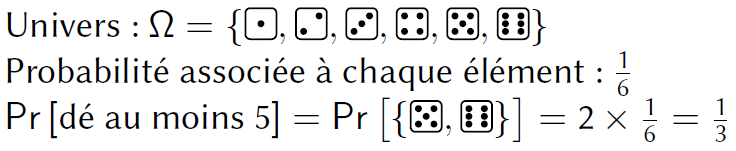
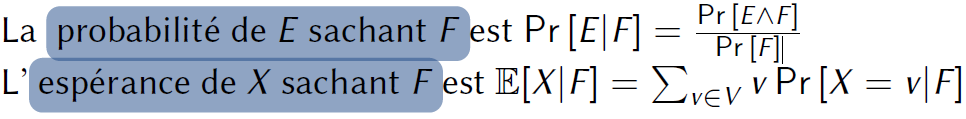
## Aléatoire :

Univers : ensemble des résultats possibles de l’expérience probabiliste

* Événement primitif : un élément de l’univers / un résultat possible
* Évènement : sous-ensemble de l’univers / ensemble de résultats possibles



Une variable aléatoire est une fonction X : Ω → V de l’univers dans un ensemble V.

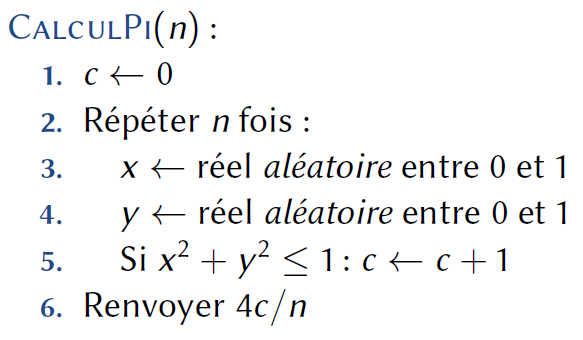


Graphiquement : Pr [E|F ] = « proportion de F occupée par E »

Deux évènements sont indépendants si Pr [E ^ F ] = Pr [E] Pr [F ]

Deux variables aléatoires sont indépendantes si Pr[X = v ^ Y = w] = Pr [X = v] Pr [Y = w] pour tous v, w

Exemple d’algorithmes probabilistes :



Algorithmes déterministes → suite fixée

* Entrée de l’algorithme : graine
* changer la graine doit modifier complètement la suite
* choix de graine : quelque chose d’imprévisible ou au contraire de fixé

Suite (Xn) définie par Xn+1 = (aXn + c) mod m

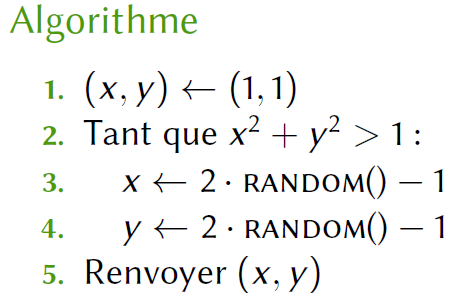
* X0 doit être fixé : graine du générateur
* a, c et m définissent le générateur
* parfois : seuls certains bits de Xn sont utilisés
* m premier, c = 0, a primitif modulo m (Lehmer)
* m = 2k, c = 0, a = 3 ou 5 mod 8
* m et c premiers entre eux, a - 1 divisible par les facteurs premiers de m

Simulation de lois :

La loi d’une variable aléatoire X : Ω→V est la donnée de Pr [X = v] pour tout v 𝛜 V

* Uniforme : Pr[X = v] = 1/|V| pour tout v 𝛜 V
* Bernoulli(p) : V = {0; 1} et Pr [X = 1] = p (pile ou face)
* Binomiale(p; n) : V = {0….. n} et Pr [X = k] = (k parmi n) pk(1-p)k n pièces : #pile
* Géométrique(p) : V = N et Pr [X = n] = p(1-p)n-1 #lancers avant pile

Tirer aléatoirement x et y dans le disque D de centre (0,0) de rayon 1 :



Utiliser l’aléatoire en algorithmique :

* On suppose une source parfaite
* On utilise des lois simples souvent
* Si loi complexe algorithme rarement

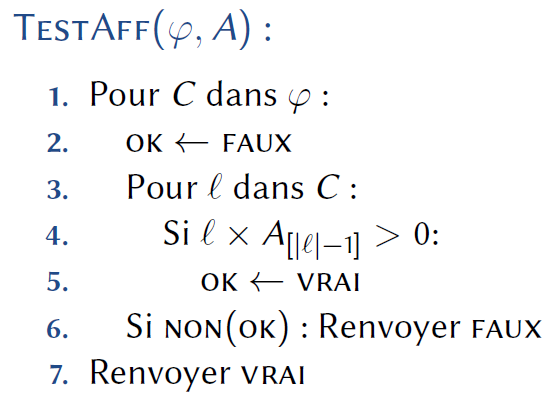
Implantations :

* bibliothèque random de Python (pseudo-aléa)
* On ignore la différence avec le vrai aléatoire

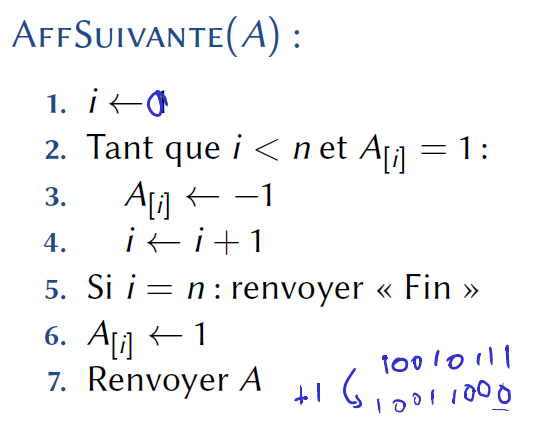
## Exhaustif :

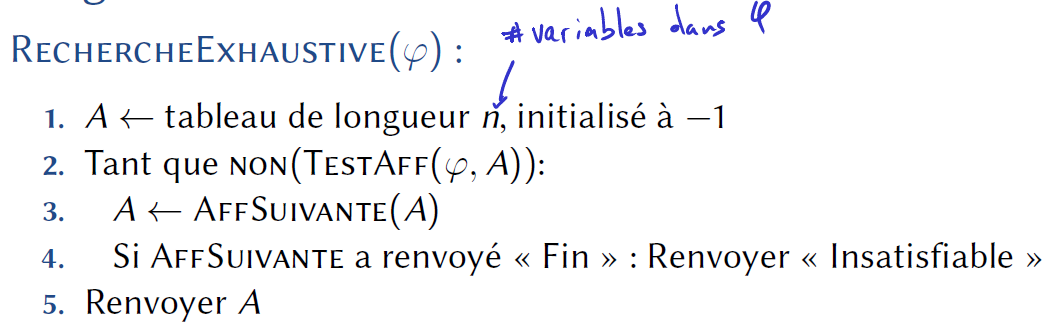
Algorithme par recherche exhaustive : tester toutes les affectations possibles.

Tester les affectations de logique :



Il faut ensuite parcourir ces affectations :





L’algorithme RechercheExhaustive trouve une affectation satisfaisante s’il en existe une, et renvoie « Insatisfiable » sinon, en temps O(|ϕ|2n).

Principe de la recherche exhaustive :

* Parcourir toutes les solutions possibles
* Tester chaque solution

Complexité = O(nombresolutions \* (coutTest + CoutPassageSuivant))

Les ensembles de solutions ont souvent une structure mathématique à exploiter.

La recherche exhaustive est en général exponentielle.

Deux façons de produire des solutions :

* Algorithme pour passer d’une solution à une autre
* Algorithme pour produire la liste de toutes les solutions, puis parcours.

Par exemple par un algorithme récursif, mais peut avoir des problèmes de mémoire.

Atouts :

* Technique algorithmique conceptuellement simple : on teste toutes les possibilités
* Analyse de complexité simple : essentiellement le nombre de solutions
* Parfois le mieux qu’on sache faire !
* Point de départ d’algorithmes plus sophistiqués (backtrack, . . . )

Limites :

* Solution algorithmiquement coûteuse (quasiment toujours exponentiel)
* Écriture en détail et implantations parfois difficiles
* Problèmes éventuels de mémoire

**Backtrack :**

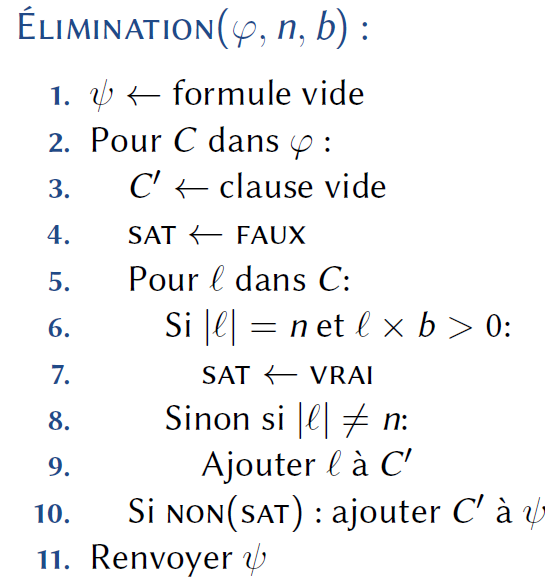
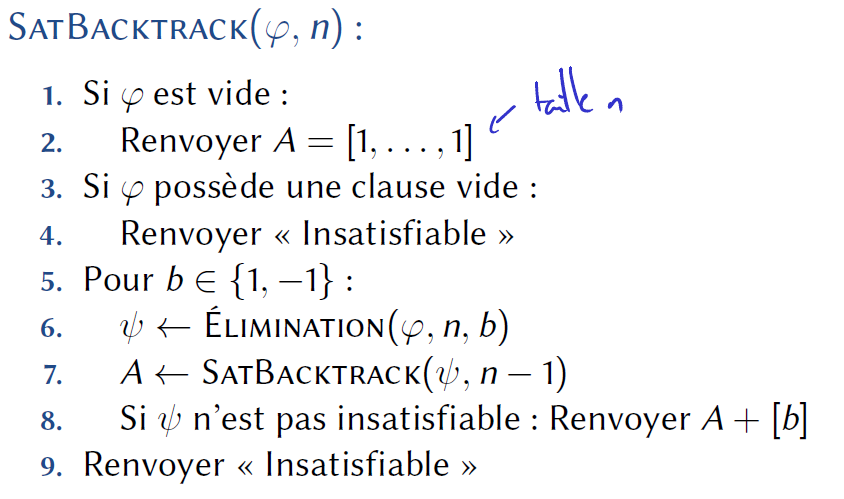
Exemple : Principe de l’algorithme des clauses :

Élimination des variables une à une :

* ⇔construction pas à pas d’une affectation
* Cas de bases :

Clause vide insatisfiable → formule insatisfiable

Formule vide satisfiable

L’algorithme SatBacktrack trouve une affectation satisfaisante de ϕ s’il en existe une,

et renvoie « Insatisfiable » sinon, en temps O(|ϕ| 2n).

Lequel des deux algorithmes est le meilleur ?

Dans le pire cas, aucun des deux

Mais dans des cas favorables, SatBacktrack peut être beaucoup plus rapide

* Arbre des solutions très peu exploré

Cas les pires pour SatBacktrack

* Arbre à explorer (quasiment) en entier
* RechercheExhaustive peut alors être légèrement plus rapide

**Le backtrack est de la recherche exhaustive récursive.**

Comparaison avec la recherche exhaustive :

Recherche exhaustive :

* parcours itératif des solutions possibles
* test de chaque solution

Backtrack :

* construction récursive progressive des solutions
* test de solutions partielles

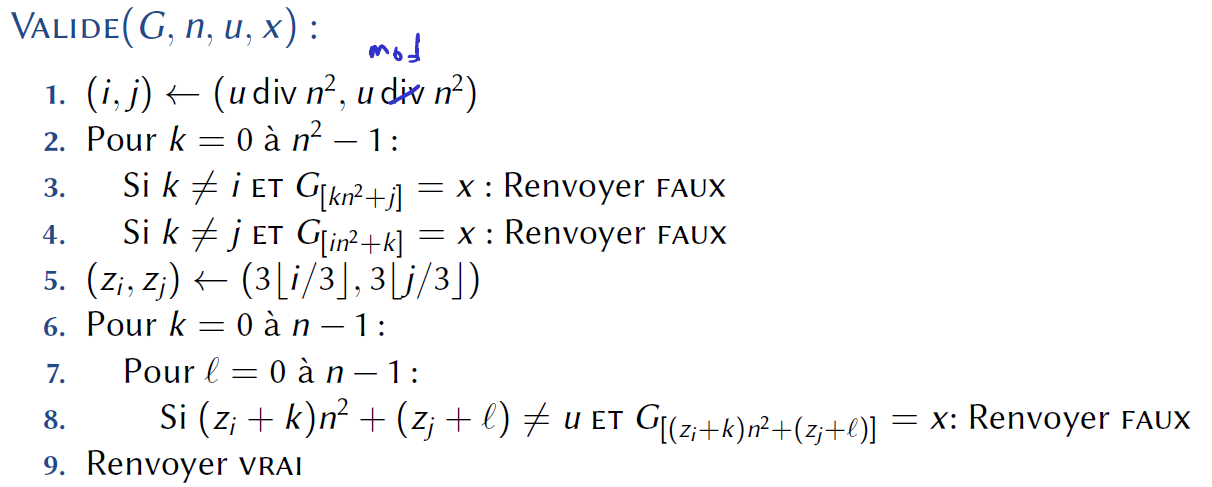
L’algorithme récursif de backtrack est un parcours en profondeur de l’arbre des solutions.

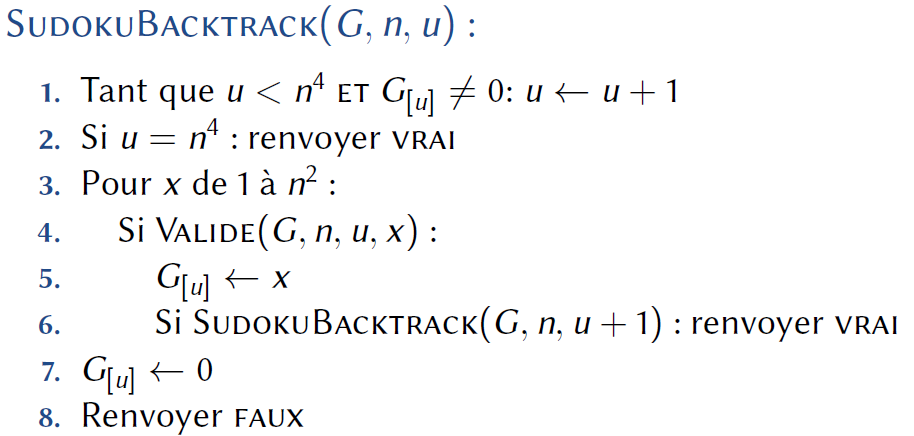
Analyse de complexité

* Algorithme **récursif** → équation de récurrence pour la complexité
* Arbre des **solutions** → complexité proportionnelle au nombre de solutions

**Backtrack** : Construction d’une solution pas à pas et retour sur nos pas si échec. Géré par les appels récursif.

Exemple Sudoku :



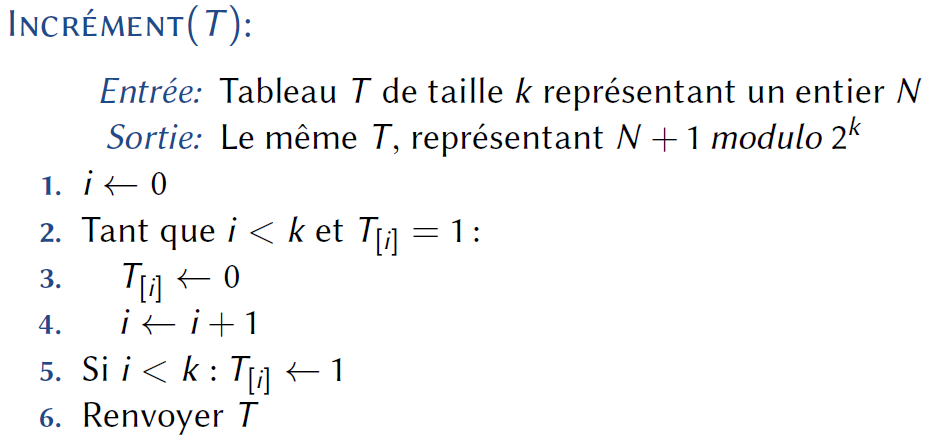


**Deux techniques proches :**

* Recherche exhaustive et backtrack sont deux faces d’une même pièce
* Algorithme souvent itératif pour la recherche exhaustive (mais parfois récursif)
* Algorithme quasiment toujours récursif pour le backtrack
* Principale différence : test de solutions partielles pour le backtrack

## Analyse :

Analyse amortie : Incrémenter un entier de 0 à 2k − 1



Pire cas : N × O(k) = O(Nk)

* Coût global : O(N) car certains Incréments peu chers
* Remarque : valable aussi pour N Incréments quelconques

Le coût amorti de l'algorithme incrément est O(1) par appel à l'incrément.

**Méthode de l’agrégat :**

Si le coût global pour N appels est Ctot(N), le coût amorti est Ctot(N)/N

Agrégat : on somme les coûts et on divise.

Mise en oeuvre

* Regarder globalement les N appels comme une seule exécution
* Regrouper des opérations venant de différents appels pour mieux compter

Exemple pour l'incrément :

* Compter le nombre total d’inversions du bit T[0], du bit T[1], etc.

**Méthode de l'acompte :**

Payer plus que le vrai coût à certains appels, et moins à d’autres.

Acompte : On imagine que les coûts sont de l’argent, et le compte doit être positif.

Mise en oeuvre

* À chaque appel,
* fixer une taxe à payer (éventuellement nulle pour certains appels)
* utiliser l'acompte pour payer le coût de l’appel
* L'acompte doit toujours rester positif
* Coût amorti par opération : taxe maximale payée
* Remarque : plus difficile que l’agrégat, mais plus puissant

Exemple pour l'incrément :

* Chaque passage de bit de 0 à 1 coûte 2, et chaque passage de 1 à 0 est gratuit
* À chaque appel : prélèvement de 1 par inversion de bits
* Coût amorti : 2

**Méthode du potentiel :**

Associer aux appels les plus chers une augmentation de potentiel.

Potentiel : métaphore de l’énergie potentielle en physique

Mise en oeuvre

* Définir une fonction potentiel Φ ≥ 0 sur l’objet manipulé

Valeur initiale Φ0

Valeur après i appels : Φi ≥ Φ0

* Si le coût d’un appel est ci , son coût amorti est ai = ci + Φi − Φi−1
* Le coût total amorti de N appels est 

Exemple pour l'incrément :

* Potentiel du tableau T : Φ(T) = nombre de 1 dans T
* Si Incrément(T) remet k bits à 0 :
* coût ci = k + 1
* différence de potentiel : Φi − Φi−1 = k − 1
* coût amorti : k + 1 − (k − 1) = 2

La **méthode par agrégation** calcule le coût amorti de chaque opération comme une moyenne arithmétique des opérations dans le cas le plus défavorable. Le coût amorti est donc un majorant du coût moyen. Cette méthode affecte le même coût amorti à chaque opération.

La **méthode comptable** « fait payer » à des opérations simples le coût des opérations complexes qu'elles occasionneront. Le surcoût, de certaines opérations ancillaires d'une suite, est ainsi totalement compensé par celles qui en sont à l'origine.

La **méthode du potentiel** « fait payer » les coûts à la structure de données et non plus à chaque opération comme dans la méthode comptable. Conceptuellement, elle s'appuie sur un potentiel (un « capital ») qui est libéré au fur et à mesure pour anticiper le surcoût de certaines opérations futures.

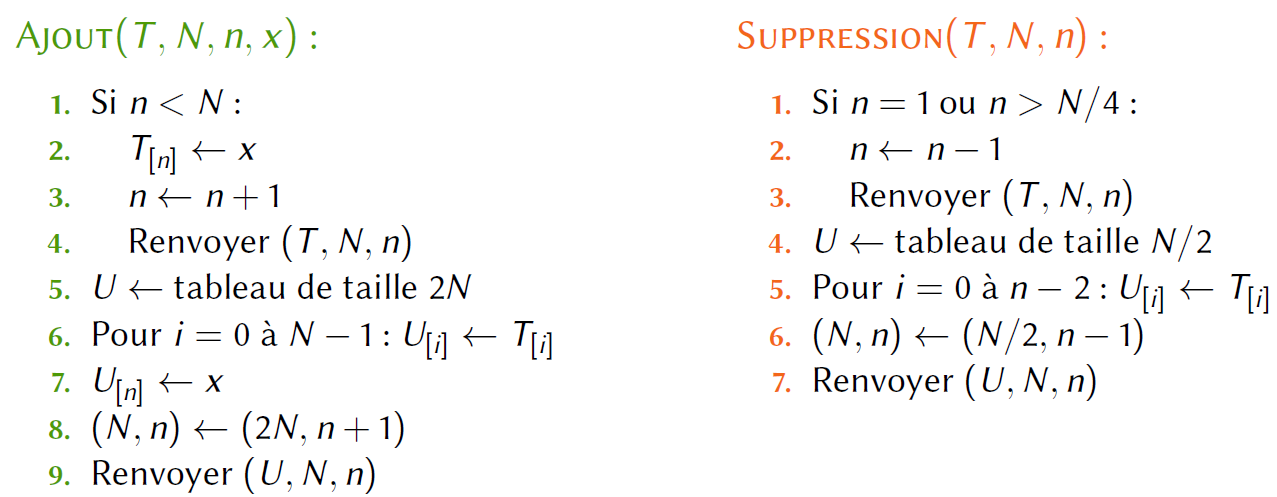
Techniques plus ou moins faciles

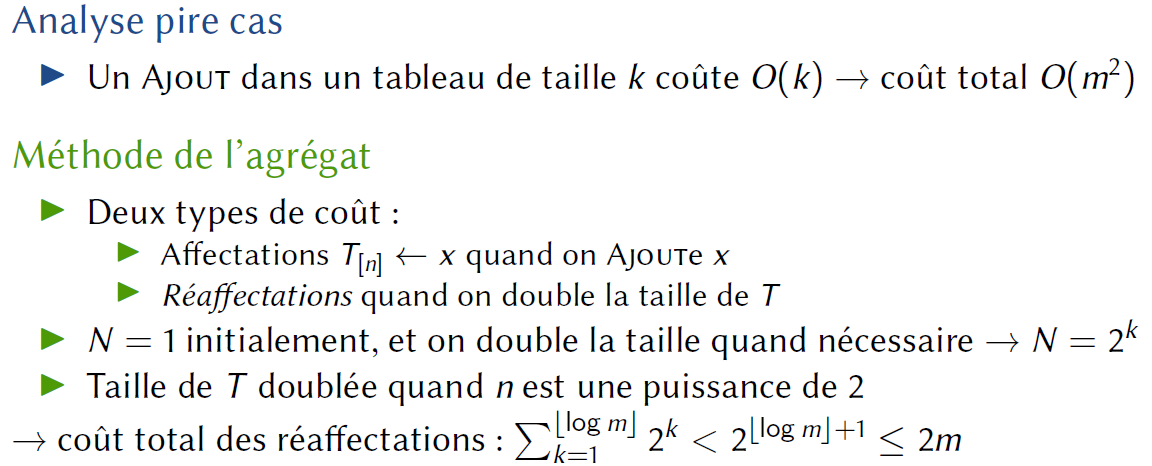
* Agrégat : idée la plus évidente, mais demande une compréhension globale
* Méthodes de l'acompte et du potentiel : plus difficile à mettre en oeuvre, mais compréhension locale

Idées communes aux méthodes de l'acompte et du potentiel

* Calcul direct d’un coût amorti pour chaque appel
* Preuve globale que le coût amorti défini est valide
* Forme d’analyse pire cas avec une notion de coût modifiée

Les tableaux dynamiques :

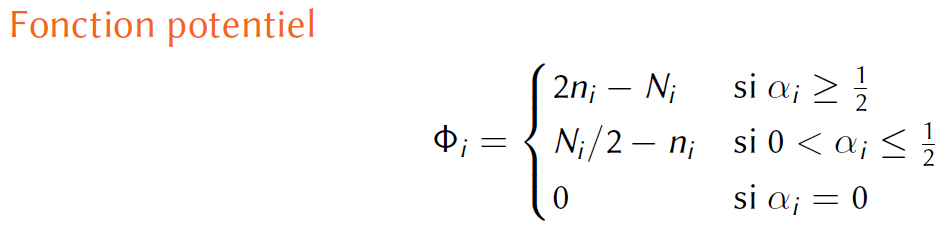




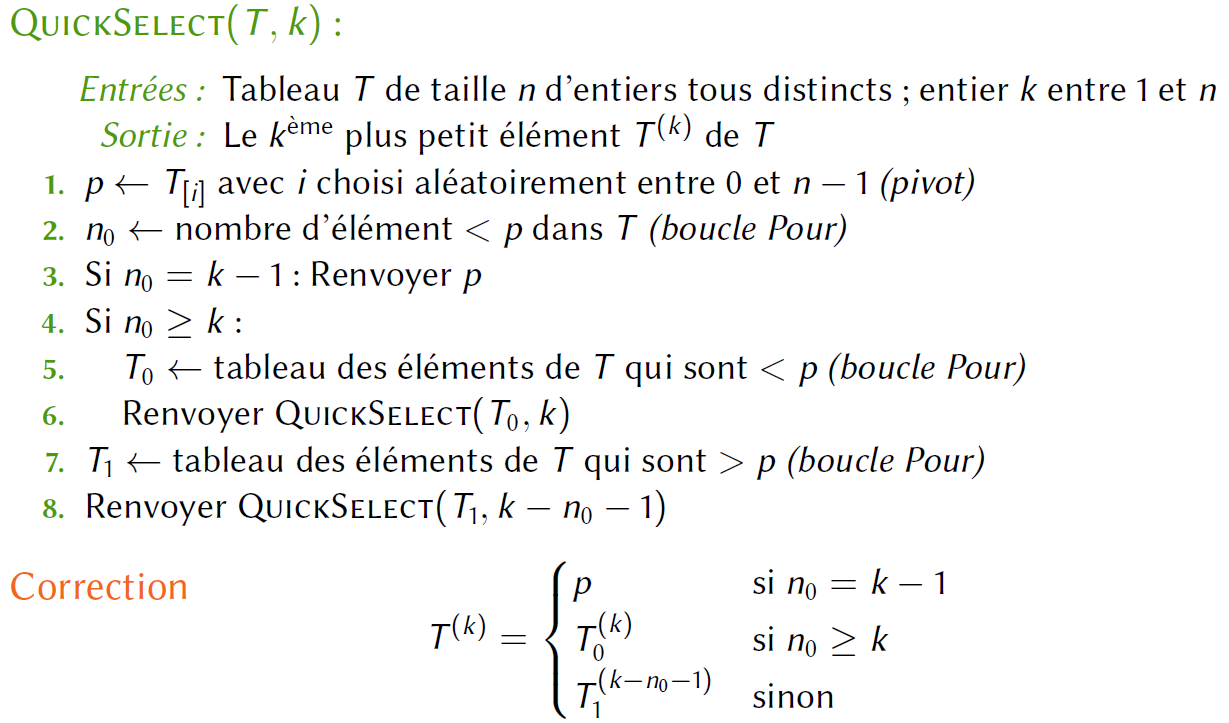
Notations :

Après la ième opération,

* ni : nombre d’élément dans le tableau
* Ni : taille du tableau
* αi = ni/Ni : coefficient de remplissage
* ci : coût de la ième opération (nombre d’affectations)



**Quickselect :**

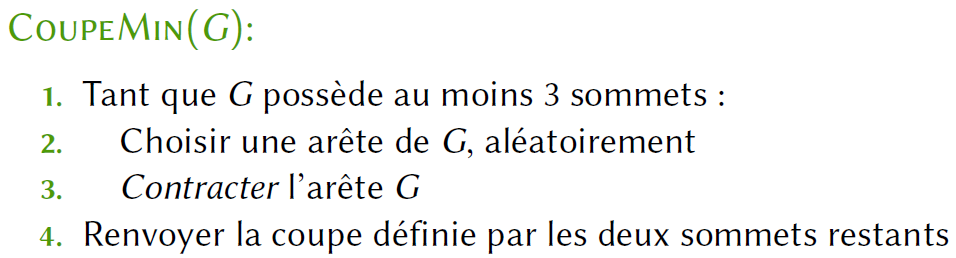


Analyse en pire cas

* Deux boucles en O(n) + appel récursif sur un tableau de taille ≤ n − 1
* Cn ≤ Cn−1 + O(n) → Cn = O(n2)

**Théorème**

Soit Cn le nombre de comparaisons effectuées par QuickSelect(T, k) où T est de taille n. Alors E[Cn] ≤ 4n, quelque soit k. Quickselect est toujours correct, l’espérance de sa complexité est O(n).



Les algorithmes probabilistes :

Un algorithme probabiliste est un algorithme qui effectue des choix aléatoires au cours de son exécution.

Algorithmes de type Las Vegas :

* Son résultat ne dépend pas des choix aléatoires
* Sa complexité dépend des choix aléatoires

Etudes de la complexité :

* Modélisé par une variable aléatoire
* Calcul de l’espérance de la variables aléatoire
* Exemple quickselect

Las Vegas : « toujours correct, souvent rapide »

Algorithmes de type Monte Carlo :

* Si sa complexité ne dépend pas des choix aléatoires
* Son résultat dépend des choix aléatoires

Etude de la correction :

* Probabilité de succès : probabilité que l’algorithme soit correct
* Exemple : CoupeMin répété

Monte Carlo : « toujours rapide, souvent correct »

Atlantic City : « souvent correct, souvent rapide »

Pourquoi des algorithmes probabilistes ?

* Algorithmes souvent simples et efficaces
* Parfois : meilleure complexité que les algorithmes déterministes
* Et pourquoi pas ? → en pratique, ils fonctionnent très bien !

Analyse des algorithmes probabilistes :

* Modélisation probabiliste, avec variable aléatoire
* Las Vegas : étude de l’espérance de la complexité
* Monte Carlo : étude de la probabilité de succès

Analyse amortie :

* Analyse de plusieurs exécutions consécutives d’un même algorithme
* Complexité amortie = temps moyen pris par les exécutions successives
* Trois techniques de preuve : agrégat, acompte et potentiel

Analyse d’algorithmes probabilistes :

* Analyse d’algorithmes qui font des choix aléatoires
* Étude de l’espérance de la complexité ou de la probabilité de succès
* Comportement moyen vis-à-vis des choix aléatoires

Analyse en moyenne :

* Analyse du comportement d’algorithmes sur des entrées aléatoires
* Calcul de l’espérance de la complexité sur une entrée aléatoire
* Question subtile : quelle distribution sur les entrées ?

## Tables de Hachage :

**Comment implanter le type dict de Python ?**

>>> d = {} (Dictionnaire vide)

>>> d[1515] = 'Bataille de Marignan' (Ajout d'élément)

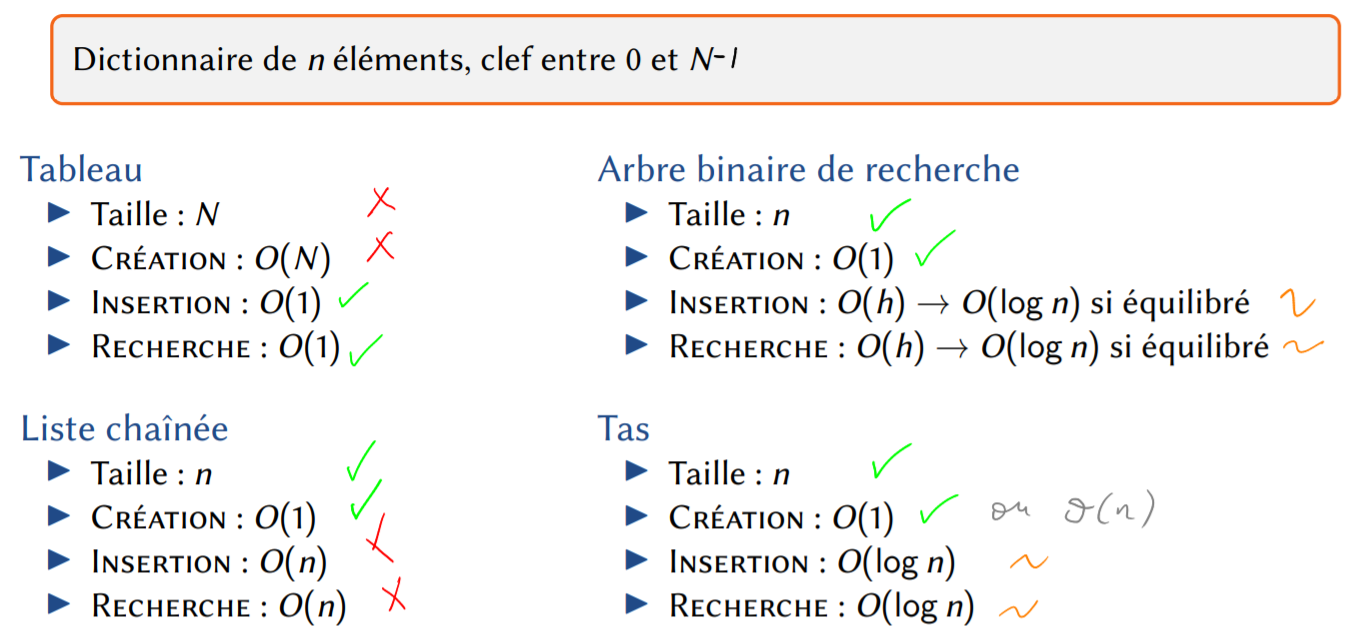
>>> d[1492] = 'Colomb en Amérique'

>>> d[1492] = 'Ascension du Mont Aiguille' (modification)

>>> 1789 in d (Recherche)

**Dictionnaire en algorithmique :**

* Ensemble de couple (clef, valeur)
* Opérations disponibles (But : elles doivent être rapides) :
* *Création* d’un dictionnaire vide
* *Insertion* d’un couple
* *Réinsertion* d’un valeur
* *Recherche* d’une clef - renvoie la valeur ou une erreur



**Table de hachage : Clefs**

* Univers U des clefs possibles : U = {0, . . . , N − 1}
* Clefs utilisées : K ⊂ U, de taille n

**Table de taille m :**

* Indices entre 0 et m − 1
* Une case contient une valeur, voire plusieurs
* Une case peut être vide

**Fonction de hachage :**

* Fonction h : U → {0, . . . , m − 1}

**Collisions** : h(k1) = h(k2) plusieurs valeurs dans une liste chaînée.

**Fonctions de hachage :**

Choix d’une fonction h : {0, . . . , N − 1} → {0, . . . , m − 1}

* Fonction utilisée pour un ensemble de clefs K de taille n <= N

Attention : Avec N>=m forcément des collisions h(k1) = h(k2).

Pas le choix : une fonction de hachage doit être choisie aléatoirement !

On tire h uniformément parmi les fonctions U dans {0, . . . , m-1)

**Représentation de h :**

* Pour chaque k, une valeur h(k) → tableau H de taille N
* Tirage de h → tirage uniforme et indépendant de chaque H[k] dans {0, . . . , m − 1}

**Avantage et inconvénient :**

* Avantage : très bonnes propriétés probabilistes
* Pour tout k1 != k2, Prh[h(k1) = h(k2)] = 1/m
* Inconvénient : totalement irréaliste → tableau de taille N, temps du tirage

**Parfois utilisé en théorie car :**

* les preuves sont (un peu) simples
* les résultats obtenus parfois (très) proches du comportement pratique
* Objectif : modèle réaliste avec propriétés proches

On fixe un ensemble H de fonctions de hachage et on tire h uniformément dans H.

Un ensemble H de fonctions de {0, . . . , N − 1} dans {0, . . . , m − 1} est universel si pour

tout k1 != k2, Prh∈H[h(k1) = h(k2)] ≤ 1/m.

L’ensemble de toutes les fonctions est universel. . . mais irréaliste !

On sait construire des ensembles H universels réalistes.

Ensemble universel intéressant

* Ensemble pas trop gros → représentation de h assez petite
* Tirer uniformément h ∈ H doit être efficace
* Calculer h(k) doit être rapide

**Résolution des collisions :**

**Deux (familles de) solutions :**

Mettre plusieurs éléments dans une même case

* Résolution par chaînage
* Hachage parfait

Trouver une autre case libre : adressage ouvert

Résolution par chaînage :

**Algorithmes**

1. *Recherche* de k :
2. Calcul de h(k)
3. Parcours de la liste contenue dans T[h(k)]
4. Complexité : O(l(k)) où l(k) est la taille de la liste T[h(k)]

**Insertion de (k, v) :**

1. Idem *Recherche* pour savoir si k est dans le dictionnaire
2. Si k apparaît déjà dans la liste contenue dans T[h(k)], on remplace sa valeur par v
3. Sinon, on ajoute (k, v) à la liste T[h(k)]
4. Complexité : O(l(k)) où l(k) est la taille de la liste T[h(k)]

Une opération coûte O(L) ou L = maxk𝜖Kl(k)

**Complexité :**

* Complexité espérée de chaque opération : O(α) où α = n m est le taux de remplissage
* Si le taux est autour de 1 : O(1) en moyenne
* Attention : l’espérance du pire cas n’est pas O(α) !
* E[maxk l(k)] != maxk E[l(k)]

La résolution par chaînage marche bien de manière amortie, mais certaines opérations

peuvent être coûteuses

**Pourquoi des listes chaînées ?**

Arbres binaires de recherche ou tas dans chaque case

* Complexité moyenne en O(log α)
* Complexité pire cas en maxk log l(k)

Résolution par hachage parfait :

Si la table est suffisamment grande, il n’y a pas de collision (avec bonne probabilité).

**Objectif :**

Un table de hachage statique

* L'*insertion* de n couples
* puis uniquement des Recherches

Chaque *Recherche* de complexité O(1) dans le pire cas

Taille totale de la table : O(n)

Temps de création de la table : O(n)

**Principes**

> Une table principale T avec fonction de hachage h, de taille m = n

> Chaque case T[i] contient une table de hachage secondaire S(i)

> La table S(i) est de taille mi, avec fonction de hachage hi : U → {0, . . . , mi − 1}

> Chaque case de S(i) ne peut contenir qu’un élément (pas de chaînage)

> Clef k en case S(i)

> [j] où i = h(k) et j = hi(k) → *Recherche* en O(1)

**Algorithme**

1. Choisir m = n et tirer h : U → {0, . . . , m − 1} dans un ensemble universel H

2. Calculer tous les hachés h(ki), 1 ≤ i ≤ n

3. Pour j = 0 à m − 1 :

4. mj ← nombre de clefs ki telles que h(ki) = j

5. Créer une table S(j) de taille m2j

6. Tirer hj : U → {0, . . . , m2j } dans un ensemble universel

7. Insérer dans S(j) tous les couples (ki, vi) tq h(ki) = j

8. En cas de collision, goto 5.

Complexités

> L’*Insertion* en complexité amortie O(1) → création complète de la table en temps O(n)

> *Recherche* en temps O(1) à tous les coups

> Mémoire nécessaire : O(n) → pas de perte de place

Résolution par adressage ouvert : Si la case pour insérer (k, v) est occupée, trouver une autre case !

**Formellement**

m fonctions de hachage h1, . . . , hm

> 1er essai : *Insertion* en case h1(k)

> 2ème essai : *Insertion* en case h2(k)

> . . .

> même essai : *Insertion* en cas hm(k)

Condition : pour tout k, {h1(k), . . . , hm(k)} est une permutation de {0, . . . , m − 1}

**Algorithmes**

*Recherche* : explorer T[h1(k)], T[h2(k)], . . .

1. si on trouve k → gagné !
2. si on trouve une case vide → k n’est pas dans T

*Insertion* : explorer jusqu’à trouver une case vide

Construire les m fonctions à partir d’une (ou deux) fonctions de hachage.

**Quelques possibilités pratiques :**

> Sondage linéaire : hi(k) = (h(k) + i) mod m

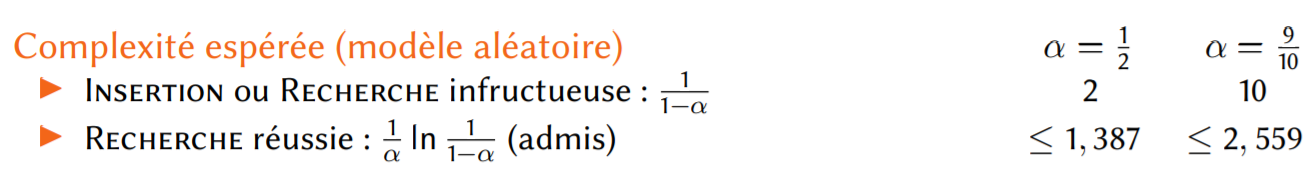
> Sondage quadratique : hi(k) = (h(k) + ai² + bi) mod m (bien choisir a et b !)

> Sondage binaire : hi(k) = h(k) ⊕ i (si m = 2l )

> Double hachage : hi(k) = (h(1)(k) + ih(2)(k)) mod m (conditions sur h(1) et h(2))

**Idée de principe**

* Une seule table principale, un seul élément par case
* Si une case est occupée, aller ailleurs !
* Plusieurs solutions pour aller ailleurs



**Pour aller plus loin : hachage du coucou**

Deux fonctions de hachage h(1) et h(2) deux emplacements possibles par clef

*INSERTION* de (k, v) :

* *Insertion* en case h(1)(k)
* Si la case contenait (k’, v’), on le déplace à son autre emplacement
* Et récursivement. . .

Les collisions sont inévitables !

Deux familles de résolutions

Chaînage, hachage parfait, . . .

* Gérer les collisions en mettant plusieurs éléments par case
* Complexité liée au nombre maximal d’éléments par case et à la structure de données

Adressage ouvert

* Gérer les collisions en cherchant une autre case libre
* Complexité liée au nombre de cases à inspecter

→ Dans les deux cas : complexité liée au nombre de collisions

**Cas des dictionnaires Python**

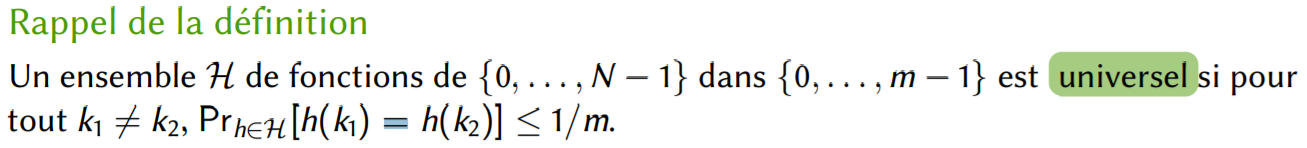
Fonction de hachage pas aléatoire ! h(i) = i mod (261 − 1) si i est un entier

Résolution des collisions par adressage ouvert

* Ordre de parcours des cases un peu complexe

Solution théoriquement faible, à peu près correcte en pratique

**Famille universelle de fonctions de hachage :**



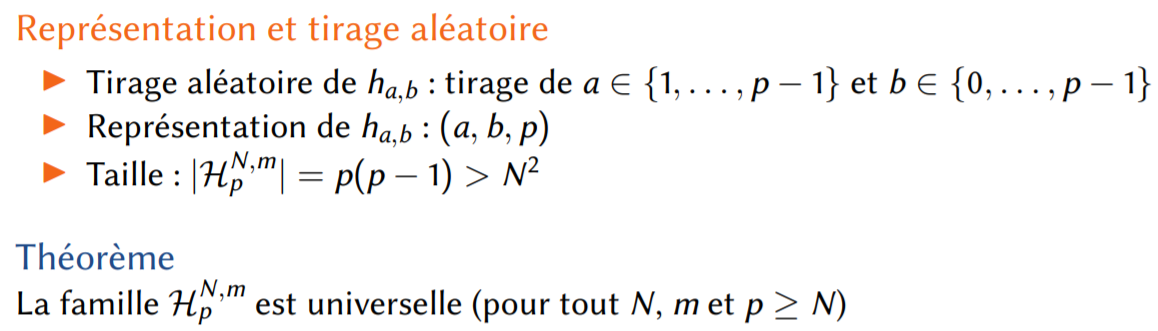
Contraintes sur H

1. Suffisamment grand pour avoir une probabilité ≤ 1/m
2. Suffisamment petit pour savoir représenter h ∈ H avec une place raisonnable
3. Suffisamment simple pour savoir tirer h ∈ H en temps raisonnable

H de taille polynomiale en N

* Nombre de couples de clefs possibles (2 parmi N) → au moins autant de fonctions h
* Représentation d’une fonction h en O(log N) bits → similaire à une clef
* Tirage aléatoire en O(log N) → équivalent au calcul de h(k)

Hachage multiplicatif :



**Utilisation de la famille :**

Création du dictionnaire : tirage aléatoire de a et b

Complexité du calcul de ha,b(k) = ((ak + b) mod p) mod m

* Additions, multiplications, divisions d’entiers ≤ p² : O(log² p) = O(log² N)
* Taille d’une clef → O(log N)

Conclusions :

Gestion des collisions

* Chaînage → complexité amortie O(1) dans le modèle universel
* Hachage parfait → complexité pire cas O(1) dans le modèle universel
* Adressage ouvert → complexité amortie O(1) dans le modèle aléatoire
* Difficile : même résultat dans le modèle (fortement-)universel

Construction de familles universelles

* ha,b(k) = (((ak + b) mod p) mod m) fournit une famille universelle
* Construction d’autres familles universelles
* Meilleures garanties : familles fortement universelles

## Approximation :

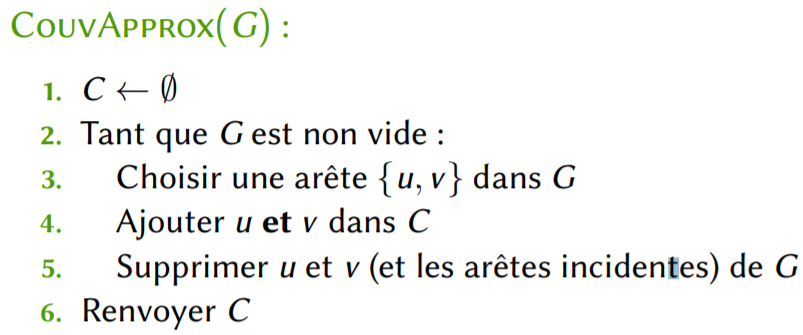
Problème de la couverture par sommets : (Vertex Cover) : Un graphe G = (S,A)

Cela nous renvoie un sous-ensemble C (le plus petit possible) qui couvre toutes les arêtes.

Algorithme par recherche exhaustive

* Tester tous les sous-ensembles possibles, par taille croissante
* Complexité : O(2nn²) où n est le nombre de sommets
* O(2kn²) si la couverture minimale est de taille k

On ne cherche plus la couverture la plus petite possible mais une couverture assez petite.



L’algorithme CouvApprox a une complexité O(n²).

**Somme Partielle :**

Entrée : Un ensemble E d’entiers strictement positifs, un entier cible T

Sortie : Un sous-ensemble S ⊂ E dont la somme est ≤ T

Objectif : Trouver S de somme la plus grande possible (la plus proche possible de T)

**Solution exacte :**

Recherche exhaustive et backtrack :

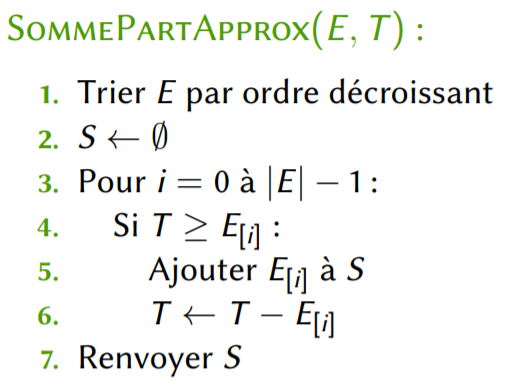
> Parcours de tous les sous-ensembles S ⊂ E

> Complexité O(n2n) où n = |E|

> Backtrack si entiers tous positifs

> Complexité O(2n)

Somme Partielle fait partie des problèmes NP-complets

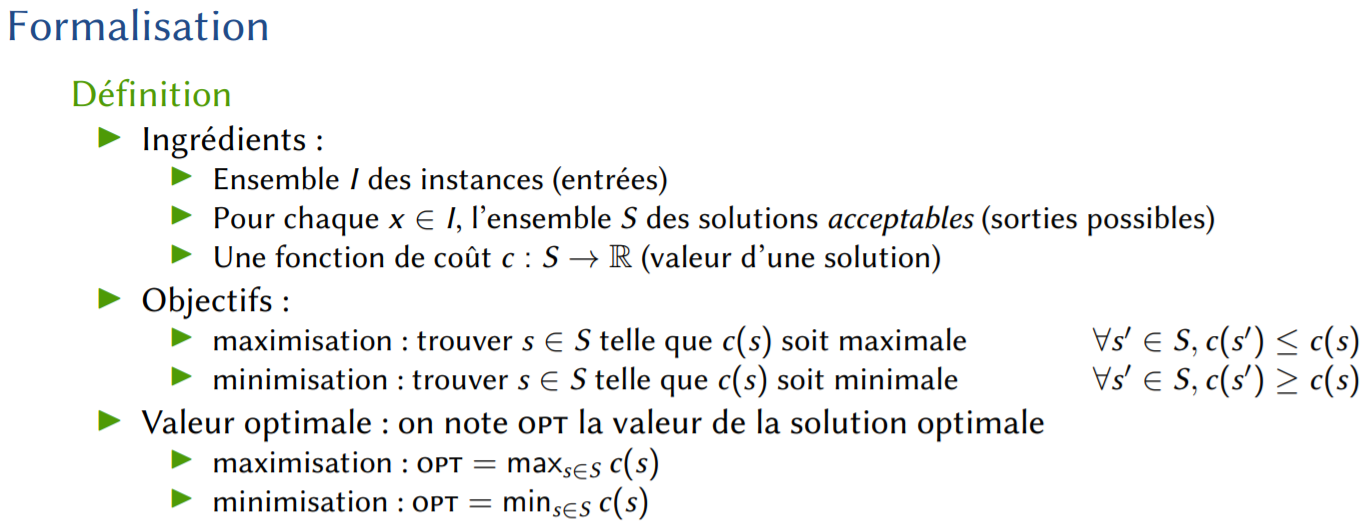


L’algorithme Somme Partielle a une complexité O(n log n)

**Les algorithmes d’approximation :**

Problèmes d’optimisation. Cadre général :

* Problème de maximisation : sur une entrée, trouver une solution qui maximise une certaine fonction
* Problème de minimisation : sur une entrée, trouver une solution qui minimise une certaine fonction



**Comment résoudre un problème d’optimisation de manière exacte ?**

Recherche exhaustive et backtrack

* Parcours (intelligent) de toutes les solutions, en gardant la meilleure
* Fonctionne toujours ; complexité (en général) exponentielle

Algorithmes gloutons

* Construction d’une solution en optimisant localement à chaque étape
* Fonctionne parfois. . . ; complexité souvent assez bonne

Programmation dynamique

* Décomposition du problème en sous-problèmes, et résolution par tailles croissantes
* Fonctionne souvent ; complexité (en général) exponentielle mais meilleure qu’en recherche exhaustive

Algorithmes de compromis

* Algorithmes efficaces → complexité polynomiale, voire linéaire
* Algorithmes non exacts → solution de valeur proche de l’optimal

Un **algorithme d’α-approximation** est un algorithme qui pour tout entrée x renvoie une solution s ∈ S telle que

1. maximisation : α · OPT ≤ c(s) ≤ c(s) 0 < α < 1
2. minimisation : OPT ≤ c(s) ≤ α · c(s) α > 1

Le réel α est appelé facteur d’approximation de l’algorithme.

**Analyser un algorithme d’approximation :**

Objectif : Montrer que pour tout entrée, l’algorithme renvoie une solution s vérifiant

* c(s) ≥ α · OPT (si maximisation)
* c(s) ≤ α · OPT (si minimisation)

Deux bornes à trouver (cas max.) (cas min.)

* Trouver une borne c1 telle que c(s) ≥ c1 c(s) ≤ c1
* Trouver une borne c2 telle que OPT ≤ c2 OPT ≥ c2

→ On en déduit que α ≥ c1/c2 α ≤ c1/c2

Pour trouver le facteur d’approximation, il faut aussi une borne sur la valeur optimale !

Borne sur OPT : L’équilibrage de charge :

**Informellement**

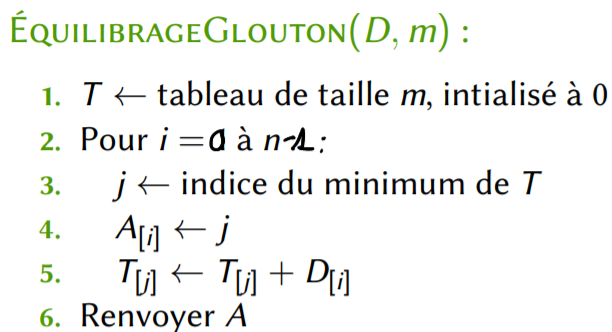
* Ensemble de n tâches à exécuter, chacune ayant une durée
* À disposition : m processeurs
* Objectif : répartir les tâches sur les processeurs, pour minimiser le temps total de calcul

**Entrée** : Tab D d’entiers positifs (durées) et entier m

**Sortie** : Tab A affectation de chaque tâche à un processeur

**Objectif** : répartir les tâches sur les processeurs, pour minimiser le temps total de clacul.

Algorithme glouton à la volée : Scénario : les tâches arrivent les unes après les autres, on doit les traiter dans l’ordre.



L’algorithme ÉquilibrageGlouton a une complexité O(nm) (ou O(n log m) avec un tas)

Algorithme glouton avec tri : Nouveau scénario : on connaît toutes les tâches à l’avance → fait-on mieux ?

Algorithme et complexité

* Même algorithme ÉquilibrageGlouton, avec tri de D initialement
* Complexité : O(n log n) pour le tri, puis pareil
* O(n(m + log n)) avec recherche naïve de minimum
* O(n(log n + log m)) avec un tas → O(n log n) car n ≥ m

Bilan :

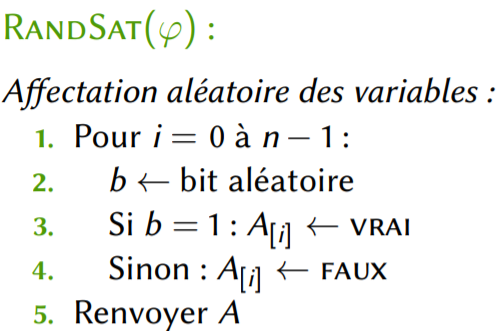
**Cas non trié**

* L’algorithme glouton est une 2-approximation
* Un peu mieux : (2 − 1/m)-approximation
* Facteur d’approximation atteint

**Cas trié**

* L’algorithme glouton fournit une 3/2-approximation
* On peut dire mieux : (4/3 − 1 /m)-approximation

Le problème MaxSAT :



L’algorithme RandSat a une complexité O(n).

**Un algorithme d’approximation :**

* RandSat est un algorithme de 1/2 -approximation pour MaxSat, de type Monte Carlo
* Il existe une version Las Vegas → tirer des affectations tant qu’elles ne satisfont pas suffisamment de clauses

Mieux ?

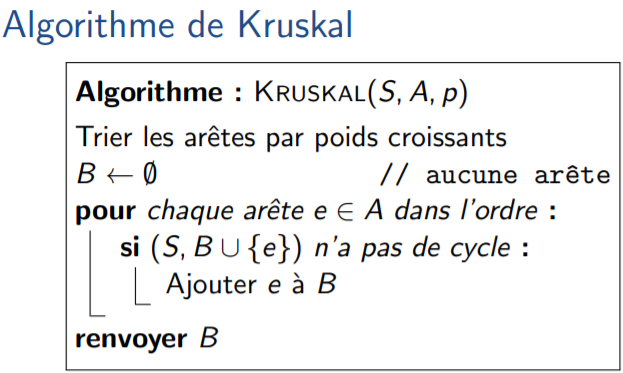
* Si la plus petite clause est de taille k → (1 − 1/2k )-approximation
* Il existe un algorithme de 3/4 -approximation, quelque soit k
* On peut dérandomiser RandSat

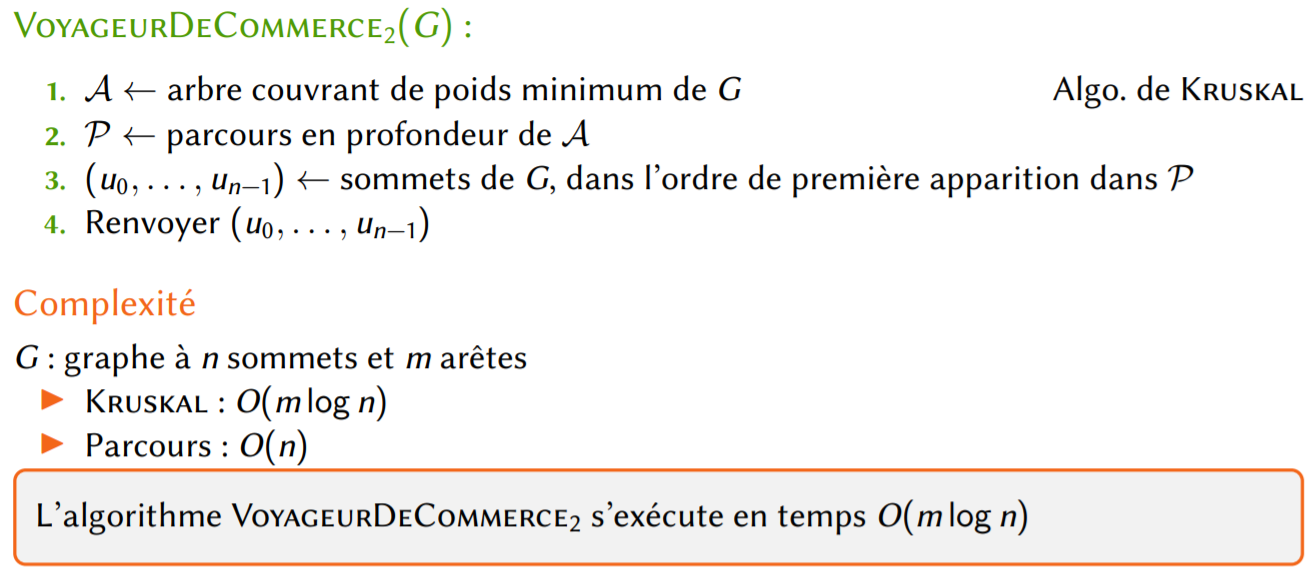
**L'algorithme de Christofides :**

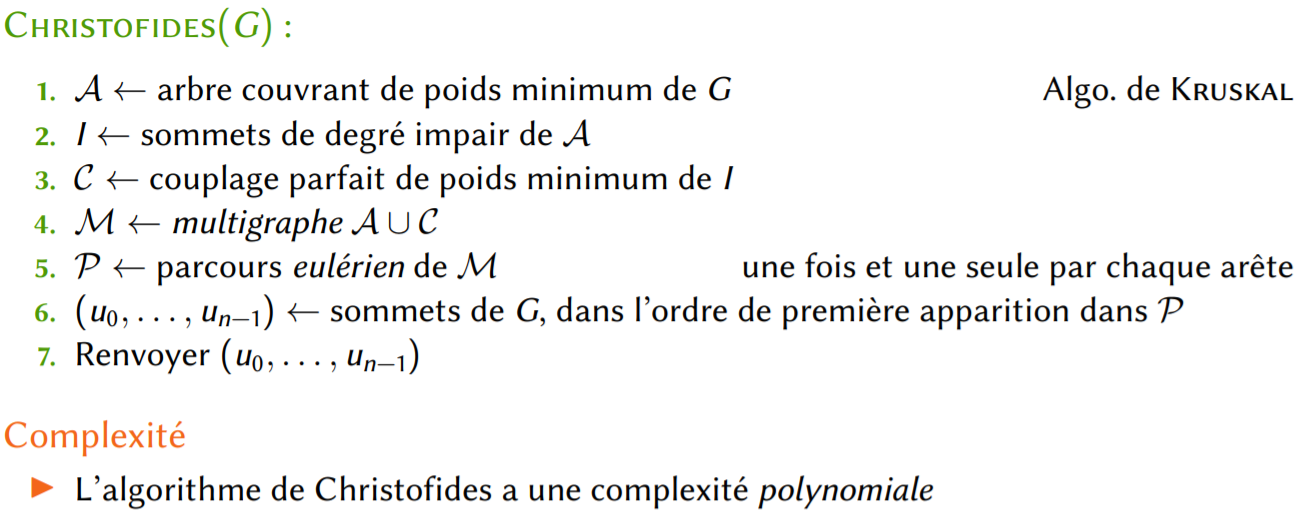
Entrée : Graphe G = (S, A) avec une longueur l(u, v) pour chaque arête vérifiant

l’inégalité triangulaire : l(u, w) ≤ l(u, v) + l(v, w) pour tous u, v, w

Sortie : Une numérotation u0, . . . , un−1 des sommets







**Algorithmes d’approximation :**

Avec l’inégalité triangulaire :

* algorithme relativement simple de 2-approximation
* algorithme plus complexe de 3/2-approximation Christofides (1976)
* algorithme très complexe de (3/2 − 1/1036)-approximation Karlin, Klein, Gharan ( 2021)

Algorithmes exacts

* Programmation dynamique en O(n22n) → nécessite l’inégalité triangulaire
* Algorithme exhaustif en O(n × n!) → marche même sans inégalité triangulaire

**Beaucoup de problèmes sont difficiles**

* Théorie de la NP-complétude HAI602I
* Deux solutions :
* algorithme exponentiel → petites instances
* algorithme d’approximation → résultat approché

**Analyse d’un algorithme d’approximation**

* Montrer que l’algorithme n’est jamais pire qu’un facteur α
* Deux ingrédients :
* borner la valeur optimale
* borner la valeur renvoyée par l’algorithme