Les algorithmes de tris itératifs: Tri rapide

Soit à trier le tableau T[g..d] (au départ g=1 et d=n)

Le principe de l'algorithme réside dans une procédure, appelée partition, qui réorganise les éléments de T autour d'un pivot (élément du tableau choisi au hasard) de sorte que :

1) Il existe un indice p (g≤ p ≤d) tel que p est la position définitive du pivot (T[p] = pivot).
2) tous les éléments T[g], …, T[p-1] sont inférieurs ou égaux à T[p].
3) tous les éléments T[p+1], …T[d] sont supérieurs ou égaux à T[p].

Les algorithmes de tris itératifs: Tri rapide

Tri rapide (quick sort)

fonction TRI_RAPIDE (T, g, d)

---

si $(g < d)$ **alors**
  Choisir *pivot* dans $T[g, \ldots, d]$,
  $m :=$ PARTITIONNER$(T, g, d, pivot)$,
  TRI_RAPIDE $(T, g, m-1)$, $\{ m-1 < d \}$
  TRI_RAPIDE $(T, m+1, d)$, $\{ m+1 > g \}$

Les algorithmes de tris itératifs: Tri rapide

La fonction partition, appliquée à un tableau T, produit trois sous-tableaux:

- Un sous-tableau réduit à un seul élément T[p] qui garde sa place définitive dans le tri de T, et
- Deux sous-tableaux T[g .. p-1], T[p+1 .. d].
- Pour trier T, il suffit d’appliquer récursivement le même algorithme sur les deux sous-tableaux.

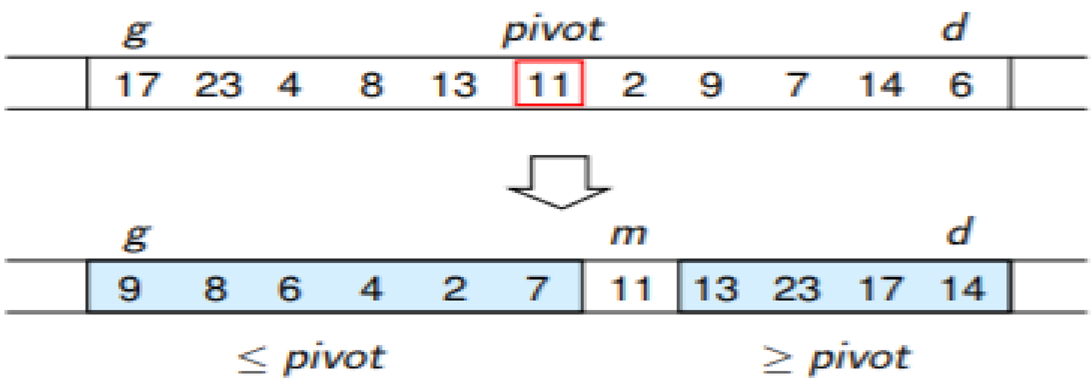
Les algorithmes de tris itératifs: Tri rapide

Tri rapide (quick sort)

fonction TRI_RAPIDE (T, g, d)

```
si (g < d) alors
    Choisir pivot dans T[g, . . . , d],
    m :=PARTITIONNER(T, g, d, pivot),
    TRI_RAPIDE (T, g, m − 1),          { m − 1 < d}
    TRI_RAPIDE (T, m + 1, d),          { m + 1 > g}
```

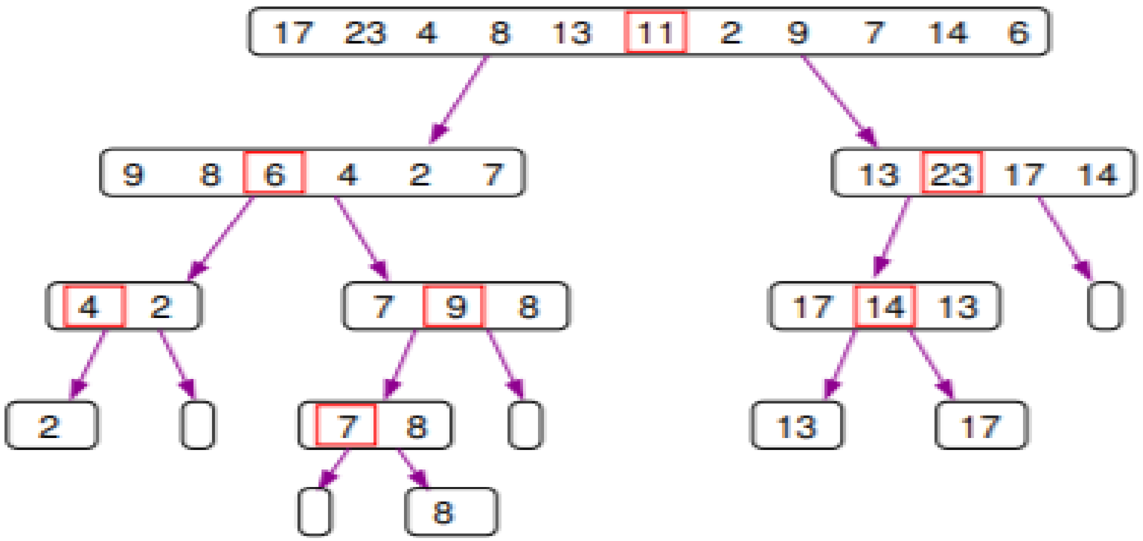
Partition :



Les algorithmes de tris itératifs: Tri rapide

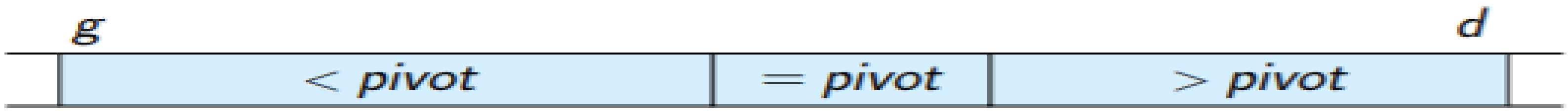
fonction TRI_RAPIDE (T, g, d)

si $(g < d)$ **alors**
- Choisir *pivot* dans $T[g, \ldots, d]$,
- $m :=$ PARTITIONNER$(T, g, d, pivot)$,
- TRI_RAPIDE $(T, g, m-1)$,
- TRI_RAPIDE $(T, m+1, d)$,

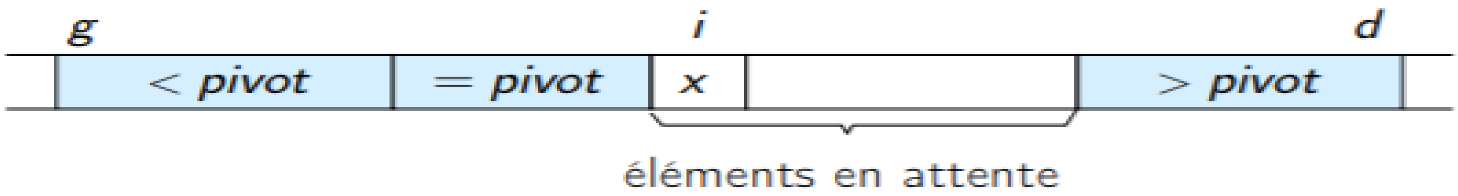


Les algorithmes de tris itératifs: Tri rapide

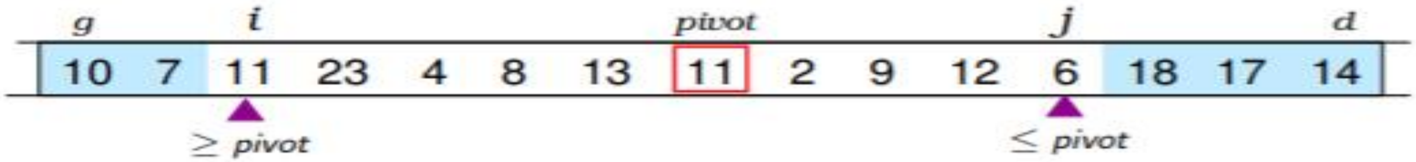
Après la partition :



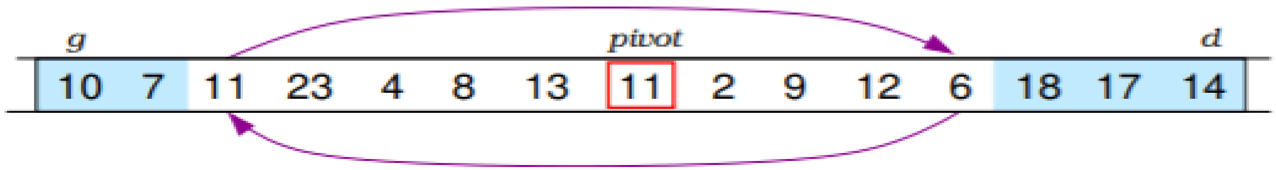
Pendant la partition :



Les algorithmes de tris itératifs: Tri rapide



à gauche : stoppe avec $x_i \geq pivot$
à droite : stoppe avec $x_j \leq pivot$



Permutation

Les algorithmes de tris itératifs: Tri rapide

g							pivot							d
10	7	6	23	4	8	13	11	2	9	12	11	18	17	14

Extension des zones

g							pivot							d
10	7	6	23	4	8	13	11	2	9	12	11	18	17	14

Recherche d'un plus grand à gauche et d'un plus petit à droite

Les algorithmes de tris itératifs: Tri rapide

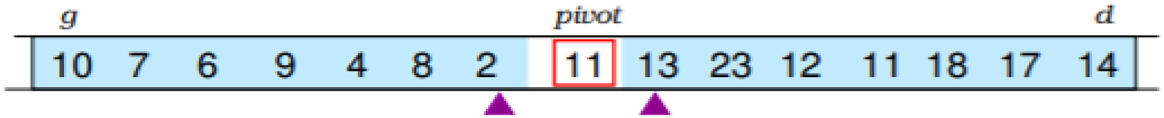
g							pivot							d
10	7	6	9	4	8	13	$\boxed{11}$	2	23	12	11	18	17	14

Permutation et extension des zones

g							pivot							d
10	7	6	9	4	8	13	$\boxed{11}$	2	23	12	11	18	17	14

Recherche d'un plus grand à gauche et d'un plus petit à droite

Les algorithmes de tris itératifs: Tri rapide



Permutation et extension des zones

Dernière permutation

Les algorithmes de tris itératifs: Tri rapide

fonction TRI_RAPIDE (T, g, d)

```
si (g < d) alors
    a := g, b := d,
    v := T[(a + b)/2],
    tant que (a ≤ b) faire
        {∀i, si g ≤ i < a alors T[i] ≤ v, et si b < i ≤ d alors T[i] ≥ v,}
        tant que (T[a] < v) faire
            a := a + 1,                     {T[a − 1] < v}
        tant que (T[b] > v) faire
            b := b − 1,                     {T[b + 1] > v}
        si (a ≤ b) alors
            PERMUTER(T, a, b),
            {T[a] ≤ v et T[b] ≥ v}
            a := a + 1,
            b := b − 1,
        {donc si i < a alors T[i] ≤ v et si i > b alors T[i] ≥ v,}
    TRI_RAPIDE (T, g, b),                   {b < a et ∀i, i < a on a T[i] ≤ v}
    TRI_RAPIDE (T, a, d),                   {a > b et ∀i, i > b on a T[i] ≥ v}
fin fonction
```

Les algorithmes de tris itératifs: Tri à bulles

- L'algorithme consiste à parcourir le tableau à trier en examinant si chaque couple d'éléments consécutifs (a_i, a_{i+1}) est dans le bon ordre ou non, si ce couple n'est pas dans le bon ordre on échange ses éléments et ce processus est répété tant qu'il reste des inversions à faire.
- On dit qu'on a une inversion s'il existe (i,j) tels que $i<j$ et $a_i > a_j$
- $(a_1,\ldots,a_i , a_{i+1},\ldots,a_n) \Longrightarrow (a_1,\ldots,a_{i+1},a_i,\ldots,a_n)$
- Un tableau est trié s'il n'a aucune inversion.

Les algorithmes de tris itératifs: Tri à bulles

Soit les éléments du tableau suivant « **5 1 4 2 8** » ; pour chaque étape, les éléments comparés sont en gras.

Étape 1.

1.1. (**5 1** 4 2 8) => (**1 5** 4 2 8). Les nombres 5 et 1 sont comparés, et comme 5 > 1, l'algorithme les échange.

1.2. (1 **5 4** 2 8) => (1 **4 5** 2 8).Échange, car 5>4.

1.3. (1 4 **5 2** 8) => (1 4 **2 5** 8).Échange, car 5>2.

1.4. (1 4 2 **5 8**) => (1 4 2 **5 8**). Pas d'échange, car 5 < 8.

À la fin de cette étape, un nombre est à sa place définitive, le plus grand : 8.

Étape 2.

2.1. (**1 4** 2 5 8) => (**1 4** 2 5 8). Pas d'échange.

2.2. (1 **4 2** 5 8) => (1 **2 4** 5 8). Échange.

2.3. (1 2 **4 5** 8) => (1 2 **4 5** 8). Pas d'échange.

Les éléments 5 et 8 du tableau ne sont pas comparés puisqu'on sait que le 8 est déjà à sa place définitive. Par hasard, tous les nombres sont déjà triés, mais cela n'est pas encore détecté par l'algorithme.

Les algorithmes de tris itératifs: Tri à bulles

Étape 3.

3.1. (**1 2** 4 5 8) => (**1 2** 4 5 8). Pas d'échange.

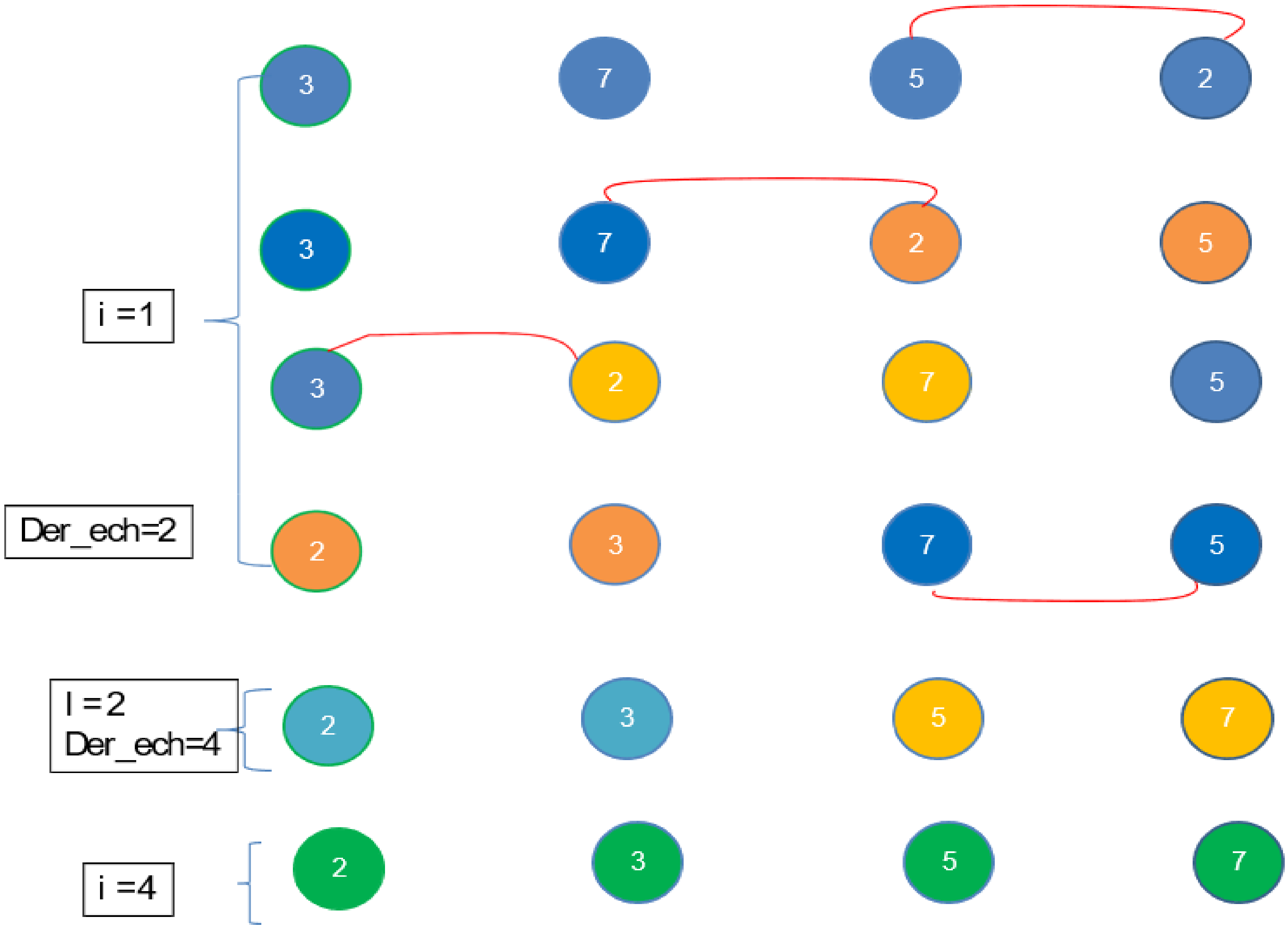
3.2. (1 **2 4** 5 8) => (1 **2 4** 5 8). Pas d'échange.

Les deux derniers nombres sont exclus des comparaisons, puisqu'on sait qu'ils sont déjà à leur place définitive. Puisqu'il n'y a eu aucun échange durant cette étape 3, le tri optimisé se termine.

Étape 4.

4.1. (**1 2** 4 5 8) => (**1 2** 4 5 8). Pas d'échange.

Le tri est terminé, car on sait que les 4 plus grands nombres, et donc aussi le 5^{e}, sont à leur place définitive.


3
7
5
2
3
7
2
5
i =1
3
2
7
5
Der_ech=2
2
3
7
5
I =2
Der_ech=4
2
3
5
7
i =4
2
3
5
7

Les algorithmes de tris itératifs: Tri à bulles

```
Procédure Tri_bulle (type tableau tab [] :d'entiers ; n : entier) ;
Var : i, j, aide entier
Début
    Pour i <= 1 à n Faire
        Pour j <= i+1 à n Faire
            Si (tab[j] < tab [j-1]) Alors
                aide <= tab[j]
                tab [j] <= tab [j-1]
                tab [j-1] <= aide
            Fin_si
        Fin_pour
    Fin_pour
Fin
```

- On remarque que les transpositions successives font pousser le maximum à la dernière position du tableau si on fait un balayage de gauche à droite et le minimum à la 1ère position si on fait un balayage de droite à gauche.

Complexité

- L'exécution d'un algorithme sur un ordinateur consomme des ressources:
  - en temps de calcul : complexité temporelle
  - en espace-mémoire occupé : complexité en espace

- Seule la complexité temporelle sera considérée pour évaluer l'efficacité et la performance de nos programmes.

- Comment choisir entre différents algorithmes pour résoudre un même problème?

Plusieurs critères de choix :

- Exactitude
- Simplicité
- Efficacité (but de ce chapitre)

Complexité

L'analyse de la complexité consiste à mesurer ces deux grandeurs pour choisir l'algorithme le mieux adapté pour résoudre un problème.(le plus rapide, le moins gourment en place mémoire)

On ne s'intéresse, ici, qu'à la **complexité temporelle** c.à d. qu'au temps de calcul (par opposition à la complexité spatiale)

- Le temps d'exécution dépend de plusieurs facteurs :
  - Les données (trier 4 nombres ne peut être comparé au trie de 1000 nombres).
  - Le code généré par le compilateur (interpréteur).
  - La nature de la machine utilisée (mémoire, cache,multi-treading,…)
  - La complexité de l'algorithme.

- La complexité d'un algorithme permet de qualifier sa performance par rapport aux autres algorithmes.

Complexité

- Si $T(n)$ dénote le temps d'exécution d'un programme sur un ensemble de données de taille n alors :

- $T(n)=c.n^2$ (c est une constante) signifie que l'on estime à $c.n^2$ le nombre d'unités de temps nécessaires à un ordinateur pour exécuter le programme.

exemple: jeu d'échec par recherche exhaustive de tous les coups possibles

10^{19} possibilités, 1 msec/poss. = 300 millions d'années

Complexité

- La complexité dépend de la taille des données de l'algorithme.
- Exemples :
  - Recherche d'une valeur dans un tableau
    → taille (= nombre d'éléments du tableau)
  - Produit de deux matrices
    → dimension des matrices
  - Recherche d'un mot dans un texte
    → longueur du mot et celle du texte

On note généralement:

n la taille de données, T(n) le temps (ou le cout) de l'algorithme.

Complexité

- Dans certains cas, la complexité ne dépend pas seulement de la taille de la donnée du problème mais aussi de la donnée elle-même.

  Toutes les données de même taille ne génèrent pas nécessairement le même temps d'exécution.

  → (Ex. la recherche d'une valeur dans un tableau dépend de la position de cette valeur dans le tableau)

Complexité: Exemple

- Écrire une fonction qui permet de retourner le plus grand diviseur d’un entier.

```
Fonction PGD1( n: entier) : entier
Variables i :entier
Debut
i ← n-1 ;
Tantque (n%i  !=0)
 i ← i-1;
finTantque
Retourner( i)


Fin
```

```
Fonction PGD2( n: entier) : entier
Variables i :entier
Debut
i ← 2;
Tantque ((i<sqrt(n))&&(n%i !=0))
        i ← i+1;
finTantque
si(n%i == 0) alors retourner (n/i)
sinon retourner (1)
finsi
Fin
```

Pour un ordinateur qui effectue 10^6 tests par seconde et $n=10^{10}$ alors le temps requis par PGD1 est d’ordre 3 heures alors que celui requis par PGD2 est d’ordre 0.1 seconde

Complexité

- Une donnée particulière d'un algorithme est appelée instance du problème.

- On distingue trois mesures de complexité:

1. **<u>Complexité dans le meilleur cas:</u>** c'est la **situation la plus favorable**, qui correspond par exemple à la recherche d'un élément situé à la première position d'un tableau, ou encore au tri d'un tableau déjà trié.

$$T_{Min}(n) = \min\{T(d)\ ;\ d \text{ une donnée de taille } n\}$$

Complexité

2. **<u>Complexité dans le pire cas</u>:** c'est la **situation la plus défavorable**, qui correspond par exemple à la recherche d'un élément dans un tableau alors qu'il n'y figure pas, ou encore au tri par ordre croissant d'un tableau trié par ordre décroissant.

$$T_{Mx}(n) = \max\{T(d)\ ;\ d \text{ une donnée de taille } n\}$$

3. **<u>dans le cas moyen:</u>** on suppose là que les données sont réparties selon une certaine loi de probabilités.

$$T_{MOY}(n) = \sum_{d \text{ de taille } n} p(d).T(d)$$

$p(d)$: probabilité d'avoir la donnée d

$$T_{MIN}(n) \leq T_{MOY}(n) \leq T_{MAX}(n)$$

Complexité: notation O

- La complexité est souvent définie en se basant sur le pire des cas ou sur la complexité moyenne. Cependant, cette dernière est plus délicate à calculer que celle dans le pire des cas.

- De façon général, on dit que $T(n)$ est $O(f(n))$ si $\exists$ c et n0 telles que $\forall$ $n \geq n0$, $T(n) \leq c.f(n)$.

- L'algorithme ayant $T(n)$ comme temps d'exécution a une complexité d'ordre $O(f(n))$

- La complexité croit en fonction de la taille du problème
  - L'ordre utilisé est l'ordre de grandeur asymptotique.
  - Les complexités n et 2n+5 sont du même ordre de grandeur.
  - n et n^2 sont d'ordres différents.

Complexité: régles

- 1- Dans un polynôme, seul le terme de plus haut degré compte.
  - Exemple : $n^3+1006n^2+555n$ est $O(n^3)$
- 2- Une exponentielle l'emporte sur une puissance, et cette dernière sur un log.

  Exemple: 2^n+n^{100} est $O(2^n)$ et $300 \log(n)+2n$ est $O(n)$

- 3- Si $T1(n)$ est $O(f(n))$ et $T2(n)$ est $O(g(n))$ alors $T1(n)+T2(n)$ est

  $O(Max(f(n),g(n)))$ et $T1(n).T2(n)$ est $O(f(n).g(n))$

- Les ordres de grandeur les plus utilisées :
  - $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^k)$, $O(2^n)$

Complexité

- **Ordre de grandeur courant**
  - $O(1)$: complexité constante
  - $O(\log(n))$: complexité logarithmique
  - $O(n)$: complexité linéaire
  - $O(n^2)$: complexité quadratique
  - $O(n^3)$: complexité cubique
  - $O(2^n)$: complexité exponentielle

Calcul de la complexité: règles pratiques

Evaluation de $\boldsymbol{T}(\boldsymbol{n})$ (séquence)

La complexité d'une suite d'instructions est la somme des complexités de chacune d'elles.

Somme des coûts.

Traitement1 $T_1(n)$

$$\longrightarrow \quad T(n) = T_1(n) + T_2(n)$$

Traitement2 $T_2(n)$

Les opérations élémentaires telle que l'affectation, test, accès à un tableau, opérations logiques et arithmétiques, lecture ou écriture d'une variable simple ... etc, sont en O(1).

Calcul de la complexité: règles pratiques

Evaluation de $\boldsymbol{T}\,(\boldsymbol{n})$ (branchement)

Max des coûts.

Si < condition $T_1(n)$ > alors

Traitement1 $T_2(n)$

sinon $\longrightarrow$ $max(T_1(n),\ max(T_2(n),\ T_3(n)))$

Traitement2 $T_3(n)$

Calcul de la complexité: règles pratiques

Evaluation de $\boldsymbol{T}\,(\boldsymbol{n})$ (boucle Tant que)

La difficulté, pour la boucle tantque, est de déterminer le nombre d'itération Nb_iter (ou donner une borne supérieure de ce nombre)

T(tantque C faire A ftantque) = O(Nb_iter x (T(C) + T(A))

Somme des coûts des passages successifs

tant que < condition > faire

Traitement $T_i(n)$ $\longrightarrow$ $\sum_{i=1}^{k} T_i(n)$

fin tq

Calcul de la complexité: règles pratiques

Evaluation de $\boldsymbol{T}\,(\boldsymbol{n})$ (boucle Pour)

Somme des coûts des passages successifs

Pour i := e_1 à e_2 faire

Traitement $A_i(n)$ $\longrightarrow$ $\displaystyle\sum_{i=e1}^{e2} T(Ai)$

fin pour

- si Ai ne contient pas de boucle dépendante de i et si Ai est de complexité O(m) alors la complexité de cette boucle « pour » est $O((e_2 - e_1 + 1)\,m)$

Méthode itérative [rappel sommations]

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

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Complexité

- Exemples

1. **Calcul de la somme 1+2+…+n**

S:=0; //O(1)

Pour i:=1 à n faire

s:= s+ i; // O(1)

$\sum_{1}^{n} O(1)$

O(n)

O(1) + O(n) = O(n)

fpour;

$\boxed{\mathbf{T(n) = O(n)}}$