatifs. Nous recherchons un sous-tableau de A dont la somme des éléments soit maximale. Autrement dit, nous recherchons un couple d'entiers i et j, $1 \leq i \leq j \leq n$ tel que $\sum_{k=i}^{j} A[k]$ soit maximale. La figure 10 montre un exemple de tableau pour lequel les valeurs cherchées sont $i = 4$ et $j = 5$.

2	5	-8	6	5	-9	3	4

FIG. 10 – Tableau dont l'intervalle de plus grande somme est défini par $i = 4$ et $j = 5$.

1. Proposez un algorithme naïf.

```
PGS-NAÏF(A)
  d ← 1
  f ← 1
  max ← A[1]
  pour i ← 1 à n faire
  s ← 0
    pour j ← i à n faire
      s ← s + A[j]
```

si $s > max$ **alors**
$d \leftarrow i$
$f \leftarrow j$
$max \leftarrow s$

2. Quelle est sa complexité ?
La première boucle comporte n itérations, pour i allant de 1 à n. La deuxième boucle comporte $n-i+1$ itérations. Toutes les autres opérations sont de coût constant. Le coût global est donc :

$$\sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

D'où une complexité en $\Theta(n^2)$.

3. Proposez un algorithme « diviser pour régner » découpant le tableau en deux moitiés.
Trois cas sont possibles : l'intervalle de plus grande somme est inclus dans la première moitié du tableau ; l'intervalle de plus grande somme est inclus dans la deuxième moitié du tableau ; l'intervalle de plus grande somme empiète sur les deux moitiés du tableau. Les deux premiers cas sont réglés par un appel récursif sur chacune des deux moitiés. Pour régler le troisième cas, on calcule l'intervalle de plus grande somme dont la limite droite est le dernier élément de la première moitié et l'intervalle de plus grande somme dont la limite gauche est le premier élément de la deuxième moitié. Ensemble, ces deux intervalles nous fournissent l'intervalle de plus grande somme empiétant sur les deux moitiés. On peut envisager deux méthodes pour réaliser ce calcul.

Première méthode

Ici tout est recalculé à chaque fois, sans tenir compte d'éventuels résultats précédents.

PGS(A, d, f)
si $d = f$ **renvoyer** (d, d, $A[d]$)
$milieu \leftarrow \lfloor \frac{f-d}{2} \rfloor$
$(début_1, fin_1, somme_1) \leftarrow$ PGS(A, d, $milieu$)
$(début_2, fin_2, somme_2) \leftarrow$ PGS(A, $milieu+1$, f)
$DemiSommeGauche \leftarrow A[milieu]$
$LimiteGauche \leftarrow milieu$
$s \leftarrow A[milieu]$
pour $i \leftarrow milieu - 1$ **à** d **faire**
$s \leftarrow s + A[i]$
si $s > DemiSommeGauche$ **alors** $DemiSommeGauche \leftarrow s$
$LimiteGauche \leftarrow i$
$DemiSommeDroite \leftarrow A[milieu+1]$
$LimiteDroite \leftarrow milieu + 1$
$s \leftarrow A[milieu+1]$
pour $i \leftarrow milieu + 2$ **à** f **faire**
$s \leftarrow s + A[i]$
si $s > DemiSommeDroite$ **alors** $DemiSommeDroite \leftarrow s$
$LimiteDroite \leftarrow i$
$(début_3, fin_3, somme_3) \leftarrow (LimiteGauche, LimiteDroite, DemiSommeGauche + DemiSommeDroite)$
si $somme_3 \geq somme_2$ et $somme_3 \geq somme_1$
alors renvoyer ($début_3$, fin_3, $somme_3$)
sinon si $somme_2 \geq somme_3$ et $somme_2 \geq somme_1$
alors renvoyer ($début_2$, fin_2, $somme_2$)
sinon renvoyer ($début_1$, fin_1, $somme_1$)

Deuxième méthode

On évite ici de parcourir entièrement chaque moitié de tableau pour calcule l'intervalle de plus grande somme dont la limite droite est le dernier élément de la première moitié et l'intervalle de plus grande somme dont la limite gauche est le premier élément de la deuxième moitié. Pour ce faire on calcule à chaque fois trois valeurs : l'intervalle de plus grande somme dont la limite droite est le dernier élément du tableau, l'intervalle de plus grande somme dont la limite gauche est le premier élément du tableau, la somme de tous les éléments du tableau.

PGS(A, d, f)
  si $d = f$ **renvoyer** (d, $A[d]$, d, $A[d]$, $A[d]$, d, d, $A[d]$)
  $milieu \leftarrow \lfloor \frac{f-d}{2} \rfloor$
  $(début_1, somme\_début_1, fin_1, somme\_fin_1, somme_1, d_1, f_1, s_1) \leftarrow$ PGS(A, d, $milieu$)
  $(début_2, somme\_début_2, fin_2, somme\_fin_2, somme_2, d_2, f_2, s_2) \leftarrow$ PGS(A, $milieu + 1$, f)
  si $somme\_début_1 \geq somme_1 + somme\_début_2$
    alors
      $somme\_début \leftarrow somme\_début_1$
      $début \leftarrow début_1$
    sinon
      $somme\_début \leftarrow somme_1 + somme\_début_2$
      $début \leftarrow début_2$
  si $somme\_fin_2 \geq somme_2 + somme\_fin_1$
    alors
      $somme\_fin \leftarrow somme\_fin_2$
      $fin \leftarrow fin_2$
    sinon
      $somme\_fin \leftarrow somme_2 + somme\_fin_1$
      $fin \leftarrow fin_1$
  $somme \leftarrow somme_1 + somme_2$
  $d_3 \leftarrow fin_1$
  $f_3 \leftarrow début_2$
  $s_3 \leftarrow somme\_fin_1 + somme\_début_2$
  si $s_3 \geq s_2$ et $s_3 \geq s_1$
    alors renvoyer ($début, somme\_début, fin, somme\_fin, somme, d_3, f_3, s_3$)
    sinon si $s_2 \geq s_3$ et $s_2 \geq s_1$
      alors renvoyer ($début, somme\_début, fin, somme\_fin, somme, d_2, f_2, s_2$)
      sinon renvoyer ($début, somme\_début, fin, somme\_fin, somme, d_1, f_1, s_1$)

L'appel initial est alors de la forme :

$$(début, somme\_début, fin, somme\_fin, somme, i, j, somme\_maximale) = \text{PGS}(A, 1, n).$$

4. Quelle est sa complexité ?

Première méthode

Un appel de PGS *sur un tableau de taille n génère deux appels récursifs sur des tableaux de taille $n/2$. La division a un coût constant (calcul du milieu). Par contre, la combinaison des résultats nécessite la recherche de l'intervalle de plus grande somme se terminant en l'extrémité droite (resp. gauche) du premier (resp. deuxième) demi-tableau. Cette double recherche a un coût en $\Theta(n)$ (parcours de tout le tableau). D'où l'équation définissant la complexité :*

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Nous avons donc ici : $a = 2$, $b = 2$ et $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$. Nous sommes donc dans le cas 2 du théorème et donc :

$$T(n) = \Theta(n \log n).$$

Deuxième méthode

Un appel de PGS *sur un tableau de taille n génère deux appels récursifs sur des tableaux de taille $n/2$. La division a un coût constant (calcul du milieu) tout comme la combinaison des résultats. D'où l'équation définissant la complexité :*

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

Nous avons donc ici : $a = 2$, $b = 2$ et $f(n) = \Theta(1) = O(n^{1-1}) = O(n^{(\log_2 2)-1})$. Nous sommes donc dans le cas 1 du théorème et donc :

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Programmation dynamique : répartition des espaces en typographie

On considère le problème de la composition équilibrée d'un paragraphe dans un traitement de texte. Le texte d'entrée est une séquence de n mots de longueurs l_1, l_2, ..., l_n mesurées en nombre de caractères. On souhaite composer ce paragraphe de manière équilibrée sur un certain nombre de lignes contenant chacune exactement M caractères. Chaque ligne comportera un certain nombre de mots, les espaces nécessaires à séparer les mots les uns des autres (une espace[2] entre deux mots consécutifs) et des *caractères d'espacement supplémentaires* complétant la ligne pour qu'elle contienne exactement M caractères. La figure 11 présente un exemple de ligne contenant trois mots, deux espaces nécessaires à séparer ces trois mots, et six caractères d'espacement supplémentaires. Si une ligne donnée contient les mots de i à j, où $i \leq j$, et étant donné

U	n		e	x	e	m	p	l	e		t	r	i	v	i	a	l						

FIG. 11 – Une ligne de 24 caractères contenant 3 mots et 6 caractères d'espacement supplémentaires.

que nous avons besoin d'une unique espace pour séparer deux mots consécutifs, le nombre de caractères d'espacement supplémentaires c nécessaires pour compléter la ligne est égal au nombre de caractères de la ligne (M), moins le nombre de caractères nécessaires pour écrire les mots ($\sum_{k=i}^{j} l_k$), moins le nombre de caractères nécessaires pour séparer les mots ($j - i$), autrement dit : $c = M - \sum_{k=i}^{j} l_k - (j - i)$. Le nombre c de caractères d'espacement supplémentaires doit bien évidemment être positif ou nul.

Notre critère « d'équilibre » est le suivant : on souhaite minimiser la somme, *sur toutes les lignes hormis la dernière*, des cubes des nombres de caractères d'espacement supplémentaires.

Travail minimum requis

Donnez un algorithme de programmation dynamique permettant de composer de manière équilibrée un paragraphe de n mots de longueurs l_1, ..., l_n données, et analysez la complexité de votre algorithme.

Travail idéal

Vous pourrez, *mais ce n'est pas obligatoire*, établir une relation de récurrence définissant la composition optimale, proposer un algorithme récursif implémentant cette récurrence et montrer que sa complexité est médiocre, proposer un algorithme de programmation dynamique et analyser sa complexité, proposer un algorithme par recensement, et finalement proposer un contre-exemple à l'algorithme glouton naïf.

[2]En typographie, « espace » est un mot féminin, comme vous le confirmera le premier dictionnaire venu.

Indication pour l'établissement d'une récurrence

Comme l'on cherche ici un algorithme suivant le paradigme de la programmation dynamique, il est raisonnable de définir la composition optimale par une formule de récurrence. Pour ce faire, vous pourrez remarquer que dans un paragraphe de m lignes composé optimalement, les $(m-1)$ *dernières* lignes sont composées optimalement.

Récurrence. *Considérons une solution optimale s'étendant sur m lignes. La composition des $m-1$ dernières lignes est optimale : sinon nous pourrions combiner la composition de la première ligne et d'une composition optimale des $m-1$ dernières pour obtenir une composition strictement meilleure de l'ensemble, ce qui contredirait l'hypothèse d'optimalité.*

Notons $c(i)$ le coût de la composition des mots à partir du i^e (inclus). La première ligne de cette composition commence donc au mot i et finit à un certain mot j (inclus). Nous avons donc :

$$c(i) = \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c(j+1).$$

Pour obtenir la valeur optimale de $c(i)$, nous minimisons la formule précédente sur toutes les valeurs possibles de j, sachant que j vaut au minimum i et que le nombre de caractères supplémentaires doit être positif ou nul :

$$c(i) = \begin{cases} 0 \text{ si } n - i + \sum_{k=i}^{n} l_k \leq M, \\ \min\left\{ \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c(j+1) \,\middle|\, i \leq j \leq n, j - i + \sum_{k=i}^{j} l_k \leq M \right\} \text{ sinon.} \end{cases}$$

Algorithme récursif.

COMPOSITION-RÉCURSIVE(l, i, n, M)
  si $n - i + \sum_{k=i}^{n} l_k \leq M$ **alors renvoyer** 0
  $c \leftarrow +\infty$
  pour $j \leftarrow i$ **à** n **faire**
    si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$
      et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RÉCURSIVE}(l,\, j+1,\, n,\, M) < c\right)$
        alors $c \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RÉCURSIVE}(l,\, j+1,\, n,\, M)$
  renvoyer c

*On pourrait optimiser légèrement cet algorithme en remplaçant la boucle **pour** par une boucle **tant que**.*

Complexité de l'algorithme récursif. *La complexité de cet algorithme est définie par la récurrence :*

$$T(i,n) = 1 + \sum_{j=i+1}^{k} (1 + T(j,n)),$$

où k est le plus grand entier supérieur à i tel que $j - i + \sum_{k=i}^{j} l_k \leq M$. La présence de la valeur de k dans la récursion nous empêche de faire une estimation fine. Nous allons poser l'hypothèse, réaliste, que les lignes sont suffisamment longues et les mots suffisamment courts pour que l'on puisse toujours écrire au moins deux mots par ligne. Sous cette hypothèse, k est toujours supérieur ou égal à $i+2$ (quand $i+2$ est inférieur à n) et :

$$T(i,n) \geq 3 + T(i+1,n) + T(i+2,n)$$

d'où, par substitution de $T(i+1,n)$ par la même minoration,

$$T(i,n) \geq 6 + 2T(i+2,n) + T(i+3,n) \geq 2T(i+2,n).$$

Par conséquent,

$$T(i,n) = \Omega(2^{\frac{n-i}{2}}) \text{ et } T(1,n) = \Omega(2^{\frac{n}{2}}).$$

L'algorithme récursif est donc de complexité au moins exponentielle et il nous faut trouver une solution moins mauvaise.

Solution par programmation dynamique.

COMPOSITION-PROGDYN(l, n, M)
  $c[n+1] \leftarrow 0$
  pour $i \leftarrow n$ **à** 1 **faire**
    si $n - i + \sum_{k=i}^{n} l_k \leq M$
      alors $c[i] \leftarrow 0$
      sinon
        $c[i] \leftarrow +\infty$
        pour $j \leftarrow i$ **à** n **faire**
          si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$ et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c[j+1] < c[i]\right)$
            alors $c[i] \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c[j+1]$

Complexité de la solution par programmation dynamique. *La complexité de cet algorithme est immédiate :*

$$T(n) = \sum_{i=n}^{1} \sum_{j=i}^{n} 1 = \sum_{i=n}^{1} (n - i + 1) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

en posant $k = n - i + 1$. D'où : $T(n) = \Theta(n^2)$.

Solution par recensement.

COMPOSITION-PARRECENSEMENT(l, n, M)
  pour $i \leftarrow 1$ **à** n **faire** $c[i] \leftarrow +\infty$
  $c[n+1] \leftarrow 0$
  renvoyer COMPOSITION-RECENSEMENT(c, l, 1, n, M)

COMPOSITION-RECENSEMENT(c, l, i, n, M)
  si $c[i] < +\infty$ **alors renvoyer** $c[i]$
  si $n - i + \sum_{k=i}^{n} l_k \leq M$ **alors renvoyer** 0
  pour $j \leftarrow i$ **à** n **faire**
    si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$
      et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RECENSEMENT}(c, l, j+1, n, M) < c[i]\right)$
        alors $c[i] \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RECENSEMENT}(c, l, j+1, n, M)$
  renvoyer $c[i]$

Contre-exemple à l'algorithme glouton. *La solution naïve est gloutonne : on place le maximum de mots sur la première ligne, puis on appelle récursivement l'algorithme sur les mots restants. Cet algorithme ne fournit pas la solution optimale, comme le montre les figures 12 et 13 qui présente deux compositions différentes du même ensemble de mots sur des lignes de 24 caractères. La figure 12 présente la solution de l'algorithme glouton : la première ligne ne contient aucune espace supplémentaire mais la deuxième en contient sept, d'où un coût de $0^3 + 7^3 = 343$. La figure 13 présente une composition optimale : la première ligne contient trois espaces supplémentaires et la deuxième quatre, d'où un coût de $3^3 + 4^3 = 27 + 64 = 91$, ce qui montre la non optimalité de l'algorithme glouton.*

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n		n	e
p	e	u	t		p	a	s		ê	t	r	e		b	o	n							
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t							

FIG. 12 – Solution de l'algorithme glouton.

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n			
n	e		p	e	u	t		p	a	s		ê	t	r	e		b	o	n				
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t							

FIG. 13 – Solution optimale.