CHAPITRE 3: OPTIMISATION DE LA COMPLEXITÉ ET LA STRATÉGIE DIVISER POUR RÉGNER (D&C)

COURS : COMPLEXITÉ ET OPTIMISATION

RÉALISÉ Par : DR. S. SLATNIA

UNIVERSITÉ DE BISKRA
MASTER D'INFORMATIQUE
2021-2022

LA STRATÉGIE DIVISER POUR RÉGNER (DIVIDE&CONQUER)

MÉTHODE DIVISER POUR RÉGNER

Définition.

- Diviser pour régner est une méthode de programmation récursive qui consiste à diviser un problème en sous-problèmes, résoudre les sous-problèmes et recombiner les résultats.
- Le calcul de la complexité génère des équations de récurrence de partitions qu'on peut résoudre facilement.

❑ Les étapes de la stratégie Diviser pour Régner

La stratégie d'un algorithme « Diviser pour Régner » repose sur trois étapes à chaque niveau de la récursivité :

- **Étape Diviser** : on divise le problème initial en un certain nombre de sous problèmes.

- **Étape Régner** : on règne sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un sous-problème est suffisamment réduite, on peut toutefois le résoudre directement, cas de la base de la récurrence.

- **Étape Combiner** : on forme la solution du problème initial en combinant les solutions partielles (les solutions des sous-problèmes).

❑ Schéma général d'un algorithme de type Diviser pour Régner

```
Fonction DiviserRégner(P : Problème) : Solution
Début
  Si (la taille de P est petit) Alors
      CasdeBase(P);
  Sinon
      Diviser P en N sous-problèmes P1, P2, …, PN.
      Pour (i← 1 à N ) Faire
          Résoudre récursivement Pi : Si ← DiviserRégner(Pi);
      finPour;
      S ← Combiner(S1, S2, …, SN);       /*  S : Solution  */
  finSi;
Fin.
```

- **CasdeBase :** procédure effectuant le traitement de la base de la récursivité.
- **Si ← DiviserRégner(Pi)** : appel récursif pour un sous-problème.
- **Combiner** : procédure effectuant la combinaison des solutions partielles en la solution globale.

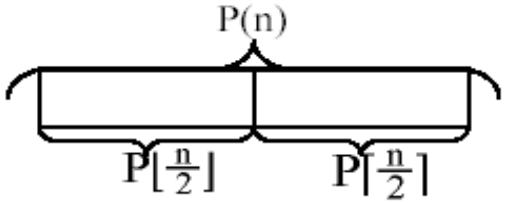
DIVISER POUR RÉGNER : TRI FUSION

- Le Tri Fusion utilise une stratégie différente :
  1. On divise le tableau à trier en **deux** parties (de tailles égales), que l'on trie,
  2. On interclasse les deux tableaux triés ainsi obtenus.

- La stratégie **sous-jacente** est du type **Diviser pour Régner** : on divise un problème sur une donnée de « **grande taille** » en sous-problèmes de même nature sur des données de plus « **petite taille** ».
  - On applique récursivement cette **division** jusqu'à arriver à un problème de très petite taille (**objets élémentaires**) et facile à traiter;
  - On **recombine** les solutions des sous-problèmes pour obtenir la solution au problème initial.

- **Input** : A list of n numbers $\sigma_1, \sigma_2, ..., \sigma_n$
- **Output** : A permutation $\sigma_1, \sigma_2, ..., \sigma_n$ of the input such that $\sigma_1 \leq \sigma_2 \leq ... \leq \sigma_n$

- Le but est de voir la stratégie "Diviser pour régner" par l'exemple du tri-fusion.
- On discutera de l'implémentation des données, forme de listes ou de tableau.
- La stratégie est la suivante :
  1. **Divide**. Eclater la donnée.
  2. **Conquer**. Contracter les données.
  3. **Combine**. Spécifique au problème…



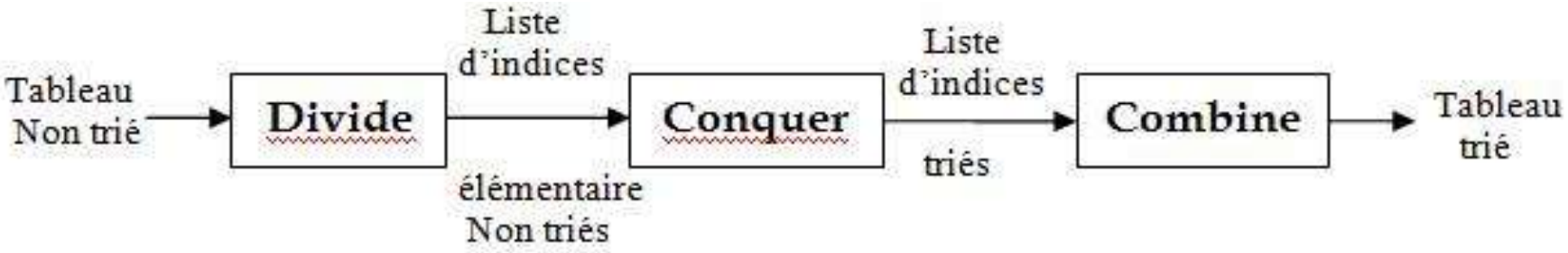


Figure. La stratégie Diviser pour Régner (Divide&Conquer)

Les clauses de la stratégie D&C : tri fusion

1. i=j → Tri(L,i,j)=i.Ø
2. i<j →Tri(L,i,j)=C(L,Tri(T,I,(i+j)/2),Tri(T,(i+j)/2+1,j))

Divide

3. L[i]≤ L[j] →C(L,i.I,j.J)=i.C(L,I,j.J)
4. L[i]> L[j] →C(L,i.I,j.J)=j.C(L,i.I,J)
5. C(L,Ø .J)=J
6. C(L,I. Ø)=I

Conquer

Indices triés

7. TF(L)=TF1(L,Tri(L,1,n),C.i)
8. TF1(L,l.c,C.i)=TF1(L,l,C[i]← L[c],i+1)
9. TF1(L, Ø,C.i)=C

Reconstitution du Tableau C à partir de la liste d'indice l.c.

- Pour reconstituer un tableau, on récupère la liste d'indice triés **l.c**, le tableau d'origine **L** est un tableau résultat **C**.

- Tant qu'il y a des indices dans la liste, alors le tableau résultat contient l'élément de cet indice dans le tableau d'origine.

❑ La stratégie D&C pour Trier un tableau d'éléments

❖ La fonction Divide

- **i = j → Fπ(T, i, j) = i.Ø** (selon le problème)
- **i < j → Fπ(T, i, j) = C(T, Fπ(T, i, (i+j)/2), Fπ(T, (i+j)/2 +1, j))**

❖ La fonction Conquer

- Contracte les données.
- Elle est déclinée sur 3 types d'entrées :
  - ✓ **C(Ø, l2) → l2,**
  - ✓ **C(l1, Ø) → l1,**
  - ✓ **C(c1.l1, c2.l2)** qui consistera à des comparaisons

❖ La fonction Combine

Reconstituer la solution de problème on utilisant la solution trouvée de la fonction Conquer.

❖ Problème :

- Si au lieu d'un tableau, on a une liste ?
- Comment la couper en deux ? On ne veut pas calculer la taille.

❖ Solution :

- Une façon plus simple est de prendre les pairs d'un côté et les impairs de l'autre. Ainsi :

$$F(\emptyset) = \emptyset$$
$$F(C, \emptyset) = C$$
$$|L| \geq 2 \rightarrow F(L) = \text{Comb}(\underbrace{F(L_{impaire}), F(L_{paire})}_{\text{Divide}\quad\text{Divide}})$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{Conquer}}$$

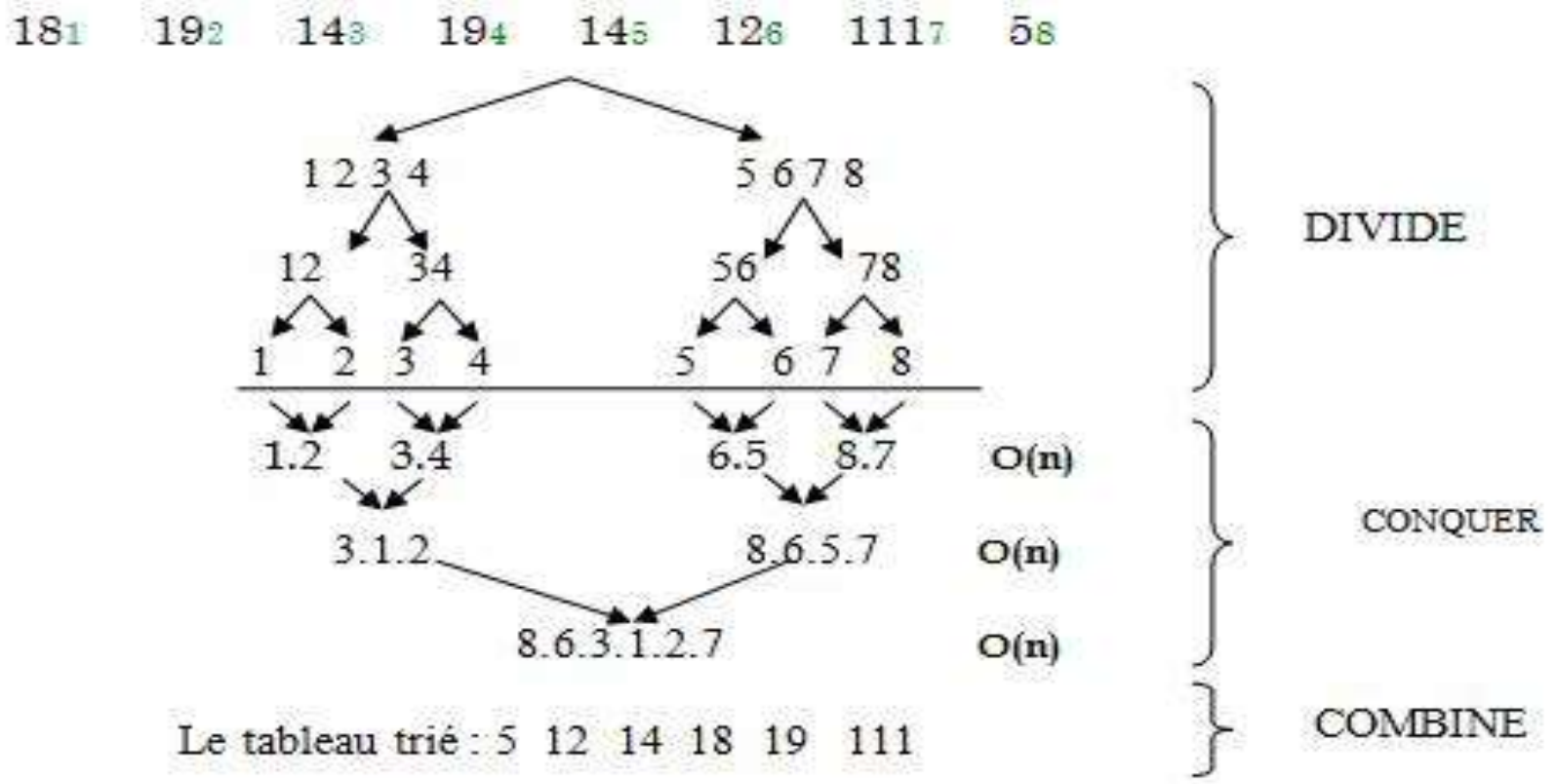
Comb : Combine, la fonction de rassemblement

NB.

→ Si on a un tableau en entrée, on retourne une liste d'indices.

→ Si on a une liste en entrée, on retourne une liste d'éléments.

La stratégie D&C pour Trier un tableau d'éléments



- La **relation de récurrence** est $T(1) = O(1)$, $T(n) = 2T(n/2) + O(Tc)$.
  - Si O(Tc) est **linéaire** → la complexité est en **O(n.log(n))** ;
  - Si O(Tc) est **constant** → la complexité est **linéaire**.

DIVISER POUR RÉGNER : PRODUIT DE DEUX MATRICES

- On considère deux matrices carrées (d'entiers) d'ordre n : M et N.
  - Le produit de M par N est une matrice carrée C définie par :

$$C_{i,j} = \sum_{k=1}^{n} M_{i,k} \times N_{k,j}$$

- On suppose que n est une puissance de 2.
  - On découpe les matrices M et N en 4 blocs (n/2)x(n/2) comme suit :

$$M=\begin{pmatrix} A & B \\ C & D \end{pmatrix},\ N=\begin{pmatrix} E & F \\ G & H \end{pmatrix} \quad \rightarrow \text{ Le produit est : } MN=\begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

  - Soient maintenant :

P1 = A(F - H); P2 = (A + B)H; P3 = (C + D)E;

P4 = D(G - E); P5 = (A + D)(E +H); P6 = (B - D)(G + H); P7 = (A - C)(E + F)

❑ Algorithme produit matriciel de dimension (n,n)×(n,n)

```
Algorithme ProduitMatrices(M,N : tableau de deux dimension [1..n][1..n])
  Var
  i, j,k : entier ;
  Début
        Pour i ← 1 à n pas +1 faire
       Pour  j ← 1 à n pas +1 faire
            C[i, j] ← 0;
            Pour k ← 1 à n faire
                  C[i, j]←C[i, j]+A[i,k]×B[k, j];
               finpour;
           finpour;
        finpour ;
    Fin.
```

Complexité

- Si l'on s'intéresse au nombre **a(n)** d'additions (ou de multiplications) effectuées par l'algorithme,
- On obtient de façon immédiate que **$a(n)= n^3$**.

  → L'algorithme est en **$\Theta(n^3)$,** Il n'ya pas **ni meilleur ni pire des cas**.

Algorithme produit matriciel de dimension (m,n)×(n,p)

Algorithme ProduitMatrices(M: tableau de deux dimension [1..m][1..n] ; N: tableau de deux dimension [1..n][1..p])

```
  Var
  i, j,k :entier;
  Début
      Pour i ← 1 à m pas +1 faire
         Pour j ← 1 à p pas +1 faire
          C[i, j] ← 0;
          Pour k ← 1 à n pas +1 faire
                C[i, j]←C[i, j]+A[i,k]×B[k, j];
            finpour;
         finpour;
     finpour;
  Fin.
```

Complexité

- Le nombre d'additions (ou de multiplications) effectuées est **a(n)= npm.**
- En déduit que l'algorithme est en **Θ(npm).**

❑ La relation des expressions

Avec les Pi et en utilisant uniquement l'addition et la soustraction.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} X \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix} = \begin{pmatrix} P6+P5-P2+P4 & P2+P1 \\ P4+P3 & P5-P7+P1-P3 \end{pmatrix}$$

❑ Algorithme de Strassen de produit de deux matrices

- Supposons que la taille des matrices soit une puissance de 2: $N=2^n$,
- On peut toujours diviser récursivement les matrices en 4 :

$$A=\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \qquad \text{avec Aij de taille } N/2=2^{n-1}$$

- On peut écrire un programme de multiplication de 2 matrices qui utilisera les règles précédentes :

C=MulMat(A,B,N), C=AddMat(A,B,N) et C=SubMat(A,B,N)

Algorithme de Strassen de produit de deux matrices

```
Fonction ProdMat(A, B, N)
Début
   Si (N=1) alors    ProdMat ← A*B;
   Sinon
        découper_en_4(A) ;     découper_en_4(B) ;
   finsi ;
   P1 ← MulMat(A11, SubMat(B12,B22,N/2),N/2);
   P2 ← MulMat(AddMat(A11,A12,N/2),B22,N/2);
   P3 ← MulMat(AddMat(A21,A22,N/2),B11,N/2);
   P4 ← MulMat(A22,SubMat(B21,B11,N/2),N/2);
   P5 ← MulMat(AddMat(A11,A22,N/2),AddMat(B11,B22,N/2),N/2);
   P6 ← MulMat(SubMat(A12,A22,N/2),AddMat(B21,B22,N/2),N/2);
   P7 ← MulMat(SubMat(A11,A21,N/2), AddMat(B11,B12,N/2),N/2);
   C11 ← SubMat(AddMat(P6,P5,N/2),AddMat(P2,P4,N/2),N/2);
   C12 ← AddMat(P2,P1,N/2);
   C21 ← AddMat(P4,P3,N/2);
   C22 ← AddMat(SubMat(P5,P7,N/2), SubMat(P1,P3,N/2),N/2);
   Recompose(C);
End.
```

Complexité de l'Algorithme de Strassen

- ProdMat(…, N) fait donc 7 appels à ProdMat(…, N/2) et 18 appels à AddMat(…, N/2).
- Il faut donc 7 multiplications et 18 additions au lieu de 8 multiplication et 4 addition.

- Si **n** est impair, on augmente les matrices d'une ligne et d'une colonne nulle.

➢ L'équation de récurrence est de type : $T(n) = c.T(n/d) + O(n^k)$, alors:

$c=7$ et $d^k = 2^2 = 4 \rightarrow c > d^k$,

- Le temps de calcul vérifie donc l'équation: **$T(n)=7T([n/2])+O(n^2)$**

- **$O(n^2)$** représente le temps des additions/soustractions et des recopies de coefficients.

$T(n) = O(n^{\log \text{ base d de c}}) = O(n^{\log \text{ base 2 de 7}}) = O(n^{\log_2^7}) = O(n^{2,80})$

❖ **<u>Avantage</u>**

C'est **intéressant** parce que le temps d'une multiplication est beaucoup plus grand que celui d'une addition.

OPTIMISATION DE LA COMPLEXITÉ ET LA DICHOTOMIE

L'élément majoritaire et la dichotomie

❑ Qu'est-ce que la majorité ?

Données : S = {{x_1, …, x_n}}, x_i appartenant à un univers U pas nécessairement ordonné.

Sortie : L'élément majoritaire de S sinon **aucune** majorité ne se détache.

- Soit MAJ(x, S) le nombre d'occurrences de x dans S. **Si MAJ(x, S) > |s|/2**, alors c'est l'élément majoritaire.

❖ Solution novice

Le principe est de compter le nombre d'occurrence de chaque élément : si cela totalise plus de la moitié des éléments du tableau, cet élément est majoritaire alors on s'arrête. Sinon, on compte le nombre d'occurrence de l'élément suivant.

- On s'arrête une fois que l'on a parcouru plus de la moitié du tableau.
- Complexité de l'algorithme en $\mathbf{n^2}$, **solution utilisée est pire**.

```
i ←1;
Algorithme MAJ(T : tableau[1..N]d'éléments ; i, n : entier)
    Début
        Si (i ≤ n/2) alors        Compter(T, i, i+1, n, 0) ;
        Sinon
            écrire (''aucun n'était majoritaire'') ;
        finsi ;
    Fin.
Algorithme Compter(T : tableau [1..n] d'éléments ; i, j, n, r :entier)
    Début
        Si (j ≤ n et T[i] = T[j]) alors   Compter(T, i, j+1, n, r+1) ;      finsi ;
        Si (j ≤ n et T[i] ≠ T[j]) alors   Compter(T, i, j+1, n, r) ;      finsi ;
        Si (j > n et r > n/2) alors       écrire (''l'élément majoritaire est :'',T[i]) ;  finsi ;
        Si (j > n et r < n/2) alors       MAJ(T, i+1, n) ;     finsi ;
    Fin.
```

❖ Solution hacker

- On trie le tableau et on compte le nombre d'occurrences de l'élément du milieu.
- L'élément majoritaire doit occuper le milieu du tableau et apparaît au moins **(n/2)** fois,
- Sinon il n'y a pas d'élément majoritaire.
  - ➢ Complexité de l'algorithme en **O(nlogn).**

✓ L'idée la plus simple est de partir du centre et de s'arrêter lorsque l'élément change :

- ➢ Partir du début, arrêter de compter quand on tombe sur notre élément
- ➢ Partir de la fin, arrêter de compter quand on tombe sur notre élément
- ➢ Faire la différence des indices ainsi obtenus.

- On pose un pointeur sur la fin et un sur le début.
- Dès que l'un des deux rencontre notre élément, on le bloque.
- Les deux ont rencontrés l'élément, on calcule la différence et on regarde
  - si elle est suffisante pour nous garantir que l'élément est majoritaire.

MAJ(T,i,j,x) = MAJ(T,1,n,T[(1+n)/2]) x est l'élément du centre, celui dont on teste la quantité

$T[i] \neq x \wedge T[j] \neq x \rightarrow MAJ(T, i, j, x) = MAJ(T, i+1, j-1, x)$
$T[i] \neq x \wedge T[j] = x \rightarrow MAJ(T, i, j, x) = MAJ(T, i+1, j, x)$
$T[i] = x \wedge T[j] \neq x \rightarrow MAJ(T, i, j, x) = MAJ(T, i, j-1, x)$
$T[i] = x \wedge T[j] = x \rightarrow MAj(T, i, j, x) = (j-i+1 > n/2)$

➢ Bonne idée :

❖ Trier le tableau et penser que l'élément du centre est le seul à pouvoir être majoritaire

❖ Utiliser une recherche dichotomique, on recherche donc la borne supérieure et la borne inférieure par dichotomie.

Exercice – dichotomie

Le jeu « devinez le nombre » de 1 à 100 est le suivant :

- La personne P1 choisit secrètement un entier X de 1 à 100
- La personne P2 propose à la personne P1 un entier Y de 1 à 100, qui lui répond :
  C'est exact si X = Y, Plus bas si X < Y et Plus haut si X > Y
- La personne P2 doit répéter jusqu'à ce que le nombre soient deviné.

1. Ecrivez une procédure récursive deviner (vmin, vmax, mystère) du jeu : elle demande à l'utilisateur un entier (dans nombre) compris dans [vmin..vmax], l'entier mystère étant le nombre à deviner et
2. Ecrivez un algorithme qui lance le jeu entre 1 et 100 en tirant au hasard l'entier mystère.

Correction – *dichotomie*

La stratégie la plus efficace est d'utiliser une recherche dichotomique. Exemple d'exécution.

Devinez l'entier entre 1 et 100? 50	Plus haut
Devinez l'entier entre 51 et 100? 75	Plus bas
Devinez l'entier entre 51 et 74? 65	Plus bas
Devinez l'entier entre 51 et 64? 57	Plus bas
Devinez l'entier entre 51 et 56? 53	Plus haut
Devinez l'entier entre 54 et 56? 55	C'est exact

Algorithme pg_rcdeviner

```
Début
| deviner ( 1 , 100 ,Aléatoire ( 100 ) + 1 )
Fin

Action deviner ( vmin , vmax , mystère : Entier )
Variable nombre : Entier
Début
|      Afficher ( "Devinez l'entier entre " , vmin , " et " , vmax , "? " )
|      Saisir ( nombre )
|      Si ( mystere = nombre ) Alors
|       |       Afficher ( "C'est exact" )
|       Sinon
|       |       Si ( mystere < nombre ) Alors
|       |         |     Afficher ( "Plus bas" )
|       |         |     deviner ( vmin , nombre - 1 , mystere )
|       |        Sinon
|       |         |     Afficher ( "Plus haut" )
|       |         |     deviner ( nombre + 1 , vmax , mystere )
|       |         FinSi
|       FinSi
Fin
```

OPTIMISATION ET ALGORITHMES AVANCÉS, COMPARAISON DE COMPLEXITÉ

I. PRÉSENTATION DES DIFFÉRENTES MÉTHODES DE TRI

- Les tableaux permettent de stocker plusieurs éléments de même type.
- Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger dans un ordre croissant ou décroissant.
- Tous les algorithmes de tri utilisent une procédure qui permet d'échanger la valeurs de deux variables.

```
Procédure échanger (a,b : entier)
    Var  elem : entier ;
    Début
       elem ← a;
       a ← b;
       b ← elem;
    Fin.
```

- Il existe plusieurs méthodes de tri qui se différencient par leur :
  - **complexité d'exécution**
  - **complexité de compréhension.**

- **Problème** : tri (sort en anglais)
  - Etant donnée une suite de n nombres, de les ranger par ordre croissant.
  - La suite de n nombres est représentée par un tableau de n nombres et on suppose que ce tableau est entièrement en machine.

→ Un petit tableau, (7, 1, 15, 8, 2) ⇒ facile de l'ordonner pour obtenir (1, 2, 7, 8, 15).
→ Un tableau de plusieurs centaines, voire millions d'éléments ⇒ moins évident.

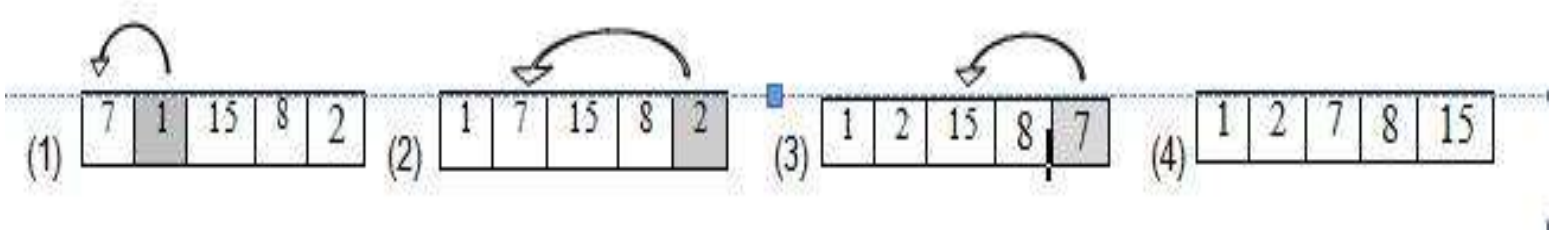
❑ Différents types de tri

- **Tri interne** : tri en mémoire centrale. **Tris externes** : données sur un disque externe.
- **Tri de tableau** : tri qui trie un tableau. Extensible à toutes structures de données offrant un accès en temps (quasi) constant à ses éléments.
- **Tri générique** : peut trier n'import quel type d'objets pour autant qu'on puisse comparer ces objets.
- **Tri comparatif** : basé sur la comparaison entre les éléments (clés)

- **Tri itératif** : basé sur un ou plusieurs parcours itératif du tableau
- **Tri récursif :** basé sur une procédure récursive
- **Tri en place :** modifie directement la structure qu'il est en train de trier. Ne nécessite qu'une quantité très limitée de mémoire supplémentaire.
- **Tri stable :** conserve l'ordre relatif des éléments égaux (au sens de la méthode de comparaison) [2] .

ALGORITHME DE TRI NAÏFS (TRI PAR SÉLECTION)

- On recherche le plus petit élément parmi les n éléments et on l'échange avec le premier élément ;
- puis on recherche le plus petit élément parmi les $n-1$ derniers éléments de ce nouveau tableau et on l'´echange avec le deuxième élément;
- plus généralement, à la k-ième étape, on recherche le plus petit élément parmi les $n-k+1$ derniers éléments du tableau en cours et on l'échange avec le k-ième élément.



Algorithme de Tri naïfs (tri par sélection)

Données : Un tableau T de N éléments comparables
Résultat : Le tableau T contient les mêmes éléments mais rangés par ordre croissant

```
Algorithme  TRI-SELECTION(T : tableau [1..N] d'entiers, entier N)
Var
i, j, min : entier;
Début
    pour i ← 1 à N - 1 faire
            min ← i ;
            pour j ← i + 1 à N faire
                    si (T[j] < T[min]) alors
                        min ←  j ;
                    finsi ;
            finpour ;
            échange (T[i]; T[min]) ;
    finpour ;
Fin.
```

❑ Complexité de l'algorithme

❖ **Complexité en temps :** le temps d'exécution t(n) du programme dépendra en général de la taille n des données d'entrée ; on distinguera :

- ✓ **la complexité moyenne** : difficile à évaluer,
  - ➢ Car il faut commencer par décider quelle donnée est « moyenne » ce qui demande un recours aux probabilités.

- ✓ **la complexité pour le pire des cas** (t(n) maximal) : pour la donnée d'entrée donnant le calcul le plus **long**,

- ✓ **la complexité pour le meilleur des cas** (t(n) minimal) : pour la donnée d'entrée correspondant au calcul le plus **court**,

❖ **Complexité en espace :** L'algorithme TRI-SELECTION trie les éléments du tableau dans le tableau lui-même et n'a besoin d'aucun espace supplémentaire.

- ✓ Il est donc très **économe en espace** car il fonctionne en espace constant

Algorithme de Tri Fusion (Merge Sort)

Données : Un tableau T d'entiers indicés de l à r, et tel que l ≤ m < r et que les sous-tableaux T[l … m] et T[m + 1 … r] soient ordonnés

Résultat : Le tableau T contenant les mêmes éléments mais rangés par ordre croissant.

```
Algorithme TRI-FUSION (T : tableau [1..N] d'entiers ; N , l, r : entiers)
 Var
  m: entier;
  Dédut
      si (l < r) alors       m ← [(l + r)/2] ;    finsi ;
     TRI-FUSION(T ; l ; m) ;
     TRI-FUSION(T; m+1, r) ;
     fusion(T; l; r; m) ;
  fin.
```

Données : Un tableau T d'entiers indicés de l à r, et tel que l ≤ m < r et que les sous-tableaux T[l … m] et T[m + 1 … r] soient ordonnés
Résultat : Le tableau T contenant les mêmes éléments mais rangés par ordre croissant.

Algorithme fusion (T : tableau [l..r] d'entiers ; l, r, m : entiers)
Var
i, j, k, n1, n2 : entiers; L: tableau [1..n1] d'entiers; R: tableau [1..n2] d'entiers;
Début
n1 ← m - l + 1 ; **n2** ← r – m ;
pour i ← 1 à n1 **faire** L[i] ← T[l + i - 1]; **finpour;**
pour j ← 1 à n2 **faire** R[j] ← T[m + j]; **finpour**;
i ← 1 ;
j ← 1 ;
L[n1 + 1] ← ∞ ; // On marque la fin du tableau gauche
R[n2 + 1] ← ∞ ; // On marque la fin du tableau droit
pour k ← l à r **faire**
si (L[i] ≤ R[j]) et (i ≤ n1) **alors** T[k] ← L[i] ; i ← i + 1;
sinon
si (j ≤ n2) **alors** T[k] ← R[j] ; j ← j + 1 ; **finsi** ;
finpour ;
Fin.

❑ Complexité de l'algorithme

- ❖ **La complexité en temps** : est en (**nlogn**),
- ❖ **La complexité en espace** : est **linéaire** dans tous les cas.

Algorithme de tri rapide (quicksort)

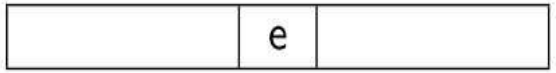
Principe de la méthode

- Est de découper (rapidement) le tableau en deux parties, une partie (T-) où tous les éléments sont inférieurs à une valeur donnée (le pivot) **e** et l'autre partie (T+) où tous les éléments sont supérieurs ou égaux au pivot.
- Ensuite on appelle récursivement Quicksort indépendamment sur chaque partie.
- Il est facile de voir qu'un tel processus garantit que le tableau résultant est trié, du moment que les parties diminuent en taille.

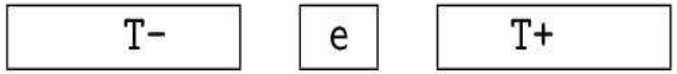
Algorithme de tri rapide (quicksort)

- Cela se fait en définissant trois fonctions :
- **La fonction TrouvePivot** qui recherche un pivot valide,
- **La fonction Partition** qui découpe le tableau, et
- **La fonctio Quicksort** qui appelle les deux autres et s'appelle récursivement.

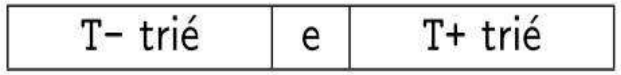
- Choisir arbitrairement un élément e :



- Constituer les sous listes T- et T+ (en temps linéaire) :



- Recommencer récursivement avec les tableaux T- et T+ puis reconstituer T :



- Le tableau T est alors trié.

- Le tableau ci-dessous résume tous les complexités des algorithmes de tri :

Complexité	Espace	Temps		
		Pire	Moyenne	Meilleure
Tri sélection	Constant	n^2	n^2	n^2
Tri fusion	Linéaire	Nlogn	nlogn	nlogn
Tri rapide	Constant	n^2	nlogn	nlogn

Algorithme de tri par tas

Introduction

- Heapsort en anglais
- inventé par Williams en 1964
- basé sur une structure de donnée très utile, le tas
- complexité bornée par $\Theta(n \log n)$ (dans tous les cas)
- Tri en place
- Mise en œuvre très simple [2]

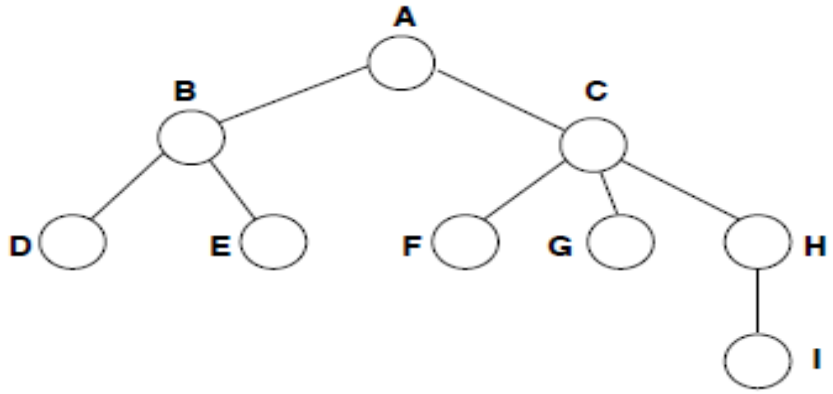
❑ Arbres : Définition

Définition : Un arbre (tree) T est un graphe dirigé (N, E), où :

- N est un ensemble de noeuds, et
- E ⊂ N x N est un ensemble d'arcs,

possédant les propriétés suivantes :

- T est connexe et acyclique
- Si T n'est pas vide, alors il possède un noeud distingué appelé racine. Cette racine est unique.
- Pour tout arc (n1, n2) ∈ E, le noeud n1 est le parent de n2.
  - ❖ La racine de T ne possède pas de parent.
  - ❖ Les autres noeuds de T possèdent un et un seul parent



❑ Arbres: Terminologie

- Si n2 est le parent de n1, alors n1 est le fils de n2.
- Deux noeuds n1 et n2 qui possèdent le même parent sont des frères.
- Un noeud qui possède au moins un fils est un noeud interne.
- Un noeud externe (non interne) est une feuille de l'arbre.
- Un noeud n2 est un ancêtre d'un noeud n1 si n2 est le parent de n1 ou un ancêtre du parent de n1.
- Un noeud n2 est un descendant d'un noeud n1 si n1 est un ancêtre de n2.

- **Un chemin** est une séquence de noeuds n1, n2, . . . , nm telle que pour tout :

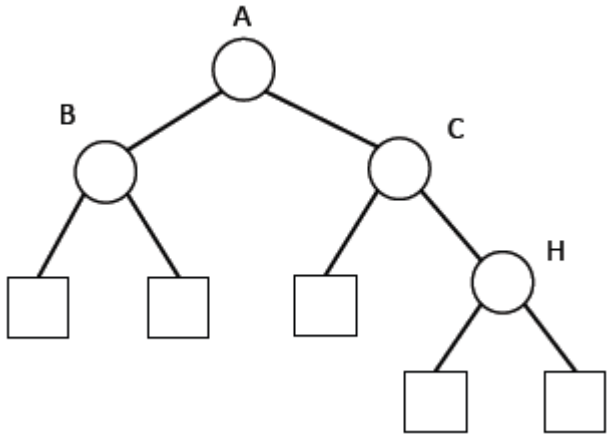
$$i \in [1, m-1], (ni , ni+1)$$ est un arc de l'arbre.

Remarque : Il n'existe jamais de chemin reliant deux feuilles distinctes.

- **La hauteur (height)** d'un noeud n est le nombre d'arcs d'un plus long chemin de ce vers une feuille. La hauteur de l'arbre est la hauteur de sa racine.

- **La profondeur (depth)** d'un noeud n est le nombre d'arcs sur le chemin qui le relie à la racine.

❑ Propriétés des arbres binaires entiers

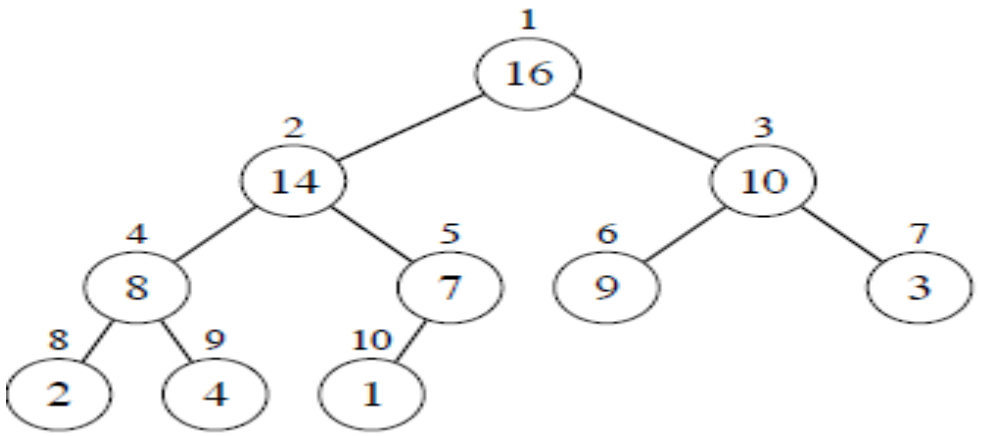


- Le nombre de noeuds externes est égal au nombre de noeuds internes plus 1.
- Le nombre de noeuds interne est égal à $(n-1)/2$, où n désigne le nombre de noeuds.
- Le nombre de noeuds à la profondeur (ou niveau) i est $\leq 2^i$
- La hauteur h de l'arbre est $\leq$ au nombre de noeuds internes.
- Le lien entre hauteur et nombre de noeuds peut etre résumé comme suit :

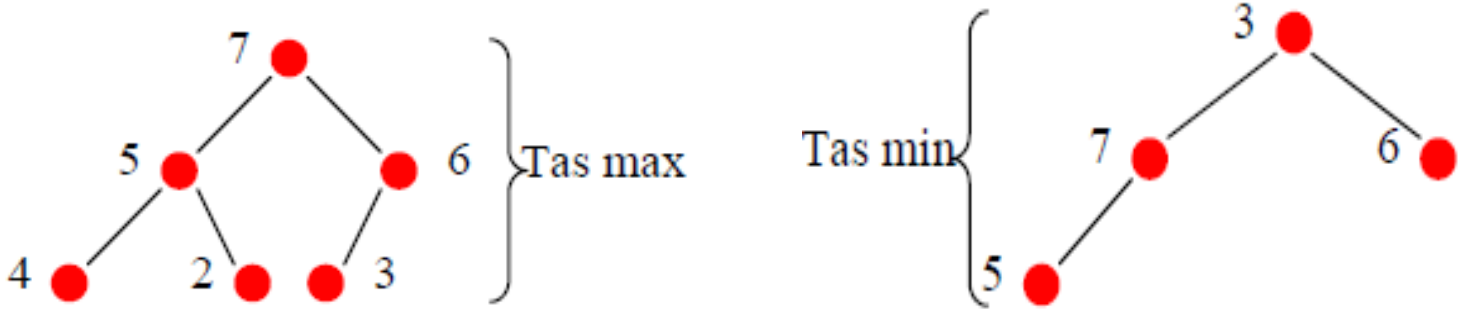
$$n \in \Omega(h) \text{ et } n \in O(2^h) \text{ (ou } h \in O(n) \text{ et } h \in \Omega(\log n))$$

Tas : Définition

- Un arbre binaire **complet** est un arbre binaire tel que :
  - Si h dénote la hauteur de l'arbre :
    - Pour tout i$\in$ [0, h − 1], il y a exactement 2^i noeuds à la profondeur i .
    - Une feuille a une profondeur h ou h − 1.
    - Les feuilles de profondeur maximale (h) sont "tassées" sur la gauche.
- Un **tas binaire** (binary heap) est un arbre binaire complet tel que :
  - Chacun de ses noeuds est associé à une clé.
  - La clé de chaque noeud est supérieure ou égale à celle de ses fils (propriété d'ordre du tas).



- Le **tas (Heap)** nous permet de résoudre le problème des files de priorité. Il est fait de la manière suivante :
  - **Structure** : il y a deux types de tas qui sont « symétriques », le tas min et le tas max. Dans le tas max, tous les noeuds des niveaux inférieurs sont plus petits. Dans le tas min, tous les noeuds des niveaux inférieurs sont plus grands.
  - **Equilibre** : tous les niveaux de l'arbre sont pleins sauf le dernier qui doit être de gauche à droite (s'il existe un fils droit alors il y a nécessairement un fils gauche).



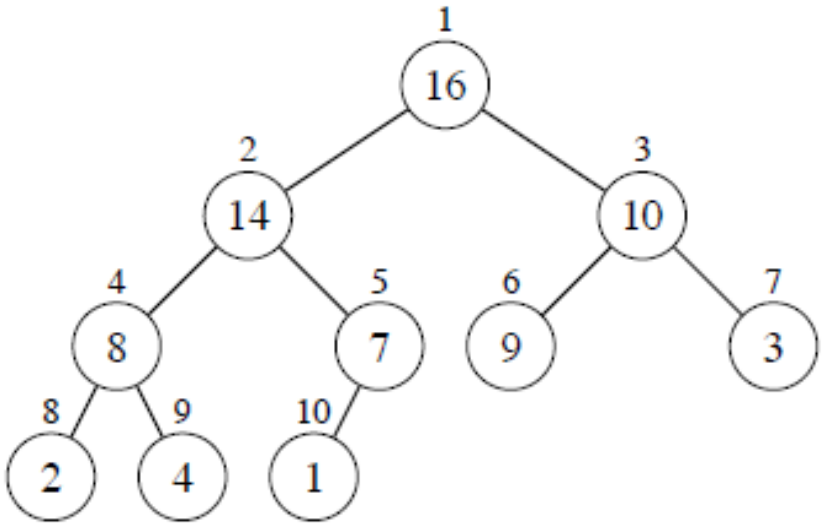
N.B : Si la propriété de remplissage de gauche à droite n'était pas respectée, on verrait des trous dans les tableaux ; il n'y a pas de gaspillage.

❑ Propriété d'un tas

- Soit T un arbre binaire complet contenant n entrées et de hauteur h :
  - ➢ n est supérieur ou égal à la taille de l'arbre complet de hauteur h − 1 plus un, soit $2^{h-1+1} = 2^h$
  - ➢ n est inférieur ou égal à la taille de l'arbre complet de hauteur h, soit

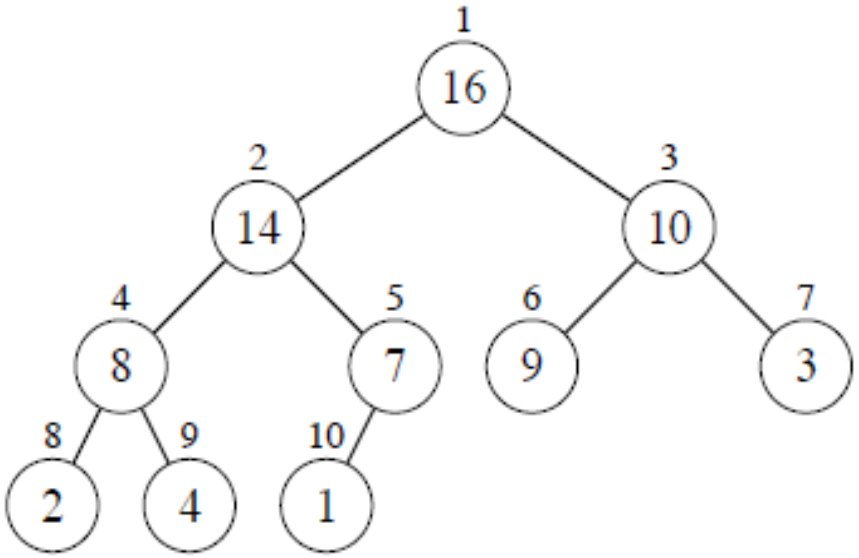
$2^{h+1} - 1$

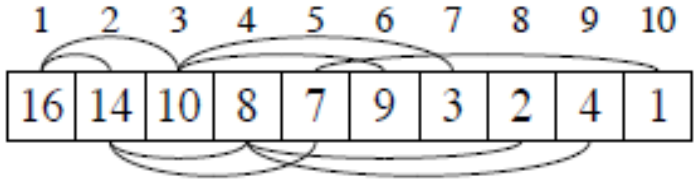
$$2^h \le n \le 2^{h+1} - 1 \Leftrightarrow 2^h \le n < 2^{h+1}$$
$$\Leftrightarrow h \le \log_2 n < h + 1$$
$$\Leftrightarrow h = \lfloor \log_2 n \rfloor$$



NB. Si on a un arbre A tel que |A| = n alors sa hauteur est floor(log n)

❑ Implémentation par un tableau





- Un tas peut être représenté de manière compacte à l'aide d'un tableau A.
  - ❖ La racine de l'arbre est le premier élément du tableau.
  - ❖ Parent(i) = [i/2]
  - ❖ Left(i) = 2i
  - ❖ Right(i) = 2i + 1
- Propriété d'ordre du tas: $\forall i, A[\text{PARENT}(i)] \geq A[i]$

- Le tableau ci-dessous résume tous les complexités des algorithmes de tri :

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
BUBBLE-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
QUICK-SORT	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	oui
HEAP-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)^*$	oui

II. PRÉSENTATION DES DIFFÉRENTES MÉTHODES DE RECHERCHE

❑ Recherche séquentielle : recherche dans un tableau non trié

Parcourir le tableau à partir du premier élément, et à s'arrêter dès que l'on trouve l'élément cherché.

Données : Un tableau T de n éléments et un élément k

Résultat : Le premier indice i où se trouve l'élément k si k est dans T, et sinon la réponse « k n'est pas dans T »

Procédure RechercheNonTrie (T : tableau [1..N] d'éléments ; k : élément)
Var **i**: entire;
Début
i 1;
Tantque ((i <=n) et (T[i] ≠ k)) **faire** i ← i + 1; **fintantque;**
Si (i < n+1) **alors** écrire (T[i] = k);
sinon écrire (''k n'est pas dans T'') ; **finsi ;**
Fin.

Complexité de l'algorithme

- La complexité en temps de RECHERCHE est linéaire (de l'ordre de n),
- il faudra au pire parcourir tout le tableau.

❑ Recherche séquentielle : recherche dans un tableau trié

Fonction RechercheTrie(T : tableau [0..N] d'éléments ; k:élément) : entier
Var
i : entier ;
Début
  i ← 0;
  Tantque (k>T[i]) **faire**
    i ← i+1 ;
  fintanque ;
  retourner(i) **;**
Fin.

Recherche dichotomique

- Cette méthode s'applique si le tableau est déjà trié et s'apparente alors à la technique Diviser pour Régner.

- Elle suppose donc :
  1. Que les éléments du tableau sont comparables
  2. Un prétraitement éventuel du tableau où s'effectue la recherche :
     - par un précalcul, trier le tableau dans lequel on veut chercher.

Données : Un tableau T[1..N] d'entiers déjà ordonné et un entier k
Résultat : Un indice i où se trouve l'élément k, ou bien -1 si k n'est pas dans T

Algorithme RechercheDicho (T : tableau [1..N] d'éléments ; k:élément) :entier
i,l,r : entier;
Début
l ← 1; r ← N; i ← [(l + r)/2];
Tantque ((k ≠ T[i]) et (l ≤ r)) **faire**
si (k < T[i]) **alors** r ← i – 1;
Sinon l ← i + 1; **finsi**;
i ← [(l + r)/2];
si (k = T[i]) **alors** retourner (i) ;
sinon retourner (-1) ; **finsi ;**
fintantque ;
Fin.

❑ Complexité de l'algorithme

- Soit t(n) le nombre d'opérations effectuées dans cet algorithme sur un tableau de taille n.
- $\mathbf{T(n)=t(n/2)+1}$, t(n) satisfait l'équation de récurrence t(n).
- comme $\mathbf{t(1) = 1}$, la complexité en **O(log n).**

❖ On remarquera que la complexité en temps est réduite de linéaire.

➢ **O(n)** à logarithmique **O(log n)** entre la recherche séquentielle et la recherche dichotomique.