ALGORITHMIQUE, cours 1

Cours : S. Peyronnet

Comment me contacter ?

sylvain.peyronnet@lrde.epita.fr

Structure du cours

- Complexité et algorithmique : définitions
- Principales méthodes de tri
- Structures de données de bases
- Structures de données avancées
- Principaux paradigmes algorithmiques

Support de cours

Introduction to algorithms par Cormen, Leiserson, Rivest et Stein.

Introduction

- Selon le Petit Robert : "ensemble des règles opératoires propres à un calcul.”

- Un peu plus précisement : Une séquence de pas de calcul qui prend un ensemble de valeurs comme entrée (input) et produit un ensemble de valeurs comme sortie (output).

- Un algorithme résout toujours un problème de calcul. L’énoncé du problème spécifie la relation input / output souhaitée.

Exemples :

1. Multiplication

   Input : deux entiers a et b

   Output : leur produit ab

   Algorithme : celui de l'école

2. Plus Grand Commun Diviseur (PGCD)

   Input : deux entiers a et b

   Output : pgcd(a,b)

   Algorithme : celui d'Euclide

3. Primalité

   Input : un entier a

   Question : a est-il un entier premier ?

Cas spécial : problème de décision - output = réponse oui/non à une question.

Donner une définition précise de la notion d’algorithme est assez difficile et complexe.

Il existe plusieurs définitions mathématiques depuis les années 30.

exemples :

- fonctions récursives
- λ-calcul
- machines de Turing
- RAM
  (Random Access Machine : machine à accès aléatoire)

Fait :

Toutes ces définitions sont équivalentes

Thèse de Church :

Tout ce qui est calculable intuitivement est aussi calculable par les objets formels ci-dessus

Mais :

Il existe des problèmes non calculables (indécidables)

exemple :

Problème de l'arrêt

Input : Un algorithme (programme) A et un input I

Output : A termine-t-il pour I ?

preuve de l'indecidabilité (par l'absurde)

Supposons que le problème de l'arrêt soit décidable.
Cela signifie alors qu'il existe un programme B qui décide si un programme s'arrête. Définissons maintenant le programme C suivant:

- C s'arrête quand le programme qu'il prend en entrée ne s'arrête pas.
- C ne s'arrête pas quand le programme qu'il prend en entrée s'arrête.

Bien sûr, un tel programme ne peut pas exister car si on l'utilise sur lui même, on obtient une contradiction. Et donc le problême de l'arrêt est indécidable.

Dans ce cours, on étudie seulement des problèmes pour lesquels il existe des algorithmes.

En fait, on s'intéressera aux problèmes pour lesquels il existe des algorithmes efficaces.

Un exemple typique de problème décidable pour lequel il n'y a pas d'algorithme efficace : le jeu d'échec (nombre de configurations est environ 10^{50})

On veut une mesure pour comparer des algorithmes

On voudrait que la complexité :

- **ne dépende pas de l'ordinateur**
- **ne dépende pas du langage de programmation**
- **ne dépende pas du programmeur**
- **... etc ...**
- **soit indépendante des détails**

de l'implémentation

On choisit de travailler sur un modèle d'ordinateur :

RAM :

- **ordinateur idéalisé**
- **mémoire infinie**
- **accès à la mémoire en temps constant**

Le langage de programmation sera un pseudo-PASCAL.

Pour mesurer la complexité en temps on choisit une opération fondamentale pour le problème de calcul particulier, et on calcule le nombre d'opérations fondamentales exécutées par l'algorithme. Le choix est bon si le nbre total d'opérations est proportionnel au nbre d'opérations fondamentales.

exemples :

Problème	Opération fondamentale
recherche d'un élément dans une liste	comparaison entre l'élément et les entrées de la liste
multiplication des matrices réelles	multiplication scalaire
addition des entiers binaires	opération binaire

Le temps de l'exécution dépend de la taille de l'entrée. On veut considérer seulement la taille essentielle de l'entrée.

Cela peut être par exemple :

- le nombre d'éléments combinatoires dans l'entrée
- le nombre de bits pour représenter l'entrée
- ... etc ...

exemples :

Problème	Taille de l'entrée
recherche d'un élément dans une liste	nombre d'éléments dans la liste
multiplication des matrices réelles	dimension des matrices
addition des entiers binaires	nbre de bits pour représenter les entiers

Soit A un algorithme

D_n l'ensemble des entrées de taille n

$I \in D_n$ une entrée

définition :

1. $cout_A(I) =$ le nbre d'op. fondamentales exécutées par A sur I
2. la complexité de A en pire cas :

$$Max_A(n) = \text{Max}\{cout_A(I); I \in D_n\}$$

Soit Pr une distribution de probabilités sur D_n

définition :

la complexité de A en moyenne :

$$Moy_A(n) = \sum_{I \in D_n} Pr[I] \cdot cout_A(I)$$

RECHERCHE

Input : L,n,x ; où L est un tableau $[1,\ldots,n]$ et x est l'élément recherché

Output : l'indice de x dans L (0 si $x \notin L$)

Algorithme : Recherche Séquentielle (RS)

```
var L : array [1,...,n] of element;
      x : element;
      j : integer;

begin
      j:=1;
      while j≤n and L[j]≠x do begin
            j:=j+1;
      end { while }
if j>n then j:=0;
end
```

Opération de base : comparaison de x avec un élément de L.

complexité en pire cas de RS :

$$Max_{RS}(n) = n$$

complexité en moyenne de RS :

On suppose que :

- tous les éléments sont distincts
- $Pr[x \in L] = q$
- si $x \in L$ alors toutes les places sont équiprobables

Pour $1 \leq i \leq n$ soit

$$I_i = \{(L,x) : L[i] = x\} \text{ et } I_{n+1} = \{(L,x) : x \notin L\}$$

On a :

$Pr[I_i] = \frac{q}{n}$ pour $1 \leq i \leq n$ et $cout_{RS}(I_i) = i$

$Pr[I_{n+1}] = 1 - q$ et $cout_{RS}(I_{n+1}) = n$

$$Moy_{RS}(n) = \sum_{i=1}^{n+1} Pr[I_i] \cdot cout_{RS}(I_i)$$

$$= \sum_{i=1}^{n} \frac{q}{n} \cdot i + (1-q) \cdot n = \frac{q}{n} \sum_{i=1}^{n} i + (1-q) \cdot n$$

$$= \frac{q}{n} \cdot \frac{n(n+1)}{2} + (1-q) \cdot n = (1 - \tfrac{q}{2}) \cdot n + \tfrac{q}{2}$$

Si $q = 1$ alors $Moy_{RS}(n) = \frac{n+1}{2}$

Si $q = \frac{1}{2}$ alors $Moy_{RS}(n) = \frac{3n+1}{4}$

Exercice Ecrire un algorithme pour trouver la médiane de trois entiers. (Par définition, la médiane de $2k-1$ éléments est le k^{eme} plus grand).

- Combien de comparaisons fait votre algorithme en pire cas?
- En moyenne?
- Quel est la complexité de ce problème?

Exercice On veut déterminer si une suite binaire de longueur n contient deux 0's consécutifs. L'opération de base est d'examiner une position dans la suite pour voir si elle contient un 0 ou un 1. Quelle est la complexité en pire cas du meilleur algorithme pour ce problème quand $n = 2,3,4,5$?

On voudrait comparer la complexité de deux algorithmes

• **Les constantes ont peu d'importance, on peut les négliger. Elles ne dépendent que des particularités de l'implémentation.**

• **On ne s'intéresse pas aux entrées de petite taille. On va comparer les deux complexités asymptotiquement.**

Notation :

$\mathbb{N} = \{0,1,2,\ldots\}$ $\mathbb{R} = reels$

$\mathbb{N}^+ = \{1,2,3,\ldots\}$ $\mathbb{R}^+ = reel\ positifs$

$\mathbb{R}^* = \mathbb{R} \setminus \{0\}$

$f,g : \mathbb{N}^+ \to \mathbb{R}^*$ (souvent on peut étendre à $\mathbb{R}^* \to \mathbb{R}^*$)

définition :

$$\mathcal{O}(f) = \{g : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+, \forall n \geq n_0,\ g(n) \leq c \cdot f(n)\}$$

On dit que "g est dans grand $\mathcal{O}$ de f".

exemples :

$$f(n) = \frac{n^3}{2} \qquad g_1(n) = n^2 + 17n + 15 \qquad g_2(n) = 5n^3$$

- $g_1 \in \mathcal{O}(f)$: on prend $n_0 = 1$ et $c = 2(1 + 17 + 15)$ et alors $n^2 + 17n + 15 \leq c \cdot \frac{n^3}{2}$ si $n \geq 1$
- $g_2 \in \mathcal{O}(f)$: on choisit $c = 10$ et alors $5n^3 \leq 10 \cdot \frac{n^3}{2}$
- $f \in \mathcal{O}(g_2)$
- $f \notin \mathcal{O}(g_1)$ sinon $\frac{n^3}{2} \leq c \cdot n^2 + 17c \cdot n + 5c$ et donc si n est grand on a $\frac{n}{2} \leq c + 2$ ce qui est impossible.

Attention : les fonctions peuvent être incomparables au sens du $\mathcal{O}$

Exercice Donner un exemple de deux fonctions f et g telles que

$$f \notin O(g) \text{ et } g \notin O(f).$$

définition :

$$\Omega(f) = \{g : \exists c > 0, \exists n_0 \in \mathbb{N}^+, \forall n \geq n_0 \; g(n) \geq c \cdot f(n)\}$$

proposition 1 :

$$g \in \mathcal{O}(f) \Leftrightarrow f \in \Omega(g)$$

preuve :

$$g(n) \leq c \cdot f(n) \Leftrightarrow f(n) \geq \frac{1}{c} \cdot g(n)$$

définition :

$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$

proposition 2 :

$$\Theta(f) = \{g : \exists c_1, c_2, \exists n_0 \in \mathbb{N}^+, \forall n \geq n_0 \; c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

$\Theta(f)$ est appelé l'ordre exact ou ordre de grandeur de f.

proposition 3 :

La relation $f \; \mathcal{R} \; g$ si $f \in \Theta(g)$ est une relation d'équivalence.

Exercice

1. L'ordre exact de la fonction $3n^2 + 10n \log n + 100$ est:
   1. $\Theta(n \log n)$
   2. $\Theta(n^3)$
   3. $\Theta(n^2)$
   4. $\Theta(\log n)$
2. Soient $0 < a < b$ deux constantes. On a:
   1. $\Theta(\log_a n) < \Theta(\log_b n)$
   2. $\Theta(\log_a n) = \Theta(\log_b n)$
   3. $\Theta(\log_a n) > \Theta(\log_b n)$
   4. $\Theta(\log_a n)$ et $\Theta(\log_b n)$ sont incomparables

Exercice

- Montrer que

$$\Theta(f+g) = \Theta(\mathrm{Max}\{f,g\}).$$

- Soit $p(n) = a_k n^k + \ldots + a_1 n + a_0$ un polynôme de degré k. Montrer que

$$p(n) \in \Theta(n^k).$$

Exercice

Supposons que $f(n) \in \Theta(g(n))$. Est-ce qu'il est vrai que $2^{f(n)} \in \Theta(2^{g(n)})$?
Dans le cas affirmatif prouvez le, sinon donnez un contre-exemple.

définition :

On définit de la façon suivante l'ordre partiel sur l'ordre de grandeur des fonctions :

$$\Theta(f) \leq \Theta(g) \; si \; f \in \mathcal{O}(g)$$

On a alors:

$$\Theta(f) = \Theta(g) \; si \; f \in \Theta(g)$$

$$\Theta(f) < \Theta(g) \; si \; f \in \mathcal{O}(g) et g \notin \mathcal{O}(f)$$

Enumerer en ordre croissant l'ordre exact des fonctions suivantes:
n, 2^n, $n \log n$, $\ln n$, $n + 7n^5$, $\log n$, $\sqrt{n}$, e^n, 2^{n-1}, n^2, $n^2 + \log n$, $\log \log n$, n^3, $(\log n)^2$, $n!$, $n^{3/2}$.
(log et ln dénotent respectivement la fonction de logarithme en base de 2 et en base de e.)

10^6 opérations par seconde.

	c	$\log n$	n	n^3	2^n
10^2	c	6.6 μs	0.1 ms	1 s	$4 \cdot 10^6$ années
10^3	c	9.9 μs	1 ms	16.6 mn	-
10^4	c	13.3 μs	10 ms	11.5 jours	-
10^5	c	16.6 μs	0.1 s	347 années	-
10^6	c	19.9 μs	1 s	10^6 années	-

proposition 4 : (admise)

- $\lim_{n\to\infty} \frac{g}{f} = c > 0$ alors $g \in \Theta(f)$
- $\lim_{n\to\infty} \frac{g}{f} = 0$ alors $g \in \mathcal{O}(f)\ et\ f \notin \mathcal{O}(g)$
  ($\Theta(g) < \Theta(f)$)
- $\lim_{n\to\infty} \frac{g}{f} = +\infty$ alors $f \in \mathcal{O}(g)\ et\ g \notin \mathcal{O}(f)$
  ($\Theta(f) < \Theta(g)$)

On va comparer les algorithmes selon l'ordre de grandeur de leur complexité.

définition :

La complexité $C(n)$ d'un problème P est la complexité du meilleur algorithme qui résoud P.

- **Si un algorithme A résoud P en temps $f(n)$ alors $C(n) \in \mathcal{O}(f)$**
- **Si l'on prouve que tt algorithme qui résoud P travaille en temps au moins $g(n)$ alors $C(n) \in \Omega(g)$**
- **Si $f \in \Theta(g)$ alors $C(n) \in \Theta(f) = \Theta(g)$ et c'est la complexité du problème**

Exercice

1. Supposons que
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$
   1. On a toujours $\Theta(f) < \Theta(g)$
   2. On a toujours $\Theta(f) = \Theta(g)$
   3. On a toujours $\Theta(f) > \Theta(g)$
   4. $\Theta(f)$ et $\Theta(g)$ peuvent être incomparables
2. Supposons que $\Theta(f) = \Theta(g)$. Est-ce que $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
   1. existe toujours et $= \infty$
   2. existe toujours et $= 0$
   3. existe toujours et $= c$, où c est une constante positive
   4. n'existe pas toujours.

Exercice

Montrer que Θ est une relation d'équivalence.

Un rapide rappel

Rappel

Un arbre est un graphe non-orienté acyclique et connexe

Définition

Un arbre enraciné (rooted tree) est un arbre avec un sommet distingué.
Le sommet distingué s'appelle la racine (root) de l'arbre.

Définition

Soit $T = (V,E)$ un arbre enraciné au sommet r, x un sommet de T.

• si $\{y,x\}$ est une arête de T et y est un ancêtre de x, alors y est le père de x, et x est un fils de y.

• la racine de l'arbre n'a pas de père. si x et y ont le même père, alors ils sont frêres.

• un sommet sans fils est un sommet externe ou feuille (leaf). les autres sommets sont les sommets internes

• le nombre de fils d'un sommet x est le degré de x.

Définition

La profondeur du sommet x est la longueur de la chaine entre la racine et le sommet x.

La hauteur (ou profondeur) de l'arbre est :

$$\operatorname*{Max}_{x \in V} \{profondeur(x)\}$$

Les arbres binaires sont définis récursivement.

Définition

un arbre binaire est une structure sur un ensemble fini de sommets.
Il est :

- soit vide (ne contient pas de sommet)
- soit l'union de 3 ensemble disjoints de sommets :
  - un sommet appelé racine
  - un arbre binaire : le sous-arbre gauche
  - un arbre binaire : le sous-arbre droit

La racine d'un sous-arbre gauche non vide est appelé le fils gauche de la racine de l'arbre.

Définition

Soit a un réel non négatif.

$\lfloor a \rfloor$ est le plus grand entier qui est $\leq a$.

$\lceil a \rceil$ est le plus petit entier qui est $\geq a$.

exemples

$$\lfloor \pi \rfloor = 3$$
$$\lceil \pi \rceil = 4$$
$$\lfloor 5 \rfloor = \lceil 5 \rceil = 5$$

notation

$\log$ désigne le logarithme en base 2.

Théorème 1

La profondeur d'un arbre binaire à n sommets est au moins $\lfloor \log n \rfloor$

Preuve

a. A la profondeur l, il y a au plus 2^l sommets dans l'arbre.

Parce que : c'est vrai à la profondeur 0, et si c'est vrai à la profondeur $l-1$, à la profondeur l il y a au plus 2 fois autant de sommets, donc au plus 2^l.

b. Un arbre binaire de profondeur d a au plus $2^{d+1}-1$ sommets.

Parce que: on somme les sommets selon leur profondeur.

et alors : $Nombre\ de\ sommets \leq \sum_{l=0}^{d} 2^l = 2^{d+1} - 1$

c. Soit d la profondeur de l'arbre. On suppose que $d < \lfloor \log n \rfloor$, alors $d \leq \lfloor \log n \rfloor - 1$. Il vient que $n \leq 2^{\lfloor \log n \rfloor} - 1 \leq 2^{\log n} - 1 < n$ ce qui est une contradiction.

TRI (sorting)

Input : Une liste de n entiers $(a_1, a_2, \ldots, a_n)$

Output : Une permutation $(a'_1, a'_2, \ldots, a'_n)$ de la liste d'entrée telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$ ou $a'_1 \geq a'_2 \geq \ldots \geq a'_n$.

Souvent les entiers sont une partie d'une donnée plus complexe, l'entier est alors la clé de la donnée. Et on triera les données d'après les clés.

La structure de donnée est une liste A à n éléments avec $A[i] = a_i$ au début et $A[i] = a'_i$ à la fin.

L'opération de base pour les algorithmes de tri est la comparaison : $A[i] : A[j]$, c'est la seule opération permise sur les clés.

Une autre opération est l'échange : on peut échanger $A[i]$ et $A[j]$, ce que l'on note par $A[i] \leftrightarrow A[j]$.
Le résultat est alors le suivant :

$$k := A[i]$$
$$A[i] := A[j]$$
$$A[j] = k$$

Un algorithme de tri est stable s'il préserve l'ordre de départ des éléments avec la même clé.

Un algorithme de tri se fait sur place s'il utilise seulement un nombre constant de variables auxiliaires.

Tri par selection

Idée :

On recherche le minimum de la liste et on le met en première position, et on recommence sur la fin de la liste où l'on a ajouté l'élément qui se trouvait en première place.

Après le k-ième placement, les k plus petits éléments de la liste sont à leur place définitive.

Exemple :

Input : 101 115 30 63 47 20

selection de 20

Placement : 20 115 30 63 47 101

selection de 30

Placement : 20 30 115 63 47 101

selection de 47

Placement : 20 30 47 63 115 101

selection de 63

Placement : 20 30 47 63 115 101

selection de 101

Placement : 20 30 47 63 101 115

output : 20 30 47 63 101 115

```
procedure TRI-SELECTION
Input :
        A : array [1..n] of integer
var i, j, k : integer
begin
        i :=1
        while i<n do begin { i-ème placement }
            j:=i
            for k:=i+1 to n do
                if A[k]<A[j] then j:=k
            A[j]↔A[i] { placement du minimum }
        end
end
```

Complexité

- Nombre de comparaisons en itération i :

$$(n - i + 1) - 1 = n - i$$

- Nombre total de comparaisons :

$$\sum_{i=1}^{n-1}(n - i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = C_n^2 \in \Theta(n^2)$$

Tri par insertion (séquentielle)

Idée :

• On fait n itérations.

• Après l'itération $(i-1)$ les premiers $(i-1)$ éléments de la liste sont triés.

• A l'itération i on insère d'une manière séquentielle le i-ème élément parmi les premiers $(i-1)$ éléments.

Exemple :

Input : 101 115 30 63 47 20

Itération 1 : 101 115 30 63 47 20

Itération 2 : 101 115 30 63 47 20

Itération 3 : 30 101 115 63 47 20

Itération 4 : 30 63 101 115 47 20

Itération 5 : 30 47 63 101 115 20

Itération 6 : 20 30 47 63 101 115

output : 20 30 47 63 101 115

```
procedure TRI-INSERTION
Input :
        A : array [1..n] of integer
var i, x, k : integer
begin
for i:=2 to n do begin
        k:=i-1 ; x:=A[i]
        while A[k]>x do begin
                A[k+1]:=A[k] ; k:=k-1 end
        A[k+1]:=x
end
end
```

Complexité

- Nombre de comparaisons à l'itération i :

$$\text{au plus } i$$

- Nombre total de comparaisons est au plus:

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 \in \mathcal{O}(n^2)$$

On considère la méthode suivante,

réalisant le tri par insertion d'un tableau T à N éléments, numérotés de 1 à N :

- pour chaque $i \in [\,2, \ldots ,N\,]$,
  - $j := i - 1$.
  - tant que $(j > 0)$ et $T[j] > T[j+1]$,
    - échanger les contenus de $T[j]$ et $T[j+1]$.
    - $j := j - 1$.

Rappelons que dans le pire des cas, le nombre d'échanges nécessaires est égal à $1 + 2 + \ldots (N-1) = \frac{N(N-1)}{2}$.

Exercice 1

Soient $(a_1, \ldots, a_p)$ et $(b_1, \ldots, b_p)$ deux suites d'entiers dont les éléments sont triés

par ordre croissant. Soit $T = (a_1, b_1, a_2, b_2, \ldots, a_p, b_p)$.

Démontrer, par récurrence sur p, que le tri par insertion de T nécessite, dans le pire des cas,

seulement $\frac{p(p+1)}{2}$ échanges.

Exercice 2

On considère à présent un tableau T arbitraire

de taille $n = 2 \times p$, et l'on considère la méthode suivante :

- trier sur place, par insertion, le sous-tableau de T d'indices paires,
- trier sur place, par insertion, le sous-tableau de T d'indices impaires,
- trier, par insertion, le tableau T résultant.

Quel est, dans le pire des cas, le coût de ce tri ? Comparer la valeur obtenue avec celle

du tri par insertion simple. Est-elle meilleure ou pire ?

Tri rapide (quicksort)

Idée :

On partage la liste à trier en deux sous-listes telles que tous les éléments de la première soient plus petits que les éléments de la seconde. Par récurrence, on trie les deux sous-listes.

Comment partager la liste en deux ?
On choisit une des clés de la liste, et on l'utilise comme pivot.

La liste d'origine est donc coupée en trois : une première liste (à gauche) composée des éléments $\leq$ au pivot, le pivot (à sa place définitive) et une liste (à droite) composée des éléments $>$ au pivot.

Notre choix : le pivot est le premier élément de la liste.

Supposons qu'on a une procédure PARTITION(A,i,j,k) qui effectue la partition de la sous-liste de A entre les indices i et j, en fonction du pivot A[i], et rend comme résultat l'indice k, où ce pivot a été placé.

Avec cette procédure, on peut écrire une procédure TRI-RAPIDE.

```
procedure TRI-RAPIDE
Input :
        A : array [1..n] of integer
        i, j : integer
var k : integer
begin
if i<j then begin
                PARTITION(A,i,j,k)
                TRI-RAPIDE(A,i,k-1)
                TRI-RAPIDE(A,k+1,j)
        end
end
```

```
procedure PARTITION
Input : A : array [1..n+1] of integer
        i, j, k : integer
var l : integer
begin
        l:=i+1; k:=j
        while l<=k do begin
                while A[l]<=A[i] do l:=l+1
                while A[k]>A[i] do k:=k-1
                if l<k then do begin
                    A[l]↔A[k]; l:=l+1; k:=k-1; end
        end
A[i]↔A[k]
end
```

Complexité

- Complexité en pire cas : $\Theta(n^2)$

En effet, le pire cas se produit lorsqu'une des partitions est vide et l'autre est de taille $n - 1$.

Dans ce cas, on a :

$$t_1 = 1$$
$$t_n = t_{n-1} + n$$

Et la solution de cette équation de récurrence est : n^2

Complexité

- Complexité en meilleur cas : $\Theta(n \times \log(n))$

En effet, le meilleur cas se produit lorsque les deux partitions sont de même taille.
Dans ce cas, on a :

$$t_1 = 1$$
$$t_n = 2t_{n/2} + n$$

Et la solution de cette équation de récurrence est : $n \times \log(n) + n$

Complexité

- Complexité en moyenne : $\mathcal{O}(n \log n)$

$$c_1 = 0$$
$$c_n = \frac{1}{n-1} \sum_{i=1}^{n-1} (c_i + c_{n-i}) + n$$

Maintenant, on retrousse ses manches :

$$c_n = \frac{2}{n-1} \sum_{i=1}^{n-1} c_i + n$$
$$c_n = \frac{2}{n-1} \times c_{n-1} + \frac{2}{n-1} \sum_{i=1}^{n-1} c_i + n$$
$$c_n = \frac{2}{n-1} \times c_{n-1} + \frac{2}{n-1} \sum_{i=1}^{n-1} c_i + n$$
$$c_n = \frac{2}{n-1} \times c_{n-1} + \frac{n-2}{n-1} \left(\frac{2}{n-2} \sum_{i=1}^{n-1} c_{i-1} + n - 1 \right) + 2$$
$$c_n = \frac{n}{n-1} \times c_{n-1} + 2$$

et maintenant les choses se corsent...

On divise à droite et à gauche par n :

$c_n/n = c_{n-1}/(n-1) + 2/n$

Puis on pose $Y_n = c_n/n$, on a alors :

$Y_n = Y_{n-1} + 2/n$, ce qui correspond à l'expression d'une série harmonique et qui donne :

$c_n = 2 \times n \times \sum_{i=1}^{n}(1/i)$

Par la formule d'euler, on trouve alors

$c_n \simeq 1{,}386 \times n \times \log(n) + 1{,}154 \times n$

c'est à dire une fonction en $n \times \log(n)$.

Tri Fusion

Le principe

En appliquant la méthode diviser pour régner au problème du tri d'un tableau, on obtient facilement le tri par fusion.

On appelle ce tri mergesort en anglais.

Le principe est de diviser une table de longueur n en deux tables de longueur n/2, de trier récursivement ces deux tables, puis de fusionner les tables triées.

Le principe

Première phase : Découpe :

1. Découper le tableau en 2 sous-tableaux égaux (à 1 case près)
2. Découper chaque sous-tableau en 2 sous-tableaux égaux
3. Ainsi de suite, jusqu’à obtenir des sous-tableaux de taille 1

Le principe

Deuxième phase : Tri/fusion :

1. Fusionner les sous-tableaux 2 à 2 de façon à obtenir des sous-tableaux de taille 2 triés
2. Fusionner les sous-tableaux 2 à 2 de façon à obtenir des sous-tableaux de taille 4 triés
3. Ainsi de suite, jusqu’à obtenir le tableau entier

Exercice

- Donner un code, dans le langage que vous voulez, qui implémente le Tri Fusion.
- Dérouler votre code sur la liste d'entiers :

3 6 78 32 5 1 10 23 9 11 33

Exercice : complexité

Quelle est la complexité du Tri Fusion ?

Tri à Bulle

Le principe

- On compare les éléments deux à deux de la gauche vers la droite, en les échangeant pour les classer quand c'est nécessaire.
- Quand on arrive à la fin de la liste, on recommence, jusqu'à ce que la liste soit classée.
- On détecte que la liste est classée quand on a fait un passage sans faire d'échanges.

Exercice

- Ecrire une procédure Tri-Bulle
- Dérouler l'algorithme sur la liste :

2 4 7 5 3 1

Combien avez vous fait de comparaisons ?

- Quelle est la complexité du tri à bulle ?

Variantes :

- Tri Boustrophedon
- Tri Shuttle

TRI PAR TAS (HEAPSORT)

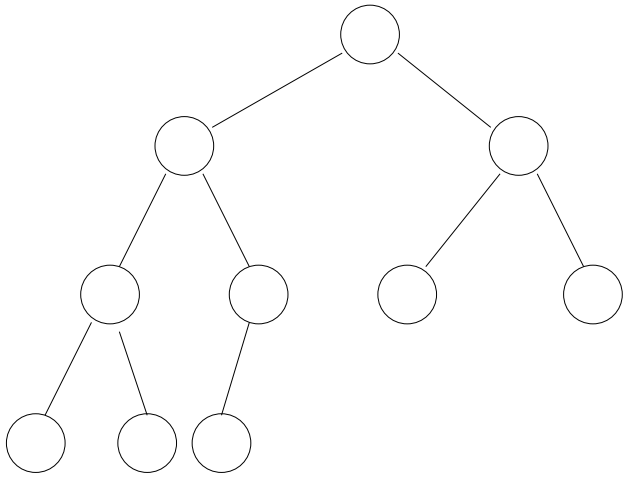
Idée :

On sélectionne les minimums successifs en utilisant une structure de donnée particulière : le tas

Définition

Un arbre parfait est un arbre binaire dont les feuilles sont situées sur deux niveaux successifs : l'avant dernier niveau est complet, et les feuilles du dernier niveau sont regroupées le plus à gauche possible.

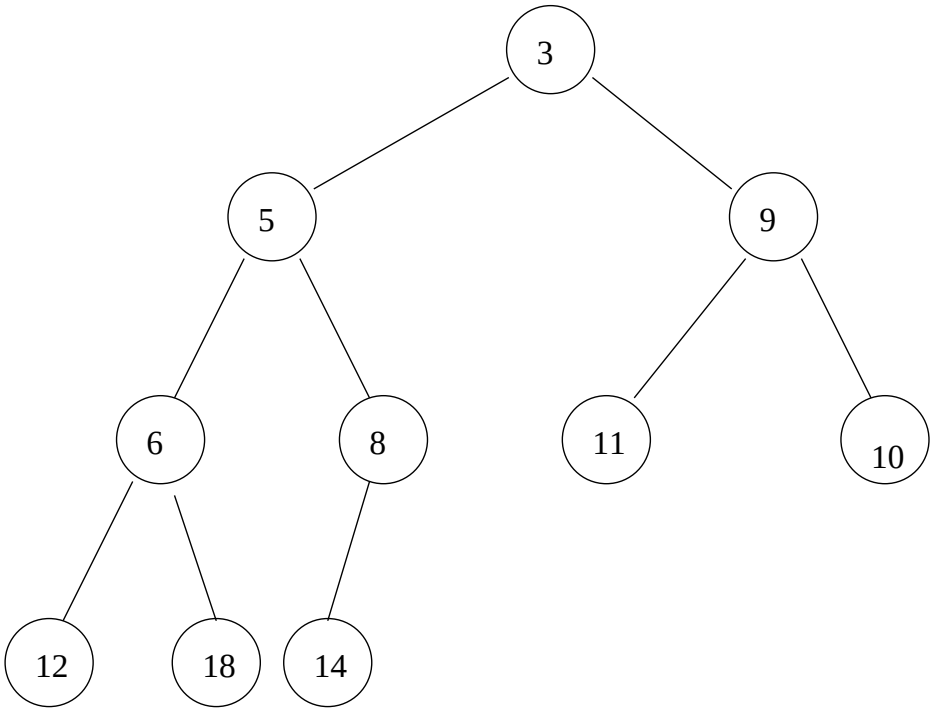
Exemple



Définition

Un arbre parfait partiellement ordonné est un arbre parfait dont les sommets sont étiquetés par des éléments à trier, et tel que l'élément associé à chaque sommet est $\leq$ aux éléments associés aux fils de ce sommet.

Exemple



Définition

Un tas (heap) est un tableau t représentant un arbre parfait partiellement ordonnée selon les règles suivantes :

- $t[1]$ est la racine
- $t[2i]$ et $t[2i+1]$ sont les fils gauche et droite de $t[i]$ (s'ils existent)

Exemple

3	5	9	6	8	11	10	12	18	14

Propriétés simples d'un tas

- $t[\lfloor i/2 \rfloor]$ est le père de $t[i]$ ($i \geq 2$)
- Si le nombre de sommets n est pair, alors $t[n/2]$ a un seul fils $t[n]$
- Si $i > \lfloor n/2 \rfloor$, alors $t[i]$ est une feuille

Idée du tri par tas

I. On construit le tas

II. On enlève successivement le minimum du tas et on reconstruit la structure de tas sur le reste des éléments

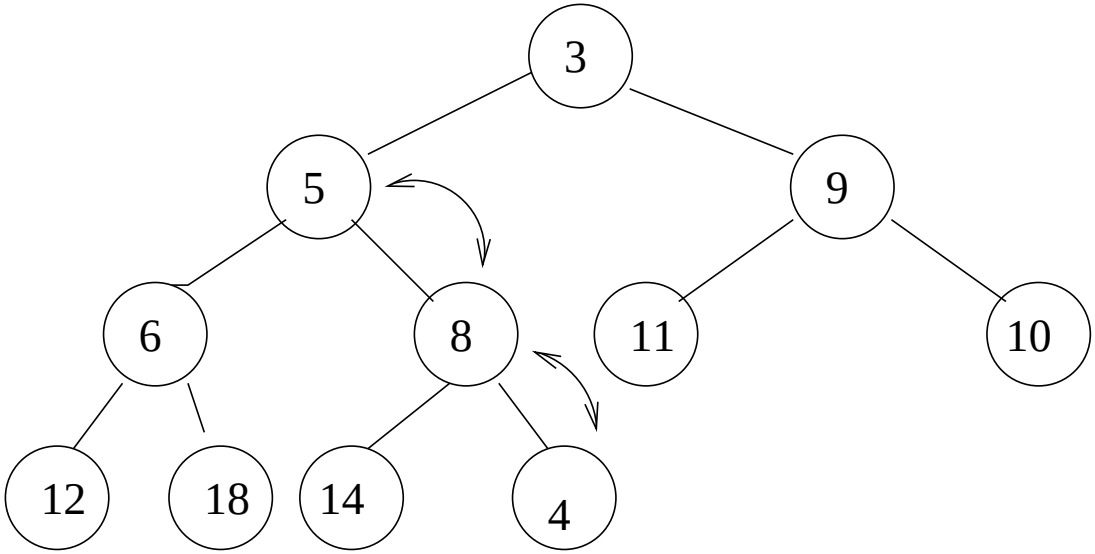
CONSTRUCTION DU TAS

L'essentiel est une procédure AJOUT qui ajoute un élément nouveau à un tas.

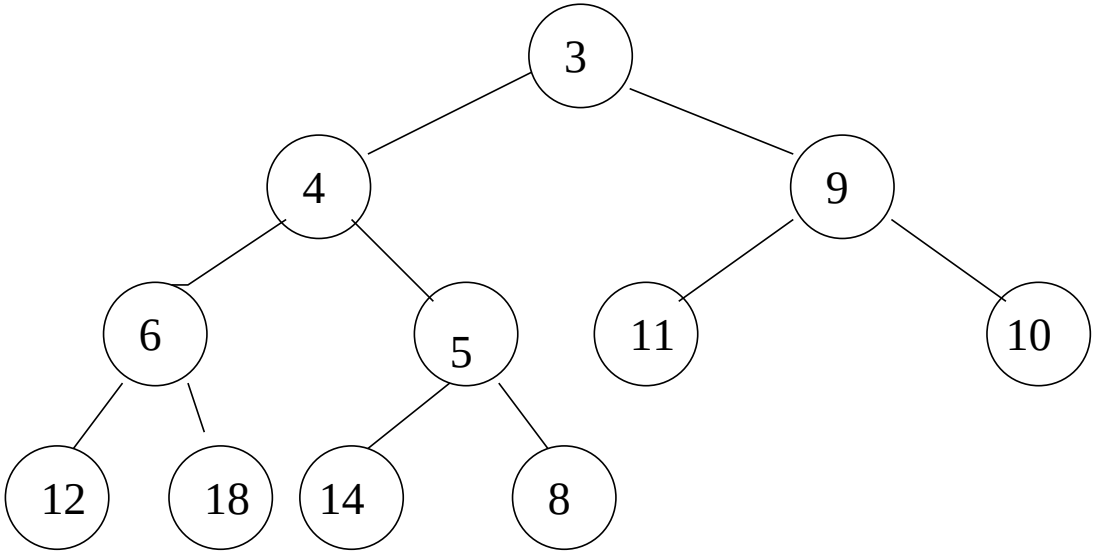
On ajoute cet élément comme une feuille (au dernier niveau), puis on l'échange avec son père tant qu'il est inférieur à celui-ci.

```
procedure AJOUT
Input : t : array [1..n] of integer ; p, x : integer {la procédure ajoute l'élément x
dans le tas t qui contient p éléments}
var i : integer
begin
    if p<n then begin {on place x à la dernière feuille}
        p:=p+1 ; t[p]:=x ; i:=p ;
{on échange tant que x est < que son père}
        while i>1 and t[i] < t[⌊i/2⌋] do begin
            t[i] ↔ t[⌊i/2⌋] ; i := ⌊i/2⌋
        end
    end
else writeln("débordement du tas")
end
```

Exemple



Résultat final



Complexité en pire cas

La procédure AJOUT fait au plus $\lfloor logp \rfloor$ comparaisons.

Pourquoi ?

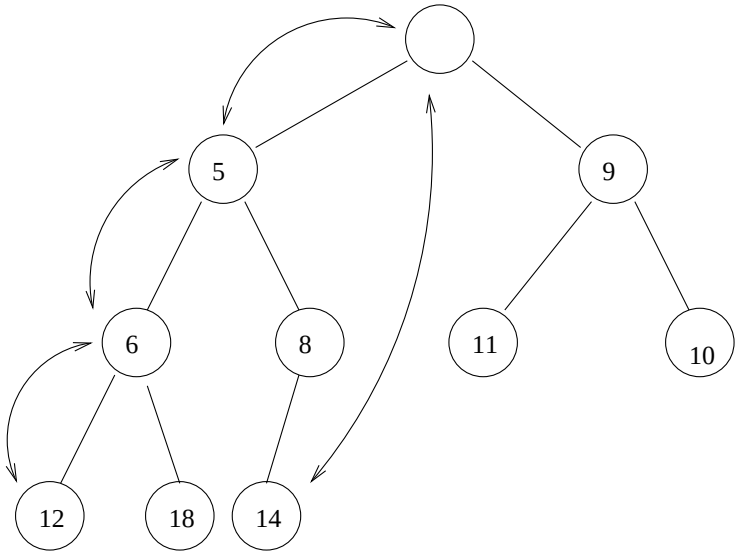
Parce que la hauteur d'un arbre parfait à p sommets est $\lfloor \log p \rfloor$.

SUPPRESSION DU MINIMUM ET REORGANISATION DU TAS

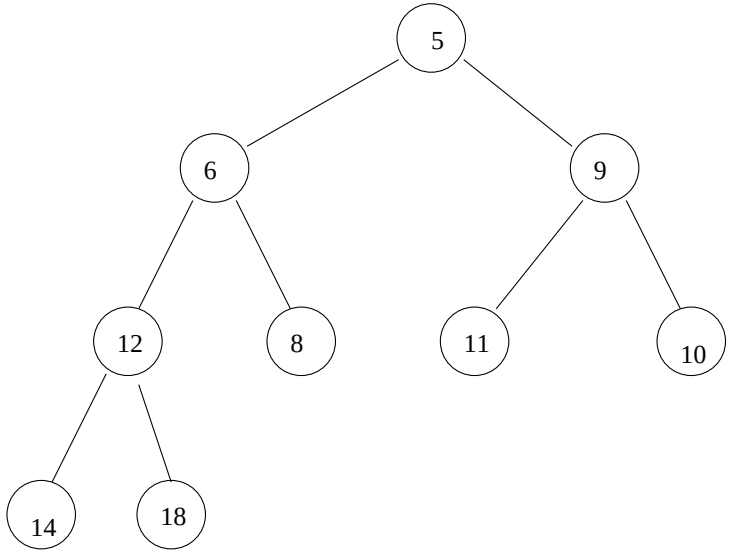
La procédure DESTRUCTION remplace la racine du tas par la dernière feuille.

Pour réorganiser le tas, on échange cet élément avec le plus petit de ses deux fils tant qu'il est supérieur à celui-ci.

Exemple



Résultat final



```
procedure DESTRUCTION
Input :
    t : array [1..n] of integer
    p, min : integer

{DESTRUCTION retourne dans min le minimum des p éléments du tas et
réorganise le tas}

var i, j : integer

begin

    min:=t[1]        {retour du minimum}

    t[1] := t[p] ; p:=p-1        {nouvelle taille du tas}

    i:=1        {réorganisation du tas}
```

```
    while i ≤ ⌊p/2⌋ do begin
    {calcul de l'indice du plus petit des deux fils de t[i], ou de son seul fils}
      if 2i=n or t[2i] < t[2i + 1] then j:=2i else j:=2i+1
    {échange entre deux éléments qui sont père et fils}
      if t[i] > t[j] then begin t[i] ↔ t[j] ; i:=j end
      else exit
    end
end
```

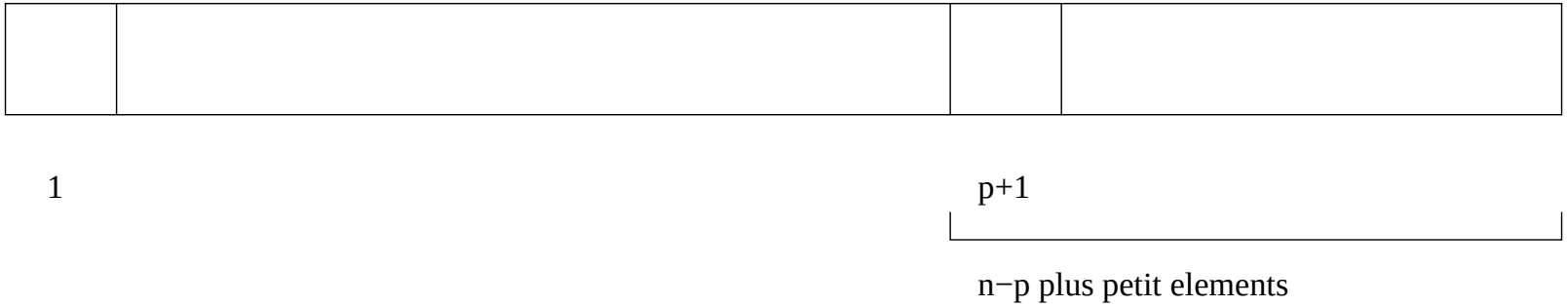
Complexité de DESTRUCTION en pire cas :

$$2\lfloor \log p \rfloor$$

L'algorithme TRI-TAS

• On ajoute successivement chacun des n éléments dans le tas $t[1 \ldots p]$, p augmente de 1 après chaque adjonction. A la fin on a un tas de taille n.

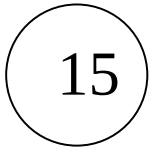
• Tant que $p > 1$, on supprime le minimum du tas, on décroit p de 1, et on range le minimum à la $(p+1)eme$ place. Puis on réorganise le tas. $t[p+1 \ldots n]$ contient les $n-p$ plus petits éléments en ordre décroissant.



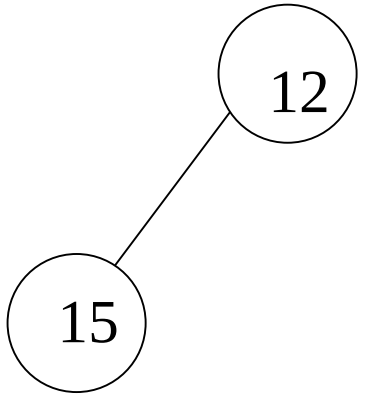
```
procedure TRI-TAS
Input : t : array [1..n] of integer
var p, min : integer
begin
    p:=0
    while p < n do {construction du tas}
        AJOUT(t,p,t[p+1]);
        {p augmente de 1 à chaque appel}
    while p > 1 do begin
        DESTRUCTION(t,p,min)
        {p diminue de 1 à chaque appel}
        t[p+1]:=min
    end
end
```

Exemple

15 | 12 11 7 13 9

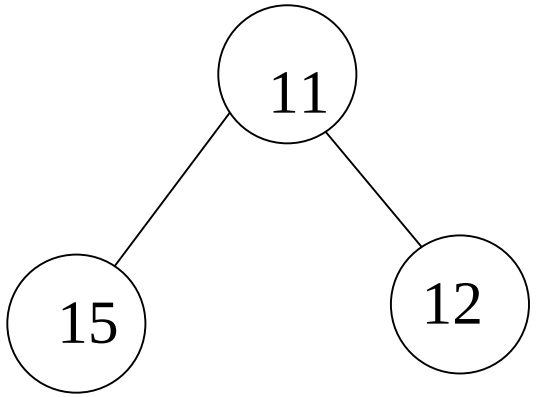


12 15 | 11 7 13 9

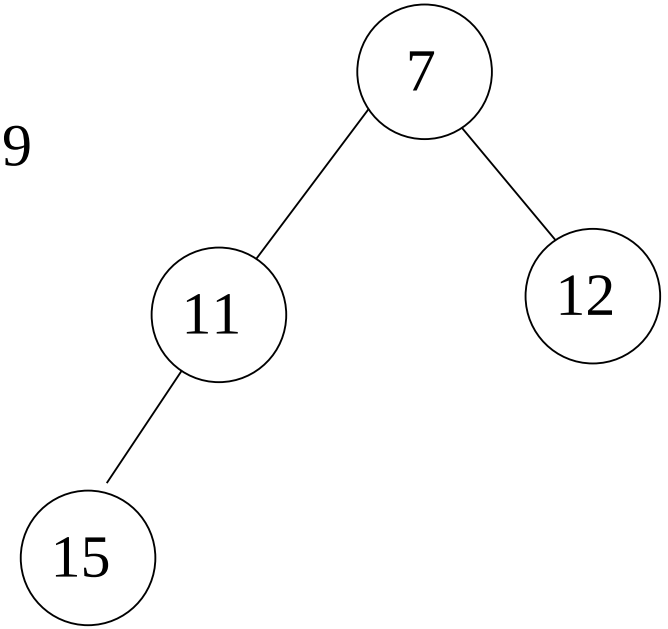


Exemple (suite)

11 12 15 | 7 13 9

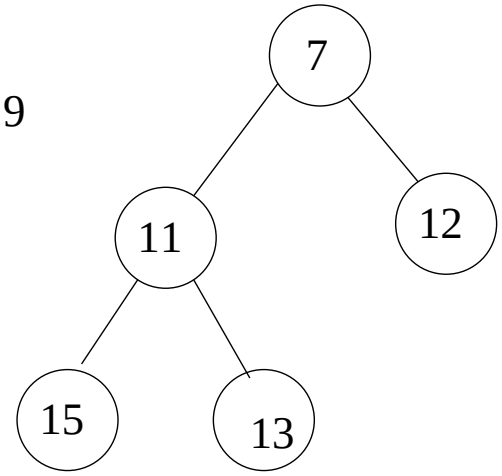


7 11 12 15 | 13 9

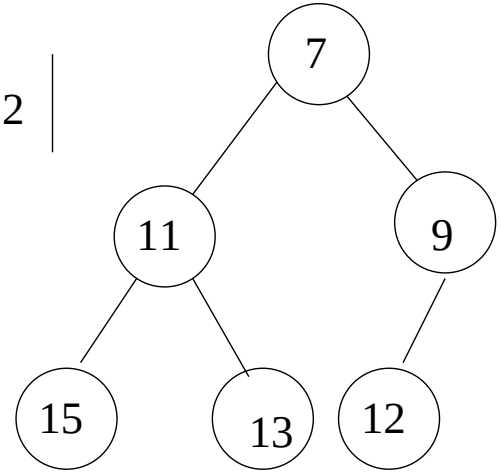


Exemple (suite)

7 11 12 15 13 | 9

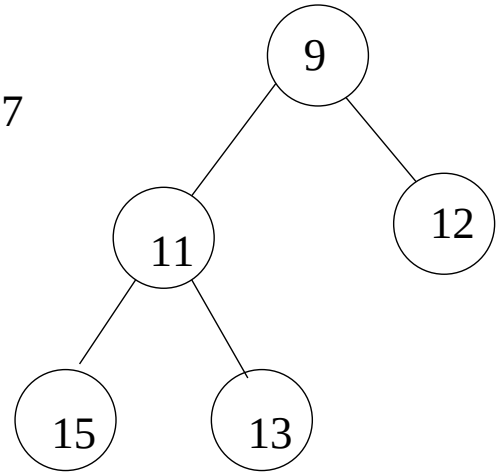


7 11 9 15 13 12 |

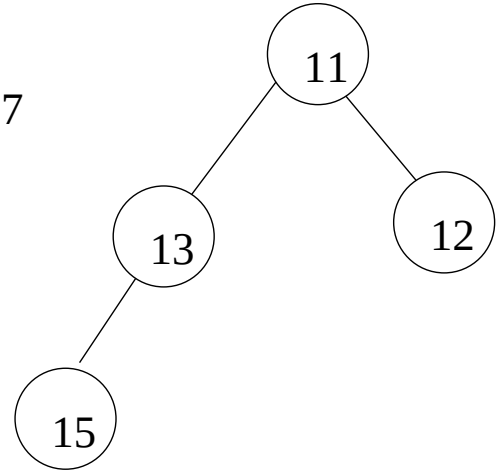


Exemple (suite)

9 11 12 15 13 | 7

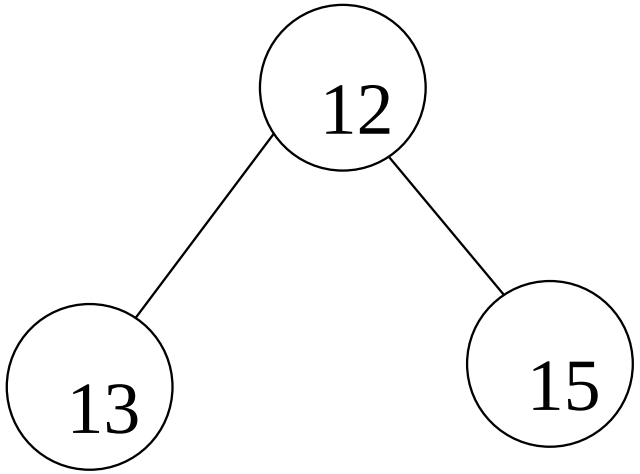


11 13 12 15 | 9 7

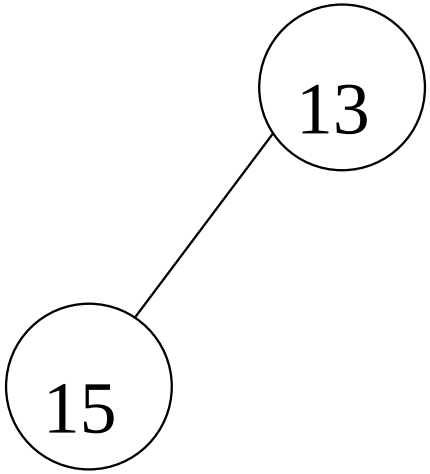


Exemple (suite)

12 13 15 | 11 9 7



13 15 | 12 11 9 7



Exemple (fin)

RESULTAT FINAL :

Complexité de TRI-TAS en pire cas

AJOUT : $\sum_{i=2}^{n-1} \log i$

DESTRUCTION : $\sum_{i=2}^{n-1} 2 \log i$

$$W(n) = 3 \sum_{i=2}^{n-1} \log i \leq 3 \int_2^n \log x dx = 3(\log_2 e) \int_2^n \ln x dx$$

$$= 3(\log_2 e)[x \ln x - x]_2^n = 3(\log_2 e)(n \ln n - n - 2 \ln 2 + 2)$$

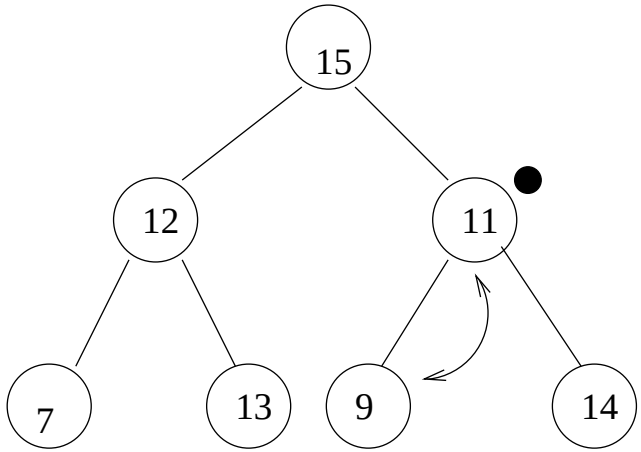
$$= 3(n \log n - n \log_2 e - 2 + 2 \log e) \leq 3n \log n = \mathcal{O}(n \log n)$$

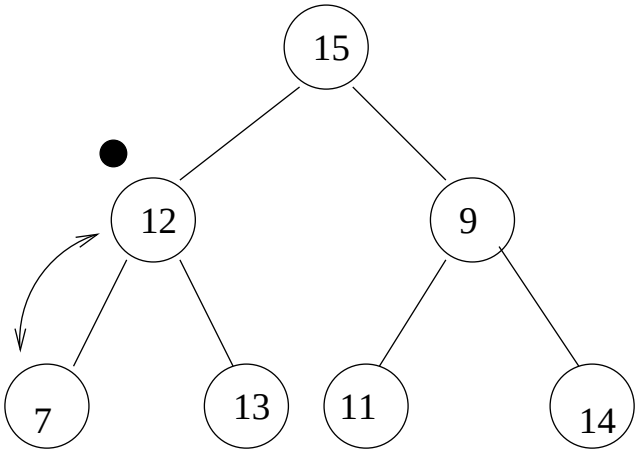
Construction du tas en $\mathcal{O}(n)$

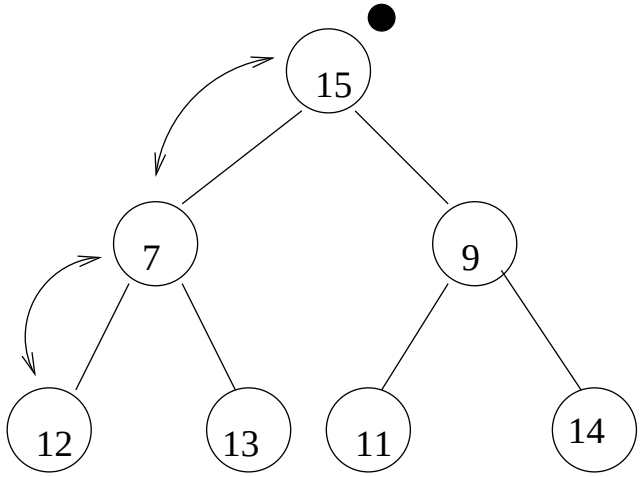
Idée

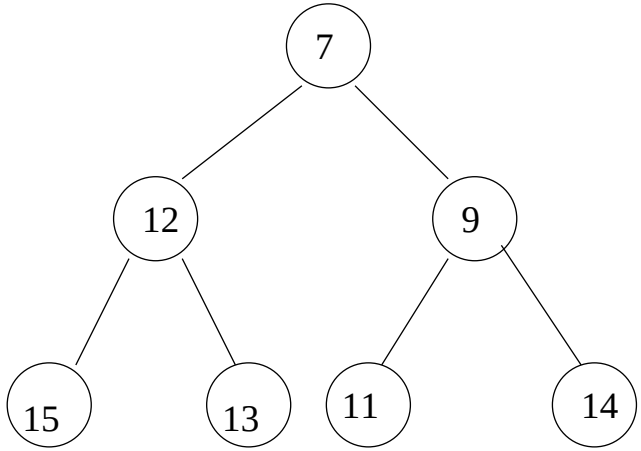
On utilise la partie de la procédure DESTRUCTION qui réorganise le tas. On lance cette sous-procédure successivement à partir de chaque sommet du tas, partant du niveau le plus bas jusqu'à la racine.

Exemple









```
procedure ORGANISATION
Input : t : array [1..n] of integer ; i : integer
{ construit un tas dont la racine est le sommet t[i] ; on suppose que les sous
arbres de t[i] sont déja un tas }

var j : integer

begin

    while i ≤ ⌊n/2⌋ do begin {t[i] n'est pas une feuille}

      if 2i=n or t[2i]<t[2i+1] then j:=2i else j:=2i+1

      if t[i]>t[j] then begin t[i]↔t[j] ; i:=j end

      else exit

    end

end
```

```
procedure CONSTRUCTION
Input : t : array [1..n] of integer
{construit le tas dans le tableau t}
var j : integer
begin
    for j:=n downto 1 do
        ORGANISATION(t,j)
end
```

Théorème

CONSTRUCTION construit le tas avec $\leq$ 2n-2 comparaisons

Exercice

Trier le tableau suivant en utilisant l'algorithme TRI-RAPIDE:

1 21 5 34 32 67 6 0 200 25 65 4 24 33.

Quel est le nombre de comparaisons utilisées?

Exercice

Quelle est la valeur retournée par la procédure PARTITION quand tous les éléments dans le tableau $A[i..j]$ ont la même valeur?

Exercice

On se donne un tableau de n éléments qui ont chacun une couleur soit rouge, soit blanc. On veut obtenir un tableau avec deux zones: à gauche tous les éléments rouges, à droite tous les éléments blancs.

Ecrivez une procédure qui réalise cette organisation du tableau en ne testant qu'une seule fois la couleur de chaque élément. Essayez de minimiser le nombre moyen des échanges d'éléments. Calculez ce nombre.

Recherche dans une liste triée

Input : L,$n \leq 0$,X où L est une liste triées à n éléments et X est l'élément recherché.

Output : l'index de lélément X dans L si $X \in L$, 0 dans le cas contraire.

Algorithme : BINARY SEARCH (recherche binaire ou dichotomique)

Idée :

```
procedure BINARYSEARCH
Input :
        L : array [1..n] of elements
        X : element
        first, last : 1..n
        index : 0..n

{ procédure récursive pour chercher X dans L, first et last sont les
indices respectivement du premier et du dernier élélement de la
section du tableau actuellement recherchée }
```

```
var m : 1..n
begin if first <= last then
      begin m := ⌊(first + last)/2⌋
         { indice de l'élément au milieu de la section }
      if X=L[m] then index:=m
      else if X < L[m] then
            BINARYSEARCH(L,X,first,m-1,index)
         else BINARYSEARCH(L,X,m+1,last,index)
      end
   else index := 0 end
Appel principal : BINARYSEARCH(L,X,1,n,index)
```

Complexité en pire cas de BINARY SEARCH

Soit $T(n) = Worst_{BINARYSEARCH}(n)$

Alors : $T(1) = 1$ et $T(n) = T(\lfloor n/2 \rfloor) + 1$

Théorème

La solution de cette équation de récurrence est :

$$T(n) = \lfloor \log n \rfloor + 1$$

Preuve

Admise (théorème fondamental du principe de récurrence du cours sur les paradigmes)

On voudrait maintenant montrer que BINARYSEARCH est un algorithme optimal pour ce problème

Un algorithme de comparaison ne peut faire d'autres opérations que de comparer X avec les éléments de la liste.

On va se concentrer sur le cas où X n'est pas dans la liste. Une preuve dans ce cas là est un énoncé de la forme :

- $L[i] < X < L[i+1]$ pour $i = 1,...,n-1$
- $X < L[1]$
- $L[n] < X$

Un algorithme de comparaison peut seulement s'arréter au cas où X n'est pas dans la liste s'il possède une preuve.

Définition

L'arbre de décision (comparaison) pour un algorithme A de comparaison qui recherche un élément dans une liste triée à n éléments est l'arbre binaire T_A dont les sommets internes sont étiquetés par les entiers entre 1 et n selon les règles suivantes :

1. La racine de T_A est étiquetée par l'indice du premier élément avec lequel A compare X

2. Supposons que l'étiquette d'un sommet soit i. L'étiquette du fils gauche de ce sommet est l'indice de l'élément avec lequel A compare X si $X < L[i]$.
Analogue pour le fils droite....

Les feuilles de T_A sont étiquetées par les preuves pour le cas où X n'est pas dans L.

T_A peut être étendu en rajoutant une feuille à chaque sommet interne étiqueté par i :
cette feuille correspond au résultat $X = L[i]$

Exemple :

Théorème

Chaque algorithme de comparaison A pour rechercher un élément X dans une liste triée à n éléments fait au moins $\lfloor \log n \rfloor + 1$ comparaisons en pire cas.

Preuve

Soit T_A l'arbre de décision pour A. Sur chaque entrée, A parcourt de façon naturelle une branche de T_A. Le nombre de comparaisons effectuées par A est la longueur de la branche. La complexité en pire cas de A est donc la longueur de la branche la plus longue, c'est à dire la profondeur de T_A.

Preuve (suite) :

Montrons par l'absurde que le nbre de sommets internes de T_A est $\geq n$. Sinon, $\exists i, 1 \leq i \leq n$ tel qu'aucun sommet interne n'est étiqueté par i. Soient deux listes L et L' tq
$\forall j \neq i,\ L[j] = L'[j] \neq X$ et $L[i] = X,\ L'[i] \neq X$.

A ne peut pas distinguer ces deux listes (voir T_A étendu), donc il donne un résultat incorrect.

Par conséquent, la profondeur de T_A est au moins $\lfloor \log n \rfloor + 1$.

Recherche du plus grand élément (maximum) dans une liste.

Input : L,$n \leq 0$ où L est une liste de n entiers.

Output : max, le plus grand élément de L.

```
procedure FINDMAX
L : array [1..n] of integer
var max : integer
var i : integer
begin
        max := L[i];
        for i:=2 to n do
                if max < L[i] then max:=L[i]
end
```

Complexité : $Worst_{FINDMAX}(n) = Av_{FINDMAX}(n) = n - 1$.

Théorème :

FINDMAX est un algorithme optimal.

Preuve :

On montre que chaque algorithme de comparaison A fait en pire cas au moins $n-1$ comparaisons pour trouver le maximum de n éléments.
Parmi les n éléments, $n-1$ ne sont pas le maximum (en pire cas).
Pour chacun de ces éléments A doit faire une comparaison dans laquelle l'élément est plus petit que celui auquel il est comparé.
Par conséquent, A fait au moins $n-1$ comparaisons.

Recherche du plus deuxième plus grand élément dans une liste.

Input : L,$n \leq 0$ où L est une liste de n entiers distincts.

Output : second, le deuxième plus grand élément de L.

Terminologie :

x, y	joueurs (de tennis)
x : y	x joue avec y
x $>$ y	x gagne, y perd

Proposition : chaque algorithme qui trouve second trouve aussi max

Preuve : L'algo. pour second doit avoir une preuve que second est deuxième et pas premier. Cela ne peut se faire qu'en faisant une comparaison où second perd. Cette comparaison se fait alors avec max.

On va donner deux méthodes (algorithmes) pour trouver second.

Méthode naïve

- On cherche max dans L en effectuant $n-1$ comparaisons.
- On cherche le plus grand élément dans le reste en faisant $n-2$ comparaisons.

Au total, l'algorithme naïf a effectué $2n-3$ comparaisons.

Méthode du tournoi (tournament method)

On organise un tournoi de tennis. On fait jouer les participants 2 à 2 en manches (tours). S'il y a un nombre impair de joueurs, un joueur progresse au tour suivant sans jeu. Le vainqueur du tournoi est max.

On peut représenter un tournoi par un arbre binaire dont les feuilles sont les joueurs.

exemple :

Proposition : Dans un tournoi avec n joueurs, il y a $\lceil \log n \rceil$ manches

Preuve : Evidente (profondeur de l'arbre/nombres de sommets)

Rq : second a du perdre dans un jeu contre max !

Pour trouver second, il suffit de trouver le plus grand des éléments battus par max sachant que le nbre de joueurs ayant joué contre max est $\lceil \log n \rceil$.

Complexité :

- Le tournoi se fait en $n - 1$ comparaisons
- Le plus grand de $\lceil \log n \rceil$ éléments se fait en $\lceil \log n \rceil - 1$ comparaisons

Ce qui fait un total de $n + \lceil \log n \rceil - 2$ comparaisons.

Borne inférieure sur le tri

On peut représenter tous les tris que nous avons rencontrés par des arbres de décision. Chaque noeud interne pose une question sur la comparaison entre 2 éléments. Le fils de gauche correspond à la réponse négative, le fils droit à l'affirmatif. Les feuilles représentent la permutation à effectuer pour obtenir le tableau trié.

Théorème :

Pour trier n éléments à l'aide d'un algorithme de tri basé sur les comparaisons, il faut $\Omega(n \log n)$ comparaisons.

Preuve :

Tout arbre de décision pour trier n éléments a $n!$ feuilles représentant toutes les permutations possibles. Un arbre binaire à $n!$ feuilles a une hauteur de l'ordre de $\log(n!)$.
Il reste alors à prouver que $\log(n!) \simeq n \log n$. C'est vrai en vertu de la formule de stirling : $n! \simeq \sqrt{2n\pi} \times (\frac{n}{e})^n$