Corrections des Fiches de TD

R. Mandiau

fiche de TD 3

Exercice 1 : Quelles sont les conséquences de trois stratégies de passage de paramètre d'un tableau de N éléments

- Un tableau est passé par adresse : le temps d'exécution est constant $T(N) = \Theta(1)$
- Un tableau est passé par copie : le temps d'exécution est linéaire $T(N) = \Theta(N)$
- Un tableau est passé via une copie uniquement du sous-intervalle susceptible d'être utilisé par la procédure appelé : temps linéaire $T(N) = \Theta(q - p + 1)$ si le sous-tableau $A[p..q]$ est transmis

1. On considère l'algorithme récursif de recherche dichotomique consistant à trouver un nombre dans un tableau trié. Donner les récurrences correspondant aux temps d'exécution, pour chacune des stratégies proposées précédemment de passage de tableau, et donner des bonnes bornes supérieures pour les solutions de récurrences. N est la taille du problème initial et n la taille d'un sous-problème.
2. Même question pour l'algorithme Tri-fusion
3. idem pour le tri par insertion

Exercice 1 : Quelles sont les conséquences de trois stratégies de passage de paramètre d'un tableau de N éléments

Question 1. On considère l'algorithme récursif de recherche dichotomique consistant à trouver un nombre dans un tableau trié. Donner les récurrences correspondant aux temps d'exécution, pour chacune des stratégies proposées précédemment de passage de tableau, et donner des bonnes bornes supérieures pour les solutions de récurrences. N est la taille du problème initial et n la taille d'un sous-problème.

```
     Entrées : A : tableau[1.. MAX] d'entier ; g,d,val : entier ;
     Sorties : booleen ;
     Données : pivot : entier ;
1.1  début
1.2      si (g ≤ d) alors
1.3          pivot ← ⌊(g+d)/2⌋ ;
1.4          si (val = A[pivot]) alors
1.5              retourner VRAI ;
1.6          sinon
1.7              si (val < A[pivot]) alors
1.8                  retourner Dicho(A,g,pivot-1,val) ;
1.9              sinon
1.10                 retourner Dicho(A,pivot+1,d,val) ;
1.11     sinon
1.12         retourner FAUX ;
```

Algorithme 1 : Dicho(A,g,d,val)

La complexité spatiale est définie par :

$$S(n) = S_{fixe}(n) + S_{variable}(n)$$

Pour les besoins fixes :
— g, d, val, $pivot$: entier
— sortie : un booléen.

$$S_{fixe}(n) = 4 \cdot |entier| + 1 \cdot |booleen|$$

,

où : $|entier|$ désigne la taille d'un entier en octet (idem $|booleen|$ pour un booléen).

Cas 1. Un tableau est passé par adresse : le temps d'exécution est constant $T(N) = \Theta(1)$.
$S_{variable}(n) = |adresse_{Dicho}| \cdot \log_2(n)$; d'où $S(n) = O(\log_2(n))$.

Cas 2. Un tableau est passé par copie : le temps d'exécution est linéaire $T(N) = \Theta(N)$
$S(n) = O(n \cdot \log_2(n))$

Cas 3. Un tableau est passé via une copie uniquement du sous-intervalle susceptible d'être utilisé par la procédure appelé : temps linéaire $T(N) = \Theta(q - p + 1)$ si le sous-tableau $A[p..q]$ est transmis

$$S_{variable}(n) = n + \frac{n}{2} + \frac{n}{2^2} + \cdots + 1 = n \cdot (1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{\log_2(n)}})$$

Or $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ pour $|x| < 1$; d'où : $S_{variable}(n) = n \cdot \frac{1}{1-\frac{1}{2}} = 2 \cdot n$.

$$S(n) = O(n)$$

Fiche TD 4

Exercice 1 : Utiliser la méthode générale pour donner des bornes asymptotiques approchées pour les récurrences suivantes :

1. $T(n) = 2.T(\frac{n}{2}) + n$
2. $T(n) = 2.T(\frac{n}{2}) + 1$
3. $T(n) = 4.T(\frac{n}{2}) + n$
4. $T(n) = 4.T(\frac{n}{2}) + n^2$
5. $T(n) = 4.T(\frac{n}{2}) + n^3$

Exercice 2 : Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 7.T(\frac{n}{2})+n^2$. Un algorithme concurrent A' a un temps d'exécution décrit par $T'(n) = a'.T'(\frac{n}{4})+n^2$

1. Appliquer la méthode générale et la méthode des arbres récursifs pour déterminer les fonctions asymptotiques correspondantes.
2. Quelle est la plus grande valeur entière de a' telle que A' soit asymptotiquement plus rapide que A ?

Exercice 1 : Utiliser la méthode générale pour donner des bornes asymptotiques approchées pour les récurrences suivantes :

1. $T(n) = 2.T(\frac{n}{2}) + n$
2. $T(n) = 2.T(\frac{n}{2}) + 1$
3. $T(n) = 4.T(\frac{n}{2}) + n$
4. $T(n) = 4.T(\frac{n}{2}) + n^2$
5. $T(n) = 4.T(\frac{n}{2}) + n^3$

Rappel : Théorème de la la méthode générale (Bentley et al.)

Soient les constantes $a \geq 1$ et $b > 1$ et une fonction $f(n)$. Soit $T(n)$ défini pour les entiers non négatifs par la récurrence : $T(n) = a.T(\frac{n}{b}) + f(n)$; où l'on interprète $\frac{n}{b}$ comme signifiant $\lfloor \frac{n}{b} \rfloor$ ou $\lceil \frac{n}{b} \rceil$.

$T(n)$ peut alors être borné asymptotiquement de la façon suivante :

1. Si $f(n) = O(n^{\log_b(a)-\epsilon})$ pour une constante $\epsilon > 0$ alors $T(n) = \Theta(n^{\log_b(a)})$
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)}.\log_2(n))$
3. Si $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ pour une constante $\epsilon > 0$ et $a.f(\frac{n}{b}) \leq c.f(n)$ pour une constante $c < 1$ et n grand alors $T(n) = \Theta(f(n))$

Question 1 : $T(n) = 2 \cdot T(\frac{n}{2}) + n$

$a = 2$, $b = 2$ et $f(n) = n$. Comparaison entre deux grandeurs : $n^{\log_b(a)}$ et $f(n)$. La condition 2 : $n^{\log_b(a)} = n^{\log_2(2)} = n = f(n)$ est vérifiée ; donc $T(n) = \Theta(n \cdot \log_2(n))$.

Question 2 : $T(n) = 2 \cdot T(\frac{n}{2}) + 1$ $a = 2$, $b = 2$ et $f(n) = 1$. Comparaison entre deux grandeurs : $n^{\log_b(a)}$ et $f(n)$. La condition 1 : $n^{\log_b(a)-\epsilon} = n^{\log_2(2)-\epsilon} = n^{1-\epsilon} = 1 = f(n)$ est vérifiée (pour $\epsilon = 1$) ; donc $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n)$.

etc.

Exercice 2 : Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 7.T(\frac{n}{2})+n^2$. Un algorithme concurrent A' a un temps d'exécution décrit par $T'(n) = a'.T'(\frac{n}{4})+n^2$.

1. Appliquer la méthode générale et la méthode des arbres récursifs pour déterminer les fonctions asymptotiques correspondantes.
2. Quelle est la plus grande valeur entière de a' telle que A' soit asymptotiquement plus rapide que A ?

Question 1. Appliquer la méthode générale
A : $T(n) = 7.T(\frac{n}{2}) + n^2$. Si nous supposons $a = 7$, $b = 2$ et $f(n) = n^2$.

Nous pouvons effectuer la comparaison des grandeurs pour les rendre égales : $n^{\log_b(a)-\epsilon} = n^{\log_2(7)-\epsilon} = n^2 = f(n)$ (prenons $\epsilon = \log_2(7) - 2 > 0$). L'application de la condition 1 permet d'en déduire $T_A(n) = \Theta(n^{\log_2(7)})$.

L'algorithme A' est défini par la relation de récurrence $T'(n) = a' \cdot T'(\frac{n}{4}) + n^2$ (avec a' un paramètre). Nous supposons $a = a'$, $b = 4$ et $f(n) = n^2$.

Considérons $a' = 16$: Pour obtenir l'égalité des deux grandeurs, il faudrait que nous ayons : $n^{\log_b(a)} = n^{\log_4(a')} = n^2 = f(n)$, i.e. $\log_4(a') = 2$; d'où $a' = 16$. Dans ce cas (condition 2 vérifiée) $T_{A'}(n) = \Theta(n^{\log_4(a')} \cdot \log_2(n)) = \Theta(n^2 \cdot \log_2(n))$.
Il faudrait également effectuer une étude similaire pour $a' < 16$ et $a' > 16$.
Considérons $a' < 16$: Pour obtenir l'égalité des deux grandeurs, il faudrait que nous ayons : $f(n) = n^2 = \Omega(n^{\log_4(a')+\epsilon})$, i.e. $\epsilon = 2 - \log_4(a') > 0$ (vérifiée pour $a' < 16$).
Nous pourrions envisager la condition 3, si la seconde sous-condition est aussi vérifiée : $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ pour une constante $c < 1$ et n grand.

$$\begin{aligned} a' \cdot f(\tfrac{n}{4}) &\leq c \cdot f(n) \\ a' \cdot \tfrac{n^2}{4^2} &\leq c \cdot n^2 \\ \tfrac{a'}{16} &\leq c \end{aligned}$$

Prenons $c = \frac{a'}{16} < 1$. Nous pouvons appliquer la condition 3 du théorème, $T_{A'}(n) = \Theta(f(n)) = \Theta(n^2)$.
Considérons $a' > 16$: Les ordres de grandeur sont égales si et seulement si $f(n) = O(n^{\log_4(a')-\epsilon})$ pour $\epsilon = \log_4(a') - 2 > 0$ (vraie pour $a' > 16$). Nous appliquons le théorème sur la condition 1 qui donne $T_{A'}(n) = \Theta(n^{\log_4(a')})$.

Question 2. Quelle est la plus grande valeur entière de a' telle que A' soit asymptotiquement plus rapide que A ?

Pour rechercher la plus grande valeur entière de a'. Nous prenons le cas où $a' > 16$ qui a conduit à une complexité pour l'algorithme $A' : T_{A'}(n) = \Theta(n^{\log_4(a')})$; et nous devons le comparer avec l'algorithme A de complexité $T_A(n) = \Theta(n^{\log_2(7)})$.

$$\begin{array}{ll} T_{A'}(n) & \leq T_{A'}(n) \\ n^{\log_4(a')} & \leq n^{\log_2(7)} \\ \log_4(a') & \leq \log_2(7) \\ \frac{\log_2(a')}{\log_2(4)} & \leq \log_2(7) \\ \log_2(a') & \leq \log_2(7^2) \\ a' & \leq 49 \end{array}$$

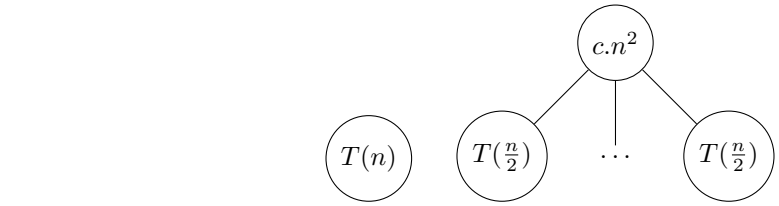
Conclusion : Prenons $a' = 49$.

Complétons le travail en utilisant la méthode des arbres récursifs : $T(n) = 7 \cdot T(\frac{n}{2}) + n^2$**.**

Trois étapes :

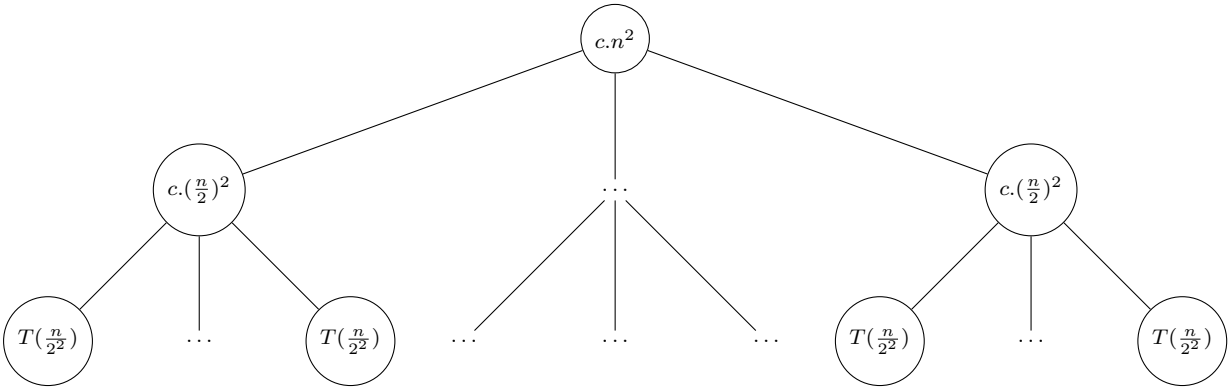
1. Phase 1 : Construction de l'Arbre Récursif
2. Phase 2 : Coût de chaque niveau
3. Phase 3 : Coût total

Phase 1 : Construction de l'Arbre Récursif

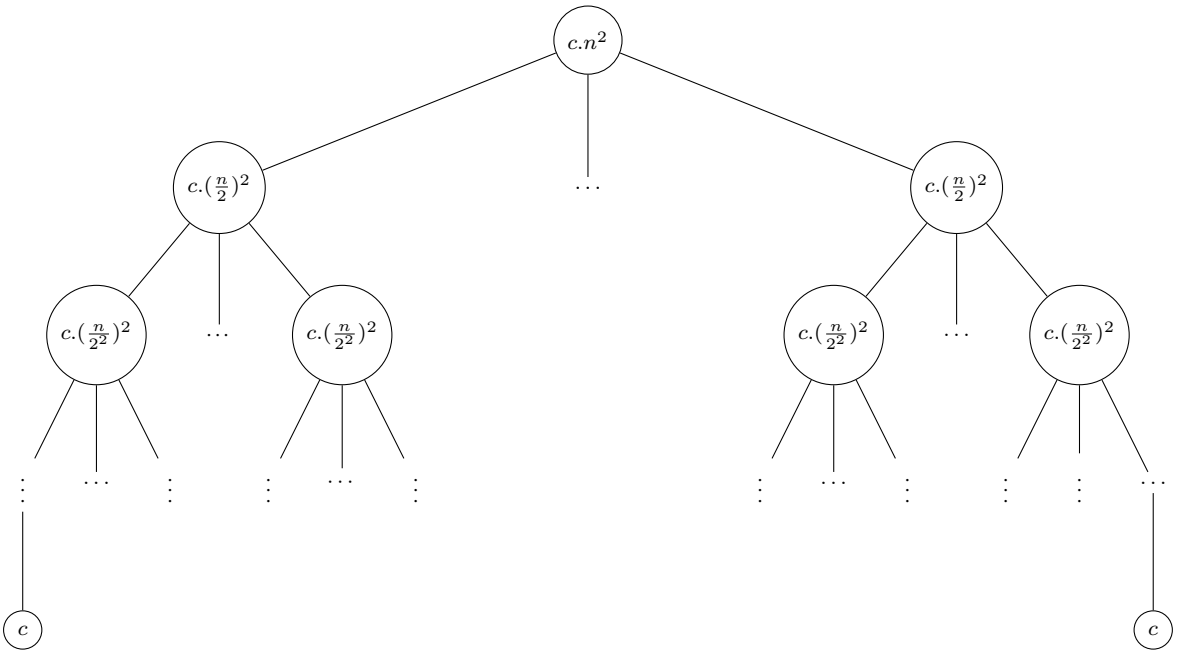


(a) Etape 0

(b) Etape 1



(c) Etape 2



(d) Dernière Etape

FIGURE 1 – Phase 1 : Construction de l'Arbre Récursif

Phase 2 : Coût de chaque niveau

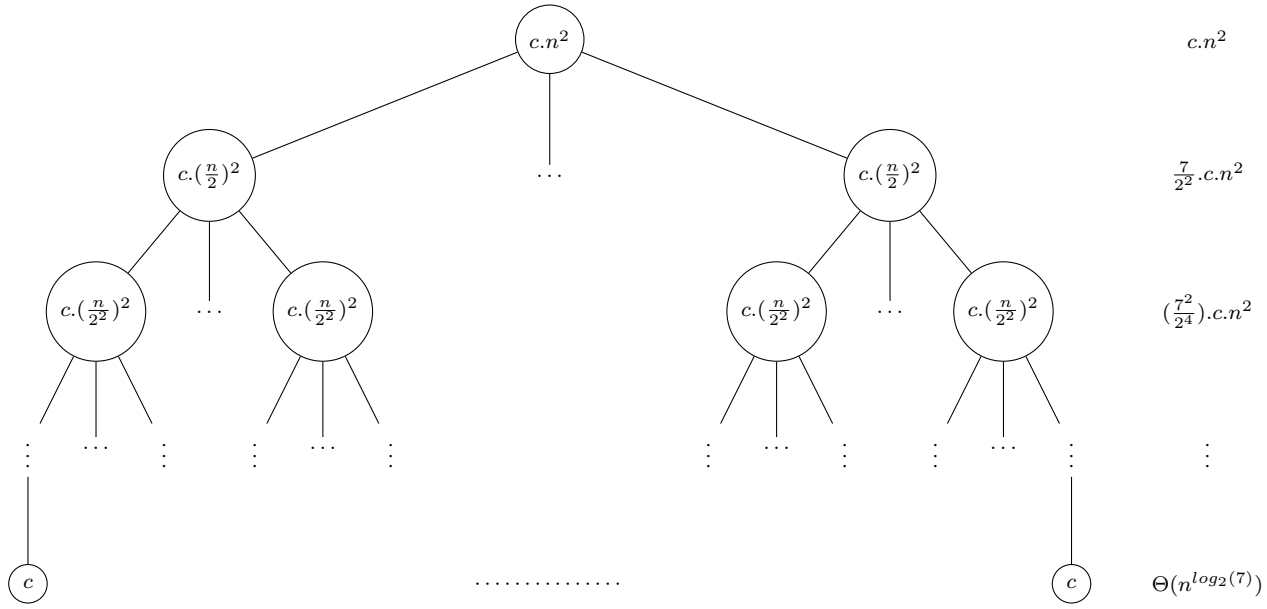


FIGURE 2 – Phase 2 : Coût de chaque niveau

1. **Hauteur de l'Arbre :** $n \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{4} \longrightarrow \ldots \longrightarrow 1$

$$\frac{n}{2^h} = 1 \textbf{ d'où } h = \log_2(n) \textbf{ (avec } h \textbf{ : hauteur)}$$

2. **Nombre de Niveaux et coûts :** $\log_2(n) + 1$ $(0, 1, 2, \ldots, \log_2(n))$: à chaque niveau i, le coût est de
$$c.(\frac{7}{4})^i.n^2$$

3. **Coûts du niveau terminal :** Rappelons que chaque niveau a 7^i nœuds. Au niveau de la feuille (i.e. de profondeur $\log_2(n)$) a un nombre de nœuds estimé à :
$$7^h = 7^{\log_2(n)} = n^{\log_2(7)}$$
(rappel : $a^{\log_b(c)} = n^{\log_b(a)}$)

Phase 3 : Coût total

$$T(n) = c.\left(n^2 + \frac{7}{4}.n^2 + (\frac{7}{4})^2.n^2 + \dots\right) + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\Sigma_{i=0}^{log_2(n)-1}(\frac{7}{4})^i.n^2 + \Theta(n^{\log_2(7)})$$

Or $\Sigma_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$, nous obtenons donc :

$$T(n) = c.\frac{(\frac{7}{4})^{log_2(n)} - 1}{\frac{7}{4} - 1}.n^2 + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left((\frac{7}{4})^{log_2(n)} - 1\right).n^2 + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(\frac{7}{4})} - 1\right).n^2 + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(7)-log_2(4)} - 1\right).n^2 + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(7)-2} - 1\right).n^2 + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(7)-2}.n^2 - n^2\right) + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(7)-2+2} - n^2\right) + \Theta(n^{\log_2(7)})$$

$$T(n) = c.\frac{4}{3}.\left(n^{log_2(7)} - n^2\right) + \Theta(n^{\log_2(7)})$$

$$T(n) \leq c.\frac{4}{3}.n^{log_2(7)} + \Theta(n^{\log_2(7)}) = c.\frac{7}{3}.n^{log_2(7)}$$

Pour conclure,

$$T(n) = \Theta(n^{log_2(7)})$$

même résultat avec le théorème de la méthode générale

Fiche TD 5

Exercice 1 : Décrire un algorithme en $\Theta(n.\log_2(n))$ qui étant donnés un ensemble S de n entiers et un autre entier x, détermine s'il existe ou non deux éléments de S dont la somme vaut exactement x.

Exercice 2 : Déterminer la complexité du pgcd de deux nombres strictement positifs n et m.

Exercice 3 : Trouver deux nombres entiers compris entre 1 et (N = 1000) tels, qu'étant multipliés entre eux ils donnent un certain produit, et que renversés ils donnent comme nouveau produit le premier renversé.

Exercice 4 : Proposer en pseudo-langage un algorithme de recherche de toutes les permutations d'une chaîne de caractères. Calculer sa complexité.

Exercice 5 : Le **problème des n-reines** consiste à placer n reines sur un échiquier de $n \times n$ cases, sans qu'aucune reine n'en menace une autre. Pour $n = 8$, nous pouvons montrer que le nombre de solutions est de 92. Comme dans le jeu d'échecs, la reine menace une autre pièce de l'échiquier située sur la même ligne, colonne ou diagonale.

1. Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.
2. En Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).
3. Décrire l'algorithme (algorithme de *backtracking*) :
   — L'algorithme *Libre* teste si la case à la ligne *lig* et à la colonne *col* n'est pas menacée par les reines déjà placées.
   — L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche).
4. En déterminer la complexité de l'algorithme.

Exercice 1 : Décrire un algorithme en $\Theta(n.\log_2(n))$ qui étant donnés un ensemble S de n entiers et un autre entier x, détermine s'il existe ou non deux éléments de S dont la somme vaut exactement x.

Algorithme 2 : Somme(A,n,x)

Entrées : $A[1..n]$ non triée, n entier, x : entier
Sorties : booléen

```
2.1 début
2.2     Trouve ← FAUX ; i ← 1 ;                                  ▷ Θ(1)
2.3     Fusion(A,n) ;                                            ▷ Θ(n.log2(n))
2.4     tant que (i ≤ (n − 1)) et (non Trouve) faire
2.5         si Dicho(A, i + 1, n, x − A[i]) alors
2.6             Trouve ← VRAI ;
2.7         sinon
2.8             i ← i + 1 ;                                      ▷ Θ(n.log2(n))
2.9     retourner Trouve ;
```

Exercice 2 : **Déterminer la complexité du pgcd de deux nombres strictement positifs n et m.**

exemple : pgcd (6,4)= 2.

n	m	reste
6	4	2
4	2	0

Algorithme 3 : pgcd(n,m)

Entrées : n entier, m : entier
Sorties : entier

```
3.1 début
3.2     reste ← n mod m ;                          ▷ Θ(1)
3.3     tant que (reste ≠ 0) faire
3.4         n ← m ;
3.5         m ← reste ;
3.6         reste ← n mod m ;
3.7     retourner m ;                              ▷ Θ(1)
```

Exercice. Proposer une version récursive équivalente.

Algorithme 4 : pgcd_rec(n,m)

Entrées : n entier, m : entier
Sorties : entier

```
4.1 début
4.2     reste ← n mod m ;
4.3     si (reste = 0) alors
4.4         retourner m ;
4.5     sinon
4.6         retourner pgcd_rec(n,m) ;
```

Supposons la suite (r_n) :

$$(r_n)\begin{cases} r_0 = n \\ r_1 = m \\ r_{j+1} = r_{j-1} \textbf{ mod } r_j, j \geq 2 \end{cases}$$

Le reste peut se définir par :

$$\begin{cases} reste = n - q.m, q \geq 1 \\ reste \leq n - m \textbf{ (a)} \\ reste \leq m - 1 \textbf{ (b)} \end{cases}$$

(a)+(b) : $2.reste \leq n - 1$, d'où $reste < \frac{n}{2}$.
Nous obtenons :

$$r_{j+1} < \frac{r_{j-1}}{2}$$

Par induction sur j, nous pouvons écrire les deux relations suivantes, selon la parité de j :

$$\begin{cases} r_j < \frac{r_0}{2^{\frac{j}{2}}} & \text{si } j \text{ est pair} \\ r_j < \frac{r_0}{2^{\frac{j-1}{2}}} & \text{si } j \text{ est impair} \end{cases}$$

Dans les deux cas, la relation suivante est vérifiée :

$$r_j < \frac{\max(n,m)}{2^{\frac{j}{2}}}$$

Dés que $r_j < 1$, la boucle **Tant que** se termine :

$$2^{\frac{j}{2}} = \max(n,m)$$

Par conséquent, le nombre de fois que la boucle **Tant que** est répétée, est égal à :

$$log_2(2^{\frac{j}{2}}) = log_2(\max(n,m))$$

$$\frac{j}{2} = log_2(\max(n,m))$$

$$j = 2.log_2(\max(n,m))$$

Le nombre d'itérations donne la complexité de l'algorithme en

$$T(n,m) = 2.log_2(\max(n,m))) + O(1)$$

d'où :

$$T(n,m) = O(log_2(\max(n,m)))$$

Exercice 3 : Proposer en pseudo-langage un algorithme de recherche de toutes les permutations d'une chaîne de caractères. Calculer sa complexité.

Algorithme 5 : permute(ch, g, d)

```
     Entrées : ch : chaine de car., g entier,d : entier
     Sorties :
5.1  début
5.2      si (g ≤ d) alors
5.3          pour (i ← g à d) faire
5.4              swap(ch,g,i) ;                          ▷ Θ(1)
5.5              permute(ch, g + 1, d) ;                 ▷ T(n − 1)
5.6              swap(ch,g,i) ;                          ▷ Θ(1)
5.7      sinon
5.8          Écrire ch
```

Nous supposons une procédure de permutation circulaire appelée **swap** de deux caractères (dont la complexité est en $\Theta(1)$).

La procédure **permute** (pour un élément de taille n) est de complexité :

$$T(n) = n.\left(T(n-1) + \Theta(1)\right)$$

$$\text{équivalent à : } T(n) = n.T(n-1) + n$$

Or par définition : $T(n-1) = (n-1).T(n-2) + (n-1)$ d'où :

$$T(n) = n.\left((n-1).T(n-2) + (n-1)\right) + n$$

$$T(n) = (n.(n-1).T(n-2)) + (n + n.(n-1))$$

d'où : $T(n) = (n.(n-1).\cdots .1) + (n + n.(n-1) + \cdots + n.(n-1).\cdots . 1)$
Revenons à la notation $n!$:

$$T(n) = n! + (n + n.(n-1) + \cdots + n!)$$

$$T(n) = n + n.(n-1) + \cdots + n! + n!$$

$$T(n) = \left(\frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \cdots + \frac{n!}{2!} + \frac{n!}{1!}\right) + n!$$

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

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

Rappel : $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. donc nous pouvons admettre que la série définie par la somme $\sum_{i=1}^{(n-1)} \frac{1}{i!}$ est bornée par une constante ($e \approx 2.7$) :

$$\sum_{i=1}^{(n-1)} \frac{1}{i!} \leq (e-1)$$

Ce raisonnement nous conduit à !

$$T(n) \leq n!.(e-1) + n!$$

$$T(n) \leq n!.(e-1+1)$$

$$T(n) \leq e.n!$$

Conclusion : e est une constante multiplicative ; d'où le résultat :

$$T(n) = O(n!)$$

n	Nombre de permutations	Temps CPU (sec.)
1	1	
2	2	
3	6	
4	24	
5	120	
6	720	
7	5 040	
8	40 320	0.001
9	362 880	0.012
10	36 288 000	0.111
11	39 916 800	0.933
12	479 001 600	11.3
13	6 227 020 800	155.05
14	87 178 291 200	-

Cf algorithmes de Steinhaus-Johnson-Trotter et algorithme de Heap [1] **en $O(n!)$ (début des années 60) et analyse de Sedgewick (fin des années 70) :**

[1] Johnson, Selmer M. (1963), "Generation of permutations by adjacent transposition", Mathematics of Computation, 17 : 282–285, doi :10.1090/S0025-5718-1963-0159764-2

[2] Knuth, Donald (2011), "Section 7.2.1.2 : Generating All Permutations", The Art of Computer Programming, volume 4A.

[3] Heap, B. R. (1963). "Permutations by Interchanges". The Computer Journal. 6 (3) : 293–4. doi :10.1093/comjnl/6.3.293.

[4] Sedgewick, R. (1977). "Permutation Generation Methods". ACM Computing Surveys. 9 (2) : 137–164. doi :10.1145/356689.356692

Algorithme de Heap

Algorithme 6 : Heap(ch, k)

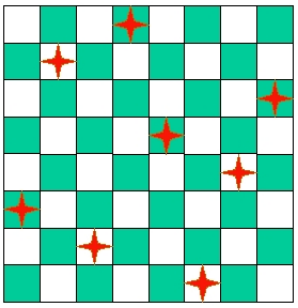
```
      Entrées : ch : chaine de car., k entier
      Sorties :
6.1   début
6.2     si (k = 1) alors
6.3       écrire ch ;
6.4     sinon
6.5       pour (i ← 0 à (k − 1)) faire
6.6         Heap(ch, k − 1);
6.7                                          ▷ éviter swap pour i = k − 1
6.8         si (i < k − 1) alors
6.9                                          ▷ dépend de la parité
6.10          si k est pair alors
6.11            swap(ch,i,k − 1);
6.12          sinon
6.13            swap(ch,0,k − 1);
```

1. Attention à ne pas confondre avec l'algorithme de tri heapsort en $O(n.log_2(n))$

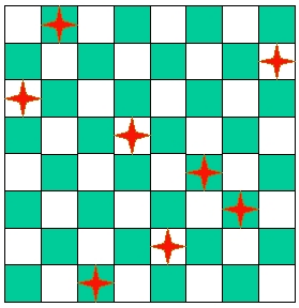
Exercice 4 : Le **problème des n-reines** consiste à placer n reines sur un échiquier de $n \times n$ cases, sans qu'aucune reine n'en menace une autre. Pour $n = 8$, nous pouvons montrer que le nombre de solutions est de 92. Comme dans le jeu d'échecs, la reine menace une autre pièce de l'échiquier située sur la même ligne, colonne ou diagonale.

1. Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.
2. En Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).
3. Décrire l'algorithme (algorithme de *backtracking*) :
   — L'algorithme *Libre* teste si la case à la ligne *lig* et à la colonne *col* n'est pas menacée par les reines déjà placées.
   — L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche).
4. En déterminer la complexité de l'algorithme.

Gauss (1777-1855) fut le premier à étudier ***problème des*** n***-reines*** *et le résolut partiellement en proposant 72 solutions pour* 8 *reines (il a fallu attendre* 1850 *avec Franck Nauck pour trouver toutes les solutions)* : cf. article de J-P Delahaye, "Le problème des 8 reines... et au-delà", Pour la Science, Dec. 2015, no 459.



Une solution ($N = 8$)



Pas une Solution

Question 1. Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.

Détermination du nombre de Reines : Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.

Définition : Le nombre max. de reines $R\,(n)$ pour un échiquier de taille $n \times n$ est $R\,(n) \leq n$.

Démonstration. On ne peut pas mettre deux reines sur la même ligne (resp. sur la même colonne) ; donc il est impossible d'avoir un nombre de reines ne vérifiant pas cette contraintes. □

Question 2. Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).

D'après la question précédente, il y a nécessairement une seule reine sur chaque ligne, et une seule reine sur chaque colonne. Elle consiste :

- à placer la première reine sur la première case libre de la première ligne,
- éliminer les cases désormais interdites et recommencer la même opération pour les lignes suivantes.
- S'il est impossible de placer la (k+1)eme reine, il faut revenir en arrière et déplacer la $k$eme reine sur la prochaine case libre, si elle existe (sinon on remonte à la ligne précédente).

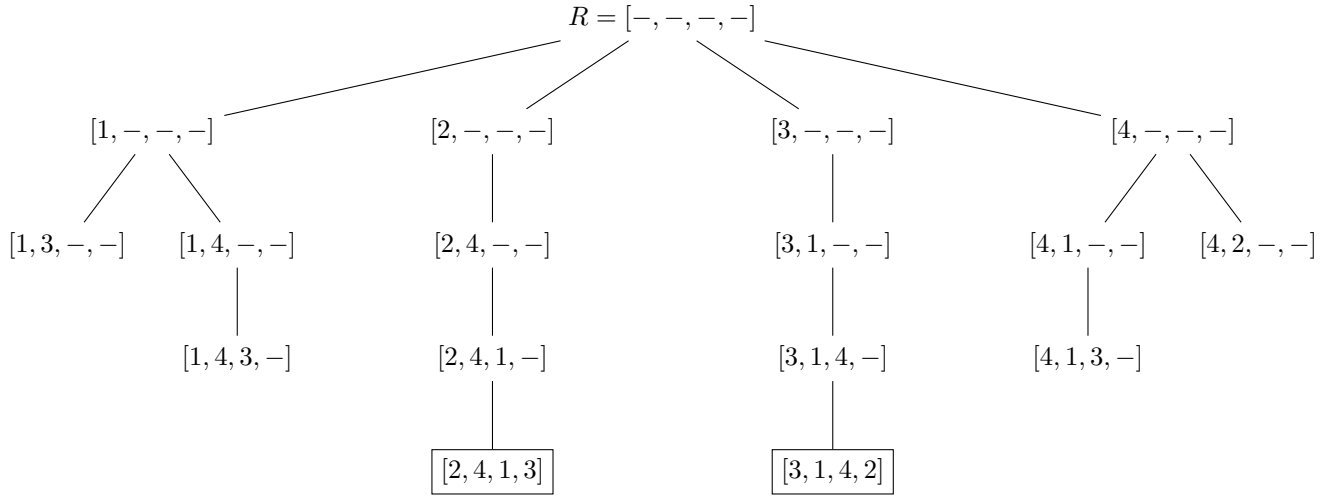


FIGURE 3 – Déroulement pour $N = 4$ Reines

Question 3. Décrire l'algorithme (algorithme de *backtracking*) :
— L'algorithme *Libre* teste si la case à la ligne *lig* et à la colonne *col* n'est pas menacée par les reines déjà placées.
— L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche).

L'algorithme *Libre* teste si la case à la ligne *lig* et à la colonne *col* n'est pas menacée par les reines déjà placées.

Algorithme 7 : Libre(R,lig,col)

```
    Entrées : R[1..n] tableau d'entiers, lig : entier, col : entier
    Sorties : booléen
7.1 début
7.2     ok ← true ;
7.3     i ← 1 ;
7.4     tant que i < lig faire
7.5         c ← R[i] ;
7.6         ok ← (ok) ∧ (|lig − i| ≠ |col − c|) ∧ (col ≠ c) ;
7.7         i ← i + 1 ;
7.8     retourner ok ;
```

L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche)

Algorithme 8 : Placer(R,n,$ligne$)

```
    Entrées : R[1..n] tableau d'entiers, n : entier, ligne : entier
8.1 début
8.2     si ligne > n alors
8.3         Afficher (R,n) ;
8.4     sinon
8.5         colonne ← 1 ;
8.6         tant que colonne ≤ n faire
8.7             si Libre(R, ligne, colonne) alors
8.8                 R[ligne] ← colonne ;
8.9                 Placer (R,n,ligne + 1) ;
8.10                R[ligne] ← 0 ;
8.11            colonne ← colonne + 1 ;
```

Appel de la procédure : Placer(R,n,1) avec n fixé.

Question 4. En déterminer la complexité de l'algorithme.

Démonstration. La fonction *Libre* dépend du nombre de cases libres sur une ligne donnée (diminuant d'une unité quand on passe à la ligne suivante). Cette fonction est donc satisfait au plus $n - k + 1$ fois à la ligne k.

Il est alors possible d'estimer le nombre d'opérations de *Placer* par un arrangement $A_n^k = \frac{n!}{(n-k)!}$ pour $k \leq n$.

Nous pouvons donc en déduire la complexité théorique

$$T(n) \leq \Sigma_{k=0}^{n-1} \frac{n!}{(n-k)!}.\Theta(n-k)$$

□

Résultats numériques : Pour information, le nombre de solutions est donné dans le tableau 1 :

TABLE 1 – Nombre de solutions pour n fixé

n	**Nombre de solutions**
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365 596
15	2 279 184

TABLE 2 – Mesure de Performances (en sec.) pour N-Reines

n	Notre Algo.	Algo trivial	min-contraintes
8	0.001	0.002	0.002
9	0.003	0.07	0.006
10	0.014	0.33	0.023
11	0.07	0.189	0.101
12	0.407	1.038	0.489
13	2.559	6.552	2.599
14	17.699	41.92	15.72
15	131.792	287.966	-
16	1010.08	2092.94	-

Comparaison entre différents algorithmes de recherche de toutes les solutions (Tableau 2) :

— **Notre Algo. N-Reines** : Notre algorithme proposé dans cet exercice
— **Algo trivial** : génération de toutes les solutions possibles avec test, pour chaque colonne
— **min-contraintes** : Extension des noeuds colonne par colonne, mais au lieu de choisir la colonne suivante, je prends celle qui a le plus de contraintes (i.e., dont le nombre de reines possibles positionnées à minimiser)

Fiche TD 6

Exercice 1 : **La suite de Fibonacci est définie par** 1, 1, 2, 3, 5 **etc. On veut calculer le** n**-ième terme de la suite de manière efficace**

1. Proposer un algorithme récursif. Donner le nombre d'appels récursifs de cet algorithme.
2. Proposer une variante où chaque valeur de la suite n'est calculée qu'une seule fois en utilisant de la mémoire.
3. Donner une solution où la complexité est logarithmique en n (optionnel)

$$(F_i) = \begin{cases} 0, & \text{si } i = 0 \\ 1, & \text{si } i = 1 \\ F_{i-1} + F_{i-2} & \text{sinon} \end{cases}$$

Exercice 2 : Le problème est de déterminer à partir de quel étage d'un immeuble sauter par la fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Il n'y a qu'une seule opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un(e) étudiant(e) par la fenêtre. S'il survit, vous pouvez le ré-utiliser ensuite, sinon, vous ne pouvez plus. Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (renvoyer $n+1$ si on survit encore au n étage) en faisant le minimum de sauts.

1. Si $k \geqslant \lceil \log_2(n) \rceil$, proposer un algorithme en $O(\log_2(n))$ sauts.
2. Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $O\left(k + \frac{n}{2^{k-1}}\right)$ sauts.
3. Si $k = 2$, proposer un algorithme en $2.\sqrt{n}$ sauts.
4. Dans ce dernier cas, proposer aussi un algorithme en $\sqrt{2.n}$ sauts.

Exercice 3 : Le **problème** k**-somme** peut être défini par : « Étant donnés un ensemble E de n entiers et un entier S, déterminer s'il existe $e_1, e_2, \ldots, e_k$ distincts dans E tels que $e_1 + e_2 + \ldots + e_k = S$. »

1. Proposer un algorithme simple pour résoudre le problème 2-somme avec une complexité quadratique
2. En généralisant cette méthode, en déduire une première borne supérieure polynomiale sur la complexité du problème k-somme, pour $k \geq 2$.
3. Proposer un algorithme permettant de résoudre le problème 2-somme sur une liste triée en complexité linéaire.
4. En déduire un algorithme de complexité quadratique pour 3-somme.
5. Proposer un algorithme de complexité $O(n.\log_2(n))$ pour 2-somme avec une liste non triée.
6. Question difficile : Proposer un algorithme de complexité $O(n^2.\log_2(n))$ pour 4-somme (on pourra utiliser des tableaux auxiliaires).
7. Question difficile : En déduire un algorithme résolvant k-somme lorsque k est pair en complexité $O(n^{\frac{k}{2}}.\log_2(n))$, et une variante en $O(n^{\frac{k+1}{2}})$ lorsque k est impair.

Exercice 1 : **La suite de Fibonacci est définie par 1, 1, 2, 3, 5 etc. On veut calculer le n-ième terme de la suite de manière efficace**

1. Proposer un algorithme récursif. Donner le nombre d'appels récursifs de cet algorithme.
2. Proposer une variante où chaque valeur de la suite n'est calculée qu'une seule fois en utilisant de la mémoire.
3. Donner une solution où la complexité est logarithmique en n (optionnel)

$$(F_i) = \begin{cases} 0, & \text{si } i = 0 \\ 1, & \text{si } i = 1 \\ F_{i-1} + F_{i-2} & \text{sinon} \end{cases}$$

Question 1. Proposer un algorithme récursif. Donner le nombre d'appels récursifs de cet algorithme.

version récursive : traduction immédiate de (F_i).

Algorithme 9 : Fibo_rec(n)

```
    Entrées : n : entier ;
    Sorties : entier
9.1 début
9.2     si (n = 0) alors
9.3         retourner 0;
9.4     sinon
9.5         si (n = 1) alors
9.6             retourner 1;
9.7         sinon
9.8             retourner Fibo_rec(n − 1)+ Fibo_rec(n − 2) ;
```

Soit (F_n) : $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$.
Nous pouvons deviner que F_n est exponentiel en n. Essayons de prouver que $F_n \leq \alpha c^n$?
Par hypothèse (raisonnement par induction), prouvons que :

$$F_n \leq \alpha.c^{n-1} + \alpha.c^{n-2} \leq \alpha.c^n$$

Cette inégalité est vérifiée si :

$$c^n \geq c^{n-1} + c^{n-2}$$

$$c^2 - c - 1 \geq 0$$

Il existe une solution à cette inéquation de second degré (l'autre solution est négative, donc ignorée).

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Nous avons ainsi une preuve inductive que : $F_n \leq \alpha.\Phi^n$, pour toute constante α.

Déterminons α : $\frac{F_0}{\Phi^0} = \frac{0}{1} = 0$, $\frac{F_1}{\Phi^1} = \frac{1}{\Phi} \approx 0.618$. Les conditions initiales sont vérifiées pour $\alpha \geq \frac{1}{\Phi}$. Donc :

$$F_n \leq \Phi^{n-1}$$

Effectuons le même raisonnement pour la borne inférieure ? $F_n \geq \beta.\Phi^n$, pour toute constante β. Nous pouvons montrer que les valeurs vérifiant les conditions initiales : $\frac{F_2}{\Phi^2} = \frac{1}{\Phi^2} \approx 0.381$ suppose $\beta \geq \frac{1}{\Phi^2}$. donc nous pouvons également en conclure :

$$F_n \geq \Phi^{n-2}$$

Nous pouvons conclure par la borne asymptotique : $F_n = \Phi^n$

$$T(n) = \Theta(\Phi^n)$$

Une autre version récursive

Algorithme 10 : Fibo_rec2(n, a, b)

```
     Entrées : n, a, b : entier ;
     Sorties : entier
10.1 début
10.2     si (n = 0) alors
10.3         retourner a;
10.4     sinon
10.5         si (n = 1) alors
10.6             retourner b;
10.7         sinon
10.8             retourner Fibo_rec2(n − 1, b, a + b) ;
```

L'appel *Fibo_rec*2(n,0,1) lance le processus de calcul. Les paramètres a et b servent d'accumulateurs. Le temps de calcul est proportionnel à n. Par contre l'espace mémoire occupé n'est *a priori* plus constant (sauf pour des langages réalisant l'optimisation de la récursivité terminale).

Mesure de Performance :

n	Récursif non terminal (Fibo_rec)	récursif terminal (Fibo_rec2)
20	0	0
30	0.008	0
40	0.65	0
50	78.4	0
51	125.36	0
...		...
10000		0
60000		0.001

Rappel : Une récursion terminale pour une fonction $f(.)$ est en position terminale si elle peut se traduire par une fonction itérative équivalente :

Algorithme 11 : Fonct(...)

```
11.1 début
11.2     si (R(x)) alors
11.3         retourner s(x);
11.4     sinon
11.5         Fonct(r(x));
```

Algorithme 12 : Fonct_iter(...)

```
12.1 début
12.2     tant que (¬R(x)) faire
12.3         x ← r(x);
12.4     retourner s(x);
```

Compléments sur les suites de Fibo :

k-bonacci : Des suites dont la relation de récurrence est d'ordre k. Un terme est la somme des k termes qui le précèdent. Parmi ces suites, on distingue la suite de **Tribonacci** (récurrence d'ordre 3) et la suite de **Tetranacci** (récurrence d'ordre 4). Selon ce nouveau classement de suites, la suite de Fibonacci serait une suite de 2-bonacci.

Si nous reprenons maintenant l'hypothèse d'une récurrence (Cf. algorithme 9) :

$$(F_i) = \begin{cases} 0, & \text{si } i = 0 \\ 1, & \text{si } i = 1 \\ F_{i-1} + F_{i-2} & \text{sinon} \end{cases}$$

Algorithme 13 : Bonacci-2(n)

```
Entrées : n : entier ;
13.1 début
13.2     si (n = 0) alors
13.3         retourner 0;
13.4     sinon
13.5         si (n = 1) alors
13.6             retourner 1;
13.7         sinon
13.8             retourner Bonacci-2(n − 1) + Bonacci-2(n − 2) ;
```

Proposons l'algorithme suivant pour la suite de Tribonacci :

$$(G_i) = \begin{cases} 0, & \text{si } i = 0 \\ 1 & \text{si } i \in \{1, 2\} \\ G_{i-1} + G_{i-2} + G_{i-3} & \text{sinon} \end{cases}$$

Algorithme 14 : Bonacci-3(n)

```
Entrées : n : entier ;
14.1 début
14.2     si (n = 0) alors
14.3         retourner 0;
14.4     sinon
14.5         si (n = 1)  ou  (n = 2) alors
14.6             retourner 1;
14.7         sinon
14.8             retourner Bonacci-3(n − 1) + Bonacci-3(n − 2) + Bonacci-3(n − 3) ;
```

idem pour la suite de Tetranacci (récurrence d'ordre 4).

n	2-bonacci (fibo rec non terminal)	3-bonacci	4-bonacci
20	0	0	0
25	0	0.008	0.01
30	0.008	0.103	0.25
35	0.061	2.195	6.802
36	0.1	3.98	13.12
37	0.15	7.25	25.491
38	0.24	13.5	49.1
39	0.397	25.19	94.85
40	0.65	46.49	**181.55**
41	1.046	85.44	
42	1.6	**156.45**	
45	7.1		
46	11.4		
47	18.5		
48	29.7		
49	48.63		
50	78.4		
51	**125.36**		

Proposition d'un algorithme généralisant k-bonacci :

Algorithme 15 : Bonacci_rec(n, k)

```
Entrées : n, k : entier ;
15.1  début
15.2      si (n = 0) alors
15.3          retourner 0;
15.4      sinon
15.5          si (n = 1) alors
15.6              retourner 1;
15.7          sinon
15.8              som ← 0;
15.9              pour i ← 1 à k faire
15.10                 si (n − i ≥ 0) alors
15.11                     som ← som+ Bonacci_rec(n − 1,k);
15.12             retourner som ;
```

Mesure de performance (version générale : algorithme 15) :

Pour rappel, nous avons obtenu avec les algorithmes précédents (Algorithmes 13 et 14) :
— $k = 2$: 125.4 secondes ($n = 51$)
— $k = 3$: 85.4 secondes ($n = 41$)
— $k = 4$: 49.1 secondes ($n = 38$)

n	2-bonacci	3-bonacci	4-bonacci
20	0	0.001	0
25	0.001	0.013	0.041
30	0.009	0.263	1.069
35	0.09	5.711	28.16
36	0.145	10.738	54.0
37	0.234	19.91	104.05
38	0.384	36.454	**200.06**
39	0.63	67.653	
40	1.01	123.36	
41	1.653	**226.12**	
42	2.694		
45	11.52		
46	18.88		
47	30.75		
48	50.3		
49	81.1		
50	131.39		
51	**213.45**		

(facultatif) Proposer une amélioration de l'algorithme 15 en considérant une version itérative.

Question 2. Proposer une variante où chaque valeur de la suite n'est calculée qu'une seule fois en utilisant de la mémoire.

Rappelons la complexité en itératif (classique) :

Algorithme 16 : Fibo_2(n)

```
    Entrées : n entier
    Sorties : entier
16.1 début
16.2     f_0 ← 0 ; f_1 ← 1;
16.3     f_2 ← n;
16.4     pour (i ← 2 à n) faire
16.5         f_2 ← f_0 + f_1;
16.6         f_0 ← f_1;
16.7         f_1 ← f_2;
16.8     retourner f_2;
```

La complexité de **Fibo_2** est $\Theta(n)$.

Nous pouvons améliorer cette version en utilisant que deux variables :

Algorithme 17 : Fibo_2bis(n)

```
    Entrées : n entier
    Sorties : entier
17.1 début
17.2     f_0 ← 0 ; f_1 ← 1;
17.3     pour (i ← 2 à n) faire
17.4         f_1 ← f_0 + f_1;
17.5         f_0 ← f_1 − f_0;
17.6     retourner f_1;
```

La complexité temporelle de **Fibo_2bis** est aussi en $\Theta(n)$.

Voici l'algorithme en récursif utilisant un tableau T.

Soit une procédure d'**Initialisation** du tableau auxiliaire T.

Algorithme 18 : Initialisation(n)

Entrées : n entier
Sorties : $T[1 \dots n]$: tableau de n entier

```
18.1 début
18.2     Créer un tableau T[1...n] à 0;
18.3     T[0] ← 0 ; T[1] ← 1;
```

Cette version est appelée « **memoization** », pour laquelle on regarde si la valeur dans le tableau a déjà été calculée. Si cette valeur existe, on la récupère sinon on la calcule et on la stocke.

Algorithme 19 : Fibo_rec(n)

Entrées : n entier
Sorties : entier

```
19.1 début
19.2     si (n < 2) alors retourner n                    ▷ Θ(1) ;
19.3     si (T[n − 1] = 0) alors
19.4         T[n − 1] ← Fibo_rec(n − 1) ;                ▷ T(n − 1)
19.5     T[n] ← T[n − 1] + T[n − 2] ;                    ▷ Θ(1)
19.6     retourner T[n] ;                                ▷ Θ(1)
```

La complexité de l'algorithme utilisant un tableau auxiliaire, dépend de l'appel récursif (ligne 4).

D'où :

$$T(n) = T(n-1) + \Theta(1)$$

Nous concluons :

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

Voici l'algorithme en itératif équivalent : On initialise le tableau et on le construit de manière itérative les différents éléments du tableau T.

Algorithme 20 : Fibo_iter(n)

Entrées : n entier
Sorties : entier

```
20.1 début
20.2     Créer un tableau T[1...n] à 0;
20.3     T[0] ← 0 ; T[1] ← 1;
20.4     pour (i ← 2 à n) faire
20.5         T[i] ← T[i − 1] + T[i − 2] ;
```

La complexité est alors

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

Question 3. Donner une solution où la complexité est logarithmique en n (optionnel)
Pour $n \geq 2$,

$$(F_n) \begin{cases} F_{n+1} = & 1.F_n + 1.F_{n-1} \\ F_n = & 1.F_n + 0.F_{n-1} \end{cases}$$

Nous pouvons caractériser la suite (F_n) par une représentation matricielle équivalente :

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

Donc, nous obtenons :

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Appelons I la matrice identité et M la matrice initiale :

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad\qquad M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Nous pouvons reformuler de la manière suivante :

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = M^n \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Les premiers résultats sont donnés pour information :

$$M^0 = \begin{bmatrix} 1 & 0 \\ \mathbf{0} & 1 \end{bmatrix} \quad M^1 = \begin{bmatrix} 1 & 1 \\ \mathbf{1} & 0 \end{bmatrix} \quad M^2 = \begin{bmatrix} 2 & 1 \\ \mathbf{1} & 0 \end{bmatrix} \quad M^3 = \begin{bmatrix} 3 & 2 \\ \mathbf{2} & 0 \end{bmatrix} \quad M^4 = \begin{bmatrix} 5 & 3 \\ \mathbf{3} & 0 \end{bmatrix} \quad M^5 = \begin{bmatrix} 8 & 5 \\ \mathbf{5} & 0 \end{bmatrix}$$

Évidemment, il nous est impossible d'utiliser l'algorithme trivial de calcul des produits de matrices. En effet, le produit de deux matrices d'ordre m serait $O(m^2)$; et la puissance d'une matrice d'ordre m à la puissance n serait de l'ordre $T(n, m) = O((m^2)^n) = O(m^{2.n})$. Dans notre cas, cela conduirait à une complexité de $T(n) = O(4^n)$.

Pour obtenir une complexité logarithmique, l'analyse conduit alors à une **exponentiation matricielle**.

Pour rappel : Une exponentiation rapide est définie par :

$$p(x, n) = \begin{cases} x & \text{si } n = 1 \\ p(x^2, \frac{n}{2}) & \text{si } n \text{ est pair} \\ x.p(x^2, \frac{n-1}{2}) & \text{si } n \text{ est impair} \end{cases}$$

Algorithme 21 : Puissance(M, n)

Entrées : M : matrice ; n : entier ;
Sorties : matrice

```
21.1  début
21.2      si (n = 0) alors
21.3          retourner I ;                          ▷ I : Matrice identité
21.4      sinon
21.5          si (n = 1) alors
21.6              retourner M ;                      ▷ M^1
21.7          sinon
21.8              n2 ← ⌊n/2⌋ ;
21.9              Temp ← Puissance(M^2, n2) ;        ▷ Creer Temp une matrice d'ordre 2
21.10             si (n mod 2 = 1) alors
21.11                 Temp ← M * Temp ;              ▷ n impair
21.12             retourner Temp ;
```

Algorithme 22 : Fibo_log(n)

Entrées : n : entier ;
Sorties : entier

```
22.1  début
22.2      Créer matrice M;
22.3      Créer matrice res;
22.4      res ← Puissance (M, n);
22.5      retourner res[2][1] ;                      ▷ Obtention F_n
```

Conclusion de l'algorithme *Fibo_log* **:**

$$T(n) = \Theta(log_2(n))$$

D'un point de vue pratique, le calcul de fibonacci en complexité logarithmique dépend essentiellement du nombre entier max. (**tester la limite ... , à faire en TP**).

Revenons au cours ... le Problème SAT.

Définition : Une forme normale conjonctive (FNC) est définie par :

$$C_1 \wedge C_2 \wedge \cdots C_i \wedge \cdots \wedge C_m$$

où C_i (une clause) est une expression de la forme :

$$x_1 \vee x_2 \vee \cdots x_j \vee \cdots \vee x_k$$

Chaque x_j est un littéral positif ou négatif (p_j ou $\neg p_j$).

Exemple La formule $(p_1 \vee \neg p_2) \wedge p_3 \wedge (p_4 \vee p_5 \vee p_6)$ est une FNC.

Question 1. Définir le « Problème de décision **SAT** ».

Étant donné un ensemble U de n variables binaires $x_1, \cdots, x_n$ et un ensemble C de m clauses, on cherche un modèle pour la formule $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, c'est-à-dire une affectation de valeurs booléennes $\{0, 1\}$ tel que $\mu(\Phi) = 1$. Le problème de la satisfiabilité (appelé **SAT**) d'un FNC pourrait s'écrire :

— **Instance :** « une formule en FNC ».
— **Question :** « déterminer s'il existe une fonction d'interprétation μ dont l'évaluation des variables booléennes satisfaisant Φ » ?

Rappel (Théorème de Cook 1971 : *The complexity of Theorem Proving Procedures, ACM 1971*) : Le problème **SAT** est **NP – complet**.

Exemple :
— $p \wedge \neg q$ est satisfiable ssi p VRAI et q FAUX
— $p \wedge \neg p$ est insatisfiable

Complexité : $O(2^n)$ pour les recherches exhaustives : Algorithmes classiques de recherche dans des graphes (problématique orientée notamment IA)
— Breath-First Search
— Depth-First Search
— Floyd-Warshall
— Davis et Putnam (DP 1960 et DPLL 1962)
— ...

Exercice 2 : Logiques Booléennes (exercice 48 page 81) Notons p_i une variable booléenne, un littéral est désigné par p_i ou $\neg p_i$. Une clause est une disjonction de littéraux, $p_i \vee \neg p_j$

Un problème de décision π est défini par : Instance : Un ensemble U de variables booléennes. Un ensemble C de conjonction de clauses défini par : $C = \{c_1, c_2, \ldots, c_n\} \,/\, |c_i| = m, 1 \leq i \leq n$. Par exemple nous pouvons avoir $C = \{p_1 \vee \neg p_2, \neg p_1 \vee p_2\}$.

Question : Existe-t-il une assignation de variables booléennes satisfaisant C ? (pour notre exemple, si p_1 et p_2 sont à valeur vraie, C est satisfait)

1. Montrer qu'un problème 2-SAT (i.e. des clauses dont le nombre de variables est $m = 2$) est résolu en polynomial (classe P).
2. Montrer que 3-SAT est NP-Complet ?
3. Le problème 3-SAT* est un cas particulier de 3-SAT ou dans chaque clause, on a que des littéraux positifs ou négatifs. Montrer que 3-SAT* est NP-complet ?

Question 1 : et quid de 1-SAT ?

Exemple $p_1 \wedge p_2 \wedge \ldots \wedge p_n$

S'il existe un littéral positif p_i et un littéral négatif $\neg p_i$ alors la fbf est satisfiable, sinon insastifiable.

1-SAT est bien résolu en temps polynomial : **Complexité :** $O(n)$

Conclusion : 1-SAT $\in$ **P**

Question 1 : 2-SAT ?

Montrer qu'un problème 2-SAT (i.e. des clauses dont le nombre de variables est $m = 2$) est résolu en polynomial (classe P).

Instance : Une formule 2-SAT Φ
Question : Φ est-elle satisfiable ?
Exemple Soit Φ défini par : $(p_1 \vee p_2) \wedge (p_1 \vee \neg p_3) \wedge (\neg p_2 \vee \neg p_4) \wedge (\neg p_1 \vee p_4)$.

Nous supposons :

C_1 $p_1 \vee p_2$

C_2 $p_1 \vee \neg p_3$

C_3 $\neg p_2 \vee \neg p_4$

C_4 $\neg p_1 \vee p_4$

Pour n variables et m clauses :

1. Créer un graphe avec $2n$ noeuds : $\{p_1, \neg p_1, \ldots, p_i, \neg p_i, \ldots, p_n, \neg p_n\}$. Intuitivement, chaque noeud est vrai ou faux pour chaque variable.
2. Relier les noeuds avec une arête $a \to b$ si on a la condition : « si a alors b » (i.e. $\neg a \vee b$)
   — Pour chaque clause $x_i \vee x_j$: créer une arête de $\neg x_i$ vers x_j et une arête de $\neg x_j$ vers x_i

C_1	$p_1 \vee p_2$	$\neg p_1 \to p_2$	$\neg p_2 \to p_1$
C_2	$p_1 \vee \neg p_3$	$\neg p_1 \to \neg p_3$	$p_3 \to p_1$
C_3	$\neg p_2 \vee \neg p_4$	$p_2 \to \neg p_4$	$p_4 \to \neg p_2$
C_4	$\neg p_1 \vee p_4$	$p_1 \to p_4$	$\neg p_4 \to \neg p_1$

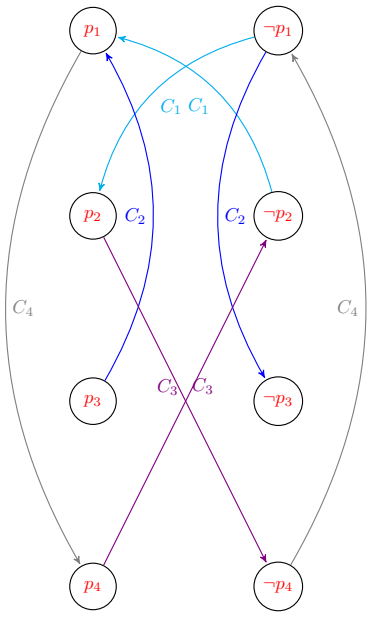


FIGURE 4 – Graphe de dérivation

Pour n variables et m clauses :

1. Créer un graphe avec $2n$ noeuds
2. Relier les noeuds avec une arête $a \to b$
3. Une formule 2-SAT Φ est insastisfiable ssi Pour chaque variable p_i, vérifier s'il existe un cycle contenant p_i et $\neg p_i$ (algorithme en temps polynomial : Complexité : $O(n^2)$)
4. donc **2-SAT est dans la classe P.**

Question 2 : Montrer que 3-SAT est NP-Complet ?

Il faut **prouver** les deux conditions suivantes (Cf. définition 32 page 67) :

1. 3-SAT $\in$ **NP** (Il existe un DTM à temps polynomial qui calcule f)
2. **SAT**$\propto$ 3-SAT (On dit que SAT est transformée en 3-SAT)

La première condition est triviale : Nous pouvons établir un algorithme qui vérifie que si pour une solution donnée, la formule est satisfaite. Il reste à prouver la **réduction**.

Soit un problème **SAT** tel que :
$\Phi = \bigwedge_i^n C_i$ avec $C_i = (z_1 \vee z_2 \vee \ldots \vee z_k)$.

$\underline{k = 1}$: chaque clause $C = z_1$
Soit $C' = (z_1 \vee x_1 \vee x_2) \wedge (z_1 \vee \neg x_1 \vee \neg x_2) \wedge (z_1 \vee \neg x_1 \vee x_2) \wedge (z_1 \vee x_1 \vee \neg x_2)$.
C' est instance de 3-SAT : nous voyons qu'elle est satisfaite ssi C est vraie. La réduction est prouvée pour ce cas.

$\underline{k = 2}$: chaque clause $C = z_1 \vee z_2$
Posons $C' = (z_1 \vee z_2 \vee x_1) \wedge (z_1 \vee z_2 \vee \neg x_1)$.
C' est une instance de 3-SAT qui est aussi satisfaite ssi C est vraie. La réduction est prouvée pour ce cas.

$\underline{k = 3}$: Chaque clause est deja dans 3-SAT (les deux problèmes sont identiques).

$\underline{k > 3}$: $C = (z_1 \vee z_2 \vee \ldots \vee z_k)$.
Posons

$$\begin{aligned} C' = \quad & (z_1 \vee z_2 \vee x_1) \wedge \\ & (\neg x_1 \vee z_3 \vee x_2) \wedge \\ & (\neg x_2 \vee z_4 \vee x_3) \wedge \\ & \ldots \wedge \\ & (\neg x_{k-4} \vee z_{k-2} \vee x_{k-3}) \wedge \\ & (\neg x_{k-3} \vee z_{k-1} \vee z_k) \end{aligned}$$

C' est une instance 3-SAT. Il faut juste prouver que $C \equiv C'$:
— $C \rightarrow C'$: si C est vrai, cela signifie qu'il existe un littéral z_i qui est vrai. La clause correspondante C est donc vraie.
— $C' \rightarrow C$: preuve par l'absurde

Conclusion : **3-SAT est NP-complet**.
Algorithme « Random walk algorithm » proposé par Schoning (1999).
Cet algorithme pour 3-SAT est en complexité $O(n^{1.34})$, meilleur algorithme connu pour 3-SAT.

Note importante : Un problème peut être NP-complet et avoir une complexité polynomiale.

Question 3 : 3-SAT* ? **Le problème 3-SAT* est un cas particulier de 3-SAT ou dans chaque clause, on a que des littéraux positifs ou négatifs. Montrer que 3-SAT* est NP-complet ?**

Autres variantes

— Clauses de Horn (Horn-SAT) : clauses contenant un seul littéral négatif
— MAX-SAT : déterminer le nombre maximum de clauses satisfaites