Correction des TD de HMIN329

Année 2006

Version 2.1

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

Rodolphe Giroudeau
161, rue Ada
34392 MONTPELLIER Cedex 5
TEL : 04-67-41-85-40
Mail : rgirou@lirmm.fr

$f(n)$	10	20	30	60
$\log n$	$2,30 \times 10^{-6}$	$3,00 \times 10^{-6}$	$3,40 \times 10^{-6}$	$4,09 \times 10^{-6}$
n	$\times 10^{-5}$	$2,00 \times 10^{-5}$	3×10^{-5}	6×10^{-5}
$n \log n$	$2,30 \times 10^{-5}$	$5,99 \times 10^{-5}$	$1,02 \times 10^{-4}$	$2,46 \times 10^{-4}$
n^2	$\times 10^{-4}$	$4,00 \times 10^{-4}$	9×10^{-4}	$3,6 \times 10^{-3}$
n^3	$\times 10^{-3}$	$8,00 \times 10^{-3}$	$2,7 \times 10^{-2}$	$2,16 \times 10^{-1}$
2^n	$1,02 \times 10^{-3}$	$1,05$	$1,07 \times 10^{3}$	$1,15 \times 10^{12}$

TABLE 1 – Tableau

Université de Montpellier 2020/2021
HMIN329 – Complexité et algorithmes Durée: 4.5h

Complexité
TD – Séance n 2

Exercice 1 – Complexité

Soit un ordinateur pour lequel toute instruction possède une durée de 10^{-6} secondes. On exécuté un algorithme qui utilise, pour une donnée de taille n, $f(n)$instructions. $f(n)$ étant l'une des fonctions ($\log n$, $n \log n$, n, n^2, n^3, 2^n. Remplir un tableau qui donne en fonction de la taille $n = 10, 20, 300$ et 60, et de la fonction $f(n)$, la durée d'exécution de l'algorithme.

Correction exercice 1

Le tableau 1 utilise le logarithme népérien.

Notons que $1,15 \times 10^{12}$s vaut 36558 ans.

Fin correction exercice 1

Exercice 2 – Complexité

Soit A et B, deux algorithmes pour résoudre un même problème. La complexité de A est en $\theta(n^2)$ et son exécution avec $n = 100$ dure $1s$. La complexité de B est en $\theta(n \log n)$ et son exécution avec $n = 100$ dure $10s$.

1. L'application prévoit une valeur de n égale à 1000. Quel algorithme faut-il à priori choisir ?
2. Même question si n égale à 10000.

Correction exercice 2

1. Nous avons $T_A(1000) = 100s$ et $T_A(10000) = 10000s$. $T_A(n_1) = \alpha n_1^2$ et $T_A(n_2) = \alpha n_2^2$. Ainsi, nous avons $T_A(n_1) = \frac{n_1^2}{n_2^2} T_A(n_2)$. Alors $T_A(10000) = 10^8/10^4.1s = 10000s$.
2. B est en $\theta(n \log n)$, et donc $T_b(n) = \beta n \log n$. En utilisant le même raisonnement, nous avons $T_b(n_1) = \frac{n_1 \log n_1}{n_2 \log n_2} T_B(n_2)$. Ainsi $T_B(1000) = (10^3 \log(10^3))/(10^2 \log(10^2)).10s = 150s$, et pour l'autre $T_B(10000) = (10^4 \log(10^4))/(10^2 \log(10^2)).10s = 2000s$.
3. En conclusion, pour a (resp. b) on choisira $\theta(n^2)$ (resp. $\theta(n \log n)$).

Fin correction exercice 2

Exercice 3 – Complexité

Soit un algorithme en $\theta(n^2)$. Un ordinateur X permet de traiter en 1 mn des problèmes de taille n_0. Quelle est la taille des problèmes que l'on pourra traiter en 1 mn avec un ordinateur 100 fois plus rapide ? Même chose pour $\theta(2^n)$.

Correction exercice 3

1. $T(n_0) = 1mn = \alpha n_0^2$ et $T(n_1) = 100mn = \alpha n_1^2$. Nous avons $n_1^2 = 100 n_0^2$ alors $n_1 = 10 n_0$. On a $\alpha n_0^2 = 1minn$ et $\alpha n_1^2 = 100min$. Nous avons de manière classique que

$$\begin{aligned} \frac{T_A(n_1)}{n_1^2} &= \frac{T_A(n_0)}{n_0^2} \\ T_A(n_0) \times n_1^2 &= T_A(n_1) \times n_0^2 \\ T_A(n_0) \times n_1^2 &= 100 \times T_A(n_0) \times n_0^2 \\ n_1^2 &= 100 n_0^2 \end{aligned}$$

2. l'autre c'est $n_1 = 0 + \log 100 = n_0 + 6$

Fin correction exercice 3

Exercice 4 – Complexité

Lesquelles des assertions suivantes sont vraies ? Prouvez vos réponses.

1. $1000 \in O(1)$;
2. $n^2 \in O(n^3)$;
3. $n^3 \in O(n^2)$;
4. $2^{n+1} \in O(2^n)$;
5. $(n+1)^2 \in O(n^2)$;
6. $n^3 + 3n^2 + n + 1996 \in O(n^3)$;
7. $n^2 * n^3 \in O(n^3)$;
8. $2^{2n} \in O(2^n)$;
9. $\frac{1}{2}n^2 - 3n \in \theta(n^2)$.

Correction exercice 4

Dans un premier temps, faisons un petit rappel :

— $f = O(g)$ alors $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}$ tels que $\forall n \geq n_0, f(n) \leq cg(n)$
— $f = \theta(g)$ alors $\exists (c,d) \in \mathbb{R}^2, \exists n_0 \in \mathbb{N}$ tels que $\forall n \geq n_0, dg(n) \leq f(n) \leq cg(n)$. Par exemple, soit $g(n) = n^3$ et $f(n) = 10n^3 + 5n^2$ et $g(n) \leq f(n) \leq 11g(n)$ avec $n_0 = 100$

1. vrai, $c = 1000, n_0 = 1$
2. vrai $c = 1; n_0 = 1$ car $n^2 \leq n^3, \forall n \geq 1$
3. faux. En effet, $\forall c, \forall n_0, \exists n, n_0$ tel que $n^3 \geq cn^2$ $(n = \max(c, n_0))$
4. Vrai $2^{(n+1)} \in O(2^n), c = 2, n_0 = 1$
5. vrai $c = 2, n_0 = 4$ $(n+1)^2 \leq 2n^2$
6. vrai $n^3 + 3n^2 + n + 1996 \leq 2001n^3$. $c = 20001$ et $n_0 = 1$ ou $c = 2$ et $n_0 = 2000$.
7. faux $f(n) = n^2 \times n^3 = n^5$ et $g(n) = n^3$ $\forall c, f(n) > c.g(n)$ pour $n \geq c$.
8. faux $f(n) = 2^{2n}$ et $g(n) = 2^n$. $\forall c, f(n) > c.g(n), n \geq c$.
   Pour trouver c, nous voulons que $2^{2n} < c2^n$ et donc $n \geq \log_2 c$
9. vrai $n_0 = 100, \alpha = 1/4, c = 1$. nous avons $0.25n^2 \leq 0.5n^2 - 3n \leq n^2$. Si la limite quand n $\leftarrow \infty$ de $f(n)/g(n)$ existe, alors on peut comparer les fonctions : Si $\lim_{n \leftarrow \infty} f(n)/g(n) = a$. trois cas se présentent à nous :
   — si $a = 0$ alors $f \in O(g)$ et $f \notin \theta(g)$
   — si $a \in \mathbb{R}^{*+}$ alors $f \in \theta(g)$
   — si $a = \infty$ alors $g \in O(f)$ et $g \notin O(f)$

Fin correction exercice 4

Exercice 5 – Complexité

Comparer deux à deux les fonctions suivantes :

1. $f_1(n) = n^2 + 100$
2. Soit f_2 défini de la manière suivante :

$$\begin{aligned} f_2(n) &= n \text{ si } n \text{ est impair} \\ &= n^3 \text{ si } n \text{ est pair} \end{aligned}$$

3. Soit f_3 défini de la manière suivante :

$$\begin{aligned} f_3(n) &= n \text{ si } n < 100 \\ &= n^3 \text{ si } n \geq 100 \end{aligned}$$

Correction exercice 5

1. f_1 et f_2 ne sont pas comparables
2. f_1 et f_3 sont comparables et $f_1 \in O(f_3)$
3. f_2 et f_3 sont comparables et $f_2(n) \leq f_3(n)$ à partir d'un certain rang et donc $f_2 \in O(f_3)$.

Fin correction exercice 5

Exercice 6 – Complexité

1. Soient f_1 et f_2 deux fonctions telles que $f_1 = \theta(g)$ et $f_2 = \theta(g)$ et $f_1 \geq f_2$. A-t'on $f_1 - f_2 = \theta(g)$?

Correction exercice 6

1. Non pas toujours, $f_1(n) = n^3 + n^2, f_1 \in \theta(n^3)$ et $f_2(n) = n^3 + n, f_2 \in \theta(n^3)$. Alors $(f_1 - f_2)(n) = n^2 - n \in \theta(n^2)$ et $(f_1 - f_2)(n) = n^2 - n \in O(g)$, mais $(f_1 - f_2) \notin \theta(n^3)$.

Fin correction exercice 6

Exercice 7 – Complexité

1. Démontrer que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ et $\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$.
2. Quelle est la complexité de la boucle suivante :

```
for i:=1 to (n-1) do
    for j:=(i+1) to n do
       for k:=1 to j do
        {instruction}
```

Correction exercice 7

1. Pour la première somme c'est classique. Pour la seconde somme :

$$\begin{aligned} n^3 &= ((n-1)+1)^3 = (n-1)^3 + 3(n-1)^2 + 3(n-1) + 1 \\ (n-1)^3 &= ((n-2)+1)^3 = (n-2)^3 + 3(n-2)^2 + 3(n-2) + 1 \\ &\vdots \\ i^3 &= ((i-1)+1)^3 = (i-1)^3 + 3(i-2)^2 + 3(i-1) + 1 \\ &\vdots \\ 2^3 = (1+1)^3 &= (1^3 + 3 + 3 + 1) \\ 1^3 = (0+1)^3 &= (0^3 + 3*0^2 + 3*0^1 + 1^3) \end{aligned}$$

En faisant la somme termes à termes nous obtenons

$$\begin{aligned} n^3 &= 3\sum_{i=1}^{n}(n-i)^2 + \sum_{i=1}^{n}(n-i) + n \\ n^3 - n &= 3\sum_{i=1}^{n} n^2 - 6n\sum_{i=1}^{n} i + 3n\sum_{i=1}^{n} 1 - 3\sum_{i=1}^{n} i + 3\sum_{i=1}^{n} i^2 \\ \sum_{i=1}^{n} i^2 &= n^3 + 3n^2/2 + n/2 \end{aligned}$$

2. Pour la complexité :

$$\begin{aligned}
C &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j}1 \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j \\
&= \sum_{i=1}^{n-1}(\sum_{j=1}^{n} j - \sum_{j=1}^{i} j) \\
&= \frac{1}{2}\sum_{i=1}^{n-1}(n(n+1) - (i(i+1)) \\
&= \frac{1}{2}(\sum_{i=1}^{n-1} n(n+1) - \sum_{i=1}^{n-1}(i(i+1))) \\
&= \frac{1}{2}(n(n+1)(n-1) - \sum_{i=1}^{n-1} i^2 - \sum_{i=1}^{n-1} i \\
&= \frac{n(n-1)(n+1)}{3} \\
&= \theta(n^3)
\end{aligned}$$

Il est important de noter que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ et $\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$.

Fin correction exercice 7

Exercice 8 – Complexité

On considère le raisonnement suivant. On sait que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Or $\sum_{i=1}^{n} i \in O(1+2+\ldots+n)$ et $O(1+2+\ldots+n) = O(\max\{1,2,\ldots,n\}) = O(n)$. Donc $\frac{n(n+1)}{2} \in O(n)$. Où se trouve l'erreur.

Correction exercice 8

On a en fait $O(1+2+\ldots+n) = nO(\max\{1,2,\ldots,n\})$.

Fin correction exercice 8

Exercice 9 – Complexité

Montrer que $O(f+g) = O(\max(f,g))$.

Correction exercice 9

Pour montrer cela, il faut montrer les deux inclusions :
— $O(f+g) \subset O(\max(f,g))$.

Pour toute fonction $h(n) \in O((f(n)+g(n)))$, il existe $k>0$ et $n_0 \geq 0$ tels que $h(n) \leq k(f(n)+g(n)) \leq 2k\max(f(n),g(n)),\ \forall n \geq n_0$
donc $h(n) \in O(\max(f(n),g(n)))$.
— Réciproquement, $O(\max(f,g)) \subset O(f+g)$
Pour toute fonction $h(n) \in O(\max(f(n),g(n)))$, il existe $k>0$ et $n_0 \geq 0$ tels que $h(n) \leq k\max(f(n),g(n)) \leq k(f(n)+g(n)),\ \forall n \geq n_0$.
Donc $h(n) \in O((f(n)+g(n)))$.

Fin correction exercice 9

Exercice 10 – Complexité : utilisation du Master Theroem

Expliquer dans chacun des cas suivants pourquoi le Master Theorem ne s'applique pas :

1. $T(n) = 2^n T(n/2) + n^3$
2. $T(n) = T(n/2) + T(n/4) + n^2$
3. $T(n) = 0,5T(n/2) + n$.

Correction exercice 10

1. a n'est pas une constante.
2. On a deux sous-problèmes de tailles différentes.
3. La condition $a \geq 1$ n'est pas vérifiée.

Fin correction exercice 10

Exercice 11 – Complexité

Regrouper en classes d'équivalence pour θ les fonctions suivantes. Trier les classes d'équivalence pour O.

$n,\ 2^n,\ n\log n, n-n^3+7n^5, n^2, \log_2 n, n^3, \sqrt(n)+\log_2 n, \log_2 n^2, n!, \log n$.

Exercice 12 – Programmes récursifs

Analyser la complexité des différentes programmes en fonction du nombre d'appels récursifs effectués :

```
Rec1(n)
si n<2 alors retourner(1)
       sinon retourner(Rec1(n-1)+2)

Rec2(n)
si n<2 alors retourner(1)
       sinon retourner(Rec2(n div 2)+2)

Rec3(n)
si n<2 alors retourner(1)
       sinon retourner(Rec3(n-1)*(Rec3(n-1)+2))

Rec4(n)
```

```
si n<2 alors retourner(1)
       sinon retourner((Rec4(n div 2)+2)*(Rec4( n div 2)+3))
```

Correction exercice 12

1. Nous appelons $t_1(n)$ le nombre d'appels récursifs par calculer $rec_1(n)$.

$$\begin{aligned} t_0 = t_1 &= 0 \\ t_n &= 1 + t_{n_1} \\ t_{n-1} &= 1 + t_{n-2} \\ &\vdots \\ t_2 &= 1 + t_1 \\ t_n &= \sum_{i=2}^{n} 1 + t_1 \\ t_n &= n - 1 \end{aligned}$$

   L'algo est $\theta(n)$.

2. Nous avons $t_n = 1 + t_{n/2}$ et $t_0 = t_1 = 0$. Nous effectuons un changement de variable : nous posons $n = 2^k$ et $v_k = t_n$. Alors nous avons $v_0 = t_1 = 0$ et

$$v_k = 1 + v_{k-1}$$

   Ainsi nous avons $v_k = k$ et $t_n = k = \log_2 n$.
   Une autre possibilité consiste à écrire :

$$\begin{aligned} t_n &= 1 + t_{n/2} \\ t_{n/2} &= 1 + t_{n/4} \\ &\vdots \\ t_{n/(2^i)} &= 1 + t_{n/(2^{i+1})} \\ &\vdots \\ t_1 &= 0 \end{aligned}$$

   Nous devons résoudre $n/(2^i) = 1 \rightarrow i = \log_2 n$. donc $t_n = \log_2 n$.

3. Un exemple où il y deux appels récursifs à $(n-1)$ c'est le problème des tour de Hanoi
   Nous avons plusieurs méthodes pour résoudre ce problème.

— La première méthode est empirique, elle consiste à calculer les premières valeurs, et d'essayer d'en déduire une valeur.

$$\begin{aligned} t_1 &= 0 \\ t_2 &= 2 \\ t_3 &= 6 \\ t_4 &= 14 \\ t_5 &= 30 \end{aligned}$$

Nous supposons donc que $t_n = 2^n - 2$. Cette formule est vraie jusqu'à $n = 5$.
Montrons pour $n+1$ en supossant que c'est vrai pour n.

$$\begin{aligned} t_{n+1} &= 2t_n + 2 \\ &= 2(2^n - 2) + 2 \\ &= 2^{n+1} - 2 \end{aligned}$$

— Second méthode en utilisant un arbres des appels récursifs. Prenons $n = 5$, en bas de l'arbre nous avons 16 appels. Ainsi, le nombre d'appels récursifs est égale au nombre de sommets de l'arbre moins un. Ainsi, nous avons $t_n = \sum_{i=1}^{n-1} 2^i = 2^n - 2$.
— La troisième méthode est calculatoire :

$$\begin{aligned} t_n &= 2t_{n-1} + 2 \\ 2 \times t_{n-1} &= (2t_{n-2} + 2) \times 2 \\ 4 \times t_{n-2} &= (2t_{n-3} + 2) \times 4 \\ &\vdots \\ 2^i \times t_{n-i} &= (2t_{n-i-1} + 2) \times 2^i \\ &\vdots \\ 2^{n-2} \times t_2 &= (2t_1 + 2) \times 2^{n-2} \\ t_n &= \sum_{i=1}^{n-1} 2^i \end{aligned}$$

— Une dernière méthode en utilisant un changement de variable linéaire : $v_n = t_n + \alpha$. Ainsi $v_1 = t_1 + \alpha = \alpha$. $v_n - \alpha = 2(v_{n-1} - \alpha) + 2$ alors $v_n = 2v_{n-1} - \alpha + 2$. On trouve $\alpha = 2$ et d'où $v_1 = 2, v_n = 2v_{n-1} = 2^n$. En revenant à la variable t_n nous obtenons $t_n = v_n - \alpha = 2^n - 2$.
Il est important de noté que nous pouvons linéariser l'algorithme, en prenant $aux := rec3(n-1)$ et faire retourner $(aux * (aux + 2))$.

4. On pose $n = 2^k$, alors $v_k = t_n$, et nous avons $v_k = 2^{k+1} - 2$. Ainsi,

$$\begin{aligned} t_n &= 2^{k+1} - 2 \\ &= 2^{\log_2 n + 1} - 2 \\ &= 2^{\log_2} \times 2 - 2 \\ &= 2n - 2 \end{aligned}$$

En conclusion
— 1 appel récursif à $(n-1)$: $\theta(n)$
— 1 appel récursif à $(n/2)$: $\theta(\log n)$
— 2 appels récursifs à $(n-1)$: $\theta(2^n)$
— 2 appels récursifs à $(n/2)$: $\theta(n)$

Fin correction exercice 12

Exercice 13 – Calculs récursifs

Calculer x^n, pour des valeurs entières et positives de n, de plusieurs possibilité en utilisant les définitions récursives suivantes (qu'il faudra compléter) et évaluer l'ordre de grandeur de la complexité de chacune des méthodes :

1. $x^n = x * x^{n-1}$
2. $x^n = x^{ndiv2} * x^{ndiv2} * x^{nmod2}$
3. Même définition que 2 mais en utilisant une variable intermédiaire pour stocker x^{ndiv2}.

Correction exercice 13

Algorithm 1 Un premier algo : $puis(x, n)$

1. **if** $n = 0$ **then**
   retourner 1
   else
   retourner $(x * puis(x, n-1))$
   end if

En se basant sur l'exercice précédent : $\theta(n)$.

Algorithm 2 Un second algo : $puis1(x, n)$

2. **if** $n \leq 1$ **then**
   if $n = 0$ **then**
   retourner (1)
   else
   retourner (x)
   end if
   else
   retrourner $(puis1(x, ndiv2) * puis1(x, ndiv2) * puis1(x, n \mod 2))$
   end if

En se basant sur l'exercice précédent : $\theta(n)$.

En se basant sur l'exercice précédent : $\theta(\log_2 n)$.

Fin correction exercice 13

Exercice 14 – Calculs récursifs

Algorithm 3 Un troisième algo : $puis2(x, n)$

3. **if** $n \leq 1$ **then**
   if $n = 0$ **then**
   retourner (1)
   else
   retourner (x)
   end if
   else
   $aux := puis2(x, ndiv2)$; retrourner $(aux * aux * puis(x, n \mod 2))$
   end if

On rappelle la définition des nombres de Fibonacci :

$$f_0 = 0,\ f_1 = 1,\ \forall n \geq 2\ f_n = f_{n-1} + f_{n-2}$$

Programmer la fonction f_n :

1. En utilisant la définition récursive (évaluer sa complexité).
2. En utilisant une version itérative efficace (évaluer sa complexité).
3. Montrer que si $n \geq 1$ alors
   — $f_{2n} = f_n^2 + 2 * f_n * f_{n-1}$
   — $f_{2n+1} = f_n^2 + f_{n+1}^2$
   Evaluer approximativement l'ordre de grandeur de la complexité d'un programme pour calculer f_n qui utilise directement cette nouvelle définition récursive.
   En utilisant des variables intermédiaires améliorer le programme précédent et calculer sa nouvelle complexité.
4. Montrer que
   $$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$
   En déduire un programme en $\theta(\log n)$ pour calculer f_n.

Correction exercice 14

Algorithm 4 Un premier algo pour fibonacci

1. **if** $n \leq 1$ **then**
   retourner (n)
   else
   retourner $(f(n-1) + f(n-2))$
   end if

D'après ce qui ce précédence une complexité légèrement inférieure à $\theta(2^n)$.
Soit t_n le nombre d'additions : $t_n = 1 + t_{n-1} + t_{n-2}$
On pose $t_n = v_n - \alpha$. $v_0 = v_1 = \alpha$ et $v_n - \alpha = 1 + v_{n-1} - \alpha + v_{n-2} - \alpha$ et donc $\alpha = 1$. $v_n = v_{n-1} + v_{n-2}$. La complexité est probablement en $\theta(\rho^n)$ avec $\rho < 2$. Il est facile de montrer en utilisant les équations récurrentes d'ordre deux que $t_n = v_n = \theta(\frac{1+\sqrt{5}}{2})^n$.

Algorithm 5 Un second algo pour fibonacci

2. $aux1 := 0; aux2 := 1; aux3 := n$;
   for $i = 2$ à n **do**
   $aux3 := aux2 + aux1$;
   $aux1 := aux2$
   $aux2 := aux3$
   end for
   retourner $(aux3)$

Etudions sa complexité $t_0 = t_1 = 0$ et pour $n \geq 2, \sum_{i=2}^{n} 1 = n - 1 = t_n$

3. si $n = 1$ $f_2 = 1$ et $f_3 = 2$. Supposons vrai pour $\forall k \leq n-1$. Montrons les formules pour n :

$$\begin{aligned} f_{2n} &= f_{2n-1} + f_{2n-2} \\ &= f_{2(n-1)+1} + f_{2(n-1)} \\ &= f_{n-1}^2 + f_n^2 + f_{n-1}^2 + 2f_{n-1} * f_{n-2} \\ &= f_n^2 + 2f_{n-1}^2 + 2f_{n-1} * f_{n-2} \\ &= f_n^2 + 2f_{n-1}(f_{n-1} + f_{n-2}) \\ &= f_n^2 + 2f_{n-1}f_n \end{aligned}$$

$$\begin{aligned} f_{2n+1} &= f_{2n} + f_{2n-1} \\ &= f_n^2 + 2f_{n-1}f_n + f_{2(n-1)+1} \\ &= f_n^2 + 2f_{n-1}f_n + f_{(n-1)}^2 + f_n^2 \\ &= f_n^2 + (f_n + f_{n-1})^2 \\ &= f_n^2 + f_{n+1}^2 \end{aligned}$$

Algorithm 6 Un troisième algo

if $n \leq 1$ **then**
  retourner (n)
else if $(n \mod 2) = 0$ **then**
  retourner $(f(ndiv2) * f(ndiv2) + 2f(ndiv2) * f(ndiv2 - 1))$
else
  retourner $(f(ndiv2) * f(ndiv2) + f(ndiv2 + 1) * f(ndiv2 + 1))$
end if

Soit t_n le nombre d'appels récursifs pour calculer $f(n)$ $t_0 = t_1 = 0$
— Si n est pair : $t_n = 4 + 3t_{n/2} + t_{n/2-1}$
— sinon $t_n = 4 + 2t_{n/2} + 2t_{n/2-1}$
Soit $v_n = \theta(t_n)$ alors $v_0 = v_1 = 0$ et $v_n = 4 + 4v_{n/2}$.
On pose $n = 2^k$ et $w_k = v_n, k = \log_2 n$. Nous avons $w_0 = v_1 = 0$ et $w_k = 4 + 4w_{k-1}$
Posons maintenant, $x_k = w_k + \alpha$, alors $x_0 = w_0 + \alpha = \alpha$. Nous avons $w_k = x_k - \alpha = 4 + 4(x_{k-1} - \alpha)$ et donc $x_k = 4 + 4w_{k-1} - 3\alpha$
Nous voulons que $4 - 3\alpha = 0$ et donc $x_1 = \frac{4^2}{3}, x_2 = \frac{4^3}{3}, \ldots, x_k = \frac{4^{k+1}}{3}$.

Nous obtenons donc $w_k = x_k - \alpha = 4/3(4^k - 1)$, d'où $v_n = 4/3(4^k - 1) = 4/3(4^{\log_2 n} - 1) = \theta(n^2)$ (où $v_k = \frac{4}{3} \times 4^k = \frac{4}{3} \times (2^k)^2$.
Dans le programme, si n est pair, $aux := f(ndiv2)$, sinon $aux := f(ndiv2), auxp := f(ndiv2 + 1)$. La formule permettant de calculer le nombres d'appels récursifs :
$t_0 = t_1 = 0$ et $n \geq 1, t_n = 2 + 2t_{n/2}$, alors $t_n = \theta(n)$.

4. Montrons que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

$f_0 = 0, f_1 = 1, f_2 = 1$
Pour $n = 1$, on a

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} f_2 & f_1 \\ f_1 & f_0 \end{pmatrix}$$

Supposons vrai pour n.

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} &= \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} f_{n+1} + f_n & f_{n+1} \\ f_n + f_{n-1} & f_n \end{pmatrix} \\ &= \begin{pmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{pmatrix} \end{aligned}$$

Pour calculer f_n, il suffit de calculer A^n où $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.
Pour calculer A^n nous avons besoin de calculer :

$$A^n = \begin{cases} (A^{n/2})^2 \text{ si } n \text{ pair} \\ (A^{n/2})^2 * A \text{ sinon} \end{cases}$$

il effectuer $\theta(\log_2 n)$ multiplications de matrices 2×2. Complexité d ela multiplication de matrices $2 \times 2 : \theta(1)$. En conclusion, on calcule f_n en $\theta(\log_2 n)$

Fin correction exercice 14

Exercice 15 – A la recherche d'une valeur encadrée

Etant donné un tableau **trié** d'entiers $T[s \ldots f]$ et deux entiers fixes ("bornes") $a \leq b$, on cherche s'il existe un élément $T[i]$ du tableau tel que $a \leq T[i] \leq b$ (s'il y en a plusieurs donnez en un).

Exemple : Soit le tableau $T[1 \ldots 6] = [3, 7, 8, 43, 45, 98]$ et les bornes $a = 44, b = 54$, alors la solution est donnée par $T[5] = 45$.

1. Donner un algorithme "diviser pour régner" qui résout ce problème. Pour cela donner les différents cas à étudier (absence de solution, ...).
2. a et b étant fixés, on notera $u_{s,f}$ la valeur renvoyée par l'algorithme appliqué à $T[s \ldots f]$. Analyser sa complexité. Pour cela, donner l'équation de récurrence.

Correction exercice 15

1. Coupons le tableau en deux moitiés : $A[s \dots m]$ et $A[m+1 \dots f]$. Les trois cas ci-dessous correspondent à trois positions de l'élément de milieu $A[m]$ par rapport à l'intervalle $[a, b]$.
   — A gauche : $A[m] < a$. Dans ce cas-là tous les éléments de la moitié gauche du tableau sont aussi $< a$, et ne peuvent pas appartenir à l'intervalle demandé $[a, b]$. On va chercher la valeur encadrée dans la moitié droite seulement.
   — Dedans : $a \leq A[m] \leq b$. On a déjà trouvé une valeur encadrée $A[m]$. On retourne son indice.
   — $b < A[m]$. Ce cas est symétrique au premier, la recherche peut être limitée à la moitié gauche du tableau.

   Cette analyse mène à la fonction récursive suivante $chercher(s, f)$ qui renvoie l'indice de la valeur encadrée dans le tableau $A[s \dots f]$ (ou *rien* s'il n'y en a pas). On suppose que le tableau A, et les bornes a, b sont des variables globales.

Algorithm 7 L'algorithme division pour régner $chercher(s, f)$

```
Cas de base
if s = f then
  if A[s] ∈ [a, b] then
    return s
  else
    return Erreur
  end if
end if
On divise pour régner
m = (s + f)/2
if A[m] < a then
  ind = chercher(m + 1, f)
else if a ≤ A[m] ≤ b then
  ind = m
else
  ind = chercher(s, m)
end if
Return ind
```

2. On a réduit un problème de taille n à un seul problème de la taille $n/2$ et des petits calculs en $O(1)$, donc la complexité satisfait la récurrence : $T(n) = T(n/2) + O(1)$. En la résolvant par Master Theorem on obtient que $T(n) = O(\log n)$.

Fin correction exercice 15

Exercice 16 – Recherche binaire versus recherche ternaire

1. Donner les deux algorithmes.
2. Pourquoi la recherche binaire est préférée à la recherche ternaire ? Evaluer la complexité en fonction du nombre de comparaisons.

Correction exercice 16

1. The following is a simple recursive Binary Search function in C++ taken from here.
2. The following is a simple recursive Ternary Search function in C++.
3. From the first look, it seems the ternary search does less number of comparisons as it makes $\log_3 n$ recursive calls, but binary search makes $\log_2 n$ recursive calls. Let us take a closer look.

   The following is recursive formula for counting comparisons in worst case of Binary Search.

   $$T(n) = T(n/2) + 2, T(1) = 1$$

   The following is recursive formula for counting comparisons in worst case of Ternary Search.

   $$T(n) = T(n/3) + 4, T(1) = 1$$

   In binary search, there are 2Log2n + 1 comparisons in worst case. In ternary search, there are 4Log3n + 1 comparisons in worst case.

   Time Complexity for Binary search = $2c \log_2 n + O(1)$ Time Complexity for Ternary search = $4c \log_3 n + O(1)$

   Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2 \log_3 n$ and $\log_2 n$. The value of $2 \log_3 n$ can be written as $(2/\log_2 3) * \log_2 n$. Since the value of $(2/\log_2 3)$ is more than one, Ternary Search does more comparisons than Binary Search in worst case.

Fin correction exercice 16

Exercice 17 – Calcul complexité

```
sum:=0;
for i:=1 to f(n) do
sum:=sum+1
```

où $f(n)$ est un appel de fonction. Donner une borne supérieure simple et approchée en grand O pour le temps d'exécution de cette procédure, en fonction de n, en supposant que

1. le temps d'exécution de $f(n)$ est $O(n)$, et la valeur de $f(n)$ est $n!$
2. le temps d'exécution de $f(n)$ est $O(n)$, et la valeur de $f(n)$ est n
3. le temps d'exécution de $f(n)$ est $O(n^2)$, et la valeur de $f(n)$ est n
4. le temps d'exécution de $f(n)$ est $O(1)$, et la valeur de $f(n)$ est 0

Correction exercice 17

1. $n!$
2. $0(n)$
3. $0(n^2)$

4. 0(1)

Fin correction exercice 17

Exercice 18 – Calcul du minimum/maximum d'un tableau

1. Donner un algorithme qui calcule le min (resp. le maximum) d'un tableau. Vous donnerez le nombre de comparaisons en fonction de n (taille du tableau).
2. On se propose une approche diviser-pour-régner pour le calcul simultané du minimum et du maximum. Donner l'algorithme et calculer le nombre de comparaisons.

Correction exercice 18

1. voic les deux solutions :

Algorithm 8 Algorithme diviser-pour-régner pour le min

```
min = A[1]
for i := 2 to n do
  if min > A[i] then
    min = A[i]
  end if
end for
return min
```

Algorithm 9 Algorithme simple max

```
max = A[1]
for i := 2 to n do
  if max < A[i] then
    max = A[i]
  end if
end for
return max
```

le nombre de comparaison est de $n-1$, donc complexité en $\theta(n)$.

2. Voici la solution
   L'appel initial $MAX - MIN(A, 1, n)$.
   le nombre de comparaison $T(n)$ est donné par $T(n) = 1$ si $n = 2$ sion $T(n) = 2T(n/2)+2$, on trouve $3/2n - 2$ soit un gain de 25%.

Fin correction exercice 18

Exercice 19 – Recherche de pics

Soit un tableau A de taille $n+2$ tel que $A[0] = A[n+1] = -\infty$. $A[i]$ est un pic s'il n'est pas plus petit que ses voisins :

$$A[i-1] \leq A[i] \geq A[i+1]$$

Algorithm 10 Algorithme diviser-pour-régner pour le max – min problème : $MAX - MIN(A, p, r)$

```
if r − p ≤ 1 then
  if A[p] < A[r] then
    return (A[r], A[p])
  else
    return (A[p], A[r])
  end if
end if
q = ⌊(p+r)/2⌋
(max1, min1) = MAX − MIN(A, p, q)
(max2, min2) = MAX − MIN(A, q + 1, r)
return (max(max1, max2), min(min1, min2))
```

$A[i]$ est un minimum local. Le but de cet exercice est de trouver un pic (n'importe lequel). Il est important de noter qu'au moins une solution existe.

1. Donner une solution force -brute (par exemple , le maximum d'un tableau est un pic)
2. Approche divider-pour-régner : Donner le nombre de comparaisons
   - Sonder un élément $A[i]$ et ses voisins $A[i-1]$ et $A[i+1]$.
   - Si c'est un pic : renvoyer i
   - sinon
     - les valeurs doivent croître au moins d'un côté $A[i-1] > A[i]$ ou $A[i] < A[i+1]$
     - Si $A[i-1] > A[i]$ alors on cherche un pic dans $A[1 \dots i-1]$
     - Si $A[i+1] > A[i]$ alors on cherche un pic dans $A[i+1 \dots n]$
   - A quel position i faut-il sonder ?

Correction exercice 19

1. Voir exercice précédent
2. voici l'algorithme :

Algorithm 11 Algorithme recherche d'un pic $PEAK(A, p, r)$

```
m = (g+d)/2
if A[q − 1] ≤ A[q] ≥ A[q + 1] then
  return q ;
else if A[q − 1] > A[q] then
  return PEAK(A, p, q − 1)
else if A[q] > A[q + 1] then
  return PEAK(A, q + 1, r)
end if
```

Le nombre de comparisons est donné par l'équation de récurrence $T(n) = T(n/2) + c_1$ et $T(1) = c_2$, ainsi $T(n) = O(\log n)$.

Fin correction exercice 19

Université de Montpellier 2019/2020
HMIN329 – Complexité et algorithmes Durée: 3h

Tris
TD – Séance n 3

1 Tri de complexité quadratique

Exercice 1 – Tri bulle

Soit l'algorithme suivant qui trie un tableau t :

```
for i:=1 to n-1 do
    for j:=n downto i+1 do
     if t[j] < t[j-1] then echanger (t[j],t[j-1])
```

1. Quel est le nombre de comparaisons dans le pire des cas et en moyenne ?
2. Même question pour le nombre d'échanges.
3. Modifier l'algorithme pour trouver le k^{ieme} plus petit élément du tableau t. Quelle est la complexité de cet algorithme ?
4. Prouver la validité de l'algorithme.

Correction exercice 1

1. Nous avons le nombre de comparaisons dans le pire des cas et en moyenne es sont identiques (il y a dans tous les cas des comparaisons)

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

   L'algorithme est en $\theta(n^2)$. Nous faisons toujours une comparaison

2. $max_e(n) = \frac{n(n-1)}{2}$ (dans le pire des cas, il faut tout le temps échangé le tableau, le tableau doit être trié au départ dans l'ordre décroissant.)
   $moy_e(n) = \sum_{i=1}^{n-1}(\sum_{j=i+1}^{n} 1/2) = \frac{n(n-1)}{4}$. En effet, quel est la proba que je fasse un échange entre deux éléments placé consécutivement, j'ai une proba de 1/2 (j'ai un coup sur deux de fair eun échange).

3. IL suffit d'arêter la première boucle *for* à k (*for* $i := 1$ *to* k *do*), car il va ramener le $k$ieme élément. Ainsi $moy_c(n) = \sum_{i=1}^{k}\sum_{i+1}^{n} 1 = \theta(nk)$. On a intérêt à mettre d'abord les éléments les plus grands à la fin du tableau : $\theta(n(n-k))$.
   En conclusion $\theta(n \times \min(k, n-k))$.
4. La terminaison de l'algorithme est triviale puisque celui-ci est constitué de deux boucles pour imbriquées et que les opérations élémentaires incluses à l'intérieur de ces deux boucles s'exécutent en un temps fini.
   Pour démontrer que l'algorithme trie le tableau par ordre croissant, nous allons considérer le même invariant de boucle que celui du tri par sélection (voir exercice tri par sélection) : « A la fin de la i^{me} itération de la boucle pour i, les i premiers éléments du tableau sont tous plus petits que les $n-i$ éléments restants et sont, de plus, triés par ordre croissant » :

$$\begin{aligned} T_i[i] &\leq T_i[j], \forall j > i \\ T_i[1] &\leq T_i[2] \leq \ldots \leq T_i[i] \end{aligned}$$

   (T_i étant la valeur du tableau T à la fin de l'itération i).
   LA propriété est vraie pour $i = 1$. En effet, la boucle interne pour j compare deux à deux tous les éléments du tableau T, en commençant par la paire $(T[n], T[n-1])$ jusqu'à la paire $(T[2], T[1])$, et procède, à chaque fois, à un échange éventuel des deux éléments considérés de telle sorte que la plus petite des deux valeurs possède l'indice le plus petit. A la fin de cette boucle, la plus petite valeur est donc remontée en première position du tableau.
   On suppose maintenant que la propriété est vraie jusqu'à l'itération i. Par un raisonnement similaire à celui effectué pour le casse base, à l'issue de la boucle interne pour j, le plus petit élément du sous-tableau $T[i+1 \ldots n]$ est donc remonté en position $i+1$. En utilisant l'hypothèse de récurrence, on obtient donc qu'à la fin de la $i+1^{me}$ itération de la boucle pour i, les $i+1$ premiers éléments du tableau sont tous plus petits que les $n-i-1$ éléments restants et sont triés par ordre croissant. La propriété rest donc vraie.
   L'algorithme se termine donc et son résultat est bien le tri du tableau par ordre croissant.

Fin correction exercice 1

Exercice 2 – Tri par sélection

Considérons l'algorithme qui consiste pour trier les éléments dans les cases de 1 à n ($n > 1$) à cherche l'indice, max du plus grand élément dans les cases de 1 à n et échanger les cases d'indice max et n, puis trier les éléments dans les cases de 1 à $(n-1)$.

1. Ecrire l'algorithme
2. Quel est le nombre de comparaisons dans le pire des cas et en moyenne ?
3. Même question pour le nombre d'échanges.
4. Quel est le nombre d'appels récursifs ?
5. Prouver la validité de l'algorithme.

Correction exercice 2

1. Voici la solution de

Algorithm 1 Algorithme de tri sélection, $TS(t, n)$

```
max := 1 ;
for i = 2 à n do
  if T[i] > T[max] then
    Max := i
  end if
end for
echanger(T[n], T[max]) ;
TS(t, n − 1) ;
```

2. Pour trier n éléments, on cherche l'élément le plus grand, nous avons ainsi $(n-1)$ comparaisons et aucun échange. Ensuite, on met cet élément à la fin (1 échange), et nous trions les $(n-1)$ premiers éléments (1 appel récursif).

   Soit t_n le nombre maximum au moyen de comparaisons $t_n = (n-1) + t_{n-1}, n > 1$, on trouve donc $t_n = \frac{n(n-1)}{2}, \theta(n^2)$.

3. Soit v_n le nombre d'échanges (au maximum et en moyenne). Nous avons $v_n = 1 + v_{n-1}$ et alors $v_n = n - 1$. Il est clair vu le fait que nous plaçons à chaque fois l'élément maximum à la fin du tableau nous avons $(n-1)$ échanges. $\max_e(n) = moy_e(n) = \theta(n)$.

4. Soit w_n le nombre d'appel récursif, $w_n = w_{n-1} + 1$ et donc $w_n = n, \theta(n)$.

   si $k \leq n/2$, on cherche le plus petit élément, on le met dans la case i, on trie les $n-1$ derniers éléments.

5. Pour le démontrerai faut montrer que l'algorithme se termine et qu'il a bien pour résultat le tri du tableau. A nouveau, la terminaison de l'algorithme es triviale puisque la boucle est exécutée, n fois et qu'à chaque itération elle fait appel mal procédure $REcheMin$ qui termine (ici avec la boucle for).

   Pour démontrer que l'algorithme trie le tableau par ordre croissant, nous allons considérer l'invariant de boucle suivant : « A la fin l'itération i, les i premiers éléments du tableau sont tous plus petits que les $n-i$ éléments restants et sont, de plus, triés par ordre croissant » :

$$\begin{aligned} T_i[i] &\leq T_i[j], \forall j > i \\ T_i[1] &\leq T_i[2] \leq \ldots \leq T_i[i] \end{aligned}$$

   (T_i étant la valeur du tableau T à la fin de l'itération i).

   LA propriété est vraie pour $i = 1$. En effet l'appel à la procédure $REchMin$ (nous la boucle for) sélectionne le plus petit élément de tout le tableau T et l'échange le place en première position du tableau (si aucun échange n'a eu lieu cela signifie que le plus petit élément se situait déjà en première position du tableau). A la fin de la première itération, le premier élément du tableau est donc bien, par construction, plus petit que les $n-1$ éléments restants et, étant seul, est bien sur trié.

   On suppose maintenant que la propriété est vraie jusqu'à l'itération i. L'appel à la procédure $REchMin(T, i+1, n)$ sélectionne alors le plus petit élément du sous-tableau $T[i+1 \ldots n]$ et l'échange éventuel le met (ou le laisse) en première position de ce sous-tableau. A la fin de l'itération $i+1, T[i+1]$ est donc à la fois, plus petit que les $n-i-1$ éléments du sous-tableau $T[i+1 \ldots n]$ (par construction) :

$$T_{i+1}[i+1] \leq T_{i+1}[j], \forall j > i+1$$

Et plus grand que les i éléments du sous-tableau $T[1 \ldots i]$ (par hypothèse de récurrence) :

$$T_{i+1}[1] \leq T_{i+1}[2] \leq \ldots \leq T_{i+1}[i] \leq T_{i+1}[i+1]$$

La propriété reste donc vraie. L'algorithme se termine donc et son résultat est bien le tri du tableau par ordre croissant.

Fin correction exercice 2

2 Tri efficace en $O(n \log n)$

Exercice 3 – Tri fusion

Nous souhaitons étudier le tri fusion.

1. Rappeler le principe.
2. Appliquer le sur le tableau suivant :

20	5	7	12	21	33	1	6

3. Calculer le nombre maximum de comparaisons. Pour simplifier, on supposera que n est une puissance de 2.
4. Prouver la validité de l'algorithme, le tri du tableau T par ordre croissant. Pour le démontrerai faut montrer que l'algorithme se termine et qu'il a bien pour résultat le tri du tableau.

Correction exercice 3

1. On divise le tableau en deux, on trie chaque partie, et on fusionne les parties triées.
2. Si applique le principe, nous obtenons les deux sous-tableaux suivant : | 20 | 5 | 7 | 12 |

   et | 21 | 33 | 1 | 6 |

   Nous trions chaque sous-tableaux et nous obtenons : | 5 | 7 | 12 | 20 |

   et | 1 | 6 | 21 | 33 |

   — si $n = 1$ le tableau est trié
   — sinon on divise par le tableau en 2 (0 comparaison)
   — on trie la partie gauche (0+appel recursif à $n/2$)
   — on trie la partie droite (même chose)
   — on fusionne les deux parties triées (entre $n/2$ (meilleur des cas) et $n-1$ (pire des cas)).

   Soit t_n le nombre de comparaisons pour trier n éléments. Nous avons $t_1 = 0$ et $n > 1, t_n = t_{n/2} + t_{n/2} + n - 1 = 2t_{n/2} + n - 1$

   Posons $n = 2^k$ et $v_k = t_n$

Algorithm 2 Algorithme de tri fusion, $TF(t, i, j)$

```
if i = j then
  fin
else
  aux := (i + j)div2
  TF(t, i, aux)
  TF(t, aux + 1, j)
  fusion(t, i, aux, j)
end if
```

$$
\begin{aligned}
v_0 = t_1 &= 0 \\
v_k &= 2v_{k-1} + 2^k - 1 \\
2 \times v_{k-1} &= (2v_{k-2} + 2^{k-1} - 1) \times 2 \\
2^2 \times v_{k-2} &= (2v_{k-3} + 2^{k-2} - 1) \times 2^2 \\
&\vdots \\
2^i \times v_{k-i} &= (2v_{k-i-1} + 2^{k-i} - 1) \times 2^i \\
&\vdots \\
2^{k-2} \times v_2 &= (2v_1 + 2^2 - 1) \times 2^{k-2} \\
2^{k-1} \times v_1 &= (2v_0 + 2 - 1) \times 2^{k-1} \\
v_k &= 2^k v_0 + k2^k - \sum_{i=0}^{k-1} 2^i \\
&= k2^k - \frac{2^k - 1}{2 - 1} \\
&= (k-1)2^k + 1
\end{aligned}
$$

Nous trouvons ainsi :

$$
\begin{aligned}
t_n &= (\log_2 n)(n-1) \\
&= n \log_2 n - n + 1 \\
&= \theta(n \log_2 n)
\end{aligned}
$$

3. La terminaison de la procédure fusion est évidente, puisque celle-ci n'est constituée que de boucles pour dont le nombre d'itérations est fixe, et à l'intérieur desquelles s'exécutent un nombre fini d'opérations élémentaires.

   On suppose que les deux sous-tableaux $T[i \dots k]$ et $T[k+1 \dots j]$ sont triés par ordre croissant. La procédure fusion commence par recopier les valeurs de $T[i \dots k]$ dans le sous-tableau $R[i \dots k]$ et les valeurs $T[k+1 \dots j]$ prises dans l'ordre inverse dans $R[k+1 \dots j]$. A l'issue de cette copie, on a donc :

$$
\begin{aligned}
R[r] \leq R[s] \quad & \forall r, s | i \leq r \leq s \leq k \\
R[t] \geq R[u] \quad & \forall t, u | k+1 \leq t \leq u \leq j
\end{aligned}
$$

On montrer alors que la propriété suivante est un invariant pour la boucle pour s de la procédure fusion : « A la fin de l'itération s, le sous-tableau $T[i \dots s]$ est trié par ordre croissant et toutes ses valeurs sont inférieures ou égales à celle des deux sous-tableaux $R[d \dots k]$ et $R[k+1 \dots f]$ ».

La propriété est vraie à l'issue de la première itération, puisque, par construction, l'algorithme affecte à $T[i]$ lapsus petite dessaleurs $R[i]$ et $R[j]$. Orr d'après l'initialisation du tableau R, cette valeur choisie est bien la plus petite valeur du tableau, puisque celle-ci ne pouvait se trouver qu'en première ou en dernière position de $R[i \dots j]$. On suppose que la propriété est vraie pour s. A l'issue de l'itération $s+1$, avaleur de $T[s+1]$ est la plus petite des deux valeurs $R[d]$ et $R[f]$, qui par hypothèse de récurrence, sont toutes les deux supérieures ou égales à toutes les valeurs de $T[i \dots s]$. Le tableau $T[i \dots s+1]$ est donc trié par ordre croissant. Par ailleurs, puisque $T[i]$ a été affecté à la plus petite des deux valeurs et que l'un des curseurs d et f a été mis à jour en conséquence, toutes les valeurs des deux sous-tableaux $R[d \dots k]$ et $R[k+1 \dots f]$ (après mise à jour de d ou de f) sont supérieures ou égales à celles du tableau $T[i \dots s+1]$. La propriété reste donc vraie l'issue de l'itération $s+1$. En conséquence, c'est un invariant de boucle de l'algorithme.

On en déduit donc qu'à la terminaison, la procédure fusion réalise bien le classement par ordre croissant des valeurs des deux sous-tableaux $T[i \dots k]$ et $T[k+1 \dots n]$.

LA procédure fusion parcourt deux fois tous les indices du tableau $T[i \dots j]$, la première fois pour effectuer la copie des valeurs de T dans R, et la deuxième fois pour réaliser la fusion. Il est clair dasn ces conditions que l'algorithme est $O(j-i)$ (et ce, que l'on considère toutes les opérations élémentaires, uniquement les affectations ou uniquement les comparaisons de clés).

La terminaison et la validité de la procédure $TRifusion$ se démontrent simultanément cet très simplement par récurrence sur la taille $n = j - i + 1$ du sous-tableau $T[i \dots j]$. Pour $n \leq 1$, la procédure se termine immédiatement et ne modifie pas le tableau T (qui, réduit à zéro ou un seul élément, est bien sûr trié). On suppose que l'appel de la procédure $Trinsertion$ pour tous paramètres i et j tels que $j - i + 1 < n$ se termine et réalise le tri du sous-tableau $T[i \dots j]$. On considère alors l'appel de la procédure $Trinsertion$ avec deux paramètres i et j tels que $j - i + 1 = n$. Cet appel découpe $T[i \dots j]$ en deux sous-tableaux $T[i \dots k]$ et $T[k+1 \dots n]$, chacun de taille strictement inférieure à n. L'hypothèse de récurrence nous permet d'affirmer que chacun des appels récursifs de la procédure sur ces deux sous-tableaux se termine et réalise le tri des sous-tableaux. La procédure de fusion, nous l'avons déjà montré, permet alors de classer les valeurs du tableau $T[i \dots j]$ par ordre croissant.

Fin correction exercice 3

Exercice 4 – Tri rapide

1. Soit T un tableau de n entiers distincts. Donner un algorithme efficace qui déplace les entiers de T de telle façon que les cases de plus petits indices contiennent les entiers inférieurs à l'entier qui était initialement en $T[1]$ (cet entier sera appelé pivot dans la suite), et les cases de plus grands indices les entiers plus grands que le pivot. Plus précisément, s'il y a dans T, $(p-1)$ entiers inférieurs au pivot alors les cases de 1 à $(p-1)$ contiennent ces entiers, la case p contient le pivot, et les cases de $(p+1)$ à n contiennent les entiers plus grand que le pivot.
2. En déduire un algorithme de tri utilisant deux appels récursifs et analyser cet algorithme au pire et en moyenne. Pour le calcul de la moyenne nous rappelons que $moy_A(n) =$

$\sum_{d\in D_n} p(d) \times cout_A(d)$ avec $p(d)$ la probabilité que l'on ait la donnée d en entrée de l'algorithme et D_n est l'ensemble des données de taille n.

(a) Quel est la probabilité que le pivot se trouve à la case k ?

(b) Quels est la conséquence sur la taille des tableaux dans les deux appels récursifs ?

(c) En déduire une formule de récurrence pour la moyenne ?

3. quelle est la taille maximum de la pile nécessaire pour gérer les appels récursifs de cet algorithme ?
4. Réécrire l'algorithme avec un seul appel récursif et une boucle **tant que** en décursifiant l'appel récursif terminal.
5. Modifier l'algorithme proposé en 4) pour que l'appel récursif porte sur la partie du tableau contenant le moins d'entiers et montrer que la pile nécessaire est alors de hauteur logarithmique au pire.
6. Modifier l'algorithme de tri rapide de façon à avoir le k^{ieme} plus petit élément du tableau.
7. **Subsidiaire** Montrer que l'algorithme précédent est linéaire (c'est à dire en $\theta(n)$) en moyenne).

Correction exercice 4

1. La procédure partiion effectue la partition en sous-tableau compris en tre les indices i et j en fonction du pivot $t[i] = p$ et rend comme résultat l'indice p où ce pivot a été placé. Voici l'algorithme

Algorithm 3 Algortihme de Partition

```
Partition(t, p, i, j)
m := i + 1 ;
p := j;
pivot := t[i]
while m ≤ p do
  while t[m] ≤ pivot do
    m := m + 1 ;
  end while
  while t[p] > pivot do
    p := p − 1 ;
  end while
  if m < p then
    echanger (t[m], t[p])
    m := m + 1 ; p := p − 1 ;
  end if
end while
echanger (t[i], t[p]) (pour mettre le pivot à sa place)
```

2. Voici l'algorithme principal

   Le plus mauvais cas : on ne partitionne pas si le pivot dans la première ou la dernière place après l'exécution de la partition.

Algorithm 4 Algorithme de tri rapide $T(i, j)$

```
if i < j then
  partition(t, p, i, j) (p =partitionT(i, j)
  TR(T, i, p − 1)
  TR(T, p + 1, j)
end if
```

Soit v_n le nombre maximum de comparaisons
$v_0 = v_1 = 0$ et $n \geq 2$,

$$\begin{aligned}
v_n &= (n+1) + v_{n-1} \\
v_{n-1} &= v_{n-2} + n \\
v_{n-2} &= v_{n-3} + (n-1) \\
&\vdots \\
v_2 &= v_1 + 3 \\
v_n &= v_1 + \sum_{i=3}^{n+1} i \\
&= \frac{(n+1)(n+2)}{2} - 3 \\
&= \frac{(n-1)(n+4)}{2} = \theta(n^2)
\end{aligned}$$

Maintenant analysons en moyenne :

Hypothèses : tous les éléments sont différents et toutes les permutations des éléments sont équiprobables.

Après l'appel à partition, le pivot à $1/n$ d'être à une case p donnée.Si le pivot est à la case p, on fait deux appels récursifs : un sur un tableau de taille $(p-1)$, et l'autre sur un tableau de taille $(n-p)$.

Soit t_n le nombre moyen de comparaison pour le tri rapide :
$n - 1 + 1/n(\sum_{p=1}^{n}(t_{p-1} + t_{n-p}) \leq t_n \leq n + 1 + 1/n(\sum_{p=1}^{n}(t_{p-1} + t_{n-p})$. Donc en prenant que la partie haute :

$$t_n = \overbrace{(n+1)}^{\text{nb comparaison dans partition}} + \overbrace{\frac{1}{n}}^{\text{proba d'avoir le pivot à}} \overbrace{\sum_{k=1}^{n}(}^{\text{Toutes les cases possibles pour le pi}} \overbrace{t_{k-1}}^{\text{premeir appel récursif}} + \overbrace{t_{n-k}}^{\text{second apel}}$$

ou $t_n = (n+1) + \sum_{k=1}^{n} \text{proba(pivot est à la case } k \times \text{cout(si le pivot est à la case } k$
Nous obtenons $t_n = (n+1) + \frac{2}{n}(\sum_{k=0}^{n-1} t_k)$.
Or $\sum_{k=1}^{n-1} t_k = \frac{n}{2}(t_n - (n+1)), n \geq 2$ et $\sum_{k=1}^{n-2} t_k = \frac{n-1}{2}(t_{n-1} - n), n \geq 3$. Et donc $t_{n-1} = \frac{n}{2}(t_n - n + 1) - \frac{n-1}{2}(t_{n-1} - n)$. En effectuant un de calculs, nous trouvons $nt_n = (n+1)t_{n-1} + 2n$ et ainsi

$$
\begin{aligned}
\frac{t_n}{n+1} &= \frac{t_{n-1}}{n} + \frac{2}{n+1} \\
\frac{t_{n-1}}{n} &= \frac{t_{n-2}}{n-1} + \frac{2}{n} \\
\frac{t_{n-2}}{n-1} &= \frac{t_{n-3}}{n-2} + \frac{2}{n-1} \\
&\vdots \\
\frac{t_4}{4} &= \frac{t_2}{3} + \frac{2}{4} \\
\frac{t_n}{n+1} &= \frac{t_2}{3} + 2\sum_{i=4}^{n+1} \frac{1}{i} \\
t_n &= (n+1)(1 + 2(H_{n+1} - 1 - 1/2 - 1/3)) \\
t_n &= (n+1)2H_{n+1} \\
t_n &= \theta(n \log n)
\end{aligned}
$$

Algorithm 5 première modification $TR(t, i, j)$

3. **while** $i < j$ **do**
   partition(t, p, i, j)
   $TR(t, i, p-1)$
   $i := p + 1$
   end while

Algorithm 6 Deuxième modification $TR + (t, i, j)$

4. **while** $i < j$ **do**
   partition(t, p, i, j)
   if $(j - p) > (p - i)$ **then**
   $TR + (t, i, p-1)$
   $i := p + 1$
   else
   $TR + (t, p+1, j)$
   $j := p - 1$;
   end if
   end while

Montrer que si t_n est la taille maximale de la pile pour $TR+$ avec un tableau de taille n, alors $t_n \leq \lfloor \log_2 n \rfloor$: Ceci est vrai pour les cas de base $t_1 = 0, t_2 = 1$ Supposons que c'est vrai $\forall n < m$ et montrons le pour m. Dans le programme $TR+$, on a un appel récursif sur un tableau de taille $r \leq n/2$, alors $t_n \leq 1 + t_{n/2} \leq 1 + \lfloor \log_2 n/2 \rfloor \leq \lfloor \log_2 n \rfloor$.

La taille à prévoir de la pile des appels récursifs est maximum quand l'arbre binaire des appels de P est de profondeur n, avec un enchaînement de n appels non terminaux (première partie des apples récursifs) cela correspond au cas où x le pivot se place à chaque fois à la fin de la sous-liste.

5. Si $k = 30$, après partition, on a $p = 100$.On peut chercher entre 1 et 99 éléments. On suppose que $i \leq k \leq j$.

Algorithm 7 Algorithme donnant le k ieme plus petit élément $K(p, k, i, j)$

partition(t, p, i, j)
if $k < p - i$ **then**
  retourner $K(t, k, i, p-1)$;
else if $k = p - i$ **then**
  retourner$(t[k])$;
else
  retourner $K(t, k, p+1, j)$
end if

6. Soit $c(n)$ le nombre moyen de comparaisons que nécessite cet algorithme :

$$\forall k, c(n) \quad \overbrace{(n+1)}^{\text{partition}} + \overbrace{\max_{1 \leq k \leq n} \{\frac{1}{n}(\sum_{i=1}^{k-1} c(n-i) + \sum_{i=k+1}^{n} c(i-1))\}}^{\text{appel récursif d'un coté ou de l'autre}}$$

Fin correction exercice 4

Exercice 5 – Arbres binaires : Application au tri par tas

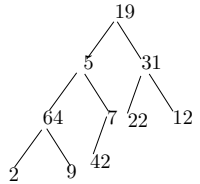


FIGURE 1 – Arbre binaire

1. Un arbre parfait est un arbre binaire dont tous les niveaux sont complétement remplis sauf éventuellement le dernier niveau, et dans ce cas les sommets (feuilles) du dernier niveau sont groupés le plus à gauche possible.

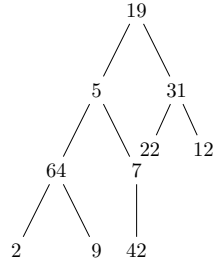


FIGURE 2 – Arbre binaire

(a) Comment peut-on représenter efficacement un arbre binaire parfait par un tableau et un entier (représentant la taille en nombre d'éléments du tas) ? Donner le tableau reprsésentant l'arbre binaire proposé à la figure 1.

(b) Donner les fonctions permettant de savoir si un sommet est une feuille, de connaître le père d'un sommet, le fils gauche, le fils droit.

(c) Calculer la hauteur d'un arbre binaire parfait.

2. Un tas est un arbre binaire parfait tel que chaque élément d'un sommet est inférieur ou égal aux éléments de ses fils.

(a) Donner les fonctions permettant d'obtenir le minimum d'un tas,

(b) de supprimer le minimum d'un tas,

(c) d'ajouter un élément à un tas.

(d) Evaluer la complexité de chacune de ces fonctions

3. Donnez une fonction qui structure un tableau quelconque en tas en ajoutant successivement les éléments au tas déjà construit (initialement le tas est vide, la partie gauche du tableau est strucutrée en tas, les éléments du tableau sont ajoutés un à un de la gauche vers la droite). Evaluer la complexité de cette construction.

4. Même question mais en construisant le tas de la droite vers la gauche. On considère que les feuilles sont des tas et on fusionne les tas en ajoutant les sommets internes. Evaluer la complexité de cette construction.

5. En déduire un algorithme de tri par ordre décroissant d'un tableau (tri par tas). Evaluer sa complexité.

6. Adapter-le pour trouver le k^{ieme} plus petit élément d'un tableau. Evaluer la complexité de recherche de cet élément.

7. Donner un algorithme qui structure un tableau en tas en $O(n)$.

Correction exercice 5

1. Organisation d'un tas sous forme d'un tableau

(a) La première case du tableau indique la taille du tableau.

10	19	5	31	64	7	22	12	2	9	42

(b) Où est le père de l'élément à la case i ? Il se trouve à la case i *div* 2. Où sont les fils de l'élément à la case i ? $2 \times i$ and $2 \times i + 1$. Si $t[0]$ est pair et $i = t[0]$ *div* 2, i n'a qu'un fils en case $2 \times i$. Si $i > t[0]$ *div* 2 l'élément est une feuille.

(c) Voici les algorithmes

Algorithm 8 Algorithme des feuilles $feuilles(t, i)$

retourner $((2 \times i) > t[0])$

(d) La hauteur d'un arbre binaire parfait est égale aux nombres de noeuds : $\lfloor \log_2 n \rfloor$.

2. Voici les algorithmes demandés :

(a) Fonction qui donne le minimum d'un tas : c'est la racine du tas

(b) Fonction qui donne la suppression

Algorithm 9 Algorithme du père $pere(t, i)$

if $i = 1$ **then**
  "erreur"
else
  retourner $(i \ div 2)$
end if

Algorithm 10 Algorithme du fils gauche $gauche(t, i)$

if $feuille(t, i)$ **then**
  "erreur"
else
  retourner $(2 \times i)$
end if

Algorithm 11 Algorithme du fils droit $droit(t, i)$

if $feuille(t, i)$ ou $(2 \times i) = t[0]$ **then**
  "erreur"
else
  retourner $(2 \times i + 1)$
end if

Algorithm 12 Algorithme minimum des fils $minfils(t, i)$

$j := 2 \times i$;
if $j \neq t[0]$ **then**

  if $t[j + 1] < t[j]$ **then**
    $j := j + 1$
  end if
end if
retourner (j)

Algorithm 13 Algorithme de suppression du minimum $supmin(t)$

$t[1] := t[t[0]]$;
$t[0] := t[0] - 1$;
$i := 1$
while not $feuille(t, i)$ **do**
  $j := minfils(t, i)$;
  if $t[i] > t[j]$ **then**
    echanger$(t[i], t[j])$
    $i := j$;
  else
    $i := t[0]$; ou exit, break
  end if
end while
retourner (t) ;

Algorithm 14 Algorithme ajouter $ajouter(t, x)$

```
t[0] := t[0] + 1 ;
t[t[0]] := x ;
i := t[0]
while i = 1 ou t[i] < t[i div 2] do
  echanger (t[i], t[i div 2])
  i := i div 2
end while
retourner (t) ;
```

Algorithm 15 Algorithme de structuration $struct(t)$

```
n := t[0] ;
t[0] := 0 ;
for i := 1 à n do
  ajouter(t, t[i])
end for
```

(c) Fonction ajouter

3. Voici la fonction qui structure un tableau en un tas quelconque (voir fonction $struct(t)$.
4. Construction d'un tas de la droite vers la gauche

Algorithm 16 Algorithme de structuration bis $structbis(t)$

```
for i := t[0] div 2 downto 1 do
  aux := i
  repeat
    j := minfils(t, aux) ;
    if t[aux] > t[j] then
      echanger(t[aux], t[j]) ;
      aux := j ;
    else
      aux := t[0]
    end if
  until feuille(t, aux)
end for
retourner(t)
```

Etudions la complexité :Si $n = 2^k$,
— si $i \in [2^{k-1}, 2^k - 1]$, 0 comparaison
— si $i \in [2^{k-2}, 2^{k-1} - 1]$, 2 comparaisons
— si $i \in [2^{k-3}, 2^{k-2} - 1]$, 4 comparaisons
— si $i \in [2^{k-j-1}, 2^{k-j} - 1]$, $2j$ comparaisons
— si
Construction d'un en temps linéaire :
Ainsi, le nombre de comparaisons au pire : $2^{k-1} \times 0 + 2 \times 2^{k-2} + \ldots 2(k-1)2 = \sum_{j=1}^{k-1} 2j \times 2^{k-j-1}$

$$\begin{aligned} C &= \sum_{j=1}^{k-1} 2j \times 2^{k-j-1} \\ &= \sum_{j=1}^{k-1} 2j \times \frac{2^k}{2^{j+1}} \\ &= 2^k \sum_{j=1}^{k-1} \frac{j}{2^j} \end{aligned}$$

Or $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$. Or $\sum_{j=1}^{k-1} \frac{j}{2^{j-1}} \leq \sum_{j=1}^{\infty} \frac{j}{2^{j-1}} = \sum_{j=1}^{\infty} j(\frac{1}{2})^{j-1} = 4$ car $= \frac{1}{(x-2)^2} = (\frac{1}{1-x})' = \sum_{i=0}^{\infty} ix^{i-1} \leftarrow x = 1/2$.
Donc $C \leq \frac{4(n+1)}{2} \leq 2n + 2$.

5. Algorithme qui effectue un tri en utilisant un tas.

Algorithm 17 Algorithme de tri par tas $tripartas(t)$

```
p := t[0] ;
for i := p downto p − k + 1 do
  aux := t[1];
  supprimermin(t)
  t[i] := aux
end for
t[0] := p ;
```

Le coût de $supprimer(t)$ est en $\log_2 i$ La complexité en nombre de comparaison $\theta(n) = \sum_{i=1}^{n} \log_2 i = \theta(n \log_2 n)$.

6. A faire
7. IL suffit d'adapter le tas en prenant comme racine le maximum, ou bien d'affecter le minimum dans la dernière case du tableau.
8. La complexité est $\theta(n + k \log_2 n)$.
9. A heap could be built by successive insertions. This approach requires $O(n \log n)$ time because each insertion takes O(logn) time and there are n elements. However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height h (measured from the bottom) have already been "heapified", the trees at height $h+1$ can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes $O(h)$ operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Le nombre de sommets à une hauteur h est $\leq \lceil \frac{n}{2^{h+1}} \rceil$
$\sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}}) \leq O((n \sum_{h=0}^{\infty} \frac{n}{2^{h+1}}) = O(n)$ sachant que la série $h/2^h$ converge vers 2.
Une autre preuve : We will use the back-to-front heap building algorithm based on the heap-adjust procedure used in deleting from a heap. That is, starting from the last parent

$[i = n/2]$ and backing up $[i - -]$ to the root of the tree, adjust from node i down using the fact that the nodes below are already in a heap structure.

To simplify the argument, assume the length of the list is a power of two. If not, expand the list and the following will be a little on the high side.

So, let $n/2^k$. There will be $n/2 = 2^{k-1}$ leaves which will not need any adjusting since the loop starts at the last parent. There will be $n/4 = 2^{k-2}$ nodes that would take one pass through the adjust loop, $n/8 = 2^{k-3}$ nodes that would take at most two passes through the adjust loop, or, in general, $n/2^{i+1} = 2^{k-i-1}$ nodes that would take at most i passes though the loop. For simplification purposes, we assume that the root will take at most $k-1$ passes even though it could take k since one extra pass will not change the basic result. Thus, the total number of passes through the loop is given by $A = \sum_{i=1}^{k-1} i2^{k-i-1}$.

Rappel $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$.

$A = \sum_{i=1}^{k-1} i2^{k-i-1} = 1.2^{k-2} + 2.2^{k-3} + 3.2^{k-3} + \ldots + (k-2).2^1 + (k-1)2^0$. Nous allons réordonner les éléments :

$$\begin{aligned}
\sum_{i=1}^{k} i2^{i-1} &= 2^{k-2} + 2^{k-3} + 2^{k-4} + \ldots + 2^1 + 2^0 \\
&= \quad +2^{k-3} + 2^{k-4} + \ldots + 2^1 + 2^0 \\
&= \quad\quad +2^{k-4} + 2^{k-5} + \ldots + 2^1 + 2^0 \\
&= \quad \vdots \\
&= \quad\quad\quad\quad\quad\quad +2^0
\end{aligned}$$

— Considérons la première ligne : $\sum_{i=1}^{k-1} 2^{i-1} = 2^{k-1} - 1$.
— Considérons la seconde ligne ligne : $\sum_{i=1}^{k-2} 2^{i-1} = 2^{k-2} - 1$.
— et ainsi de suite ; la dernière ligne correspond à $2^1 - 1$
Nous obtenons donc

$$\begin{aligned}
\sum_{i=1}^{k} i2^{i-1} &= (2^{k-1} - 1) + (2^{k-2} - 1) + \ldots + (2^1 - 1) \\
&= (2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 2) - (k-1) \\
&= 2^k - k - 1 = n - \log n - 1
\end{aligned}$$

Fin correction exercice 5

Exercice 6 – Tri basé sur un Arbre binaire de recherche

1. Appliquer le principe sur les données suivantes : $10, 8, 6, 9, 3, 7, 2$.
2. Donner l'algorithme d'un tri basé sur les arbres binaires
3. Expliquer la correction de l'algorithme.
4. Donner la complexité dans le pire des cas. Comment améliorer la complexité pour éviter le pire des cas ?
5. Donner la complexité en moyenne.

Profondeur	Nombres d'éléments stockés
0	1
1	$1 + 2 = 3$
2	$1 + 2 + 4 = 7$
j	$2^0 + 2^1 + 2^2 + \ldots + 2^j$

TABLE 1 – Nombre de comparisons

Correction exercice 6

1. Rappeler la structure des arbres de binaires de recherche.
2. Facile
3. L'algorithme 3

Algorithm 18 Algorithme de tri par des arbres binaires de recherche

```
L the input set of elements.
A := ∅
for e ∈ L do
   A := insertion(A, e)
end for
L := ParcoursInfixe(A)
Return L ;
```

4. Pour n il faut comparer au pire $n-1$ éléments (on a arbre filiforme). $T(n) = T(n-1)+n-1$ et $T(2) = 1$, donc $0(n^2)$ dans le pire des cas. Pour contrer ceci, il suffit de maintenir un arbre binaire de recherche équilibré.
5. $0(n \log n)$ en moyenne.
   Quand j^{iem} niveau est rempli, il ya donc le nombre d'éléments suivant dans l'arbre (voir Table 1) :

$$\sum_{i=0}^{j} 2^j = 2^{j+1} - 1$$

   Donc le nombre de comparaisons pour insérer le p^{ieme} élément au niveau j est $p = 2^{j+1} - 1$ d'où $j = \log_2(p-1) - 1$. Pour insérer les n éléments de l'ensemble à classer dans l'arbre, et donc construire ce même arbre, il effectuer le nombre de comparaisons $T(n) = \sum_{p=1}^{n} \log_2(p+1) - 1 \leq n \log_2 n$.

Fin correction exercice 6

3 Tri linéaire

Exercice 7 – Tri linéaire : tri par dénombrement

1. Appliquer l'algorithme de tri par dénombrement au tableau suivant :

20	5	7	12	21	33	1	6

2. On suppose que la dernière boucle (dans l'algorithme donné en cours) est réécrite de la manière suivante :

   For $i = 1$ à n **do**

   Montrer que l'algorithme fonctionne encore correctement. L'algorithme modifié est-il stable (l'ordre des éléments identiques est équivalent entre le tableau initial et final) ?

Correction exercice 7

1. Facile
2. Oui il fonctionne mais le tri n'est plus stable.

Fin correction exercice 7

Exercice 8 – Tri linéaire : tri par paquets

Algorithm 19 Tri par paquets (A)

$n := longueur(A)$
for $i = 1$ à n **do**
  insérer $A[i]$ dans liste $B[\lfloor nA[i] \rfloor]$
end for
for $i = 1$ à $n - 1$ **do**
  Trier la liste $B[i]$ à l'aide du tri par insertion
end for
concaténer les listes $B[0], B[1], \ldots, B[n-1]$ dans l'ordre
Retourner B ;

1. Appliquer l'algorithme de tri par dénombrement au tableau suivant :

.78	.17	.39	.26	.72	.94	.21	.12	.23	.68

2. Quelle est la complexité au pire ? Comment peut-on résoudre ce pire cas ?

Correction exercice 8

1. Facile
2. La complexité au pire est donnée quand toutes les listes sont vides sauf une. Tous les éléments sont regroupés dans une seule liste. La complexité est donc donnée par la complexité de l'algorithme du tri par insertion, soit en $0(n^2)$. Une solution serait augmenter la valeur de n afin de garantir des listes non vides.

Fin correction exercice 8

Exercice 9 – Tri linéaire : tri par dénombrement

Appliquer l'algorithme de tri par dénombrement au tableau suivant :

20	5	7	12	21	33	1	6

Correction exercice 9

Facile

Fin correction exercice 9

Exercice 10 – Tri linéaire : tri par base

Algorithm 20 Tri par base (A, d)

for $i = 1$ à d **do**
  Utiliser un tri stable pour trier le tableau A selon le chiffre i
end for

1. Appliquer l'algorithme sur les éléments suivants : $329, 457, 657, 839, 436, 720, 355$.
2. Montrer : Etant donnés n nombres de d chiffres dans lesquels chaque chiffre peut prendre k valeurs possibles, le tri-base trie correctement ces nombres en un temps $\theta(d(n + k))$. On utilisera la récurrence sur les nombre de colonnes en cours de tri.

Correction exercice 10

1. Facile
2. L'analyse du temps d'exécution dépend du tri stable qui sert d'algorithme de tri intermédiaire. Quand chaque chiffre est dans l'intervalle 0 à $k-1$ (il peut donc prendre k valeurs possibles) et quand k n'est pas trop grand, le tri par dénombrement est le choix évident. Chaque passe sur les n nombres à c chiffres prend alors un temps $\theta(n + k)$. Comme il y a c passes, le temps total du tri par base est $\theta(d(n + k))$. Quand d est constant et quand $k = O(n)$, le tri par base s'exécute en temps linéaire.

   Dans la récurrence il est important de mettre en avant le fait que le tri est stable. C'est dans l'hypothèse de récurrence où l'on doit mettre en avant que le tri est stable.

Fin correction exercice 10

4 Autour des tris

Exercice 11 – Tri par insertion : analyse du coût moyen

L'analyse en moyenne d'un algorithme A sur l'ensemble D_n des données de taille n est donnée par la formule suivante :

$$Moy_A(n) = \sum_{d \in D_n} p(d).cout_A(d)$$

où $p(d)$ est la probabilité que l'on ait la donnée d en entrée de l'algorithme. On note par $cout_A(d)$ la complexité en temps de l'algorithme A sur la donnée d.

1. Rappeler l'algorithme du tri par insertion.
2. Calculer le coût en termes de comparaisons
3. Quelle la probabilité que l'élément à la position i se trouve à la position k. Conclure.

Correction exercice 11

1. Voici l'algorithme :

Algorithm 21 Tri d'un tableau par insertion

for $i = 2$ à n **do**
  $j := i, v := T[i]$
  while $j > 1$ et $v < T[j-1]$ **do**
    Echanger$(T[j], T[j-1])$;
    $j := j - 1$;
  end while
  $T[j] := v$;
end for

2. Pour faire l'analyse en moyenne on va d'abord représenter l'espace des données. Ici on dispose d'un tableau de n éléments comparables 2 à 2 avec une relation d'ordre total $<$. Sans perte de généralité, comme seul l'ordre compte, on peut numéroter ces éléments de 1 à n (correspondant à leur rang), et le tableau est associé à une permutation de $\{1, 2, \ldots, n\}$ (un mélange). Le nombre total de permutations possibles est $n! = n.(n-1)\ldots 3.2.1$. Sans information supplémentaires sur les données, on calculera le coût moyen de l'algorithme comme la moyenne du coût pour chacune des permutations. Ce qui revient à construire un modèle probabiliste de la situation : on suppose que l'entrée de l'algorithme est tirée uniformément (équiprobabilité) dans l'espace des données, on note C la variable aléatoire correspondant au coût de l'algorithme sur l'entrée proposée et on souhaite calculer le coût moyen, c'est à dire EC. Dans ce qui suit, on suppose n fixé. Comme l'algorithme est composé d'une itération, le For, le coût total de l'algorithme se décompose en $C = C2 + C3 + \ldots + Cn$ avec Ci le coût associé à l'itération i. On en déduit que $EC = E(C2 + C3 + \ldots + Cn) = EC2 + EC3 + \ldots + ECn$.
   Pour l'itération i, la boucle While va insérer l'élément qui est dans la case i dans la case k avec $1 \leq k \leq i$, pour cela il y aura $i - k + 1$ comparaison d'éléments, sauf pour la première case où on ne fait que $(i-1)$ comparaisons. Comme le tirage de la permutation est uniforme, l'élément en position i peut se retrouver dans n'importe laquelle des positions de 1 à i et avec la même probabilité $1/i$. D'où le coût moyen de l'itération

$$ECi = \frac{1}{i}((i-1) + (i-1) + \ldots + 3 + 2 + 1) = \frac{i+1}{2} - \frac{1}{i}$$

   Le coût moyen de l'algorithme se déduit

$$C_{moy}(n) = \sum_{i=2}^{n}(\frac{i+1}{2} - \frac{1}{i}) = \frac{(n+1)(n+2)}{4} - H_n - 1/2$$

Fin correction exercice 11

Exercice 12 – Recherche du $k$ième élément

On dispose d'une liste S non triée de n entiers deux à deux distincts, et on cherche pour k donné, $1 \leq k \leq n$, le $k$ième élément de cette liste (c'est-à-dire l'entier tel que $k-1$ autres entiers lui soient inférieurs). On propose l'algorithme récursif suivant :

— Si $|S| = 1$, l'entier cherché est l'élément de S.
— Sinon on choisit dans S un élément s et on pose $S_1 = \{x \in S | x < s\}$:
  — Si $|S_1| \geq k$, on applique à nouveau l'algorithme sur S_1
  — Si $|S_1| = k - 1$, l'élément cherché est s.
  — Si $|S_1| < k - 1$, on applique à nouveau l'algorithme sur S_2.

On suppose que S est représenté par un tableau $t[1 \ldots n]$ et qu'on prend pour s l'élément du tableau le plus à gauche.

1. Ecrire la procédure $slection - du - kime$ qui implémente l'algorithme (on pourra utiliser la procédure partition).
2. Soit $CS(n)$ le nombre moyen de comparaisons que nécessite cet algorithme $\forall k$. Etablir l'inégalité :

$$CS(n) \leq n + 1 + \max_{1 \leq k \leq n}\{\frac{1}{n}(\sum_{i=1}^{k-1} CS(n-i) + \sum_{i=k+1}^{n} CS(i-1))\}$$

Correction exercice 12

1.

Fin correction exercice 12

Exercice 13 – Borne inférieure

Soit à trier une séquence de n éléments. La séquence est constituée de n/k sous- séquences, chacune contenant k éléments. Les éléments d'une sous-séquence donnée sont tous plus petits que les éléments de la sous-séquence suivante et tous plus grands que les éléments de la sous-séquence précédente. Pour trier la séquence complète de longueur n, il suffit donc de trier les k éléments de chacune des n/k sous-séquences. Prouver l'existence d'un minorant $\Gamma(n \log k)$ pour le nombre de comparaisons requis par la résolution de cette variante du problème de tri.

Correction exercice 13

1. Pour chaque liste de taille k, il faut $k \log k$ au minimum, en multipliant par les n/k listes on trouve $n/k \times k \log k$.

Fin correction exercice 13

Exercice 14 – Tableau de 0 et de 1

Soit un tableau T de 0 et de 1. Donner un algorithme simple de complexité linéaire pour trier le tableau T.

Correction exercice 14

1. Plusieurs stratégies sont possibles dont une porte sur la technique pour le tri rapide avec deux pointeurs.

Fin correction exercice 14

Exercice 15 – Drapeau Hollandais

Soit un tableau de n éléments, chacun coloré en bleu, blanc ou rouge. L'objectif est de réorganiser le tableau pour que

— les éléments bleus soient sur la partie gauche
— les éléments blancs au centre
— les rouges en fin de tableau

La contrainte : utiliser un minimum de mémoire supplémentaire

1. Donner les règles d'échanges
2. Donner l'algorithme.
3. Quel est le meilleur cas ?
4. Quel est le pire cas ?
5. Quelle est la complexité en termes de comparaisons ?
6. Quelle est la complexité en termes d'échanges ?

Correction exercice 15

1. On utilise trois pointeurs : bleu, blanc et rouge. Pour le bleu celaa indique que la case de valeur bleu−1 est la dernière case bleu. Bleu indique la case sur laquelle on travaille. Blanc indique la case non blanche à gauche (blanc+1 est la dernière case blanche). Même chose pour le pointeur rouge (noter que rouge pointe sur le dernier blanc).
   — Soit C_1 le cas où la case est bleu (un échange)
   — Soit $_2$ où la case est blanche (un échange)
   — Soit C_3 où la case est rouge (deux échanges)
2. Voici l'algorithme 2 :
3. Le meilleur est cas est donné quand le tableau admet que des bleus. complexité en nombre d'échanges $O(1)$.
4. le pire est donné quand il ya que des rouges. On fait deux échanges.
5. La complexité en termes de comparaisons : n. Chaque est traitée une seule fois.
6. La complexité en terme d'échanges est bornée par $2n$. Il ya au plus deux échanges cas 3).

Fin correction exercice 15

Algorithm 22 Drapeau Hollandais

```
i_1 = 1 ; i_2 = i_3 = n ;
while i_1 ≤ i_2 do
  if Couleur(T[i_1]) = bleu then
    i_1 + +
  else if Couleur(T[i_1]) = blanc then
    Echanger(T[i_1], T[i_2]) ;
    i_2 − −
  else
    Echanger(T[i_1], T[i_2]) ; Echanger(T[i_2], T[i_3]) ; i_2 − − ; i_3 − − ;
  end if
end while
```

Université de Montpellier 2020/2021
HMIN329 – Complexité et algorithmes Durée: 4.5h

Graphes
TD – Séance n 4

Exercice 1 – Graphe Combien y-a-t'il de graphes simples non isomorphes avec 4 sommets ? Dessiner chacun de ces graphes.

Correction exercice 1

Nous rappelons qu'un graphe isomorphe :

Soit $G = (V, E)$ et $G' = (V, E')$. Soit f une fonction bijective. Il existe un isomorphisme de graphe entre G et G' si et seulement si $(x, y) \in E \leftrightarrow (f(x), f(y)) \in E'$.

IL y en a 11 graphes.

Fin correction exercice 1

Exercice 2 – Graphe

Calculez les paramètres n (nombre de sommets), m (nombre d'arcs ou d'arêtes), δ (degré min), Δ (degré max), D (diamètre du graphe) des graphes suivants : B_d (les arbres binomiaux de dim. d), C_n (cycle à n sommets), K_n (graphe complet à n sommets), $GR_{p\times q}$ (grille $p \times q$), $TR_{p\times q}$ (tore $p \times q$), H_d (hypercube de dim. d). Dessinez B_3, K_5, $GR_{4\times4}$, $TR_{4\times4}$, H_2, H_3, H_4.

Correction exercice 2

	n	m	δ	Δ	D
C_n	n	$n-1$	2	2	$\lceil n/2 \rceil$
K_n	n	$n(n-1)/2$	$n-1$	$n-1$	1
$GR_{p\times q}$	pq	$p(q-1)+(p-1)q$	2	4	$p+q-1$
$TR_{p\times q}$	pq	$2pq$	4	4	$(p+q-1)/2$

Fin correction exercice 2

Exercice 3 – Graphe

Soit $G(V, E)$ un graphe d'ordre n. Soient $d_1, d_2, \ldots, d_n$ les degrés du graphe. Montrer que $\sum_{i=1}^{n} d_i = 2|E|$.

Correction exercice 3

Chaque arête dans la somme est compté deux fois.

Fin correction exercice 3

Exercice 4 – Graphe Montrer que dans un graphe il y a toujours un nombre pair de sommets de degré impair.

Correction exercice 4

— Soit V_P l'ensemble des sommets à degré pair.
— Soit V_I l'ensemble des sommets à degré impair.

Il est clair $V_I \cap V_P = \varnothing$ et soit $V = V_P \cup V_I = \{v_1, v_2, \ldots, v_n\}$.

D'après l'exercice précédent, nous savons que $\sum_{i=1}^{n} d_i = 2|E| = \sum_{v_i \in V_I} d_i + \sum_{v_i \in V_P} d_i$ Donc $\sum_{v_i \in V_I} d_i = 2m - \sum_{v_i \in V_P} d_i = 2m - 2k$ donc $\sum_{v_i \in V_I} d_i$ est pair. Donc la somme des degrés impair est pair, donc il faut que le nombre de sommets impair soit pair, donc $|V_I|$ est pair.

Fin correction exercice 4

Exercice 5 – Graphe Quel est le nombre d'arêtes d'un graphe $B(V_1, V_2, E)$ biparti complet où $|V_1| = n_1$ et $|V_2| = n_2$?

Correction exercice 5

1. On a $|X| \times |Y|$.
2. Dans le pire des cas concernant le nombre d'arêtes c'est lorsque on a $|X| = n/2$ et $|Y| = n/2$. D'après 1, on a alors $|E| \leq n^2/4$. Autre méthode : $|X| = n - x, |Y| = x$ avec $x \in [1, n-1]$ et on cherche a maximiser la fonction : $f(x) = (n-x)x$.

Fin correction exercice 5

Exercice 6 – Graphe Soit $G(X_1, X_2, E)$ un graphe simple biparti d'ordre n. Montrer que $|E| \leq n^2/4$.

Correction exercice 6

Dans le pire des cas concernant le nombre d'arêtes c'est lorsque on a $|X| = n/2$ et $|Y| = n/2$. D'après 1, on a alors $|E| \leq n^2/4$. Autre méthode : $|X| = n - x, |Y| = x$ avec $x \in [1, n-1]$ et on cherche a maximiser la fonction : $f(x) = (n-x)x$.

Fin correction exercice 6

Exercice 7 – Graphe

Les nombres $\delta(G)$ et $\Delta(G)$ représentent respectivement les degrés minimum et maximum d'un graphe $G(X, E)$, où $n = |X|$ et $m = |E|$. Montrer que $\delta(G) \leq 2m/n \leq \Delta(G)$.

Correction exercice 7

Toujours la même chose : $2m = \sum_{x \in X} d(x)$ donc $\delta \leq d(x) \leq \Delta$ et $n\delta \leq 2m \leq n\Delta$

Fin correction exercice 7

Exercice 8 – Graphe

Montrer que si un graphe biparti $G(X_1, X_2, E)$ est k-régulier ($k > 0$), alors $|X_1| = |X_2|$.

Correction exercice 8

— le nombre d'arête partant de X_1 : $k|X_1|$
— le nombre d'arête entrant en X_2 : $k|X_2|$

Donc le nombr d'arcs qui sortent de X_1 et égale au nombres d'arcs qui rentrent dans X_2, donc $k|X_1| = k|X_2|$.

Fin correction exercice 8

Exercice 9 – Graphe

Montrer que tout graphe simple possède au moins deux sommets de même degré.

Correction exercice 9

Un graphe simple est un graphe sans boucle et sans multi-arête. Procédons par l'absurde ; Nous supposons que tous les degrés des sommets du graphe sont distincts $d(x_i) \neq d(x_{i+1}), i \in \{1, \ldots, n\}$

Soit i le degré de x_i. Nous savons que $d(x_i) \leq n-1, \forall i \in \{1, \ldots, n\}$, donc
— $x_n \to n-1$ (si $d(x_n) = n$ alors x_n est relié a lui-même, impossible).
— $x_{n-1} \to n-2$
— ...
— $x_2 \to 1$
— $x_1 \to 0$

Or x_n est relié à x_1, donc l'hypothèse est fausse.

Fin correction exercice 9

Exercice 10 – Graphe Un n-cube (ou hypercube de dimension n) est un graphe dont les sommets représentent les éléments de $\{0,1\}^n$ et deux sommets sont adjacents si et seulement si les n-uplets correspondants diffèrent en exactement une composante. Montrer que :

1. Le n-cube possède 2^n sommets
2. Le n-cube est n-régulier
3. Le nombre d'arêtes est $n2^{n-1}$
4. Représenter le 1-cube, le 2-cube et le 3-cube.

Correction exercice 10

1. mot de longueur n sur un alphabet $\{0,1\}$ implique qu'il y a $N = 2^n$ sommets.
2. On peut faire varier $\bar{x_i}$ de n position. Il y a donc n voisins pour chaque sommet.
3. On sait $\sum_{i=1}^{2^n} d(x_i) = 2|E| = n2^n$ et donc $|E| = n2^{n-1}$.
4. le diamètre est donné par les sommets de coordonnées 00..0 et 11.11 est donc le diamètre est $n = log_2 N$.
5.

Fin correction exercice 10

Exercice 11 – Graphe Montrer que tout arbre d'ordre n a au moins 2 sommets pendants (un sommet pendant est un sommet de degré 1).

Correction exercice 11

Un arbre est un graphe non orienté connexe sans cycle Soit p une chaîne de longueur maximum T. Soit a et b les extrémités de la chaîne P. a et b n' ont pas d'autres voisisn appartenant à P que leur voisin immédiat puisque T ne contient pas de cycle.

Fin correction exercice 11

Exercice 12 – Graphe On note $E(G)$ le nombre d'arêtes du graphe G. Montrer que

1. $\forall n, n' \in \mathbb{N}$, $E(K_n) + E(K_{n'}) < E(K_{n+n'})$.
2. Si G est un graphe simple avec n sommets et p composantes connexes, donner une borne supérieure sur le nombre d'arêtes dans G.

Correction exercice 12

1. IL est facile de voir que la propriété est vraie en comptant simplement.
2. Le graphe G admet comme configuration maximale concernant le nombre d'arêtes avec $(p-1)$ composantes connexes la configurations avec une composante du type $K_{n-(p-1)}$ et $p-1$ composantes connexes de taille une chacune et donc le nombre maximum d'arête est le nombre d'arête de $K_{n-(p-1)} = \frac{(n-(p-1))(n-p)}{2}$. Il est important de noter que si on retire un ou plusieurs sommets du graphe complet $K_{n-(p-1)}$ pour les mettre dasn une composante connexe quelqconque et que le graphe engendré par l'ajout de ce ou ces sommets est un graphe complet, alors d'après la question précédente, le nombre d'arêtes dans ce nouveau graphe est strictement inférieur au nombre d'arêtes de l'ancien graphe.

Fin correction exercice 12

Exercice 13 – Graphe Montrer qu'un graphe simple avec n sommets et plus de $\frac{1}{2}(n-1)(n-2)$ arêtes est connexe.

Correction exercice 13

Un graphe est connexe s'il n'admet qu'une seule composante connexe. Un graphe simple avec n sommets admettant $\frac{1}{2}(n-1)(n-2)$ arêtes n'étant pas connexe, admet u moins 2 composantes connexes et la structure d'après l'exercice précédent est un graphe K_{n-1} et un point isolé. Donc le graphe G avec n sommets et plus de $\frac{1}{2}(n-1)(n-2)$ est connexe.

Fin correction exercice 13

Exercice 14 – Graphe Soit G un graphe alors les trois assertions suivantes sont équivalentes :

1. $G = (X, Y, E)$ est un graphe biparti.
2. G n'a pas de cycle élémentaire impair.
3. G n'a pas de cycle impair.

Correction exercice 14

— (1) $\Rightarrow$ (2) Si $G = (X, Y, E)$ est un graphe biparti alors X et Y peuvent être coloriés respectivement en rouge et bleu (deux sommets adjacents n'ont pas la même couleur). Donc si G admet un cycle élémentaire impair $C = x_1, \ldots, x_{2n+1}, x_1$ avec $n \geq 1$ alors le sommet x_1 aura deux couleurs contradictoires, puisque $x_1 = b, x_2 = r, \ldots, x_{2n+1} = b, x_1 = r$.

— (2) $\Rightarrow$ (3) Supposons que G n'a pas de cycles élémentaires impairs, et qu'il existe un cycle impair. Soit $C = x_1, \ldots, x_{2n+1}, x_1$ un cycle non-élémentaire impair. Chaque fois qu'on trouvera deux sommets x_j et x_k avec $j < k$ et $x_j = x_k$, alors C est décomposé en deux cycles $C_1 = x_1, \ldots, x_j = x_k, \ldots, x_{2n+1}, x_1$ et $C_2 = x_j, \ldots, x_k = x_j$. En plus C_1 ou C_2 a une longueur impaire, sinon le cycle C serait de longueur paire. Avec cette décomposition il restera toujours un cycle de longueur impaire. Donc à la fin on aura un cycle élémentaire de longueur impaire, ce qui est contradictoire.

— (3) $\Rightarrow$ (1) Considérons un graphe sans cycle impair et montrons qu'il est biparti. Supposons que le graphe est connexe, sinon on traitera séparément les composantes connexes. Nous allons colorier les sommets en deux couleurs :

1. On colorie un sommet arbitraire a en bleu.
2. Si un sommet x est bleu, on colorie ses voisins en rouge. Si un sommet x est rouge, on colorie ses voisins en bleu.

Puisque G est connexe tout sommet sera colorié, un sommet ne peut avoir deux couleurs sinon il serait dans un cycle élémentaire de longueur impaire.

Fin correction exercice 14

Exercice 15 – Graphe Prouver qu'un graphe G de degré minimum $\delta \geq 2$ contient au moins un chaîne élémentaire (chaîne qui ne passe une et une fois par les sommets) de longueur (nombres d'arêtes) supérieure ou égale à δ et un cycle élémentaire de longueur supérieur ou egale à $\delta + 1$.

Correction exercice 15

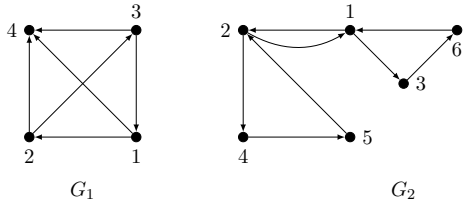


FIGURE 1 – Deux graphes G_1 et G_2.

Considérons une chaîne élémentaire de longueur maximum, et appelons a et b ses extrémités. La chaîne ne pouvant être prolongée, les voisins de a par exemple sont sur cette chaîne qui contient donc au moins le sommet et δ autres sommets, puisque a possède au moins δ voisins. La chaîne est donc de longueur au moins δ.

Considérons la même chaîne et l'arête qui lie a à un sommet c qui lui soit adjacent (un de ses voisins donc) et qui soit le plus loin possible de a sur la chaîne. L' union de cette arête $\{a, c\}$ et d ela portion de chgaîne entre a et c constitue un cycle ; or, cette portion de chaîne contient au moins $\delta + 1$ sommets, d' où le résultat annoncé.

Fin correction exercice 15

Exercice 16 – Graphe Soient les graphes $G_1(X_1, A_1)$ et $G_2(X_2, A_2)$ donnés par la figure 1 :

1. Donner $\Gamma^+(x)$ (la liste des successeurs), $\Gamma^-(x)$, $\Gamma(x)$, $d^+(x)$, $d^-(x)$ et $d(x)$, $\forall x \in X_2$.
2. Donner les matrice d'adjacence de G_1 et G_2.
3. Étant donné un graphe orienté $G(X, A)$ ayant n sommets et m arcs, sa matrice d'incidence est une matrice $n \times m$, notée $P = (p_{ie})$, telle que $p_{ie} = 1$ ssi le sommet i est origine de l'arc e, $p_{ie} = -1$ ssi le sommet i est extrémité de l'arc e, et $p_{ie} = 0$ sinon. Donner la matrice d'incidence des graphes G_1 et G_2.
4. Représenter les deux graphes par leurs listes d'adjacence (les sommets d'une liste d'adjacence sont rangés consécutivement dans un tableau).

Correction exercice 16

Il suffit d'appliquer les définitions du cours.

Fin correction exercice 16

Exercice 17 – Représentation d'un graphe par une matrice d'adjacence Le but de cet exercice est de voir les propriétés lorsqu'un graphe est donné par une matrice adjacence. On représente un graphe orienté par sa matrice d'adjacence (n, n).

1. Ecrire un algorithme pour déterminer si le graphe est sans boucle.
2. Ecrire un algorithme pour faire afficher tous les successeurs d'un sommet donné.
3. Ecrire un algorithme pour faire afficher tous les prédédecesseurs d'un sommet donné.

4. Soit G un graphe non orienté. Quel est la particularité de la matrice d'adjacence M associé au graphe G ? Quel implication concernant le stockage des données.
5. Soit G un graphe orienté. Soit M sa matrice d'adjacence. Que représente M^t où M^t désigne la matrice transposée. Appliquer les graphes G_1 et G_2 définiis précédemment.

Correction exercice 17

Algorithm 1 *Graphe sans boucle*

```
1. est-sans-boucle (M)
   for i allant de 1 à n do
     if m[i,i] = 1 then
       répondre "non" ;
       sortir ;
     end if
   end for
   répondre "oui"
```

Algorithm 2 *Affichage des successeurs*

```
2. afficher-successeurs(M, i)
   for j allant de 1 à n do
     if m[i,j] = 1 then
       écrire j ; ;
     end if
   end for
```

Algorithm 3 *Affichage des prédecesseurs*

```
3. afficher-prédecesseurs (M, i)
   for j allant de 1 à n do
     if m[j,i] = 1 then
       écrire j ; ;
     end if
   end for
```

4. La matrice est triangumaire supérieure ou inférieure. Il suffit donc de stocker que la moitié des données.
5. La matrice transposée represente le graphe G^{-1} c'est à dire si dans G il existe un arc $a \to b$ alors dans G^{-1} il existe un arc $b \to a$.

Fin correction exercice 17

Exercice 18 – Notion de base Soit la matrice binaire ou matrice d'adjacence, M associé au graphe $G = (S, U)$ orienté.

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

1. Tracer le graphe représentatif de cette matrice.
2. Donner la matrice d'incidence sommets-arcs de ce graphe
3. Calculer $M^2,\ M^3,\ M^4$. Que pouvez-vous en dire ?
4. Calculer : $A = I + M + M^2 + M^3 + M^4$. Interpréter A.
5. Appliquer l'algorithme de Roy Warshall. Que constatez-vous ?

Correction exercice 18

1. facile, on $x_1 \to x_2 \to x_3 \to x_5,\ x_1 \to x_4 \to x_5$ et $x_4 \to x_3$. On pose $u_1 = (x_1, x_2),\ u_2 = (x_1, x_4),\ u_3 = (x_2, x_3),\ u_4 = (x_3, x_5),\ u_5 = (x_4, x_3),\ u_6 = (x_4, x_5)$.
2. La matrice d'incidence sommet-arc est donné par la relation suivante :

$$a_{iu} = \begin{cases} 0 & i, u \notin U \quad \text{il n'y a pas d'arc connecté au sommet } i \\ 1 & i, u \in U \quad \text{l'arc } u \text{ a pour extrémité initiale le sommet } i \\ -1 & i, u \in U \quad \text{l'arc } u \text{ a pour extrémité terminale le sommet } i \end{cases}$$

On trouve donc

	u_1	u_2	u_3	u_4	u_5	u_6
x_1	1	1	0	0	0	0
x_2	−1	0	1	0	0	0
x_3	0	0	−1	1	−1	0
x_4	0	−1	0	0	1	1
x_5	0	0	0	−1	0	−1

3.
$$M^2 = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Chaque élément m_{ij} de la matrice M^k appartenant à l'ensemble des réels représente le nombre de chemin de longueur k, du sommet i au sommet j. De M^2 on déduit qu'il existe :
— Deux chemins de longueurs 2 entre x_1 et x_3 et un chemin de longueur 2 entre x_1 et x_5,
— Un chemin de longueur 2 entre x_2 et x_5,
— Un chemin de longueur 2 entre x_4 et x_5.

$$M^3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

De M^3 on en déduit que :
— 2 chemins de longueur 3 entre x_1 et x_3,

$$M^4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Il n'y a aucun chemin de longueur 4.

4. A représente la fermeture transitive du graphe, et

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

5. On retrouve le même résultat que précédemment. Voici le calcul

$$C_1 = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$C_2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$C_3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Nous avons $C_3 = C_4 = C_5$. Pour $k = 5$, c'est normal car le sommet 5 est un puits.

Fin correction exercice 18

Exercice 19 – Notion de base bis Dans le graphe G, représenté par sa matrice d'incidence sommets sommets.

	A	B	C	D	E	F
A	0	1	0	0	0	1
B	0	1	1	0	1	0
C	0	0	0	1	0	0
D	0	1	1	0	1	0
E	0	1	0	0	1	1
F	1	0	1	0	0	0

1. Enumérer la liste des successeurs et des prédécesseurs de tous les sommets.
2. Donner les demi-degré intérieurs et extérieurs de chaque sommet.
3. Donner un exemple de chemin simple mais non élémentaire.
4. Existe-il un circuit hamiltonien dans G ?
5. Tracer le graphe non orienté déduit de G.
6. G est-il connexe ? Fortement connexe ?

Correction exercice 19

1. Voir la question 2.
2. $\Gamma^+(A) = \{B, F\}$, $\Gamma^-(A) = \{F\}$, $\Gamma^+(B) = \{B, C, E\}$, $\Gamma^-(B) = \{A, B, D, E\}$, $\Gamma^+(C) = \{D\}$, $\Gamma^-(C) = \{B, D, F\}$, $\Gamma^+(D) = \{B, C, E\}$, $\Gamma^-(D) = \{C\}$, $\Gamma^+(E) = \{B, E, F\}$, $\Gamma^-(E) = \{B, D, E\}$, $\Gamma^+(F) = \{A, C\}$, $\Gamma^-(F) = \{A, E\}$.
3. Ceux-ci se calculent immédiatement à partir de la matrice d'incidence. Les demis-degrés extérieurs (resp. intérieurs), d^+ (resp. d^-) du sommet X = somme des 1 de la ligne du sommet X (resp. des colonnes).

	A	B	C	D	E	F	d^+
A	0	1	0	0	0	1	2
B	0	1	1	0	1	0	3
C	0	0	0	1	0	0	1
D	0	1	1	0	1	0	3
E	0	1	0	0	1	1	3
F	1	0	1	0	0	0	2
d^-	1	4	3	1	3	2	

4. Un chemin est une suite d'arcs dont l'extrémité initiale de l'arc correspond à l'extrémité terminale de l'arc précédent et dont l'extrémité initiale correspond à l'extrémité terminale de l'arc suivant. Exception faite naturellement pour l'arc initial qui n'a pas de précédent et l'arc terminal qui n'a pas de suivant.

   Un chemin est simple s'il ne passe qu'une fois par chacun de ses arcs du chemin. Il est élémentaire s'il ne passe qu'une fois par chacun de ses arcs du chemin. Un chemin simple mais non élémentaire ne passe qu'une fois par chacun des arcs mais peut rencontrer plusieurs fois le même sommet. Un chemin peut être simple et non élémentaire s'il comprend un circuit. Le chemin (A, B, C, D, B, E, E, F) formé des arcs $\{(A, B),\ (B, C),\ (C, D),\ (D, B),\ (B, E),\ (E, F)\}$ est simple mais no élémentaire.

5. Un circuit hamiltonien est un chemin hamiltonien qui se referme sur lui-même. Un circuit hamiltonien est un chemin élémentaire il passe une fois et une seule par chacun des sommets du graphe. Le chemin (A, B, C, D, E, F, A) est un circuit hamiltonien dans G.
6. Il suffit de retirer l'orientation de chacun des arcs sur le graphe initial.
7. Un graphe $G = (X, U)$ est connexe si pour tout couple de sommet $(x_i, x_j) \in X$, il existe une chaîne les reliant. IL est fortement connexe si pour tout coupole de sommet $x_i, x_j) \in X$, l existe au moins un chemin reliant de x_I à x_j et au moins un chemin reliant x_j à x_i. Le graphe est connexe et même fortement connexe puisque nous avons trouvé un circuit hamiltonien (circuit passant par les sommets une fois et une seule).

Fin correction exercice 19

Exercice 20 – Fermeture Transitive d'un graphe

Soit C la matrice d'adjacence de G. On veut calculer la matrice d'adjacence C^* de G^* la fermeture transitive du graphe G. Supposons que $C_k[i, j]$ représente l'existence d'une chaîne de i à j passant par des sommets inférieurs ou égaux à k.

1. Quelles sont les conditions d'existence d'une chaîne de i à j passant par des sommets inférieurs ou égaux à k ?
2. Ecrire un algorithme qui calcule la fermeture transitive d'une graphe G représenté par une matrice d'adjacence.
3. Quelle est la complexité de cet algorithme dans un graphe orienté ?
4. Donner un exemple de matrice M pour laquelle n éléments seulement sont égaux à 1 et telle que tout élément de M^* est égal à 1.
5. Donner un exemple de graphe orienté à matrice M dans laquelle tous les éléments, sauf $n - 1$ sont égaux à 1, mais tel qu'il existe au moins un élément de M^* différent de 1.
6. Quel est la fermeture transitive d'un graphe non orienté ?

Correction exercice 20

1. — Il existe une chaîne de i à j passant par des sommets inférieurs ou égaux à $k - 1$
   — Il existe une chaîne de i à k passant par des sommets inférieurs ou égaux à $k - 1$ et une chaîne de k à j passant par des sommets inférieurs ou égaux à $k - 1$.
2. cf. plus loin.
3. la complexité est clairement en $\Theta(n^3)$.
4. En fait il s'agit de trouver des graphes connexes orientés avec exactement $n - 1$ arêtes.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

5. Ici on a toujours un graphe orienté connexe, mais de l'un des sommets on ne peut aller nulle part c'est un puits.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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

int C[7][7]={
  {0,1,0,0,0,0,0},
  {0,0,1,0,0,0,0},
  {0,0,0,1,0,0,0},
  {0,0,0,0,1,0,0},
  {0,0,0,0,0,1,0},
  {0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0}};
int A[7][7];

void warshall()
{
  int i,j,k;

  for(i=0;i<7;i++)
    for(j=0;j<7;j++)
      A[i][j]=C[i][j];
  for(k=0;k<7;k++)
    for(i=0;i<7;i++)
      for(j=0;j<7;j++)
        A[i][j] = (A[i][j] || (A[i][k] && A[k][j]));
}

void Affichage()
{
  int i,j;

  for(i=0;i<7;i++)
    {
      for(j=0;j<7;j++)
        printf(" %d",A[i][j]);
      printf("\n");
    }
}

main()
{
```

```
    warshall();
    Affichage();
  }
```

6. C'est le graphe complet.

Fin correction exercice 20

Exercice 21 – Composantes fortement connexes

	A	B	C	D	E	F	G	H	J	K	L
A	0	0	0	1	0	0	0	0	0	0	0
B	1	0	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	1	0	0	0	0	0
D	0	0	1	0	0	0	1	0	0	0	0
E	0	0	0	0	1	0	0	0	1	0	0
F	0	0	0	0	0	0	1	0	1	1	0
G	0	0	0	0	0	1	0	1	0	0	0
H	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	0
K	0	0	0	0	0	0	0	0	0	0	1
L	0	0	0	0	0	0	0	0	1	0	0

1. Décomposer, le graphe défini par sa matrice d'adjacence, en composantes fortement connexes et les ordonner en utilisant la fermeture transitive et en utilisant un parcours en profondeur.

Correction exercice 21

1. Un graphe est fortement connexe si et seulement si s'il n'a qu'une seule composante connexe. Soit $C_x = \hat{\Gamma}_x \cap \hat{\Gamma}_x^{-1}$ où $\hat{\Gamma}_x$ désigne la fermeture trasi tive associé au sommet x dans le graphe G et $\hat{\Gamma}_x^{-1}$ désigne la fermeture transistive associé au sommet x dans le graphe G^{-1} où G^{-1} désigne le symétrique de G. Nous allons raisonner sur les matrices. Donc l'ensemble des composantes fortement connexes s'obtiendra en calculant $C_G = \hat{M} \cap \hat{M}^t$ où $\hat{M}$ représente la matrice de la fermeture transitive pour le graphe G et $\hat{M}^t$ la fermeture transitive pour le graphe G^{-1}.

Donc,

$\hat{M} =$

	A	B	C	D	E	F	G	H	J	K	L
A	1	1	1	1	1	1	1	1	1	1	1
B	1	1	1	1	1	1	1	1	1	1	1
C	1	1	1	1	1	1	1	1	1	1	1
D	1	1	1	1	1	1	1	1	1	1	1
E	0	0	0	0	1	0	0	0	1	1	1
F	0	0	0	0	0	1	1	1	1	1	1
G	0	0	0	0	0	1	1	1	1	1	1
H	0	0	0	0	0	0	0	1	0	0	0
J	0	0	0	0	0	0	0	0	1	1	1
K	0	0	0	0	0	0	0	0	1	1	1
L	0	0	0	0	0	0	0	0	1	1	1

$\hat{M}^t =$

	A	B	C	D	E	F	G	H	J	K	L
A	1	1	1	1	0	0	0	0	0	0	0
B	1	1	1	1	0	0	0	0	0	0	0
C	1	1	1	1	0	0	0	0	0	0	0
D	1	1	1	1	1	1	1	1	1	1	1
E	1	1	1	1	1	0	0	0	0	0	0
F	1	1	1	1	0	1	1	0	0	0	0
G	1	1	1	1	0	1	1	0	0	0	0
H	1	1	1	1	0	1	1	1	0	0	0
J	1	1	1	1	1	1	1	0	1	1	1
K	1	1	1	1	1	1	1	0	1	1	1
L	1	1	1	1	1	1	1	0	1	1	1

et donc

$\hat{M}^t \cap \hat{M} =$

	A	B	C	D	E	F	G	H	J	K	L
A	1	1	1	1	0	0	0	0	0	0	0
B	1	1	1	1	0	0	0	0	0	0	0
C	1	1	1	1	0	0	0	0	0	0	0
D	1	1	1	1	0	0	0	0	0	0	0
E	0	0	0	0	1	0	0	0	0	0	0
F	0	0	0	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	1	0	0	0	0
H	0	0	0	0	0	0	0	1	0	0	0
J	0	0	0	0	0	0	0	0	1	1	1
K	0	0	0	0	0	0	0	0	1	1	1
L	0	0	0	0	0	0	0	0	1	1	1

Important : $\hat{M}^t$ est la matrice symétrique de $\hat{M}$, on n'a pas besoin de calculer G^{-1} et effectué la fermeture transitive. Les composantes fortement connexes, C_i correspondent aux sous-matrices carrées correspondant à $C_G = \hat{\Gamma}(X) \cap \hat{\Gamma}^{-1}(X) = \hat{M} \cap \hat{M}^t$. Ici nous avons $C_1 = \{A, B, C, D\},\ C_2 = \{E\},\ C_3 = \{F, G\},\ C_4 = \{H\},\ C_5 = \{J, K, L\}$. Nous avons cinq composantes fortement connexes.

2. Les algorithmes de parcours dans un graphe permettent de composer un algorithme simple calculant la composante fortement connexe d'un sommet s, pour un graphe orienté $G = (V, E)$. Un sommet x appartiendra à la classe de s s'il existe un chemin de s à x et un autre de x à s. On obteint simplement les sommets accessibles par des chemins d'origine s (descendant de s) par une exploration en largeur ou en profondeur, coutant $O(m)$.

Trouver les ancêtres de s revient à faire une exploration en $O(m)$ dans le graphe inverse $H = G^{-1}$. Les sommets de la classe de s sont ceux atteints par les deux explorations.

Fin correction exercice 21

On considère un graphe simple $G = (S, A)$ orienté et sans circuit. On appelle numérotation topologique du graphe G une bijection $r : S \to \{1, \ldots, n\}$ qui possède la propriété suivante : si $(x, y) \in A$ alors $r(x) < r(y)$.

1. Soit x un sommet sans successeur de G. On note $G\backslash\{x\}$ le"sous-graphe de G construit en retirant le sommet x et tous les arcs incidents à x.
   Montrer que si r' est une numérotation topologique de $G\backslash\{x\}$ alors la numérotation définie par :
   $$r(x) = n \text{ et } \forall y \in S,\ y \neq x\ r(x) = r'(y)$$
   est une numérotation topologique de G. En déduire qu'un graphe sans circuit admet une numérotation topologique.
   Nous supposons maintenant que les sommets de G sont numérotés de manière topologique et l'on identifie chaque sommet avec son numéro : un sommet x tel que $r(x) = i$ sera appelé sommet i. On notera $\mathcal{C}_k(i, j)$ l'ensemble des chemins du sommet i au sommet j dont tous les sommets intérieurs (s'ils existent) sont des numéros inférieurs ou égal à k. $\mathcal{C}_0(i, j)$ est l'ensemble des chemins d'intérieur vide de i à j. M^k désignera la matrice booléenne $n \times n$ définie par : $M^k[i, j] = 1$ si $\mathcal{C}_k(i, j) \neq \varnothing$; $M^k[i, j] = 0$ sinon. Par convention, on pose $M^[i, j] = 1$ pour tout i.
2. Montrer que pour $k \in \{1, \ldots, n\}$
   (a) Si $j < i,\ \mathcal{C}_k(i, j) = \varnothing$
   (b) Si $k < i,\ \mathcal{C}_k(i, j) = \mathcal{C}_0(i, j)$
   (c) Si $k > j,\ \mathcal{C}_k(i, j) = \mathcal{C}_{j-1}(i, j)$.
3. En déduire que pour tout $k > 0,\ M^k[i, j] = M^{k-1}[i, j]$ si $I \geq k$ ou $j \leq k$.
4. Indiquer un algorithme de calcul de fermeture transitive de G inspiré de l'algorithme de Roy-Warshall qui tienne compte de la propriété de la question précédente et prouver sa validité.
5. Calculer la complexité de l'algorithme précédent.

Correction exercice 21

1. Soient x un sommet sans successeur de G et r' une numérotation topologique de $G\backslash\{x\}$. Soit r définie par :
   $$r(x) = n \text{ et } \forall y \in S,\ y \neq x\ r(x) = r'(x)$$
   r représente bien une bijection de l'ensemble des sommets dans $\{1, \ldots, n\}$ car r' associe à tout sommet de $G\backslash\{x\}$ un numéro unique dans $\{1, \ldots, n-1\}$.
   Soit $(u, v) \in A$ un arc de G.
   Si $v = x$ alors $r(u) = r'(u) \leq n - 1 < n = r(v)$.

Sinon (u, v) est un arc de $G\backslash\{x\}$ et $r(u) = r'(u) < r'(v) = r(v)$ puisque r' est une numérotation topologique de $G\backslash\{x\}$.
Montrons alors la propriété par récurrence sur n.
— Si $n = 1$ alors tout graphe sans boucle à un sommet est sans circuit et numéroter le sommet définit une numérotation topologique.
— Si G est un graphe sans circuit possédant n sommets, on considéré un sommet x sans successeur de G, $G\backslash\{x\}$ possède $n - 1$ sommets et pas de circuit et par hypothèse de récurrence, il possède alors une numérotation topologique r'. On peut donc construire une numérotation topologique de G.

2. L'ordre topologique des sommets possède une propriété remarquable : si $\mu(x_1, \ldots, x_n)$ est un chemin de G, on a $r(x_1) < r(x_2) < \ldots < r(x_n)$. Il suffit, pour trouver ce résultat, d'écrire la relation d'ordre sur les numéros des extrémités de chacun des arcs du chemin. On en déduit alors que :
   — si $j < i$ il n'y a aucun chemin de i à j dan G,
   — Si $i < j$ tout chemin de i à j est soit d'intérieur vide, soit possède des sommets intermédiaires dont les numéros sont compris entre i et j. En particulier, si $k < i$ aucun chemin de i à j ne peut comporter des sommets intermédiaires de numéros inférieurs ou égaux à k. Ils sont donc d'intérieur vide. D'où $\mathcal{C}_k(i, j) = \mathcal{C}_0(i, j)$:
   — si $k > j$, pour les mêmes raisons, un chemin de i à j ne peut comporter des sommets intermédiaires de numéros supérieurs à $j - 1$. Par conséquent, $\mathcal{C}_k(i, j) = \mathcal{C}_{j-1}(i, j)$.
3. Si $i < k$, on d'après la question précédente, $\mathcal{C}_k(i, j) = \mathcal{C}_0(i, j)$. Si $i = k$ on a également cette égalité car tous les chemins de k à j de sommets intermédiaires inférieurs ou égaux à k, sont d'intérieur vide. On en déduit que $\mathcal{C}_k(i, j) = \mathcal{C}_{k-1}(i, j)$. Comme $M^k[i, j] = 1$ si et seulement si $\mathcal{C}_k(i, j)$ n'est pas vide, on en déduit immédiatement que $M^k[i, j] = M^{k-1}[i, j]$.
   De même, si $k > j$, on $\mathcal{C}_k(i, j) = \mathcal{C}_{j-1}(i, j)$. En particulier, on a $\mathcal{C}_k(i, j) = \mathcal{C}_{k-1}(i, j)$. Lorsque $k = j$ comme j ne peut être un sommet intermédiaire sur un chemin de i à j car G ne possède pas de circuit, on aura également $\mathcal{C}_j(i, j) = \mathcal{C}_{j-1}(i, j)$. Ainsi, $M^k[i, j] = M^{k-1}[i, j]$.
4. On suppose que la matrice M est initialisée avec les valeurs contenues dans la matrice d'adjacence du graphe. Seules les valeurs sur la diagonale seront égales à 1. L'algorithme 4 qui modifie la matrice M est définie par :

Algorithm 4 Algorithme donnant la fermeture transitive

```
for k = 1 à n do
  for i = 1 à k − 1 do
    for j = k + 1 à n do
      M[i, j] = M[i, j] ou (M[i, j] et M[k, j])
    end for
  end for
end for
```

On admet que l'algorithme de Roy-Warshall est valide. Cet algorithme est basé sur l'équation de récurrence suivante :

$$M^k[i, j] = M^{k-1}[i, j] \text{ ou } (M^{k-1}[i, k] \text{ et } M^{k-1}[k, j])$$

qui reste valable dans le cas d'un graphe sans circuit. On lui ajoute simplement les cas particuliers décrits à la question précédente ($k \leq i$ ou $j \leq k$). Pour ces cas particuliers, il

est inutile d'effectuer un calcul pour obtenir $M[i,j]$ connaissant $M^{k-1}[i,j]$. Par ailleurs, on sait que dans l'algorithme de de Roy-Warshall, à la fin de l'itération $k-1$ de la boucle externe, la matrice M vaut M^{k-1}. A la fin de l'itération k, pour $i < k$ et $j > k$ d'après l'algorithme de Roy-Warshall, on a bien $M[i,j] = M^k[i,j]$ et sinon $M[i,j] = M^{k-1}[i,j] = M[i,j]$ d'après la question précédente.

5. L'algorithme de fermeture transitive pour les graphe sans circuit est similaire à l'algorithme de Roy-Warshall. La différence réside dans les bornes des indices de boucles, c'est à dire le nombre d'itérations. Pour cet algorithme, le nombre d'itération N est

$$\begin{aligned} N &= \sum_{k=1}^{n}(k-1)\times(n-k) \\ &= n\times\sum_{k=1}^{n}(k-1) - \sum_{k=1}^{n}k^2 + \sum_{k=1}^{n}k \\ &= (n+1)\sum_{k=1}^{n}k + n\sum_{k=1}^{n}(-1) - \sum_{k=1}^{n}k^2 \\ &= n + (n+1)\sum_{k=1}^{n-1}k - \frac{n(n+1)(2n+1)}{6} \\ N &= n + \frac{n(n+1)(n-4)}{6} \end{aligned}$$

Fin correction exercice 21

Université de Montpellier 2017/2018
HMIN329 – Complexité et algorithmes Durée: 3h

Arbres
TD – Séance n 5

Exercice 1 – Propriété des arbres

Soit T_1 et T_2 deux arbres de recouvrement d'un graphe connexe G. Montrer que T_1 peut être transformé en T_2 par une séquence d'arbres intermédiaires.

Correction exercice 1

Soit $E(T_1)$ et $E(T_2)$ les arêtes de T_1 et de T_2. Notons $d(T_1, T_2) = |E_2 - E_1|$. Supposons que $d(T_1, T_2) > 0$. Soit $e \in E(T_2) - E(T_1)$. On ajoute cette arête à T_1. On obtient donc un graphe avec un cycle. Comme T_2 est un arbre l'une des arêtes de ce cycle n'appartient pas à T_2. On obtient donc un arbre T_1' tel que $d(T_1, T_2) > d(T_1', T_2)$. En continuant cette opération, on obtiendra donc un arbre T tel que $d(T, T_2) = 0$.

Fin correction exercice 1

Exercice 2 – Propriété des arbres

Soit $\{v_1, v_2, \ldots, v_n\}$ des points fixés et $\{d_1, d_2, \ldots, d_n\}$ des nombres entiers donnés tels que $\sum_{i=1}^{n} d_i = 2n-2,\ d_i \geq 1$. Montrer que le nombre d'arbres sur l'ensemble $\{v_1, v_2, \ldots, v_n\}$ où le sommet v_i admet pour degré $d_i,\ i = 1, \ldots, n$ est $\frac{(n-2)!}{(d_1-1)!\ldots(d_n-1)!}$

Correction exercice 2

— Par induction sur n. Pour $n = 2$ le résultat est trivial. Sachant que $\sum_{i=1}^{n} d_i = 2n-2 < 2n$, on a $d_1 = d_2 = 1$.

— Enlevons v_n un sommet pendant. Le sommet v_n est adjacent à un des sommets $v_j, 1 \leq j \leq n-1$ et le retrait de v_n propose un nouvel arbre sur les sommets $\{v_1, \ldots, v_{n-1}\}$ avec pour degré $d_1, \ldots, d_{j-1}, d_j - 1, d_{j+1}, \ldots d_{n-1}$. Réciproquement, si on considère un arbre sur les sommets $\{v_1, v_2, \ldots, v_{n-1}\}$ avec $d_1, \ldots, d_{j-1}, d_j - 1, d_{j+1}, \ldots d_{n-1}$ alors en joignant v_j à v_n on obtient un arbre $\{v_1, v_2, \ldots, v_n\}$ avec degrés $\{d_1, d_2, \ldots, d_n\}$. Le nombre d'arbre sur $\{v_1, v_2, \ldots, v_{n-1}\}$ avec $d_1, \ldots, d_{j-1}, d_j - 1, d_{j+1}, \ldots d_{n-1}$ est $\frac{(n-3)!}{(d_1-1)!\ldots(d_{j-1}-1)!(d_j-2)!(d_{j+1}-1)!\ldots(d_n-1)!} = \frac{(n-3)!(d_j-1)}{(d_1-1)!\ldots(d_n-1)!}$. Ceci est aussi valide si $d_j = 1$ sachant que c'est 0. Ainsi, le nombre d'arbres sur $\{v_1, v_2, \ldots, v_n\}$ avec $\{d_1, d_2, \ldots, d_n\}$ est $\sum_{j=1}^{n-1} \frac{(n-3)!(d_j-1)}{(d_1-1)!\ldots(d_n-1)!} = (\sum_{j=1}^{n-1}(d_j-1))\frac{(n-3)}{(d_1-1)!\ldots(d_{j-1}-}$ $= \frac{(n-2)(n-3)!}{(d_1-1)!\ldots(d_1-1)!\ldots(d_n-1)!}$. Il est important de noter que $\sum_{j=1}^{n-1} d_j = 2n-3$ car $d_n = 1$ et donc $(d_n - 1)! = 1(pasdercurrence)$

Fin correction exercice 2

Exercice 3 – Propriété des arbres

Montrer que le nombre de tous les arbres sur n points est n^{n-2} (formule de Cayley).

Correction exercice 3

$$\sum_{d_1,d_2,\ldots,d_n \geq 1, d_1+d_2+\ldots+d_n=2n-2} \frac{(n-2)!}{(d_1-1)!\ldots(d_n-1)!} = \sum_{k_1,k_2,\ldots,k_n \geq 0, k_1+k_2+\ldots+k_n=n-1} \frac{(n-2)!}{k_1!\ldots(k_n)!} = (1+1+\ldots+1)^{n-2}.$$

Fin correction exercice 3

Exercice 4 – Propriété des arbres

Soient $1 \leq d_1 \leq d_2 \leq \ldots \leq d_n$ des entiers. Nous allons prouver qu'il existe un arbre avec des degrés $d_1, d_2, \ldots, d_n$ si et seulement si $d_1 + d_2 + \ldots + d_n = 2n - 2$.

1. Montrer que si il existe un arbre avec des degrés $d_1, d_2, \ldots, d_n$ alors $d_1 + d_2 + \ldots + d_n = 2n - 2$.
2. On suppose maintenant que $d_1 + d_2 + \ldots + d_n = 2n - 2$. La preuve est par récurrence sur n. Que pensez-vous du cas $n = 2$?
3. On suppose $n \geq 3$. Quelle est forcément la valeur de d_1 ? Montrer que $d_n > 1$.
4. Montrer qu'il existe un arbre avec les degrés $d_2, d_3, \ldots, d_{n-1}, d_n - 1$.
5. Conclure.

Correction exercice 4

1. Un arbre avec n sommets admet $n - 1$ arêtes.
2. On suppose que $d_1 + d_2 + \ldots + d_n = 2n - 2$. Pour $n \leq 2$ c'est évident.
3. soit $n \geq 3$. On a $d_1 = 1$ puisque $d_1 \geq 2$ impliquerait $d_1 + d_2 + \ldots + d_n \geq 2n > 2n - 2$. Aussi $d_n > 1$ puisque $d_n = 1$ impliquerait que $d_1 + d_2 + \ldots + d_n = n < 2n - 2$. Puisque $d_2 + \ldots + (d_n - 1) = 2(n-1) - d_1 - 1 = 2n - 4 = 2(n-2)$. Iy a un arbre T avec des degrés $d_2, \ldots, d_{n-1}, d_n - 1$. Ajouter un nouveau sommet et le relie au sommet de T degré $d_n - 1$ on obtient un arbre de degré désiré.

Fin correction exercice 4

Exercice 5 – Arbre couvrant

Soit G le graphe de la figure 1.

1. Indiquer dans quel ordre les procédures d'exploration en largeur et en profondeur visitent les sommets de G en partant du sommet a.
2. Donner toutes les possibilités et les arbres couvrants associés.

Correction exercice 5

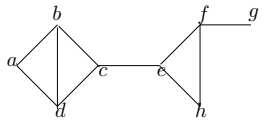
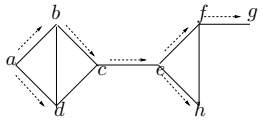
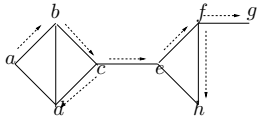


FIGURE 1 – Exemple

1. La solution est donné par le graphe 2 (il y a deux arbres en largeur)



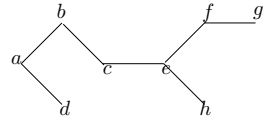
En largeur



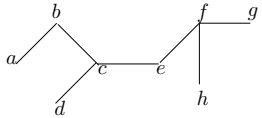
En profondeur

FIGURE 2 –

2. La solution est donné par le graphe 3 (il y a 8 arbres en profondeur)



Arbre en largeur



Arbre en profondeur

FIGURE 3 –

Fin correction exercice 5

Exercice 6 – Propriété des arbres

Prouver que si tous les poids sur les arêtes sont différents, il y a unicité de l'arbre couvrant de poids minimum.

Correction exercice 6

Suppons qu'il existe deux arbre T et T' tel que $T \neq T'$. (on tous les poids poids différents).

$T = \{e_1, \ldots, e_i, f_{i+1}, \ldots, f_{n-1}\}$ et $T' = \{e_1, \ldots, e_i, g_{i+1}, \ldots, g_{n-1}\}$. $w(T)$ et $w(T')$ sont de poids minimum. avec $w(f_{i+1}) < \ldots < w(f_{n-1})$ et $w(g_{i+1}) < \ldots < w(g_{n-1})$.

Supposons qu'il existe au moins une arête qui différent c'est à dire $w(f_{i+1}) < w(g_{i+1})$.

Puisque $f_{i+1} \notin T'$ et T' est un arbre alors $T' \bigcup \{f_{i+1}\}$ contient un cycle C. Le cycle $C \notin T$, donc $\exists e' \in C$ tel que $e' \in T'$ et $e' \notin T$. (car T est un arbre) Ainsi $e' \in \{g_{i+1}, \ldots, g_{n-1}\}$ Soit $T'' = T' \bigcup \{f_{i+1}\} - \{e'\}$.

— T'' ne contient pas de cycle : en effet ce cycle ne pouvant être dans T' serait dans $T' \bigcup \{f_{i+1}\}$, donc serait C ceci est impossible puisque $e' \notin T''$.

— T'' est donc un arbre couvrant et $w(T'') = w(T') + w(f_{i+1}) - w(e')$. Or $w(f_{i+1}) < (g_{i+1}) \leq w(e')$ et donc $w(T'') < w(T')$ contradiction.
faire figure.

Fin correction exercice 6

Exercice 7 – Propriétés générales

Soit $K_n(V)$ un graphe complet sur n sommets $V = \{x_1, x_2, \ldots, x_n\}$, où les arêtes sont munies d'un poids w vérifiant $w(xy) + w(yz) \geq w(xz)$ pour tous $x, y, z \in V$. Le but de cet exercice est de montrer qu'un cycle hamiltonien de poids minimum dans K_n a un poids au plus égal à $2w(T)$, où T est un arbre couvrant de K_n de poids minimum. On procédera par récurrence sur n le nombre de sommets.

1. Montrer que le résultat est vrai lorsque $n = 3$.
2. Soit T un arbre couvrant optimal de K_n et x_n un sommet pendant de T. Soit $T' = T - x_n$ l'arbre obtenu à partir de T en enlevant le sommet x_n. Montrer que T' est un arbre couvrant optimal de $K_{n-1}(V \setminus \{x_n\})$.
3. Soit c' un cycle optimal hamiltonien de K_{n-1}, et soit $c = c' - [a, b] \cup \{[a, x_n], [b, x_n]\}$, a et b étant deux sommets consécutifs du cycle c' (on peut supposer que l'arête $[a, x_n]$ est de poids la plus faible). Montrer que le cycle c vérifie $w(c) \leq 2w(T)$. Conclure.

Correction exercice 7

1. Par récurrence, $n = 3$ Supposons que $w(xy) \leq w(yz) \leq w(xz)$ et l'arbre est donné par les arête xy, yz. On a alors $w(cycle\ hamiltonien) = w(T) + w(xz) \leq 2w(T)$;
2. Supposns la propriété vraie pour $n - 1$. Soit T un arbre couvrant optimal de G et x_n un sommet pendant (on sait qu'il en existe un) Soit $T' = T - x_n$. T' est un arbre couvrant(pas forcément optimal) de K_{n-1} on a donc $w(T') \geq w(arbre\ couvrant\ optimal)$. T' est un arbre optimal, en effet si $\exists T''$ tel que $w(T'') < w(T')$ alors $w(T'') + w(ax_n) < w(T') + w(ax_n) = w(T)$ donc $w(T)$ ne serait un acpm de K_n.
3. Considérons un cycle optimal hamiltonien de K_{n-1} noté c' Soit $c = c' - ab \bigcup \{ax_n, bx_n\}$ un cycle sur K_n. On alors $w(c) = w(c') - w(ab) + w(ax_n) + w(x_nb) \leq 2w(T') + 2w(ax_n) + w(x_nb) - w(ab) - w(ax_n) \leq 2w(T)$ A fortiori, c'est vrai pour un cycle hamiltonien optimal.

Fin correction exercice 7

Exercice 8 – Propriétés des arbres

1. Soit e une arête de poids minimal dans un graphe G. Montrer que e appartient à tous les arbres couvrant minimaux de G.
2. Est-ce encore vrai pour deux arêtes, c'est à dire s'il existe deux arêtes $e,\ f$ telles que $w(e) = w(f)$ et pour tout $u \in E,\ u \neq e$ et $u \neq f,\ w(e) < w(u)$?
3. Est-ce que cela est vraie pour trois arêtes ?
4. Supposons que les arêtes admettent de valeurs deux à deux distinctes. Soit C un cycle quelconque dasn G, et soit $e = (v, w)$ l'arête de poids la plus grande appartenant à C Montrer que e ne peut appartenir à aucun arbre couvrant de poids minimum.

Correction exercice 8

1. Il est clair que cette arête sera incluse dans l'arbre couvrant minimal. EN effet, si cette arête n'est pas incluse dans l'arbre alors on l'inclut e(on obtient donc un graphe avec un cycle) et on enlève une arête quelconque. Donc le graphe résultant et un arbre de poids inférieur au premier. Impossible. Soit e tel que $w(e) = \min_{e' \in E} w(e')$. Soit $T = ACPM(G)$ tel que $e \notin T$. Soit $T' = T \cup \{e\} \backslash \{e'\}$ avec $e \neq e'$ et $w(e') < w(e)$. Ceci implique que $w(T') = w(T) + w(e) - w(e')$ alors $w(T') < w(T)$. Ainsi $e \in C5S, \bar{S})$ et donc e vérifie la condition de la coupe minimale.
2. C'est vrai aussi, car il n'y a pas de cycle possible avec les deux arêtes, mais a trois ce n'est plus possible. Même chose que précédemment mais c'est la seconde arête que nous rajoutons (la première fait forcément partie de l'arbre). $T' = T \cup \{f\} \backslash \{e'\}$ tel que $e' \neq f$ et $e' \neq e$.
3. C'est non, car on peut tomber sur un triangle.
4. De manière classique, si cette arête ee appartient à un ACPM, alors en prenant une autre arête e', et en faisant toujours la même opération on obtient un arbre de pins plus faible.

Fin correction exercice 8

Exercice 9 – Second arbre couvrant

1. Nous souhaitons déterminer un arbre couvrant T qui minimise la fonction objectif $[\sum_{(i,j) \in T} (c_{ij})^2]^{1/2}$. Quelle solution utiliseriez-vous ?
2. Soit $[p, q]$ l'arête de coût minimal dans le graphe G. Montrer que $[p, q]$ appartient à quelques arbre couvrant de poids minimum de G. L'arête $[p, q]$ appartient-elle à tous les arbres couvrants de poids minimum ?
3. Montrer que le problème du sous-graphe sans cycle ayant un poids maximum dans un graphe non orienté avec des arêtes de valuations positives doit être un arbre couvrant.
4. **Difficile :** Dans un graphe G, la contraction d'un arête $e = [u, v]$ est le remplacement dans G de deux sommets u et v par un sommet uv et les arêtes incidentes à u et v sont maintenant incidente au sommet uv. Le graphe résultant est noté $G.e$. Donner un exemple de graphe contracté. Soit $\tau(G)$ le nombre d'arbres couvrants d'un graphe G simple (sans boucle). Montrer que si $e \in E$ alors $\tau(G) = \tau(G - e) + \tau(G.e)$.
5. Comment peut-on modifier les algorithmes de Kruskal et de Prim afin de résoudre le problème de l'arbre couvrant de poids maximum ?

6. Soit T un arbre couvrant. Pour toutes paires de sommets $[i, j]$, nous notons par $\beta[i, j]$ l'arête de poids le plus faible parmi toutes les arêtes joignant les sommets i et j dans l'arbre T . Montrer comment calculer $\beta[i, j]$ pour n'importe quelle paire de sommets en temps $O(n^2)$?
7. **Analyse de sensibilité :**
   Soit T^* un arbre couvrant de poids minimum d'un graphe $G = (N, A)$. Pour n'importe quel arête $[i, j] \in A$, nous définissons un intervalle de coût comme étant l'ensemble de valeurs de c_{ij} pour lesquels T^* continuent à être un arbre couvrant de poids minimum.
   (a) Décrire une méthode efficace pour déterminer l'intervalle de coût pour une arête $[i, j]$. (Aide : considérer deux cas : quand $[i, j] \in T^*$ et quand $[i, j] \notin T^*$ et utiliser les conditions de la coupe et du chemin optimal).
   (b) Décrire une méthode pour déterminer l'intervalle de coût pour n'importe quelle arête de A. Votre méthode devra plus rapide que de déterminer les intervalles de coûts arête par arête. (Aide : utiliser la question traitant de $\beta[i, j]$).
8. **Les arêtes les plus vitales** Dans un arbre couvrant de poids minimum, nous appelons une arête une arête vitale si sa suppression augmente strictement le poids de l'arbre couvrant de poids minimum. Une arête la plus vitale est une arête vitale pour laquelle la suppression augmente, de manière la plus importante le coût de l'arbre couvrant de poids minimum.
   (a) Est-ce qu'un graphe contient une arête vitale ?
   (b) Supposons que le graphe contienne une arête vital. Décrire une méthode en $O(nm)$ pour identifier une arête le plus vital. Pouvez-vous développer un algorithme dont la complexité est inférieur à $O(nm)$? (Aide : utiliser les conditions de la coupe optimale).
9. **Arbre couvrant balancé** Un arbre couvrant est un arbre couvrant balancé si parmi les arbres couvrants, la différence entre l'arête de poids maximum dans T et l'arête de poids minimum dans T est aussi petit que possible. Décrire une méthode en $O(m^2)$ pour déterminer un arbre couvrant balancé.
10. **Second arbre couvrant minimal**
    Soit $G = (S, A)$ un graphe connexe non orienté avec une fonction de poids $w : A \to \mathbb{N}$ et supposons que $|A| \geq |S|$.
    (a) Montrer que le second arbre couvrant minimal n'est pas unique même si toutes les arêtes admettent des valeurs différentes.
    (b) Soit T un arbre couvrant de poids minimal de G. Montrer qu'il existe une arête $[u, v] \in T$ et $[x, y] \notin T$ tel que $T - \{[u, v]\} + \{[x, y]\}$ est un second arbre couvrant de poids minimal de G.
    (c) Soit T un arbre couvrant de G, et pour toutes paires de sommets $u, v \in V$, soit $\max[u, v]$ l'arête de poids maximum sur l'unique chemin entre u et v dans T. Proposer un algorithme qui calcule $\max[u, v], \forall u, v \in V$ sachant T.
    (d) Donner un algorithme efficace qui détermine le second arbre couvrant minimal.
11. **Construction d'un Arbre couvrant de poids minimal ?**
    soit $G = (S, A)$ un graphe valué positivement. On partage G en deux sous-graphes de G, $G_1 = (S_1, A_1)$ et $G_2 = (S_2, A_2)$ avec $S = S_1 \cup S_2$. Soit T_1 (respectivement T_2) un arbre couvrant minimalde G_1 (respectivement de G_2). On choisit une arête e de coût minimum parmi les arêtes ayant une extrémité dans S_1, et l'autre dans S_2, T est alors le graphe partiel de G résultant de la réunion des arêtes de T_1, de T_2, et de e.

(a) Examiner si l'algorithme précédent produit un arbre couvrant de poids minimal ; si oui, expliquer pourquoi, sinon construire un contre-exemple.

Correction exercice 9

1. si $e \in E$ alors $\tau(G) = \tau(G-e) + \tau(G.e)$. Les arbres couvrants de G ne contenant pas l'arête e sont les arbres couvrants de $G-e$. Dans le but de montrer que G admet $\tau(G.e)$ arbres couvrants contenant l'arête e, nous allons montrer qu'il existe une bijection entre l'ensemble des des arbres couvrants de G contenant e et l'ensemble des arbres couvrants de $G.e$. Quand nous contractons e dans un arbre couvrant contenant e, nous obtenons un arbre couvrant de $G.e$ car le sous-graphe résultant de $G.e$ est un arbre connecté possédant le bon nombre de d'arêtes. Les autres arêtes maintiennent leur identité malgré la contraction, ainsi il n'y a pas deux arbres qui seront mappé (injection) sur le même arbre de $G.e$ par cette opération. De plus, chaque arbre couvrant de $G.e$ survient de cette manière, sachant que recréant l'arête $e = [u, v]$ à partir de $G.e$ nous obtenons un arbre couvrant de G. Sachant que chaque arbre couvrant de $G.e$ apparaît seulement une fois, la fonction est une bijection.cd
2. **Second arbre minimal**
   (a) si on prend un carré avec un ordre 3, 4 5 et 6 et une liaison entre le sommet adjacent aux arêtes 4 et 5 de poids 2 vers son opposé. Il existe deux second arbre
   (b) Soit T' le second arbre minimal. Soit $[u, v] \in T - T'$. Alors $T' + [u, v]$ contient un cycle et une arête de ce cycle n'est pas dans T. Soit $[x, y]$ cette arête nous devons avoir $w[x, y] \geq w[u, v]$, car sinon nous pourrions remplacer $[u, v]$ dans T par $[x, y]$ pour obtenir un acpm plus faible que T. Maintenant $T'' = T' - [x, y] + [u, v]$ est également un arbre couvrant sachant que $[u, v]$ et $[x, y]$ sont dans le même cycle. $w(T'') \leq w(T')$, alors T'' est le meilleur acpm.
   (c) Pour chaque $u \in S$ utiliser un parcours en largeur pour toruver le poids maximum entre u et tout les autres $v \in S$. La parcours en largeur est en $O(m+n)$. Sachant que T est un arbre donc $m = n - 1$. Ainsi la complexité est en $O(n^2)$.

Algorithm 1 Algorithme donnant un second ACPM

(d) Trouver un ACPM T
Trouver l'arête $[x, y] \in A - T$ qui maximise $w[x, y] - w(\max[x, y])$, avec $\max[x, y]$ déterminer par la question précédente
return $T' = T - \max[x, y] + [x, y]$

La complexité est en $O(n^2)$

Fin correction exercice 9

Exercice 10 – Propriété des arbres

On rappelle l'énoncé du théorème d'optimalité pour l'arbre couvrant de poids minimum : On appelle $F = (F_1, F_2, \ldots, F_k)$ une forêt du graphe $G = (V, E)$, si chaque $F_i = (V_i, E_i)$, $i \in \{1, 2, \ldots, k\}$, est un arbre et si $\forall p \neq q$, $V_p \cap V_q = \emptyset$. Soit $[u, v]$ l'arête de G avec le plus faible poids ayant une extremité dans V_1. Alors parmi tous les arbres couvrant de G incluant la forêt F il en existe un, parmi les meilleurs, qui contient l'arête $[u, v]$.

1. Expliquer comment on peut utiliser ce théorème pour montrer que l'algorithme de Prim est correct. On précisera quelle est la forêt F à chaque étape de l'algorithme.
2. Même question, cette fois-ci pour l'algorithme de Kruskal.

Correction exercice 10

1. Pour Prim on peut séparer les sommets en deux sous-ensembles , le sommets faisant parti de l'arbre en construction $\in F_1$, les autres sommets forment un arbre pour chaque sommet et constitué d'un seul sommet. Du fait qu'on prenne à chaque fois l'arete de poids le plus faible, l'algorithme de prim est correct
2. On peut séparer les osmmets du graphe en deux sous-ensembles : les osmmets faisiant parties des arbres en constructions, les autres forment un arbre pour chaque sommet et constitué d'un seul sommet. Du fait qu'on prennn à chaque fois l'arête d epoids le plus faible, l'algorithme de kruskal est correct.

Fin correction exercice 10

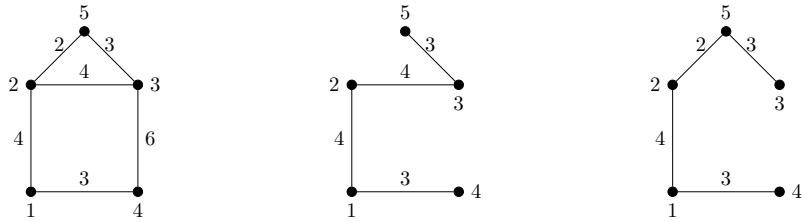
Exercice 11 – Propriété des arbres couvrants

On considère une variante de l'arbre couvrant de poids minimum (ACPM) dans laquelle on cherche à minimiser non plus la somme des poids des arêtes faisant partie de l'arbre couvrant, mais le poids de l'arête maximum (c'est à dire celle ayant le plus grand poids) faisant partie de l'arbre couvrant. On appelle cette variante l'arbre couvrant de poids maximum minimum (ACPMM).

1. Montrer que que toute solution optimale pour le problème de l'ACPM est une solution optimale pour le problème ACPMM.
2. La réciproque est-elle vraie ?

Correction exercice 11

1. Si on a un ACPM, on a forcément minimiser le poids des arêtes à chaque étape de l'agorithme, et donc le poids maximum des arêtes faisant partie de l'ACPM.
2. faux voir Îl suffit de regarder le graphe donné par la figure **??**



$ACPMM = 14$ $ACPM = 12$

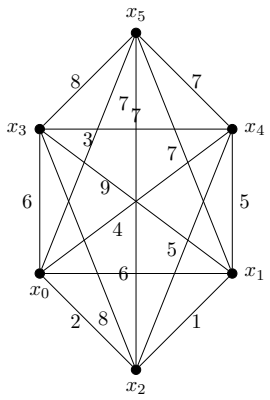


FIGURE 4 – A la recherche d'un ACPM.

Fin correction exercice 11

Exercice 12 – Applications d'algorithmes

Exécuter les algorithmes de Prim et de Kruskal sur les graphes donnés par la figure 4 et 5.

Correction exercice 12

a prori c'est 14 pour le premier et 90 pour le second.

Fin correction exercice 12

Exercice 13 – Algorithme

Un étudiant propose l'algorithme 2 pour calculer un arbre couvrant de valeur minimale d'un graphe $G = (S, A)$. Le graphe G est supposé non orienté et connexe. La fonction est sensée retourner l'ensemble des arêtes de l'arbre.

1. Montrer que la fonction retourne bien un arbre (raisonner par récurrence sur le cardinal de A ; utiliser un théorème du cours).
2. Montrer que l'arbre couvrant retourné par la fonction est de valeur minimale ou montrer que l'algorithme est faux en exhibant un contre-exemple.

Correction exercice 13

1. On utilise le théorème qui dit qu'un graphe de n sommets est un arbre si et seulement si il est connexe et comporte $n-1$ arêtes. Le graphe est connexe : c'est une des hypothèses. On raisonne par récurrence sur le cardinal $|A|$ de A. Base d ela récurrence : si $|A| = 1$ alors $n = 2$ puisque le graphe est connexe donc A est un arbre d'après le théorème. Le

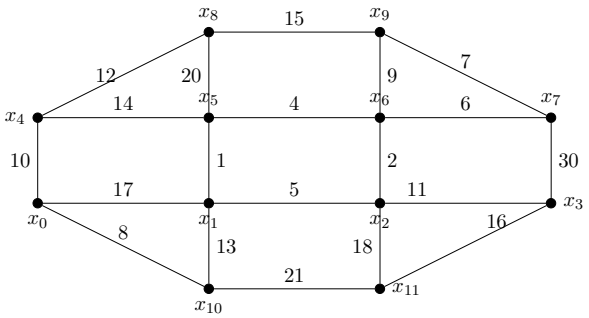


FIGURE 5 – A la recherche d'un ACPM (suite).

Algorithm 2 Algorithme donnant un arbre couvrant de poids minimale : fonction $arbre\_couvrant\_minimal(S, A)$

Soit $G = (S, A)$ un graphe non orienté connexe
if A ne contient qu'une arête **then**
  retourner A
else
  Partionner S en deux sous-ensembles S_1 et S_2 de même cardinal ± 1
  $A_1 := A \cap S_1 \times S_1$
  $A_2 := A \cap S_2 \times S_2$
  $T_1 := arbre\_couvrant\_minimal(S_1, A_1)$
  $T_2 := arbre\_couvrant\_minimal(S_2, A_2)$
  $a :=$ une arête de valeur minimale parmi celles qui n'appartiennent ni à A_1 ni à A_2
  retourner $T_1 \cup T_2 \cup \{a\}$
end if

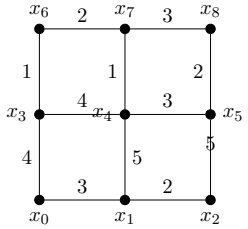


FIGURE 6 – Graphe pour l'algorithme de Sollin

cas général : le cas $n \geq 2$. Hypothèse de récurrence : l'algorithme retourne un arbre si on l'applique à un graphe connexe de moins de n somets. Notons n_1 et n_2 les nombres de sommets de S_1 et de S_2. On a $n_1 + n_2 = n$ et $n_1, n_2 < n$. Les graphes (S_1, A_1) et (S_2, A_2) sont connexes. L'hypothèse de récurrence s"applique : les deux apples récursifs retorunent des arbres et on a $|T_1| = n_1 - 1$ et $|T_2| = n_2 - 1$ et donc $|T_1 \cup T_2| = n_1 + n_2 - 2 = n - 2$. L'arête a connecte T_1 et T_2 donc $T_1 \cup T_2 \cup \{a\}$ est connexe. Ce graphe comporte n sommets et $n - 2 + 1 = n - 1$ arêtes. C'est donc un arbre.

2. L'algorithme est faux. considérer par exemple un carré dont les arêtes horizontales ont pour valeur 1 et les arêtes verticales ont pour valeur 2. Si l'algorithme découpe le carré suivant une ligne verticale, l'arbre obtenu est d e valeur 5 alors que l'arbre couvrant de valeur miniamle est de valeur 4.

Fin correction exercice 13

Exercice 14 – Algorithme de Sollin

Soit un graphe $G = (N, A)$ les sommets du graphe.

Algorithm 3 Algorithme de Sollin

for $i \in N$ **do**
  $N_i := \{i\}$
end for
while $|T^*| < (n-1)$ **do**
  for chaque arbre N_k **do**
    $Plusprochevoisin(N_k, i_k, j_k)$
  end for
  for chaque arbre N_k **do**
    if les sommets i_k et j_k appartiennent à deux arbres différents **then**
      $Fusion(i_k, j_k)$;
      $T^* = T^* \cup \{(i_k, j_k)\}$
    end if
  end for
end while

— La fonction $plusprochevoisin(N_k, i_k, j_k)$ prend en entrée un arbre couvrant sur les sommets de N_k et détermine l'arête de poids minimale parmi toutes les arêtes de N_k ($c_{i_k j_k} = \min\{c_{ij} : (i,j) \in A, i \in N_k \text{ et } j \notin N_k\}$)

— La fonction $Fusion(i_k, j_k)$ prend en entrée deux sommets i_k et j_k, et si ces deux sommets appartiennent à deux arbres différents, on fusionne les deux arbres.

1. Expliquer pourquoi il termine et pourquoi il donne un arbre couvrant. Quel théorème l'algorithme vérifie ?
2. Expliquer pourquoi il donne bien un arbre couvrant de poids minimum. Pour cela considérer un autre arbre B et montrer que forcément le poids de B est supérieur ou égal à A.
3. Donner sa complexité. Pour cela donner la complexité des deux fonctions *plusproche* et $fusion$.
4. Appliquer cet algorithme sur la figure 6.

Correction exercice 14

1. A chaque passage dans la boucle, deux composantes connexes distinctes de A sont reliés, ce qui ne crée pas de cycle, l'algorithme termine (diminution d'une composante connexe à chaque passage dans la boucle), et à la fin A est un arbre (graphe sans cycle à une seule coposante connexe). L'algorithme vérifie le théorème de la coupe minimale.
2. Soit B un arbre couvrant de G : à chaque arête a de A, on peut associer une et une seule arête b de B reliant les deux composantes connexes de A privé de l'arête a. Par construction de A, a est de poids minimal parmi les arêtes de G reliant ces deux composantes, donc de poids inférieure ou égal à celui de b : cette propriété étant vraie pour toutes les arêtes, le poids total d A est inférieur ou égal à celui de B. On peut utiliser le théorème e la coupe minimale.
3. La boucle tant que de l'algorithme est exécuté au plus $|V|-1$ fois. en fait chaque exécution de la boucle réduit au moins par deux le nombre d'arbres, donc la complexité est en $\log_2 n$) La recherche d'un voisin de plus faible coût nécessite $O(m)$. (à chaque fois, on ajoute une arête à A) ; le maintien de la liste des composantes connexes est de l'ordre de $|N|$ (il suffit de maintenir une liste de couleurs de sommets, les sommets d'une même composante ayant la meme couleur). Ainsi la complexité est en $O(m \log_2 n)$
4. la solution est 18.

Fin correction exercice 14

Exercice 15 – Algorithme

Soit G un graphe non orienté connexe à n sommets et p arêtes, dont les arêtes sont valuées par des nombres. Vous employez un jeune programmeur qui, pour construire un arbre recouvrant de poids minimal de G, vous propose l'algorithme suivant :

```
Soit A le graphe dont les sommets sont ceux de G,
et qui ne contient aucune arête ;
soit s un sommet de A ;
TANT QUE l'on modifie quelque chose
  parmi les arêtes de G d'extrémités
 qui ne sont pas dans A et ne créent pas de cycle dans A,
choisir une arêtes s-s'de poids minimal ;
```

ajouter l'arête s-s' à A;
s:=s'
FIN TANT QUE

1. Montrez que l'algorithme se termine et donner sa complexité.
2. Trouver un graphe pour lequel l'algorithme propose un graphe non connexe.
3. Votre employé propose la correction suivante : si le graphe A obtenu à la fin de l'algorithme n'est pas connexe. on recommence l'exécution de la boucle TANT QUE à partir d'une feuille de A (c'est-à-dire un sommet extrémité d'au plus une arête de A). Cette correction vous satisfait-elle ? Pourquoi ?

Correction exercice 15

1. A chaque passage dans la boucle TANT QUE, on ajoute une arête au graphe H, qui comporte au plus $n-1$ arêtes, il y a donc au plus $(n-1)$ passages dans la boucle. Par conséquent l'algorithme se termine. $0(n^2)$.
2. En appliquant l'algorithme au graphe suivant : $[a,b](poids=1)$, $(a,d)(poids=2)$, $[b,c](poids=1)$, montrez-lui que le graphe A obtenu à la fin de l'algorithme peut ne pas être connexe. On aura $s=a$ et on choisira l'arête $[a,b]$ puis $s=b$ et on choisit $[b,c]$ puis on pose $s=c$ et l'algorithme se termine . Le sommet d est isolé.
3. (ou toujours pas connexe) On obtient un arbre mais pas forcément de poids minimale. Il suffit de prendre le graphe suivant :$[a,b](poids=1)$, $(a,c)(poids=1)$, $[b,c](poids=2)$ On commence à choisir l'arête $[a,b]$ puis $s:b$ puis l'arête $[c,b]$. On obtient un arbre de poids 3.

Fin correction exercice 15

Exercice 16 – Algorithme Soit G un graphe non orienté connexe ayant n sommets et m arêtes, dont les arêtes sont valuées par des nombres. Vous employez un jeune programmeur qui, pour construire un arbre couvrant de poids minimum de G, vous propose l'algorithme suivant : Soit H le graphe dont les sommets sont ceux de G, et qui ne contient aucune arêtes. Soit s un sommet de H.
tant que l'on modifie quelque chose
parmi les arêtes de G d'extremité s
qui ne sont pas dans H et qui ne créent pas de cycle dans H,
choisir une arête $[s,s']$ de poids minimal
ajouter l'arête $[s,s']$ à H
$s:=s'$
fin tant que

1. Montrer que l'algorithme se termine.
2. En appliquant l'algorithme au graphe comportant une arête $[a,b]$ de poids 1, une arête $[a,d]$ de poids 2, et une arête $[b,c]$ de poids 1, montrez-lui que le graphe H obtenu à la fin de l'algorithme peut ne pas être connexe.
3. Votre employé propose la correction suivante : si le graphe H obtenu à la fin de l'algorithme n'est pas connexe, on recommence l'exécution de la boucle **tant que** à partir d'une feuille de H (c'est-à-dire un sommet extremité d'au plus une arête de H). Cette correction vous satisfait-elle ? Pourquoi ?

Correction exercice 16

a faire

Fin correction exercice 16

Exercice 17 – Algorithme
Déterminer un arbre couvrant de poids minimum du graphe donné par la figure 6 :

1. d'abord à l'aide de l'algorithme de Kruskal,
2. puis à l'aide de l'algorithme de Prim.

Il ne sera pas nécessaire de dessiner à chaque étape l'arbre couvrant ou la forêt en cours de construction, mais on indiquera à chaque étape quelle arête est sélectionnée, et on dessinera l'arbre couvrant de poids minimum obtenu à la fin des algorithmes.

On considère maintenant l'algorithme suivant :
On part de la liste des arcs dans un ordre quelconque $u_1, u_2, \ldots, u_n$.
Au depart $T := u_1$ et $k = 1$.
À l'étape k faire :
$T := T + u_k$
Si T ne contient pas de cycle, continuer (passer à l'étape $k+1$).
Si T contient un cycle C, et si v est un arc de poids maximum dans C, faire $T := T - v$ et continuer (passer à l'étape $k+1$).

1. Expliquer brièvement pourquoi cet algorithme termine et donne un arbre couvrant de poids minimum.
2. Appliquer cet algorithme sur la figure 6.
3. Expliquer brièvement comment on peut modifier l'algorithme précédent pour obtenir un arbre couvrant de poids maximum (il n'est pas demandé d'appliquer l'algorithme ainsi modifié sur le graphe).

Correction exercice 17

1. On obtient un graphe sans cycle. Lorsqu'il y a un cycle on enlève à chaque fois l' arête de poids maximum dans le graphe avec le cycle. L' algorithe termine lorsqu'on a traité toute les arêtes
2.
3. On enlève à chaque l'arête de poids minimum lorsqu'un cycle est détecté.l'ordre des poids est inversé, on pose $w'(u) = -w(u)$

Fin correction exercice 17

Exercice 18 – Arbres de recouvrement

Mettre en œuvre l'algorithme de `Prim` lorsque le graphe est représenté par une liste d'adjacence et ensuite lorsque le graphe est représenté par une matrice d'adjacence.

Correction exercice 18

a priroir il faut faire un parcours de la matrice d'adjacence.

Fin correction exercice 18

Exercice 19 – Implémentation de l'algorithme de Kruskal

Soit $G = (X, E)$ un graphe non orienté valué, on notera $\omega(e)$ le poids de l'arête e. L'algorithme consiste à partir d'un graphe T vide qui possède les mêmes sommets que G. A chaque étape on prend l'arête de poids minimum qui n'est pas déjà dans T et qui ne forme pas de cycle lorsque elle est ajoutée à T.

1. Montrer que cet algorithme donne bien un arbre de recouvrement minimum (on supposera pour simplifier que toutes les arêtes ont un poids différent).

   On se propose maintenant de trouver un algorithme efficace pour déterminer si une arête va créer un cycle si on l'ajoute à un graphe. Pour cela on utilise un tableau qui pour chaque sommet contient sa composante connexe, représentée par un des nœuds de la composante connexe. Une arête $\{x, y\}$ crée un cycle si x et y appartiennent à la même composante connexe.
2. Déterminer la complexité pour savoir si une arête crée un cycle et quelle est la durée de mise à jour de la structure lorsque l'on ajoute une arête (il faut faire la fusion des composantes connexes contenant les extrémités)..
3. On structure maintenant les composantes connexes sous forme d'arborescences, la racine étant le représentant de la composante connexe. Que deviennent les complexités de la recherche de la composante connexe d'un sommet et la fusion de deux composantes connexes ?
4. Modifier la structure pour que la composante la plus grosse absorbe la plus petite en cas de fusion. Que deviennent la complexité des fonctions de recherche de composante connexe et de fusion ?
5. Proposer une implémentation de l'algorithme de Kruskal et évaluer sa complexité.

Correction exercice 19

1. Raisonnons par l'absurde. Soit $e_1,\ e_2, \ldots,\ e_{n-1}$ les arêtes données par l'algorithme avec $i < j \Longrightarrow w(e_i) < w_(e_j)$ pour un arbre T. Supposons qu'il ne soit pas minimal. Considérons les arêtes de l'ACPM $e'_1,\ e'_2, \ldots,\ e'_{n-1}$ tel que $i < j \Longrightarrow w(e'_i) < w_(e'_j)$ pour un arbre T'. On suppose que $\sum_{i=1}^{n-1} w(e'_i) < \sum_{i=1}^{n-1} w(e_i)$.

   Soit $i = \min j$ tel que $e_j \neq e'_j$ on a $w(e_i) < w(e'_i)$ et $e_1 = e'_i,\ e_2 = e'_2,\ \ldots, e_{i-1} = e'_{i-1}$, car sinon e_i aurait été sélectionné par l'algorithme, e'_i ne forme pas de cycle avec $e_1,\ e_2, \ldots, e_{i-1}$ car toutes ces arêtes appartiennent à T'.

   Soit $G' = T' + \{e_i\}$, G' possède un cycle, soit f l'arête la plus lourde de ce cycle (remarque $w(f) > w(e_i)$ car les arêtes $e_1,\ e_2, \ldots, e_i$ ne peuvent pas former de cycle à elles seules). Soit $G'' = G' - \{f\}$ est un arbre avec $w(T'') = w(T') + w(e_i) - w(f)$ et donc $w(T'') < w(T')$ impossible donc T' n'est pas un acpm, et donc les hypothèses sont fausses.

Algorithm 4 Algo1

```
Comp(x, T)
retourner (cc[x])
Fusion (x, y, T)
for i := 1 to n do
  if Comp(i, T) = Comp(y, T) then
    cc[i] < − − Comp(x, T)
  end if
end for
```

Algorithm 5 Algorithme de Kruskal

```
for i = 1 to n do
  cc[i] ← i
  cpt ← 0
  T ← ∅
  cptarete ← 1
end for
while cpt ≠ (n − 1) do
  prendre l'arête la plus légère {x, y} , O(1) θ(log n)
  if comp(x, T) ≠ comp(y, T), O(1) then
    T ← T ∪ {x, y}, O(1)
    cpt ← cpt + 1
    fusion(x, y, T) en O(n)
  end if
  cptarete ← cptarete + 1
end while
```

2. Complexité sans prendre en compte le tri des arêtes $O(m+n^2)$ avec m = nombre d'arêtes

   Pour le tri des arêetes, on va construire un tas avec les poids des arêtes $\theta(n)$. Soit p le nombre d'arêtes testées p peut aller de $n-1$ à $m \leq n^2$, d'où la complexité globale de $\theta(p \log n + n^2)$.
3. Soit le pseudo code suivant :
4. Soit le pseudo code suivant :

   Si on fusionne une composante de taille n_1 représentée par une arborescence de profondeur h_1 avec une composante de taille n_2 représentée par une arborescence de profondeur h_2 : on obtient une composante de taille $(n_1 + n_2)$ représenté par une arborescence de profondeur (si $n_1 \geq n_2$ $\max(h_1, h_2 + 1)$).

   Montrer qu'une composante de taille n est représentée par une arborescence de profondeur $\leq \log_2 n$

   — Vrai pour $n = 1,\ 2,\ 3$

   — On suppose que c'est vrai pour $\forall k < n$ Soit t_n la profondeur maximum de l'arborescence représentant une composante de taille ($t_k \leq \log_2 k$ si $k < n$) La composante de

Algorithm 6 ALgo2

```
comp(x, T)
aux <−− x
while cc[aux] ≠ aux do
  aux <−− cc[aux]
  retourneraux
end while
fusion(x, y, T)
aux1 <−− comp(x, T)
aux2 <−− comp(y, T)
if aux1 ≠ aux2 then
  cc[aux1] <−− aux2
end if
```

Algorithm 7 ALgo3

```
comp(x, T)
aux <−− x
while aux ≠ cc[aux].pere do
  aux <−− cc[aux].pere
  fusion(x, y, T)
  aux1 <−− comp(x, T)
  aux2 <−− comp(y, T)
  if aux1 ≠ aux2 then
    if cc[aux1].taille > cc[aux2].taille then
      cc[aux2].pere <−− aux1
      cc[aux1].taille <−− cc[aux1].taille + cc[aux2].taille
    else
      meme chose aux1 <−− aux2
    end if
  end if
end while
```


taille n a été obtenue en fusionnant une composante de taille n_1 avec une composante de taille n_2. par hypothèse , nous avons $n_1 \geq n_2$ avec $n = n_1 + n_2,\ n_2 \leq n/2$.
Donc

$$\begin{aligned} t_n &\leq \max(t_{n_1}, t_{n_2} + 1) \\ &\leq \max(\log_2 n_1, \log_2 n_2 + 1) \\ &\leq \log_2 n \\ &\leq \log_2 n/2 + 1 = \log_2 \end{aligned}$$

Donc $t_n \leq \log_2 n$
Donc le coût de la recherche d'une composante connexe d'un sommet $\theta(\log_2 n)$

Algorithm 8 Algorithme de Kruskal bis

```
Construire un tas avec les arêtes θ(n)
while n − 1 arêtes n'ont pas été sélectionnées do
  Mettre l'arête {x, y} la plus légère θ(log n)
  if comp(x) ≠ comp(y), θ(log n) then
    sélectionné l'arête
    fusion des composantes connexes θ(log n)
  end if
end while
```

Complexité global $\theta(p \log n)$

Fin correction exercice 19

Exercice 20 – Chaine hamiltonienne

Prouver que l'algorithme suivant ne donne pas toujours une chaîne hamiltonienne de poids minimum dans le graphe complet :

1. choisir une arête e_1 telle que $w(e_1)$ soit minimum
2. à toute étape k, choisir une arête e_k telle que le graphe engendré par $e_1, e_2, \ldots, e_k$ soit union disjointe de chaînes, et telle que $w(e_k)$ soit minimum.

Correction exercice 20

Voici deux solutions dont une fonctionne 7 et l'autre ne marche pas **??**

Fin correction exercice 20

Exercice 21 – Le voyageur de commerce

Le tableau suivant donne les distances en km entre cinq villes :

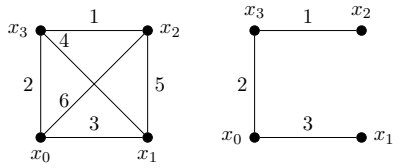


FIGURE 7 – Graphe pour lequel l'algorithme fonctionne

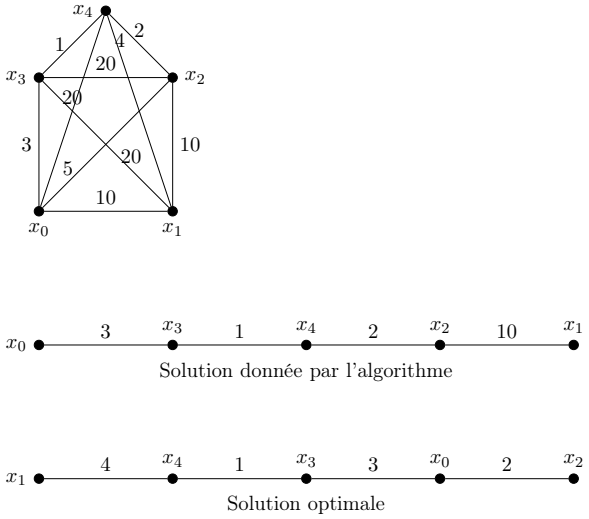


FIGURE 8 – Graphe pour lequel l'algorithme ne fonctionne pas

	A	B	C	D	E
A	0	9	7	5	7
B	9	0	9	9	8
C	7	9	0	7	6
D	5	9	7	0	6
E	7	8	6	6	0

Trouver une borne inférieure de la solution du problème du voyageur de commerce en enlevant

1. la ville B,
2. la ville E,

Correction exercice 21

poids est 17 en enlevant la ville B. Si on enlève la ville E le poids est de 21.

Fin correction exercice 21

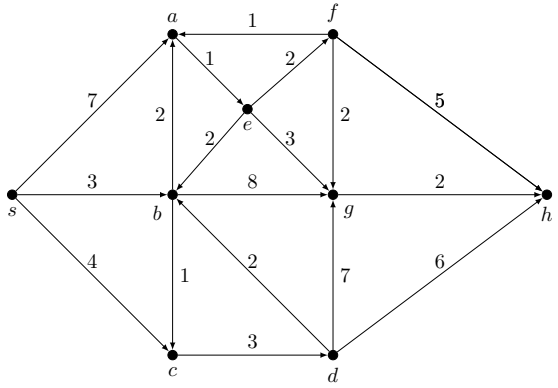


FIGURE 1 – Un graphe orienté valué.

Université de Montpellier 2020/2021
HMIN329 – Complexité et algorithmes Durée: 4.5h

Plus courts chemins
TD – Séance n 6

Exercice 1 – Applications d'algorithmes

Soit P un plus court chemin de s à t dans un graphe G, et x, y deux sommets rencontrés dans cet ordre sur P. Montrer que la portion P_{xy} de P comprise entre x et y est un plus court chemin de x à y.

Correction exercice 1

S'il existait un chemin Q_{xy} = plus court que P_{xy} entre x et y , on pourrait raccourcir P en remplacant P_{xy} par Q_{xy}.

Fin correction exercice 1

Exercice 2 – Arborescence

On considère le graphe orienté valué défini ci-dessous (figure 1). Déterminer une arborescence des plus courts chemins depuis le sommet s en utilisant l'algorithme de Dijkstra.

Correction exercice 2

Simple.

Fin correction exercice 2

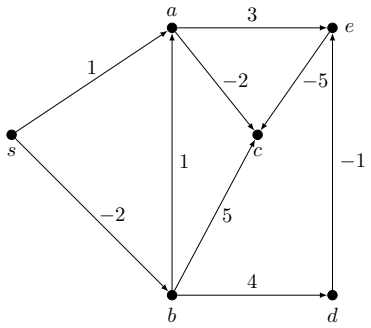


FIGURE 2 – Un graphe orienté valué.

Exercice 3 – Arborescence

On considère le graphe orienté valué donné par la figure 2.

1. Déterminer une arborescence des plus courts chemins depuis le sommet s en utilisant la programmation dynamique.
2. Qu'obtient-on avec l'algorithme de Dijkstra ? Conclure.
3. Une solution consisterait d'augmenter de $|a|$ où a est la valeur minimale sur les arcs et après d'enlever de $-a$. Que pensez-vous de cette solution ?

Correction exercice 3

1. Avec la programmation dynamique, on obtient bien les plus courts chemins.
2. Avec l'algorithme de Dijkstra, on n'obtient pas les plus courts chemins, ceci confirme le fait que des arcs ayant une valuation positive.
3. Ceci ne donne pas le bon concernant les plus court chemin

Fin correction exercice 3

Exercice 4 – Plus courtes chaînes dans un graphe non orienté

Le but de cet exercice est de montrer comment adapter l'algorithme de Dijkstra pour le calcul des plus courtes chaînes dans un graphe non orienté. Dans une première étape nous montrerons comment appliquer l'algorithme de Dijkstra à partir du graphe orienté obtenu par dédoublement des arêtes du graphe initial. Dans une second étape nous montrerons comment il est possible de s'affranchir de cette transformation du graphe initial.

Soit $G = (S, A, W)$ un graphe non orienté valué positivement que nous supposerons connexe et un sommet $s \in S$. L'objectif est de déterminer, pour chaque sommet $x \in S$, une plus courte chaîne reliant s à x ; nous notons $d(x)$ la longueur de cette chaîne. Soit $\hat{G}$ le graphe orienté obtenu à partir de G de la manière suivante : l'ensemble des sommets de $\hat{G}$ est S ; à chaque arête $[i, j]$ de valuation w_{ij} de G correspond les deux arcs symétriques (i, j) et (j, i) de $\hat{G}$ tout deux de valuation w_{ij}.

1. Soit $c = (s = i_1, \ldots, i_k = x)$ une chaîne reliant s à x dans G et soit $\mu = (s = i_1, \ldots, i_k = x)$ le chemin correspondant de $\hat{G}$. Montrer que c est une plus courte chaîne si et seulement si μ est un plus court chemin.
2. Déduire de la question précédente comment l'algorithme de Dijkstra permet de fournir les plus courtes chaîne issues du sommet s. La transformation proposée précédemment étant trop lourde, nous présentons une version de l'algorithme de Dijkstra applicable aux graphes non orientés. Considérons l'algorithme suivant qui est l'adaptation de l'algorithme de Dijkstra aux graphes non orientés :

Algorithm 1 L'algorithme de Dijkstra pour les graphe non orienté à partir d'un sommet s

$d(s) := 0$;
$d(i) := +\infty$; pour tout $i \neq s$
$S = \emptyset$
while $S \neq X$ **do**
  choisir $i \in \overline{S}$ avec $d(i)$ minimum
  $S = S \cup \{i\}$
  for chaque arc $[i, j]$ **do**
    if $d(j) > d(i) + w_{ij}$ **then**
      $d(j) := d(i) + w_{ij}$
    end if
  end for
end while

3. En utilisant le fait que l'algorithme de Dijkstra pour les graphes orientés est valide, montrer que l'algorithme précédent calcule le plus courtes chaînes ayant s pour extrémité.

Correction exercice 4

1. La relation faisant correspondre une chaîne $c = (i_1, \ldots, i_k)$ de G avec le chemin $\mu = (i_1, \ldots, i_k)$ de $\hat{G}$ est une bijection de l'ensemble des chaînes de G vers l'ensemble des chemins de $\hat{G}$; c et μ ont la même longueur. En conséquence, aux plus courtes chaîne de G correspondent les plus courts chemins de $\hat{G}$.
2. En appliquant l'algorithme de Dijkstra à $\hat{G}$ on obtient les plus courts chemins d'origine s. A ces plus courts chemins correspondent les plus courtes chaînes de G ayant s comme extrémité.
3. L'algorithme de Dijkstra calcule correctement les plus courts chemins dans les graphes orientés. De plus, nous avons vu plus haut que la relation faisant correspondre une chaîne $c = (i_1, \ldots, i_k)$ de G avec le chemin $\mu = (i_1, \ldots, i_k)$ de $\hat{G}$ est une bijection de l'ensemble des chaînes de G vers l'ensemble des chemins de $\hat{G}$. La valeur $d(x)$ calculée dans la version orientée de l'algorithme est la longueur d'un chemin $(s, \ldots, x)$ de $\hat{G}$. Cette valeur est aussi la longueur d'une chaîne correspondante de G. Le sommet $x \in S$ de valeur $d(x)$ minimale considéré au cours de chaque itération de la version non orientée de l'algorithme est aussi un sommet de valeur $d(x)$ minimale dans la version orientée. Cette valeur est celle d'un plus court chemin de $\hat{G}$, elle est donc aussi celle d'une plus courte chaîne de G.

Fin correction exercice 4

Exercice 5 – Propriété d'un plus court chemin

1. Which of the following claims are true and which are false ? Justify your answer by giving o proof or by constructing a counterexample.
   (a) If all arcs in a network have different costs, the network has a unique shortest path tree.
   (b) In a directed network with positive are lengths, if we eliminate the direction on every arc (i.e. make it undirected), the shortest path distances will not change.
   (c) In a shortest path problem, if each arc length increases by k units, shortest path distances increase by a multiple of k
   (d) In a shortest path problem, if each arc length decreases by k units, shortest path distances decrease by a multiple of k
   (e) Among all shortest paths in a network, Dijkstra's algorithm always finds a shortest path with the least number of arcs.
2. Suppose that you are given a shortest path problem in which all arc lenght are the same. How will you solve this problem in the least possible time ?
3. Suppose that you want to determine a path of shortest length that can start at either of the noses s_1 or s_2 and can terminate at either of the node t_1 and t_2. How would you solve this problem ?
4. Show that in the shortest path problem if the length of some are decreases by k units, the shortest path distance between any pair of nodes decreases by at most k units.
5. **Most vital arc** A vital arc of a network is an arc whose removal from the network causes the shortest distance between two specified nodes, say note node s and node t ; to increase. A most vital arc is a vital arc whose removal yields the greatest increase in the shortest distance from node s to node t. Assume that the network is directed, arc lengths are positive, and some arc vital. Prove that the following statements are true or show through counterexamples that trey are false.
   (a) A mot vital arc is an arc with the maximum value of c_{ij}.
   (b) A most vital arc is an arc with the maximum value of c_{ij} on some shortest path from a node s to t.
   (c) An arc that does not belong to any shortest path from node s to node t cannot be a most vital arc.
   (d) A network might contain several most vital arcs.
6. Describe an algorithm for determining a most vital arc in a directed network. What is the running time of your algorithm ?

Correction exercice 5

1.

Fin correction exercice 5

Exercice 6 – Propriétés des chemins

Est-il vrai que

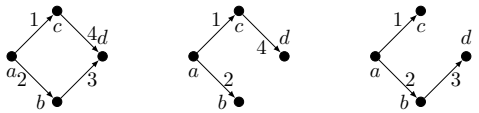
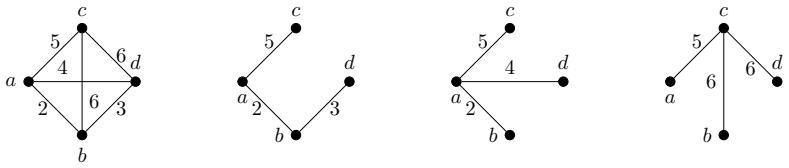


FIGURE 3 – Contre-exemple



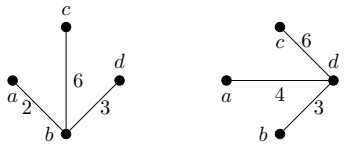


FIGURE 4 – Contre-exemple

1. Si tous les arcs ont des poids différents, l'arborescence des plus courts chemins à partir d'un sommet s fixé est unique ?
2. Parmi les n arborescences des plus courts chemins au moins l'un d'eux correspond à un arbre couvrant optimal ?

Correction exercice 6

1. la réponse est non. (voir la figure 3) (faire le lien avec les arbres couvrant de poids minimum, dans lequel si tous les arcs sont de poids différents alors il y a unicité de l'arbre).
2. La réponse est également non. (voir la figure 4)

Fin correction exercice 6

Exercice 7 – Algorithme

Soit un graphe orienté G, s un sommet de G. On cherche à déterminer le poids minimal des chemins reliant s à tous les autres sommets de G en utilisant l'algorithme glouton suivant : on pose $dist(s) = \infty$ pour tout sommet s, et $dist(s) = 0$. On visite ensuite le sommet s au moyen de la procédure récursive


```
visiter un sommet s
 Si s n'a pas encore été visité ALORS
   soit s' le successeur de s le plus proche de s
   dist(s'):=dist(s)+poids(s,s')
   visiter s'
 FIN SI
```

1. Quelle est la complexité de cet algorithme ?
2. Expliquer pourquoi il est incorrect (il y a plusieurs raisonsà cela ; on illustrera chacun des problèmes par un exemple de graphe pour lequel l'algorithme ne donne pas le résultat cherché).

Correction exercice 7

1. On visite au plus une fois chaque sommet et on examine au plus une fois chaque arc donc $\max(n, m)$.
2. Soit le graphe $s_0 \to s_1$(*poids de* 2), $s_0 \to s_2$ (poids de 1), on ne visite pas s_1. On n'a pas forcément le plus court des chemins un triangle pour exemple.

Fin correction exercice 7

Exercice 8 – Algorithme de Floyd

L'algorithme de Floyd calcule les plus courts chemins pour tout couple de sommets dans un graphe orienté valué $G = (V, E)$. Pour que ce problème ait une solution le graphe ne doit pas contenir de circuit absorbant. La donnée du problème est un graphe orienté valué représenté par sa matrice D d'adjacence : les entrées $d[i, j]$ de D sont les valuations v_{ij} des arcs ; par convention nous aurons $d[i, j] = \infty$ pour $i \neq j$ si $(i, j) \notin E$ et $d[i, j] = 0$ pour $i = j$. Le résultat de l'algorithme est une matrice Δ pour laquelle l'entrée $\Delta[i, j]$ est la longueur d'un plus court chemin d'origine i et d'extrémité j ; s'il n'existe pas de chemin $(1, \ldots, j)$ dans le graphe alors $\Delta[i, j] = \infty$.

Voici l'algorithme :

Algorithm 2 Algorithme de Floyd

```
Soit Δ une matrice
Δ ← D
for k ← 1à n do
  for j ← 1à n do
    for i ← 1à n do
      Δ[i, j] ← min{Δ[i, j], Δ[i, k] + Δ[k, j]}
    end for
  end for
end for
```

1. Donner la complexité de cet algorithme.
2. Appliquer l'algorithme de Floyd pour obtenir les plus courts chemins entre tout couple de sommets du graphe donné par la figure 5.

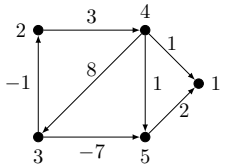


FIGURE 5 – Un graphe orienté valué.

Correction exercice 8

1. La complexité est en $O(n^3)$.
2. — On a $\Delta^0 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 3 & \infty \\ \infty & -1 & 0 & \infty & -7 \\ 1 & \infty & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

   — Sommet 1 : $\Delta^1 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 3 & \infty \\ \infty & -1 & 0 & \infty & -7 \\ 1 & \infty & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

   — Sommet 2 : $\Delta^2 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 3 & \infty \\ \infty & -1 & 0 & 2 & -7 \\ 1 & \infty & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

   — Sommet 3 : $\Delta^3 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 3 & \infty \\ 3 & -1 & 0 & 2 & -7 \\ 1 & 7 & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

   — Sommet 4 : $\Delta^4 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ 4 & 0 & 11 & 3 & 4 \\ \infty & -1 & 0 & 2 & -7 \\ 1 & 7 & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

   — Sommet 5 : $\Delta^5 = \Delta = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ 4 & 0 & 11 & 3 & 4 \\ -5 & -1 & 0 & 2 & -7 \\ 1 & 7 & 8 & 0 & 1 \\ 2 & \infty & \infty & \infty & 0 \end{pmatrix}$.

Fin correction exercice 8

Exercice 9 – Matrices et plus court chemins

Une structure de données simple pour représenter un graphe est la matrice d'adjacence M. Pour obtenir M, on numérote les sommets du graphe de façon quelconque. $X = \{x_1, \ldots, x_n\}$. M est une matrice carrée $n \times n$ dont les coefficients sont 0 et 1 telle que $M_{i,j} = 1$ si $(x_i, x_j) \in E$ et $M_{i,j} = 0$ sinon. Démontrer le la proposition suivante : **Proposition : Soit M^p la puissance $p-ieme$ de la matrice M, le coéfficient $M^p_{i,j}$ est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j.**

Correction exercice 9

On effectue une récurrence sur p. Pour $p = 1$ le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Le calcul de M^p, pour $p > 1$ donne :

$$M^p_{i,j} = \sum_{k=1}^{k=n} M^{p-1}_{i,k} \times M_{k,j}$$

Or tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur $p-1$ entre x_i et un certain x_k suivi d'un arc reliant x_k et x_j. Le résultat découle alors de l' hypothèse de récurrence suivant laquelle $M^{p-1}_{i,k}$ est le nombre de chemins de longueur $p-1$ joignant x_i à x_k.

Fin correction exercice 9

Exercice 10 – Détection d'un circuit absorbant

Nous allons montrer comment il est possible de détecter la présence de circuits absorbants en utilisant l'algorithme de Floyd.

Afin de rendre compte de circuits absorbants consistant en une boucle de valuation strictement négative, les éléments $d[i,j]$ de la matrice d'adjacence du graphe sont définis de la façon suivante : $d[i,i] = 0$ s'il n'y a pas de boucle au sommet i ou si $v_{ii} < 0$, $d[i,i] = v_{ii}$. Soit Δ la matrice résultat obtenu en exécutant l'algorithme de Floyd.

1. Montrer que G a un circuit absorbant si et seulement si il existe un sommet i de G tel que $\Delta[i,i] < 0$.

Correction exercice 10

1. Remarquons que l'algo de floyd tel qu'il est présenté, appelé avec la matrice d'adjacence D, se termine et que d'autre part, lors de son exécution les suites des valeurs $\Delta[i,j]$ sont décroissantes.

   Si G a un circuit absorbant qui est une boucle de valuation $v_{ii} < 0$,à l'issue de lexécution nous avons $\Delta[i,i] < 0$ Si G a un circuit absorbant qui n'est pas une boucle, il existe deux sommets i et k, $i \neq k$, et deux chemins $(i, \ldots, k)$ de longueur l_{ik} et $(k, \ldots, i)$ de longueur l_{ki} tels que $l_{ik} + l_{ki} < 0$. A l'issue de l'exécution de l'algorithme nous avons $\Delta[i,k] \leq l_{ik}$, $\Delta[k,i] \leq l_{ki}$ et $\Delta[i,i] \leq \Delta[i,k] + \Delta[k,i]$. Ainsi nous obtenons $\Delta[i,i] \leq \Delta[i,k] + \Delta[k,i] \leq l_{ik} + l_{ki} < 0$.

Soit i un sommet tel que $\Delta[i,i] < 0$. Nous notons Δ^k la valeur de la matrice Δ après l'itération k de la boucle la plus externe de l'algorithme. Après initialisation $\Delta^0[i,i] = 0$ s'il n'y a pas de boucle de valuation strictement négative en i et $\Delta^0[i,i] < 0$ s'il existe un eboucle de valaution strictement négative. Pour tout couple de sommet i et j la suite des valeurs $\Delta^0[i,j], \Delta^1[i,j], \ldots, \Delta^n[i,j]$ est décroissante.Il existe donc une itération k où $\Delta^k[i,i] < 0$. Si $k = 0$ il y a une boucle de valaution strictement négative en i qui contstitue un circuit absorbant. Si $k > 0$ nous avons $\Delta^k[i,i] = \Delta^{k-1}[i,k] + \Delta^{k-1}[k,i] < 0$ avec $\Delta^{k-1}[i,k]$ la longueur d'un chemin du sommet i au sommet k et $\Delta^{k-1}[k,i]$ la longueur du chemin de kà i ; G contient donc un circuit de longueur strictement négative passant par i.

Fin correction exercice 10

Exercice 11 – Impression de plus courts chemins

Écrire une procédure qui imprime un plus court chemin entre deux sommets i et j, s'il existe, en utilisant la matrice P calculée dans la procédure *Floyd*. Dans un premier temps vous modifierez l'algorithme de Floyd pour y inclure l'instruction nécéssaire à l'affichage du chemin de i à j.

Correction exercice 11

Il suffit de remplaçant l'instruction interne par : $if\ \Delta[i,j] > \Delta[i,k] + \Delta[k,j]\ then\ \Delta[i,j] \leftarrow \Delta[i,j] + \Delta[k,j],\ P[i,j] = k$.

```
/* Attention ici il faut noter qu'avant d'appeler la procedure il faut */
/* verifier que le chemin existe bien cad si A[i][j] != infini (notation */
/* du cours). */

void chemin(int i, int j)
{
  if (P[i][j] != i)
    chemin(i,P[i][j]);
  else
    printf("%d, ",i);
  printf("%d ", j);
}
```

Fin correction exercice 11

Exercice 12 – Diamètre et excentricité

Soit un graphe $G = (V, E)$. On appelle *excentricité* d'un sommet X, et on la note $e(X)$, la plus grande distance entre le sommet X et les autres sommets du graphe :

$$e(X) = \max_{v \in V} d(X, v)$$

Le diamètre D d'un graphe est alors défini de la façon suivante :

$$D = \max_{v \in V} e(v)$$

A l'aide de l'algorithme de Dijkstra, vous calculerez pour un sommet donné v son excentricité. Vous généraliserez votre algorithme de façonà calculer le diamètre du graphe G. Vous avez toute latitude pour choisir la représentation du graphe.

Correction exercice 12

On calcule l'excentricité pour chaque sommet, et on prend le maximum pour le diamètre.

Fin correction exercice 12

Exercice 13 – Exploration en profondeur

Utiliser la procédure d'exploration en largeur d'un graphe $G(X, A)$, $|X| = n$, $|A| = m$, pour obtenir en $O(m)$ les plus court chemins en nombre d'arcs d'un sommet s vers tout autre sommet.

Correction exercice 13

a faire

Fin correction exercice 13

Exercice 14 – Plus courts chemins avec des longueurs unitaires

Dans cet exercice nous considérons le problème de déterminer les plus courts chemins d'un sommet sà tous les autres dans le cas où la valuation sur les arcs est unitaire.

1. Donner un algorithme qui donne l'arborescence des plus courts chemins d'un sommet sà tous les autres en $O(m)$.

Correction exercice 14

1. Pour cela il suffit de considérer l'algorithme de Berge suivant basé sur un parcours en largeur.

Fin correction exercice 14

Exercice 15 – Chemins de capacité maximale

Algorithm 3 Algorithme de Berge basé sur un parcours en largeur

Soit $V_0 = \{s\}$
for chaque i **do**
Calculer V_{i+1} tel que V_{i+1} est égalà l'ensemble des sommets $v \in V \backslash (V_0 \cup V_1 \cup \ldots \cup V_i)$ pour chaque $(u, v) \in E$ avec $u \in V_i$.
end for

L'objectif de cet exercice est d'adapter l'algorithme de Dijsktra à la recherche de chemins de capcité maximale d'un sommet s à tout autre sommet d'un graphe G.

Soit $G = (S, A)$ un graphe admettant une source s et soit $c : A \to N$ une focntion capacité définie sur l'ensemble des arcs. La capacité d'un chemin est la plus peite capacité des arcs du chemin : si $\mu = (x_0, \ldots, x_q)$ est un chemin de G, $\lambda(\mu)$ sa capacité est $\lambda = \min_{0 \leq i \leq q-1} \{c(x_i, x_{i+1}\}$.

1. Montrer que l'ensemble des chemins élémentaires est dominant.
2. Montrer que contrairement au problème de recherche d'un plus court chemin, un sous-chemin d'un chemin de capacité maximale n'est pas nécessairement de capacité maximale.
   Nous notons $\lambda(x)$ la capacité d'un chemin $(s, \ldots, x)$. Soit y un successeur de x et $(s, y_1, \ldots, y_{q-1}, y)$, $q \geq 1$, un chemin tel que $y_{q-1} \neq x$. Par convention, si un tel chemin n'existe pas $\lambda(y) = 0$.
3. Montrer que si $\min(\lambda(x), c(x, y)) > \lambda(y)$ alors il existe un chemin $(s, \ldots, x, y)$ de capacité $\min(\lambda(x), c(x, y))$.
4. Montrer comment adapter l'algorithme de Dijkstra à la recherche de chemins de capacité maximale.
5. Montrer que l'évaluation du sommet x choisi à chaque itération est la capacité maximale d'un chemin de s à x dans G.

Correction exercice 15

1. Soit $\mu = (x_0, \ldots, x_i, \ldots, x_i, \ldots, x_q)$ un chemin contenant un circuit et soit $\mu' = (x_0, \ldots, x_q)$ le chemin obtenu à partir de μ en supprimant le circuit $\sigma = (x_i, \ldots, x_i)$. Nous avons $\lambda(\mu) = \min(\lambda(\mu'), \lambda(\sigma)) \leq \lambda(\mu')$, la capacité du chemin μ' est donc supérieure à celle de μ.
2. Considérons l'exemple donné par la figure 6 : le chemin (s, b, c) est un chemin de capacité maximale ; le chemin (s, b) de capacité 1 n'est pas de capacité maximale car (s, a, b) est de capacité 2.
3. y est un successeur de x donc $(s, \ldots, x, y)$ est un chemin de s à y de capacité $\min(\lambda(x), c(x, y))$
4. Dans la phase d'initialisation de l'algorithme de Dijkstra il faut $\lambda(s) = 0$ et $\lambda(x) < 0$ pour tout sommet $x \neq s$ de G. Le sommet à examiner à chaque étape est la sommet $x \in S$ d'évaluation $\lambda(x)$ maximale. L'itrération fondamentale de l'algorithme devient : si $\min(\lambda(x), c(x, y)) > \lambda(y)$ alors $\min(\lambda(x), c(x, y)) = \lambda(y)$
5. Après la phase d'initialisation, la propriété est vraie pour le sommet s. Supposons que cette propriété soit vérifiée pour les sommets $x \notin S$. Soit $\mu = (s, \ldots, x)$ un chemin de s à x et (y, z) tel que $y \in S$ et $z \notin S$. Supposons que $z \neq x$, alors $\lambda(z) \leq \lambda(x)$ et $\lambda(\mu) \leq \lambda(x)$, il existe donc un chemin de capacité maximale de s à x n'empruntant que des sommets

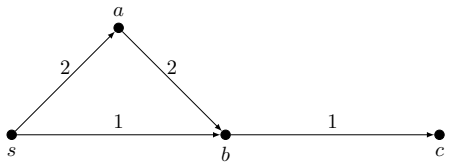


FIGURE 6 – Un graphe orienté valué.

$t \in S$. Ces sommets ayant été examinés avant x par l'algorithme, $\lambda(x)$ est la capcité maximale d'un chemin de s à x.

Fin correction exercice 15

Exercice 16 – Second plus court chemin

Soit G un graphe orienté défini par une matrice d'adjacence. On cherche à adapter l'algorithme de Roy-Warshall pour calculer le plus court chemin (en nombre d'arcs) entre les sommets i et j. On note d_{ij} la longueur (notée l) d'un plus court chemin entre deux sommets i et j. Soit $c = (s_1, \ldots, s_k)$ avec $s_1 = i$ et $s_k = j$ un plus court chemin entre i et j.

Soit d^k_{ij} la longueur d'un plus court chemin entre i et j passant par les sommets intermédiaires d'indices $\leq k$.

1. Que vaut d^0_{ij} ?
2. Quelle relation de récurrence lie d^{k+1} à d^k ?
3. Soit d une matrice carrée d'ordre n contenant toutes les valeurs $d_{ij}, 1 \leq i \leq n$ et $1 \leq j \leq n$. Ecrire un algorithme de calcul de d.
4. Comparer cet algorithme avec l'algorithme de Roy-Warshall décrit précédemment.
   On souhaite obtenir les longueurs des deux premiers plus courts chemins entre i et j. Soit d'^k_{ij} la longueur du deuxième plus court chemin entre i et j passant par des sommets intermédiaires d'indices $\leq k$.
5. Que vaut d'^0 ?
6. Quelle relation de récurrence lie d'^{k+1} à d'^k et à d^k ?
7. Soit d' une matrice carrée d'ordre n contenant toutes les valeurs $d'_{ij}, 1 \leq i \leq n$ et $1 \leq j \leq n$ correspondant aux longueurs des deuxièmes plus courts chemins entre i et j. Ecrire un algorithme de calcul de d et d'.

Correction exercice 16

1. On a $d^0_{ij} = 0$ si $i = j$ et $d^0_{ij} = \infty$ sinon.
2. On définit l'intérieur $I(c)$ du chemin $c = (s_1, \ldots, s_k)$ comme l'ensemble des sommets du sous-chemin $(s_2, \ldots, s_{k-1})$. Si $k \leq 2$ alors $I(c) = \varnothing$. Soient c un plus court chemin de i à j tel que $I(c) \subseteq \{s_1, \ldots, s_k\}$, c' un plus court chemin de i à s_{k+1} et $c"$ un plus court chemin de s_{k+1} à j.
   On aura $d^k_{i,k+1} = l(c')$ et $d^{k+1}_{k+1,j} = l(c")$. Ainsi $d^{k+1}_{ij} = \min(d^k_{ij}, d^k_{i,k+1} + d^k_{k+1,j})$.

3. Pour cela il suffit de considérer l'algorithme de Berge suivant basé sur un parcours en largeur.
4. Soit l'algorithme 4

Algorithm 4 Algorithme Long Premier(D :matrice)

```
i, j, k entiers
for k = 1 à n do
  for i = 1 à n do
    for j = 1 à n do
      d[i, j] := min(d[i, j], d[i, k] + d[k, j])
    end for
  end for
end for
```

5. La valeur d_{ij}^{k+1} de l'algorithme de fermeture transitive décrit précédemment est définie par :

$$d_{ij}^{k+1} = d_{ij}^{k} + d_{i,k+1}^{k} \times d_{k+1,j}^{k}$$

En remplaçant l'opération = par min et l'opération $\times$ par +, nous obtenons :

$$d_{ij}^{k+1} = \min(d_{ij}^{k}, d_{i,k+1}^{k} + d_{k+1,j}^{k})$$

6. On a $d_{ij}^{'0} = \infty, \forall i, j$
7. On peut aisément établir que :

$$d_{ij}^{'k+1} = \max(d_{ij}^{k}, \min(d_{ij}^{'k}, d_{i,k+1}^{k} + d_{k+1,j}^{k}))$$

En effet, on a $d_{ij}^{'k+1} = d_{ij}^{k}$ si $d_{ij}^{k} > \min(d_{ij}^{'k}, d_{i,k+1}^{k} + d_{k+1,j}^{k})$

8. Soit l'algorithme 5

Algorithm 5 Algorithme Long Deux(D :matrice)

```
i, j, k entiers
for k = 1 à n do
  for i = 1 à n do
    for j = 1 à n do
      d[i, j] := min(d[i, j], d[i, k] + d[k, j])
      d'[i, j] := max(d[i, j], min(d'[i, j], d[i, k] + d[k, j]))
    end for
  end for
end for
```

Fin correction exercice 16

Exercice 17 – Ordonnancement

Soit $G = (S, A, V)$ un graphe orienté valué, une source $s \in S$ et un sommet puits $p \in S$ (s n'a pas de prédécesseur, p n'a pas de successeur et pour tout sommet $x,\ x \neq s,\ x \neq p$ il existe un chemin de s à p passant pas $x,\ \mu = (s \dots x \dots p)$).

Une fonction potentiel $\Pi : S \to R$ doit satisfaire aux conditions suivantes :
— $\pi(s) = 0$,
— $\forall (x, y) \in A,\ \pi(x) - \pi(y) \geq v_{xy}$.
La quantité $\Pi(G) = \pi(p)$ est appelée potentiel du graphe G.

1. Montrer qu'une condition nécessaire et suffisante d'existence d'une fonction potentiel est que G ne possède pas de circuit de longueur strictement positive.
   Nous supposons maintenant que cette condition est satisfaite. Soit $\Delta(x)$ la longueur d'un plus long chemin d'origine s et d'extrémité x.
2. Montrer que $\Delta(x)$ définit une fonction potentiel.
   Une fonction potentiel π est dite minimale si $\Pi(G)$ est minimum.
3. Montrer que la fonction potentiel définie par les valeurs $\Delta(x)$ est minimale. Soit $\delta(x)$ la longueur d'un long chemin d'origine x et d'extrémité p.
4. Montrer que $\pi(x) = \Delta(p) - \delta(x)$ est une fonction potentiel minimale.

Correction exercice 17

1. Supposons que $(x_1, x_2, \dots, x_k, x_1)$ soit un circuit de longueur $c = v_{12} + \dots v_{k1}$ strictement positive et que Π soit une fonction potentiel.
   Les inégalités suivantes sont vérifiées par Π :

$$\pi(x_2) - \pi(x_1) \geq v_{12}, \pi(x_3) - \pi(x_2) \geq v_{32}, \dots, \pi(x_1) - \pi(x_k) \geq v_{k1}$$

   En sommant ces inégalités nous obtenons :

$$\pi(x_2) - \pi(x_1) + \pi(x_3) - \pi(x_2) + \dots + \pi(x_1) - \pi(x_k) \geq v_{12} + v_{23} + \dots + v_{k1}$$

   et après simplification $0 \geq c$ ce qui est une contradiction car $c > 0$.
   Si G ne possède pas de circuit de longueur strictement positive, il existe un plus long chemin de sà tout sommet x de G. Nous notons $\Delta(x)$ la longueur de ces chemins et nous montrons que $\pi(x) = \Delta(x)$ est une fonction de potentiel.
   Nous avons $\Delta(s) = 0$ car G ne contient pas de circuit de longueur strictement positive. considérons un arc $(x, y) \in A$. On a $\Delta(y) \geq \Delta(x) + v_{xy}$ car dans le cas contriare il y aurait un chemin $(s, \dots, x, y)$ de longueur supérieureà $\Delta(y)$; nous obtenons alors $\Delta(y) - \Delta(x) \geq v_{xy}$.
2. Cela a été démontré dans la preuve de la question précédente.
3. Supposons qu íl existe une fonction de potentiel Π telle que $\pi(p) > \Delta(p)$. Considérons un plus long chemin $(s = x_1, \dots, x_k = p)$ rreliant la source au puits. Pour ce chemin Π doit satisfaire $\pi(x_2) - \pi(x_1) \geq v_{12}, \dots, \pi(x_k) - \pi(x_{k-1}) \geq v_{k-1k}$ en faisant la somme de ces inégalités nous obtenons : $\pi(x_k) - \pi(x_1) \geq v_{12} + \dots + v_{k-1k}$ soit $\pi(p) \geq \Delta(p)$ car $\pi(s) = 0$, ce qui contredit notre hypothèse.
4. Montrons premièrement que les valeurs $\pi(x)$ définissent une fonction de potentiel : nous avons $\pi(s) = \Delta(p) - \delta(s)$; il suit d ela définition des valeurs $\delta(x)$ que $\delta(s) = \Delta(p)$ et alors $\pi(s) = 0$.

Soit un arc $(x, y) \in A$ on a

$$\pi(y) - \pi(x) = (\Delta(p) - \delta(y)) - (\Delta(p) - \delta(x)) = \delta(x) - \delta(y)$$

nous avons $\delta(y) + v_{xy} \leq \delta(x)$ puisque $\delta(x)$ est la longueur d'un plus long chemin de xà p; ainsi $\pi(y) - \pi(x) \geq v_{xy}$.
Montrons maintenant la minimalité :
nous avons $\Pi(G) = \pi(p) = \Delta(p) - \delta(p) = \Delta(p)$ et nous avons montrerà la question précédente que $\Delta(p)$ est minimale.

Fin correction exercice 17

Exercice 18 – Ordonnancement bis

Cet exercice consiste à montrer comment le calcul d'une fonction potentiel; définie dans l'exercice précédent, permet de déterminer un calendrier d'exécution des tâches d'un projet.

On considère un ensemble de tâches $\{1, \ldots, n\}$ de durée connnue $p_i \in N$. L'exemple suivant indique l'ensemble des tâches à réaliser pour la construction d'une maison ainsi que leur durée d'exécution en semaine.

tâche	description	durée
1	maçonnerie	7
2	pose de la charpente	3
3	électrticité et eau	8
4	pose de la toiture	1
5	façade	2
6	pose des fenêtres	1
7	plafonds	3
8	aménagement du jardin	1
9	peinture	2
10	emménagement	1

Les tâches reliées par des relations de précédence. Par exemple, il faut avoir fini de poser la charpente avant de commencer la toiture, et donc la tâche 2 précède la tâche 4.

On considère $G = (S, A, V)$ le graphe modélissant les relations de précédence. Il est obtenu de la façon suivante : les tâches sont les sommets et on a un arc $(i, j) \in A$ si la tâche i précède j. La valuation v_{ij} de l'arc (i, j) est la durée p_i de la tâche i.

1. Construisez le graphe de précédence.
2. Montrer que pour les précédences soient cohérentes G doit être sans circuit.
   On suppose que G est sans circuit et qu'il existe un sommet source s et un sommet puits p (s rt p sont des tâches fictives de durée nulle, s précédant toutes les tâches du projet et s succédant à toutes ces tâches).
3. Montrer qu'une fonction potentiel existe pour G.
4. Soient Π une fonction potentiel et $\mu = (i_1, \ldots, i_k)$ un chemin de G. Montrer que $\pi(i_k) \geq \pi(i_1)$. On cherche à déterminer un ordonnancement des tâches, c'està dire à calculer pour chasue tâche i du projet sa date de début d'exécution t_i de sorte que les contraintes de précédence soient satisfaites c'està dire $\forall (i, j) \in A,\ t_j \geq t_i + p_i$. Par convention, nous pososns $t_s = 0$ comme date de début du projet.

5. Montrer que les valeurs t_i définissent une fonction potentiel de G. On définit la durée d'un ordonnancement, notée D, comme la plus grande date de fin d'exécution d'une tâche : $D = \max_{i \in \{1,\ldots,n\}}(t_i + p_i)$
6. Montrer que $D = t_p$.
7. Montrer qu'un ordonnancement de durée D minimale correspondà une fonction potentiel minimale de G.
   Dans la suite, nous considérons des ordonnancements de durée minimale. La date d'exécution au plus tôt d'une tâche $\theta(i)$ est la date minimale de début de i dans tout ordonnancement : $\theta(i) = \min_{\Pi \in \Pi^*}\{\pi(i)\}$ où Π^* est l'ensemble des fonctions potentiel (des ordonnancements) de G. L'ordonnancement au plus tôt noté Θ est l'ordonnancement dans lequel la date de début de chaque tâche est sa date de début au plus tôt $\theta(i)$.
8. Montrer que l'ordonnancement au plus tôt Θ est obtenu en faisant $\theta(i) = \Delta(i)$ avec $\Delta(i)$ la longueur d'un plus long chemin $(s, \ldots, i)$. Utiliser l'algorithme de Bellman pour calculer Θ dans l'exemple présenté plus haut.
   La date d'exécution au plus tard d'une tâche $\tau(i)$ est la date maximale de début de i dans tout ordonnancement minimal : $\tau(i) = \max_{\Pi \in \Pi^*}\{\pi(i)\}$ où Π^* est l'ensemble des fonctions potentiel minimales de G. L'ordonnancement au plus tard noté T est l'ordonnancement dans lequel la date de début de chaque tâche est sa date de début au plus tard $\tau(i)$.
9. Montrer que l'ordonnancement au plus tard T est obtenu en faisant $\tau(i) = \Delta(p) - \delta(i)$ avec $\delta(i)$ la longueur d'un plus long chemin $(i, \ldots, p)$. Calculer T dans l'exemple présenté plus haut.
   On définit la marge d'une tâche comme étant la différence entre sa date de début au plus tôt et sa date de début au plus tard dans un ordonnancement de durée minimale : $M_i = \tau(i) - \theta(i)$. Une tâche i est dite critique lorsque $M_i = 0$.
10. Déterminer les marges et les tâches critiques pour l'exemple présenté plus haut.
11. Montrer qu'il existe un chemin $\mu = (s, \ldots, p)$. tel que toute tâche i de μ est critique.

Correction exercice 18

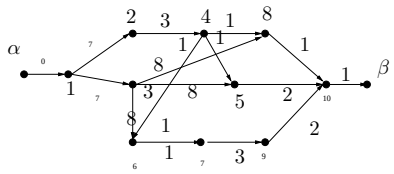


FIGURE 7 – Graphe de précédence

1. La solution est donnée par la figure 7.
2. Si G posède un circuit alors une tâche du circuit se précéde elle-même ce qui est incohérent.
3. G est sans circuit, il ne contient donc pas de circuit de longueur strictement positive. Il existe donc une fonction potentiel.

4. La fonction potentiel Π induit $\pi(i_k) - \pi(i_1) \geq v_{i_1 i_2} + \ldots + v_{i_{k-1} i_k} = p_{i_1} + \ldots + p_{i_{k-1}}$. Les durées des tâches sont positives, donc $\pi(i_k) - \pi(i_1) \geq 0$.

5. Nous avons $t_s = 0$ et pour tout arc $(i, j) \in A,\ t_j \geq t_i + p_i$ soit $t_j - t_i \geq p_i = v_{ij}$ ce qui correspondà la définition d'une fonction potentiel de G.

6. p étant un puits, pour toutes tâche $i \neq p$ il existe un chemin $(i = i_1, l\, dots, p = i_k)$ avec $t_p \geq t_I + p_i + \ldots + p_{i_{k-1}}$ car les valeurs t_i définissent une focntion potentiel. Les valautions étant positives, nous avons $t_p \geq t_i + p_i$ donc $D = \max_{i \in \{1,\ldots,n\}}(t_i + p_i) = t_p + p_p = t_p$.

7. Le potentiel de G est par définition $\Pi(G) = \pi(p) = D$. Une fonction potetntiel est minimale quand $\Pi(G)$ est minimum.

8. La preuve est identiqueà celle donnée pour la question 3 de l'exercie précédent. En ajoutant le sommet s prédécesseur du sommet 1 et le sommet p successeur du sommet 10 et en appliquant l'adaption de l'algorithme de Bellman pour le calcul des pluslong chemins d'origine s on obtient les dates au plus tôt suivantes :$\theta(s) = 0, \theta(1) = 0,\ \theta(2) = 7,\ \theta(3) = 7,\ \theta(4) = 10,\ \theta(5) = 15, \theta(7) = 16,\ \theta(8) = 15,\ \theta(9) = 19,\ \theta(10) = 21,\ \theta(p) = 22$.

9. Nous avons montré dans l'exercice précédent que les valeurs $\Delta(p) - \delta(i)$ définissent une fonction potentiel minimale. Si pour une tâche i nous avons $t_i > \tau(i) = \Delta(p) - \delta(i)$ alors $t_p > \Delta(p)$ et la fonction potentiel n'est pas minimale.

   En appliquant l'adaptation de l'algorithme de Bellman pour le calcul des plus longs chemins d'origine p (en inversant l'orientation des arcs de G) on obtient les dates au plus plus tard suivantes : $\tau(s) = 0,\ \tau(1) = 0,\ \tau(2) = 11,\ \tau(3) = 7,\ \tau(4) = 14,\ \tau(5) = 19,\ \tau(6) = 15,\ \tau(7) = 16,\ \tau(8) = 20,\ \tau(9) = 19,\ \tau(10) = 21,\ \tau(p) = 22$.

10. Nous obtenons les marges suivantes : $M_1 = 0,\ M_2 = 4,\ M_3 = 0,\ M_4 = 4,\ M_5 = 4,\ M_6 = 0,\ M_7 = 0,\ M_8 = 5,\ M_9 = 0,\ M_{10} = 0$. Les tâches critiques sont : $1, 3, 6, 7, 9$ et 10.

11. Soit $\mu = (s, \ldots, p)$ un chemin de longueur maximale de G. Pour tout sommet i de μ nous avons $\Delta(i) + \delta(i) = \Delta(p)$ avec $\Delta(i)$ la longueur d'un plus long chemin de s à i et $\delta(i)$ la longueur d'un plus long chemin de ià p. Nous obetnons alors pour la tâche $i,\ \theta(i) = \tau(i)$, ce qui implique $M_i = 0$.

Fin correction exercice 18