Chapitre 2 : Méthodes de programmation et analyses d'algorithmes

Thomas MEGARBANE

PCSI

Plan

Plan

Spécification des données

Définition

*On appelle **signature** d'un algorithme (ou d'une fonction) la donnée de :*

- *son **nom***
- *ses **arguments** (ou **entrées**) et leurs types*
- *ses **valeurs renvoyées** (ou **sorties**) et leurs types*

Remarque

*On peut rajouter des **spécifications** :*

- *des **pré-conditions** : conditions à vérifier sur les entrées*
- *des **post-conditions** : conditions à vérifier sur les sorties*

avec la commande `assert`, et qui, si elles ne sont pas vérifiées, arrêtent le programme, et sont donc à éviter.

Spécification des données

Exemple

Pour l'algorithme de division euclidienne en n'utilisant que les opérations `+` et `-` :

- *nom :* `Divisioneuclidienne`
- *entrées : deux entiers* `a` *et* `b` *(type* `int`*)*
- *sorties : couple* `(q,r)` *(type* `tuple`*)*
- *pré-conditions :* `b` *non nul ;* `a` *et* `b` *entiers naturels*
- *post-conditions :* `q` *et* `r` *entiers naturels et* `0<= r<b`

```
1  def Divisioneuclidienne(a:int,b:int) -> tuple :
2      assert (a>=0) and (type(a)==int)
3      assert (b>0) and (type(b)==int)
4      q=0
5      r=a
6      while r>=b :
7          q+=1
8          r-=b
9      assert (q>=0) and (type(q)==int)
0      assert (r>=0) and (type(r)==int) and (r<b)
1      return (q,r)
```

Spécification des données

Remarque

L'utilisation des pré-conditions permet de simplifier un programme, en limitant les valeurs possibles sur les entrées.

Exemple

Pour coder si un nombre entier est pair, on peut écrire :

```
1 def estpair(n:int) -> bool :
2     assert (n>=0) and (n==int(n))
3     if n==0 :
4         return True
5     return not estpair(n-1)
```

```
1 def estpair(n:int) -> bool :
2     assert (n==int(n))
3     if n==0 :
4         return True
5     if n>0 :
6         return not estpair(n-1)
7     return not estpair(n+1)
```

Ou écrire un programme qui tournerait aussi avec des flottants :

```
1 def estpair(n:float) -> bool :
2     if n>=0 :
3         if n<2 :
4             return (n==0)
5         return estpair(n-2)
6     return estpair(-n)
```

Terminaison et correction d'un programme

Objectifs

*Un programme **doit** remplir les critères suivants :*

- ***clarté** : le programme doit être compréhensible facilement, soit parce que la stratégie mise en œuvre est claire, soit parce qu'elle est clairement expliquée*
- ***terminaison** : le programme, quelles que soient ses entrées (valides) s'arrête en un temps fini (éventuellement très long)*
- ***correction (partielle)** : lorsqu'il termine, les valeurs des sorties sont bien celles imposées*

Définition

*Étant donné un programme, on parlera de **correction partielle** si, quand il termine, il est correct. On parlera de **correction totale** si de plus il termine toujours.*

Remarque

*La clarté s'obtient parfois au prix de **commentaires**, qui ne devront pas être systématiques : il est mieux d'écrire un programme sans commentaires mais compréhensible, que de pallier un manque de clarté par des commentaires.*
La correction (partielle ou totale) se démontre !

Terminaison : cas des boucles

Remarques

- *Quand on parcourt une liste avec une boucle `for`, on veille à ne pas modifier la liste pendant le parcours : la terminaison est alors toujours acquise. La correction est en revanche plus subtile à prouver.*
- *Quand on écrit une boucle `while`, la terminaison n'est pas systématique : on la montre en général par un raisonnement par l'absurde. La correction est alors plus facile, et part de la condition qui n'est plus vérifiée. La situation avec un programme récursif est très proche.*

Exemples

Les programmes suivants plantent :

```
1 def plantage_for1(L) :
2     for l in L :
3         L.append(0)
4     return True
```

```
1 def plantage_for2(L) :
2     for i in range(len(L)) :
3         if L[i]%2==0 :
4             L.pop()
```

à l'inverse de la division euclidienne codée précédemment, qui termine toujours.

```
1 def Divisioneuclidienne(a,b) :
2     q,r=0,a
3     while r>=b :
4         q+=1
5         r-=b
6     return (q,r)
```

Par l'absurde si ce n'était pas le cas, les valeurs successives de `r` vérifieraient toujours la condition `r>=b`. Mais cette suite des valeurs est définie par : $r_0 = a$ et $\forall n \in \mathbb{N}$, $r_{n+1} = r_n - b$ qui tend vers $-\infty$, donc n'est pas minorée. Contradiction, donc le programme termine.

Variants et invariants

Définition

On considère un programme construit par itérations d'un processus (sous forme de boucle ou de manière récursive). On dira qu'une quantité est :

- *un **invariant (de boucle)** si une quantité est préservée à chaque itération ;*
- *un **variant (de boucle)** si une quantité qui change de manière maîtrisée à chaque itération.*

Remarque

En pratique, on travaillera avec :

- *pour invariant : une propriété vraie. Le fait que ce soit un invariant veut dire que, si elle est vraie au départ du processus, elle reste vraie. Elle sert à montrer la **correction (partielle)**.*
- *pour variant : une suite strictement décroissante d'entiers naturels. Cette suite prend alors un nombre fini de valeurs, et sert à montrer la **terminaison**.*

Proposition

Une boucle `for` dans laquelle on ne modifie pas la liste parcourue termine toujours.

Démonstration.

On choisit comme variant le nombre d'éléments qu'il reste à parcourir : comme la liste est inchangée, le variant diminue de 1 à chaque étape, donc la boucle termine toujours. □

Variants et invariants

Exemple

On reprend le code de la division euclidienne suivant :

```
1 def Divisioneuclidienne(a,b) :
2     q,r=0,a
3     while r>=b :
4         q+=1
5         r-=b
6     return (q,r)
```

Montrons la correction :

- *on a déjà vu que r est un variant, ce qui assure la terminaison ;*
- *montrons que la quantité "$b \times q + r$" est un invariant : si on effectue l'opération de la boucle, alors q devient $q+1$ et r devient $r-b$ donc $b \times q + r$ devient :*
  $b \times (q+1) + (r-b) = b \times q + r$. On a bien un invariant.

À l'initialisation des variables, $q = 0$ et $r = a$ donc $b \times q + r = a$.
Par construction, r et q sont des entiers naturels, et en sortie de boucle la condition `r>=b` n'est plus vérifiée donc $r < b$. Et ainsi en sortie de boucle on a :

$$a = bq + r,\ q, r \in \mathbb{N},\ r < b$$

ce qui est bien la définition de la division euclidienne.

Variants et invariants

Exemple

On cherche à savoir ce que fait le code suivant :

```
1 def test1(a:int,b:int)-> int :
2     assert (a>=0) and (b>=0)
3     if a*b == 0 :
4         return a+b
5     if a>b :
6         return test1(a-b,b)
7     return test1 (a,b-a)
```

Terminaison :
Ce programme termine si on arrive à un moment où $a = 0$ ou $b = 0$. Par l'absurde, si ce n'est pas le cas, on a toujours $a, b \in \mathbb{N}^*$. Mais la quantité $a + b$ diminue strictement (de a ou b selon les cas). D'où la contradiction.
Correction partielle :
Le pgcd est conservé à chaque étape comme $a \wedge b = (a - b) \wedge b = a \wedge (b - a)$. Et c'est donc un invariant. En sortie, on a $a = 0$ ou $b = 0$. Mais $a \wedge 0 = a = a + 0$ et $0 \wedge b = b = 0 + b$. Et donc si $a = 0$ ou $b = 0$ la fonction rend le pgcd de a et b.
Correction totale : *Le programme prend en argument deux entiers naturels et rend leur pgcd. Il termine toujours.*

Variants et invariants

Exemple

On cherche à savoir ce que font les codes suivants :

```
1  def test2(a:int,b:int)-> int :
2      assert (a>=0) and (b>=0)
3      x,y,p=a,b,0
4      while x!=0 :
5          if x%2==1 :
6              p+=y
7              x-=1
8          x=x//2
9          y=y*2
10     return p
```

```
1  def test3(n:int)-> int :
2      assert (n>=0)
3      i,s=0,0
4      while s<=n :
5          s=s+2*i+1
6          i=i+1
7      return i-1
```

Variant ?
Invariant ?
Que fait l'algorithme ?

Variants et invariants

Exemple

On reprend les codes suivants pour faire une recherche d'un élément dans une liste, ou d'une recherche par dichotomie dans une liste triée :

```
1  def appartient(l,L):
2      for x in L :
3          if l==x :
4              return True
5      return False
```

```
1  def appartientdicho(l,L):
2      a,b=0,len(L)-1
3      while b>=a:
4          c=(a+b)//2
5          if L[c]==l:
6              return True
7          if L[c]>l:
8              b=c-1
9          else:
10             a=c+1
11     return False
```

Variant ?
Invariant ?
Correction ?

Plan

Notion de complexité

Définition

Étant donné un programme, qui prend en argument différentes entrées, on appelle :

- ***complexité temporelle*** *le temps d'exécution du programme (qu'on estime à l'aide du nombre d'opérations effectuées par le programme)*
- ***complexité spatiale*** *la mémoire nécessaire au programme (qu'on estime à l'aide du nombre de cellules élémentaires de mémoires rajoutées durant l'exécution du programme)*

que l'on paramètre à chaque fois par les entrées.

Remarques

- *La complexité spatiale est moins utilisée : quand on dira simplement complexité, il faudra comprendre qu'il s'agit de la complexité temporelle.*
- *Pour simplifier, on étudiera souvent la complexité temporelle* ***dans le pire des cas*** *(qui est en général la plus probante).*
- *Toujours pour simplifier, on raisonnera en termes d'****ordres de grandeur*** *: avec des* O*, ou idéalement avec des* Θ *(des* O *mutuels, c'est-à-dire des* O *dont on ne peut rabaisser l'ordre de grandeur).*
- *Pour les complexités spatiales, on verra pour les tris la notion d'algorithme* ***en place*** *(on modifie directement la liste en entrée) ou* ***pas en place*** *(on crée une nouvelle liste qui prendra la forme modifiée). Les tris en place ayant l'avantage de limiter la mémoire allouée (donc la complexité spatiale).*

Notion de complexité

Exemple

Reprenons, et comparons, les recherches dans une liste triée suivant la méthode naïve et la méthode dichotomique :

```
1  def appartient(l,L):
2      for x in L :
3          if l==x :
4              return True
5      return False
```

```
1  def appartientdicho(l,L):
2      a,b=0,len(L)-1
3      while b>=a:
4          c=(a+b)//2
5          if L[c]==l:
6              return True
7          if L[c]>l:
8              b=c-1
9          else:
10             a=c+1
11     return False
```

- *complexité spatiale :* 0 *pour le premier et* 3 *allocations de mémoire pour le second*
- *complexité temporelle : si* `l` *se trouve à l'indice i de* `L`
  - *méthode naïve : on fait* $i+1$ *comparaisons*
  - *méthode dichotomique : on fait* $2+2(r-1)+1$ *affectations et* $3r+1$ *comparaisons avec r le nombre de passages dans la boucle* `while` *(*$r \leq \log_2(n)+1$*)*
- *complexité temporelle dans le pire des cas :* `l` *n'est pas dans* `L` *; on pose* $n = \text{len}(L)$ *:*
  - *méthode naïve : on fait n comparaisons*
  - *méthode dichotomique : on fait environ* $2\log_2(n)$ *affectations et* $2\log_2(n)$ *comparaisons*

La recherche dichotomique est (asymptotiquement) plus efficace : la complexité (dans le pire des cas) naïve est $C_1(n) = \Theta(n)$ *et la dichotomique est* $C_2(n) = \Theta(\log(n))$ *donc* $C_2(n) = o(C_1(n))$.

Exemples de complexité

Remarques

- *On s'intéresse à la complexité **asymptotique** (quand on fait devenir arbitrairement grands les paramètres) : dans le cas précédent, la recherche dichotomique est beaucoup plus efficace pour les grandes valeurs de n, mais pas si $n = 0$ ou $n = 1$ par exemple. En pratique, cela veut dire que l'on s'intéresse au cas où :*
  - *si un entier est en paramètre : il devient arbitrairement grand*
  - *si une liste est en paramètre : sa longueur devient arbitrairement grande*
  - *etc.*
- *On précisera parfois les opérations comptées ou autorisées pour le calcul de complexité. En général on se limitera à des opérations dont le temps de calcul (le **coût**) est constant, à savoir :*
  - *affectation*
  - *addition/multiplication/soustraction/comparaison d'entiers*
  - *addition/multiplication/soustraction/division/comparaison de flottants*
  - *accès à longueur ou un élément d'une liste (par indice), réaffectation d'un élément d'une liste, ajout/suppression d'un élément **en queue de liste** (par `append` ou `pop` sans argument)*
  - *recherche d'un élément dans un **dictionnaire** (pas dans une liste !)*
- *Selon le contexte, certains opérations seront considérées en temps constant ou non. Par exemple, selon le contexte, le calcul d'une division euclidienne pourra avoir soit un coût constant, soit de l'ordre du quotient (ce qui se comprend bien par l'algorithme du début du chapitre).*

Exemples de complexité

Définition

On classe les complexités, de paramètre $n \in \mathbb{N}$, de la manière suivante :

Complexité	*Nom*
$\Theta(1)$	*constante*
$\Theta(\ln(n))$	*logarithmique*
$\Theta(n)$	*linéaire*
$\Theta(n\ln(n))$	*semi-linéaire*
$\Theta(n^2)$	*quadratique*
$\Theta(n^3)$	*cubique*
$\Theta(n^k)$ *(k ≥ 2)*	*polynomiale*
$\Theta(\alpha^n)$ *(α > 1)*	*exponentielle*
$\Theta(n!)$	*factorielle*

qui sont rangées par ordre de grandeur.

Remarque

Les complexités au delà de la complexité semi-linéaire sont en général à éviter.
Les complexités au delà de la complexité polynomiale sont en général inacceptables.

Calculs de complexité

Remarque

La complexité ne s'estime pas simplement en regardant le nombre de lignes de code. Il faut chercher à l'estimer rigoureusement, par des méthodes adaptées à la programmation choisie.

Exemple

On verra que les deux codes suivants, pour calculer le n-ème terme dans la suite de Fibonnacci, sont respectivement de complexité exponentielle et linéaire. Le second code est bien meilleur !

```
1 def fibo1(n):
2     if n<=1 :
3         return n
4     return fibo1(n-1)+fibo1(n-2)
```

```
1 def fibo2(n):
2     if n<=1 :
3         return n
4     a,b=0,1
5     for i in range(n-2) :
6         a,b=b,a+b
7     return a+b
```

Et le code suivant est de complexité bien supérieure à la factorielle :

```
1 def A(m,n):
2     if m==0 :
3         return n+1
4     if n==0 :
5         return A(m-1,1)
6     return A(m-1,A(m,n-1))
```

Calculs de complexité

Méthode

Étant donnée $T(n)$ une instruction qui dépend de n, on note $\mathrm{C}(T(n))$ la complexité associée. Alors on a les situations suivantes :

Situation	*Instruction $T(n)$*	*Complexité $C(T(n))$*
Suite d'instructions	$\begin{cases} T_1(n) \\ T_2(n) \end{cases}$	$\mathrm{C}(T_1(n)) + \mathrm{C}(T_2(n))$
Conditionnelle	$\begin{cases} \text{si condition alors } T_1(n) \\ \text{sinon } T_2(n) \end{cases}$	$\max\left(\mathrm{C}(T_1(n)) + \mathrm{C}(T_2(n))\right)$
Boucle (while ou for)	$\begin{cases} \text{while condition/for k in...} \\ \text{faire } T_k(n) \end{cases}$	$\sum_k \mathrm{C}(T_k(n))$
Récursivité	$\begin{cases} \text{si condition alors } \begin{cases} T(n/2) \\ C_1(n) \end{cases} \\ \text{sinon } \begin{cases} T(n/2) \\ C_2(n) \end{cases} \end{cases}$	$\mathrm{C}(T(n/2)) + \max(C_1(n), C_2(n))$

Remarque

Le `while` est un peu plus subtile comme on ne sait pas à l'avance toutes les instructions qui seront effectuées.

Calculs de complexité

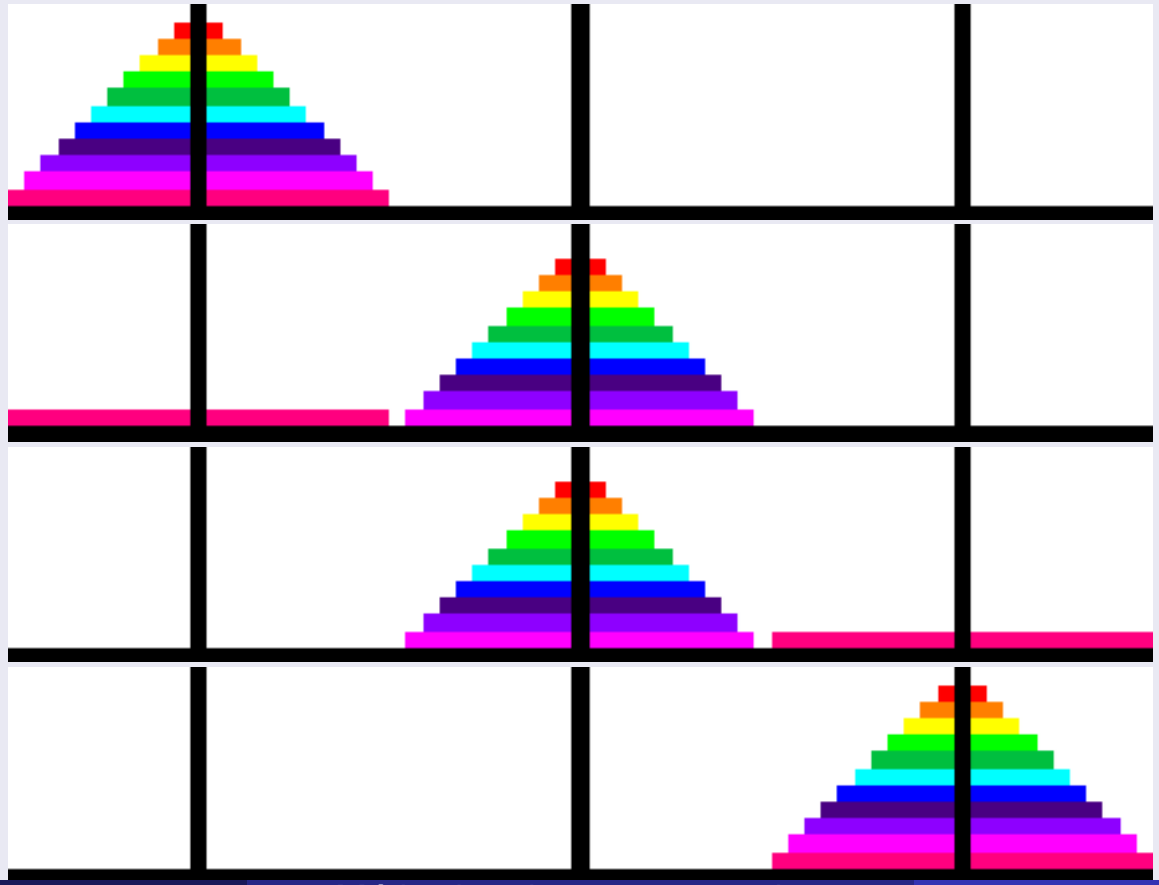
Proposition

Le problème des tours de Hanoï avec n disques se résout de manière optimale en $2^n - 1$ coups.

Démonstration.

Le nombre $C(n)$ de coups pour une résolution optimale à n disques vérifie :

$$C(1) = 1 \text{ et } \forall n \in \mathbb{N}^*,\ C(n+1) = 2 \cdot C(n) + 1.$$



Calculs de complexité

Exemple

On s'intéresse à la complexité du calcul de pgcd en utilisant l'algorithme d'Euclide suivant :

```
1 def pgcd(a:int,b:int)-> int :
2     while b>0 :
3         a,b=b,a%b
4     return a
```

Théorème (de Lamé)

Soient $a, b \in \mathbb{N}^*$ avec $a > b$. On suppose que l'algorithme d'Euclide appliqué à a et b se fait en k divisions euclidiennes (pour $k \in \mathbb{N}^*$). Alors $a \geq F_{k+2}$ et $b \geq F_{k+1}$, avec (F_n) la suite de Fibonacci. En particulier, les plus petits nombres a, b nécessitant k divisions euclidienne dans l'algorithme d'Euclide sont F_{k+2} et F_{k+1}.

Corollaire

En considérant la division euclidienne comme étant une opération élémentaire, le calcul du pgcd de a et b par algorithme d'Euclide a une complexité en $\log(\min(a, b))$.

Calculs de complexité

Exemple

On considère une liste `L` à n éléments. On a les opérations non élémentaires suivantes :

Opération	*Commande*	*Complexité*
Ajout en début de liste	`L=[x]+L`	$\mathrm{O}(n)$
Ajout en place i	`L.insert(x,i)`	$\mathrm{O}(n)$
Appartenance	`x in L`	$\mathrm{O}(n)$
Suppression en place i	`L.pop(i)`	$\mathrm{O}(n)$
Comparaison	`L==S`	$\mathrm{O}(n)$
Minimum	`min(L)`	$\mathrm{O}(n)$
Maximum	`max(L)`	$\mathrm{O}(n)$
Trier	`L.sort()`	$\mathrm{O}(n\ln(n))$
Copie	`L.copy()`	$\mathrm{O}(n)$
Tranche	`L[a:b]`	$\mathrm{O}(b-a)$ *(donc* $\mathrm{O}(n)$*)*

Calculs de complexité

Exemple

Dans l'exemple précédent, chaque opération peut être codée à l'aide des opérations sur les listes en temps constant avec cette complexité. Par exemple, l'ajout ou la suppression en i-ème position se codent par complexité en $\mathrm{O}(n)$ par les algorithmes pas en place :

```
1  def insertion(L,x,i):
2      LL=[]
3      for j in range(len(L)+1) :
4          if j<i :
5              LL.append(L[j])
6          if j==i :
7              LL.append(x)
8          if j>i :
9              LL.append(L[j-1])
10     return LL
```

```
1  def supression(L,i):
2      LL=[]
3      for j in range(len(L)-1) :
4          if j<i :
5              LL.append(L[j])
6          else :
7              LL.append(L[j+1])
8      return LL
```

ou par les algorithmes en place :

```
1  def insertionep(L,x,i):
2      L.append(L[len(L)-1])
3      for j in range(len(L)-1,i,-1) :
4          L[j]=L[j-1]
5      L[i]=x
```

```
1  def supressionep(L,i):
2      for j in range(i,len(L)-1) :
3          L[j]=L[j+1]
4      L.pop()
```

Calculs de complexité

Exemple

Reprenons les deux codes suivants de détermination des éléments dans la suite de Fibonacci :

```
1 def fibo1(n):
2     if n<=1 :
3         return n
4     return fibo1(n-1)+fibo1(n-2)
```

```
1 def fibo2(n):
2     if n<=1 :
3         return n
4     a,b=0,1
5     for i in range(n-2) :
6         a,b=b,a+b
7     return a+b
```

- *Pour `fibo1` : on a une programmation récursive, et donc si on note $C_1(n)$ la complexité pour l'entrée n on a :*

$$C_1(0) = C_1(1) = 1 \text{ et } \forall n \geq 2,\ C_1(n) = C_1(n-1) + C_1(n-2) + 2$$

et donc la suite C_1 est du même ordre de grandeur que la suite de Fibonacci : on a $C_1(n) = \Theta(\varphi^n)$ et la complexité est exponentielle.

- *Pour `fibo2` : on a une boucle `for` avec dans chacune deux affectations et un calcul de somme. Avec le test et la double affectation au départ et l'addition à la fin, la complexité $C_2(n)$ vérifie :*

$$C_2(0) = C_2(1) = 1 \text{ et } \forall n \geq 2,\ C_2(n) = 3 + 3 \times (n-2) + 1 = 3n - 2$$

et donc $C_2(n) = \Theta(n)$: la complexité est linéaire.

Calculs de complexité

Remarque

On peut donner des résultats plus précis en séparant bien les opérations : chaque opération élémentaire, même si elles ont des coûts constants, n'ont pas les mêmes coûts.
Pour les programmes précédents, si on note c le coût d'une comparaison, a celui d'une affectation et s celui d'une somme, on obtient plus précisément :

$$C_1(0) = C_1(1) = c \text{ et } \forall n \geq 2,\ C_1(n) = C_1(n-1) + C_1(n-2) + c + s$$

$$C_2(0) = C_2(1) = c \text{ et } \forall n \geq 2,\ C_2(n) = (2a + s) \cdot (n-1) + c$$

Calculs de complexité

Exemple

Posons $P(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_n X^n = \sum_{i=0}^{n} a_i X^i$ *et calculons* $P(b)$ *:*

Méthode	*Algorithme*	*Complexité*

naïve	`def P(a,b) :`
	`  s=0`
	`  for i in range(len(a)) :`
	`  s+=a[i]*(b**i)`
	`  return s`
		$\mathrm{O}(n^2)$ *(ou* $\mathrm{O}(n\ln(n))$*)*

subtile	`def P(a,b) :`
	`  s,p=0,1`
	`  for i in range(len(a)) :`
	`  s+=a[i]*p`
	`  p*=b`
	`  return s`
		$\mathrm{O}(n)$

(Ruffini-)Horner	`def P(a,b) :`
	`  s=0`
	`  for i in range(len(a)-1,-1,-1) :`
	`  s=s*b+a[i]`
	`  return s`
		$\mathrm{O}(n)$

Calculs de complexité

Remarque

C'était la méthode pour donner l'écriture décimale d'un nombre en base b : on part des premiers chiffres, qui correspondent aux plus grandes puissances de b, et qui sont les plus significatives. Elle correspond à l'écriture :

$$P(b) = a_0 + b \cdot (a_1 + b \cdot (a_2 + \ldots b \cdot (a_{n-2} + b \cdot (a_{n-1} + b \cdot a_n)) \ldots)) .$$

On a les invariants suivants : à la k-ème itération de la boucle, la somme s vaut :

- *en méthode naïve ou subtile :* $\sum_{i=0}^{k-1} a_i \cdot b^i$
- *en méthode de Horner :* $\sum_{i=n-k+1}^{n} a_i b^{i-n+k-1}$

qui donnent bien $P(b)$ *pour* $k = n + 1$ *(dernière itération de la boucle).*

La dichotomie

Méthode

La dichotomie est une méthode qui consiste à couper un problème en deux sous-problèmes pour diminuer la complexité du problème initial. Pour être :

- ***effective** : les deux sous-problèmes doivent être **solubles** et leur résolution doit permettre de résoudre le problème initial*
- ***utile** : la résolution du problème brut doit être rendue plus rapide par la résolution des deux sous-problèmes et leur synthèse*

Exemple

On se retrouve dans le désert face à un mur infini qu'on assimile à une droite graduée, dont on se trouve initialement à l'origine. Sur ce mur se trouve une porte. On considère x la distance (inconnue a priori) dont on est éloigné de la porte. On veut atteindre la porte, en cherchant à minimiser la distance parcourue.

- *méthode 1 : on choisit un sens, et on continue toujours jusqu'à trouver la porte.*
  Distance parcourue dans le pire des cas : $+\infty$ (et on ne trouve pas la porte)
- *méthode 2 : on zigzague entre les points de coordonnée $1, -1, 2, -2, 3, -3, 4, -4, \ldots$ jusqu'à trouver la porte.*
  Distance parcourue dans le pire des cas : la porte est en $-k$ pour $k \in \mathbb{N}$:

$$d = 1 + 2 + 3 + 4 + \cdots + (2k-1) + 2k = k(2k+1) = \Theta(k^2) = \Theta(x^2)$$

Complexité quadratique.

La dichotomie

Exemple

- *méthode 3 : on zigzague entre les points de coordonnée* $1, -2, 4, -8, 16, -32, \ldots, (-2)^k, \ldots$.
  Distance parcourue dans le pire des cas : la porte est en $x = (-2)^k$.

$$\begin{aligned} d &= 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 4 + 2 \cdot 8 + \cdots + 2 \cdot 2^{k-1} + 2^k \\ &= 2 \sum_{i=0}^{k-1} 2^i + 2^k \\ &= 2(2^k - 1) + 2^k \\ &= 3 \cdot 2^k - 2 \end{aligned}$$

  donc : $d = \Theta(2^k) = \mathrm{O}(x)$ *Et donc une complexité linéaire !*

- *méthode 4 : on zigzague entre les points de coordonnée* $1, -a, a^2, -a^3, a^4, -a^5, \ldots, (-a)^k, \ldots$ *pour* $a > 1$.
  Pour quel a est-ce optimal dans le pire des cas ?

La dichotomie

Exemple

On reprend la recherche par dichotomie dans une liste triée :

```
1  def appartientdicho(l,L):
2      a,b=0,len(L)-1
3      while b>=a:
4          c=(a+b)//2
5          if L[c]==l:
6              return True
7          if L[c]>l:
8              b=c-1
9          else:
10             a=c+1
11     return False
```

Ici la dichotomie est légitime. Pourquoi ? Évaluons la complexité dans le pire des cas l'élément `l` n'est pas dans `L` et on fait :

- *deux affectations au départ ;*
- *deux affectations et deux comparaisons à chaque passage dans la boucle.*

On pose $N(k)$ la longueur maximale d'une liste qui se traite de manière certaine en au plus k passages dans la boucle. On a : $N(0) = 0$ et $\forall k \in \mathbb{N},\ N(k+1) = 2 \cdot N(k) + 1$ donc :
$\forall k \in \mathbb{N},\ N(k) = 2^k - 1$.
Complexité : *si $n = \text{len}(L) \in [2^{k-1}; 2^k - 1]$ alors complexité dans le pire des cas en $\Theta(k)$. Donc complexité logarithmique comme $k = \lfloor \log_2(n) + 1 \rfloor = \mathrm{O}(\ln(n))$.*

Remarque

Dans une liste non-triée, on ne peut faire mieux que la recherche naïve...

La dichotomie

Exemple

Utilisons une méthode dichotomique pour calculer rapidement des puissances.

```
1 def puissancenaive(x,n):
2     p=1
3     for i in range(n):
4         p=p*x
5     return p
```

On calcule une puissance n-ème par n multiplications. On a donc une complexité en $\Theta(n)$

La méthode dichotomique repose sur le fait que :

$$\forall n \in \mathbb{N}^*,\ x^n = \begin{cases} \left(x^{n/2}\right)^2 = \left(x^2\right)^{n/2} & \text{si} \quad n \text{ est pair} \\ x \cdot \left(x^{(n-1)/2}\right)^2 = x\left(x^2\right)^{(n-1)/2} & \text{si} \quad n \text{ est impair} \end{cases}$$

On a les codes récursifs et itératifs :

```
1 def puissancedichorec(x,n):
2     if n==0 :
3         return 1
4     r=puissancedichorec(x,n//2)
5     r2=r*r
6     if n%2==0 :
7         return r2
8     return x*r2
```

```
1 def puissancedichoiter(x,n):
2     p,m,X=1,n,x
3     while m>0 :
4         if m%2==1 :
5             p=p*X
6         X=X*X
7         m=m//2
8     return p
```

La dichotomie

Exemple

On peut reprendre les deux méthodes précédentes pour trouver de manière efficace un entier naturel choisi secrètement par quelqu'un, avec le protocole suivant :

- *un entier naturel est choisi aléatoirement*
- *il peut être arbitrairement grand*
- *on peut proposer des nombres, jusqu'à tomber sur l'entier choisi initialement*
- *on cherche à tomber sur l'entier choisi en le moins de propositions possibles*

On propose une méthode dichotomique en deux temps :

- *on cherche déjà l'ordre de grandeur du nombre (en le comparant à des puissances de a avec* $a > 1$ *fixé)*
- *on trouve ensuite l'entier exact par dichotomie.*

Avec $a = 2$*, cela donne le code suivant :*

La dichotomie

Exemple

```
1  def devine(D:int)-> int :
2      n=rd.randint(0,2**(40*D**2))
3      N=1
4      compteur=1
5      while N<n :
6          N=2*N
7          compteur=compteur+1
8      test=N
9      while test > n :
0          N=N//2
1          compteur=compteur+1
2          if (test-N) >= n:
3              test-=N
4      return compteur
```

Pour trouver l'entier n, le nombre de proposition est au pire des cas de $2\log_2(n)$ *: la complexité est logarithmique.*

Remarques

- *On pourrait chercher à améliorer la première partie. Mais comme la seconde partie sera nécessairement logarithmique, cela n'a pas d'intérêt.*
- *Concrètement, cela revient à déterminer le nombre de chiffres dans l'écriture de n en base 2, puis chacun de ses chiffres (en commençant par les plus significatifs).*

Plan

Principe

Objectif

*On considère une liste d'objets que l'on peut ordonner (des entiers ou des flottants par exemple). On souhaite **trier** cette liste, c'est-à-dire la transformer en une nouvelle liste dont les éléments sont rangés par ordre croissant.*

Définition

*Étant donné une méthode de tri, on parle de tri **en place** si on modifie directement la liste en entrée de programme, sans créer un grand nombre de variables auxiliaires pour les calculs (c'est-à-dire que le nombre de variables à introduire ne dépendra pas de la taille de la liste). Dans le cas contraire, on dira que le tri n'est **pas en place**.*

Remarques

- *L'intérêt d'un tri en place est pour la complexité spatiale : il y a moins de mémoire utilisée. De plus, on n'a pas besoin de terminer le programme par un `return` : le simple fait de l'exécuter trie la liste.*
- *Un tri pas en place peut être utile pour quantifier le désordre dans une liste : on trie la liste, et on compare la liste initiale et la liste triée.*
- *Tout tri en place peut être rendu facilement comme un tri pas en place : il suffit de faire une copie de la liste initiale, et de faire le tri en place sur la liste copiée.*
- *L'inverse est faux : certains tris se codent de manière élémentaire pas en place, mais sont beaucoup plus difficiles à coder en place.*

Le tri à bulles

Méthode

*Le **tri à bulles** est un tri qui consiste à parcourir en boucle une liste, et à chaque parcours :*

- *comparer les éléments consécutifs*
- *les échanger s'ils ne sont pas dans le bon ordre*

Remarques

- *Cela revient à faire remonter les éléments qui ne sont pas à la bonne place, un peu comme des bulles d'air qui remontent à la surface de l'eau.*
- *Ce tri se code naturellement en place*

Le tri à bulles

Exemples

- *Si on considère la liste* $[5, 1, 4, 2, 8]$*, on a les étapes suivantes :*

État initial	*5*	*1*	*4*	*2*	*8*	*Échanges cumulés*
Parcours 1	***1***	***5***	*4*	*2*	*8*	*1*
	1	***4***	***5***	*2*	*8*	*2*
	1	*4*	***2***	***5***	*8*	*3*
Parcours 2	*1*	***2***	***4***	*5*	*8*	*4*
Parcours 3,4	*1*	*2*	*4*	*5*	*8*	*4*

- *Si on considère la liste* $[5, 4, 3, 2, 1]$*, on a les étapes suivantes :*

État initial	*5*	*4*	*3*	*2*	*1*	*Échanges cumulés*
Parcours 1	***4***	***5***	*3*	*2*	*1*	*1*
	4	***3***	***5***	*2*	*1*	*2*
	4	*3*	***2***	***5***	*1*	*3*
	4	*3*	*2*	***1***	***5***	*4*
Parcours 2	***3***	***4***	*2*	*1*	*5*	*5*
	3	***2***	***4***	*1*	*5*	*6*
	3	*2*	***1***	***4***	*5*	*7*
Parcours 3	***2***	***3***	*1*	*4*	*5*	*8*
	2	***1***	***3***	*4*	*5*	*9*
Parcours 4	***1***	***2***	*3*	*4*	*5*	*10*

Le tri à bulles

Théorème

Si on considère une liste `L` possédant n éléments (pour $n \in \mathbb{N}$), alors il suffit de faire $n-1$ parcours pour la trier par tri à bulle.
Plus précisément : il suffit, pour le k-ème parcours, de parcourir les $n-k+1$ premiers éléments de la liste (c'est-à-dire comparer les éléments d'indice i et $i+1$ pour $0 \leq i \leq n-k-1$).

Démonstration.

On a le variant de boucle "à la fin du k-ème parcours, les k derniers éléments de L sont ses k plus grands éléments, rangés dans le bon ordre". □

Corollaire

La complexité du tri à bulle dans le pire des cas est en $\Theta(n^2)$.

Démonstration.

Dans le pire des cas, la liste est triée dans l'ordre inverse, et on fait systématiquement des échanges, ce qui donne :

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}\sim\frac{n^2}{2}$$

comparaisons, et autant d'échanges. □

Le tri à bulles

Remarque

On devra effectuer les $\frac{n(n-1)}{2}$ comparaisons, et la complexité sera toujours (dans le meilleur comme dans le pire des cas) en $\Theta(n^2)$. On peut rajouter un indicateur pour noter un parcours dans lequel on ne fait rien : cela veut dire que la liste est triée et on arrête le programme.

Proposition

On a les codes suivants :

```
1  def TriBulle(L):
2      n=len(L)
3      for k in range(n) :
4          for i in range(n-k-1):
5              if L[i]>L[i+1] :
6                  L[i],L[i+1]=L[i+1],L[i]
7      return L              # optionnel comme tri en place
```

```
1  def TriBulleIndic(L):
2      n=len(L)
3      indic=True
4      for k in range(n) :
5          if indic :
6              indic=False
7              for i in range(n-k-1):
8                  if L[i]>L[i+1] :
9                      L[i],L[i+1]=L[i+1],L[i]
0                      indic=True        # la liste n'etait pas triee
1      return L             #  optionnel comme tri en place
```

Le tri par sélection

Méthode

*Le **tri par sélection** est un tri qui consiste à parcourir en boucle une liste, et à chaque parcours :*

- *chercher le plus petit élément parmi ceux qui ne sont pas encore triés*
- *le mettre à la bonne place qui serait la sienne dans une version triée*

Remarques

- *Le fonctionnement est assez proche du tri à bulles. Les différences étant que :*
  - *à chaque nouveau parcours, on ne regarde pas un élément au début de plus*
  - *on ne fait (au plus) qu'une seule inversion*
- *On peut aussi faire une version où ou cherche le plus grand élément (au lieu du plus petit), ce qui ressemble encore plus au tri à bulles.*
- *Ce tri se code naturellement en place.*

Le tri par sélection

Exemples

- *Si on considère la liste* $[5, 1, 4, 2, 8]$*, on a les étapes suivantes :*

État initial	*5*	*1*	*4*	*2*	*8*	*Indices à échanger*
Parcours 1	*5*	***1***	*4*	*2*	*8*	$1 \leftrightarrow 2$
Parcours 2	*1*	*5*	*4*	***2***	*8*	$2 \leftrightarrow 4$
Parcours 3	*1*	*2*	***4***	*5*	*8*	$3 \leftrightarrow 3$
Parcours 4	*1*	*2*	*4*	***5***	*8*	$4 \leftrightarrow 4$
État final	*1*	*2*	*4*	*5*	*8*	

- *Si on considère la liste* $[5, 1, 2, 3, 4]$*, on a les étapes suivantes :*

État initial	*5*	*1*	*2*	*3*	*4*	*Indices à échanger*
Parcours 1	*5*	***1***	*2*	*3*	*4*	$1 \leftrightarrow 2$
Parcours 2	*1*	*5*	***2***	*3*	*4*	$2 \leftrightarrow 3$
Parcours 3	*1*	*2*	*5*	***3***	*4*	$3 \leftrightarrow 4$
Parcours 4	*1*	*2*	*3*	*5*	***4***	$4 \leftrightarrow 5$
État final	*1*	*2*	*3*	*4*	*5*	

Le tri par sélection

Théorème

Si on considère une liste `L` possédant n éléments (pour $n \in \mathbb{N}$), alors il suffit de faire $n-1$ parcours pour la trier par tri par sélection.
Plus précisément : il suffit, pour le k-ème parcours, de parcourir les $n-k+1$ derniers (resp. premiers) éléments de la liste si on cherche à chaque parcours le plus petit (resp. grand) élément parmi ceux à trier.

Démonstration.

On a le variant de boucle “à la fin du k-ème parcours, les k premiers éléments de L sont ses k plus petits éléments, rangés dans le bon ordre”. □

Corollaire

La complexité du tri par sélection dans le pire des cas est en $\Theta(n^2)$.

Démonstration.

Dans tous les cas, on recherche l'indice du minimum dans une tranche à $n-k+1$ éléments, ce qui demande $n-k$ comparaisons. Et donc :

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=\Theta(n^2)$$

comparaisons. Comme on fait une ou aucune inversion par parcours, cela fait au plus $n-1=\Theta(n)$ inversions. Et finalement une complexité en $\Theta(n^2)$. □

Le tri par selection

Proposition

On a les codes suivants :

```
1  def Indicemin(L,a,b):
2      indice,min=a,L[a]
3      for i in range(a+1,b):
4          if L[i]<min :
5              indice,min=i,L[i]
6      return indice
7
8  def TriSelection(L):
9      n=len(L)
0      for k in range(n-1):
1          i=Indicemin(L,k,n)
2          L[k],L[i]=L[i],L[k]
3      return L                # optionnel comme tri en place
```

Le tri par insertion

Méthode

*Le **tri par insertion** est un tri qui consiste à parcourir à trier progressivement la liste jusqu'à un indice de plus en plus grand. Pour étendre la partie triée de la liste :*

- *on détermine l'indice où on devrait placer le premier élément non trié*
- *on l'insère à cette place*

Remarques

- *C'est le fonctionnement inverse des tris précédents :*
  - *par rapport au tri à bulle : on choisit successivement nos bulles que l'on fait descendre parmi celles déjà en place ;*
  - *par rapport au tri par sélection : au lieu de chercher l'élément à placer à un endroit donné, on cherche où placer un élément donné.*
- *C'est l'un des tris les plus utilisés pour trier des cartes dans une main.*
- *Pour insérer, on fait une permutation circulaire des éléments, qu'on fait par des échanges (à la manière du tri à bulles).*
- *Ce tri se code naturellement en place.*

Le tri par insertion

Exemples

- *Si on considère la liste* $[5, 1, 4, 2, 8]$*, on a les étapes suivantes :*

État initial	*5*	*1*	*4*	*2*	*8*	*Indices où insérer*	*Cycle à effectuer*
Étape 1	***1***	*5*	*4*	*2*	*8*	*1*	$2 \rightarrow 1$
Étape 2	*1*	***4***	*5*	*2*	*8*	*2*	$3 \rightarrow 2$
Étape 3	*1*	***2***	*4*	*5*	*8*	*2*	$4 \rightarrow 3 \rightarrow 2$
Étape 4	*1*	*2*	*4*	*5*	*8*	*5*	*aucun*

- *Si on considère la liste* $[5, 4, 3, 2, 1]$*, on a les étapes suivantes :*

État initial	*5*	*4*	*3*	*2*	*1*	*Indices où insérer*	*Cycle à effectuer*
Étape 1	***4***	*5*	*3*	*2*	*1*	*1*	$2 \rightarrow 1$
Étape 2	***3***	*4*	*5*	*2*	*1*	*1*	$3 \rightarrow 2 \rightarrow 1$
Étape 3	***2***	*3*	*4*	*5*	*1*	*1*	$4 \rightarrow 3 \rightarrow 2 \rightarrow 1$
Étape 4	***1***	*2*	*3*	*4*	*5*	*1*	$5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Le tri par insertion

Théorème

Si on considère une liste `L` *possédant n éléments (pour* $n \in \mathbb{N}$*), alors il suffit de faire* $n-1$ *étapes pour la trier par tri par insertion.*
Plus précisément : il suffit, pour le k*-ème parcours, de parcourir les* k *premiers éléments de la liste pour trouver où insérer le* $k+1$*-ème.*

Démonstration.

On a le variant de boucle “à la fin de k-ème étape, les $k+1$ premiers éléments de L sont ses k plus petits éléments, rangés dans le bon ordre”. □

Corollaire

La complexité du tri par insertion dans le pire des cas est en $\Theta(n^2)$*.*

Démonstration.

Pour insérer le $k+1$-ème élément, on parcourt les k premiers. Pour l'insérer à l'indice i, on fait :

- i ou $k-i$ comparaisons (selon qu'on parte des plus petits ou des plus grands)
- $k+1-i$ échanges de valeurs

et donc $\Theta(k)$ opérations élémentaires dans le pire des cas. On a le résultat en sommant. □

Le tri par insertion

Remarque

Le "pire des cas" dépend du sens dans lequel on cherche l'indice, et surtout de ce qui est le plus couteux entre des inversions ou des comparaisons.
Avec une méthode de recherche dichotomique pour trouver l'indice où insérer, on diminue le nombre de comparaisons, mais la complexité dans le pire des cas reste en $\Theta(n^2)$ (nombre d'inversions à faire).

Proposition

On a les codes suivants :

```
1  def TriInsert(L):
2      n=len(L)
3      for k in range(1,n):
4          i=k
5          v=L[k]
6          while (i>0) and (L[i-1]>v):
7              L[i]=L[i-1]
8              i=i-1
9          L[i]=v
10     return L   # optionnel comme tri
           en place
```

```
1  def IndiceDicho(L,a,b,x):
2      while b>=a :
3          c=(a+b)//2
4          if L[c]==x :
5              return c+1
6          if L[c]>x :
7              b=c-1
8          else :
9              a=c+1
10     return a
11
12 def TriInsertDicho(L):
13     n=len(L)
14     for k in range(1,n):
15         i=IndiceDicho(L,0,k-1,L[k])
16         for j in range(i,k):
17             L[j],L[j+1]=L[j+1],L[j]
18     return L    # optionnel comme
           tri en place
```

Le tri fusion

Méthode

On coupe (naïvement) la liste en deux et on cherche à trier chacune des listes : on peut efficacement fusionner les listes obtenues pour obtenir une version triée de la liste initiale.

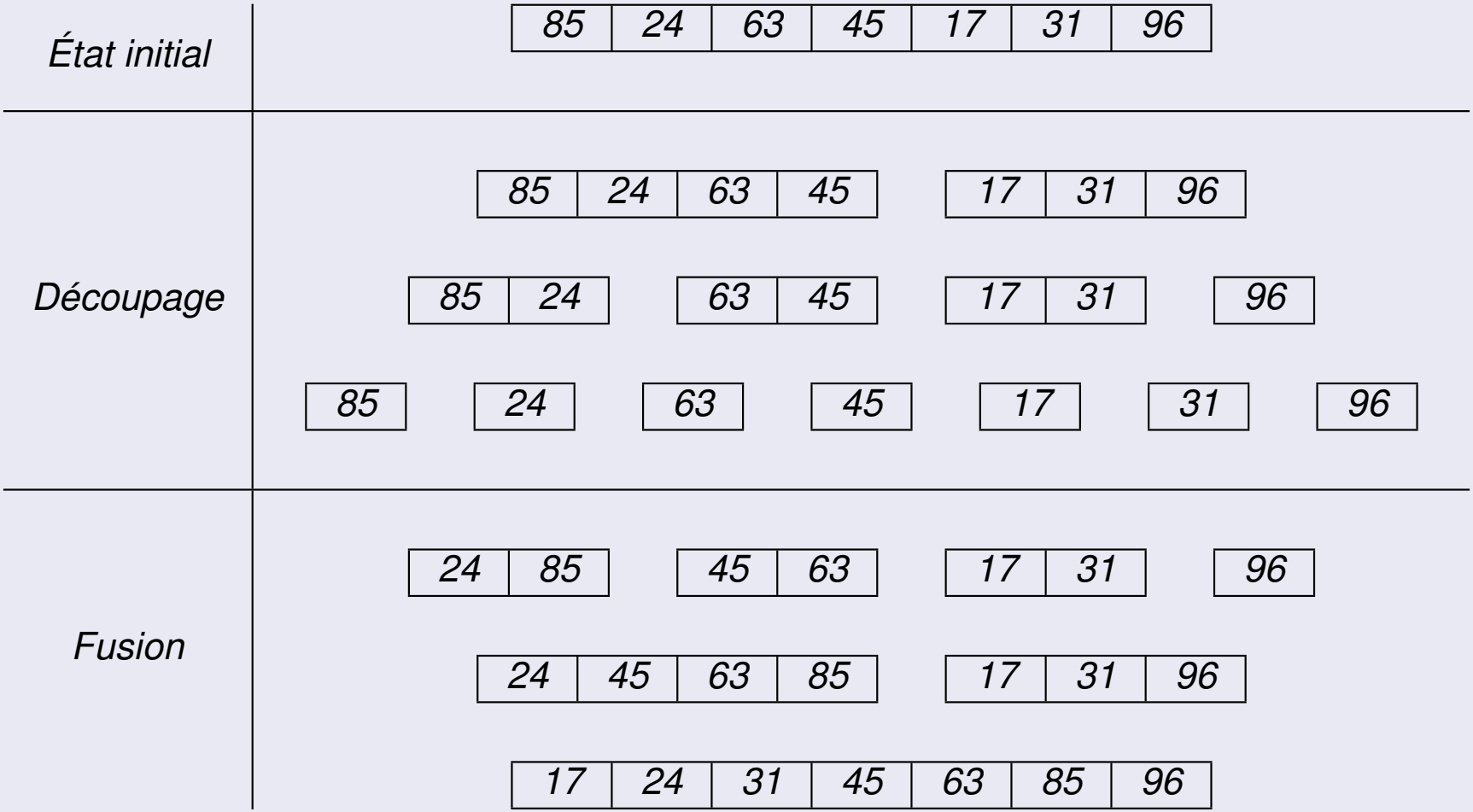
Remarques

1. *Les sous-listes sont triées aussi par tri fusion : on procède donc récursivement ! Il faut une initialisation : les listes de un élément ou moins sont triées.*
2. *L'enjeu est exclusivement sur la fusion : il faut pouvoir fusionner efficacement des listes triées pour y gagner en complexité : celle-ci peut se faire récursivement (avec une comparaison) ou itérativement (en itérant deux pointeurs).*
3. *Ce tri se code naturellement pas en place. On peut tout de même en faire une version en place, mais c'est beaucoup plus compliqué.*
4. *Il n'y a pas tellement de "pire des cas" : tous les cas prennent autant de temps, puisqu'il n'y a pas d'incidence sur les manières de fusionner les listes.*

Le tri fusion

Exemple

On va trier la liste $[85, 24, 63, 45, 17, 31, 96]$ *:*



Le tri fusion

Proposition

Fusionner deux listes triées de longueurs n_1 et n_2 se fait de manière optimale avec une complexité en $\Theta(n_1 + n_2)$.

Démonstration.

On balaye dans l'ordre et simultanément les éléments des liste L_1 et L_2. À chaque étape :

- on fait une comparaison ;
- on ajoute un élément.

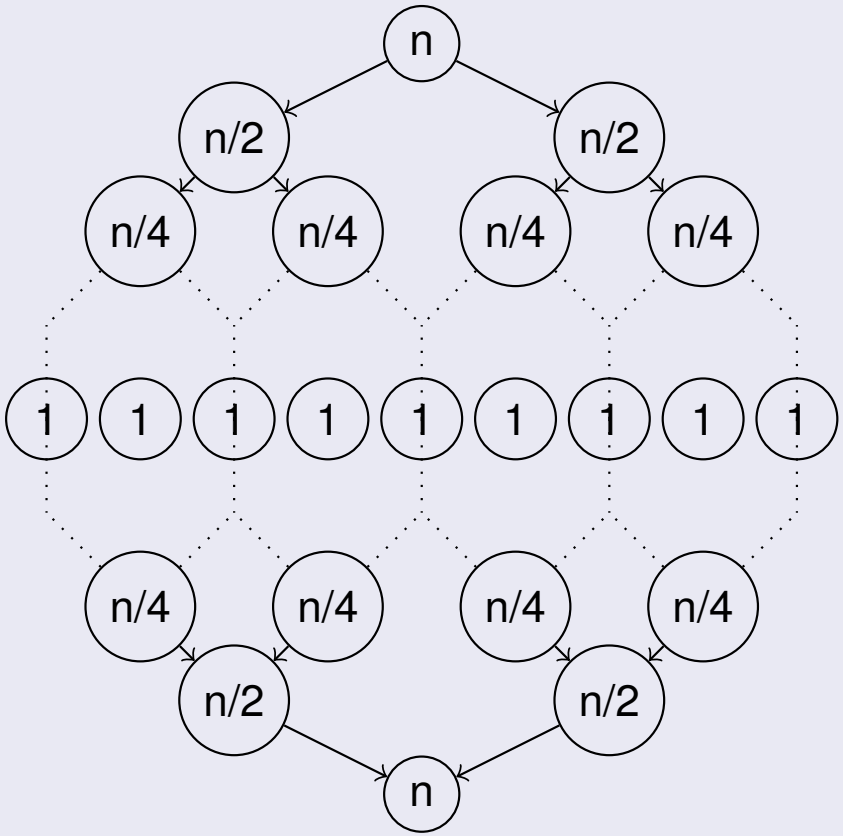
Il y a autant d'étapes que dans l'ensemble des deux listes, donc $n_1 + n_2$. □

Théorème

La complexité du tri fusion (dans le pire des cas) est en $\Theta(n\ln(n))$.

Le tri fusion

Démonstration.



coût en $\Theta(n)$
coût en $\Theta(n)$
⋮
coût en $\Theta(n)$
coût en $\Theta(n)$

$\log_2(n)$ étapes
coût en $\Theta(n\ln(n))$

⋮
coût en $\Theta(n)$
coût en $\Theta(n)$

$\log_2(n)$ étapes
coût en $\Theta(n\ln(n))$

Et un coût total en $\Theta(n\ln(n))$

□

Le tri fusion

Remarque

On peut aussi le faire en :

- *le prouvant par récurrence sur les listes de tailles une puissance de 2 : si on note $C(p)$ le coût pour une liste de taille $n = 2^p$ (de sorte que $p = \log_2(n)$) alors on a une relation du type :*

$$C(p) = 2 \cdot C(p-1) + \alpha \cdot 2^p$$

avec $\alpha \cdot m$ le coût (linéaire) de fusion de listes dont la somme fait m.
En posant $u_p = \dfrac{C(p)}{2^p}$, on a la relation :

$$u_p = \frac{C(p-1)}{2^{p-1}} + \alpha = u_{p-1} + \alpha$$

donc u_p est arithmétique de raison α, donc de la forme $u_p = u_0 + \alpha p$. Et ainsi :

$$C(p) = 2^p(u_0 + \alpha p) = \Theta(2^p \cdot p) = \Theta(n\ln(n)).$$

- *le coût est croissant avec la taille des listes : si $n \in \mathbb{N}$, en notant $p = \lfloor \log_2(n) \rfloor$ on a : $2^p \leq n < 2^{p-1}$ et par croissance le coût C pour une liste de taille n vérifie :*

$$C(p) \leq C \leq C(p+1)$$

ce qui donne par encadrement : $C = \Theta(n\ln(n))$.

Le tri fusion

Proposition

On a les codes suivants :

```
1  def FusionIter(L1,L2) :
2      n1,n2 = len(L1),len(L2)
3      i,j=0,0
4      L=[]
5      while i<n1 and j<n2 :
6          if L1[i]<L2[j] :
7              L.append(L1[i])
8              i+=1
9          else :
10             L.append(L2[j])
11             j+=1
12     return L+L1[i:]+L2[j:]
13
14 def TriFusionIter(L) :
15     n=len(L)
16     if n < 2 :
17         return L
18     L1 = TriFusionIter(L[:n//2])
19     L2 = TriFusionIter(L[n//2:])
20     return FusionIter(L1,L2)
```

```
1  def FusionRec(L1,L2) :
2      if len(L1)*len(L2)==0 :
3          return L1+L2
4      a,b=L1[0],L2[0]
5      if a<b :
6          return [a]+FusionRec(L1[1:],L2)
7      return [b]+FusionRec(L1,L2[1:])
8
9  def TriFusionRec(L) :
10     n=len(L)
11     if n < 2 :
12         return L
13     L1 = TriFusionRec(L[:n//2])
14     L2 = TriFusionRec(L[n//2:])
15     return FusionRec(L1,L2)
```

Le tri rapide

Méthode

*On fixe un **pivot**, qui est un élément de la liste (souvent le premier élément), et on répartit les éléments de la liste en deux nouvelles listes selon qu'ils sont plus petits ou plus grands que le pivot. On trie chacune des sous-listes suivant la même méthode, qu'on fusionne naïvement par concaténation.*

Remarques

1. *Les sous-listes sont triées aussi par tri rapide : comme pour le tri fusion, c'est une méthode récursive, qui repose sur la même initialisation : les listes de un élément ou moins sont triées.*
2. *La fusion des listes triées est ici naïve : si on note p le pivot, et L_1, L_2 les deux sous-listes obtenues, on a : $\forall x \in L_1,\ \forall y \in L_2,\ x \leq p < y$ ce qui justifie que la fusion n'est qu'une concaténation.*
3. *Comme pour le tri fusion, ce tri se code naturellement pas en place. On peut tout de même en faire une version en place, mais c'est beaucoup plus compliqué.*
4. *On verra que le "pire des cas" est assez surprenant, et surtout qu'il est de même complexité que le tri à bulles : le tri rapide est seulement rapide "en moyenne", mais pas dans le pire des cas.*

Le tri rapide

Exemple

On va trier la liste $[85, 24, 63, 45, 17, 31, 96]$ *:*

Le tri rapide

Théorème

Le tri rapide a une complexité :

- *dans le pire des cas en* $\Theta(n^2)$ *;*
- *dans le meilleur des cas en* $\Theta(n\ln(n))$ *;*
- *en moyenne en* $\Theta(n\ln(n))$*.*

Démonstration.

La fusion étant en $\Theta(n)$ qui est négligeable devant le découpage, dont les complexités sont données par les relations de récurrence :

- dans le pire des cas : le pivot est le plus petit ou plus grand élément :

$$C(n) = \alpha(n-1) + C(n-1)$$

- dans le meilleur des cas : le pivot est la médiane :

$$C(n) = \alpha(n-1) + 2 \cdot C(n/2)$$

- en moyenne : le pivot est au premier ou dernier quartile :

$$C(n) = \alpha(n-1) + C(n/4) + C(3n/4)$$

□

Le tri rapide

Remarques

- *La complexité dans le pire des cas n'est pas probante ici, car on est alors du même ordre de grandeur que les tris naïfs.*
- *On peut forcer d'aller dans le meilleur des cas, en prenant comme pivot la médiane : par un algorithme de complexité linéaire (type médiane des médianes), on ajoute un coût en* $\mathrm{O}(n\ln(n))$ *et on est bien en* $\Theta(n\ln(n))$ *en coût total.*
- *On peut être plus précis dans la complexité en moyenne : les éléments (rangés par ordre croissant) d'indices i et j (pour* $i < j$*) sont comparés si, et seulement si, aucun des éléments entre eux n'est choisi comme pivot avant, soit un probabilité de* $\frac{2}{j-i+1}$*. Le nombre moyen total de comparaison est donc :*

$$C(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k+1} \leq 2 \sum_{i=1}^{n} \ln(n+1) = \mathrm{O}(n\ln(n)).$$

Le tri rapide

Proposition

On a le code suivant :

```
1  def tri_rapide(L):
2      n=len(L)
3      if n < 2 :
4          return L
5      pivot=L[0]
6      moins,plus=[],[]
7      for i in range(1,n) :
8          if L[i]<pivot :
9              moins.append(L[i])
0          else :
1              plus.append(L[i])
2      return tri_rapide(moins)+[pivot]+tri_rapide(plus)
```

Autres tris

Remarque

Il existe de nombreux autres tris, qui sont adaptés à un certain contexte, ce qui change la complexité :

- *tri par dénombrement : lorsqu'il y a peu d'éléments différents (complexité linéaire)*
- *tri par base : pour des nombres qui s'écrivent avec peu de chiffres connus à l'avance, adapté du tri par dénombrement (complexité linéaire)*
- *tri de crêpes : qui se visualise comme une pile de crêpes, qu'on trie/mélange en faisant sauter et se retourner le haut de la pile (complexité linéaire, mais non connue exactement)*

Remarque

Un autre critère pour juger un tri, outre sa complexité et le fait qu'il soit en place ou non, est le fait qu'il soit stable ou pas : un tri est dit ***stable*** *s'il laisse des éléments égaux dans le même ordre entre avant et après le tri. Cela a surtout du sens pour un tri en place, et permet de voir si un tri fait des déplacements inutiles d'éléments.*
Le tri à bulle précédent était stable, et on peut le rendre non stable comme suit :

```
1 def TriBullepasstable(L):
2     n=len(L)
3     for k in range(n) :
4         for i in range(n-k-1):
5             if L[i]>L[i+1] :
6                 L[i],L[i+1]=L[i+1],L[i]
7     return L                # optionnel comme tri en place
```