Algorithmique avancée

Didier Lime

École Centrale de Nantes – LS2N

Dernière modification: 6 septembre 2023

Outline

Outline

Pourquoi étudier l'algorithmique ?

- Résoudre plus facilement les problèmes (*Problem solving*) ;
- Ne pas réinventer la roue ;
- Produire des programmes :
  - corrects,
  - efficaces,
  - et lisibles.

Pourquoi faire des programmes efficaces ?

Pour optimiser l'utilisation des ressources

- temps de calcul,
- mémoire,
- énergie.

Puissance de crête d'un processeur de type Core i7 = entre 100 et 200W
$\approx$ 1m^2 de panneau photovoltaïque en plein soleil

Exemple

```
1 | def fib1(n):
2 |     if n == 0 or n == 1:
3 |         return 1
4 |     else:
5 |         return fib1(n-1) + fib1(n-2)
```

```
1 | def fib2(n):
2 |     x = 1
3 |     y = 1
4 |     for i = 2 to n:
5 |         y = x + y
6 |         x = y - x
7 |
8 |     return y
```

Objectifs du cours

- Prouver la correction de programmes ;
- Analyser leur performance ;
- Connaître et savoir utiliser les principales stratégies génériques de résolution de problèmes (algorithmiques) ;
- Connaître et savoir utiliser les principaux types abstraits ;
- Comprendre le fonctionnement de leurs principales implémentations et leurs limites.

Outline

Exemple d'algorithme

Exemple

```
1  def sort(A, n):
2      for i = 1 to n - 1:
3          insert(i, A)
4
5  def insert(x, A):
6      key = A[x]
7
8      j = x - 1
9      while j ≥ 0 and A[j] > key:
10         A[j + 1] = A[j]
11         j = j - 1
12
13     A[j+1] = key
```

Outline

Propriétés des algorithmes (et programmes)

- Propriétés non-fonctionnelles :
  - Terminaison,
  - Indices de tableaux,
  - Pointeurs,
  - Débordements de capacité,
  - etc.
- Propriétés fonctionnelles : le résultat est celui attendu.
- Performances

Prouver des propriétés

- Test : non-exhaustif ;
- Preuve ≪ à la main ≫ ;
- Méthodes formelles :
  - par exemple norme DO-333 pour l'avionique
  - Preuve assistée par ordinateur (Coq, Lean, ...)
  - Model-checking (SPIN, NuSMV, Uppaal, Roméo, ...)

Exemple de propriétés

Exemple

```
1  def sort(A, n):
2      for i = 1 to n - 1:
3          insert(i, A)
4
5  def insert(x, A):
6      key = A[x]
7
8      j = x - 1
9      while j ≥ 0 and A[j] > key
10         A[j + 1] = A[j]
11         j = j - 1
12
13     A[j+1] = key
```

- tous les accès à T sont à des indices dans $[0, \text{len}(T) - 1]$;
- l'algorithme termine ;
- à la terminaison T est trié par ordre croissant.

Invariants de boucles

Pour prouver ce type de propriétés :

- Invariants de boucles :
  - Propriétés inductives ;
  - À prouver par induction sur les itérations.
- Contrats de type *assume-guarantee* :
  - Sous réserve de la **précondition** X, la **postcondition** Y est vraie
  - Version informelle des triplets de Hoare dans la logique du même nom.

Exemple d'invariants : accès au tableau

Exemple

```
1  # assume ∅
2  def sort(A, n):
3      for i = 1 to n − 1:  # invariant : après la k^e itération,
4          insert(i, T)     # i vaut k + 1 et i < n
5  #guarantee 1 ≤ i < n
6
7  # assume 1 ≤ x < n
8  def insert(x, A):
9      key = A[x]
10
11     j = x − 1
12     while j ≥ 0 and A[j] > key: # invariant : après la k^e itération,
13         A[j + 1] = A[j]         # j vaut x − 1 − k et 0 ≤ j < x
14         j = j − 1               # et donc j ≤ n − 2
15
16     A[j+1] = key                # donc ici j ≥ 0
17 #guarantee les accès sont corrects
```

Lignes 12–14 : après la k^e itération, on a $0 \leq j < x$

1. vrai pour $k = 0$, par la ligne 11 et l'hypothèse $x \geq 1$
2. si vrai pour $k \geq 0$, alors :
   - soit $j = 0$ (ou $A[x] \leq A[j]$), la boucle s'arrête et pas d'itération $k + 1$;
   - soit $j > 0$ (et $A[j] < A[x]$) et après la ligne 14, au début de l'itération $k + 1$, on a bien $0 \leq j < x$.

Terminaison des algorithmes

Définition (Ordre bien fondé)

Une relation binaire $\prec \in X \times X$ est **bien fondée** s'il n'existe pas de suite infinie $x_1, x_2, \ldots$, telle que pour tout i on a $x_{i+1} \prec x_i$.

Exemple

- Sur $\mathbb{N}$, l'ordre strict usuel $<$ est bien fondé (mais pas sur $\mathbb{Z}$!) ;
- Sur $\mathbb{N}^k$, l'ordre strict $\ll$ lexicographique $\gg$ est bien fondé : $(x_1, y_1) < (x_2, y_2)$ ssi $x_1 < x_2$ ou $(x_1 = x_2$ et $y_1 < y_2)$;
- Sur $\mathbb{N}^k$, l'ordre partiel strict $\ll$ composante par composante $\gg$ est bien fondé.

- Pour prouver la terminaison d'une boucle ou d'appel récursifs, associer aux variables/paramètres une valeur dans un ensemble bien fondé ;
- Montrer que cette valeur décroît strictement à chaque itération.

Exemple

Dans la boucle while d'`insert`, la valeur de j est dans $\mathbb{N}$ (par l'invariant qu'on a prouvé) et décroit strictement à chaque itération.

Exercice : correction fonctionnelle du tri par insertion

Exercice

On veut montrer que l'algorithme trie le tableau dans l'ordre croissant.

```
1  def sort(A, n):
2      for i = 1 to n - 1:
3          insert(i, A)
4
5  def insert(x, A):
6      key = A[x]
7
8      j = x - 1
9      while j ≥ 0 and A[j] > key
10         A[j + 1] = A[j]
11         j = j - 1
12
13     A[j+1] = key
```

- quel invariant pour la boucle dans sort ?
- quelles préconditions et postconditions pour insert ?
- quel invariant pour le while d'insert ?

Exercice : Algorithme (historique) d'Euclide

```
1 | # pour a et b strictement positifs
2 | def euclid(a, b):
3 |     while a != b:
4 |         if a > b:
5 |             a = a - b
6 |         else:
7 |             b = b - a
8 |
9 |     return a
```

Lemme

$\text{pgcd}(a, b) = \text{pgcd}(a, b - a)$

Exercice

On note a_i la valeur de a à la fin de l'itération i (donc a_0 est la valeur initiale) et on définit b_i de façon similaire.

1. Quel invariant de la boucle while pour la correction de l'agorithme ?
2. Quel invariant de la boucle while pour la terminaison ?
3. Conclure quand à la terminaison et la correction de l'algorithme en utilisant les deux invariants.

Invariants pour les programmes récursifs

- Dans les programmes récursifs, on énonce et on prouve des invariants sur la structure récursive :

Exemple

```
1  # Pour i < n = len(A)
2  def maximum(i, A):
3      if i == len(A) - 1:
4          return A[i]
5      else:
6          return max(A[i], maximum(i + 1, A))
```

Par induction/récurrence sur len(A) - i : la fonction termine et renvoit le maximum du tableau à partir de l'indice i.

Invariants pour les programmes récursifs

- Dans les programmes récursifs, on énonce et on prouve des invariants sur la structure récursive :

Exemple

```
1 | # Pour i < n = len(A)
2 | def maximum(i, A):
3 |     if i == len(A) - 1:
4 |         return A[i]
5 |     else:
6 |         return max(A[i], maximum(i + 1, A))
```

Par induction/récurrence sur len(A) - i : la fonction termine et renvoit le maximum du tableau à partir de l'indice i.

Exercice

```
1 | # pour a et b strictement positifs
2 | def euclid(a, b):
3 |     if a == b:
4 |         return a
5 |     else:
6 |         if a > b:
7 |             return euclid(a - b, b)
8 |         else:
9 |             return euclid(a, b - a)
```

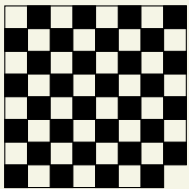
Prouver par induction que la fonction termine et revoit le pgcd de a et b.

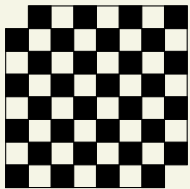
Invariants et problèmes insolubles

- On peut aussi utiliser les invariants pour démontrer qu'un problème n'a pas de solution :
- On trouve un invariant qui doit être satisfait par les itérations de tout hypothétique algorithme ;
- ... mais qui n'est pas satisfait par la configuration finale recherchée.

Exercice

Donner un algorithme pour couvrir un échiquier 8×8, dont certaines cases manquent, avec des dominos 2×1 (ou 1×2), dans les deux cas suivants :





Preuve automatique et indécidabilité

Peut-on écrire un programme qui prend un programme quelconque en entrée et décide s'il a une propriété donnée ?

Exemple

Écrire un programme qui décide si un programme quelconque s'arrête (sur une entrée donnée).

- Supposons qu'un tel programme existe : on l'appelle H ;
- Pour un programme P et une entrée i donnés, $H(P, i)$ vaut `vrai` si P s'arrête pour l'entrée i et `faux` sinon ;

Preuve automatique et indécidabilité : diagonalisation

- Soit le programme H', qui prend en entrée un programme P dont l'entrée est un programme, et défini par :

Programme $H'(P)$:

```
if H(P,P):
    while True: pass
else:
    print("coin !")
```

Preuve automatique et indécidabilité : diagonalisation

- Soit le programme H', qui prend en entrée un programme P dont l'entrée est un programme, et défini par :

Programme $H'(P)$:

```
if H(P,P):
    while True: pass
else:
    print("coin !")
```

- Que se passe-t-il pour l'exécution de H' pour le programme H' ?
  H' est bien un programme qui prend un programme en entrée
  - Si l'exécution se termine alors $H(H', H')$ est vrai... et on devrait aller dans le while qui ne s'arrête jamais ;
  - Si l'exécution ne se termine pas, alors $H(H', H')$ est faux... et on devrait s'arrêter en affichant `"coin !"`.

Preuve automatique et indécidabilité : diagonalisation

- Soit le programme H', qui prend en entrée un programme P dont l'entrée est un programme, et défini par :

Programme $H'(P)$:

```
if H(P,P):
    while True: pass
else:
    print("coin !")
```

- Que se passe-t-il pour l'exécution de H' pour le programme H' ?
  H' est bien un programme qui prend un programme en entrée
  - Si l'exécution se termine alors $H(H', H')$ est vrai... et on devrait aller dans le while qui ne s'arrête jamais ;
  - Si l'exécution ne se termine pas, alors $H(H', H')$ est faux... et on devrait s'arrêter en affichant `"coin !"`.
- Donc le programme H' ne peut pas exister !

Preuve automatique et indécidabilité

- Un problème algorithmique dont la réponse est vrai ou faux est appelé **problème de décision** ;
- Un problème de décision pour lequel il n'existe pas d'algorithme qui le résout est dit **indécidable** ;
- Le **Théorème de Rice** dit (informellement) que toute propriété non-triviale sur les programmes est indécidable.

Indécidabilité : le 10e problème de Hilbert

- Une **équation diophantienne** est une équation polynômiale à coefficients entiers (relatifs) et dont les solutions recherchées sont également entières.
- Par exemple :
  - $x^2 + y^2 = z^2$: il y a une infinité de solutions positives, les **triplets pythagoriciens** ;
  - $x^n + y^n = z^n$ pour $n > 2$: il n'y a aucune solution non-triviale (dernier théorème de Fermat) ;
  - $x^2 - ny^2 = 1$ pour n qui n'est pas un carré parfait. Si (x, y) est solution (il y en a une infinité) alors $\frac{x}{y}$ est une approximation rationnelle de $\sqrt{2}$.
- Le 10e problème de Hilbert est « Décider si une équation diophantienne (quelconque) a une solution » ;
- Ce problème est **indécidable**.

Indécidabilité : Problème de correspondance de Post (PCP)

- Le problème de correspondance de Post est un des problèmes indécidables les plus simples :
- On se donne n dominos de la forme $\begin{array}{|c|}\hline w_2 \\ \hline w_1 \\ \hline \end{array}$ où w_1 et w_2 sont des mots sur un alphabet fini donné (p. ex. $\{0,1\}$) ;
- On dispose d'un ensemble D d'autant de copies de ces dominos qu'on veut ;
- L'objectif est de savoir s'il existe une séquence de dominos pris dans D telle que la concaténation des parties supérieures est égale à la concaténation des parties inférieures ;
- Par exemple, pour $\begin{array}{|c|}\hline 01 \\ \hline 110 \\ \hline \end{array}$, $\begin{array}{|c|}\hline 1 \\ \hline 010 \\ \hline \end{array}$, $\begin{array}{|c|}\hline 10 \\ \hline 1 \\ \hline \end{array}$, et $\begin{array}{|c|}\hline 101 \\ \hline 01 \\ \hline \end{array}$ numérotés de 1 à 4, une correspondance est la séquence $3, 2, 1, 3, 4$:

$$\begin{array}{|c|}\hline 10 \\ \hline 1 \\ \hline \end{array}\begin{array}{|c|}\hline 1 \\ \hline 010 \\ \hline \end{array}\begin{array}{|c|}\hline 01 \\ \hline 110 \\ \hline \end{array}\begin{array}{|c|}\hline 10 \\ \hline 1 \\ \hline \end{array}\begin{array}{|c|}\hline 101 \\ \hline 01 \\ \hline \end{array}$$

- Ce problème est **indécidable**.

Indécidabilité : Machines à deux compteurs

- Une **machine déterministe à deux compteurs** (M2C) est un programme dans lequel il y a :
  - un état représentant le numéro de l'instruction courante et valant initialement 1 ;
  - deux compteurs C_1 et C_2 à valeurs initialement nulles ;
  - des instructions génériques de deux types (n_j, n_k sont des états = numéros d'instruction) :
    1. incrément : $C_i \leftarrow C_i + 1$; goto n_j
    2. décrément : if $C_i = 0$ goto n_j else $\{\ C_i \leftarrow C_i - 1$; goto $n_k\ \}$
  - Un état correspond a une instruction spéciale `halt` qui arrête la machine.

Exemple

1	$C_1 \leftarrow C_1 + 1$; goto 2
2	$C_2 \leftarrow C_2 + 1$; goto 3
3	if $C_1 = 0$ goto 4
	else $\{\ C_1 \leftarrow C_1 - 1\ \}$; goto 2
4	halt

1	$C_1 \leftarrow C_1 + 1$; goto 2
2	if $C_1 = 0$ goto 5
	else $\{\ C_1 \leftarrow C_1 - 1\ \}$; goto 3
3	$C_2 \leftarrow C_2 + 1$; goto 4
4	$C_2 \leftarrow C_2 + 1$; goto 2
5	if $C_1 = 0$ goto 2
	else $\{\ C_2 \leftarrow C_2 - 1\ \}$; goto 6
6	$C_1 \leftarrow C_1 + 1$; goto 7
7	$C_1 \leftarrow C_1 + 1$; goto 5
8	halt

- Savoir si la machine va s'arrêter ou si les compteurs vont rester bornés sont des problèmes indécidables.

Outline

Complexité

- Pour des problèmes décidable ou calculables, on veut comparer les **performances des algorithmes** qui les résolvent ;
- La **complexité d'un problème** est celle du meilleur algorithme qui le résoud ;
- Deux mesures principales :
  1. temps de calcul ;
  2. mémoire maximale utilisée.
- Ces mesures dépendent de l'instance du problème considéré :
  - on mesure en fonction de la **taille de l'entrée** ;
  - meilleur cas ou **pire cas** ;
  - cas moyen (espérance) ;
- Elles dépendent aussi du modèle de calcul considéré :
  - Informatique théorique : machines de Turing ;
    Par exemple ≪ le problème de décision du voyageur de commerce est NP-complet ≫
  - Analyse pratique : modèle RAM.
    Par exemple ≪ le tri fusion opère en $O(n \log_2 n)$ ≫

Machines de Turing

- Une machine de Turing est une machine qui contient :
  - un état représentant le numéro de l'instruction courante, et valant initialement 1 ;
  - une mémoire sous la forme d'un tableau infini M indexé par $\mathbb{Z}$
    - qui contient des symboles issus d'un alphabet fini A ;
    - A contient un symbole spécial $\square$ indiquant qu'une case est vide ;
    - une tête de lecture cur $\in \mathbb{Z}$ désigne la case courante $M[\text{cur}] \in A$ de la mémoire ;
    - M contient des données d'entrée : un mot sur $A \setminus \{\square\}$ commençant à la case 0.
  - des instructions d'un unique type (avec $a, b \in A$, $i \in \{-1, 0, 1\}$ et n_j un état) :

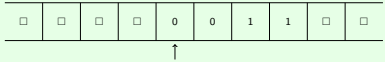
if	$M[\text{cur}] = a$	then $M[\text{cur}] \leftarrow b$; cur $\leftarrow$ cur $+ i$;	goto n_j
elsif	$M[\text{cur}] = b$	then ...	goto ...
elsif	$M[\text{cur}] = \ldots$	then ...	goto ...
elsif	...		

  - un état correspond à une instruction spéciale `accept` qui arrête la machine et répond ≪ oui ≫ ;
  - un état correspond à une instruction spéciale `reject` qui arrête la machine et répond ≪ non ≫ ;
  - la machine peut bloquer (*deadlock*) si la case courante ne correspond pas à l'instruction courante.

Machines de Turing : Exemple

Exemple

	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

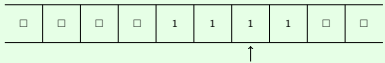
Exemple

	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			

Machines de Turing : Exemple

Exemple

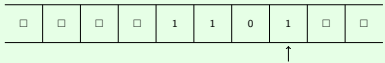
	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

Exemple

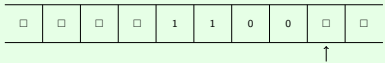
	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

Exemple

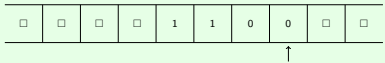
	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

Exemple

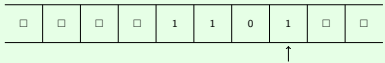
	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

Exemple

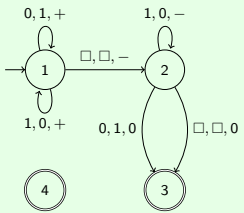
	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Machines de Turing : Exemple

Exemple

	read	write	dep	goto
1	0	1	+1	1
	1	0	+1	1
	□	□	-1	2
2	0	1	0	3
	1	0	-1	2
	□	□	0	3
3	accept			
4	reject			



Exercice

Écrire une machine de Turing sur l'alphabet $\{a, b\}$ qui décide si un mot est un palindrome. Par exemple abba ou ababa sont acceptés mais abbaa est rejeté.

Complexité d'une machine de Turing

Thèse de Church-Turing

Les machines de Turing peuvent simuler n'importe quel algorithme.

- La machine de Turing est un modèle très simple d'algorithme ;
- Un seul type d'instruction ;
- Un seul type de données (symbole) ;
- La taille de l'entrée est le nombre de cases initialement non vides ;
- Les notions de complexité sont simples et naturelles :
  1. Complexité temporelle : nombre d'instructions réalisées avant arrêt ;
  2. Complexité spatiale : nombre maximal de cases non vides simultanément.

Quelques classes de complexité des problèmes de décision

- PTIME (ou P) : les problèmes qu'on peut résoudre en temps polynomial ($O(n^k)$) ;
  p. ex. existence d'une solution dans un programme linéaire
- PSPACE : les problèmes qu'on peut résoudre en espace polynomial
  p. ex. décider si on peut passer d'un configuration à une autre dans *Rush Hour* ou *Sokoban*
- EXPTIME : les problèmes qu'on peut résoudre en temps exponentiel ($O(2^n)$) ;
  p. ex. déterminer si le 1er joueur gagne pour une position de Go (en fonction de la taille du plateau)
- EXPSPACE : les problèmes qu'on peut résoudre en espace exponentiel
  p. ex. déterminer si 2 expression régulières (simples) ont le même langage
  $(a^*b^* + c)^* = (a + b + c)^*$?

$$\text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE}$$

$$\text{PTIME} \subsetneq \text{EXPTIME}$$

La classe NP et le problème SAT

Le problème SAT

- **entrée** : une formule booléenne avec des variables propositionnelles.
  Par exemple :
  $$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$
- **sortie** : existe-t-il une valeur des variables telle que la formule est vraie ?

La classe NP et le problème SAT

Le problème SAT

- **entrée** : une formule booléenne avec des variables propositionnelles.
  Par exemple :
  $$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$
- **sortie** : existe-t-il une valeur des variables telle que la formule est vraie ?
  Dans l'exemple, oui car (entre autres) $x = 1, y = 1, z = 1$ fonctionne.

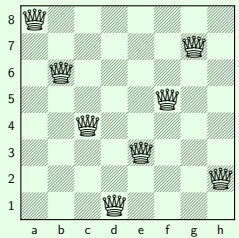
La classe NP et le problème SAT

Le problème SAT

- **entrée** : une formule booléenne avec des variables propositionnelles.
  Par exemple :
  $$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$
- **sortie** : existe-t-il une valeur des variables telle que la formule est vraie ?
  Dans l'exemple, oui car (entre autres) $x = 1, y = 1, z = 1$ fonctionne.

Exemple

Le problème des huit reines : Placer huit reines sur un échiquier telles qu'elles ne soient pas en prise.
Comment résoudre ce problème avec le problème SAT ?



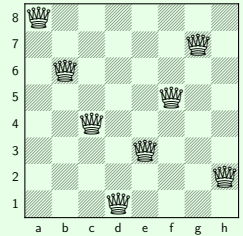
La classe NP et le problème SAT

Le problème SAT

- **entrée** : une formule booléenne avec des variables propositionnelles.
  Par exemple :
  $$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$
- **sortie** : existe-t-il une valeur des variables telle que la formule est vraie ?
  Dans l'exemple, oui car (entre autres) $x = 1, y = 1, z = 1$ fonctionne.

Exemple

Le problème des huit reines : Placer huit reines sur un échiquier telles qu'elles ne soient pas en prise.
Comment résoudre ce problème avec le problème SAT ?



$$\bigwedge_{i,j}\left(x_{ij} \Rightarrow \bigwedge_{k\neq 0}(\neg x_{i,j+k} \wedge \neg x_{i+k,j} \wedge \neg x_{i+k,j+k} \wedge \neg x_{i-k,j+k})\right) \wedge \bigwedge_{i} \bigvee_{j} x_{ij}$$

Complexité de SAT

- On peut résoudre le problème SAT en temps exponentiel ;
- On peut encoder une partie d'échecs ou de go avec SAT
  $\Rightarrow$ **mais** avec une formule de taille **exponentielle** !

Complexité de SAT

- On peut résoudre le problème SAT en temps exponentiel ;
- On peut encoder une partie d'échecs ou de go avec SAT
  $\Rightarrow$ **mais** avec une formule de taille **exponentielle** !
- Si on donne une valeur des variables, il est $\ll$ facile $\gg$ de **vérifier** que c'est une solution ou non ;
- De nombreux problèmes utiles ont cette propriété :
  - existence d'un cycle hamiltonien dans un graphe orienté ;
  - coloration de graphes ;
  - problème du sac-à-dos (*Knapsack*) ;
  - problème des cartons (*Bin packing*) ;
  - programmation linéaire en nombre entiers (ILP).
  - ...

Complexité de SAT

- On peut résoudre le problème SAT en temps exponentiel ;
- On peut encoder une partie d'échecs ou de go avec SAT
  $\Rightarrow$ **mais** avec une formule de taille **exponentielle** !
- Si on donne une valeur des variables, il est $\ll$ facile $\gg$ de **vérifier** que c'est une solution ou non ;
- De nombreux problèmes utiles ont cette propriété :
  - existence d'un cycle hamiltonien dans un graphe orienté ;
  - coloration de graphes ;
  - problème du sac-à-dos (*Knapsack*) ;
  - problème des cartons (*Bin packing*) ;
  - programmation linéaire en nombre entiers (ILP).
  - ...
- Pour les étudier : abstraire la recherche de la solution
  - Permettre à la machine de $\ll$ deviner $\gg$ une solution avant de la vérifier ;
  - Permettre à la machine de se dupliquer et vérifier plusieurs candidats en même temps
    $\rightarrow$ Machine **non-déterministe**

Machines de Turing non-déterministes

- Les instructions des machines de Turing sont de la forme :

$$\begin{array}{rlll} \text{if} & M[\text{cur}] = a & \text{then } M[\text{cur}] \leftarrow b\,;\ \text{cur} \leftarrow \text{cur} + 1\,; & \text{goto } 10 \\ \text{if} & M[\text{cur}] = b & \text{then } M[\text{cur}] \leftarrow c\,;\ \text{cur} \leftarrow \text{cur} - 1\,; & \text{goto } 20 \\ \text{elsif} & M[\text{cur}] = \ldots & \text{then } \ldots & \text{goto } \ldots \\ \text{elsif} & \ldots & & \end{array}$$

Machines de Turing non-déterministes

- Les instructions des machines de Turing sont de la forme :

$$\begin{array}{lll} \text{if} & M[\text{cur}] = a & \text{then } M[\text{cur}] \leftarrow b\,;\ \text{cur} \leftarrow \text{cur} + 1\,;\quad \text{goto } 10 \\ \text{if} & M[\text{cur}] = b & \text{then } M[\text{cur}] \leftarrow c\,;\ \text{cur} \leftarrow \text{cur} - 1\,;\quad \text{goto } 20 \\ \text{elsif} & M[\text{cur}] = \ldots & \text{then } \ldots \qquad\qquad\qquad\quad \text{goto } \ldots \\ \text{elsif} & \ldots & \end{array}$$

- On autorise maintenant plusieurs if à porter sur le **même symbole** :

$$\begin{array}{lll} \text{if} & M[\text{cur}] = \mathbf{a} & \text{then } M[\text{cur}] \leftarrow b\,;\ \text{cur} \leftarrow \text{cur} + 1\,;\quad \text{goto } 10 \\ \text{elsif} & M[\text{cur}] = \mathbf{a} & \text{then } M[\text{cur}] \leftarrow c\,;\ \text{cur} \leftarrow \text{cur} - 1\,;\quad \text{goto } 20 \\ \text{elsif} & M[\text{cur}] = \ldots & \text{then } \ldots \qquad\qquad\qquad\quad \text{goto } \ldots \\ \text{elsif} & \ldots & \end{array}$$

Machines de Turing non-déterministes

- Les instructions des machines de Turing sont de la forme :

if	$M[\text{cur}] = a$	then $M[\text{cur}] \leftarrow b$; $\text{cur} \leftarrow \text{cur} + 1$;	goto 10
if	$M[\text{cur}] = b$	then $M[\text{cur}] \leftarrow c$; $\text{cur} \leftarrow \text{cur} - 1$;	goto 20
elsif	$M[\text{cur}] = \ldots$	then ...	goto ...
elsif	...		

- On autorise maintenant plusieurs if à porter sur le **même symbole** :

if	$M[\text{cur}] = \mathbf{a}$	then $M[\text{cur}] \leftarrow b$; $\text{cur} \leftarrow \text{cur} + 1$;	goto 10
elsif	$M[\text{cur}] = \mathbf{a}$	then $M[\text{cur}] \leftarrow c$; $\text{cur} \leftarrow \text{cur} - 1$;	goto 20
elsif	$M[\text{cur}] = \ldots$	then ...	goto ...
elsif	...		

- Dans ce cas, la machine se duplique et chaque copie exécute l'une des possibilités !

Machines de Turing non-déterministes

- Les instructions des machines de Turing sont de la forme :

$$\begin{array}{rlll} \text{if} & M[\text{cur}] = a & \text{then } M[\text{cur}] \leftarrow b\,;\ \text{cur} \leftarrow \text{cur} + 1\,; & \text{goto } 10 \\ \text{if} & M[\text{cur}] = b & \text{then } M[\text{cur}] \leftarrow c\,;\ \text{cur} \leftarrow \text{cur} - 1\,; & \text{goto } 20 \\ \text{elsif} & M[\text{cur}] = \ldots & \text{then } \ldots & \text{goto } \ldots \\ \text{elsif} & \ldots & & \end{array}$$

- On autorise maintenant plusieurs if à porter sur le **même symbole** :

$$\begin{array}{rlll} \text{if} & M[\text{cur}] = \mathbf{a} & \text{then } M[\text{cur}] \leftarrow b\,;\ \text{cur} \leftarrow \text{cur} + 1\,; & \text{goto } 10 \\ \text{elsif} & M[\text{cur}] = \mathbf{a} & \text{then } M[\text{cur}] \leftarrow c\,;\ \text{cur} \leftarrow \text{cur} - 1\,; & \text{goto } 20 \\ \text{elsif} & M[\text{cur}] = \ldots & \text{then } \ldots & \text{goto } \ldots \\ \text{elsif} & \ldots & & \end{array}$$

- Dans ce cas, la machine se duplique et chaque copie exécute l'une des possibilités !
- La machine accepte si l'**une des copies** atteint l'état `accept` ;
- La machine rejette si **toutes les copies** atteignent l'état `reject`.

Machines de Turing non-déterministes

- Les instructions des machines de Turing sont de la forme :

if	$M[\text{cur}] = a$	then $M[\text{cur}] \leftarrow b$; cur $\leftarrow$ cur $+ 1$;	goto 10
if	$M[\text{cur}] = b$	then $M[\text{cur}] \leftarrow c$; cur $\leftarrow$ cur $- 1$;	goto 20
elsif	$M[\text{cur}] = \ldots$	then ...	goto ...
elsif	...		

- On autorise maintenant plusieurs if à porter sur le **même symbole** :

if	$M[\text{cur}] =$ **a**	then $M[\text{cur}] \leftarrow b$; cur $\leftarrow$ cur $+ 1$;	goto 10
elsif	$M[\text{cur}] =$ **a**	then $M[\text{cur}] \leftarrow c$; cur $\leftarrow$ cur $- 1$;	goto 20
elsif	$M[\text{cur}] = \ldots$	then ...	goto ...
elsif	...		

- Dans ce cas, la machine se duplique et chaque copie exécute l'une des possibilités !
- La machine accepte si l'**une des copies** atteint l'état `accept` ;
- La machine rejette si **toutes les copies** atteigent l'état `reject`.

Exemple

Écrire une machine de Turing non-déterministe qui étant donnée une séquence de 0, 1 et x décide s'il existe une valeur de $x \in \{0, 1\}$ telle que la séquence est un palindrome sur $\{0, 1\}$.
Par exemple, la réponse est oui pour $xx0x1$ et non pour $1x0$.

La classe de complexité NP

- Une machine non-déterministe résout SAT en temps polynomial ;
- On définit les classes de complexité pour les **machines non-déterministe** comme précédemment.
- NPTIME (ou NP) : les problèmes qu'on peut résoudre en temps polynomial ;
  p. ex. SAT, knapsack, ILP, etc.
- Mais aussi NPSPACE, NEXPTIME, NEXPSPACE, etc.
- Le **Théorème de Savitch** implique que PSPACE = NPSPACE et EXPSPACE = NEXPSPACE.

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$$

- Une question ouverte importante est P=NP ?
  Un des 7 défis du millénaire du *Clay Institute of Mathematics*

Bornes inférieures et complétude

- Soit un problème P tel que :
  1. on peut transformer toute instance d'un problème quelconque de la classe de complexité $\mathcal{C}$ en une instance de P
  2. la transformation n'est pas trop coûteuse (typiquement polynomiale, ou logarithmique pour PTIME)
- P est **$\mathcal{C}$-difficile** ;
- Si de plus $P \in \mathcal{C}$ alors P est **$\mathcal{C}$-complet** ;
- Les problèmes $\mathcal{C}$-complets sont les plus $\ll$ difficiles $\gg$ de la classe $\mathcal{C}$;
- Le **Théorème de Cook-Levin** dit que SAT est NP-difficile (et donc NP-complet).

Exercice

Le problème de décision ILP est : étant donnée une intersection de demi-espaces, contient-elle un point entier ?
Par exemple : $2x_1 - x_2 \leq 0, x_1 \geq 0, x_2 \geq 0$ contient le point $(0, 0)$ (entre autres)

Montrer que ILP est NP-difficile.
3SAT (formule en CNF et au plus 3 littéraux par clause) est déjà NP-complet.

Modèles de type RAM

- Les opérations arithmétiques ont une complexité polynomiale pour une machine de Turing ;
- Idem pour l'accès à une case particulière de la mémoire ;
- Pour l'analyse pratique des algorithmes, on considère un modèle plus réaliste :
  - opérations arithmétiques simples, indexage (*Random access memory*), etc. en temps constant
  - accès direct à des variables (registres)
  - contrôle du flot d'exécution (branchements)
  - types de données classiques à précision infinie (en général)
- On peut adapter le modèle en fonction de ce qu'on cherche à mesurer.

Complexité asymptotique

- Comment évolue la complexité quand la taille de l'entrée tend vers l'infini ?
- On met en rapport cette complexité avec des fonctions plus simples (notations de Bachmann-Landau) :
  - **Domination** :

$$f(n) = O(g(n)) \text{ si } \exists c, \exists N \text{ t.q. } \forall n > N, f(n) \leq cg(n)$$

  - et la relation inverse (Knuth) :

$$f(n) = \Omega(g(n)) \text{ si } g(n) = O(f(n))$$

  - **Encadrement** :

$$f(n) = \Theta(g(n)) \text{ si } f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n))$$

Complexité asymptotique

- Comment évolue la complexité quand la taille de l'entrée tend vers l'infini ?
- On met en rapport cette complexité avec des fonctions plus simples (notations de Bachmann-Landau) :
  - **Domination** :

$$f(n) = O(g(n)) \text{ si } \exists c, \exists N \text{ t.q. } \forall n > N, f(n) \leq cg(n)$$

    - et la relation inverse (Knuth) :

$$f(n) = \Omega(g(n)) \text{ si } g(n) = O(f(n))$$

    - **Encadrement** :

$$f(n) = \Theta(g(n)) \text{ si } f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n))$$

  - **Négligeabilité** :

$$f(n) = o(g(n)) \text{ si } \forall c, \exists N \text{ t. q. } \forall n > N, f(n) \leq cg(n)$$

    - et symétriquement (Knuth) :

$$f(n) = \omega(g(n)) \text{ si } g(n) = o(f(n))$$

    - **Même ordre** :

$$f(n) \sim g(n)) \text{ si } \lim_{n \to +\infty} \frac{f(n)}{g(n)} = 1$$

Calcul de la complexité pire cas : exemple

Exemple

```
1
2   def sort(A, n):
3       for i = 1 to n − 1:          # n - 1 fois
4           insert(i, A)
5
6   def insert(x, A):
7       key = A[x]                   # c= + c[]
8
9       j = x − 1                    # c= + c+
10      while j ≥ 0 and A[j] > key # n' + 1 fois 2c> + c[] + cand
11          A[j + 1] = A[j]          # n' fois 2[] + c+ + c=
12          j = j − 1                # n' fois c+ + c=
13
14      A[j+1] = key                 # c= + c[] + c+
```

- Pour `insert`, on a donc : $f_x(n) = 3c_= + 3c_{[]} + c_+ + 2c_> + c_{\text{and}} + n'(2c_> + 3c_{[]} + 3c_+ + c_{\text{and}} + 2c_=)$
- Le pire cas est un tableau trié dans l'ordre décroissant : le while va toujours jusqu'à $j = -1$
  $\Rightarrow$ le nombre d'itérations n' est égal à x
- Donc $f_x(n) = k + k'x$ avec $x \leq n-1$ donc $f_x(n) = \Theta(x)$ et aussi $f_x(n) = \Theta(n)$
- Et pour `sort`, $g(n) = \sum_{x=1}^{n-1} f_x(n) = (n-1)k + k' \sum_{x=1}^{n-1} x = (n-1)k + k' \frac{(n-1)n}{2}$ et donc $g(n) = \Theta(n^2)$.

Calcul de la complexité pire cas : exercice

Exercice

On veut évaluer en un point `x` un polynôme de degré `n` en donné par la liste `A` de ses coefficients. On considère deux approches :

```
1  def naive(A, x, n):
2      s = A[0]
3      for i = 1 to n:
4          p = A[i]
5          for j = 1 to i:
6              p = p * x
7
8          s = s + p
9
10     return s
```

```
1  def horner(A, x, n):
2      s = 0
3      for i = n downto 0:
4          s = s * x + A[i]
5
6      return s
```

Calculer la complexité des deux approches.

Calcul de la complexité pire cas : exercice

Exercice

On veut évaluer en un point `x` un polynôme de degré `n` en donné par la liste `A` de ses coefficients. On considère deux approches :

```
1  def naive(A, x, n):
2      s = A[0]
3      for i = 1 to n:
4          p = A[i]
5          for j = 1 to i:
6              p = p * x
7
8          s = s + p
9
10     return s
```

```
1  def horner(A, x, n):
2      s = 0
3      for i = n downto 0:
4          s = s * x + A[i]
5
6      return s
```

Calculer la complexité des deux approches.

Exercice

Quelle est la complexité du tri par insertion en fonction du nombre d'inversions I (le nombre de couples (i, j) tels que $i < j$ et $A[i] > A[j]$) et de n ?

Complexité pire cas des algorithmes récursifs

- Pour la complexité, la récursivité conduit à des récurrences ;
- On peut les résoudre parfois directement :
  1. récurrences linéaires à coefficients constants ;
  2. deviner une expression et la prouver ;
  3. théorèmes dédiés : *Master Theorem* (voir *divide and conquer*), récurrences d'Akra-Bazzi.

Complexité pire cas des algorithmes récursifs

Exemple

```
1 | def fact(n):
2 |     if n == 0:
3 |         return 1
4 |     else:
5 |         return n * fact(n-1)
```

- $T(n) = T(n-1) + 1$ et $T(0) = 1$
- donc $T(n) = n + 1 = \Theta(n)$

Complexité pire cas des algorithmes récursifs

Exemple

```
1 | def fib(n):
2 |     if n == 0 or n == 1:
3 |         return 1
4 |     else:
5 |         return fib(n-1) + fib(n-2)
```

- $T(n+2) = T(n+1) + T(n) + 1 \geq T(n+1) + T(n)$ et $T(0) = 1, T(1) = 2$
- Polynôme caractéristique : $x^2 - x - 1$. Racines $\varphi_1 = \frac{1+\sqrt{5}}{2}$ et $\varphi_2 = \frac{1-\sqrt{5}}{2}$
- $T(n) \geq A\varphi_1^n + B\varphi_2^n$ et avec les conditions initiales $A + B = 1$ et $B - A = \frac{3}{\sqrt{5}}$.
- $T(n) \geq (\frac{1}{2} + \frac{3\sqrt{5}}{10})\varphi_1^n + (\frac{1}{2} - \frac{3\sqrt{5}}{10})\varphi_2^n$ et $T(n) = \Omega((\frac{3}{2})^n)$

Complexité pire cas des algorithmes récursifs

Exemple

```
1 | def fib(n):
2 |     if n == 0 or n == 1:
3 |         return 1
4 |     else:
5 |         return fib(n-1) + fib(n-2)
```

- $T(n+2) = T(n+1) + T(n) + 1 \geq T(n+1) + T(n)$ et $T(0) = 1, T(1) = 2$
- Polynôme caractéristique : $x^2 - x - 1$. Racines $\varphi_1 = \frac{1+\sqrt{5}}{2}$ et $\varphi_2 = \frac{1-\sqrt{5}}{2}$
- $T(n) \geq A\varphi_1^n + B\varphi_2^n$ et avec les conditions initiales $A + B = 1$ et $B - A = \frac{3}{\sqrt{5}}$.
- $T(n) \geq (\frac{1}{2} + \frac{3\sqrt{5}}{10})\varphi_1^n + (\frac{1}{2} - \frac{3\sqrt{5}}{10})\varphi_2^n$ et $T(n) = \Omega((\frac{3}{2})^n)$

Exercice

Montrer que la complexité de `fib` est $O(2^n)$.

Complexité moyenne

Exemple

```
1  # on suppose n > 0 et les éléments de A tous distincts
2  def maximum(A, n):
3      r = A[1]
4      for i = 2 to n - 1:
5          if A[i] > r:
6              r = A[i]
7
8      return r
```

Combien de fois le `if` est-il pris ?

- Au **pire cas**, la liste est triée dans l'ordre ascendant, et le if est pris à chaque fois ($n-1$ fois) ;
- Mais en moyenne ?

Complexité moyenne

Exemple

```
1  # on suppose n > 0 et les éléments de A tous distincts
2  def maximum(A, n):
3      r = A[1]
4      for i = 2 to n − 1:
5          if A[i] > r:
6              r = A[i]
7
8      return r
```

Combien de fois le `if` est-il pris ?

- Au **pire cas**, la liste est triée dans l'ordre ascendant, et le if est pris à chaque fois ($n-1$ fois) ;
- Mais en moyenne ?
- Pour calculer la **complexité moyenne**, il faut une **distribution de probabilité** sur les données d'entrée
  $\Rightarrow$ Sans information supplémentaire, on suppose ici que chaque permutation des éléments de A est **équiprobable**
- On calcule, **l'espérance de la complexité** selon cette distribution.

Complexité moyenne : variables aléatoires indicatrices

- Soit X_i la **variable aléatoire indicatrice** de l'événement $E_i = \ll$ le `if` est pris à l'itération $i \gg$;
- Le nombre total de prises est :

$$X = \sum_{i=2}^{n-1} X_i$$

- L'espérance de X est donc (linéarité de l'espérance) :

$$E(X) = \sum_{i=2}^{n-1} E(X_i)$$

- **Par construction**, $E(X_i)$ est la probabilité que que l'événement soit vrai :

$$E(X_i) = P(E_i)X_i(E_i) + P(\neg E_i)X_i(\neg E_i) = P(E_i) \times 1 + P(\neg E_i) \times 0 = P(E_i)$$

- Donc :

$$E(X) = \sum_{i=2}^{n-1} P(E_i)$$

- Reste à calculer $P(E_i)$.

Complexité moyenne

- Comme tous les ordres pour A sont équiprobables :
  - il y a autant de séquences avec $A[1] > A[0]$ qu'avec $A[1] \leq A[0]$ donc $P(E_2) = \frac{1}{2}$
  - on peut découper les séquences possibles en trois groupes de même taille :
    1. dans les 3 premiers chiffres, le 1er est le plus grand : E_3 est faux
    2. dans les 3 premiers chiffres, le 2e est le plus grand : E_3 est faux
    3. dans les 3 premiers chiffres, le 3e est le plus grand : E_3 est vrai donc $P(E_3) = \frac{1}{3}$
  - plus généralement, $P(E_i) = \frac{1}{i}$
- Et finalement $E(X) = \sum_{i=2}^{n-1} \frac{1}{i}$ et donc :

$$E(X) = \ln(n-1) + O(1)$$

.

Complexité moyenne : Exercices

Exercice

On réalise une permutation aléatoire uniforme de la la liste $[1, 2, \ldots, n]$? Quel est l'espérance du nombre de **points fixes** ? (le nombre d'éléments qui sont toujours à la même place).

Complexité moyenne : Exercices

Exercice

On réalise une permutation aléatoire uniforme de la la liste $[1, 2, \ldots, n]$? Quel est l'espérance du nombre de **points fixes** ? (le nombre d'éléments qui sont toujours à la même place).

Exercice

On a vu que la complexité du tri par insertion dépend fortement du nombre d'**inversions** dans le tableau A : le nombre de couples (i, j) tels que $i < j$ et $A[i] > A[j]$.

Quel est l'espérance du nombre d'inversions si on suppose que le tableau A est une permutation aléatoire uniforme de l'ensemble $[1, 2, \ldots, n]$?

Complexité moyenne : Exercices

Exercice

On réalise une permutation aléatoire uniforme de la la liste $[1, 2, \ldots, n]$? Quel est l'espérance du nombre de **points fixes** ? (le nombre d'éléments qui sont toujours à la même place).

Exercice

On a vu que la complexité du tri par insertion dépend fortement du nombre d'**inversions** dans le tableau A : le nombre de couples (i, j) tels que $i < j$ et $A[i] > A[j]$.

Quel est l'espérance du nombre d'inversions si on suppose que le tableau A est une permutation aléatoire uniforme de l'ensemble $[1, 2, \ldots, n]$?

Exercice

1. Écrire un algorithme pour décider si un entier positif x est dans une liste d'entiers positifs de longueur n non triée ;
2. Quelles sont les complexités au meilleur cas, au pire cas et en moyenne de cette algorithme ?
   (Pour la moyenne, on suppose x borné par une valeur $N \geq n$)

Algorithmes probabilistes

- Pour la complexité, le cas moyen est souvent plus favorable que le pire cas ;
- Pour ne pas tomber systématiquement sur des cas défavorables, on peut **mélanger aléatoirement** les données

Exemple

```
1 | # on suppose n > 0
2 | def maximum_alea(A, n):
3 |       permutation_aleatoire (A) # permute les données de façon aléatoire
4 |       r = A[1]
5 |       for i = 2 to n - 1:
6 |           if A[i] > r:
7 |               r = A[i]
8 |
9 |       return r
```

Algorithmes probabilistes

- Un **algorithme probabiliste** est un algorithme qui utilise des instructions `random` ou similaires qui fournsissent leur résultat en suivant une certaine **distribution de probabilité** ;
- L'exécution d'un algorithme probabiliste est caractérisée par son **temps d'exécution attendu**
- La complexité attendue au pire cas est l'espérance de la complexité quand on résoud les **choix non-probabilistes** de la façon la plus défavorable ;
- L'espérance du nombre de fois où le `if` est pris au pire cas dans `maximum_alea` est $O(\ln n)$ **quelles que soient les données d'entrée**
  même si elles sont fournies par une entité adverse

Algorithmes probabilistes : Monte-Carlo

- Les algorithmes de décision de type **Monte-Carlo** fournissent toujours une réponse mais peuvent se tromper ;

Exemple

```
1 |  def fermat_prime(x, n):
2 |      for k = 1 to n:
3 |          x = random_int(2,n-2) # un nombre entier entre 2 et n -2
4 |          if x^{n-1} ≢ 1(mod n):
5 |              return false
6 |
7 |      return true # seulement probablement vrai !
```

- Pour les problèmes d'optimisation, ils donnent une réponse approchée (typiquement avec une garantie probabiliste)

Exemple

```
1 |  def eval_pi (n): # estimer la valeur de π
2 |      r = 0
3 |      for k = 1 to n:
4 |          x = random(0,1) # un nombre réel entre 0 et 1
5 |          y = random(0,1)
6 |          if x^2 + y^2 ≤ 1:
7 |              r = r + 1
8 |
9 |      return 4*r/n
```

Algorithmes probabilistes : Las Vegas

- Les algorithmes de type **Las Vegas** peuvent échouer à trouver une solution. Mais s'ils en renvoient une elle est correcte.
- L'**espérance** de leur temps d'exécution est finie ;

Exemple

```
1  def random_queen(L, i, n):
2      if i == n:
3          return L
4      else:
5          M = []
6          for j = 1 to n - 1 :
7              if not aligned(L, i, j):
8                  M = M ++ [j]
9
10         j = random_int(0, len(M) - 1)
11         random_queen(L++[(i,j)], i + 1, n)
12
13     return DO_NOT_KNOW # failed
```

Classes de complexité des algorithmes probabilistes

- On considère des **machines de Turing probabilistes** ;
- **RP** (*randomized polynomial time*) est la classe des problèmes de décision résolu par un algorithme probabiliste tel que :
  1. L'algorithme se termine en temps polynomial ;
  2. Il répond ≪ non ≫ quand la réponse est non ;
  3. Il répond ≪ oui ≫ avec probabilité > 0.5 quand la réponse est oui.
- **co-RP** : idem en échangeant ≪ oui ≫ et ≪ non ≫
- **BPP** (*bounded-error probabilistic polynomial time*) : résolu par un algorithme probabiliste tel que :
  1. L'algorithme se termine en temps polynomial ;
  2. Que la réponse soit ≪ oui ≫ ou ≪ non ≫, il se trompe avec probabilité $< \frac{1}{3}$.
- **ZPP** (*zero-error probabilistic polynomial time*) : résolu par un algorithme probabiliste tel que :
  1. S'il répond ≪ oui ≫ ou ≪ non ≫ sa réponse est correcte ;
  2. l'**espérance** de son temps d'exécution est polynomiale
     ou il peut répondre ≪ je ne sais pas ≫ mais son temps d'exécution est toujours polynomial.
- On a **ZPP** $\subseteq$ **RP** $\cap$ **co-RP** $\subseteq$ **BPP** ;
- Et **P** $\subseteq$ **BPP** (l'inclusion dans l'autre sens est ouverte).

Algorithmes probabilistes : Exercices

Exercice

On considère la function suivante :

```
1 | def permutation_aleatoire (A, n):
2 |     for i = 0 to n - 1:
3 |         swap(A[i], A[random_int(i,n-1)])
```

Une k-permutation de A est une séquence qui contient k éléments de A. Si la taille de A est n alors une n-permutation est une permutation.

1. Démontrer l'invariant suivant de la boucle pour : *après l'itération k, les k premiers éléments de A contiennent chaque k-permutation de A avec probabilité* $\frac{(n-k)!}{n!}$.
2. Quelle est la complexité de l'algorithme ?

Algorithmes probabilistes : Exercices

Exercice

On considère la function suivante :

```
1 | def permutation_aleatoire (A, n):
2 |     for i = 0 to n - 1:
3 |         swap(A[i], A[random_int(i,n-1)])
```

Une k-permutation de A est une séquence qui contient k éléments de A. Si la taille de A est n alors une n-permutation est une permutation.

1. Démontrer l'invariant suivant de la boucle pour : *après l'itération k, les k premiers éléments de A contiennent chaque k-permutation de A avec probabilité* $\frac{(n-k)!}{n!}$.
2. Quelle est la complexité de l'algorithme ?

Exercice

On dispose d'une fonction `biased_random()` qui renvoit 1 avec une certaine probabilité p inconnue, et 0 sinon.

1. Écrire un algorithme qui renvoit 1 avec probabilité 0.5 et 0 avec probabilité 0.5.
2. Donner l'espérance de son temps d'exécution.

Outline

Plus simple, plus petit

- Pour résoudre des problèmes compliqués, **décomposer** ;
- Pour obtenir les sous-problèmes :
  1. Relaxation = supprimer des contraintes :

Définition (Relaxation)

P' est une **relaxation** de P si toute solution de P est une solution de P'. On note $P \Rightarrow P'$.

Pour un problème de minimisation, on demande aussi que l'infimum de l'objectif sur les solutions de P' ne soit pas plus grand que celui sur les solutions de P.

  2. Diviser pour régner
     - ne chercher qu'une partie de la solution
     - réduire la taille de l'entrée

Outline

Énumeration exhaustive

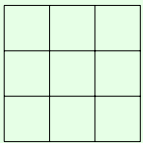
1. On considère une **relaxation** P' du problème P ;
2. P' doit avoir un ensemble **fini** de solutions ;
3. On énumère ces solutions jusqu'à trouver une solution de P ;
4. S'il n'y en a pas alors P n'a pas de solution.

Énumeration exhaustive

1. On considère une **relaxation** P' du problème P ;
2. P' doit avoir un ensemble **fini** de solutions ;
3. On énumère ces solutions jusqu'à trouver une solution de P ;
4. S'il n'y en a pas alors P n'a pas de solution.

Exemple

Carré magique. Pour $n \in \mathbb{N}_{>0}$, placer une fois et une seule chaque nombre de 1 à n^2 dans une grille carrée de taille de côté n de façon à ce que la somme de chaque ligne, chaque colonne, et chaque grande diagonale soit la même.



Énumeration exhaustive

1. On considère une **relaxation** P' du problème P ;
2. P' doit avoir un ensemble **fini** de solutions ;
3. On énumère ces solutions jusqu'à trouver une solution de P ;
4. S'il n'y en a pas alors P n'a pas de solution.

Exemple

Carré magique. Pour $n \in \mathbb{N}_{>0}$, placer une fois et une seule chaque nombre de 1 à n^2 dans une grille carrée de taille de côté n de façon à ce que la somme de chaque ligne, chaque colonne, et chaque grande diagonale soit la même.

7	8	9
4	5	6
1	2	3

Relaxation : Pour $n \in \mathbb{N}_{>0}$, placer une fois et une seule chaque nombre de 1 à n^2 dans une grille carrée de taille de côté n.

Énumeration exhaustive

1. On considère une **relaxation** P' du problème P ;
2. P' doit avoir un ensemble **fini** de solutions ;
3. On énumère ces solutions jusqu'à trouver une solution de P ;
4. S'il n'y en a pas alors P n'a pas de solution.

Exemple

Carré magique. Pour $n \in \mathbb{N}_{>0}$, placer une fois et une seule chaque nombre de 1 à n^2 dans une grille carrée de taille de côté n de façon à ce que la somme de chaque ligne, chaque colonne, et chaque grande diagonale soit la même.

8	1	6
3	5	7
4	9	2

Relaxation : Pour $n \in \mathbb{N}_{>0}$, placer une fois et une seule chaque nombre de 1 à n^2 dans une grille carrée de taille de côté n.

Énumeration exhaustive

- En général, P' a beaucoup plus de solutions que P ;
- On peut parfois utiliser des propriétés du problème pour définir un P' assez contraint ;
  la somme vaut forcément $\frac{n(n^2+1)}{2}$ et pour $n = 3, 5$ est forcément au milieu
- Les **symmétries** sont souvent utiles.
  ici rotations et réflexions

Exemple

```
1 | def carre(C, T, k n):
2 |     if k == n^2 and test\_carre(C, n):
3 |         return C
4 |     for i = 1 to n^2:
5 |         if not T[i]:
6 |             T[i] = true
7 |             C[⌊k/n⌋, k mod n] = i
8 |             carre(C, T, k+1, n)
9 |             T[i] = false
```

Énumeration exhaustive

Exercice

Peut-on résoudre les problèmes suivants par énumération exhaustive ? Si oui, avec quelle relaxation ?

1. les huit reines
2. SAT
3. programmation linéaire (pas ILP) : inf d'une expression linéaire sur un polyèdre convexe (borné pour simplifier).
4. problème de correspondance de Post

Outline

Backtracking

- On **décompose** le problème P en deux :
  - un problème Q pour lequel on peut énumérer les solutions
  - un problème P' incorporant la solution de Q
- P' est typiquement une variante plus simple ou plus contrainte de P
- On le résoud de la même façon
- Par exemple :
  - 8 reines : placer une reine puis résoudre le problème avec une reine déjà placée ;
  - carré magique : remplir une case puis remplir les autres étant donnée celle-ci ;
  - SAT : affecter une variable, puis satisfaire la formule résultant ;
  - chemin hamiltonien : choisir une sommet du chemin puis trouver une chemin depuis celui-ci :
- Chaque solution de Q donne lieu à un sous-problème ;
- En considérant ce lien, les solutions pour tous ces problèmes forment un **arbre** ;
- On explore cet arbre en **profondeur** d'abord.

Bactracking : Exemple

Exemple

```
1  def aligned(L, i, j):
2      for (k,l) in L:
3          if k == i or l == j or abs(k - i) == abs(l - j):
4              return True
5
6      return False
7
8  def queen(L, i, n):
9      if i == n:
10         return L
11     else:
12         for j = 1 to n - 1 :
13             if not aligned(L, i, j):
14                 L2 = L # copie
15                 L2 = L2 ++ [(i, j)]
16                 queen(L2, i + 1, n)
17
18     return []
```

Bactracking : Exercices

Exercice

Trouver des valeurs de x, y, z et w pour satisfaire cette formule de 3-SAT par *backtracking* :

$$(x \vee z \vee w) \wedge (x \vee y \vee \neg w) \wedge (\neg x \vee \neg y \vee w) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg w) \wedge \neg z$$

Bactracking : Exercices

Exercice

Trouver des valeurs de x, y, z et w pour satisfaire cette formule de 3-SAT par *backtracking* :

$$(x \vee z \vee w) \wedge (x \vee y \vee \neg w) \wedge (\neg x \vee \neg y \vee w) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg w) \wedge \neg z$$

Exercice

Le voyage du cavalier. On considère un échiquier de taille n avec un cavalier sur la case en haut à gauche. On rappelle que le cavalier se déplace de 2 cases dans une direction puis 1 case dans une direction orthogonale.

Écrire un algorithme pour trouver dans quel ordre le cavalier doit visiter les cases de l'échiquier pour les voir toutes exactement une fois.

Outline

Diviser pour régner (Divide & Conquer)

- Principe général
  1. Diviser en sous-problèmes ;
  2. Résoudre les sous-problèmes ;
  3. Combiner les solutions.
- En général, on parle de ≪ diviser pour régner ≫, quand on divise en (au moins deux) problèmes **indépendants identiques** au problème initial mais **plus petits** ;
- Donne naturellement des algorithmes **récursifs**.

Diviser pour régner

Exemple

Tri fusion (merge sort).

1. On divise le tableau en deux en coupant au milieu ;
2. On trie chaque sous-tableaux ;
3. On combine les deux sous-tableaux.

```
1  def merge_sort(A, d, f):
2      if f − d > 1:
3          m = ⌊(f+d)/2⌋
4          merge_sort(A, 0, m )
5          merge_sort(A, m, n)
6          B = A # copie A dans B
7
8          i = 0
9          j = m
10         k = 0
11         while i < m or j < n:
12             if i < m and (j == n or B[i] < B[j]):
13                 A[k] = B[i]
14                 i = i + 1
15             else:
16                 A[k] = B[j]
17                 j = j + 1
18
19             k = k + 1
```

Analyse des algorithmes de type ≪ diviser pour régner ≫

- On obtient naturellement un algorithme récursif
- Sa complexité est donnée par une récurrence

Exemple

Complexité du tri fusion : $T(n) = 2T(n/2) + \Theta(n)$
En fait, $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$

- Le **Master theorem** permet de résoudre (certaintes de) ces récurrences.

Master theorem

Théorème (Master theorem)

Soient $a, a', a'', b, k \in \mathbb{N}$, $a = a' + a''$, $b > 1$. *Si*

$$T(n) = a'T(\lceil n/b \rceil) + a''T(\lfloor n/b \rfloor) + \Theta(n^k)$$

On écrit simplement $T(n) = aT(n/b) + \Theta(n^k)$
Alors

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log_b n) & si\ a = b^k \\ \Theta(n^{\log_b a}) & si\ a > b^k \end{cases}$$

Exemple

Pour le tri fusion, $a = b = 2$ et $k = 1$ donc $T(n) = \Theta(n \log_2 n)$.

Diviser pour régner : exercices

Exercice

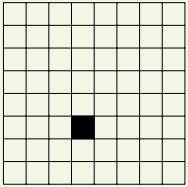
Recherche dichotomique. Proposer un algorithme pour la recherche d'un élement dans un tableau trié et donner sa complexité au pire cas.

Exercice

Occurrences. Écrire un algorithme pour compter le nombre d'occurrences d'un élément donné dans un tableau trié et donner sa complexité pire cas.

Exercice

Pavage de trominos.



1. Proposer un algorithme pour paver une grille de taille $2^n \times 2^n$, comportant un trou, avec des *trominos* de taille 2.

   Afficher la liste des trominos. On représente un tromino par les coordonnées de la case en haut à gauche du carré dans lequel il est inscrit plus les coordonnées de la case manquante dans ce carré.
2. Quelle est la complexité au pire cas de cet algorithme ?

Diviser pour régner : exercices

Exercice

Pesées. On a n pièces de monnaies dont une exactement est fausse et plus légère que les autres. On dispose d'une balance de type Roberval ou à fléau : en une pesée on peut déterminer si un ensemble de pièces est plus lourd qu'un autre.

Écrire un algorithme pour trouver la pièce fausse à l'aide de la balance et donner sa complexité au pire cas.

On représente les pièces par un tableau de poids et on veut l'indice de la pièce fausse. On dispose d'une fonction `compare(A, d1, f1, d2, f2)` qui renvoit $-1, 0$ ou 1 si la somme des poids des pièces aux indices d1 jusqu'à f1 est respectivement plus petite, égale, ou plus grande que celle des pièces de d2 à f2.

Exercice

Tri rapide (quick sort). Si on choisit un élément du tableau (le *pivot*), on peut partitionner les autres éléments en ceux qui sont plus petits, et ceux qui sont plus grands que le pivot. En exploitant ce principe, proposer un algorithme de tri d'un tableau et donner sa complexité au pire cas.

Diviser pour régner : Exercices

```
1  def quicksort(A, d, f):
2      if f > d:
3          p = partition(A, d, f)
4
5          # le pivot est à sa place
6          quicksort(A, d, p-1)
7          quicksort(A, p+1, f)
8
9  def partition(A, d, f)
10     # on prend un élément au hasard comme pivot
11     p = random_int(d, f)
12     swap(A, f, p) # on le met à la fin
13     j = d
14     for i = d to f - 1:
15     if A[i] ≤ A[f]:
16             swap(A, j, i)
17             j = j + 1
18
19     swap(A, j, f)
20     return j
```

Exercice

On veut calculer la complexité en moyenne du tri rapide avec un **choix de pivot aléatoire**. Cette complexité est dominée par le nombre de comparaisons.

1. Montrer que pour chaque paire $\{i, j\}$, on a au plus une comparaison de $A[i]$ et $A[j]$
2. Soit S le tableau trié. Montrer que pour $i < j$, la probabilité que $S[i]$ soit comparé à $S[j]$ est $\frac{2}{j-i+1}$
3. Conclure en utilisant le fait que $\sum_{k=1}^{n} \frac{1}{k} = O(\ln n)$.

Outline

Programmation dynamique

Exemple

Problème du change. On dispose de pièces (autant qu'on veut) de 1, 3, 4, 10, 30 et 40¤. Pour n donné, quel nombre minimum de chaque pièce faut-il pour obtenir n¤ ? Par exemple, pour $n = 25$, il faut une $2 \times 10 + 1 \times 4 + 1 \times 1$. Pour $n = 6$, il faut 2×3.

- une fois choisie une pièce de p¤, il reste à résoudre le même problème pour $n - p$.
- on obtient un algorithme récursif, avec un soupçon d'énumération exhaustive ;
- soit $L = \{1, 3, \ldots, 40\}$

- le nombre de pièces nécessaires est :

$$c(n) = \begin{cases} 1 + \min_{p \in L} c(n-p) & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ +\infty & \text{si } n < 0 \end{cases}$$

- pour la liste des pièces :

```
1  def change(n, L):
2      if n < 0:
3          return (+∞, [])
4      else if n == 0:
5          return (0, [])
6      else :
7          c = +∞
8          for p in L:
9              (x, R) = change(n − p, L)
10             if x < c:
11                 c = x; S = R; q = p
12
13         return (1 + c, S ++ [q])
```

Complexité de l'algorithme naïf de change

- Si $T(n)$ est le nombre d'appels à la fonction c, on a :

$$T(0) = 1 \text{ et } T(n) = 1 + \sum_{p \in L} T(n-p)$$

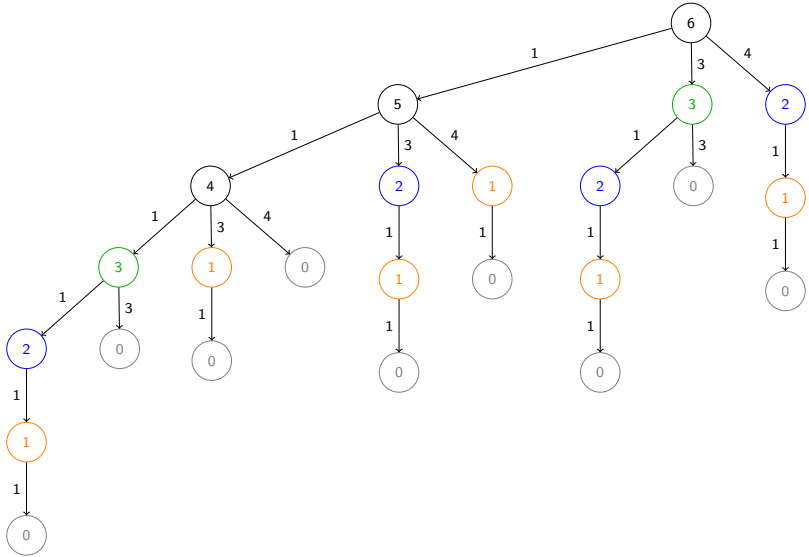
- Si $n > p_{\max}$, $c(n - p_{\max})$ est plus petit que tous les autres et :

$$T(n) \geq |L| T(n - p_{\max})$$

- Et finalement :

$$T(n) = \Omega(|L|^{\frac{n}{p_{\max}}})$$

Complexité de l'algorithme naïf de change



Programmation dynamique : sous-sous-problèmes communs

- Dans le problème du change, les sous-problèmes initiaux se **chevauchent** : ils **partagent** des sous-sous-problèmes
- L'algorithme naïf résoud donc plusieurs fois ces mêmes sous-sous-problèmes.
- Pour optimiser cela on peut utiliser la **mémoïsation** (*memoization*)

Programmation dynamique : mémoïsation

- **Mémoïsation** : mémoriser les résultats des sous-sous problèmes dans une table :
  - Si le résultat est dans la table, le récupérer directement ;
  - Sinon, faire le calcul complet et stocker le résultat dans la table.

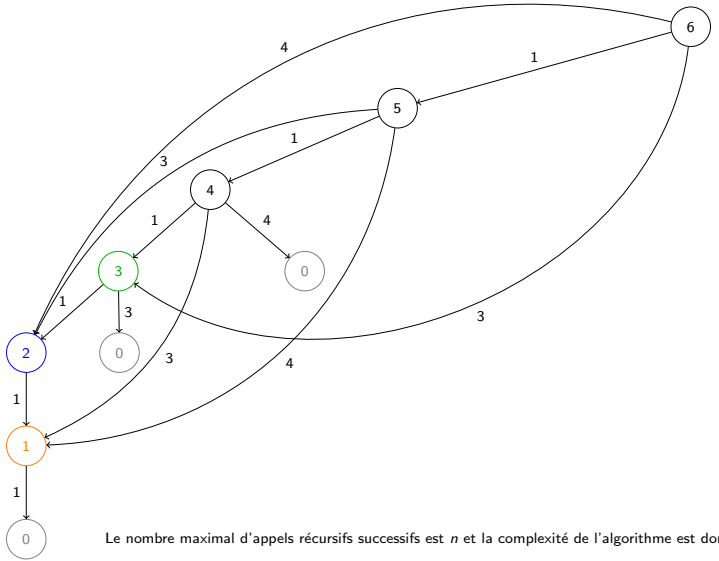
Exemple

```
1  def memo_nb_change(n, L, table): # initialement table contient des +infty
2      if n < 0:
3          return +∞
4      else if n == 0:
5          return (0, [])
6      else if table[n] ≠ +∞: # verification dans la table
7          return table[n]
8      else:
9          c = +∞
10         for p in L:
11             x = memo_nb_change(n − p, L, table)
12             if x < c:
13                 c = x
14
15         table[n] = c # mémorisation du résultat
16         return 1 + c
```

g

- Chaque sous-sous-problème n'est résolu qu'**une seule fois**.

Complexité de l'algorithme de change mémoïsé



Le nombre maximal d'appels récursifs successifs est n et la complexité de l'algorithme est donc $O(|L|n)$

Programmation dynamique : approche ascendante

- En général, on sait qu'on va devoir calculer la valeur de (presque) tous les sous-problèmes ;
- On peut le faire de façon **systématique** en remontant le long de l'arbre de dépendance des sous-problèmes ;
- Cela revient le plus souvent à les calculer par ≪ **taille** ≫ croissante ;
- Si la plupart des sous-problèmes est visitée par l'algorithme naïf, alors l'approche ascendante est **plus efficace** que l'approche descendante avec mémoïsation.

Exemple

```
1  def asc_nb_change(n, L):
2      r[0] = 0
3      for i = 1 to n:
4          c = +∞
5          for p in L:
6              if i − p ≥ 0 and r[i − p] < c:
7                  c = r[i − p]
8
9          r[i] = 1 + c
10
11     return r[n]
```

Comme précédemment la complexité de l'algorithme est $O(|L|n)$.

Programmation dynamique : approche ascendante

- En analysant les dépendances, l'approche ascendante permet parfois de ne pas mémoriser toute la table

Exemple

$f(n) = f(n-1) + f(n-2)$ pour $n \geq 2$, et $f(0) = 0$ et $f(1) = 1$.

```
1  # on initialise t avec des zéros
2  def fibm(n, t):
3      if n == 0 or n == 1:
4          return n
5      else if t[n] > 0:
6          return t[n]
7      else:
8          t[n] = fibm(n-1) + fibm(n-2)
9          return t[n]
```

```
1  def fibasc(n):
2      t[0] = 0
3      t[1] = 1
4
5      for i = 2 to n
6          t[i] = t[i - 1] + t[i -2]
7
8      return t[n]
```

```
1  def fib(n):
2      tprev = 0
3      tcour = 1
4
5      for i = 2 to n:
6          tcour = tcour + tprev
7          tprev = tcour - tprev
8
9      return tcour
```

Programmation dynamique : bilan

- La programmation dynamique s'applique lorsque l'on a :
  1. une **sous-structure optimale** : on peut décomposer en sous-problèmes **indépendants** tels qu'une solution globale optimale fournit des solutions optimales à ses sous-problèmes
     et donc des solutions optimales aux sous-problèmes peuvent être recombinées en une solution optimale globale
     $\Rightarrow$ Permet de dériver une résolution récursive du problème
  2. des **sous-problèmes qui se chevauchent** : ils partagent des sous-sous problèmes
     mais sont bien indépendants car une solution optimale d'un sous-sous problème conduit à une solution optimale de tous les sous-problèmes qui le contiennent
     $\Rightarrow$ rend la résolution naïve inefficace.
- Pour traiter efficacement les sous-sous-problèmes communs :
  - soit approche descendante (récursive) avec **mémoïsation** ;
  - soit approche ascendante (itérative) qui traite tous les sous-sous problèmes de façon systématique par $\ll$ taille $\gg$ croissante.
- L'approche ascendante est souvent plus efficace ;
- La mémoïsation peut être utilisée dans d'autres contextes.

Programmation dynamique : Exercice

Exercice

Sommes descendantes. On considère une matrice carrée d'entiers positifs de taille n. Écrire un algorithme qui calcule, pour chaque case sur la première ligne, le chemin (et la somme) qui réalise la plus grande somme en descendant d'une ligne à la fois vers la dernière ligne, et en se déplaçant à chaque fois de 0 à une colonne vers la gauche ou la droite.

$$\begin{bmatrix} 4 & 5 & 8 & 9 \\ 7 & 3 & 4 & 6 \\ 1 & 5 & 2 & 5 \\ 9 & 2 & 6 & 4 \end{bmatrix}$$

Programmation dynamique : Exercice

Exercice

Sommes descendantes. On considère une matrice carrée d'entiers positifs de taille n. Écrire un algorithme qui calcule, pour chaque case sur la première ligne, le chemin (et la somme) qui réalise la plus grande somme en descendant d'une ligne à la fois vers la dernière ligne, et en se déplaçant à chaque fois de 0 à une colonne vers la gauche ou la droite.

$$\begin{bmatrix} 4 & 5 & 8 & 9 \\ 7 & 3 & 4 & 6 \\ 1 & 5 & 2 & 5 \\ 9 & 2 & 6 & 4 \end{bmatrix}$$

Exercice

Chaînes de multiplications de matrices. On considère le produit de n matrices pas forcément carrées, mais aux dimensions compatibles $A_1 A_2 \cdots A_n$. On le résoud par des produits deux-à-deux via l'algorithme classique. Le produit de matrices est associatif ($(AB)C = A(BC)$) et l'ordre dans lequel on fait les produits deux-à-deux influe grandement sur le nombre de multiplications scalaires réalisées. Le but de cet exercice est d'écrire une fonction qui calcule l'ordre optimal dans lequel faire ces produits.

1. Écrire l'algorithme classique de produit de deux matrices. Combien de multiplications scalaires réalise-t-il ?
2. Donner un exemple d'un produit de 3 matrices (juste leurs dimensions) pour lequel l'ordre des produit deux à deux donne des nombres de multiplications scalaires (très) différents.
3. Donner une expression récursive du nombre de parenthésages possibles $P(n)$ et montrer que $P(n) = \Omega(2^n)$.

Programmation dynamique : Exercice

Exercice

Chaînes de multiplications de matrices (suite). On va maintenant écrire une fonction qui calcule le nombre minimal de multiplications scalaires pour un produit de n matrices donné.

1. Supposons qu'on a parenthésé le produit complètement **et de façon optimale** pour n'avoir que des multiplications deux à deux. Faire apparaître deux sous-problèmes identiques au problème original mais de tailles inférieures ;
2. Montrer que le parenthésage optimal global pour la solution considérée fournit des parenthésages optimaux pour les sous-problèmes ;
3. En déduire une expression récursive du nombre de multiplications scalaires pour la solution optimale considérée ;
4. En déduire une expression récursive du nombre de multiplications scalaires pour n'importe quel produit de n matrices ;
5. Montrer que les sous-problèmes dans cette expression partagent des sous-sous-problèmes ;
6. Écrire un algorithme descendant avec mémoïsation pour résoudre le problème ;
7. Écrire un algorithme ascendant pour résoudre le problème ;
8. Donner la complexité au pire cas des deux algorithmes ;
9. Écrire un algorithme pour visualiser le parenthésage optimal.

Outline

Algorithmes gloutons

- Plutôt pour des problèmes d'optimisation ;
- Comme dans le *backtracking*, on décompose le problème en deux :
  1. Faire un choix ;
  2. Résoudre le sous-problème resultant de choix.
- À chaque étape on fait le choix qui **optimise localement** l'objectif ;
- On construit donc **une seule** solution.

Algorithmes gloutons

- Plutôt pour des problèmes d'optimisation ;
- Comme dans le *backtracking*, on décompose le problème en deux :
  1. Faire un choix ;
  2. Résoudre le sous-problème resultant de choix.
- À chaque étape on fait le choix qui **optimise localement** l'objectif ;
- On construit donc **une seule** solution.

Exercice

Choix des activités. Étant donnée une liste d'activités caractérisées par une date de début et une date de fin, trouver un ensemble maximum d'activités à faire tel qu'il n'y ait aucun chevauchement.

Activité	Début	Fin
τ_1	1	5
τ_2	2	3
τ_3	3	5
τ_4	3	7
τ_5	4	8
τ_6	5	6
τ_7	6	7
τ_8	8	10

Algorithmes gloutons

Exemple

```
1  def activites(D, F, n):
2      # tri selon F
3      A = sortBy(F, [0..(n-1)])
4      resultat = [A[0]]
5      date = F[A[0]]
6
7      for i = 0 to (n-1):
8          if D[A[i]] ≥ date:
9              date = F[A[i]]
10             resultat = resultat ++ [A[i]]
11
12     return resultat
```

- Le ≪ choix glouton ≫ n'est pas optimal pour tous les problèmes
  Par exemple, plus court chemin sur un réseau routier et choix selon la distance à vol d'oiseau
- Quand il n'est pas optimal, il peut parfois fournir une bonne approximation ;
- Mais peut aussi fournir la pire solution !

Algorithmes gloutons : optimalité

- Un algorithme glouton est optimal s'il satisfait :
  1. **choix glouton** : il existe une solution optimale qui contient le choix glouton
  2. **sous-structure optimale** : on peut décomposer en sous-problèmes **indépendants** tels qu'une solution globale optimale fournit des solutions optimales à ses sous-problèmes
     et donc des solutions optimales aux sous-problèmes peuvent être recombinées en une solution optimale globale
     $\Rightarrow$ Permet de dériver une résolution récursive du problème

Algorithmes gloutons : optimalité du choix des activités

- On décompose le choix des activités P en :
  1. Q : choisir la première activité a_1
  2. P' : choisir les activités à partir de la date de fin de a_1
- P et P' sont des cas particuliers de P_t : choisir les activités à partir de la date t
- choix glouton :

Lemme

S'il y a une solution optimale pour P, il y en a une en choisissant pour a_1 la tâche avec la date de fin minimum.

- sous-structure optimale :

Lemme

Si a_k fait partie d'une solution optimale, $A_k^<$ est une solution optimale pour le problème finissant au début de a_k et $A_k^>$ une solution optimale pour le problème commençant à la fin de a_k, alors $A_k^< \cup \{a_k\} \cup A_k^>$ est une solution optimale.

Exercice

Démontrer les deux lemmes.

Algorithmes gloutons : Exercice

Exercice

Un problème d'ordonnancement. On dispose de n tâches à exécuter. Chaque tâche τ_i a un **temps d'exécution** C_i, le temps qu'elle met a s'exécuter en isolation,

Le **temps de réponse** de τ_i est le temps entre son activation et sa fin (incluant les temps d'attente).

1. On suppose qu'une tâche commencée ne peut pas être interrompue (ordonnancement **non-préemptif**) et que toutes les tâches sont activées à l'instant initial.
   Donner un algorithme qui ordonnance les tâches en minimisant le temps de réponse moyen ;
2. On suppose maintenant l'ordonnancement **préemptif** : une tâche peut être interrompue le temps que d'autres s'exécutent. Elle reprend là où elle s'était arrêtée quand elle est choisie à nouveau pour s'exécuter. On suppose également que chaque tâche τ_i a maintenant une **date d'activation** d_i qui peut ne pas être 0.
   Donner un algorithme qui ordonnance les tâches en minimisant le temps de réponse moyen.

Algorithmes gloutons et matroïdes

- Certains problèmes ont une structure particulière appelée **matroïde** ;
- Sur un matroïde (pondéré) on peut définir un algorithme glouton générique et optimal par construction

Définition (Matroïde)

Un **matroïde** pondéré (ou valué) est un triplet (E, I, w) tel que :

- E est un ensemble fini ;
- I est un ensemble de parties, dites **indépendantes**, de E ($I \subseteq 2^E$) vérifiant :
  - **hérédité** : si $A \in I$ et $B \subseteq A$ alors $B \in I$
  - **échange** : si $A, B \in I$ et $|A| > |B|$ alors **il existe** $e \in A \setminus B$ tel que $B \cup \{e\} \in I$.
- $w : E \to \mathbb{R}$ est une fonction de pondération.

Algorithmes gloutons et matroïdes

Définition (Algorithme glouton sur un matroïde pondéré)

```
1 | def matroid_max(E, I, w):
2 |     X = ∅
3 |     while ∃e ∈ E t. q. X ∪ {e} ∈ I
4 |         X = X ∪ argmax{w(e)|e ∈ E t. q. X ∪ {e} ∈ I}
5 |
6 |     return X
```

Théorème

Si (E, I, w) est un matroïde pondéré, le résultat X de `matroid_max(E, I, w)` est une partie de I telle que $w(X) = \sum_{x \in X} w(x)$ est maximale.

Algorithmes gloutons et matroïdes

Exemple

Problème du change. On dispose de pièces de 1, 2, 5, 10, 20 et 50¤. Pour n donné, quel nombre minimum de chaque pièce faut-il pour obtenir n¤ ? Par exemple, pour $n = 9$, il faut une $1 \times 5 + 2 \times 2$.

- Avec $L = \{1, 2, 5, 10, 20, 50\}$ l'algorithme glouton intuitif fonctionne !
- On va montrer qu'on a un matroïde pondéré :
  - E est l'ensemble des pièces tel que pour chaque dénomination p_i on en a $\left\lfloor \frac{n}{p_i} \right\rfloor$ exemplaires.
  - I est l'ensemble des parties de E dont la valeur totale est $\leq n$ et tel qu'il n'existe pas d'autre partie de même valeur totale et comportant moins de pièces
  - w associe sa valeur à chaque pièce.

Algorithmes gloutons et matroïdes

- Pour tout $X \subseteq E$, on note $|X|_i$ le nombre d'éléments de x dont la valeur est $p_i \in L$.

Lemme

Pour la liste L donnée, X est indépendant ($x \in I$) ssi

$$|X|_1 \leq 1 \wedge |X|_1 + |X|_2 \leq 2 \wedge |X|_5 \leq 1 \wedge |X|_{10} \leq 1 \wedge |X|_{10} + |X|_{20} \leq 2$$

- **hérédité** : immédiat d'après la forme des contraintes dans le lemme ;
- **échange** : Soit $X, Y \in I$ avec $|X| < |Y|$
  - Si $|Y|_{50} > 0$ alors on peut ajouter une pièce de 50 à X et les contraintes sont toujours satisfaites ;
  - Si $|Y|_{50} = 0$ alors :
    - on a $|X|_1 + |X|_2 + |X|_5 + |X|_{10} + |X|_{20} + |X|_{50} < |Y|_1 + |Y|_2 + |Y|_5 + |Y|_{10} + |Y|_{20}$
    - donc $|X|_1 + |X|_2 + |X|_5 + |X|_{10} + |X|_{20} < |Y|_1 + |Y|_2 + |Y|_5 + |Y|_{10} + |Y|_{20}$
    - donc soit $|X|_1 + |X|_2 < |Y|_1 + |Y|_2$, soit $|X|_5 < |Y|_5$, soit $|X|_{10} + |X|_{20} < |Y|_{10} + |Y|_{20}$
    - ce qui implique que l'une des quantités en $|Y|$ est > 0
    - comme $X_1 + X_2 < 2$ implique $|X|_1 \leq 1$ (et pareil pour 10 et 20), on peut prendre une pièce de la quantité en Y positive et la mettre dans X en respectant les contraintes.

Algorithmes gloutons et matroïdes

- Si X est indépendante et $p \in L$ réalise le maximum de $w(p)$ tel que $w(X) + w(p) \leq n$ alors $X \cup \{p\}$ est indépendante ; ajouter la dénomination maximale préserve l'indépendance
- En on déduit l'algorithme glouton :

```
1  def change_glouton(n, L):
2      X = ∅
3      while w(X) ≤ n:
4          X = X ∪ max{w(p)|w(X) + w(p) ≤ n}
5
6      return X
```

- Si la somme n n'était pas réalisable on obtiendrait la plus grande valeur inférieure.
- On peut implémenter cet algorithme un peu abstrait comme cela :

```
1  def decomp(N):
2      pieces = [1, 2, 5]
3      resultat = [0]*pieces[len(pieces) - 1]
4
5      i = len(pieces) - 1
6      while N > 0 and i >= 0:
7          if N - pieces[i] > 0:
8              N = N - pieces[i]
9              resultat[pieces[i]] = resultat[pieces[i]] + 1
10         else:
11             i = i - 1
12
13     return resultat
```

Algorithmes gloutons et matroïdes : Exercice

Exercice

Un autre problème d'ordonnancement. On dispose de n tâches de durée 1 activées à l'instant initial. À chaque tâche τ_i est associée une **échéance** d_i et une pénalité p_i : la tâche doit avoir été exécutée avant d_i, sinon il faut payer la pénalité p_i. Par exemple :

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
d_i	4	3	4	2	1	6	1
p_i	10	20	30	40	50	60	70

Si on exécute les tâches dans l'ordre des indices on obtient que τ_5 et τ_7 sont en retard et la pénalité totale est donc $50 + 70 = 120$.

Modéliser sous la forme d'un matroïde le problème d'obtenir un algorithme calculant un ordonnancement minimisant la pénalité totale.

Outline

Transformations de problèmes

- On peut résoudre de nombreux problèmes en les transformant vers d'autres ;
- À condition que la transformation ne soit pas plus complexe que le problème !
- Quelques problèmes cibles classiques, avec quelques outils :
  - Problèmes de graphes : plus court chemin, coloration, cycle hamiltonien, arbre couvrant, cliques, etc.
  - SAT : par exemple MiniSAT, CP-SAT (Google)
  - SMT (*satisfiability modulo theory*) : une extension de SAT avec des prédicats (p. ex. $x \geq 3$). Z3 (Microsoft Research) est le solver SMT le plus connu.
  - Programmation linéaire (en nombre entiers) : par exemple CPLEX (IBM), GLPK, Gurobi

Outline

Structures de données

- Représenter et organiser des **collections** d'objets ;
- Choisir la ≪ bonne ≫ **structure** de données ;
- Spécialisation sur certaines opérations : **types de données abstraits** ;
- Comprendre leurs **implémentations** possibles :
  - maîtriser leur complexité ;
  - pouvoir les **étendre** avec de nouvelles opérations.

Opérations sur les collections

accès direct successeur recherche

insertion suppression

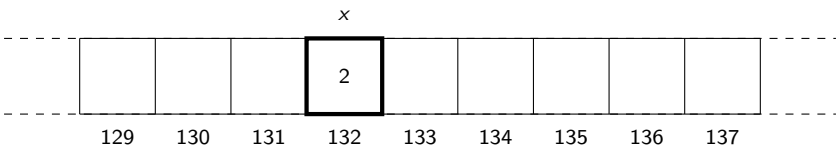
minimum maximum

Outline

Tableaux et listes

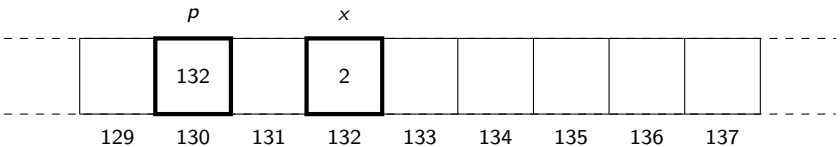
- Tableaux et listes organisent (conceptuellement) les données de façon **linéaire** ;
- Tableaux :
  - accès direct
  - taille fixe
- Listes :
  - structure récursive : tête et queue
  - accès séquentiel (en particulier accès direct à la tête) ;
  - taille variable

Implémentation des tableaux : adresses et pointeurs



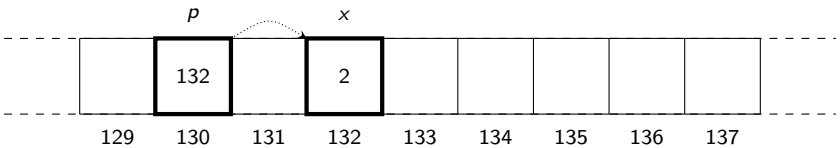
- x est une variable. Elle a un nom (x), un **type** (entier sur 8 bits), un contenu (2), et une **adresse** (132) ;
- Un adresse est représentée par un entier (numéro de la case mémoire) ;

Implémentation des tableaux : adresses et pointeurs



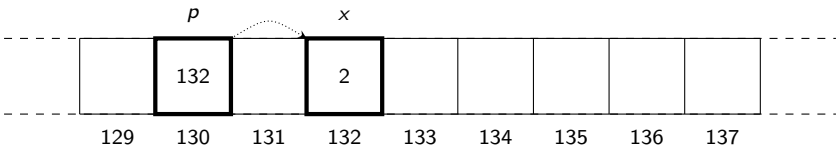
- x est une variable. Elle a un nom (x), un **type** (entier sur 8 bits), un contenu (2), et une **adresse** (132) ;
- Un adresse est représentée par un entier (numéro de la case mémoire) ;
- p est une variable qui **contient une adresse** ;

Implémentation des tableaux : adresses et pointeurs



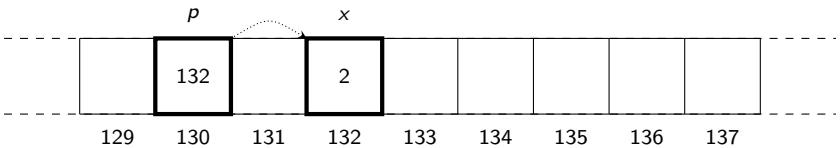
- x est une variable. Elle a un nom (x), un **type** (entier sur 8 bits), un contenu (2), et une **adresse** (132) ;
- Un adresse est représentée par un entier (numéro de la case mémoire) ;
- p est une variable qui **contient une adresse** ;
- Son type est **pointeur sur un entier sur 8 bits** ;

Implémentation des tableaux : adresses et pointeurs



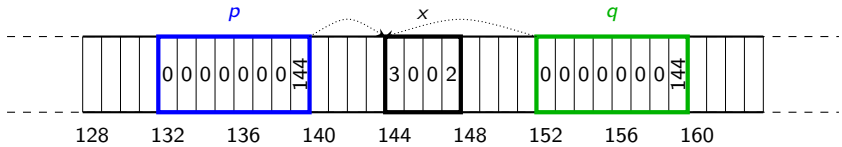
- x est une variable. Elle a un nom (x), un **type** (entier sur 8 bits), un contenu (2), et une **adresse** (132) ;
- Un adresse est représentée par un entier (numéro de la case mémoire) ;
- p est une variable qui **contient une adresse** ;
- Son type est **pointeur sur un entier sur 8 bits** ;
- Opération spécifique sur les types pointeurs : le **déréférencement**
  $\Rightarrow$ *p est la variable dont l'adresse est dans p

Implémentation des tableaux : adresses et pointeurs



- x est une variable. Elle a un nom (x), un **type** (entier sur 8 bits), un contenu (2), et une **adresse** (132) ;
- Un adresse est représentée par un entier (numéro de la case mémoire) ;
- p est une variable qui **contient une adresse** ;
- Son type est **pointeur sur un entier sur 8 bits** ;
- Opération spécifique sur les types pointeurs : le **déréférencement**
  $\Rightarrow$ `*p` est la variable dont l'adresse est dans p
- On a aussi une opération pour **obtenir l'adresse** d'une variable :
  $\Rightarrow$ `&x` donne l'adresse (de début) de x

Implémentation des tableaux : adresses et pointeurs



- En réalité, les variables (et notamment les pointeurs) ont souvent une taille > 1.
- x est un entier sur 32 bits qui contient la valeur $2 + 0 \times 256 + 0 \times 256^2 + 3 \times 256^3 = 16\,777\,218$ (*big endian*) ;
- p est un pointeur sur un entier sur 32 bits :
  - Sa taille est 64 bits
  - Il contient l'adresse (du début) de x : 144
  - $*p$ vaut 16 777 218
- q est un pointeur sur un entier sur 8 bits :
  - Sa taille est 64 bits
  - Il contient l'adresse (du début) de x : 144
  - $*q$ vaut 3

Dans la suite on **abstrait la taille des données** *pour simplifier.*

Implémentation des tableaux

- Pour permettre l'accès direct, on place les éléments côte-à-côte en mémoire ;

- La variable tableau A contient l'**adresse** du 1er élément ;
- C'est une variable de type **pointeur**
- Tous les éléments ont la même taille n, pas forcément 1 ;
- Le i^e élément est à l'adresse $A + n \times i$ et donc

$$A[i] \equiv *(A + n \times i)$$

accès	$O(1)$
insertion	$O(n)$
supression	$O(n)$
insertion/suppression à la fin	$O(1)$

Tableaux dynamiques

- Comment ajouter un élément dans un tableau A quand il est plein ?
  1. Allouer un nouveau tableau plus grand en mémoire ;
  2. Copier l'ancien tableau dans le (début du) nouveau ;
  3. Faire pointer A sur le nouveau tableau ;
  4. Ajouter l'élément.
- Complexité : $O(n)$
- Mais si on fait beaucoup d'accès et peu de redimensionnements, ils sont négligeables.
  $\Rightarrow$ Notion de **complexité amortie**.

Complexité amortie

- On considère une séquence $i_1, i_2, \ldots, i_n$ d'instructions ;
- Soit c_k le coût de l'instruction i_k ;
- Le **coût amorti** de chaque instruction est la moyenne $\frac{1}{n}\sum_{k=1}^{n} c_k$

Exemple

On considère une chaîne de caractères (un tableau) contenant initialement des espaces et suffisamment grande. On dispose de trois opérations :

1. Ajouter un caractère à la fin de la chaîne, remplaçant le premier espace (`'a'`, ..., `'z'`) : $O(1)$
2. Effacer le dernier caractère (`del`) O(1)
3. Effacer tous les caractères (`clear`) : 1 + nb de caractères à effacer = $O(n)$.

Séquence	nb. instr.	coût
`'m', 'e', 'a', del, 'u', clear, 'c', 'o', 'i', 'n'`	10	14
`'a', clear, clear, clear`	4	5
`'a', 'b', 'c', del, del, clear`	4	7

Plus généralement, le coût d'une séquence de n instructions est $O(n)$.
Le **coût amorti** de `clear` est donc $\frac{O(n)}{n} = O(1)$.

- On note avec un $\tilde{}$ la complexité amortie : pour `clear` c'est $\tilde{O}(1)$.

Coût amorti : méthode du potentiel

- Pour calculer le coût amorti on peut utiliser une sur-approximation basée sur une **fonction de potentiel** bien choisie ;
- Elle donne une valeur dans $\mathbb{R}$ à une configuration (valeurs des variables) donnée ;
- Elle représente un crédit qu'on peut stocker pour payer un coût plus tard ;
- On considère :
  - la valeur initiale des variables $\vec{v}_0$;
  - la séquence d'instruction $i_1, \ldots, i_n$;
  - la valeur des variables $\vec{v}_k$ obtenue par l'instruction i_k
  - la fonction de potentiel ϕ.
- Le **coût potentiel** de l'instruction i_k est $p_k = c_k + \phi(\vec{v}_k) - \phi(\vec{v}_{k-1})$;
- Si $\phi(\vec{v}_n) \geq \phi(\vec{v}_0)$, on a donc : $\sum_{k=1}^{n} c_k \leq \sum_{k=1}^{n} p_k$;

Méthode du potentiel : exemple

Exemple

Quel est le coût amorti de `clear` ?

- Soit s le nombre de caractères différents d'espace dans la chaîne ;
- On choisit la fonction $\phi(s) = s + 1$
- Initialement, la chaîne est vide donc $\phi(\vec{v}_0) = 1$;
- Comme il y a toujours au moins 0 caractères dans la chaîne on a toujours ; $\phi(\vec{v}_n) \geq 1 = \phi(\vec{v}_0)$;
- Le coût potentiel de la séquence est donc $\geq$ au coût total de la séquence.
- On calcule le coût potentiel de chaque opération :
  - si $i_k =$ `'c'`, $p_k = 1 + \phi(\vec{v}_k) - \phi(\vec{v}_{k-1}) = 1 + ((s+1)+1) - (s+1) = 2$
  - si $i_k =$ `del`, $p_k = 1 + (s+1) - ((s+1)+1) = 0$
  - si $i_k =$ `clear`, $p_k = (1+s) + 1 - (s+1) = 1$
- Chaque opération a bien un coût potentiel $O(1)$ donc le coût potentiel total est $O(n)$, donc le coût amorti total est aussi $O(n)$ et le coût amorti de `clear` est $\tilde{O}(1)$.

Méthode du potentiel : exercice

Exercice

On considère l'incrémentation n fois d'un compteur binaire à k chiffres. Exemple pour $k = 3$, $n = 7$:

Nombre à incr.	Résultat	Chiffres à changer
000	001	1
001	010	2
010	011	1
011	100	3
100	101	1
101	110	2
110	111	1

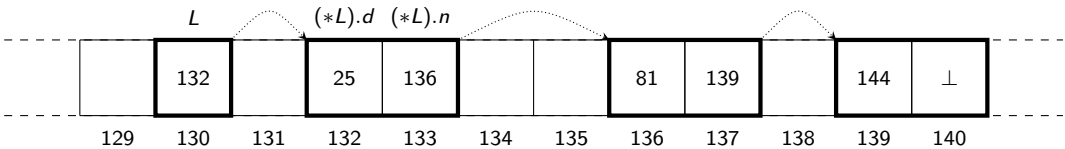
Déterminer le coût pire cas et, par la méthode du potentiel, le coût amorti de chaque incrémentation.

Complexité amortie du redimensionnement de tableau

- Comment ajouter un élément dans un tableau A quand il est plein ?
  1. Allouer un nouveau tableau plus grand en mémoire ;
  2. Copier l'ancien tableau dans le (début du) nouveau ;
  3. Faire pointer A sur le nouveau tableau ;
  4. Ajouter l'élément.
- On suppose une séquence d'insertions en fin de tableau plus grande que le tableau ;
- On augmente la taille du tableau de k cases quand il est plein ;
- Soit A_i le tableau après l'instruction i et s_i sa taille ;
- On définit la **fonction de potentiel** $\phi(i) = 2i - s_i$;
- Coût potentiel des insertions :
  - Si $i < s_i$: $1 + (2 * (i+1) - s_{i+1}) - (2 * i - s_i) = 3$ car $s_{i+1} = s_i$;
  - Si $i = s_i$: $(1 + s_i) + (2 * (i+1) - s_{i+1}) - (2 * i - s_i) = 3 + s_i - k$ car $s_{i+1} = s_i + k$;
- Avec $k = s_i$, on a donc une complexité amortie $\tilde{O}(1)$.

Implémentation des listes : listes chaînées

- Deux points faibles importants des tableaux :
  1. Insertion et suppression en $O(n)$ (sauf à la fin) ;
  2. Nécessité d'allouer un bloc de mémoire $\ll$ connexe $\gg$.
- Un alternative est la **liste chaînée** :



- Chaque cellule C contient la donnée `C.d` et un **pointeur** vers la cellule suivante `C.n` ;
- Chaque peut se trouver n'importe où en mémoire
- Le début de la liste est un pointeur vers la première cellule ;

accès	$O(n)$
insertion/suppression	$O(n)$
insertion/suppression avec pointeur	$O(1)$

Listes chaînées : insertion/suppression

```
1  # insertion de l'élément x après la cellule pointée par L
2  def  insert (L, x):
3      # On suppose le type enregistrement cell bien défini
4      # L' est un pointeur vers la nouvelle cellule
5      L' = memory_alloc(cell)
6
7      (*L').n = (*L).n
8      (*L').d = x
9
10     (*L).n = L'
```

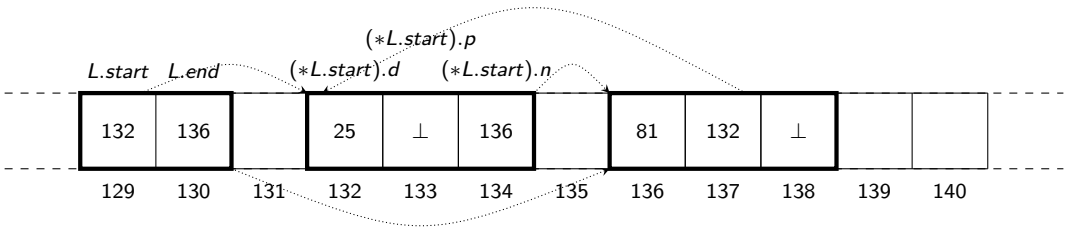
```
1  # insertion de l'élément x en tête de la liste pointée par L
2  def  insert_front (L, x):
3      # On suppose le type enregistrement cell bien défini
4      # L' est un pointeur vers la nouvelle cellule
5      L' = memory_alloc(cell)
6
7      (*L').n = L
8      (*L').d = x
9
10     return L'
```

Exercice

1. Écrire une fonction qui supprime la cellule **suivant** celle pointée par L (si elle existe) dans une liste chaînée ;
2. Écrire une fonction qui supprime la première occurrence de l'élément x dans un liste chaînée.

Listes doublement chaînées

- Pour simplifier et améliorer l'utilisation des listes chaînées on peut ajouter un **chaînage inverse** ;
- Au **prix** d'un pointeur supplémentaire par cellule vers la cellule **précédente**.



accès	$O(n)$
insertion/suppression	$O(n)$
insertion/suppression avec pointeur (dont début et fin)	$O(1)$

Listes doublement chaînées : insertion/suppression

```
1  # insertion de l'élément x après la cellule pointée par L
2  def  insert2_after (L, x):
3      # On suppose le type enregistrement cell bien défini
4      # L' est un pointeur vers la nouvelle cellule
5      L' = memory_alloc(cell)
6
7      (*L').n = (*L).n
8      (*L').p = L                  # new !
9      (*L').d = x
10
11     if (*L).n ≠ ⊥ :              # new !
12         (*((*L).n)).p = L'       # new !
13
14     (*L).n = L'
15
16     # si L était la fin de la liste, il faudra mettre à jour
17     # le pointeur end
18     return L'
```

```
1  # insertion de l'élément x avant la cellule pointée par L
2  def  insert2_before (L, x):
3      # On suppose le type enregistrement cell bien défini
4      # L' est un pointeur vers la nouvelle cellule
5      L' = memory_alloc(cell)
6
7      (*L').n = L
8      (*L').p = (*L).p             # new !
9      (*L').d = x
10
11     if (*L).p ≠ ⊥ :              # new !
12         (*((*L).p)).n = L'       # new !
13
14     (*L).p = L'
15
16     # si L était le début de la liste, il faudra mettre à jour
17     # le pointeur start
18     return L'
```

Exercice

1. Écrire une fonction qui supprime la cellule pointée par L dans une liste doublement chaînée ;
2. Écrire une fonction qui supprime la première occurrence de l'élément x dans un liste doublement chaînée.

Tableaux dynamiques et listes

- On peut aussi implémenter une liste avec un tableau dynamique ;
- Intérêt des listes chaînées :
  - fragmentation mémoire (attention **cache**),
  - taille variable à coût constant (non amorti),
  - travail sur les extrémités.
- Accéder à la fin d'une liste **simplement chaînée** en $O(1)$: ajouter juste un pointeur.

Outline

Files et piles

- On doit souvent **mémoriser** des éléments en attendant qu'ils soient traités ;
- On définit deux structures en fonction de l'ordre de traitement :
  - **File** : *First In First Out (FIFO)*
  - **Pile** : *Last In First Out (LIFO)*
- Opérations :
  - insérer / empiler (*enqueue* / *push*)
  - supprimer / dépiler (*dequeue* / *pop*)
  - prochain élément / dessus de pile (*next* / *top*)
  - vide

Files et piles : exercices

Exercice

La Bataille. Dans ce jeu de cartes, les 52 cartes sont distribuées entre les deux joueurs et gardées faces cachées. À chaque tour, les deux joueurs retournent la première carte de leur paquet. Le joueur avec la carte la plus forte (on considère uniquement la valeur, pas la couleur) remporte les deux cartes qu'il met au dessous de son paquet (d'abord la sienne puis l'autre). En cas d'égalité, il y a *Bataille !* Chaque joueur ajoute la prochaine carte de son paquet face cachée sur sa carte, puis retourne à nouveau la carte du dessus de son paquet. Le joueur avec la carte la plus forte, remporte toutes les cartes qu'il met en dessous de son paquet dans l'ordre (du dessus vers le dessous et comme précédemment). S'il y a encore égalité on répète l'opération autant de fois que nécessaire. À tout moment, un joueur qui ne peut pas jouer perd et son adversaire gagne.

Écrire un algorithme qui à partir de deux listes de cartes (les paquets des deux joueurs) détermine qui gagne la partie.

Files et piles : exercices

Exercice

Notation polonaise inversée. Une expression arithmétique en notation polonaise inversée (NPI) est telle que les arguments d'un opérateur précèdent cet opérateur. Par exemple : $12\ 34 + 8\ 5 - *4+$ est l'expression $((12 + 34) * (8 - 5)) + 4$.

On suppose que l'expression est bien formée et représentée par une chaîne de caractères et qu'elle contient exactement une espace après chaque nombre, et uniquement là. On dispose de deux fonctions :

- `is_digit` qui pour un caractère indique s'il représente un chiffre ;
- `c2i` qui pour un caractère représentant un chiffre renvoit l'entier correspondant.

1. On suppose que les nombres dans l'expression ne comportent qu'un seul chiffre. Donner un algorithme pour évaluer une telle expression.
2. Même question pour des nombres quelconques.

Exercice

1. Simuler une file avec deux piles ;
2. Simuler une pile avec deux files ;
3. Établir, dans les deux cas, la complexité (amortie) des opérations simulées.

Dérécursification

- Dans de nombreux langages (non fonctionnels), la récursion fait appel à la **pile du processus** qui a une taille limitée ;
- Certains conventions de programmation l'interdisent ;
- On peut donc vouloir supprimer cette récursion ;
- Dans le cas d'une fonction récursive terminale, on transforme directement l'appel récursif en une boucle ;
- Les paramètres et résultats de la fonction deviennent des variables :

Exemple

```
1 | # on appelle factr(n, 1)
2 | def factr(n, r):
3 |     if n == 0:
4 |         return r
5 |     else:
6 |         return factr(n - 1, r * n)
```

```
1 | def fact(n):
2 |     r = 1
3 |     while n != 0:
4 |         r = r * n
5 |         n = n - 1
6 |
7 |     return r
```

- Les autres cas sont moins directs, voire infaisables sans mémoire supplémentaire.

Dérécursification

- On peut utiliser une **pile** pour simuler celle des appels récursifs ;

Exemple

```
1  def fact2(n):
2      if n == 0:
3          r = 1
4      else:
5          r = n * fact2(n - 1)
6
7      return r
```

```
1  def fact3(n):
2      # frame est un type enregistrement avec les champs
3      # n : entier, r : entier, p : pointeur sur un entier
4      # a : entier, grow : booléen
5      r = 0
6      W.push(frame(n, 0, &r, 1, True))  # W est une pile
7      while not W.empty():
8          f = W.top()
9
10         if f.grow:  # la pile croît
11             if f.n == 0:
12                 *f.p = 1
13                 W.pop()
14             else:
15                 W.top().grow = False
16                 W.push(frame(f.n - 1, 0, &f.r, f.n, True))
17         else:       # la pile décroît
18             *f.p = f.r * f.a
19             W.pop()
20
21     return r
```

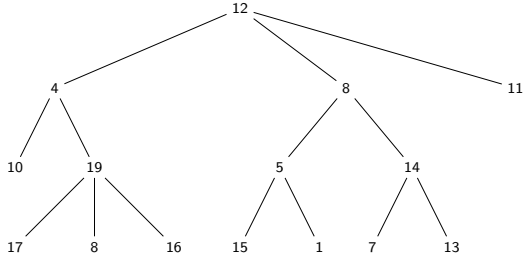
Outline

Files de priorité

- Toujours sortir le **plus petit** élément de la file
  P. ex. plus courts chemins, arbres couvrants, codage de Huffman, qualité de service, etc.
- On peut trier la file (typiquement) $O(n \log_2 n)$) mais il faut recommencer à chaque insertion ($O(n)$) ;
- Pour faire mieux : passer de la liste/tableau à l'**arbre**.

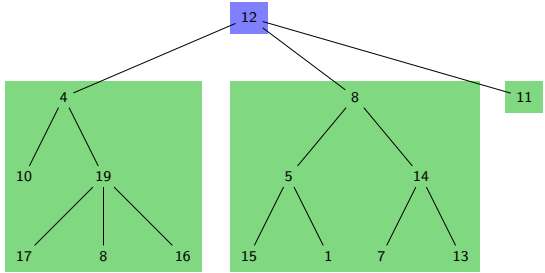
Arbres

- L'arbre étend la liste avec potentiellement plusieurs successeurs pour chaque élément ;
- L'arbre a aussi une **structure récursive** :
  - un élément : la racine ;
  - plusieurs sous-arbres.
- Un **arbre *n*-aire** a au plus n sous-arbres qui sont eux-aussi n-aires ;
- Un arbre ternaire :



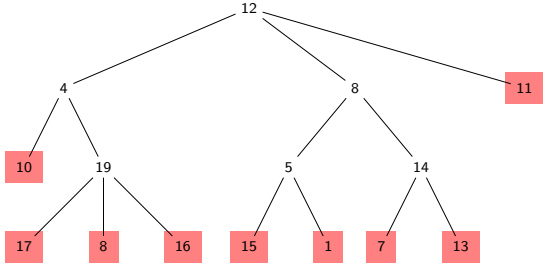
Arbres

- L'arbre étend la liste avec potentiellement plusieurs successeurs pour chaque élément ;
- L'arbre a aussi une **structure récursive** :
  - un élément : la racine ;
  - plusieurs sous-arbres.
- Un **arbre *n*-aire** a au plus n sous-arbres qui sont eux-aussi n-aires ;
- Un arbre ternaire :



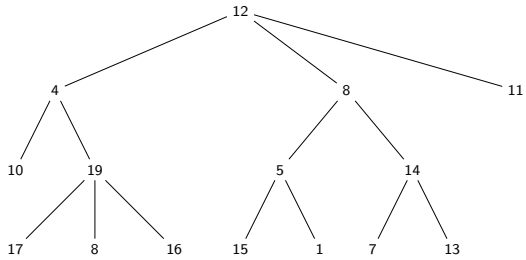
Arbres

- L'arbre étend la liste avec potentiellement plusieurs successeurs pour chaque élément ;
- L'arbre a aussi une **structure récursive** :
  - un élément : la racine ;
  - plusieurs sous-arbres.
- Un **arbre *n*-aire** a au plus n sous-arbres qui sont eux-aussi n-aires ;
- Un arbre ternaire :



- Les sous-arbres dont les sous-sous-arbres sont vides sont les **feuilles** ;

Hauteur d'un arbre et complexité

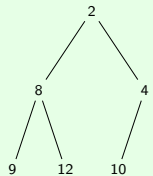


- La **hauteur** de l'arbre est la longueur du plus long chemin de la racine à une feuille (ici 4).
- Un arbre est **équilibré** en hauteur si les hauteurs de ses sous-arbre diffèrent d'au plus 1 ;
- Un arbre n-aire équilibré avec m nœuds a une hauteur d'au plus $\log_n m$;
- On construit une structure avec des opérations en O(hauteur).

Tas binaire

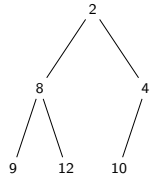
- Un **tas binaire** (*binary heap*) est un arbre binaire tel que :
  1. seul le dernier niveau n'est pas rempli
     donc il est équilibré
  2. dans chaque sous-arbre la racine est plus petite que les racines de ses sous-arbres
     donc plus petite que tous les éléments de ces sous-arbres

Exemple



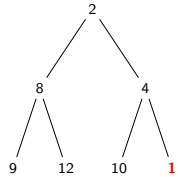
- Pour une file de priorités, il nous faut trois opérations :
  1. test du vide (trivial)
  2. insertion d'un élément
  3. extraction du minimum

Tas binaire : insertion



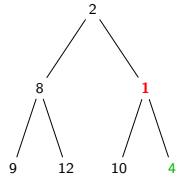
1. On ajoute l'élément à la fin ;
2. On le fait remonter à sa place le cas échéant.

Tas binaire : insertion



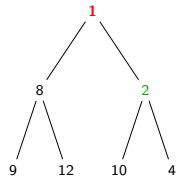
1. On ajoute l'élément à la fin ;
2. On le fait remonter à sa place le cas échéant.

Tas binaire : insertion



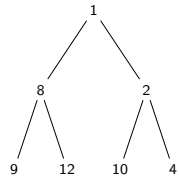
1. On ajoute l'élément à la fin ;
2. On le fait remonter à sa place le cas échéant.

Tas binaire : insertion



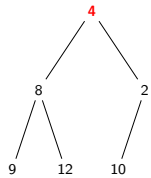
1. On ajoute l'élément à la fin ;
2. On le fait remonter à sa place le cas échéant.

Tas binaire : extraction du minimum



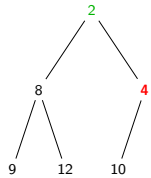
1. Le minimum est **la racine** ;
2. On le remplace par le **dernier élément** de l'arbre
   le plus à droite dans le niveau le plus profond
3. On fait descendre cet élément à sa place le cas échéant
   on échange avec le plus petit successeur

Tas binaire : extraction du minimum



1. Le minimum est **la racine** ;
2. On le remplace par le **dernier élément** de l'arbre
   le plus à droite dans le niveau le plus profond
3. On fait descendre cet élément à sa place le cas échéant
   on échange avec le plus petit successeur

Tas binaire : extraction du minimum

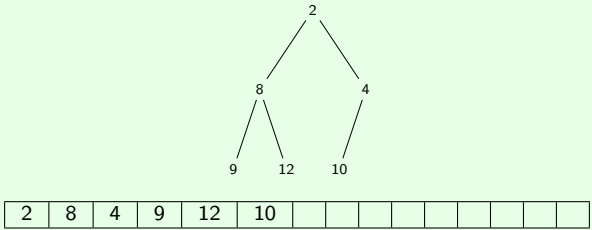


1. Le minimum est **la racine** ;
2. On le remplace par le **dernier élément** de l'arbre
   le plus à droite dans le niveau le plus profond
3. On fait descendre cet élément à sa place le cas échéant
   on échange avec le plus petit successeur

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple

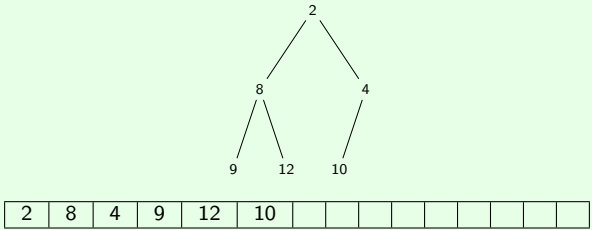


- Pour un indice i donné :
  - indice du nœud parent :

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple

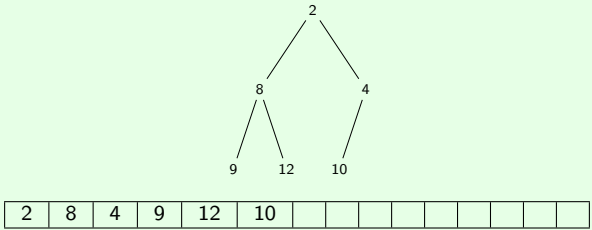


- Pour un indice i donné :
  - indice du nœud parent : $\lfloor \frac{i-1}{2} \rfloor$

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple

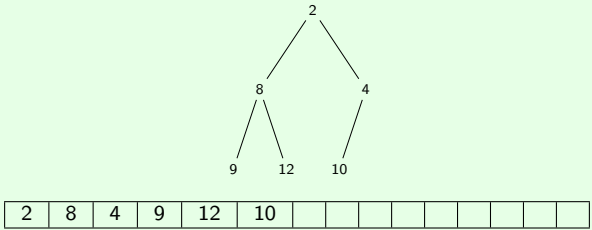


- Pour un indice i donné :
  - indice du nœud parent : $\lfloor \frac{i-1}{2} \rfloor$
  - indice du nœud successeur (fils) gauche : et droit :

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple

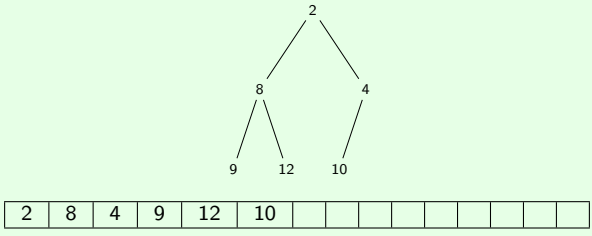


- Pour un indice i donné :
  - indice du nœud parent : $\lfloor \frac{i-1}{2} \rfloor$
  - indice du nœud successeur (fils) gauche : $2i + 1$ et droit :

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple

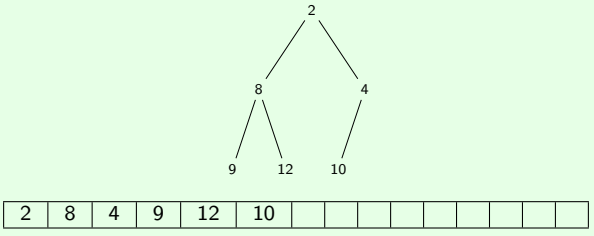


- Pour un indice i donné :
  - indice du nœud parent : $\lfloor \frac{i-1}{2} \rfloor$
  - indice du nœud successeur (fils) gauche : $2i+1$ et droit : $2i+2$

Tas implicites

- Si on a un accès en $O(1)$ au parent et aux successeurs, insertion et extraction sont en $O(\log_2 n)$;
- On a un arbre binaire où seule la « fin » manque :
- On peut facilement l'encoder **implicitement** dans un tableau :
  pour une taille maximale donnée (ici 15)

Exemple



- Pour un indice i donné :
  - indice du nœud parent : $\lfloor \frac{i-1}{2} \rfloor$
  - indice du nœud successeur (fils) gauche : $2i+1$ et droit : $2i+2$

Exercice

Écrire l'algorithme d'insertion pour le tas binaire implicite, avec la fonction `heap_fix_up(A, i)` qui fait monter l'élément d'indice i à sa place dans le tas binaire A.

Tas implicites

```
1 | def heap_insert(A, x, n):
2 |     if n == len(A):
3 |         error("heap is full")
4 |     else:
5 |         A[n] = x
6 |         n = n + 1
7 |
8 |         heap_fix_up(A, n - 1)
```

```
1 | def heap_fix_up(A, i):
2 |     p = ⌊(i-1)/2⌋
3 |     while p ≥ 0 and A[i] < A[p]:
4 |         swap(A, i, p)
5 |         i = p
6 |         p = ⌊(i-1)/2⌋
```

Tri par tas (heapsort)

- Par extraction successive du minimum, on peut trier efficacement un tableau de n éléments :
  1. insertion des n éléments : $n \times O(\log_2 n)$;
  2. extraction dans l'ordre des n éléments : $n \times O(\log_2 n)$
  3. au total $2n \times O(\log_2 n) = O(n \log_2 n)$
  4. mais il faut un tableau supplémentaire pour le tas.

Tri par tas (heapsort)

- Par extraction successive du minimum, on peut trier efficacement un tableau de n éléments :
  1. insertion des n éléments : $n \times O(\log_2 n)$;
  2. extraction dans l'ordre des n éléments : $n \times O(\log_2 n)$
  3. au total $2n \times O(\log_2 n) = O(n \log_2 n)$
  4. mais il faut un tableau supplémentaire pour le tas.
- On peut tout faire **en place**, dans le tableau initial :
  - on construit le tas de la droite vers la gauche :
    - la partie droite respecte l'invariant du tas
    - chaque nouvel élément descend suffisamment pour le respecter à son tour
  - on part du milieu, à l'indice $\lfloor \frac{n}{2} \rfloor - 1$:

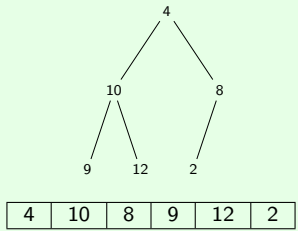
Lemme

Si A est un tas binaire implicite à n éléments alors les feuilles sont au indices $\lfloor \frac{n}{2} \rfloor \ldots (n-1)$.

- on construit un tas pour le **maximum** au lieu du minimum
- chaque maximum extrait est remis **à la place** du dernier élément du tas qui l'a remplacé.
- le coût est toujours $O(n \log_2 n)$, mieux que *merge sort* qui nécessite un peu de mémoire supplémentaire et *quick sort* qui est en $O(n^2)$.

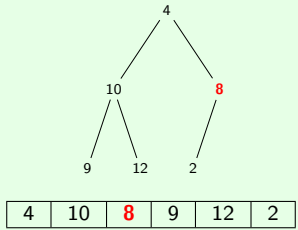
Tri par tas : exemple

Exemple



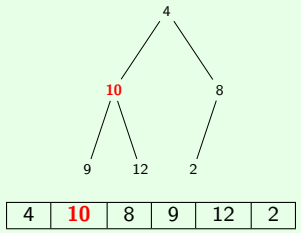
Tri par tas : exemple

Exemple



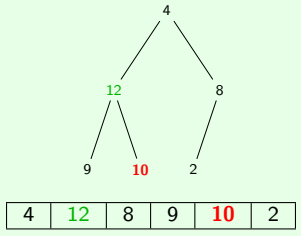
Tri par tas : exemple

Exemple



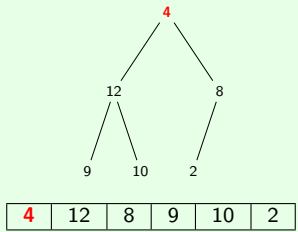
Tri par tas : exemple

Exemple



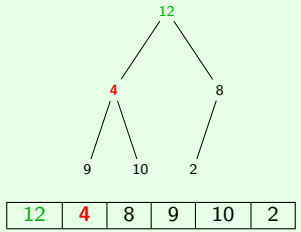
Tri par tas : exemple

Exemple



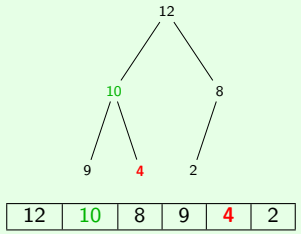
Tri par tas : exemple

Exemple



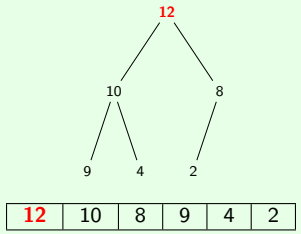
Tri par tas : exemple

Exemple



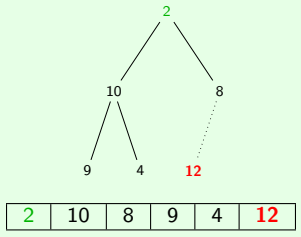
Tri par tas : exemple

Exemple



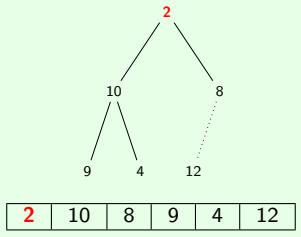
Tri par tas : exemple

Exemple



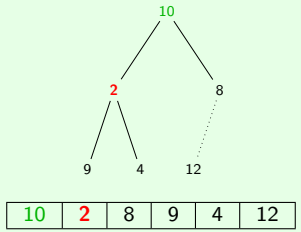
Tri par tas : exemple

Exemple



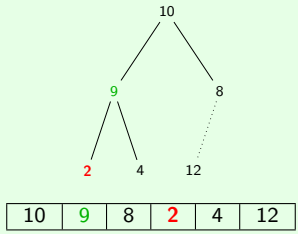
Tri par tas : exemple

Exemple



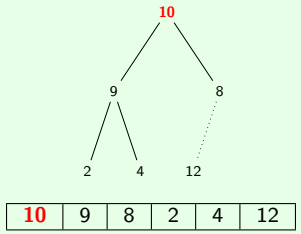
Tri par tas : exemple

Exemple



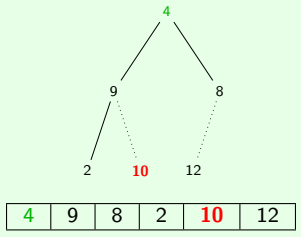
Tri par tas : exemple

Exemple



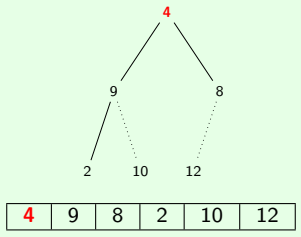
Tri par tas : exemple

Exemple



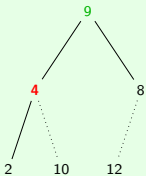
Tri par tas : exemple

Exemple



Tri par tas : exemple

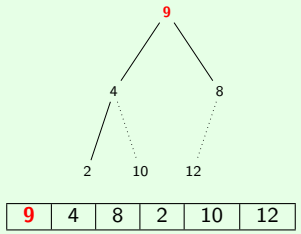
Exemple



9	4	8	2	10	12

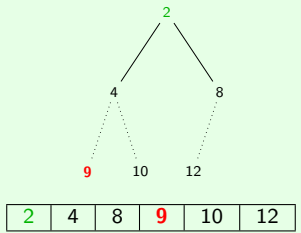
Tri par tas : exemple

Exemple



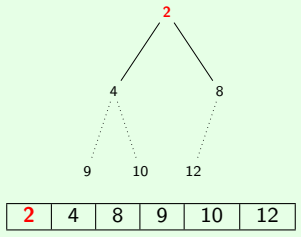
Tri par tas : exemple

Exemple



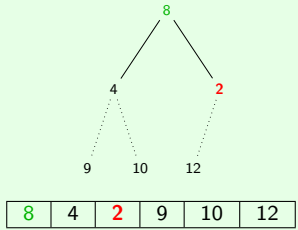
Tri par tas : exemple

Exemple



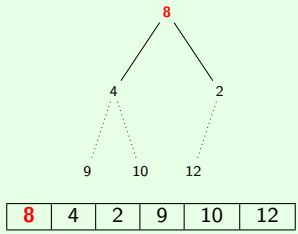
Tri par tas : exemple

Exemple



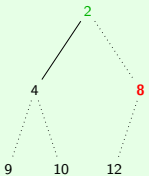
Tri par tas : exemple

Exemple



Tri par tas : exemple

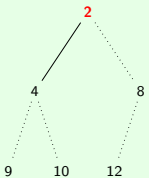
Exemple



2	4	8	9	10	12

Tri par tas : exemple

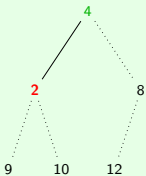
Exemple



2	4	8	9	10	12

Tri par tas : exemple

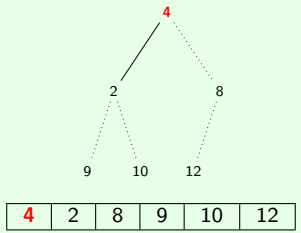
Exemple



4	2	8	9	10	12

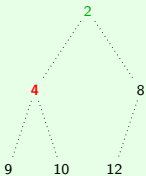
Tri par tas : exemple

Exemple



Tri par tas : exemple

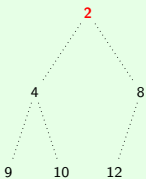
Exemple



2	4	8	9	10	12

Tri par tas : exemple

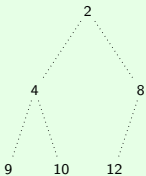
Exemple



2	4	8	9	10	12

Tri par tas : exemple

Exemple



2	4	8	9	10	12

Tri par tas

```
1 | def heapsort(A, n):
2 |     for i = ⌊n/2⌋ − 1 downto 0:
3 |         heap_fix_down_max(A, i, n) # variante max au lieu de min
4 |
5 |     for i = n − 1 to 1:
6 |         swap(A, 0, i)
7 |         heap_fix_down_max(A, 0, i + 1)
```

On écrira la fonction `heap_fix_down_max` en TP.

Tas et files de priorité : conclusion

- Le tas implicite est très simple à implémenter et possède de bonnes complexités ;

vide	$O(1)$
min	$O(1)$
extract_min	$O(\log_2 n)$
insertion	$O(\log_2 n)$

- C'est souvent le **plus efficace** en pratique ;
- Il existe aussi des implémentations de tas **explicites** :
  - Tas de Fibonacci (strict)
  - Tas binomial
  - Tas d'appariement
  - ...
- Elles ont de souvent de meilleures complexités **pire cas** ;
- Une autre opération utile est la **diminution de la priorité** d'un élément aussi $O(\log_2 n)$ pour le tas implicite. Comment ?

Outline

Ensembles et tableaux associatifs

- Dans de nombreux algorithmes, on veut ajouter, supprimer, et chercher rapidement des objets dans un **ensemble** (*set*) ;
  - On associe souvent des valeurs à ces objets, appelés alors **clés**
    $\Rightarrow$ **Tableau associatif** (dictionnaire, *map*, ...).
  - Un tableau associatif encode une **relation** : **ensemble** de couples ;
  - Un **multi-ensemble** est un tableau associatif où la valeur est un entier ;
  - Une **multimap** est un tableau associatif où la valeur est une liste ;

Ensembles et tableaux associatifs

- Dans de nombreux algorithmes, on veut ajouter, supprimer, et chercher rapidement des objets dans un **ensemble** (*set*) ;
  - On associe souvent des valeurs à ces objets, appelés alors **clés**
    $\Rightarrow$ **Tableau associatif** (dictionnaire, *map*, ...).
  - Un tableau associatif encode une **relation** : **ensemble** de couples ;
  - Un **multi-ensemble** est un tableau associatif où la valeur est un entier ;
  - Une **multimap** est un tableau associatif où la valeur est une liste ;
- Recherches et insertions répétées dans les listes/tableaux n'est pas très efficace ;

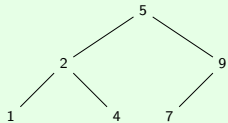
Ensembles et tableaux associatifs

- Dans de nombreux algorithmes, on veut ajouter, supprimer, et chercher rapidement des objets dans un **ensemble** (*set*) ;
  - On associe souvent des valeurs à ces objets, appelés alors **clés**
    $\Rightarrow$ **Tableau associatif** (dictionnaire, *map*, ...).
  - Un tableau associatif encode une **relation** : **ensemble** de couples ;
  - Un **multi-ensemble** est un tableau associatif où la valeur est un entier ;
  - Une **multimap** est un tableau associatif où la valeur est une liste ;
- Recherches et insertions répétées dans les listes/tableaux n'est pas très efficace ;
- Deux implémentations majeures pour un **ensemble** :
  1. ordonné : arbre (binaire) de recherche
  2. non ordonné : tables de hachage

Arbres binaires de recherche (ABR)

- Dans tout sous-arbre :
  1. la racine du fils gauche (si elle existe) est plus petite que la racine
  2. la racine du fils doit (si elle existe) est plus grande que la racine

Exemple



Arbres binaires de recherche : implémentation – recherche

- Pas d'encodage facile (stable par insertion) dans un tableau comme pour le tas binaire ;
- On encode donc l'ABR **explicitement** à la manière d'une liste chaînée.
  - Chaque nœud contient :
    - un champ r : la donnée liée à la racine de l'arbre
    - un pointeur g vers le (nœud racine du) sous-arbre gauche
    - un pointeur d vers le (nœud racine du) sous-arbre droit
  - L'arbre vide correspond au pointeur nul $\perp$
  - On ajoute aussi souvent un pointeur p vers le nœud parent ($\perp$ pour la racine).
- La recherche d'un élément dans un ABR à n éléments est en O(hauteur)

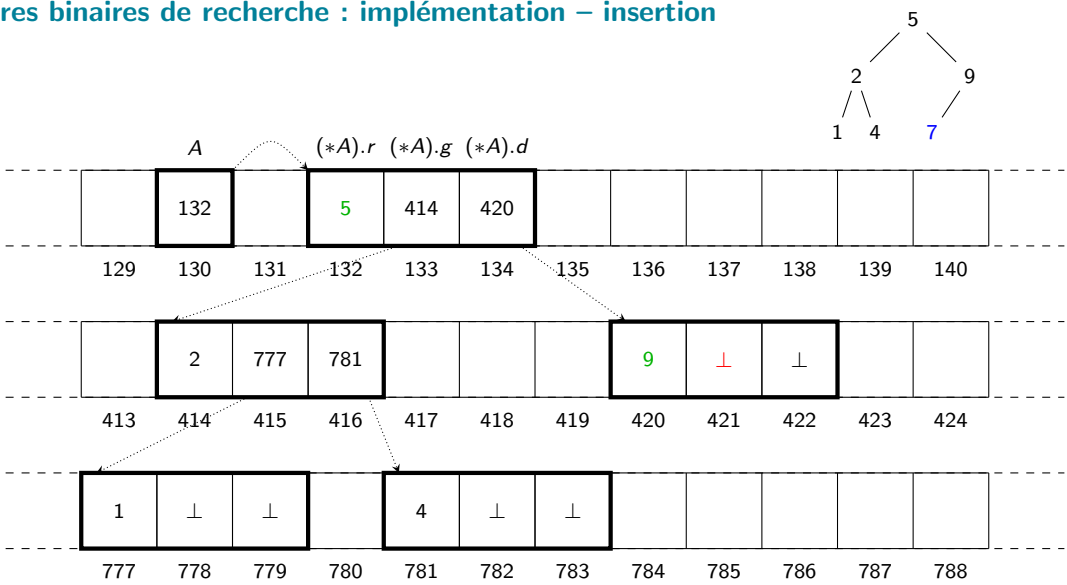
```
1  def abr_search(A, x):
2      if A == ⊥:
3          return false
4      else if x == (*A).r:
5          return true
6      else if x < (*A).r:
7          return abr_search((*A).g, x)
8      else:
9          return abr_search((*A).d, x)
```

Arbres binaires de recherche : implémentation – insertion

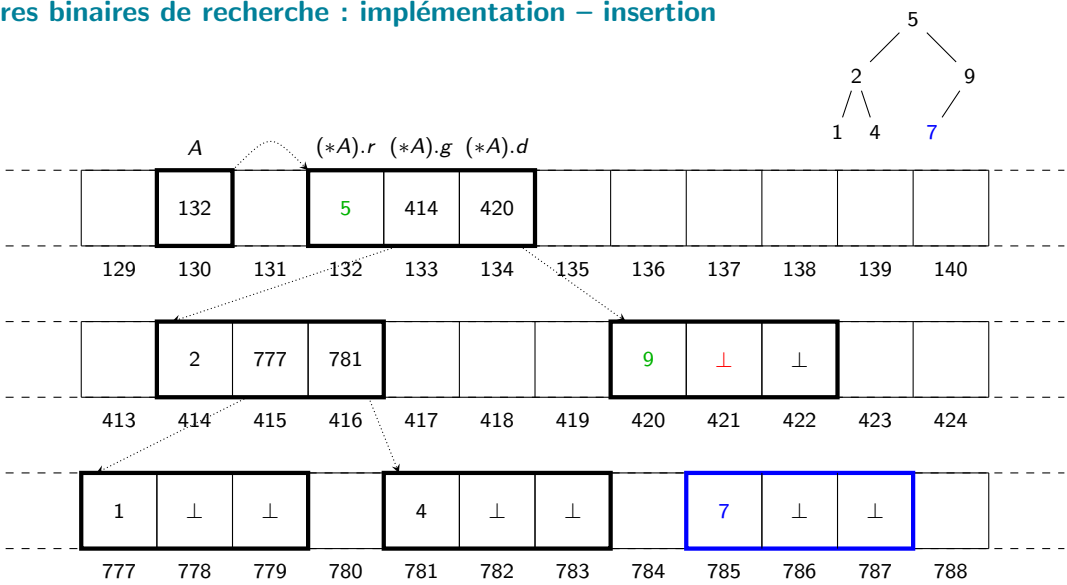
- On insère un élément comme une feuille en maintenant l'invariant

```
1  def abr_insert (A, x):
2      if A == ⊥:
3          # on suppose le type node bien défini
4          B = memory_alloc(node)
5          (*B).r = x
6          (*B).g = ⊥
7          (*B).d = ⊥
8
9          return B
10     else :
11         if x < (*A).r:
12             (*A).g = abr_insert ((*A).g, x)
13         else :
14             (*A).d = abr_insert ((*A).d, x)
15
16         return A
```

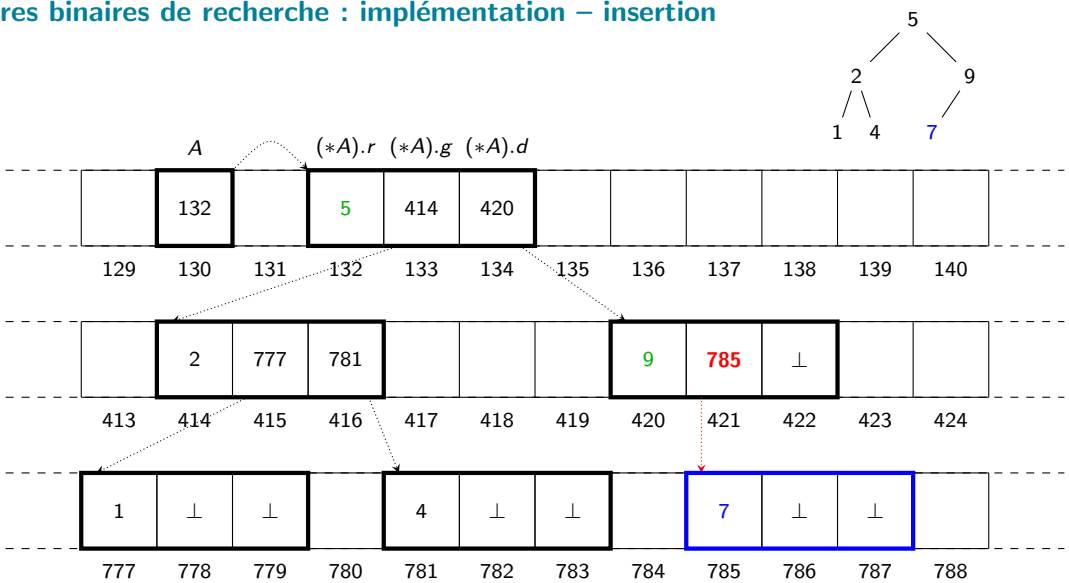
Arbres binaires de recherche : implémentation – insertion



Arbres binaires de recherche : implémentation – insertion



Arbres binaires de recherche : implémentation – insertion



Arbres binaires de recherche : implémentation – suppression

- Quand le nœud `N` à supprimer a zéro ou un successeur, la suppression est facile
- Sinon, remplacer `N` par la plus petit nœud `Y` plus grand que lui (ou le plus grand plus petit)
- Deux possibilités :
  - remplacer `(*N).r` par la valeur `(*Y).r` et supprimer cette valeur dans le sous-arbre correspondant
  - remplacer `N` par `Y` et recoller les morceaux soigneusement.

Exercice

1. Écrire un algorithme pour trouver le nœud réalisant le minimum dans un ABR ;
2. Écrire un algorithme pour la suppression d'un élément dans un ABR par la première méthode ;
3. Écrire un algorithme pour trouver le parent du nœud réalisant le minimum dans un ABR ;
4. Écrire un algorithme pour la suppression d'un élément dans un ABR par la deuxième méthode.

Arbres auto-équilibrant

- Les opérations sur les arbres sont efficaces si l'arbre est **équilibré** ;
- Après une insertion, il ne l'est plus forcément
  après insertion de 1,2,3,4,5... l'arbre est une liste
- On ajoute un mécanisme pour équilibrer l'arbre à chaque insertion ou suppression :
  - **arbres AVL** ;
  - arbres 2-3
  - arbres rouges et noirs
  - *splay trees*
  - ...

Arbre AVL

- Un arbre AVL (Adelson-Velsky & Landis) est un ABR dans lequel :
  Pour tout nœud x, le sous-arbre enraciné en x est équilibré (en hauteur).
- Opérations d'équilibrage par **rotation** à l'insertion et la suppression ;
- On ajoute à l'ABR :
  - 2 bits `b` pour mémoriser le **facteur d'équilibre** : `A.b = hauteur(A.d) - hauteur(A.g)`
  - `A.b` sera toujours compris entre -2 et 2.

Lemme

Un arbre équilibré en hauteur avec n nœuds a une hauteur $O(\log_2 n)$.

Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

1

Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

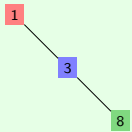
Séquence d'insertions : $1, 3, 8, 6, 5, 4$

Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

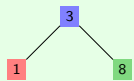


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

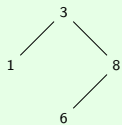


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

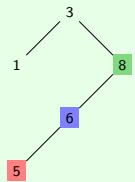


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

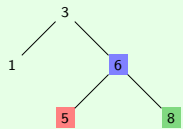


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

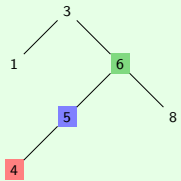


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

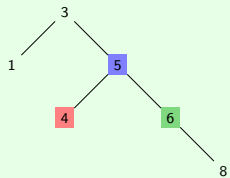


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

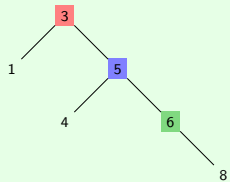


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

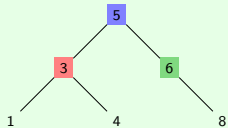


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

Séquence d'insertions : $1, 3, 8, 6, 5, 4$

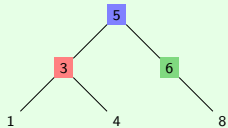


Arbres AVL : rotations

- Pour équilibrer l'arbre on va faire des **rotations** qui préservent l'invariant de l'ABR

Exemple

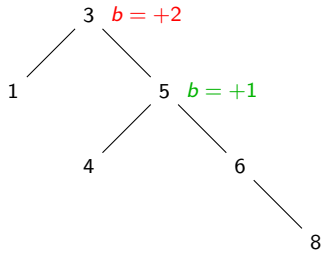
Séquence d'insertions : $1, 3, 8, 6, 5, 4$

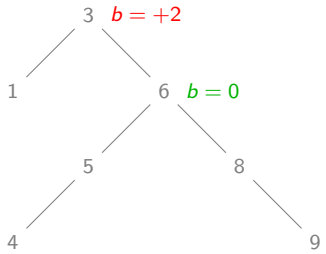


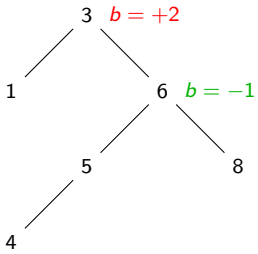
Exercice

Écrire l'algorithme de la rotation à gauche : le fils droit remplace la racine. Renvoyer la nouvelle racine.
Ne pas oublier le champ `b`.

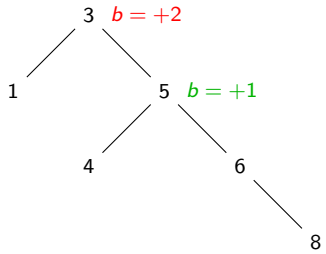
Arbres AVL : rotations à gauches

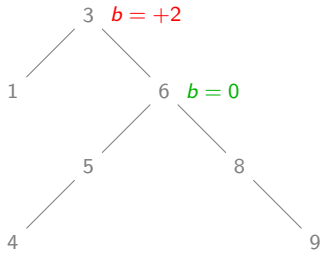


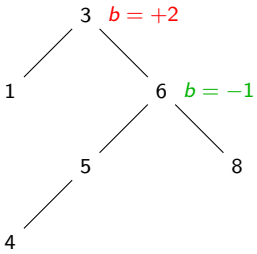


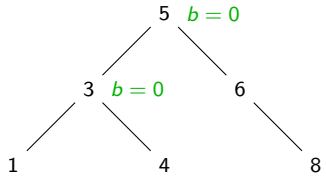


Arbres AVL : rotations à gauches

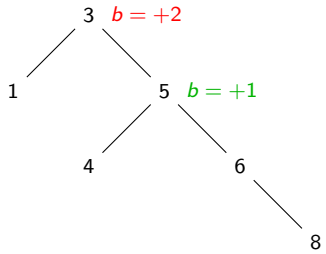


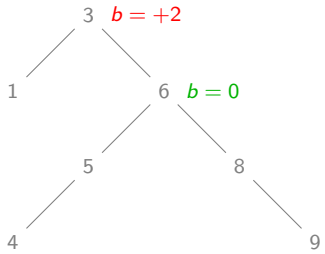


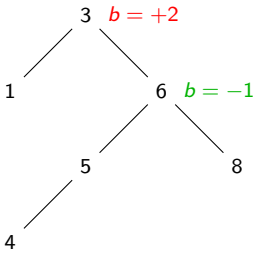


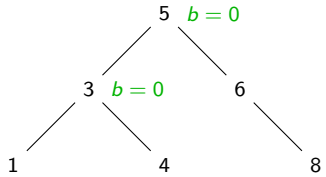


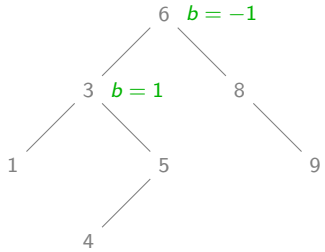
Arbres AVL : rotations à gauches



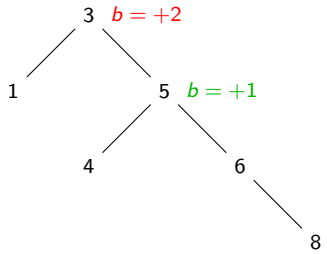


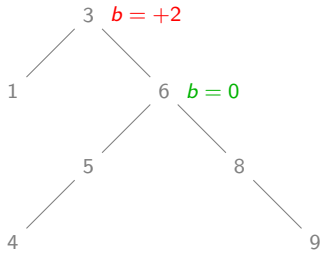


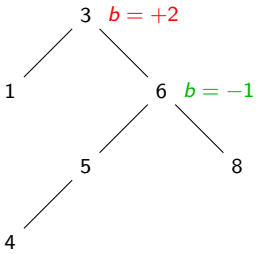


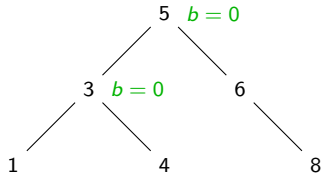


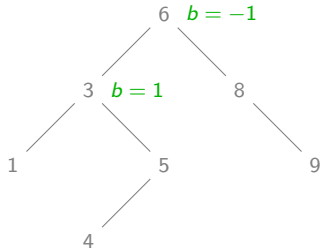
Arbres AVL : rotations à gauches

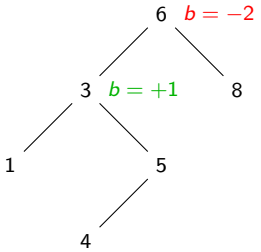




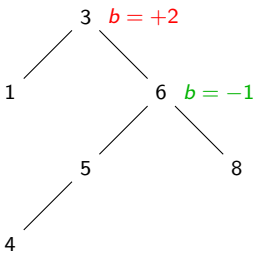




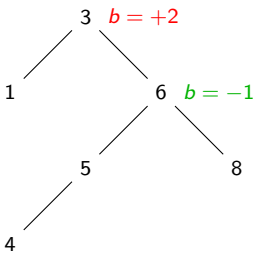


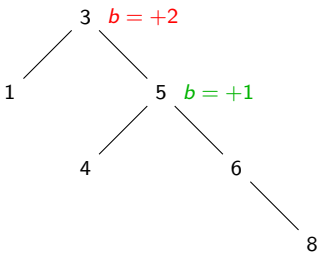


Arbres AVL : double rotation

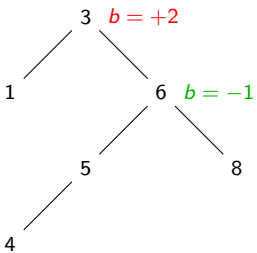


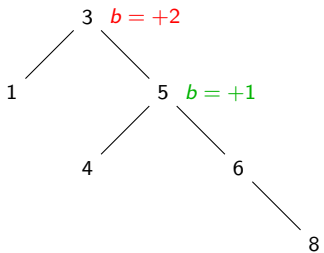
Arbres AVL : double rotation

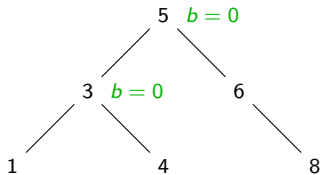




Arbres AVL : double rotation







Arbres AVL : insertion

```
1  def avl_balance(A):
2      R = A
3      if A ≠ ⊥:
4          if (*A).b == 2:
5              if (*(*A).d).b < 0:
6                  (*A).d = avl_rotate_right ((*A).d)
7
8              R = avl_rotate_left (A)
9          else if (*A).b == −2:
10             if (*(*A).g).b > 0:
11                 (*A).g = avl_rotate_left ((*A).g)
12
13             R = avl_rotate_right (A)
14
15     return R
```

Lemme

Soit A un arbre AVL de taille n. Après insertion de x, la hauteur de A est augmentée de 1 si et seulement si le facteur d'équilibrage de tous les ancêtres de x est non nul 0.

```
1  def avl_insert (A, x, p, bfu): # init. avl_insert(A, x, ⊥, 0)
2      retrace = (p ≠ ⊥)
3
4      if A == ⊥:
5          B = memory_alloc(node) # type node défini ailleurs
6          (*B).r = x; (*B).g = ⊥; (*B).d = ⊥
7          (*B).b = 0;
8      else:
9          if x < (*A).r:
10             (*A).g = avl_insert ((*A).g, x, A, −1)
11         else:
12             (*A).d = avl_insert ((*A).d, x, A, +1)
13
14         B = avl_balance(A)
15         if (*B).b == 0:
16             retrace = False
17
18     if retrace:
19         (*p).b = (*p).b + bfu
20
21     return B
```

Arbres AVL : conclusion

- Dans l'insertion, dès qu'on a un équilibrage, la propriété AVL est globalement satisfaite ;
- On a donc **au plus 2 rotations** pour chaque insertion
  - on doit de toute façon descendre tout le chemin jusqu'au point d'insertion.
  - et le rééquilibrage peut avoir lieu dans la racine (remontée de tout le chemin).
- Pour la **suppression**, on fonctionne de façon similaire à l'ABR et à l'insertion ;
- Le cas grisé précédemment peut se produire et le nombre maximum de rotations n'est que $O(\log_2 n)$.
- On peut aussi définir des opérations efficaces ($O(n_1 \log(\frac{n_1}{n_2} + 1))$) et parallélisables d'union, intersection, différence d'ensemble.

recherche	$O(\log_2 n)$
insertion	$O(\log_2 n)$
supression	$O(\log_2 n)$

Arbres AVL : conclusion

- Dans l'insertion, dès qu'on a un équilibrage, la propriété AVL est globalement satisfaite ;
- On a donc **au plus 2 rotations** pour chaque insertion
  - on doit de toute façon descendre tout le chemin jusqu'au point d'insertion.
  - et le rééquilibrage peut avoir lieu dans la racine (remontée de tout le chemin).
- Pour la **suppression**, on fonctionne de façon similaire à l'ABR et à l'insertion ;
- Le cas grisé précédemment peut se produire et le nombre maximum de rotations n'est que $O(\log_2 n)$.
- On peut aussi définir des opérations efficaces ($O(n_1 \log(\frac{n_1}{n_2} + 1))$) et parallélisables d'union, intersection, différence d'ensemble.

recherche	$O(\log_2 n)$
insertion	$O(\log_2 n)$
supression	$O(\log_2 n)$

Exercice

En supposant qu'on n'ait besoin que de l'union, l'intersection, la différence d'ensembles et la recherche d'un élément, proposer une structure de données et des algorithmes pour réaliser les opérations ensemblistes en $O(n_1 + n_2)$ et la recherche en $O(\log_2 n)$.

Tables de hachage

- L'implémentation la plus efficace d'un ensemble est un **tableau de taille** m si :
  1. on a une **injection** i des objets vers $[0..m-1]$,
  2. i est facile à calculer ($O(1)$),
  3. m n'est pas trop grand.
- On met chaque objet o (ou un pointeur) à l'indice $i(o)$;
- On utilise une valeur spéciale (p. ex. $\perp$) pour indiquer l'absence d'un élément ;
- Insertion, suppression, recherche sont en $O(1)$.

Tables de hachage

- L'implémentation la plus efficace d'un ensemble est un **tableau de taille** m si :
  1. on a une **injection** i des objets vers $[0..m-1]$,
  2. i est facile à calculer ($O(1)$),
  3. m n'est pas trop grand.
- On met chaque objet o (ou un pointeur) à l'indice $i(o)$;
- On utilise une valeur spéciale (p. ex. $\perp$) pour indiquer l'absence d'un élément ;
- Insertion, suppression, recherche sont en $O(1)$.

Exercice

Chemins. Soit M une matrice de taille $m \times n$ contenant des 0 et des 1. On s'intéresse aux chemins qui vont de la case $(0,0)$ à la case $(m-1, n-1)$ en ne passant que par des 1 et en ne se déplaçant que sur une case immédiatement à droite ou en bas.

En s'inspirant de la programmation dynamique, donner un algorithme non-récursif qui calcule le nombre de tels chemins.

Tables de hachage

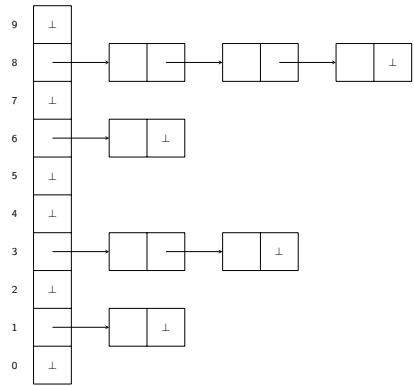
- Il est ≪ facile ≫ d'avoir une fonction h respectant ces contraintes **sauf l'injectivité** ;
- h est une **fonction de hachage** ;
- On peut appliquer la même idée que précédemment mais il faut alors gérer les **collisions** :
  - en mémorisant l'**ensemble** des éléments de **même valeur** h dans la case du tableau correspondante
    ⇒ **Chaînage**
  - ou en trouvant une autre place dans le tableau
    ⇒ **Addressage ouvert**

Tables de hachage : chaînage

- Dans la case d'indice i du tableau, on mémorise l'**ensemble** des éléments o tels que $h(o) = i$
- La taille de ce sous-ensemble est normalement $\ll$ petite $\gg$;
- On le représente historiquement par une **liste chaînée**.

Tables de hachage : chaînage

- Dans la case d'indice i du tableau, on mémorise l'**ensemble** des éléments o tels que $h(o) = i$
- La taille de ce sous-ensemble est normalement $\ll$ petite $\gg$;
- On le représente historiquement par une **liste chaînée**.



Tables de hachage : chaînage

```
1  def ht_search(T, x):
2      return list_search(T[h(x)], x)
3
4  def ht_insert(T, x):
5      T[h(x)] = list_insert_front(T[h(x)])
6
7  def ht_delete(T, x):
8      list_delete(T[h(x)], x)
```

Exercice

Quelle est la complexité pire cas de ces opérations ?

Tables de hachage : chaînage – complexité

- La complexité de l'utilisation de la table dépend de celle de la **recherche**. Pour la suppression :
  1. **Recherche** d'un pointeur sur la cellule contenant x (ou son parent si la liste est simplement chaînée) ;
  2. Suppression en $O(1)$ de la cellule en utilisant ce pointeur
- La complexité **pire cas** ne dépend pas de la taille m de la table
- La complexité **moyenne** dépend de m et de la **qualité** de la fonction de hachage :
  - **indépendance** : $\forall x, yP(h(x)|h(y)) = P(h(x))$
  - **uniformité** : $\forall x, P(h(x)) = \frac{1}{m}$

Exercice

Une fonction de hachage à valeurs dans $[0..m-1]$ est **universelle** si $P(h(x) = h(y)) = \frac{1}{m}$.

1. Montrer que si h est indépendante et uniforme alors elle est universelle.
2. En supposant h universelle et qu'on a n éléments dans notre ensemble, quelle est l'espérance du nombre de collisions ?
3. Combien de personnes ont leur anniversaire le même jour dans la classe ?

Tables de hachage : chaînage – complexité moyenne

- Une clé x qui n'est pas dans la table n'est pas dans liste $T[h(x)]$ qui peut être vide ou non ;

Exercice

On suppose que h est indépendante et uniforme. Soient m la taille du tableau et n le nombre d'éléments déjà dans T.

1. Montrer que la longueur moyenne (espérance) de la liste $T[h(x)]$ est $\frac{n}{m}$;
2. En déduire que la complexité moyenne d'une recherche **infructueuse** dans T est $O(1 + \frac{n}{m})$

Tables de hachage : chaînage – complexité moyenne

- Une clé x qui n'est pas dans la table n'est pas dans liste $T[h(x)]$ qui peut être vide ou non ;

Exercice

On suppose que h est indépendante et uniforme. Soient m la taille du tableau et n le nombre d'éléments déjà dans T.

1. Montrer que la longueur moyenne (espérance) de la liste $T[h(x)]$ est $\frac{n}{m}$;
2. En déduire que la complexité moyenne d'une recherche **infructueuse** dans T est $O(1 + \frac{n}{m})$

- Une clé x qui est dans la table appartient forcément à une liste $T[h(x)]$ **non vide**
  donc à priori de longueur $\geq \frac{n}{m}$. On suppose que toutes clés ont la même chance d'être cherchées.

Exercice

On rappelle que trouver un élément dans une liste de longueur ℓ qui le contient requiert en moyenne $\frac{\ell}{2}$ comparaisons.

1. Quelle est la probabilité qu'une liste donnée soit vide ? non-vide ?
2. Quelle est la probabilité que le i^{e} des n éléments ait été ajouté à $T[h(x)]$ sachant que cette liste est non-vide ?
3. Quelle est la longueur moyenne d'une liste non-vide ?
4. Montrer qu'il existe n_0 tel que pour tout $n \geq n_0$, la probabilité qu'une liste donnée soit vide est $\leq \frac{n}{2}$
5. Conclure en montrant que la complexité moyenne d'une recherche **fructueuse** dans T est aussi $O(1 + \frac{n}{m})$

Tables de hachage : chaînage – complexité moyenne

- Une clé x qui n'est pas dans la table n'est pas dans liste $T[h(x)]$ qui peut être vide ou non ;

Exercice

On suppose que h est indépendante et uniforme. Soient m la taille du tableau et n le nombre d'éléments déjà dans T.

1. Montrer que la longueur moyenne (espérance) de la liste $T[h(x)]$ est $\frac{n}{m}$;
2. En déduire que la complexité moyenne d'une recherche **infructueuse** dans T est $O(1 + \frac{n}{m})$

- Une clé x qui est dans la table appartient forcément à une liste $T[h(x)]$ **non vide**
  donc à priori de longueur $\geq \frac{n}{m}$. On suppose que toutes clés ont la même chance d'être cherchées.

Exercice

On rappelle que trouver un élément dans une liste de longueur ℓ qui le contient requiert en moyenne $\frac{\ell}{2}$ comparaisons.

1. Quelle est la probabilité qu'une liste donnée soit vide ? non-vide ?
2. Quelle est la probabilité que le i^e des n éléments ait été ajouté à $T[h(x)]$ sachant que cette liste est non-vide ?
3. Quelle est la longueur moyenne d'une liste non-vide ?
4. Montrer qu'il existe n_0 tel que pour tout $n \geq n_0$, la probabilité qu'une liste donnée soit vide est $\leq \frac{n}{2}$
5. Conclure en montrant que la complexité moyenne d'une recherche **fructueuse** dans T est aussi $O(1 + \frac{n}{m})$

- Donc la complexité de la recherche est en $O(1 + \frac{n}{m})$

Tables de hachage : chaînage – complexité moyenne

- Une clé x qui n'est pas dans la table n'est pas dans liste $T[h(x)]$ qui peut être vide ou non ;

Exercice

On suppose que h est indépendante et uniforme. Soient m la taille du tableau et n le nombre d'éléments déjà dans T.

1. Montrer que la longueur moyenne (espérance) de la liste $T[h(x)]$ est $\frac{n}{m}$;
2. En déduire que la complexité moyenne d'une recherche **infructueuse** dans T est $O(1+\frac{n}{m})$

- Une clé x qui est dans la table appartient forcément à une liste $T[h(x)]$ **non vide**
  donc à priori de longueur $\geq \frac{n}{m}$. On suppose que toutes clés ont la même chance d'être cherchées.

Exercice

On rappelle que trouver un élément dans une liste de longueur ℓ qui le contient requiert en moyenne $\frac{\ell}{2}$ comparaisons.

1. Quelle est la probabilité qu'une liste donnée soit vide ? non-vide ?
2. Quelle est la probabilité que le i^{e} des n éléments ait été ajouté à $T[h(x)]$ sachant que cette liste est non-vide ?
3. Quelle est la longueur moyenne d'une liste non-vide ?
4. Montrer qu'il existe n_0 tel que pour tout $n \geq n_0$, la probabilité qu'une liste donnée soit vide est $\leq \frac{n}{2}$
5. Conclure en montrant que la complexité moyenne d'une recherche **fructueuse** dans T est aussi $O(1+\frac{n}{m})$

- Donc la complexité de la recherche est en $O(1+\frac{n}{m})$
- Et si $n = O(m)$ (ou encore $m = \Omega(n)$), alors la recherche est en $O(1)$
  mais attention à la valeur du coefficient constant caché !

Tables de hachage – chaînage

- Si on ne peut pas avoir m assez grand, on peut remplacer la liste chaînée par un ABR ;
- On peut aussi utiliser un vecteur pour maximiser la **localité spatiale** pour les caches ;
- Le choix d'une bonne fonction de hachage est crucial :
  - bonnes propriétés aléatoires (`mod m` n'est pas terrible)
  - rapide à calculer
- Il vaut mieux (comme pour la cryptographie) ne pas réinventer la roue.

Tables de hachage – adressage ouvert

- Plutôt que le chaînage, on peut mémoriser tous les éléments dans la table ;
- La fonction de hachage doit nous donner une **permutation** aléatoire de $[0..m-1]$

$$h : X \times [0..m-1] \to [0..m-1]$$

- $h(x, i)$ représente le i^{e} élément de la permutation
- La permutation donne, par ordre de préférence, les emplacements où la clé peut se trouver

```
1  def htoa_search(T, x):
2      i = 0
3      while i < m and T[h(x, i)] ≠ ⊥ and T[h(x, i)] ≠ x:
4          i = i + 1
5
6      return (i < m and T[h(x, i)] == x)
```

```
1  def htoa_insert(T, x):
2      i = 0
3      while i < m and T[h(x, i)] ≠ ⊥:
4          i = i + 1
5
6      if T[k] == ⊥:
7          T[k] = x
8      else:
9          error("Table pleine")
```

Exercice

Écrire l'algorithme pour la suppression d'un élément. On pourra modifier l'insertion et la recherche si nécessaire.

Tables de hachage – adressage ouvert

- Chaque opération consiste à sonder (*probe*) plusieurs emplacements ;
- Le nombre de sondages domine la complexité ;
- Si h fournit de manière **indépendante** une **permutation** aléatoire **uniforme**, on n'a pas de suppressions, et $\alpha = \frac{n}{m} < 1$:
  - le nombre moyen de sondages pour une recherche infructueuse est au plus $\frac{1}{1-\alpha}$
  - le nombre moyen de sondages pour une recherche fructueuse est au plus $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- Mais trouver une fonction h avec ces propriété est compliqué
- Approximations :
  - Sondage linéaire (*Linear probing*)
  - Hachage double (*Double hashing*)

Tables de hachage – adressage ouvert : sondage linéaire

- Soit une fonction de hachage classique h_1
- Pour tout x, on définit $h(x, i) = h_1(x) + i$
- **Avantages** :
  - très simple
  - bonnes performances de cache
- **Inconvénients** :
  - engendre une seule des $m!$ permutations possibles
  - peut créer des *clusters* qui dégradent la performance

Tables de hachage – adressage ouvert : hachage double

- Soient deux fonctions de hachage classiques indépendantes h_1 et h_2 avec $h_2(x)$ premier avec m pour tout x par exemple m premier, ou $m = 2^k$ et $h_2(x)$ impair pour tout x
- Pour tout x, on définit $h(x, i) = h_1(x) + ih_2(x) \mathrm{mod} m$
- **Avantages** :
  - raisonnablement simple
  - engendre m^2 des $m!$ permutations possibles
  - préserve la borne supérieure sur le nombre de sondages
- **Inconvénients** :
  - h_2 doit être très rapide à calculer
  - ne tient pas compte des caches

Tables de hachage – adressage ouvert

- L'adressage ouvert utilise mieux la mémoire et le cache que le chaînage :
  - Pas de pointeurs
  - Localité spatiale
- Mais ses performances décroissent rapidement quand $\alpha > 0.7$;
- On a forcément $\alpha \leq 1$;
- La suppression est compliquée (contournable pour le sondage linéaire).

Outline

Conclusion générale

- Comprendre et évaluer le comportement des algorithmes :
  - Propriétés fonctionnelles
  - Propriétés non fonctionnelles : terminaison, accès aux tableaux, blocages...
  - Complexité
- Choisir la $\ll$ bonne $\gg$ structure de données selon les opérations requises
- Utiliser (si possible) un paradigme classique de résolution de problèmes algorithmiques
- Pour aller plus loin :
  - complexité théorique
  - structures de données : *skip list*, arbres de préfixes (*tries*), *splay trees*, arbres rouges et noirs, tas d'appariement, de Fibonacci, ...
  - algorithmes en ligne : traitement à la volée de flux de données
  - algorithmes d'approximation
  - algorithmes parallèles
  - ...

References


Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorihthms (4th edition), The MIT Press, 2022.