Bases de la Complexité

S3INE / O5E52ICY : Licence 3 Informatique / ICY 3A

R. Mandiau

Université Polytechnique Hauts-de-France
Valenciennes (France)
e-mail: Rene.Mandiau@uphf.fr

Plan du cours I

Plan

Revenons sur l'algorithme tri fusion I

Exercice 1 : Supposons que nous proposons un tri fusion avec deux partitions :

- $\frac{1}{3}$ pour la première partition
- la seconde est un partitionnement en $\frac{2}{3}$

Déterminer la complexité (cas défavorable) du nouvel algorithme ?

Revenons sur l'algorithme tri fusion II

Algorithme 1 : Tri_Fusion2(A,p,r)

Entrées : $A[1..n]$ non triée , p, r : entier
Sorties : $A[1..n]$ triée

début
  si $p < r$ **alors**
    $q \leftarrow \lfloor \frac{p+r}{3} \rfloor$;
    Tri_fusion2 (A,p,q) ;
    Tri_fusion2 (A,$q+1$, r) ;
    Fusion (A,p,q,r)

Déterminer la relation de récurrence I

Relation de récurrence pour la nouvelle version du tri fusion (appelée *Tri_fusion2*) : $T(n) = T(\frac{n}{3}) + T(\frac{2.n}{3}) + O(n)$

Démonstration.

$T(n) = T\_Fusion2(A, 1, \lfloor \frac{1+n}{3} \rfloor) + T\_Fusion2(A, \lfloor \frac{1+n}{3} \rfloor + 1, n) + T\_Fusion(A, 1, \lfloor \frac{1+n}{3} \rfloor, n)$

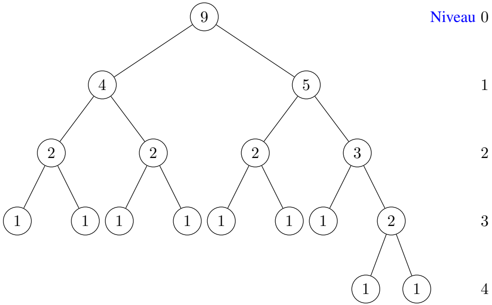
Or $T\_Fusion(n) = O(n)$

Donc : $T(n) = T(\frac{n}{3} \rfloor) + T(\lfloor \frac{2.n}{3} \rfloor) + O(n)$ $\square$

Conjecture ? ? ? ? ? non évident

Hypothèse pour la construction de l'arbre récursif I

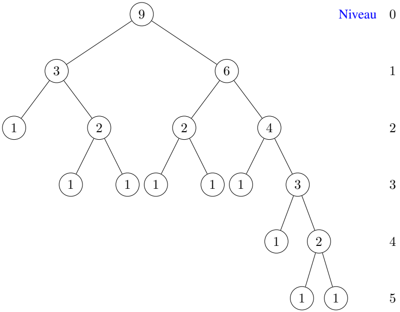
Pour le tri Fusion classique



- Note : $h = \log_2 9 \approx 3.2$

Hypothèse pour la construction de l’arbre récursif II

Nouvelle version ...



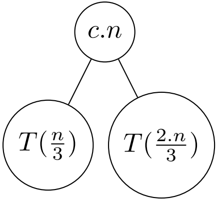
- Note : $\log_{\frac{3}{2}} 9 \approx 5.4$ et $\log_3 9 = 2$
- Hypothèse : **Prendre l’hypothèse que le coût est uniformément distribué** pour les différents niveaux de l’arbre

Etape 0 : $T(n) = T(\frac{n}{3}\rfloor) + T(\lfloor\frac{2.n}{3}\rfloor) + O(n)$ I

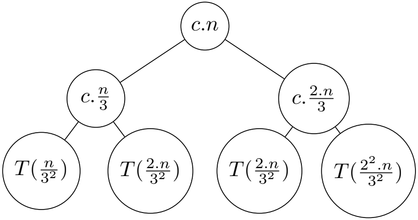
Différentes étapes pour la construction de l'arbre récursif

$$T(n)$$

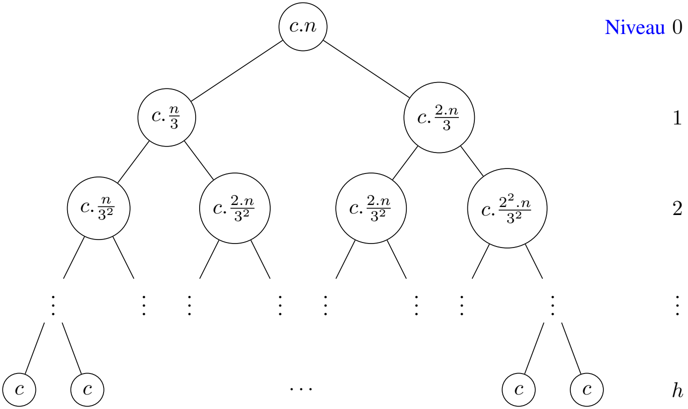
Etape 1 : $T(n) = T(\frac{n}{3}\rfloor) + T(\lfloor\frac{2.n}{3}\rfloor) + O(n)$ I



Etape 2 : $T(n) = T(\frac{n}{3}\rfloor) + T(\lfloor\frac{2.n}{3}\rfloor) + O(n)$ I



Etape h : $T(n) = T(\frac{n}{3}\rfloor) + T(\lfloor\frac{2.n}{3}\rfloor) + O(n)$ I



Application arbres récursifs I

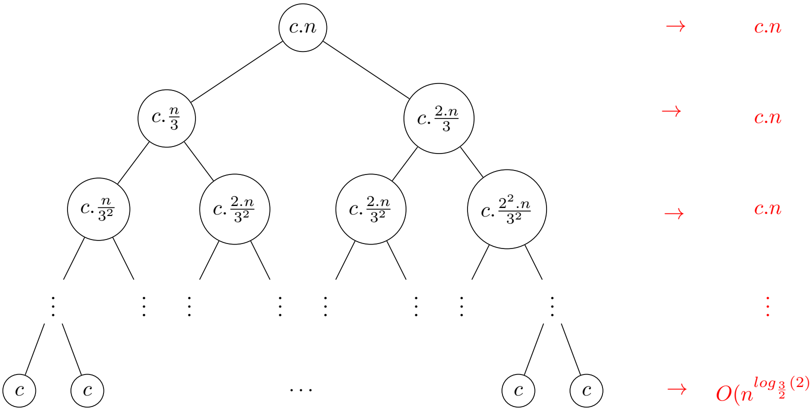
Methode des arbres récursifs : Principe en 3 phases

Méthode de construction d'arbre permettant de trouver une « bonne conjecture ».

1. Déterminer le coût de chaque nœud (coût d'un sous-problème individuel)
2. Totaliser les coûts pour chaque niveau de l'arbre
3. Cumuler les coûts de chaque niveau pour obtenir le coût total, puis vérifier avec la méthode de substitution

Recherche du plus long chemin pour la hauteur I

Phase 1 : coût de chaque nœud



Recherche du plus long chemin pour la hauteur II

Phase 2 : coût de chaque niveau

1. **Hauteur de l'Arbre :** prendre le plus long chemin
$n \longrightarrow \frac{2}{3}.n \longrightarrow (\frac{2}{3})^2.n \longrightarrow \dots \longrightarrow 1$

$$(\frac{2}{3})^h.n = 1 \textbf{ d'où } h = \log_{\frac{3}{2}}(n)$$

.

2. **Nombre de Niveaux et coûts :** $\log_{\frac{3}{2}}(n) + 1$
$(0, 1, 2, \dots, \log_{\frac{3}{2}}(n))$, à chaque niveau, le coût est de $c.n$
3. **Coûts du niveau terminal :** Rappelons que chaque niveau a 2^i nœuds. Au niveau de la feuille (i.e. de profondeur $\log_{\frac{3}{2}}(n)$) a un nombre de nœuds estimé à

$$2^h = 2^{\log_{\frac{3}{2}}(n)} = n^{\log_{\frac{3}{2}}(2)}$$

Recherche du plus long chemin pour la hauteur III

Phase 3 : coût total

- **Prendre l'hypothèse que le coût est uniformément distribué** pour les différents niveaux de l'arbre (En fait, l'arbre n'est pas un arbre binaire complet et ne donne pas toujours $c.n$).

Démonstration.

Le coût total est déterminé par :

$$T(n) = \sum_{i=0}^{\log_{\frac{3}{2}}(n)-1} (c.n) + \Theta(n^{\log_{\frac{3}{2}}(2)})$$

$$T(n) \approx c.n.\log_{\frac{3}{2}}(n) + \Theta(n^{1.71})$$

Donc $T(n) = O(n.\log_{\frac{3}{2}}(n))$

□

Recherche du plus long chemin pour la hauteur IV

En résumé, nous obtenons :

Hauteur	$\log_{\frac{3}{2}}(n)$
Niveau	$\log_{\frac{3}{2}}(n) + 1$
coût d'un niveau i	$c.n$
coût du niveau racine	$\Theta(n^{\log_{\frac{3}{2}}(2)})$
coût total	$T(n) = O(n.\log_{\frac{3}{2}}(n))$

Recherche du plus court chemin pour la hauteur I

- Effectuer un travail similaire pour déterminer sa complexité en prenant le plus court chemin pour calculer h.

Phase 2 : coût de chaque niveau (Phase 1 Cf. arbre rec.)

1. Hauteur de l'Arbre : prendre le plus long chemin
$n \longrightarrow \frac{1}{3}.n \longrightarrow (\frac{1}{3})^2.n \longrightarrow \ldots \longrightarrow 1$

$$(\frac{1}{3})^h.n = 1 \textbf{ d'où } h = \log_3(n)$$

.

2. Nombre de Niveaux et coûts : $\log_3(n) + 1$ $(0, 1, 2, \ldots, \log_3(n))$, à chaque niveau, le coût est de $c.n$
3. Coûts du niveau terminal : Rappelons que chaque niveau a 2^i nœuds. Au niveau de la feuille (i.e. de profondeur $\log_3(n)$) a un nombre de nœuds estimé à

$$2^h = 2^{\log_3(n)} = n^{\log_3(2)}$$

Recherche du plus court chemin pour la hauteur II

Phase 3 : coût total

- **Prendre l'hypothèse que le coût est uniformément distribué** pour les différents niveaux de l'arbre.

Démonstration.

Le coût total est déterminé par :

$$T(n) = \sum_{i=0}^{\log_3(n)-1}(c.n) + \Theta(n^{\log_3(2)})$$

$$T(n) = c.n.\log_3(n) + \Theta(n^{0.63})$$

Donc $T(n) = \Omega(n.\log_3(n))$

□

Recherche du plus court chemin pour la hauteur III

En résumé, nous obtenons :

Hauteur	$\log_3(n)$
Niveau	$\log_3(n) + 1$
coût d'un niveau i	$c.n$
coût du niveau racine	$\Theta(n^{\log_3(2)})$
coût total	$T(n) = \Omega(n.\log_3(n))$

première conclusion I

Nous pouvons en déduire que la complexité varie :

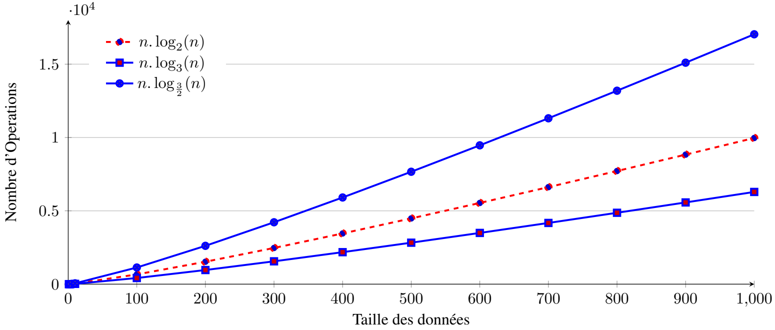
- $\Omega(n.\log_3(n))$
- $O(n.\log_{\frac{3}{2}}(n))$

Est ce suffisant ? ? ?

première conclusion II

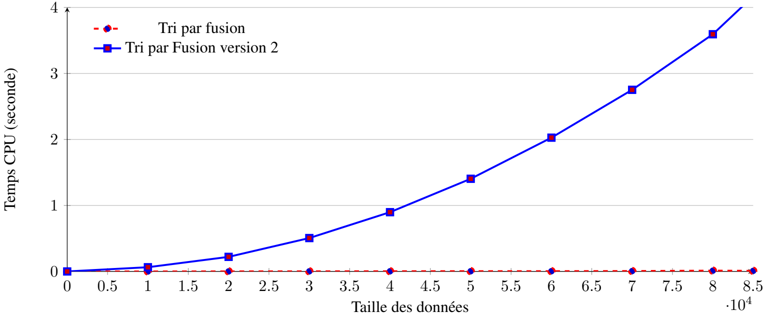
Déduire la Conjecture : $\Theta(n.\log_2(n))$ par meth. substitution.

Nombre d’Opérations I



Mesure de Performance I

performance sur cette machine ...



Mesure de Performance II

Par exemple, pour $n = 85000$:

Fusion	**Nb. Oper.**	hauteur max.	**Temps (sec.)**
Algo class.	1391889	16.4	0.01
version 2	878184	10.3	
	2379451	28	4.1

Rappel Théorème I

Theorem

Théorème proposé par Akra et Bazzi (1998) qui permet de résoudre des récurrences (avec q divisions du problème en des sous-problèmes de tailles très inégales)

$T(n) = \sum_{i=1}^{k} a_i.T(\left\lfloor \frac{n}{b_i} \right\rfloor) + f(n)$, *avec* $k \geq 1$, $a_i > 0$, $\sum a_i \geq 1$ *et* $b_i > 1$

$f(n)$ *est bornée, positive et non décroissante*

$\forall c > 0, \exists n_0, d > 0 / f(\frac{n}{c}) \geq d.f(n), \forall n \geq n_0$

1. *Déterminer la valeur de p (unique et positive) telle que* $\sum_{i=1}^{k}(a_i.b_i^{-p}) = 1$
2. *la solution de la récurrence est alors* $T(n) = \Theta(n^p) + \Theta(n^p.\int_{n'}^{n} \frac{f(x)}{x^{p+1}}dx)$ *avec* n' *suffisamment grand*

Illustration sur la version 2 de Fusion I

$T(n) = \sum_{i=1}^{k} a_i.T(\left\lfloor \frac{n}{b_i} \right\rfloor) + f(n)$, avec $k \geq 1$, $a_i > 0$, $\sum a_i \geq 1$ et $b_i > 1$

Démonstration.

Or : $T(n) = T(\frac{n}{3}) + T(\frac{2.n}{3}) + O(n)$
Donc : $k = 2$, $\forall i, a_i = 1$, $b_1 = 3$, $b_2 = \frac{3}{2}$ et $f(n) = n$ $\square$

$\forall c > 0, \exists n_0, d > 0 / f(\frac{n}{c}) \geq d.f(n), \forall n \geq n_0$

Démonstration.

Le coût total est déterminé par :

$$f(\frac{n}{c}) \geq d.f(n)$$

$\frac{n}{c} \geq d.n$, avec $n > 0$

$$\frac{1}{c} \geq d > 0$$

$\square$

Illustration sur la version 2 de Fusion II

Déterminer la valeur de p (unique et positive) telle que $\sum_{i=1}^{k}(a_i.b_i^{-p}) = 1$

Démonstration.

$$3^{-p} + (\frac{3}{2})^{-p} = 1$$

$$(\frac{1}{3})^{p} + (\frac{2}{3})^{p} = 1$$

$$1^{p} + 2^{p} = 3^{p}$$

Prenons $p = 1$, valeur unique et positive.

$\square$

Illustration sur la version 2 de Fusion III

la solution de la récurrence est alors
$T(n) = \Theta(n^p) + \Theta(n^p. \int_{n'}^{n} \frac{f(x)}{x^{p+1}} dx)$ avec n' suffisamment grand

Démonstration.

$$T(n) = \Theta(n) + \Theta(n. \int_{n'}^{n} \frac{x}{x^2} dx)$$

$$T(n) = \Theta(n) + \Theta(n. \int_{n'}^{n} \frac{1}{x} dx)$$

$$T(n) = \Theta(n) + \Theta(n.(\ln(n) + C)), \text{ avec } C > 0$$

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

Nous pouvons prendre $T(n) = \Theta(n. \log_2(n))$ à une constante multiplicative près : $\frac{\ln(n)}{\log_2(n)} = \frac{\log_2(n)}{\log_2(e)} . \frac{1}{\log_2(n)} = \frac{1}{\log_2(e)} \approx 0.693$ □

Plan

fiche TD 6 exo 1 I

Exercice 2 : 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.

fiche TD 6 exo 1 II

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

Algorithme 2 : Fibo(n)

```
Entrées : n : entier
Sorties : entier
début
    si n ≤ 0 alors
        retourner 0 ;
    sinon
        si n = 1 alors
            retourner 1 ;
        sinon
            retourner Fibo (n-1) + Fbo (n-2) ;
```

---

fiche TD 6 exo 1 III

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).

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

fiche TD 6 exo 1 IV

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 : $F_n = \Phi^n$

Plan

fiche TD 5 exo 5 I

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.

fiche TD 5 exo 5 II

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.

fiche TD 5 exo 5 III

Proposer un algorithme simple pour résoudre le problème 2-somme avec une complexité quadratique

---

Algorithme 3 : 2-Somme(E,S,n)

```
      Entrées : E[1..n] tableau d'entiers, S : entier, n : entier
début
    trouve ← false ;
    i ← 1 ;
    tant que (i < n) ∧ (¬trouve) faire
        j ← (i + 1) ;
        tant que (j ≤ n) ∧ (¬trouve) faire
            si (E[i] + E[j] = S) alors trouve ← true ;
            j ← j + 1 ;
        i ← i + 1 ;
    retourner trouve ;
```

---

fiche TD 5 exo 5 IV

Complexité : $T(n) = O(n^2)$

fiche TD 5 exo 5 V

En généralisant cette méthode, en déduire une première borne supérieure polynomiale sur la complexité du problème k-somme, pour $k \geq 2$.

Complexité : $T(n, k) = O(n^k)$

Proposer un algorithme permettant de résoudre le problème 2-somme sur une liste triée en complexité linéaire.

Algorithme 4 : Tri-2-SOM(E,S,n)

Entrées : $E[1..n]$ tableau d'entiers, S : entier, n : entier

4.1 **début**

4.2   *trouve* $\leftarrow$ *false* ;

4.3 $i \leftarrow 1$; $j \leftarrow n$;

4.4   **tant que** $(i < j)$ **faire**

4.5   **si** $(E[i] + E[j] = S)$ **alors** *trouve* $\leftarrow$ *true* ;

4.6   **sinon**

4.7   **si** $(E[i] + E[j] > S)$ **alors** $j \leftarrow j - 1$;

4.8   **sinon** $i \leftarrow i + 1$;

4.9   **retourner** *trouve* ;

Complexité : $T(n) = O(\frac{n}{2})$

fiche TD 5 exo 5 VII

En déduire un algorithme de complexité quadratique pour 3-somme.

Algorithme 5 : 3-SOM(E,S,n)

Entrées : $E[1..n]$ tableau d'entiers, S : entier, n : entier

```
début
    trouve ← false ;
    Tri_insertion(E,n) ;
    pour i ← 1 à n faire
        si Tri-2-SOM (E[i+1..n], S − E[i], n − i) alors
            trouve ← true ;
    retourner trouve ;
```

Complexité : $T(n) = O(n^2)$, car Tri_insertion en $O(n^2)$ et la boucle Pour en $n.O(n)$.

fiche TD 5 exo 5 VIII

Cet algorithme est le meilleur connu actuellement. Le problème de savoir si n^2 est l'ordre de grandeur d'une borne inférieure à la complexité de *3-SOM* reste un problème ouvert.

fiche TD 5 exo 5 IX

Proposer un algorithme de complexité $O(n.\log_2(n))$ pour 2-somme avec une liste non triée.

Algorithme 6 : 2-SOM(E,S,n)

Entrées : $E[1..n]$ tableau d'entiers, S : entier, n : entier

```
début
    trouve ← false ;
    Tri_Fusion(E,n) ;
    pour i ← 1 à n faire
        si Dicho (E, n, S − E[i]) alors trouve ← true ;
    retourner trouve ;
```

Complexité : $O(n.\log_2(n))$, car Tri_Fusion en $O(n.\log_2(n))$ et la boucle Pour en $n.O(\log_2(n))$.
L'algorithme est optimal pour *2-SOM*.

Plan

fiche TD 5 exo 3 I

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.

fiche TD 5 exo 3 II

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.

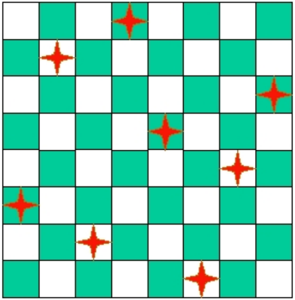
1. Pour un échiquier de taille $n \times n$, le nombre max. de reines, noté $R(n)$, est contraint par $R(n) \leqslant n$.
2. En Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).
3. Décrire l'algorithme (algorithme de *backtracking*) :
   - 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.

fiche TD 5 exo 3 III

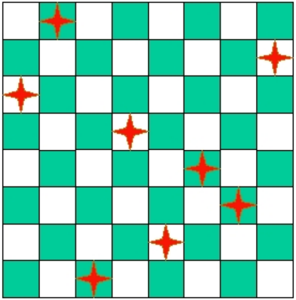
- L'algorithme *Placer* est chargé de positionner les reines les unes après les autres en parcourant l'arbre de décision. L'algorithme doit donner toutes les solutions (i.e., poursuit sa recherche).

4. En déterminer la complexité de l'algorithme.

fiche TD 5 exo 3 IV



Une solution ($N = 8$)



Pas une Solution

fiche TD 5 exo 3 V

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

Definition

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. □

fiche TD 5 exo 3 VI

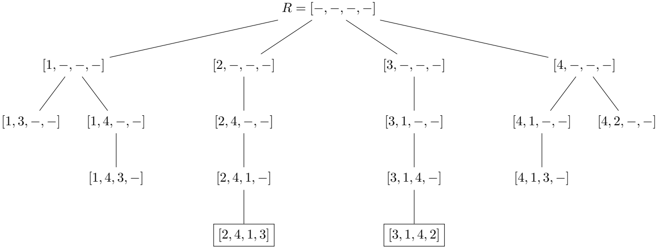
En Proposer une stratégie et déterminer les solutions pour $n = 4$ (optionnel de même pour les cas simples : $n = 2$ ou $n = 3$).

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).

fiche TD 5 exo 3 VII



fiche TD 5 exo 3 VIII

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

```
début
    ok ← true ;
    i ← 1 ;
    tant que i < lig faire
        c ← R[i] ;
        ok ← (ok) ∧ (|lig − i| ≠ |col − c|) ∧ (col ≠ c) ;
        i ← i + 1 ;
    retourner ok ;
```

---

fiche TD 5 exo 3 IX

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)

fiche TD 5 exo 3 X

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

Entrées : $R[1..n]$ tableau d'entiers, n : entier, $ligne$: entier

```
début
    si ligne > n alors
        Afficher (R,n) ;
    sinon
        colonne ← 1 ;
        tant que colonne ≤ n faire
            si Libre(R, ligne, colonne) alors
                R [ligne] ← colonne ;
                Placer (R,n,ligne + 1) ;
                R [ligne] ← 0 ;
            colonne ← colonne + 1 ;
```

fiche TD 5 exo 3 XI

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

fiche TD 5 exo 3 XII

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

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

$\square$

fiche TD 5 exo 3 XIII

Pour information, le nombre de solutions est le suivant :

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

fiche TD 5 exo 3 XIV

Comparaison entre différents algorithmes de recherche de toutes les solutions :

- **Notre Algo. *N*-Reines** (couleur rouge) : Notre algorithme proposé dans cet exercice
- **Algo trivial** (couleur bleue) : 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 5 exo 3 XV

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	-

Plan

Problème SAT I

Definition

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

Problème de décision SAT I

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 » ?

Theorem

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

Problème de décision **SAT** II

Exemple 2 : $p \wedge \neg q$ est satisfiable ssi p VRAI et q FAUX

Exemple 3 : $p \wedge \neg p$ 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)
- ...

Revenons sur l'exercice 48 page 81 I

Exercice 5 : Logiques Booléennes

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 ? I

Exemple 4 : $p_1 \wedge p_2 \wedge \ldots \wedge p_n$

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 ? I

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

Question 1 : 2-SAT ? II

Instance : Une formule 2-SAT Φ
Question : Φ est-elle satisfiable ?

Exemple 5 : 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 $\neg p_1 \vee p_4$

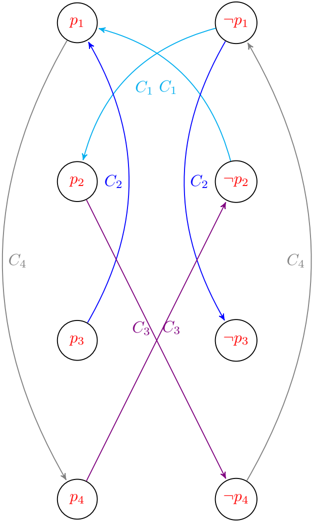
Question 1 : 2-SAT ? III

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 $a \rightarrow b$ 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	$p_1 \vee p_2$	$\neg p_1 \rightarrow p_2$	$\neg p_2 \rightarrow p_1$
C_2	$p_1 \vee \neg p_3$	$\neg p_1 \rightarrow \neg p_3$	$p_3 \rightarrow p_1$
C_3	$\neg p_2 \vee \neg p_4$	$p_2 \rightarrow \neg p_4$	$p_4 \rightarrow \neg p_2$
C_4	$\neg p_1 \vee p_4$	$p_1 \rightarrow p_4$	$\neg p_4 \rightarrow \neg p_1$

Question 1 : 2-SAT ? IV



Question 1 : 2-SAT ? V

Pour n variables et m clauses :

1. Créer un graphe avec $2n$ noeuds
2. Relier les noeuds avec une arête $a \rightarrow b$
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 : 3-SAT ? I

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**.

Question 2 : 3-SAT ? II

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

<u>$k = 1$</u> : 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.

<u>$k = 2$</u> : 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.

<u>$k = 3$</u> : Chaque clause est deja dans 3-SAT (les deux problèmes sont identiques).

Question 2 : 3-SAT ? III

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

Posons

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

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**.

Question 2 : 3-SAT ? IV

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* ? I

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 I

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