UNIVERSITE MOHAMED KHIDER – BISKRA
FACULTE DES SCIENCES EXACTES, DES SCIENCES DE LA NATURE ET DE LA VIE
DEPARTEMENT D'INFORMATIQUE

Complexité et Optimisation

Support de cours et TD

Master d'informatique

Réalisé par : Dr. S. SLATNIA

Version améliorée

05/01/2016

CHAPITRE 3.
OPTIMISATION DE LA COMPLEXITE ET LA STRATEGIE DIVISER POUR REGNER (D&C)

I. LA STRATÉGIE DIVISER POUR RÉGNER (DIVIDE&CONQUER)

1. Méthode Diviser pour Régner

Définition 1.
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. Dont 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 **:**

1. **Étape Diviser** : on divise le problème initial en un certain nombre de sous problèmes.
2. **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.
3. **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.

2. Diviser pour Régner : tri fusion

Le Tri Fusion utilise une stratégie différente : on divise le tableau à trier en deux parties (de tailles égales), que l'on trie, puis 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 du jour est de voir la stratégie "Diviser pour régner" par l'exemple du tri-fusion. On discutera de l'implémentation des données sans 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…

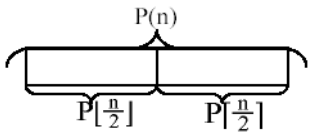


Figure 4. La fonction Divide pour éclater la donnée.

→ 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.

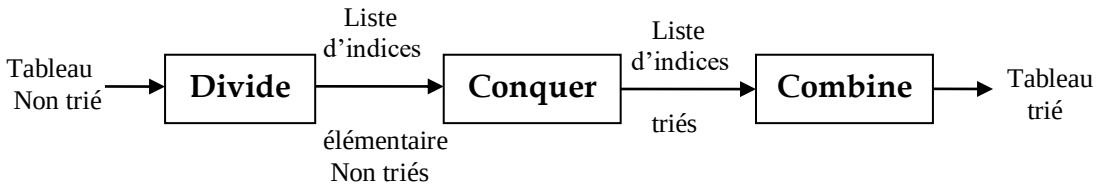


Figure 5. La stratégie Diviser pour Régner (Divide&Conquer)

A. Les clauses de la stratégie Diviser pour Régner : tri fusion

1. i=j → Tri(L,i,j)=i.∅
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,∅ .J)=J
6. C(L,I, ∅)=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, ∅,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.

Et si au lieu d'un tableau, on a une liste ? Comment la couper en deux ? On ne veut surtout pas calculer la taille. Une façon plus simple est de prendre les pairs d'un côté et les impairs de l'autre. Ainsi :

$F(\emptyset) = \emptyset$

$F(C, \emptyset)=C$

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

Comb : Combine, la fonction de rassemblement

B. La stratégie Diviser pour Régner pour Trier un tableau d'éléments

Les règles des trois sous fonction sont les suivantes:

- **La fonction Divide**

  i = j → Fп(T, i, j) = i.Ø (selon le problème)

  i < j → Fп(T, i, j) = C(T, Fп(T, i, (i+j)/2), Fп(T, (i+j)/2 +1, j))

- **La fonction Conquer**

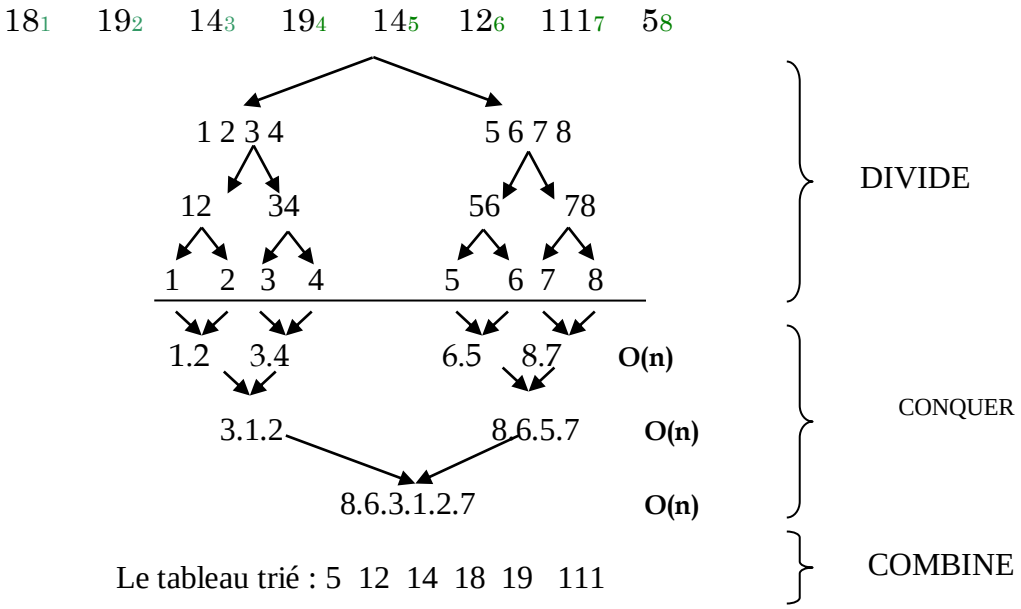
C est la fonction Conquer qui contracte les données. Elle est déclinée sur 3 types d'entrées :

C(Ø, l2) → l2, C(l1, Ø) → l1, et 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.

Exemple : application D&C pour le tri d'un tableau T, T=[18 19 14 19 14 12 111 5]



La relation de récurrence est T(1) = O(1), T(n) = 2T(n/2) + O(Tc).

Si O(Tc) est linéaire alors la complexité est en O(n.log(n)) ; s'il est constant, la complexité est linéaire.

3. 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 :

$$Ci,j=\sum_{k=1}^{n} Mi,k \times Nk,j$$

1. Donner un algorithme calculant le produit de deux matrices représentées sous forme d'un tableau à deux dimensions. Calculer la complexité de cet algorithme. Doit-on préciser dans quels cas (pire cas, meilleur des cas, cas moyen) cette complexité est obtenue?

2. Modifier l'algorithme précédent lorsque la matrice M est de dimension (m,n) et la matrice N de dimension (n,p). Quelle est alors la complexité de l'algorithme?

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=$\begin{pmatrix} E & F \\ G & H \end{pmatrix}$ Et donc 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).

3. Avec les Pi en utilisant uniquement l'addition et la soustraction, Exprimer les expressions suivantes :

AE+BG;
AF+BH;
CE+DG et
CF+DH

4. En déduire un algorithme pour calculer le produit de matrices et évaluer sa complexité.

A. Algorithme produit matriciel de dimension (n,n)×(n,n)

L'algorithme calculant le produit des deux matrices M et N est le suivant:

```
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.

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

L'algorithme modifié est le suivant:

```
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 cette fois-ci a(n)= npm. En déduit que l'algorithme est en $\Theta(npm)$.

C. La relation des expressions

Avec les (Pi=Xi) en utilisant uniquement l'addition et la soustraction.

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

D. Algorithme de Strassen de produit de deux matrices

Supposons que la taille des matrice soit une puissance de 2: $N=2^n$, alors on peut toujours diviser récursivement les matrices en 4 de la manière suivante :

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

On peut alors écrire un programme de multiplication de 2 matrices qui utilisera les règles précédentes. Il utilisera les deux procédures suivantes :
C=MulMat(A,B,N), C=AddMat(A,B,N) et C=SubMat(A,B,N)

```
Fonction ProdMat(A, B, N)
    Début
    Si (N=1) alors    ProdMat ← A*B;
    Sinon
        début
            découper_en_4(A) ;
            découper_en_4(B) ;
         end ;
         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.

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

➢ **<u>Complexité</u>**

- Si **n** est impair, la taille des matrices est augmenté (une ligne et une colonne nulle).
- Le temps de calcul vérifie l'équation de récurrence : **$T(n)=7T([n/2])+O(n^2)$**
  - ❖ $O(n^2)$ représente le temps des additions/soustractions et des recopies de coefficients.
  - ❖ L'équation de récurrence est de type : $T(n) = c.T(n/d) + O(n^k)$, alors:
    $c=7$ et $d^k = 2^2 = 4 \rightarrow c > d^k$,

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

II. OPTIMISATION DE LA COMPLEXITE ET LA DICHOTOMIE

1. L'élément majoritaire et la dichotomie

1. 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 ou bottom si aucune majorité ne se détache.

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

1.1. 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 n^2, solution utilisée est pire.

```
i← 1;    j ← 2;
Algorithme MAJ(T : tableau[1..N]d'éléments ; i, n : entier)
Début
   si (i ≤ n/2) alors
     Compter(T, i, j, 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.
```

1.2. 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 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.

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] ≠ x ^ T[j] ≠ x → MAJ(T, i, j, x) = MAJ(T, i+1, j-1, x)
T[i] ≠ x ^ T[j] = x → MAJ(T, i, j, x) = MAJ(T, i+1, j, x)
T[i] = x ^ T[j] ≠ x → MAJ(T, i, j, x) = MAJ(T, i, j-1, x)
T[i] = x ^ T[j] = x → MAj(T, i, j, x) = (j-i+1 > n/2)

On pose un pointeur sur la fin et un sur le début. Dès que l'un des deux rencontre notre élément, on le bloque. Lorsque 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.

Bonne idée : Trier le tableau et penser que l'élément du centre est le seul à pouvoir être majoritaire et d'utiliser une recherche dichotomique, on recherche donc la borne supérieure et la borne inférieure par dichotomie.

MAJ(T, n) = (BS(T, 1, n, T[(i+1)/2]) – BI(T, 1, n, T[(i+1)/2]) + 1) ≥ n / 2

On regarde si la différence entre la borne supérieure (BS) et la borne inférieure (BI) est suffisante pour garantir qu'il y a majorité (supérieur strictement à n/2). BS et BI sont des recherches par dichotomie.

$BI(T, g, d, x) : g \leq d \wedge T[(g+d)/2] \geq x \rightarrow BI(T, g, d, x) = BI(T, g, (g+d)/2 - 1, x)$

$g \leq d \wedge T[(g+d)/2] < x \rightarrow BI(T, g, d, x) = BI(T, (g+d)/2 + 1, d, x)$

$g > d \rightarrow BI(T, g, d, x) = g$

$BS(T, g, d, x) : g \leq d \wedge T[(g+d)/2] \geq x \rightarrow BI(T, g, d, x) = BI(T, (g+d)/2 + 1, d, x)$

$g \leq d \wedge T[(g+d)/2] < x \rightarrow BI(T, g, d, x) = BI(T, g, (g+d)/2 - 1, x)$

$g > d \rightarrow BI(T, g, d, x) = d$

2. Optimisation et algorithmes avances, comparaison de complexite des algorithmes de tri et de recherche

1. 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 (permuter) les valeurs de deux variables.

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

Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant. Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension**.**

Problème : tri (sort en anglais) il s'agit, étant 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.

Nous choisissons par la suite la présentation des trois méthodes ou algorithmes de tri, chaque méthode permet de présenter un type de classe de complexité trouvée (pire des cas, moyenne et meilleur).

❖ **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).

1.1. 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**.**

Exemple 1.

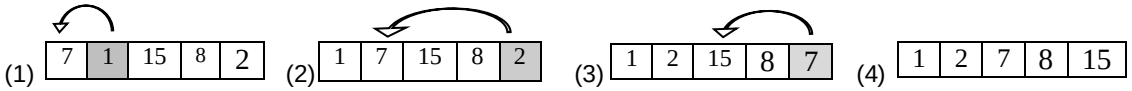


Figure 6. Comment le Tri par Sélection trié le tableau [7, 1, 15, 8, 2]

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, c'est-à-dire la valeur moyenne des t(n) : elle est en général 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, c'est-à-dire pour la donnée d'entrée donnant le calcul le plus long, soit t(n) maximal ;
  - la complexité pour le meilleur des cas, c'est-à-dire pour la donnée d'entrée correspondant au calcul le plus court, soit t(n) minimal.
- **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 : on trie le tableau sur place, sans avoir besoin de stocker une partie du tableau dans un tableau auxiliaire. Par contre il n'est pas économe en temps

1.2. Algorithme de Tri Fusion (Merge Sort)

Données : Un tableau T de N entiers indicés de l à r
Résultat : Le tableau T contient 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 ;
           finsi ;
     finpour ;
   Fin.
```

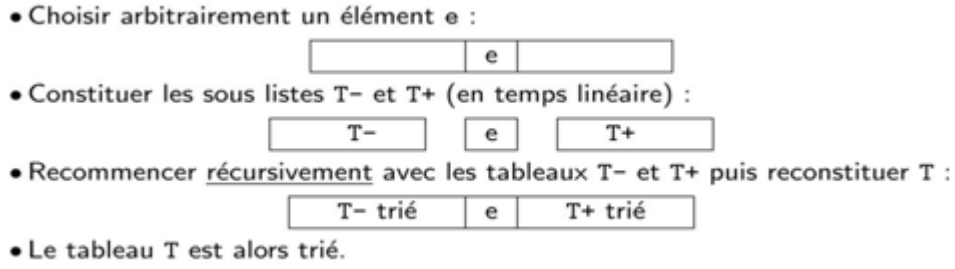
Complexité de l'algorithme

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

1.3. Le tri rapide (quicksort)

Principe de la méthode

Le principe du tri rapide « quicksort » 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.
Cela se fait en définissant trois fonctions : la fonction TrouvePivot qui recherche un pivot valide, la fonction Partition qui découpe le tableau, et l'action Quicksort qui appelle les deux autres et s'appelle récursivement.



Algorithme de tri rapide

1. Algorithme Fonction Trouver le Pivot

Fonction TrouverPivot(T : tableau d'entier ; i,j : entier) : entier
/* Retourne l'indice d'un pivot si les cases T[i]…T[j] contiennent au moins deux valeurs distinctes, sinon retourne -1. */
Var k : Entier ;
Début
k ←i+1;
Tant que (k≤j) **faire**
Si (T[k]>T[i]) **alors Retourner(k) ;**
Si (T[k]<T[i]) **alors Retourner(i) ;**
k ← k + 1;
fintq;
Retourner (-1);
Fin.

2. Algorithme fonction Partition

Fonction Partition(T : tableau d'entier ; i,j : entier ; p :entier) : entier
/* Découper T en deux parties : de i à k-1 les éléments sont inférieurs à p, de k à j les éléments sont supérieur ou égaux à p. retourner cet indice k qui sépare ces deux parties. */
Var k,l :entier ;
Début

```
  k← i;
  l← j;
   Répéter
        Echange(T[k],T[l]) ;
        Tantque (T[k]<p) faire  k← k + 1;
        Tantque (T[l]≥p) faire  l← l − 1;
   Jusqu'à  (k>l)
   Retourner(k) ;
Fin.
```

3. Algorithme Quicksort

```
Fonction Quicksort(T : tableau d'entier ; i,j : entier)
/* Trie le tableau T entre les indices i et j. */
Var id,k : entier ;
Début
    id ← TrouverPivot(T, i, j);
    Si (id≠(-1)) alors
        k← Partition (T, i, j, T[id]);
       Quicksort(T,i,k-1) ;
       Quicksort(T,k,j) ;
    finsi ;
Fin.
```

Complexité de l'algorithme

- **La complexité en temps**, comptée en nombre de comparaisons d'éléments :
  – **Le pire cas** est obtenu si le tableau de départ est déjà trié et chaque appel de partition ne fait que constater que le premier élément est le plus petit du tableau. Soit p(n) la complexité la pire pour trier un tableau de longueur n,
  c'est-`a-dire une **complexité quadratique**, la même que pour le tri par sélection
  – **La meilleure** complexité en temps pour Tri Rapide est obtenue lorsque chaque appel récursif de Tri Rapide se fait sur un tableau de taille la moitié du tableau précédente, ce qui minimise le nombre d'appels récursifs : O(nlog(n)).
  La complexité moyenne de Tri Rapide est plus difficile à calculer; la moyenne de la complexité en temps sur toutes les permutations possibles est de l'ordre de O(nlogn), la même que pour le tri fusion. La complexité en moyenne est donc égale à la complexité dans le meilleur cas, et de plus c'est la meilleure complexité possible pour un algorithme de tri, ce qui nous permet de dire que Tri Rapide est un bon algorithme.
- **La complexité en espace**, QUICKSORT s'exécute en espace constant : on n'a pas besoin de recopier le tableau à trier, par contre il faut stocker les appels récursifs de QUICKSORT, et il peut y avoir O(n) appels récursifs à QUICKSORT.

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

1.4. Le tri par Tas (Heapsort)

Le tri par Tas (Heapsort en anglais), inventé par Williams en 1964, le principe de tri est basé sur une structure de donnée très utile, le tas.
La complexité bornée par $\Theta(n \log n)$ (dans tous les cas) et la mise en œuvre très simple

❑ **Arbres : D**éfinition

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

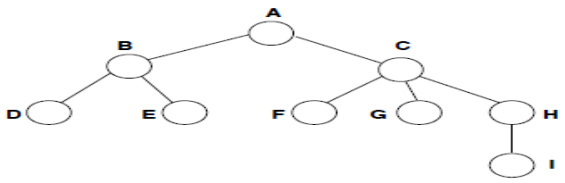


Figure 7. Exemple d'arbre

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 (ancestor) 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)

La hauteur 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)

La profondeur 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).

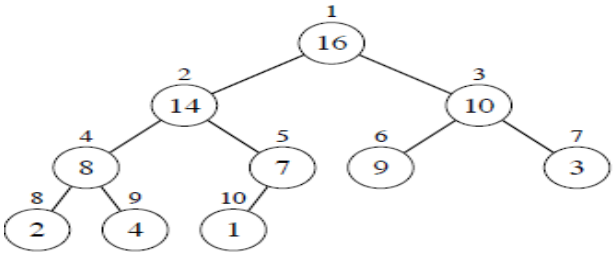


Figure 8. Exemple de tas

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 l'être de gauche à droite (s'il existe un fils droit alors il y a nécessairement un fils gauche).

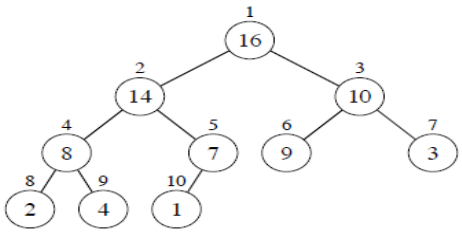
Propriété d'un tas

- 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

$$2^{h-1+1} - 1 + 1 = 2^h$$

- 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

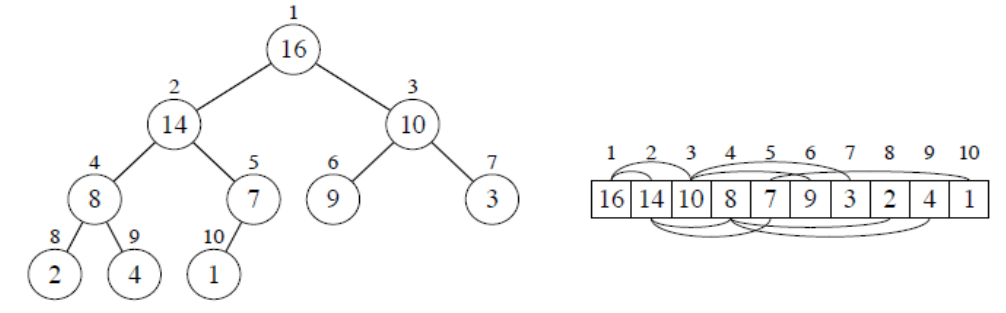


Figure 9. Implémentation par un tableau

- 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

2. Présentation des différentes méthodes de recherche

2.1. 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.

Soient T un tableau de n éléments et k l'élément qu'on recherche.
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), puisqu'il faudra au pire parcourir tout le tableau.

2.2. Recherche séquentielle : recherche dans un tableau trié

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

2.3. 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 : on va, 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 : t(n) satisfait l'équation de récurrence t(n) = t(n /2)+1, comme t(1) = 1, on en déduit t(n) = 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.