Problème d'énumération

Jean-Marc Vincent[1]

[1] Laboratoire LIG
Équipe-Projet POLARIS
Jean-Marc.Vincent@univ-grenoble-alpes.fr

2024

PROBLÈME, VOUS AVEZ DIT PROBLÈME

Un **problème** en informatique est une question ou un ensemble de questions qu'un ordinateur peut résoudre (traitement d'information).

Exemples de types de problèmes

- problème de **décision** : il existe un chemin de A à B dans un graphe
- problème de **recherche** : étant donné n trouver un diviseur de n
- problème de **comptage** : combien de chemins élémentaires y a-t'il de A à B dans un graphe
- problème d'**optimisation** : trouver le plus court chemin de A à B dans un graphe
- ...

Spécification d'un problème

Description précise des entrées (types, propriétés,...) et des sorties attendues avec leur propriétés.

QUELQUES PROBLÈMES

Algorithme de recherche

- Trouver toutes les manières de placer n reines sur un échiquier sans qu'elles soient en prise.
- Dans un sac à dos charger un ensemble des objets ayant une valeur totale maximale (contrainte de volume)
- Trouver les configurations du jeu Tic-Tac-Toe
- Trouver les k-coloriages d'un graphe
- etc

QUELQUES PROBLÈMES

Algorithme de recherche

- Trouver toutes les manières de placer n reines sur un échiquier sans qu'elles soient en prise.
- Dans un sac à dos charger un ensemble des objets ayant une valeur totale maximale (contrainte de volume)
- Trouver les configurations du jeu Tic-Tac-Toe
- Trouver les k-coloriages d'un graphe
- etc

Le problème SAT

$$\Phi(x_1, x_2, , \cdots, x_n) = \bigwedge_{i=1}^{m} C_i$$

$$C_i = x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_{k_i}},$$

- n variables
- m clauses, k_i taille de la clause C_i avec avec x_{i_l} l'une des variables x_j ou sa négation.
- Forme normale conjonctive

Problème de la classe $\mathcal{NP}$

UN PROBLÈME SIMPLE : LE CRÊPIER PSYCHORIGIDE

Le problème

- Entrée : une pile de crêpes (de tailles différentes)
- Sortie : une pile de crêpes (les mêmes) mais rangées joliment par taille croissante
- Opération (machine) : spatule qui permet de retourner sur la pile un bloc de crêpes

  Question : écrire un algorithme qui résolve ce problème et évaluer le nombre de retournements de blocs

Concepts de base

- expression d'un algorithme
- exécution et observation du déroulement de l'algorithme
- recherche empirique de meilleur/pire cas

UN PROBLÈME SIMPLE : LE CRÊPIER PSYCHORIGIDE

Le problème

- Entrée : une pile de crêpes (de tailles différentes)
- Sortie : une pile de crêpes (les mêmes) mais rangées joliment par taille croissante
- Opération (machine) : spatule qui permet de retourner sur la pile un bloc de crêpes

Question : écrire un algorithme qui résolve ce problème et évaluer le nombre de retournements de blocs

Concepts de base

- expression d'un algorithme
- exécution et observation du déroulement de l'algorithme
- recherche empirique de meilleur/pire cas

Concepts avancés

- récursion simple
- preuve de l'algorithme
- et coût de l'algorithme

Irem Grenoble

Page de Marie Duflot

UN PROBLÈME DIFFICILE : PILZEGAL

Les pièces

Objectif

Réaliser 2 piles de hauteurs égales

- en utilisant chaque pièce une seule fois

Le problème

- Écrire un algorithme qui prend en argument un ensemble de pièces arbitraire et, par une série d'essais arrive à deux piles égales

Les configurations sont données dans le dépliant

Notations

- N nombre de pièces
- $h_1, h_2, \cdots, h_N$ la hauteur des pièces

— Composition du jeu —

Ce jeu est composé de 28 pièces avec des valeurs allant de 4 à 30. Le numéro d'une pièce correspond à sa longueur (échelle 1/2). Les pièces 18, 26 et 27, utilisées dans aucune des instances proposées, ne sont pas fournies.

— But du jeu —

L'objectif de ce jeu est de faire deux piles de même taille en utilisant toutes les pièces (et le plus rapidement possible).

— Déroulement —

Préparation : choisir un niveau de difficulté et une partie sur la page suivante, puis rassembler toutes les pièces correspondantes.

Jeu : mettre bout-à-bout les pièces de manière à obtenir deux piles exactement de la même taille en utilisant toutes les pièces.

— Débutants —

1. 4 5 9
2. 4 6 7 9
3. 4 5 5 6

— Amateurs —

4. 4 5 6 8 11
5. 4 4 5 9 10 14
6. 4 5 7 8 8
7. 7 8 10 11 13 15
8. 12 14 20 21 29 30

— Intermédiaires —

9. 4 5 7 9 10 11 12
10. 5 7 10 10 11 14 16 25
11. 4 6 10 11 13 15 19 23 24 25 30
12. 6 10 14 17 20 24 29

— Professionnels —

13. 4 5 5 7 11 15 17 24 28
14. 5 7 8 10 12 15 17 20 22 30

— Exemple —

Nous allons jouer avec les pièces :

4 5 6 7 8

Les deux piles ne sont pas de même taille : ça ne va pas.

La pièce 6 n'a pas été utilisée : ça ne va pas.

C'est gagné !

Remarque : il peut y avoir plusieurs solutions !

— Variantes —

Vous avez réussi toutes les parties proposées ? Essayez les vôtres !

Choisissez les pièces qui vous voulez : le jeu est maintenant de déterminer s'il est possible de faire deux piles égales. Ou trois piles...

ANALYSE SUR DES EXEMPLES

Tâtonnement

- déplacer les pièces
- trouver un principe/stratégie
- tester sur différents exemples

C'est faisable (conviction) ?

- Bifurcation : certaines instances ne seraient pas faisables
- Remarque : différence entre faisabilité et construction
- Réponse attendue : faisable oui/non et si oui une solution

Expression d'un algorithme

- faut-il donner un ordre aux pièces ?
- faut-il donner un ordre aux piles ?
- comment être certain d'avoir tout regardé ?

Mise en oeuvre : Jeu de rôle

- difficulté de ne pas regarder l'instance
- beaucoup de conditions sur les valeurs...
- pas de preuve

C'est difficile à exprimer, à écrire
il faut une méthode systématique

ALGORITHME D'ÉNUMÉRATION

Regarder toutes les possibilités

Combien de possibilités sont à regarder ?
un peu, beaucoup, "très beaucoup" ?

ALGORITHME D'ÉNUMÉRATION

Regarder toutes les possibilités

Combien de possibilités sont à regarder ?
un peu, beaucoup, "très beaucoup" ?
Raisonnement :
Chaque pièce à le choix entre les 2 piles, comme il y a N pièces il y a 2^N possibilités
ce qui est très important

Démonstration
Cela semble marcher

ALGORITHME D'ÉNUMÉRATION

Regarder toutes les possibilités

Combien de possibilités sont à regarder ?
un peu, beaucoup, "très beaucoup" ?
Raisonnement :
Chaque pièce à le choix entre les 2 piles, comme il y a N pièces il y a 2^N possibilités
ce qui est très important

Démonstration
Cela semble marcher

Algorithme

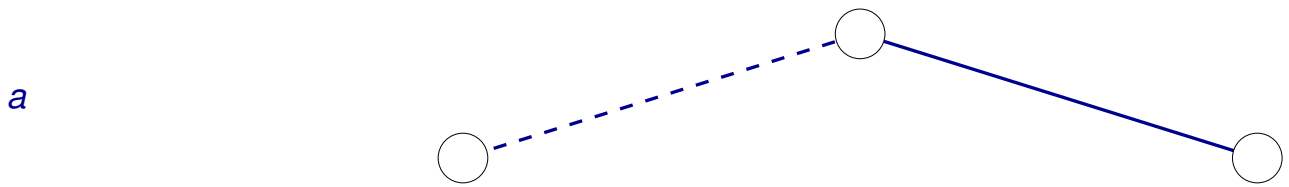
- Représentation des sous-ensembles
- Codage binaire

EXEMPLE : $n = 4$

$$\mathcal{E} = \{a, b, c, d\}$$

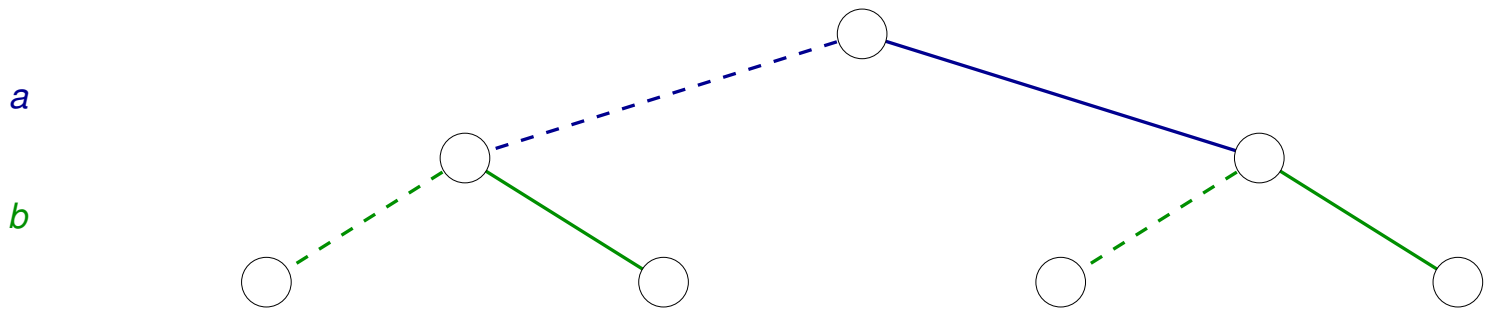
EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



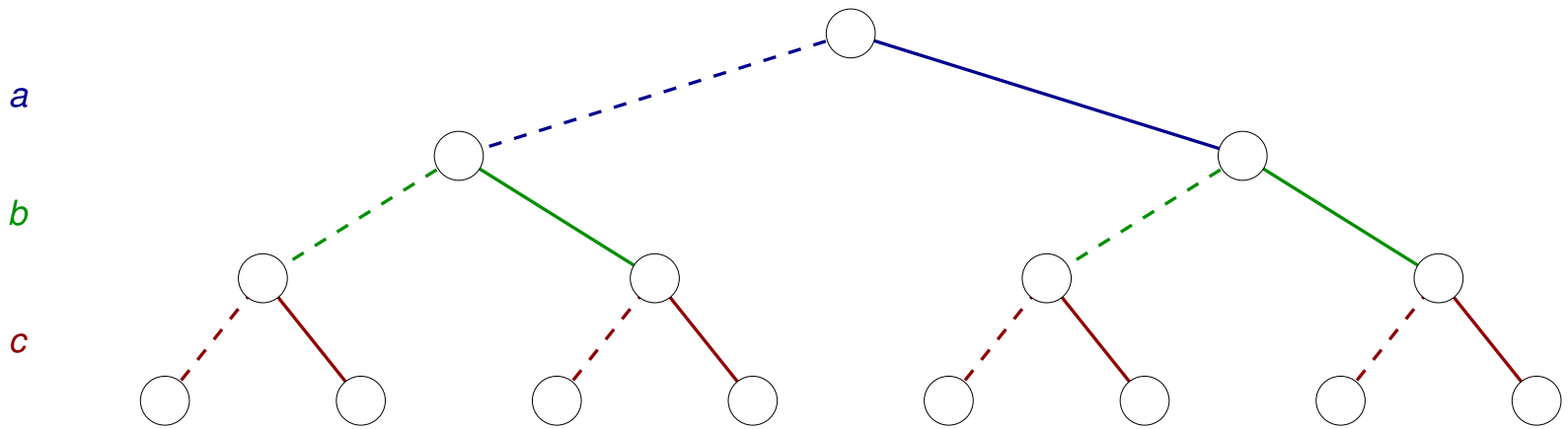
EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



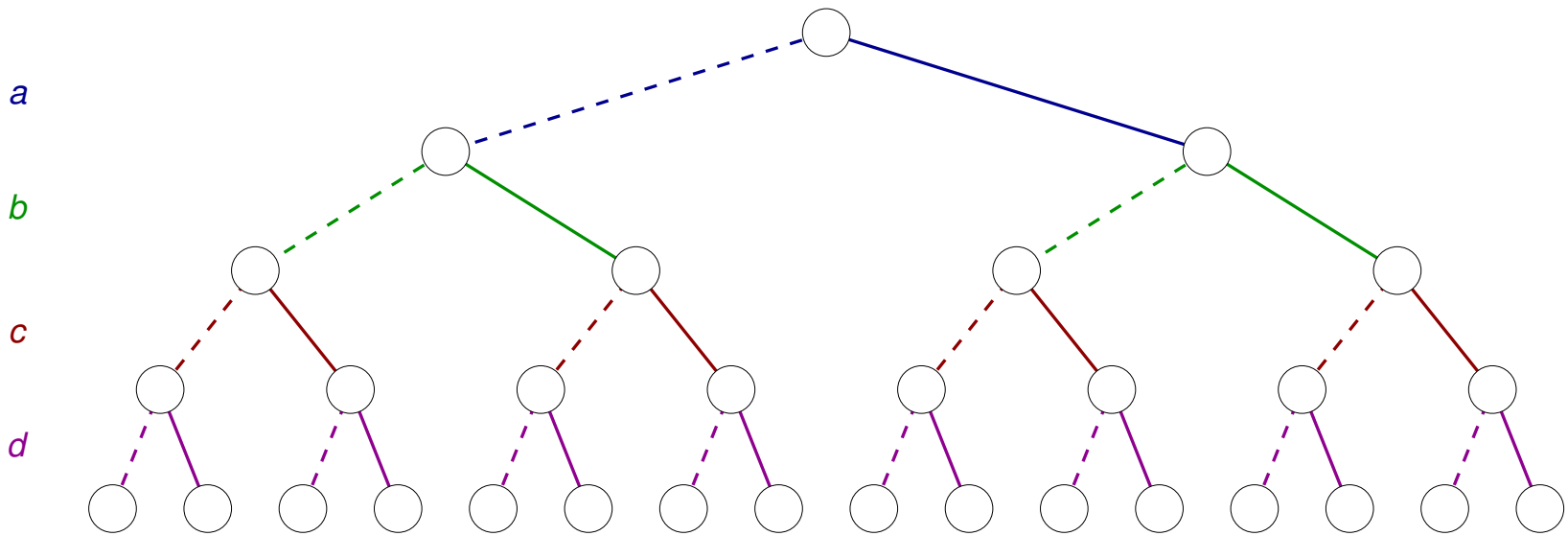
EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



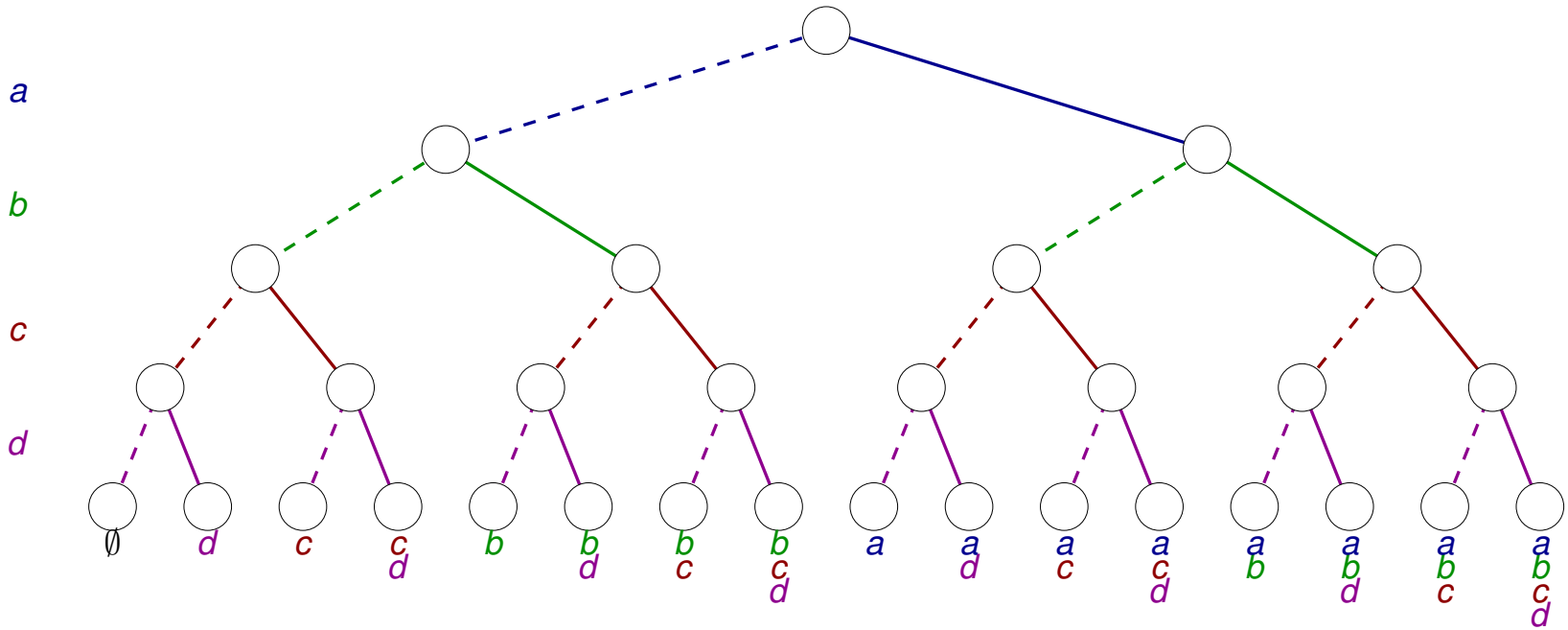
EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



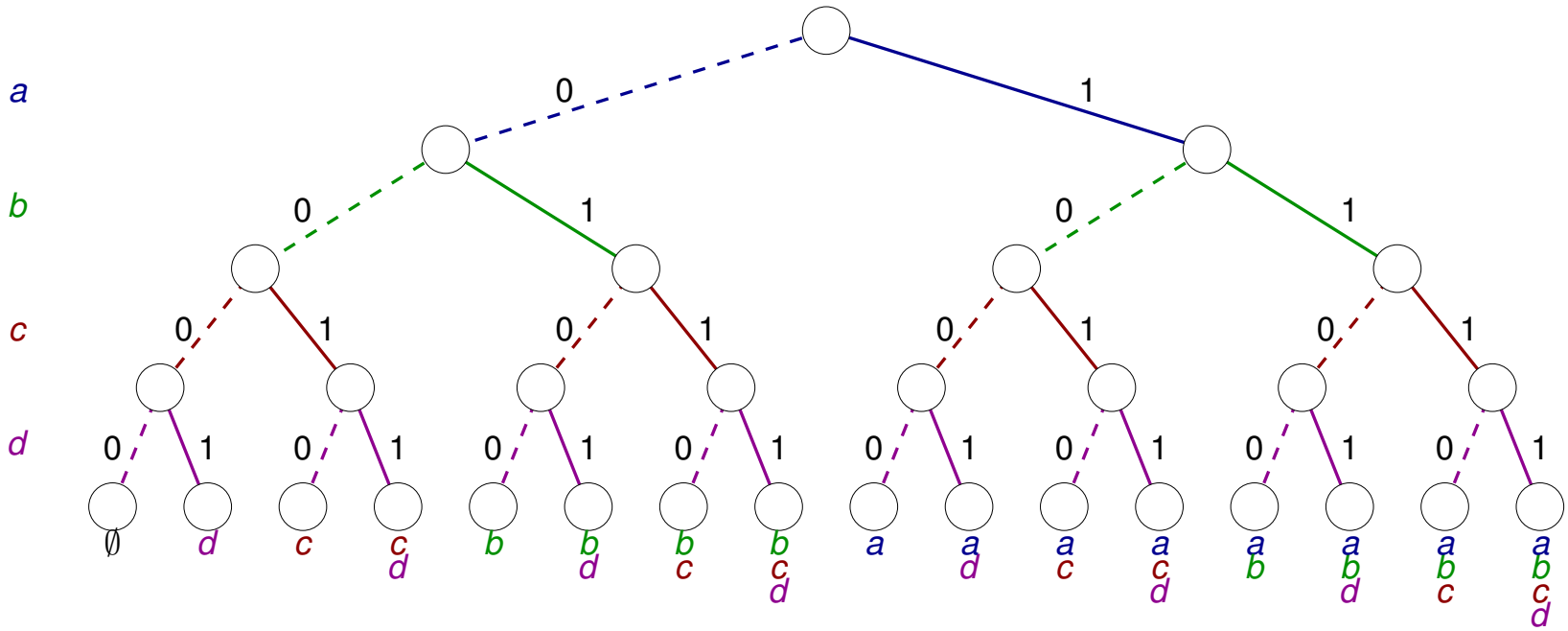
EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



EXEMPLE : $n = 4$

$\mathcal{E} = \{a, b, c, d\}$



À chaque élément on associe un booléen selon qu'il est présent ou non dans la partie.
On a une **bijection** entre l'ensemble des parties d'un ensemble de taille n et l'ensemble des vecteurs de bits de dimension n (choix d'un rang par élément)

Ici la partie $\{b, d\}$ pourrait être codée par

a	b	c	d
0	1	0	1

ÉNUMÉRATION DIRECTE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_Codage` (Y)
Données: $Y = \{y_0, \cdots, y_{n-1}\}$ ensembles d'éléments
Résultat: Une et une seule visite de chaque partie de Y

for $i = 0$ **to** $2^n - 1$
  Écrire i en binaire
  Convertir le vecteur de bits en une partie p_i de Y
  `Visiter` p

- Coût de l'énumération 2^n étapes
- Parcourt fixé par la numérotation des éléments de Y
- Difficile de faire le lien entre le nombre entier i et des propriétés de la partie p_i (par exemple le cardinal)

ÉNUMÉRATION RÉCURSIVE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X, Y)

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$ `Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Comment se fait le passage de paramètres ?
- Quel est l'appel initial ?
- Que fait l'algorithme ?
- Quel est le coût de cet algorithme ?

ÉNUMÉRATION RÉCURSIVE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X, Y)

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$ `Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Comment se fait le passage de paramètres ? passage par valeurs
- Quel est l'appel initial ?
- Que fait l'algorithme ?
- Quel est le coût de cet algorithme ?

ÉNUMÉRATION RÉCURSIVE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X,Y)

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X\backslash\{x\},Y)$
  `énumérer_visiter_parties` $(X\backslash\{x\},Y\cup\{x\})$

- Comment se fait le passage de paramètres ? passage par valeurs
- Quel est l'appel initial ? `énumérer_visiter_parties` $(X,\emptyset)$
- Que fait l'algorithme ?
- Quel est le coût de cet algorithme ?

ÉNUMÉRATION RÉCURSIVE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X,Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Comment se fait le passage de paramètres ? passage par valeurs
- Quel est l'appel initial ? `énumérer_visiter_parties` $(X,\emptyset)$
- Que fait l'algorithme ?
- Quel est le coût de cet algorithme ?

ÉNUMÉRATION RÉCURSIVE DES PARTIES D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X, Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Comment se fait le passage de paramètres ? passage par valeurs
- Quel est l'appel initial ? `énumérer_visiter_parties` $(X, \emptyset)$
- Que fait l'algorithme ?
- Quel est le coût de cet algorithme ? 2^n appels, avec n la taille de X

ÉNUMÉRATION DES PARTIES D'UN ENSEMBLE : PREUVE

procédure `énumérer_visiter_parties` (X, Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?
- Comment faire la preuve ?

ÉNUMÉRATION DES PARTIES D'UN ENSEMBLE : PREUVE

procédure `énumérer_visiter_parties` (X, Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?
  `énumérer_visiter_parties` $(X,\emptyset)$ visite une et une seule fois chaque partie de X
- Comment faire la preuve ?

ÉNUMÉRATION DES PARTIES D'UN ENSEMBLE : PREUVE

procédure `énumérer_visiter_parties` (X,Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?

  `énumérer_visiter_parties` $(X,\emptyset)$ visite une et une seule fois chaque partie de X
- Comment faire la preuve ?
  - Correction partielle
  - Terminaison

`énumérer_visiter_parties` (X,Y)
visite une et une seule fois chaque partie de $X \cup Y$ contenant Y

CORRECTION (1)

```
procédure énumérer_visiter_parties (X,Y)
    if X = ∅
     └ Visiter (Y)                                                    (0)
    else
     │ x =Choisir (X)
     │ énumérer_visiter_parties (X\{x},Y)                             (1)
     └ énumérer_visiter_parties (X\{x},Y ∪ {x})                        (2)
```

Correction partielle

- pré-condition de l'appel : X et Y sont 2 ensembles disjoints
- Si $X = \emptyset$ (on est dans le cas de base **(0)**) la récursion s'arrête et on visite Y, comme Y est la seule partie de $X \cup Y$ contenant Y (car X est vide) le résultat est correct.
- Si $X \neq \emptyset$ (on est dans le cas général) alors les paramètres des appels récursifs **(1)** et **(2)** sont également disjoints. Pour un élément x donné de X, les parties de $X \cup Y$ contenant Y, soit ne contiennent pas x et sont donc visitées par l'appel **(**1**)**, soit contiennent x et sont donc visitées par l'appel **(2)**. Ici chaque partie visitée ne l'est qu'une seule fois car les deux appels visitent des parties différentes.
- lors de l'appel initial la pré-condition est vérifiée
- au retour de l'appel toutes les parties de $X \cup Y$ contenant Y ont été visitées

CORRECTION (2)

procédure `énumérer_visiter_parties` (X, Y)

if $X = \emptyset$
  `Visiter` (Y) **(0)**
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$ **(1)**
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$ **(2)**

Terminaison

- Soit $V(X, Y) = |X|$,
- $V(X, Y)$ est un entier positif et strictement décroissant sur l'arbre des appels.

Donc l'arbre des appels est fini et donc l'appel se termine.

CORRECTION D'UNE PROCÉDURE RÉCURSIVE

Correction partielle

- Principe de la preuve (de correction partielle) d'une procédure récursive

  Si, avec l'hypothèse que les appels récursifs sont corrects et que les paramètres vérifient la pré-condition, on prouve que le corps de la procédure est correct, alors le code de la procédure est correct.
- Appel initial

  Les paramètres de l'appel initial vérifient la pré-condition.
- Terminaison

  Si l'appel se termine alors la post-condition implique la correction de l'algorithme.

Terminaison

- Variant associé aux appels
- V fonction des paramètres d'appel (entier)
- V décroissant
- V borné inférieurement

l'appel récursif se termine.

COÛT DE L'ALGORITHME

```
procédure énumérer_visiter_parties (X, Y)
    if X = ∅
        Visiter (Y)                                              (0)
    else
        x = Choisir (X)
        énumérer_visiter_parties (X\{x}, Y)                      (1)
        énumérer_visiter_parties (X\{x}, Y ∪ {x})                (2)
```

- Nombre d'appels effectués par l'algorithme $C(N)$ (N taille de X lors de l'appel initial)
  remarque : le coût ne dépend que de la taille du premier paramètre de l'appel.
- Équation de récurrence

  $$C(N) = C(N-1) + C(N-1);$$
  $$C(0) = 1$$

  $$C(N) = 2^N.$$

- Coût en temps exponentiel dans la taille de la donnée.
- Coût en mémoire : hauteur de l'arbre d'appel N

  $\mathcal{O}(N^2)$ si recopie des paramètres

  $\mathcal{O}(N)$ si passage par référence

ÉNUMÉRATION DES PARTIES DE CARDINAL k D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X, Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?
- Comment faire la preuve ?

ÉNUMÉRATION DES PARTIES DE CARDINAL k D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X,Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de cardinal k de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  if $|Y| = k$ `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?

  `énumérer_visiter_parties` $(X,\emptyset)$ visite une et une seule fois chaque partie de X de cardinal k
- Comment faire la preuve ?

ÉNUMÉRATION DES PARTIES DE CARDINAL k D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X,Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de cardinal k de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  if $|Y| = k$ `Visiter` (Y)
else
  $x =$`Choisir` (X)
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?

  `énumérer_visiter_parties` $(X,\emptyset)$ visite une et une seule fois chaque partie de X de cardinal k
- Comment faire la preuve ?
  - Correction partielle : fixée par la condition dans le if
  - Terminaison : héritée de l'énumération des parties

ÉNUMÉRATION DES PARTIES DE CARDINAL k D'UN ENSEMBLE

procédure `énumérer_visiter_parties` (X,Y)
Données: X et Y ensembles disjoints d'éléments
Résultat: Une et une seule visite de chaque partie de $X \cup Y$ de cardinal k de la forme $p \cup Y$ avec $p \subset X$

if $X = \emptyset$
  `Visiter` (Y)
else
  $x =$`Choisir` (X)
  if $|X| - 1 + |Y| \geqslant k$ `énumérer_visiter_parties` $(X \backslash \{x\}, Y)$
  if $|Y| + 1 \leqslant k$ `énumérer_visiter_parties` $(X \backslash \{x\}, Y \cup \{x\})$

- Que faut-il prouver ?

  `énumérer_visiter_parties` $(X,\emptyset)$ visite une et une seule fois chaque partie de X de cardinal k
- Comment faire la preuve ?
  - Correction partielle : fixée par la condition dans le if ou dans les conditions de branchement
    Invariant complémentaire $|Y| \leqslant k \leqslant |X| + |Y|$
  - Terminaison : héritée de l'énumération des parties

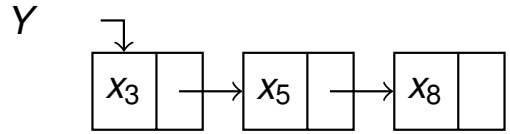
SOUS-ENSEMBLE

$X = \{x_1, x_2, \cdots, x_n\}$ ensemble de n objets distincts..

Y Sous-ensemble de X

Par exemple, $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ et $Y = \{x_3, x_5, x_8\}$

- liste d'éléments de X : chaînée, doublement chaînée, circulaire, avec sentinelle ...



- fonction indicatrice d'appartenance de l'élément au sous-ensemble

1	2	3	4	5	6	7	8
0	0	1	0	1	0	0	1

EXERCICES

- Proposer une(des) structure(s) de donnée pour coder une partie d'un ensemble
- Réécrire l'algorithme avec cette structure de donnée
- Écrire la procédure lorsque le passage d'argument se fait par référence
- Adapter l'algorithme pour `Visiter` toutes les parties de cardinal k d'un ensemble. On pourra introduire des conditions d'explorations des branches de l'arbre (ne pas faire les appels inutiles).

PROBLÈME

Le problème SOMME

Étant donné un ensemble de n nombres entiers $\{x_1, \cdots, x_n\}$ et un entier S, existe-t'il un sous-ensemble d'indices distincts $\{i_1, \cdots, i_k\}$ tels que

$$x_{i_1} + \cdots x_{i_k} = S$$

Le cas échéant donner toutes les solutions possibles.

1. Écrire un algorithme de recherche d'une solution
2. Écrire la preuve de l'algorithme.
3. Proposer des conditions de branchement.
4. Programmer l'algorithme et évaluer le nombre d'appels effectués.
5. Quels pré-calculs peut-on faire pour améliorer l'algorithme ?

On pourra écrire un générateur de jeux de tests, qui à partir d'un germe de générateur aléatoire, fournit un ensemble de n entiers. On pourra faire varier la valeur de S pour analyser l'efficacité des conditions de branchement.

UNE IMPLÉMENTATION EN C

```
void enumere(int i,int N,int* Partie,int* Valeur,int Residu,int SommeCourante, int Objectif)
// Partie est un vecteur de booleens: de 0 a i-1 on a la representation de Y, la partie courante
// X est l'ensemble des elements non encore choisis X est l'ensemble i, i+1,..., N-1
// Somme Courante est la somme de la partie courante Y
// Objectif est somme que l'on souhaite atteindre
// N est la taille de l'ensemble

// invariant SommeCourante <= Objectif <= SommeCourante + Residu
{
    if (i == N) {// on a parcouru tous les elements
            Affiche(N,Partie,Valeur,Objectif);
    }
    else
    {
        Partie[i]=1;
        if ( SommeCourante+Valeur[i] <= Objectif)
            enumere(i+1,N,Partie,Valeur,Residu-Valeur[i],SommeCourante+Valeur[i],Objectif);
        Partie[i]=0;
        if ( Objectif <= SommeCourante+Residu-Valeur[i] )
            enumere(i+1,N,Partie,Valeur,Residu-Valeur[i],SommeCourante,Objectif);
    }
```

UNE VERSION PYTHON

Une version basée sur la structure set

```
def enumeration_partie_set(x, y=None):
    """ énumère les parties de x u y contenant y
    x, y : ensembles disjoints
    résultat : affichage des parties
    """
    y = set() if y == None else y
    if not x:
        print(y)
    else:
        elem = next(iter(x))
        enumeration_partie_set(x-{elem}, y)
        enumeration_partie_set(x-{elem}, y | {elem})
```

UNE VERSION PYTHON

Une version basée sur la structure set

```
def enumeration_partie_set(x, y=None):
    """ énumère les parties de x u y contenant y
    x, y : ensembles disjoints
    résultat : affichage des parties
    """
    y = set() if y == None else y
    if not x:
        print(y)
    else:
        elem = next(iter(x))
        enumeration_partie_set(x-{elem}, y)
        enumeration_partie_set(x-{elem}, y | {elem})
```

Une version basée sur la structure list

```
def enumeration_partie_liste(x, y=None):
    y = [] if y == None else y
    if not x:
        print(y)
    else:
        elem = x.pop()
        enumeration_partie_liste(x, y)
        y.append(elem)
        enumeration_partie_liste(x, y)
        x.append(y.pop())
```

UGA UFR IM²AG Université Grenoble Alpes

COMPLEXITÉ D'ALGORITHMES

La **complexité** ($\sim$ le coût) d'un algorithme est l'ordre de grandeur du **nombre d'opérations** élémentaires qu'il exécute en fonction de la **taille des données**.

Le temps d'exécution d'un programme est lié à la complexité de l'algorithme qu'il met en œuvre.
n : taille des données

Complexité	Temps d'exécution	Pour $n = 10^3$	Pour $n = 10^6$
$\mathcal{O}(n)$	$k \times n$	$k \times 10^{-6}$ s	$k \times 10^{-3}$ s
$\mathcal{O}(n^3)$	$k \times n^3$	k s	$k \times 30$ ans
$\mathcal{O}(2^n)$	$k \times 2^n$	$> 10^{300}$ ans	...

(sur un processeur à 1 GHz)

COMPLEXITÉ DE PROBLÈME

La complexité d'un **problème** est celle du **meilleur** algorithme qui le résout.

Par exemple :

- la complexité de la recherche d'un élément dans un tableau est $\mathcal{O}(n)$
- la complexité de la recherche d'un élément dans un tableau trié est $\mathcal{O}(\log n)$
- la complexité du tri de n valeurs dans un tableau est $\mathcal{O}(n \times \log n)$
- la complexité de l'algorithme "classique" de multiplication de matrices carrées $n \times n$ est $\mathcal{O}(n^3)$ mais la complexité de ce problème, actuellement inconnue, est de la forme $\mathcal{O}(n^{2+\varepsilon})$ algorithme de Strassen en $\mathcal{O}(n^{2,807})$.

COMPLEXITÉ DE PROBLÈME

La complexité d'un **problème** est celle du **meilleur** algorithme qui le résout.

Par exemple :

- la complexité de la recherche d'un élément dans un tableau est $\mathcal{O}(n)$
- la complexité de la recherche d'un élément dans un tableau trié est $\mathcal{O}(\log n)$
- la complexité du tri de n valeurs dans un tableau est $\mathcal{O}(n \times \log n)$
- la complexité de l'algorithme "classique" de multiplication de matrices carrées $n \times n$ est $\mathcal{O}(n^3)$ mais la complexité de ce problème, actuellement inconnue, est de la forme $\mathcal{O}(n^{2+\varepsilon})$ algorithme de Strassen en $\mathcal{O}(n^{2,807})$.

Et pour certains problèmes (parmi ceux proposés en info sans ordi) :

- trouver un sous-ensemble de somme fixée (*Pilzegal*)
- garantir le remplissage d'un camion/sac à dos . . . (*Alice déménage*)

on ne connaît pas d'algorithme **efficace** qui permette de les résoudre.

algorithme **efficace** = algorithme de complexité polynomiale en fonction de la taille des données $\mathcal{O}(n^k)$

COMPLEXITÉ DE PROBLÈME (3)

Pour ces problèmes, on connaît par contre un algorithme “inefficace” qui consiste à **énumérer tous** les chemins du graphe, **énumérer toutes** les façons de placer les cartons dans le camion . . . .

Par contre, on peut **vérifier facilement** (efficacement, i.e. avec un coût polynômial) si une solution répond au problème :

- vérifier si un circuit est hamiltonien
- vérifier que le remplissage du camion répond aux contraintes

CLASSIFICATION DES PROBLÈMES

Soit l'algorithme suivant :

- exhiber une solution (au hasard ?) $\rightarrow$ polynomial
- vérifier qu'elle est correcte $\rightarrow$ polynomial

si la réponse est oui : le problème est résolu
si la réponse est non : on n'a pas eu de chance

On a donc un algorithme **non déterministe** polynomial qui résout notre problème. (si on a de la chance . . .)

CLASSIFICATION DES PROBLÈMES

Classes de problèmes

- Classe $\mathcal{P}$: problèmes pour lesquels il existe un algorithme polynômial
- Classe $\mathcal{NP}$: problèmes pour lesquels il existe un algorithme non déterministe polynômial

Évidemment : $\mathcal{P} \subseteq \mathcal{NP}$

Question à 1 million de dollars : Clay Institute

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

Problèmes classiques dans $\mathcal{NP}$

- existence d'une solution à une formule logique (SAT)
- sous-ensemble de somme fixée (Pilzegal)
- empilement (Alice déménage)
- existence un k-coloriage de graphe
- existence d'un chemin passant par tous les sommets d'un graphe une et une seule fois (chemin hamiltonien)
- existence d'un k-stable (k-couverture, k-clique) de graphe

ET POUR UN PROBLÈME DIFFICILE ...

Aider la chance

- Énumérer tous les cas possibles (force brute) : de l'ordre de 2^n avec n la taille des entrées
- Stratégies (heuristiques) prometteuses :
  - placer les plus hauts cartons d'abord
  - . . .
- Approximation (facteur d'approximation)

SYNTHÈSE

Informatique sans ordinateur

- exemples simples pour initialiser l'activité
- le coût apparaît rapidement pour les problèmes difficiles
- énumération (branchements/coupes)
- nécessité de l'ordinateur pour résoudre des problèmes plus gros

SYNTHÈSE

Informatique sans ordinateur

- exemples simples pour initialiser l'activité
- le coût apparaît rapidement pour les problèmes difficiles
- énumération (branchements/coupes)
- nécessité de l'ordinateur pour résoudre des problèmes plus gros

Complexité de problème

1. Tester sur des exemples simples
2. Trouver des cas extrêmes
3. reconnaître un schéma algorithmique (ou un problème classique)
4. utiliser l'algorithme adapté
5. si le problème est difficile, réorienter la spécification (approximation)

Énumération

1. Méthode de décomposition
2. Schéma récursif
3. Preuve de correction partielle et terminaison
4. Complexité (en général exponentielle)