LICENCE D'INFORMATIQUE

Sorbonne Université

LU3IN003 – Algorithmique

Cours 3 : Programmation récursive II

Algorithmes d'exploration d'un arbre d'énumération

Année 2024-2025

Responsables et chargés de cours

Fanny Pascual

Olivier Spanjaard

Le problème de la somme d'un sous-ensemble

- Etant donné un ensemble X d'entiers naturels et un ***entier but*** B, existe-t-il un sous-ensemble S d'éléments de X qui somment à B ?

  Exemples
  $X = \{8, 6, 7, 5, 3, 10, 9\}$ et $B = 15 \rightarrow$ réponse *vrai*, pour $S = \{7, 5, 3\}$
  $X = \{11, 6, 5, 1, 7, 13, 12\}$ et $B = 15 \rightarrow$ réponse *faux*

- **Cas de base :**
  – si $B = 0$ alors la réponse est *vrai*, en prenant $S = \varnothing$,
  – si $B < 0$ alors la réponse est *faux*,
  – si $B > 0$ et $X = \varnothing$ alors alors la réponse est *faux*.

- **Récurrence :**
  Soit $x \in X$. Il existe un sous-ensemble d'éléments qui somment à B si et seulement si au moins une des deux propositions suivantes est vraie :
  – il existe un sous-ensemble d'éléments de $X \setminus \{x\}$ qui somment à B,
  – il existe un sous-ensemble d'éléments de $X \setminus \{x\}$ qui somment à $B - x$.

Plus formellement

L'ensemble *X* est représenté en machine sous forme d'un tableau X[1..n], où n le nombre d'éléments de *X* et X[i] est la valeur du i-ème entier de *X*.

X = {8, 6, 7, 5, 3, 10, 9} ⇨ X = [8,6,7,5,3,10,9]

L'algorithme de retour arrière mettant en œuvre la récurrence précédente s'écrit alors :

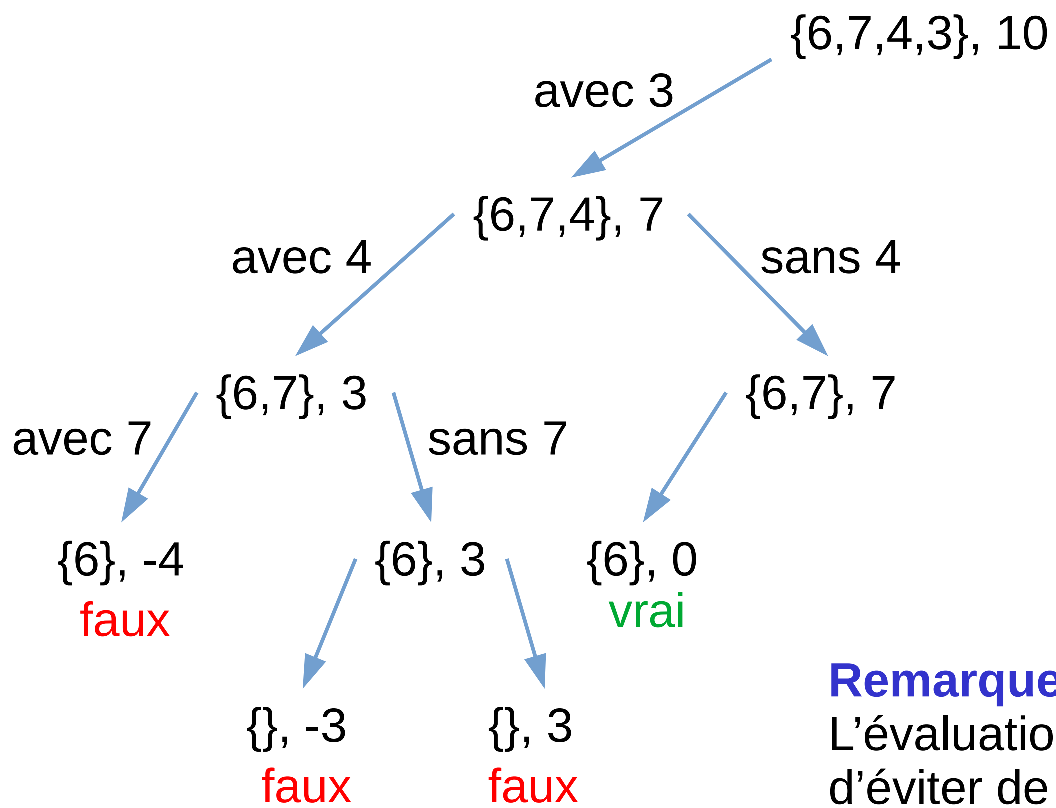
```
fonction Somme(X,i,b)
    si b=0
            retourner vrai
    sinon si b<0 ou i=0
            retourner faux
    sinon
            retourner Somme(X,i-1,b-X[i]) ou Somme(X,i-1,b)
```

Appel initial
Somme(X, n, B)

L'appel **`Somme(X,i,b)`** retourne vrai s'il existe un sous-ensemble d'éléments de X[1..i] dont la somme est b.

Arbre des appels récursifs I

L'arbre des appels récursifs obtenu pour $X = \{6, 7, 4, 3\}$ et $B = 10$ est représenté ci-dessous :

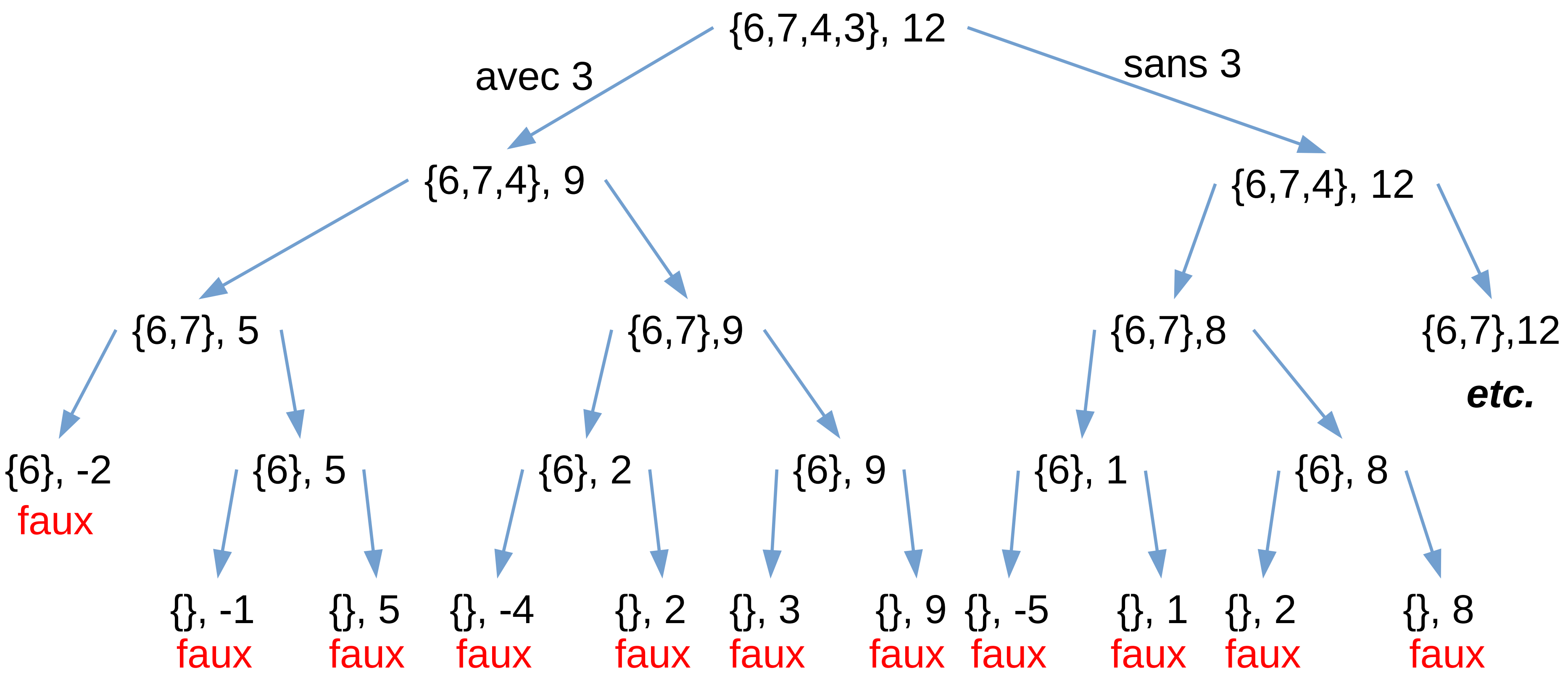


Remarque
L'évaluation paresseuse du **ou** logique permet d'éviter de nombreux appels récursifs inutiles.

`Somme([6,7,4,3],4,10)` retourne vrai.

Arbre des appels récursifs II

L'arbre des appels récursifs obtenu pour $X = \{6, 7, 4, 3\}$ et $B = 12$ est représenté ci-dessous :



`Somme([6,7,4,3],4,12)` retourne faux.

Correction de l'algorithme

```
fonction Somme(X,i,b)
si b=0
    retourner vrai
sinon si b<0 ou i=0
    retourner faux
sinon
    retourner Somme(X,i-1,b-X[i]) ou Somme(X,i-1,b)
```

On prouve par récurrence sur (i,b) la correction (terminaison + validité) de l'algorithme.

$HR_{i,b}$ « **`Somme(X,i,b)`** se termine et retourne vrai ssi il existe un sous-ensemble d'éléments de X[1..i] qui somme à b. »

Cas de base : i=0 ou b$\leq$0. Les appels de la forme **`Somme(X,0,b)`** ou **`Somme(X,i,b)`** pour b$\leq$0 se terminent et retournent la bonne valeur de vérité (évident).

Etape inductive : montrons que $HR_{i-1,b'}$ est vérifiée pour b'$\leq$b $\Rightarrow$ $HR_{i,b}$ est vérifiée.
Trois cas :

- si $\exists$ un sous-ensemble de X[1..i] qui contient X[i] et somme à b, **`Somme(X,i-1,b-X[i])`** se termine et retourne *vrai* d'après $HR_{i-1,b-X[i]}$ d'où **`Somme(X,i,b)`** se termine et retourne *vrai*.

- sinon, si $\exists$ un sous-ensemble de X[1..i] qui ne contient pas X[i] et somme à b, alors **`Somme(X,i-1,b-X[i])`** se termine et retourne *faux* d'après $HR_{i-1,b-X[i]}$ et **`Somme(X,i-1,b)`** se termine et retourne *vrai* d'après $HR_{i-1,b}$. D'où **`Somme(X,i,b)`** se termine et retourne *vrai*.

- sinon **`Somme(X,i-1,b-X[i])`** et **`Somme(X,i-1,b)`** se terminent et retournent *faux* d'après $HR_{i-1,b-X[i]}$ et $HR_{i-1,b}$ d'où **`Somme(X,i,b)`** se termine et retourne bien *faux*.

Analyse de complexité

```
fonction Somme(X,i,b)
si b=0
    retourner vrai
sinon si b<0 ou i=0
    retourner faux
sinon
    retourner Somme(X,i-1,b-X[i]) ou Somme(X,i-1,b)
```

On suppose que les différentes opérations élémentaires (**`ou`** logique, comparaison, soustraction) se font en O(1). Chaque appel **`Somme(X,i,b)`** est donc en temps constant.

Comptons le nombre A(i) de nœuds dans l'arbre obtenu pour l'appel **`Somme(X,i,b)`**. On a :

$A(0) = 1$
$A(i) \leq 2*A(i-1)+1$ (le $\leq$ est lié au fait qu'un appel déclenche *au plus* deux appels récursifs)

Soit A'(i) définie par :

$A'(0) = 1$
$A'(i) = 2*A'(i-1)+1$

On montre facilement que le terme général est $A'(i)=2^{i+1}-1$. Comme $A(i) \leq A'(i)$, on en déduit que $A(i) \in O(2^i)$.

La complexité de **`Somme(X,n,B)`** est donc $O(2^n)$.

Remarque importante

Supposons que l'on mémorise dans une table `M[i,b]` le résultat des appels à `Somme` pour les différents couples de paramètres `(i,b)`, et que l'on retourne directement le résultat lorsque l'on réalise un appel à `Somme(X,i,b)` pour lequel `M[i,b]` a déjà été calculé.

On parle de ***mémoïsation***.

```
fonction Somme(X,i,b)
si M[i,b] est connu
    retourner M[i,b]
sinon si b=0
    retourner vrai
sinon si b<0 ou i=0
    retourner faux
sinon
    M[i,b]=Somme(X,i-1,b-X[i]) ou Somme(X,i-1,b)
    retourner M[i,b]
```

Il y a alors de l'ordre de *nB* appels récursifs (autant que de couples de paramètres `(i,b)` possibles), et la complexité devient O(nB).

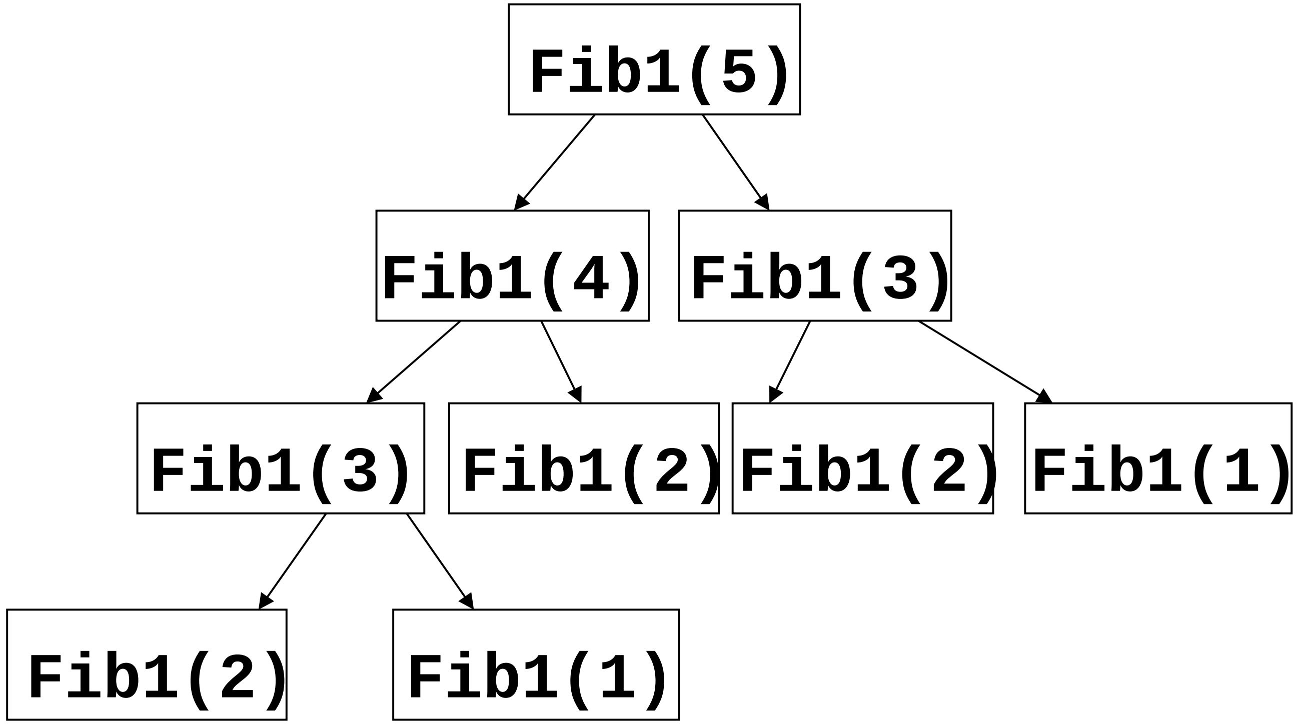
Cette complexité est **pseudopolynomiale** car le paramètre B est encodé sur $\log_2 B$ bits.

Il s'agit d'un algorithme de programmation dynamique (vu plus tard dans le semestre).

Retour sur **`Fib1`**

Lors de la première séance, on a vu la fonction **`Fib1`**(n) pour déterminer F_n (n-ième terme de la suite de Fibonacci) :

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```



Nombre de nœuds dans l'arbre

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```

Revenons sur son analyse de complexité. Soit A(n) le nombre de nœuds dans l'arbre des appels récursifs :

A(1) = 1
A(2) = 1
A(n) = A(n-1) + A(n-2) + 1

On remarque que A(n) = A'(n) – 1, où :

A'(1) = 2
A'(2) = 2
A'(n) = A'(n-1) + A'(n-2)

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**
A(n)	1	1	3	5	9	15	25
A'(n)	2	2	4	6	10	16	26

Suite récurrente linéaire d'ordre 2

A'(1) = 2
A'(2) = 2
A'(n) = A'(n-1) + A'(n-2)

Théorème
Soit a_1 et a_2 deux réels, et une suite u_n vérifiant :

$u_n = a_1u_{n-1} + a_2u_{n-2}$. **(suite récurrente linéaire d'ordre 2)**

Le polynôme caractéristique associée est : $r^2 - a_1r - a_2$.
Posons $\Delta = a_1^2+4a_2$. On a :

- Si $\Delta>0$, i.e., le polynôme caractéristique admet deux racines réelles distinctes r_1 et r_2, alors il existe deux réels C_1 et C_2 tels que $u_n = C_1r_1^n + C_2r_2^n$.
- Si $\Delta=0$, i.e., le polynôme caractéristique admet une unique racine réelle r, alors il existe deux réels λ_0 et λ_1 tels que $u_n = (\lambda_0 + \lambda_1n)r^n$.

Remarque : Dans notre contexte algorithmique où la suite A'(n) est définie pour compter le nombre d'appels à une fonction, on a $a_1\geq0$ et $a_2\geq0$, avec au moins une inégalité stricte, et par conséquent $\Delta>0$.

Esquisse de preuve dans le cas $\Delta>0$

- Cherchons un terme général de la forme $u_n = Cr^n$, où C une constante
- $u_n = a_1 u_{n-1} + a_2 u_{n-2} \iff Cr^n = a_1 Cr^{n-1} + a_2 Cr^{n-2} \iff$ $\mathbf{r^2 - a_1 r - a_2 = 0}$
  (en divisant par Cr^{n-2})
- Pour $\Delta>0$, on a **deux racines r_1 et r_2** à l'équation du second degré
- $C_1 r_1^n = a_1 C_1 r_1^{n-1} + a_2 C_1 r_1^{n-2}$ et $C_2 r_2^n = a_1 C_2 r_2^{n-1} + a_2 C_2 r_2^{n-2}$
  $\Rightarrow C_1 r_1^n + C_2 r_2^n = a_1(C_1 r_1^{n-1} + C_2 r_2^{n-1}) + a_2(C_1 r_1^{n-2} + C_2 r_2^{n-2})$
- Autrement dit **$u_n = C_1 r_1^n + C_2 r_2^n$ vérifie l'équation de récurrence**
- **Connaissant** les valeurs **u_1 et u_2**, on peut résoudre un **système de deux équations à deux inconnues (C_1 et C_2)** pour déterminer C_1 et C_2 vérifiant :

  $$C_1 r_1 + C_2 r_2 = u_1$$
  $$C_1 r_1^2 + C_2 r_2^2 = u_2$$

Quiz : Suite récurrente d'ordre 2

On considère la suite u_n définie par :

$u_0 = 1$

$u_1 = 1$

$u_n = 3u_{n-1}+4u_{n-2}$

Quelles sont les racines du polynôme caractéristique ?

A) $r_1=3$ et $r_2=-2$

B) $r_1=1$ et $r_2=5$

C) $r_1=4$ et $r_2=-1$

D) $r_1=2$ et $r_2=6$

Quiz : Suite récurrente d’ordre 2 (suite)

On considère la suite u_n définie par :

$u_0 = 1$

$u_1 = 1$

$u_n = 3u_{n-1}+4u_{n-2}$

Dans l’expression $u_n=C_1r_1^n+C_2r_2^n$, que valent C_1 et C_2 ?

A) $C_1=1/2$ et $C_2=3/2$
B) $C_1=1/3$ et $C_2=2/3$
C) $C_1=1/4$ et $C_2=3/4$
D) $C_1=2/5$ et $C_2=3/5$

Complexité de **`Fib1`**

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```

$A'(1) = 2$, $A'(2) = 2$, $A'(n) = A'(n-1) + A'(n-2)$

Le **polynôme caractéristique** est $r^2 - r - 1$, avec $\Delta=1+4=5$, de racines $r_1=(1+\sqrt{5})/2\approx 1.618(\simeq 2^{0,694})$ et $r_2=(1-\sqrt{5})/2\approx -0.618$.

Le terme général est de la forme $A'(n)\approx C_1 1.618^n+C_2(-0.618)^n$.

Les termes $A'(1)$ et $A'(2)$ permettraient de déduire C_1 (>0) et C_2 mais cela n'importe pas pour la complexité : **le nombre de nœuds dans l'arbre est en $O(1.618^n)$**.

Chaque appel comporte une addition **en $O(n)$** (voir cours 1).

La complexité de Fib1 est donc $O(n\ 1.618^n)$.

Formule logique 3FNC

Soit $x_1, x_2, \ldots, x_n$ un ensemble de variables booléennes.

- Un littéral est soit une variable x_i, soit sa négation $\neg x_i$.
- Une clause est une disjonction (ou logique, noté $\vee$) de littéraux.
  - Par exemple $x_1 \vee x_2 \vee \neg x_4$ est une clause.
- Une formule en Forme Normale Conjonctive (FNC) est une conjonction (et logique, noté $\wedge$) de clauses.
  - Par exemple $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3) \wedge x_5$ est une formule FNC.
- Une formule Φ est 3FNC si chaque clause dans Φ comporte 3 littéraux.
  - Par exemple $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_1)$ est une 3FNC,
    mais $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3) \wedge x_5$ n'en est pas une.

Problème 3-SAT

Problème SAT
Donnée : Une formule FNC Φ.
Question : Existe-t-il une affectation de valeurs de vérité aux variables de Φ en sorte que Φ soit vraie ?

Problème 3-SAT
Donnée : Une formule 3FNC Φ.
Question : Existe-t-il une affectation de valeurs de vérité aux variables de Φ en sorte que Φ soit vraie ?

Exemple :

$(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_1)$ est satisfiable ; par ex. pour $x_1, \ldots, x_4$ toutes vraies.

$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ n'est pas satisfiable.

Idée d'algorithme

Exemple $\Phi = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$

Dans une affectation de valeurs de vérité rendant Φ vraie, on a :

- soit $\neg x_1 = \text{vrai}$,
- soit $\neg x_1 = \text{faux}, \neg x_2 = \text{vrai}$,
- soit $\neg x_1 = \text{faux}, \neg x_2 = \text{faux}, \neg x_3 = \text{vrai}$.

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \xrightarrow{x_1=\text{faux}} (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

$$(x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \xrightarrow{x_2=\text{vrai}} (x_3) \wedge (\neg x_3) \xrightarrow{x_3=\text{vrai}} \text{faux}$$

$$(x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \xrightarrow{x_2=\text{faux},\ x_3=\text{faux}} \text{vrai}$$

La formule Φ est **satisfiable**.

Un algorithme pour 3-SAT

```
fonction Test(Φ)
    si Φ est vide
          retourner vrai
    sinon si Φ comporte une clause faux
          retourner faux
    sinon
          (l1∨l2∨l3)∧Φ' =Φ    /* si k<3 littéraux, alors k appels récursifs */
          si Test(Φ' | l1=vrai) retourner vrai
          si Test(Φ' | l1=faux,l2=vrai) retourner vrai
          si Test(Φ' | l1=faux,l2=faux,l3=vrai) retourner vrai
          retourner faux
```

où Φ **$|x_i$=vrai** (resp. Φ **$|x_i$=faux**) est la simplification de Φ obtenue en affectant la valeur de vérité vrai (resp. faux) à x_i.

Exemple

Φ $(x_2 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$

$\Phi |x_2$=vrai $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_3 \vee x_4)$

$\Phi |x_2$=vrai,x_3=faux,x_4=faux faux

Complexité de **`Test`** (1/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ’=Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ’ | l1=vrai) retourner vrai
            si Test(Φ’ | l1=faux,l2=vrai) retourner vrai
            si Test(Φ’ | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

Soit A(n) le nombre d’appels réalisés par Test(Φ) si Φ comporte *n* variables non instanciées. On a :

A(0)=1 (pas d’appel récursif car toutes les variables sont instanciées)
A(1)=2 (un unique appel récursif pour instancier la variable restante)
$A(2)\leq A(1)+A(0)+1=4$
$A(n)\leq A(n-1)+A(n-2)+A(n-3)+1$ (l’appel courant et les trois appels récursifs avec une, deux ou trois variables supplémentaires instanciées)

Pour obtenir une complexité en O(.), on va chercher à déterminer le terme général A(n) si l’on avait des = à la place des $\leq$.

Complexité de `Test` (2/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ' =Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ' | l1=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

$A(0)=1$
$A(1)=2$
$A(2)=4$
$A(n)=A(n-1)+A(n-2)+A(n-3)+1$

Afin de se débarrasser de la constante 1, on considère la suite $A'(n)$ de la forme :

$A'(0)=a$
$A'(1)=b$
$A'(2)=c$
$A'(n)=A'(n-1)+A'(n-2)+A'(n-3)$

Comment choisir a,b,c de manière à avoir $A(n)\leq A'(n)$ pour tout $n\geq 0$?

Complexité de **Test** (3/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ' =Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ' | l1=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

On va faire en sorte que $A'(n)=A(n)+x$ (avec $x\geq 0$) pour tout $n\geq 0$.

On aurait donc :

$A'(n)=A'(n-1)+A'(n-2)+A'(n-3)$

$=A(n-1)+x+A(n-2)+x+A(n-3)+x$

$=A(n-1)+A(n-2)+A(n-3)+1+3x-1$

$=A(n)+3x-1$

Par conséquent, $x=3x-1$, ce qui donne $x=1/2$. D'où :

$A'(0)=A(0)+1/2=1,5$

$A'(1)=A(1)+1/2=2,5$

$A'(2)=A(2)+1/2=4,5$

$A'(n)=A'(n-1)+A'(n-2)+A'(n-3)$

n	0	1	2	3	4	5	6
A(n)	1	2	4	8	15	28	52
A'(n)	1,5	2,5	4,5	8,5	15,5	28,5	52,5

Complexité de `Test` (4/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ' =Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ' | l1=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

Cherchons maintenant à majorer A'(n) par une suite récurrente A''(n) vérifiant A''(n)=A''(n-1)+A''(n-2)+A''(n-3)
et de terme général de la forme $A''(n)=Cr^n$. On a alors : $Cr^n=Cr^{n-1}+Cr^{n-2}+Cr^{n-3}$,
ce qui se simplifie pour donner le **polynôme caractéristique** : $r^3=r^2+r+1$.

Cette équation a une unique racine dans $\mathbb{R}$: r=1.84 (trouvée à l'aide d'un solveur).
Considérons les premiers termes de la suite 1.84^n :

n	0	1	2	3	4	5	6
A(n)	1	2	4	8	15	28	52
A'(n)	1,5	2,5	4,5	8,5	15,5	28,5	52,5
1.84^n	1	1,84	3,39	6,23	11,46	21,09	39,81

Afin d'obtenir $A''(0)\geq A'(0)$, $A''(1)\geq A'(1)$ et $A''(2)\geq A'(2)$, on peut prendre par exemple C=1.5.

Complexité de **`Test`** (5/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ' =Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ' | l1=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=vrai) retourner vrai
            si Test(Φ' | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

Afin d'obtenir une majoration, considérons donc la suite définie par $A''(n)=1.5 \times 1.84^n$:

n	0	1	2	3	4	5	6
A(n)	1	2	4	8	15	28	52
A'(n)	1,5	2,5	4,5	8,5	15,5	28,5	52,5
A''(n)	1,5	2,76	5,09	9,35	17,19	31,64	59,72

On voit que l'on a bien $A'(0) \leq A''(0)$, $A'(1) \leq A''(1)$ et $A'(2) \leq A''(2)$.

Or, si $A'(n-1) \leq A''(n-1)$, $A'(n-2) \leq A''(n-2)$ et $A'(n-3) \leq A''(n-3)$:
$A'(n)=A'(n-1)+A'(n-2)+A'(n-3) \leq A''(n-1)+A''(n-2)+A''(n-3)=A''(n)$
On en déduit par récurrence que $A'(n) \leq A''(n)=1.5 \times 1.84^n$ et donc que $A(n) \in O(1.84^n)$.

Complexité de **`Test`** (6/6)

```
fonction Test(Φ)
    si Φ est vide
            retourner vrai
    sinon si Φ comporte une clause faux
            retourner faux
    sinon
            (l1∨l2∨l3)∧Φ’=Φ   /* si k<3 littéraux, alors k appels récursifs */
            si Test(Φ’ | l1=vrai) retourner vrai
            si Test(Φ’ | l1=faux,l2=vrai) retourner vrai
            si Test(Φ’ | l1=faux,l2=faux,l3=vrai) retourner vrai
            retourner faux
```

On vient de montrer qu’il y a $O(1.84^n)$ appels récursifs.

Chaque appel récursif est en $O(m)$ pour faire la simplification de la formule Φ’ en parcourant la formule.

On en déduit que la complexité de l’algorithme **`Test`** est $O(1.84^n m)$.

Remarque importante

Explorer toutes les affections possibles de valeurs de vérité serait en $O(2^n m)$.

La fonction 1.84^n croît significativement moins vite que 2^n.

Plus généralement

Considérons une **fonction récursive F(n)** dont le **nombre A(n) de noeuds** dans l'arbre des appels récursifs s'écrit sous la forme

$A(n)=a_1A(n-1)+...+a_kA(n-k)+1$ avec $a_1\geq 0,\ldots,a_k\geq 0$.

Comme montré pour 3-SAT2, on peut considérer à la place une suite récurrente de la forme

$A'(n)=a_1A'(n-1)+...+a_kA'(n-k)$ (R)

telle que $A(n)\leq A'(n)$ pour tout $n\geq 0$, dont le polynôme caractéristique est :

$P(r)=r^k-a_1r^{k-1}-...-a_{k-1}r-a_k$

Comme $a_1\geq 0,\ldots,a_k\geq 0$, la *règle des signes de Descartes* implique que ce polynôme ne comporte qu'une seule racine réelle positive (les autres racines sont négatives ou dans $\mathbb{C}$), que l'on notera r_0. On considère alors une suite (qui vérifie la relation de récurrence R)

$A''(n)=Cr_0^n$

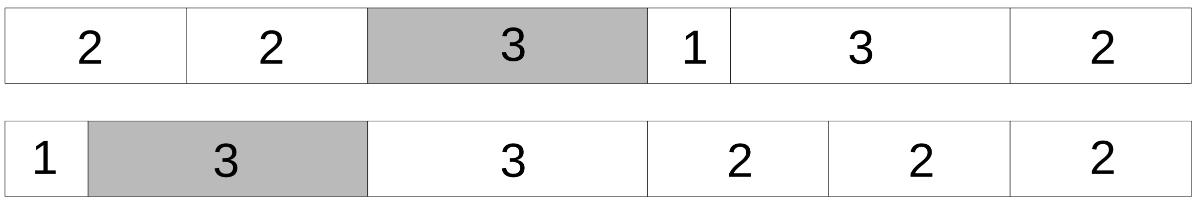
où C une constante suffisamment grande pour que $A'(n)\leq A''(n)$ pour tout $n\geq 0$.

Comme $A(n)\leq A'(n)\leq A''(n)$ pour tout $n\geq 0$, on en déduit que le **nombre A(n) de nœuds** de l'arbre des appels récursifs est en **$O(r_0^n)$**.

Si la complexité de **chaque appel est en $O(n^d)$** alors **la complexité de F(n) est $O(r_0^n n^d)$**.

Quiz : Au pied du mur

On dispose de briques de longueur 1, 2 et 3 pour bâtir un mur. Les briques de longueur 1 et 2 sont toutes blanches, tandis que les briques de longueur 3 existent en blanc et en gris. Par exemple, voici deux façons de construire une rangée de longueur 13 :



Quelle est la complexité de la fonction **`mur(n)`** (non-optimisée !), qui dénombre le nombre de façons de construire une rangée de longueur n ?

```
fonction mur(n)
si n = 1 retourner 1
si n = 2 retourner 2
si n = 3 retourner 5
retourner mur(n-1) + mur(n-2) + mur(n-3) + mur(n-3)
```

A) $O(2^n)$
B) $O(n2^n)$
C) $O(3^n)$
D) $O(n3^n)$

Une remarque pour conclure

Afin d'obtenir des algorithmes de retour arrière ***de complexité exponentielle mais néanmoins efficaces en pratique*** pour trouver **une** solution, il est nécessaire de recourir à des calculs supplémentaires (pas trop coûteux en temps…) lors de chaque appel afin de :

- déclencher des retours arrières le plus haut possible dans l'arbre,
- et ainsi **explorer une portion réduite de l'arbre** des appels récursifs.

On obtient alors ce que l'on appelle :

- un **branch and bound** (pour la recherche d'une solution optimale à un problème d'optimisation),
- ou un algorithme de **résolution de problèmes de satisfaction de contraintes**.

Ces méthodes algorithmiques seront vues en **master informatique** selon votre choix d'orientation.