

Révision du cours Algorithmique et Complexité

Hamrouni Kamel

Esprit - 28 Juin 2011

Complexité

- La complexité = la mesure du nombre d'opérations fondamentales qu'exécute un algorithme
- La complexité est fonction de la taille de la donnée: $T(N)$
- Exemple: Calcul de la somme d'un tableau

```
s=0  
for (k=0 ; k < N ; k++)  
{   s=s + X[k]; }
```

→ $T(N) = N$

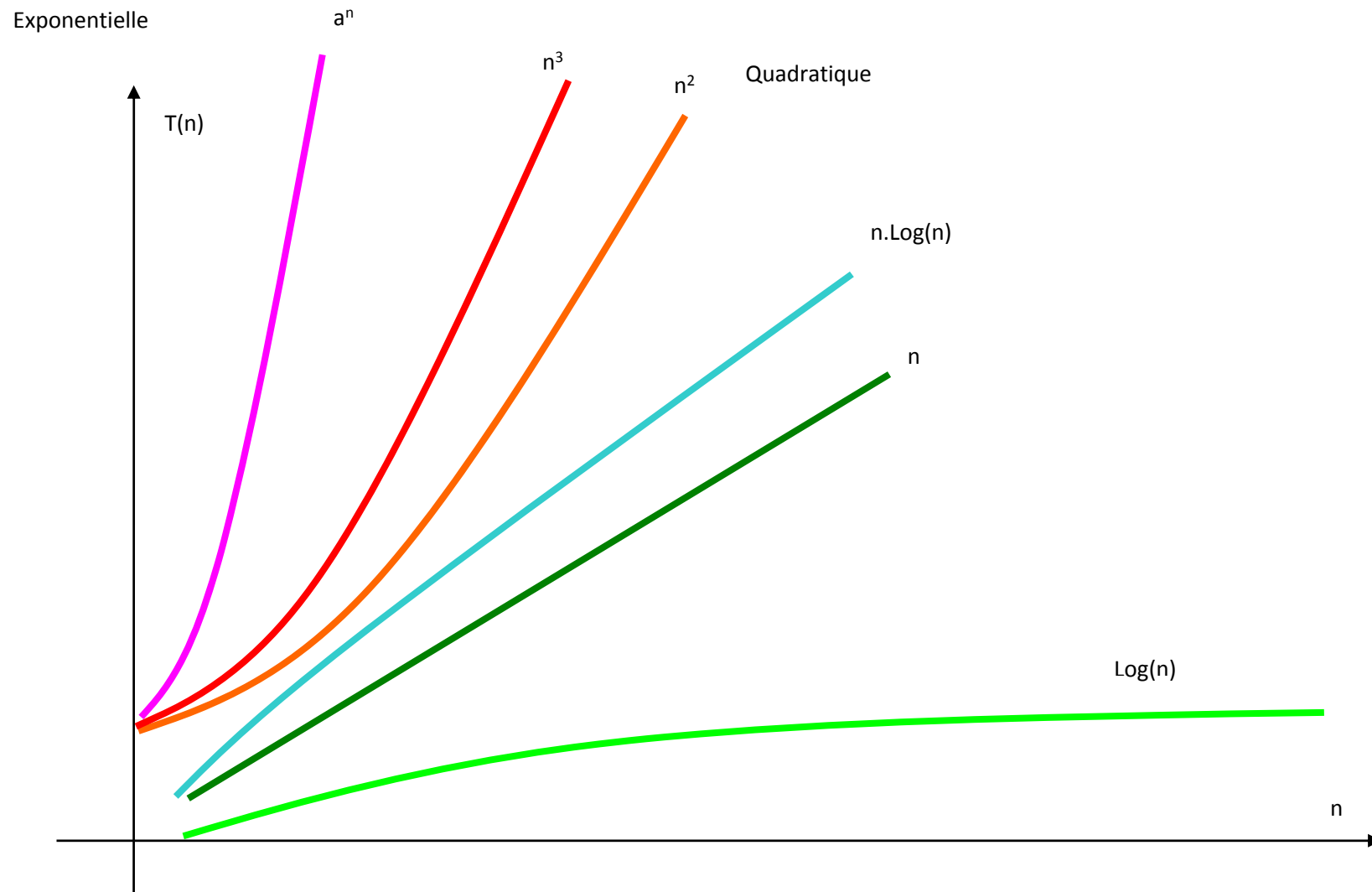
Classes de complexité

$$T(n) = 5n^2 + 375n + 2543$$

$$T(n) \approx n^2 = \Theta(n^2)$$

- **5 classes de complexité**
 - **$O(\log n)$** : Les algorithmes sub-linéaires dont la complexité est en général en $O(\log n)$.
 - **$O(n)$** : Les algorithmes linéaires en complexité $O(n)$
 - **$O(n \log n)$** : et ceux en complexité en $O(n \log n)$
 - **$O(n^k)$** : Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$
 - **$\text{Exp}(n)$** : Les algorithmes exponentiels

Classes de complexité



Comparaison de qq fonctions

	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	n^4	$\exp(n)$
1	0,00	1	0,00	1	1	1	3
2	1,00	2	2,00	4	8	16	7
3	1,58	3	4,75	9	27	81	20
4	2,00	4	8,00	16	64	256	55
5	2,32	5	11,61	25	125	625	148
6	2,58	6	15,51	36	216	1296	403
7	2,81	7	19,65	49	343	2401	1097
8	3,00	8	24,00	64	512	4096	2981
9	3,17	9	28,53	81	729	6561	8103
10	3,32	10	33,22	100	1000	10000	22026
11	3,46	11	38,05	121	1331	14641	59874
12	3,58	12	43,02	144	1728	20736	162755
13	3,70	13	48,11	169	2197	28561	442413
14	3,81	14	53,30	196	2744	38416	1202604
15	3,91	15	58,60	225	3375	50625	3269017
16	4,00	16	64,00	256	4096	65536	8886111
17	4,09	17	69,49	289	4913	83521	24154953
18	4,17	18	75,06	324	5832	104976	65659969
19	4,25	19	80,71	361	6859	130321	178482301
20	4,32	20	86,44	400	8000	160000	485165195

Types d'algorithmes

- Algorithme itératif
- Algorithme récursif
 - Récursivité linéaire d'ordre 1
 - Récursivité linéaire d'ordre 2
 - Récursivité de type « Diviser Régner »

Complexité d'un algorithme itératif

```
S=0  
for (k=0; k<N ; k++)  
    S=S+X[k]
```

$$T(n) = n$$

```
S=0  
for (k=0; k<N ; k++)  
    for (j=0; j<N ; j++)  
        S=S+X[k,j]
```

$$T(n) = n^2$$

```
k=0;  
while (X[k] != V && k<N)  
    k++;  
If(k<N)  
    cout << « Trouvé »;  
else  
    cout << « Non Trouvé »;
```

$$T(n) = \begin{cases} \textit{Meilleur} = 1 \\ \textit{Moyenne} = n / 2 \\ \textit{Pire} = n \end{cases}$$

```
for (k=0; k<N ; k++)  
    for (j=0; j<k ; j++)  
        S=S+X[k,j]
```

$$T(n) = \frac{n(n+1)}{2}$$

```
i=1  
While (i<N)  
{  
    i=2*i;  
    cout <<« Bonjour »;  
}
```

$$T(n) = \log n$$

```
K=1  
For (i=1; i<= N ; i++)  
{  
    k=k*2; }  
For (j=1; j <= k ; j++)  
{  
    cout<<« Bonjour »}
```

$$T(n) = 2^n$$

Algorithme récursif

- Récursivité : Souplesse et complexité
- de l'art et la manière de résoudre des problèmes qu'on ne sait pas résoudre soit même
- Définition: un algorithme récursif est un algorithme qui s'appelle soit même
- Exemple:

```
int Somme (int N)
{
    S=0
    for (k=0; k<N ; k++)
        S=S+k
    return S;
}
```

```
int Somme (int N)
{
    if (N==0) return 0;
    return Somme(N-1)+N;
}
```


Types de récursivité

- Récursivité simple :
un seul appel

```
int Somme (int N)
{
    if (N==0) return 0;
    return Somme(N-1)+N;
}
```

$$u_n = \begin{cases} 1 & \text{si } n < 2 \\ u_{n-1} + u_{n-2} & \text{sin on} \end{cases}$$

- Récursivité multiple
: 2 appels ou plus

```
Int U (int n)
{
    if (n < 2) return 1;
    else
    {
        return ( U(n-1) + U (n-2) );
    }
}
```

Types de récursivité

- Récursivité imbriquée

```
int Som (int N)
{   if (N==0) return 0;
    return Som(Som(N-1))+N;
}
```

- Récursivité mutuelle

```
int Pair(int n)
{   if (n == 0 ) return 1;
    else
    {   return ( Impair (n-1) ); }
}
```

```
int Impair (int n)
{   if (n == 0 ) return 0;
    else
    return ( Pair (n-1));
}
```

Récurtivité

- Règles et astuce:
 - Prévoir une condition d'arrêt
 - Le paramètre de la récursivité doit décrire un ordre strictement monotone
- Pour faire de la récursivité
 - Penser à la démonstration par récurrence en math
 - Pour trouver la solution à un problème il suffit de supposer qu'on connaît la solution du même problème à un ordre inférieur. Il faut connaître une solution particulière.

Récurtivité

- Tout algorithme contenant au moins une boucle admet une version récursive

```
int Somme (int N)
{
    S=0
    for (k=0; k<N ; k++)
        S=S+k
    return S;
}
```

```
int Somme (int N)
{
    if (N==0) return 0;
    return Somme(N-1)+N;
}
```

```
int Max (int X[ ], int N)
{
    PosMax=0
    for (k=1; k<N ; k++)
    {
        if (X[k] > X[PosMax])
            PosMax=k;
    }
    return PosMax;
}
```

```
int Max (int X[ ], int N)
{
    if (N==1) return 0;
    k= Max (X, N-1);
    if (X[k] > X[N-1]) return k;
    else return (N-1);
}
```

Récurtivité

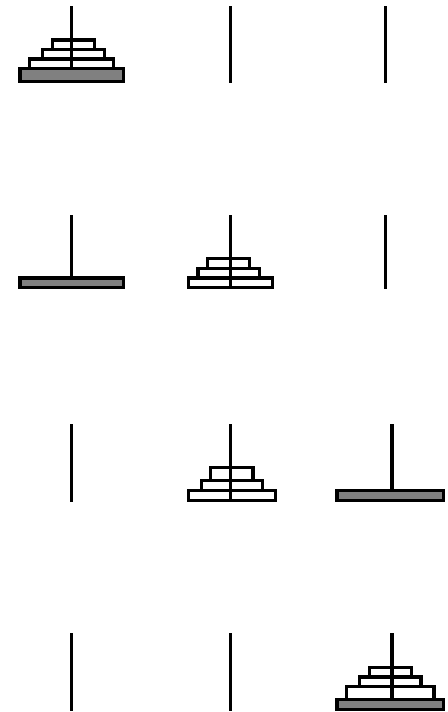
- Importance de la position de l'appel récursif
 - Les instructions placées avant l'appel sont exécutées à l'allée et
 - les instructions placées après l'appel sont exécutées au retour

```
void Toto (int N)
{
    if ( N==0) return;
    cout << N;
    Toto (N-1);
}
```

```
void Toto (int N)
{
    if ( N==0) return;
    Toto (N-1);
    cout << N;
}
```

Récurtivité / Pb de Hanoi

```
void Hanoi (int N, int a, int b , int c)
{  if (N==0)return;
   Hanoi (N-1, a, c , b);
   cout <<« Un disque de »<<a<<«  vers »<<c;
   Hanoi (N-1, b, a, c);
}
```



Récurtivité / Complexité

- Trois cas:
 - Récurtivité « linéaire d'ordre 1 »
 - Récurtivité « linéaire d'ordre 2 »
 - Récurtivité « diviser régner »
- 1^{er} cas: «linéaire d'ordre 1 »

```
int Somme (int N)
{
    if (N==0) return 0;
    return Somme(N-1)+N;
}
```

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n-1) + 1 & \text{si } n \geq 1 \end{cases}$$

```
int Max (int X[ ], int N)
{
    if (N==1) return 0;
    k= Max (X, N-1);
    if (X[k] > X[N-1]) return k;
    else return (N-1);
}
```

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(n-1) + 1 & \text{sin on} \end{cases}$$

Récurrance linéaire d'ordre 1

$$T(n) = a.T(n-1) + f(n) \longrightarrow T(n) = a^n \left(T(0) + \sum_{i=1}^n \frac{f(i)}{a^i} \right)$$

$$T(n) = \begin{cases} 0 & \text{si } n=0 \\ T(n-1)+1 & \text{si } n \geq 1 \end{cases} \longrightarrow T(n) = 1^n \left(0 + \sum_{i=1}^n \frac{1}{1^i} \right) = N$$

$$T(n) = \begin{cases} 0 & \text{si } n=1 \\ T(n-1)+1 & \text{sin on} \end{cases} \longrightarrow T(n) = 1^n \left(0 + \sum_{i=2}^n \frac{1}{1^i} \right) = N-1$$

$$T(n) = \begin{cases} 0 & \text{si } n=0 \\ 2T(n-1)+1 & \text{sin on} \end{cases} \longrightarrow T(n) = 2^n \left(0 + \sum_{i=1}^n \frac{1}{2^i} \right) = 2^n \cdot \frac{1}{2} \left(\frac{1 - \left(\frac{1}{2}\right)^n}{1 - \frac{1}{2}} \right)$$

$$T(n) = 2^n - 1$$

Exercices

Vérification de l'ordre d'un tableau:

Un tableau X est trié par ordre croissant si $x(i) \leq x(i+1)$ pour tout i. Elaborer un algorithme récursif permettant de vérifier qu'un tableau X est trié ou non Estimer sa complexité

Solution:

Pour que X soit ordonné il faut que les (N-1) premiers éléments soient ordonnés et que les deux derniers soient ordonnés. La condition d'arrêt est lorsque N=1. Dans ce cas le tableau est ordonné

```
int Vérifier (int *X, int N)
{
    if (N==1) return 1;
    if(X[N-2]> X[N-1]) return 0;
    return ( Vérifier (X, N-1) );
}
```

Complexité:

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(n-1) + 1 & \text{sin on} \end{cases}$$



$$T(n) = 1^n \left(0 + \sum_{i=2}^n \frac{1}{1^i} \right) = \sum_{i=2}^N 1 = N - 1$$

Exercices

Longueur d'une chaîne de caractères

Soit une chaîne de caractères terminée par le caractère '\0'. Ecrire un algorithme récursif permettant de déterminer sa longueur. Estimer sa complexité

Solution:

La longueur d'une chaîne est égale à 1 + la longueur de la chaîne qui commence au 2^{ème} caractère

```
int Longueur (char *S)
{
    if (S[0] == '\0') return 0;
    return (Longueur (S+1) + 1);
}
```

Complexité:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n-1) + 1 & \text{sin on} \end{cases}$$



$$T(n) = 1^n \left(0 + \sum_{i=1}^n \frac{1}{1^i} \right) = \sum_{i=1}^N 1 = N$$

Exercices

Conversion d'un nombre décimal en binaire:

Ecrire une fonction permettant de convertir un nombre entier positif en binaire .

Solution:

La représentation binaire d'un nombre est égale à la représentation binaire du quotient de ce nombre par 2 suivi du reste de sa division par 2. dans cette opération, les différents restes doivent être affichés à l'envers. Il faudra donc exploiter la propriété de la récursivité selon laquelle, les instructions placées après l'appel seront exécutées au retour

```
Int Binaire (int X)
{   if (X==0) return;
    Binaire (X/2);
    cout << X%2;
}
```

Récurrance linéaire d'ordre 2

Suite de Fibonacci

$$u_n = \begin{cases} 1 & \text{si } n < 2 \\ u_{n-1} + u_{n-2} & \text{sin on} \end{cases}$$

```
Int U (int n)
{  if (n < 2) return 1;
    else
    {   return ( U(n-1) + U (n-2) );
    }
}
```

Complexité

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ T(n-1) + T(n-2) + 1 & \text{si } n \geq 1 \end{cases}$$

Elle est de la forme:

$$T(n) = T(n-1) + T(n-2) + 1$$

C'est une équation linéaire
d'ordre 2

Comment la résoudre?

Récurrance linéaire d'ordre 2

- Résolution d'une équation linéaire d'ordre 2:

$$T(n) = T(n-1) + T(n-2) + 1$$

- Etape-1: annuler le second membre

$$S(n) = T(n) + 1$$

- Changement de variable:
 - On obtient :

$$S(n) - S(n-1) - S(n-2) = 0$$

- Déduire le polynôme caractéristique:

$$x^2 - x - 1 = 0$$

- Déterminer les racines de ce polynôme:

$$\frac{1+\sqrt{5}}{2} \quad \text{et} \quad \frac{1-\sqrt{5}}{2}$$

- La solution est:

$$S(n) = a \left(\frac{1+\sqrt{5}}{2} \right)^n + b \left(\frac{1-\sqrt{5}}{2} \right)^n$$

- Les coefficients a et b peuvent être déterminés en utilisant les conditions aux limites:

Récurrance linéaire d'ordre 2

- Conditions aux limites donnent deux équations à deux inconnues
- La résolution de ce système donne
- La solution finale est donc:
- Cette complexité est donc exponentielle

$$S(0) = 1 = a + b$$

$$S(1) = 1 = \frac{a+b}{2} + \frac{(a-b)\sqrt{5}}{2}$$

$$a = \frac{5+\sqrt{5}}{10} \quad \text{et} \quad b = \frac{5-\sqrt{5}}{10}$$

$$T(n) = \frac{5+\sqrt{5}}{10} \left(\frac{1+\sqrt{5}}{2} \right)^n + \frac{5-\sqrt{5}}{10} \left(\frac{1-\sqrt{5}}{2} \right)^n - 1$$

$$S(n) = T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Paradigme « diviser pour régner »

Exemple: Chercher le maximum d'un tableau de taille N

- Idée:
 - diviser le tableau en 2 sous-tableau de taille N/2 chacune
 - Déterminer le maximum de chaque sous-tableau
 - Prendre le plus grand des deux

```
int Maximum (int X[ ], int g, int d)
{
    if (G==D) return X[g] ;
    m= (g+d)/2;
    max1= Maximum (X, g, m);
    max2= Maximum (X, m+1, d);
    if (max1 > max2 ) return max1;
        else return max2;
}
```

Complexité:

$$T(n) = \begin{cases} 0 & \text{si } n=1 \\ 2T\left(\frac{n}{2}\right)+1 & \text{sinon} \end{cases}$$

C'est une formule récurrente de type « Diviser Régner »

Comment la résoudre?

Résolution des récurrences de type « diviser régner »

- Théorème:
- Si on a : $T(n) = a.T(n/b) + c.n^k$ alors $T(n)$ peut être approximée comme suit:

$a > b^k$	$T(n) = \Theta(n^{\log_b a})$
$a = b^k$	$T(n) = \Theta(n^k \log n)$
$a < b^k$	$T(n) = \Theta(n^k)$

Résolution des récurrences de type « diviser régner »

Exercice du maximum

$a=2$, $b = 2$, $k=0$

donc on a : $2 > 2^0$

On est donc dans le premier cas, donc:

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Θ

????

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + 1 & \text{sin on} \end{cases}$$

Notation de Landau

- Exemples: $T(n) = 2n + 2500$ alors $T(n) = \Theta(n)$

$$T(n) = 3n^2 + 2n + 15 \quad \text{alors} \quad T(n) = \Theta(n^2)$$

- Définition « Borne supérieure » : T est majoré par S

$$O: \quad T = O(S) \quad \Leftrightarrow \quad \exists n_0, \exists c \geq 0, \quad / \quad \forall n \geq n_0, \quad T(n) \leq c.S(n)$$

- Exemples: $n = O(n^2)$

- Définition « Borne exacte »

$$\Theta : \quad T = \Theta(S) \Leftrightarrow T = O(S) \quad \text{et} \quad S = O(T)$$

- Exemples: $2n + 5 = \Theta(n)$

Exercices « diviser pour régner »

Recherche d'une valeur V dans un tableau ordonné X composé de N éléments, selon la méthode « diviser régner »

Solution: connue sous le nom recherche dichotomique:
Diviser le tableau en deux. Si V est inférieur à l'élément du milieu chercher à gauche sinon chercher à droite

```
int Position (int X[ ], int g, int d, int V)
{
    if (g==d)
        if (X[g]==V) return g;
        else return -1;

    m=(g+d)/2;
    if (V <= X[m])
        return Position (X,g,m,V);
    else
        return Position (X,m+1,d,V);
}
```

Complexité:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{sinon} \end{cases}$$

$$a = 1, b = 2, k = 0 \quad \text{donc} \quad a = b^k \quad (1 = 2^0)$$

On est donc dans le 2^{ème} cas:

$$T(n) = \Theta(n^k \log n) = \Theta(\log n)$$

Pourquoi Estimer la complexité ?

Suite de Fibonacci

Solution itérative:

```
int U (int n)
{
    int a,b,c,k;
    a=b=c=1;
    for (k=2; k<=n ; k++)
    {
        c=a+b;
        a=b; b=c;
    }
    return c;
}
```

Complexité:

$$T(n) = n - 1$$

$$u_n = \begin{cases} 1 & \text{si } n < 2 \\ u_{n-1} + u_{n-2} & \text{sin on} \end{cases}$$

Solution récursive:

```
int U (int n)
{
    if (n < 2) return 1;
    else
    {
        return ( U(n-1) + U (n-2) );
    }
}
```

Complexité:

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Temps d'exécution

	Nb-additions		Nb-Secondes		
n	Non Récursif	Récursif	Non Récursif	Récursif	Récursif
20	19	10945	0,0000019	0,0010945	2*10-5 mn
30	29	1346260	0,0000029	0,134626	2*10-3 mn
40	39	165580140	0,0000039	16,558014	3*10-1 mn
50	49	3185141889	0,0000049	318,514189	5 mn
100					251 000 années

Temps d'exécution

n	Cout	sec	min	h	jour	an
10	122,991869	1,23E-05	2,05E-07	3,42E-09	1,42E-10	3,90E-13
20	15126,9999	1,51E-03	2,52E-05	4,20E-07	1,75E-08	4,80E-11
30	1860498	1,86E-01	3,10E-03	5,17E-05	2,15E-06	5,90E-09
40	228826127	2,29E+01	3,81E-01	6,36E-03	2,65E-04	7,26E-07
50	2,8144E+10	2,81E+03	4,69E+01	7,82E-01	3,26E-02	8,92E-05
60	3,4615E+12	3,46E+05	5,77E+03	9,62E+01	4,01E+00	1,10E-02
70	4,2573E+14	4,26E+07	7,10E+05	1,18E+04	4,93E+02	1,35E+00
80	5,2361E+16	5,24E+09	8,73E+07	1,45E+06	6,06E+04	1,66E+02
90	6,44E+18	6,44E+11	1,07E+10	1,79E+08	7,45E+06	2,04E+04
100	7,9207E+20	7,92E+13	1,32E+12	2,20E+10	9,17E+08	2,51E+06

Méthodes de Tri

Méthodes de tri

- Enoncé du problème
 - X: un tableau de longueur N
 - objectif: le trier par ordre croissant
- Plusieurs méthodes de tri
 - Tri par sélection
 - Tri par permutation (tri à bulles)
 - Tri par insertion
 - Insertion linéaire
 - Insertion binaire
 - Tri par fusion
 - Tri par partition (Tri rapide ou Quick sort)
- Objectif:
 - connaître le principe,
 - savoir établir l'algorithme,
 - évaluer sa complexité

Tri par Sélection

- Principe:
 - Sélectionner le minimum du tableau
 - Le placer au début du tableau
 - Répéter le processus

Solution itérative

```
void TriSelection(int *X, int N)
{  int i,j,k;
  for (i=0; i<N-1; i++)
  {    k=i;
    for (j=i+1; j<N; j++)
    {  if (X[j] < X[k]) k=j; }
    w=X[k]; X[k]=X[i]; X[i]=w;
  }
}
```

Complexité

La boucle sur j a une
longueur = (N-i-1)

Comme i est répété de 0 à
N-2, alors:

$$T(n) = \sum_{i=0}^{n-2} (n-i-1) = \sum_{y=n-1}^1 y = \frac{n(n-1)}{2}$$

Tri par Sélection

Solution itérative

```
void TriSelection(int *X, int N)
{   int i,j,k;
    for (i=0; i<N-1; i++)
    {
        k=i;
        for (j=i+1; k<N; k++)
        {   if (X[j] < X[k]) k=j; }
        w=X[k]; X[k]=X[i]; X[i]=w;
    }
}
```

Solution récursive

```
void TriSelection(int *X, int N)
{   int j,k;
    if( N==1) return;
    k=0;
    for (j=1; k<N; k++)
    {   if (X[j] < X[k]) k=j; }
    w=X[k]; X[k]=X[i]; X[i]=w;
    TriSelection (X, N-1)
}
```

Complexité

$$T(n) = \begin{cases} 0 & \text{si } n=1 \\ T(n-1) + (N-1) & \text{sinon} \end{cases}$$



Linéaire d'ordre 1

$$T(n) = 1^n \left(0 + \sum_{i=2}^n \frac{(i-1)}{1^i} \right) = \sum_{i=1}^{n-1} j = \frac{n(n-1)}{2}$$

Tri par permutation (à bulles)

- Principe:
 - Comparer les éléments adjacents 2 à 2
 - Permuter s'ils ne sont pas ordonnés
 - Parcourir le tableau plusieurs tant qu'il y a des éléments à permuter

Solution itérative

```
void TriBulle(int *X, int N)
{   int i,j,k;
    do
    {   perm=0;
        for (i=0; i<N-1; i++)
        {   if (X[i] > X[i+1])
                { w=X[i]; X[i]=X[i+1]; X[i+1]=w;
                    perm=1;
                }
        }
    } while (perm);
}
```

Complexité

La longueur de la boucle do-while est indéterminée.

Sa longueur est égale à:

1 si le tableau est déjà ordonné

N si le tableau est ordonné à l'envers

N/2 en moyenne

On doit donc parler de complexité au pire , au meilleur et moyenne

$$T(n) = \begin{cases} \text{meilleur} & : (n - 1) \\ \text{pire} & : n(n - 1) \\ \text{moyenne} & : \frac{n(n - 1)}{2} \end{cases}$$

Tri par permutation (à bulles)

- Solution récursive:
 - On remarque que à chaque itération, le maximum prend sa place à la fin du tableau
 - Donc: faire un passage sur le tableau
 - Rappeler la fonction pour qu'elle fasse la même chose sur le tableau de taille (n-1)

Solution récursive

```
void TriBulle(int *X, int N)
{   int i,j,k;
    if ( N ==1 ) return;
    perm=0;
    for (i=0; i<N-1; i++)
    {   if (X[i] > X[i+1])
        {   w=X[i]; X[i]=X[i+1]; X[i+1]=w;
            perm=1;
        }
    }
    if (perm) TriBulle (X, N-1);
}
```

Complexité

La complexité au pire est:

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(n-1) + (n-1) & \text{sinon} \end{cases}$$

Linéaire d'ordre 1:

$$T(n) = 1^n \left(0 + \sum_{i=2}^n \frac{(i-1)}{1^i} \right) = \sum_{i=2}^n (i-1) = \frac{n(n-1)}{2}$$

Tri par insertion linéaire

- Principe:
 - Supposer qu'une partie du tableau (0 à i-1) est triée (« Destination ») et l'autre partie (de i à N-1) n'est pas triée (« Source »)
 - Prendre le premier élément de la partie « Source » X[i]
 - L'insérer dans la partie « destination » de telle sorte que celle-ci reste triée

Solution itérative

```
void TriInsertionLinéaire(int *X, int N)
{   int i,j,k;
    for (i=1;i<N;i++)
    {   A=X[i];
        k=i-1;
        while (k>=0 && A < X[k])
        {   X[k+1]=X[k];
            k--;
        }
        X[k+1] = A;
    }
}
```

Complexité

La longueur de la boucle while est indéterminée.

Sa longueur est égale à:

1 si le tableau est déjà ordonné

i si le tableau est ordonné à l'envers

i/2 en moyenne

On doit donc parler de complexité au pire , au meilleur et moyenne

$$T(n) = \begin{cases} \text{meilleur} &= \sum_{i=1}^{n-1} 1 = (n-1) \\ \text{pire} &= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \\ \text{moyenne} &= \sum_{i=1}^{n-1} i / 2 = \frac{n(n-1)}{4} \end{cases}$$

Tri par insertion binaire

- Principe:
 - C'est le même principe sauf que la recherche de la position se fait d'une façon binaire (dichotomique)

```
void TriInsertionBinaire(int *X, int N)
{
    int i,k;
    for (i=1;i<N;i++)
    {
        A=X[i];
        k=Position (X,0, i-1,A);
        Decaler (X, k,i-1)
        X[k] = A;
    }
}
```

```
Void Decaler (int *X, int p, int q)
{
    int i;
    for (i=q; i>=p; i--)
        X[i+1] = X[i]
}
```

```
Int Position (int *X, int g, int d, int A)
{
    if (g==d)
        if (A< X[g]) return g;
        else return g+1;
    m=(g+d)/2;
    if (A<= X[m])
        return Position (X, g,m, A);
    else
        return Position (X,m+1, d, A);
}
```

$$T(n) = \sum_{i=1}^{n-1} T_{\text{Position}}(i)$$

$$T_{\text{Position}}(i) = \Theta(\log i)$$

$$T(n) = \sum_{i=1}^{n-1} \Theta(\log i) = \Theta\left(\sum_{i=1}^{n-1} \log i\right) = \Theta(n \log n)$$

Tri par Fusion

- Principe:
 - Cette méthode opère selon le méthode « diviser régner »
 - Divise le tableau en 2 parties de même longueur
 - Trie chaque partie
 - Fusionne les deux parties triées

Solution récursive

```
void TriFusion(int *X, int g, int d)
{   int m;
    if (N==1) return;
    m = (g+d)/2;
    TriFusion (X, g, m);
    TriFusion (X, m+1, d);
    Fusionner (X, g, m, d);
}
```

```
Void Fusionner(int *X, int g, int m, int d)
{   int T[1000], i, j, k;
    i = g ; j = m+1 ; k = 0;
    while (i <= m && j <= d)
    {   if (X[i] < X[j])
        T[k++] = X[i++];
        else
        T[k++] = X[j++];
    }
    while (i <= m)
        T[k++] = X[i++];
    while (j <= d)
        T[k++] = X[j++];
    for (i = 0; i < k; i++)
        X[g+i] = T[i];
}
```

Tri par Fusion

Complexité:

En analysant la fonction TriFusion:

La complexité de Fusionner est
égale au meilleur ($n/2$) et au pire
(n)

En appliquant le théorème on
obtient:

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ 2T(n/2) + T_{\text{Fusionner}}(n) \end{cases}$$

$$T_{\text{Fusionner}}(n) = \Theta(n)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$a = 2, b = 2, k = 1 \quad \text{donc} \quad a = b^k$$

$$T(n) = \Theta(n \log n)$$

Tri par Partition (Quick Sort)

Principe:

- Cette méthode opère selon le méthode « diviser régner »
- Partitionne le tableau en 2 parties P1 et P2 telles que $P1 < P2$
- Tous les éléments de P1 sont inférieurs au plus petit de P2
- Répète le processus de partitionnement pour chaque partie

Solution récursive

```
void TriRapide(int *X, int N)
{   int k;
    if ( N > 1 )
    {       k = partitionner(X,0 , n-1 ) ;
            Trier (X, k +1 );
            Trier (X + k+1 , n-k-1);
    }
}
```

```
int partitionner (int x[],int g,int d)
{   int i,j;
    int y,t;
    if ( g >=d ) return l;
    i=g; j=d;
    y=x[g];
    do
    {       while(x[i]<y && i<=d) i++;
            while(x[j]>y && g<=j) j--;
            if(i<j)
                { t=x[i];x[i]=x[j];x[j]=t;}
    }while(i<j );
    return j;
}
```

Tri par Partition (Quick Sort)

Complexité:

La complexité de TriRapide admet deux cas:

Cas meilleur: à chaque fois le partitionnement fournit deux parties de taille égale à $n/2$

Cas pire: à chaque fois le partitionnement fournit deux parties de taille égale à (1) et (n-1)

La complexité de Partitionner est toujours égale à n

$$T_{\text{Partition}}(n) = \Theta(n)$$

Complexité: Cas meilleur

A chaque fois le partitionnement fournit deux parties de taille égale à $n/2$

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ 2T(n/2) + T_{\text{Partition}}(n) \end{cases}$$

$$T(n) = \Theta(n \log n)$$

Complexité: Cas pire

A chaque fois le partitionnement fournit deux parties de taille égale à (n-1) et 1

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(n-1) + T(1) + T_{\text{Partition}}(n) \end{cases}$$

$$T(n) = T(n-1) + \Theta(n)$$

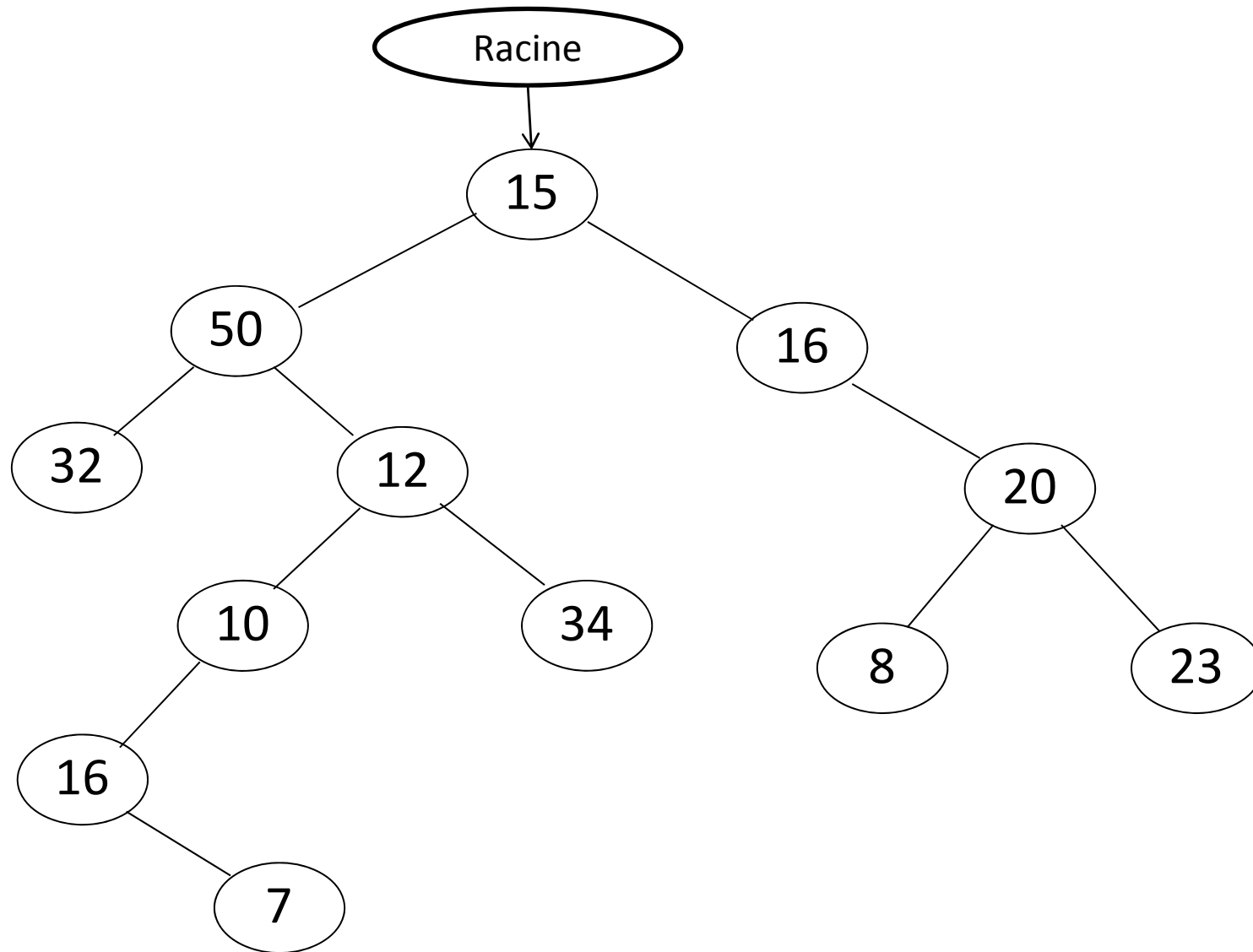
$$T(n) = \sum_{k=1}^n \Theta(k) = \Theta\left(\sum_{k=1}^n k\right) = \Theta(n^2)$$

Récapitulatif

Méthode	Complexité
Tri par sélection	$T(n) = \Theta(n^2)$
Tri par permutation (à bulles)	$T(n) = \Theta(n^2)$
Tri par insertion linéaire	$T(n) = \Theta(n^2)$
Tri par insertion binaire	$T(n) = \Theta(n \log n)$
Tri par fusion	$T(n) = \Theta(n \log n)$
Tri par partition (Tri rapide)	$T(n) = \Theta(n \log n)$

Arbre Binaire & Arbre Binaire de Recherche

Arbre Binaire



Arbre Binaire

- Définition récursive d'un Arbre Binaire (AB)

$$\textit{ArbreBinaire} = \begin{cases} \textit{Vide} & \textit{ou} \\ \textit{Père} + \textit{ArbreGauche} + \textit{ArbreDroit} \end{cases}$$

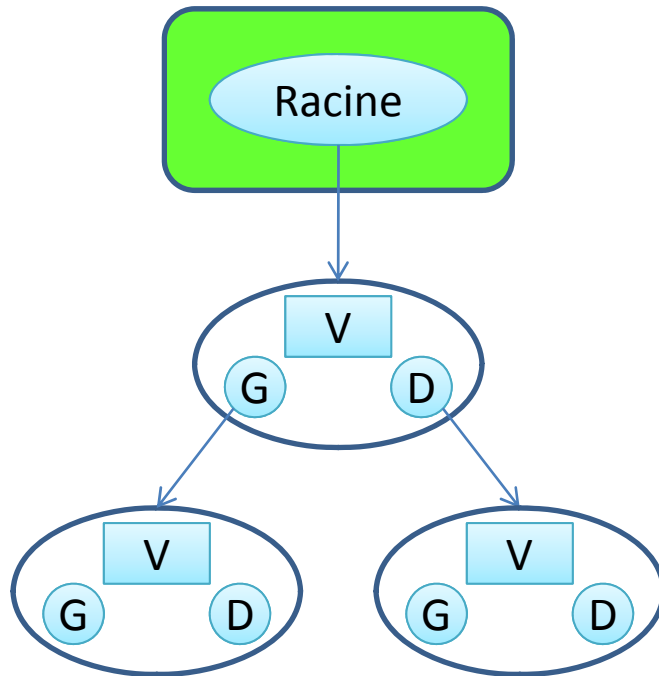
- ➔ Les algorithmes de parcours des arbres sont récursifs:
- ➔ à chaque appel récursif, on se retrouve en face d'un père, de sa famille gauche et de sa famille droite
- ➔ Si le père existe: traiter le père, faire un appel récursif avec la famille gauche et un appel récursif avec la famille droite

AB / types de parcours

- Parcours en profondeur
 - Priorité aux descendants avant de voir les frères
 - Algorithmes récursifs (Pile)
 - 3 sous-types de parcours
 - Parcours préfixe: Père, fils gauche, fils droit
 - Parcours infixé: fils gauche, Père, fils droit
 - Parcours postfixé: fils gauche, fils droit, Père
- Parcours en largeur
 - Priorité aux frères ➔ par génération
 - Algorithmes utilisant une File d'attente

AB/ Structures de données

Représentation d'un arbre par chaînage



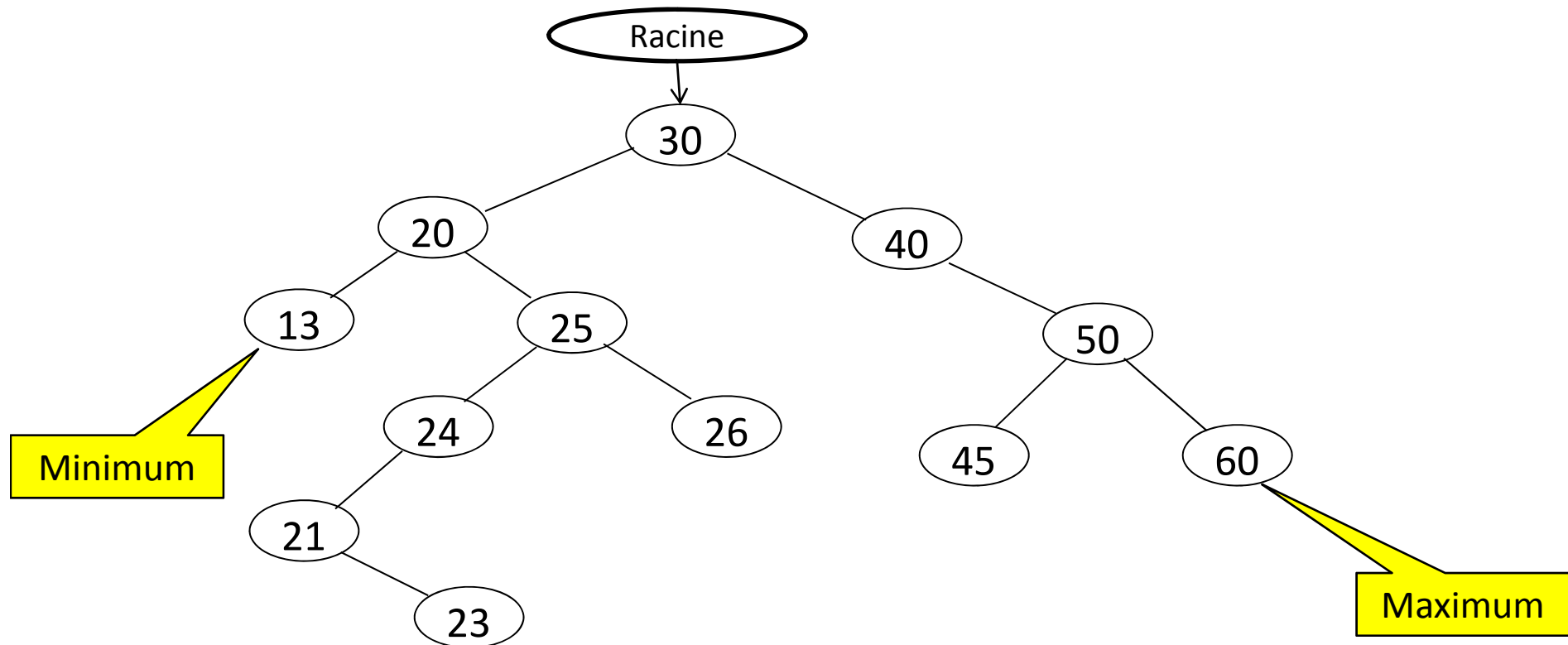
```
struct Arbre  
{  Node *Racine;  
};
```

```
struct Node  
{  int Val;  
    Node *G,*D;  
};
```


Arbre Binaire de Recherche (ABR)

- Arbre binaire vérifiant les conditions:
 - Tous les éléments sont distincts
 - Fils gauche < père < Fils droit

- Intérêt:
 - Recherche rapide
 - Tri immédiat



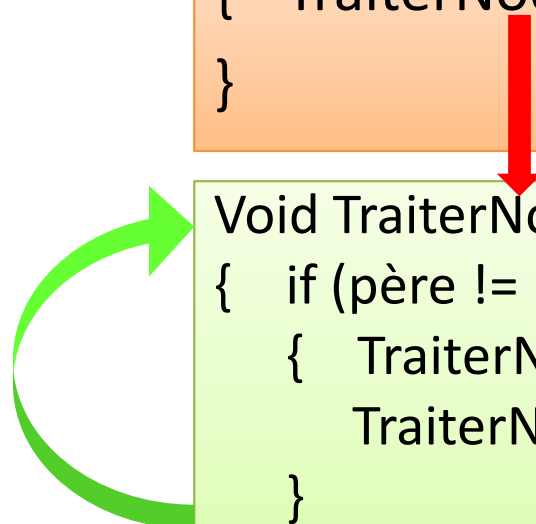
Arbre Binaire de Recherche (ABR)

- Pour travailler avec un ABR il faut résoudre les problèmes suivants:
 - Comment accepter la multiplicité des valeurs ?
 - Ajouter un champ dans chaque nœud donnant le nombre d'exemplaires
 - Comment ajouter une valeur dans un ABR ?
 - Comment supprimer une valeur ?
 - Comment chercher une valeur ?
 - Comment afficher les valeurs ordonnées ?

ABR: Remarque générale

- Sachant que la définition d'un ABR est récursive tous les algorithmes sont récursifs
- La récursivité porte sur un nœud et non pas sur l'arbre

```
Void TraiterArbre (Arbre A)
{  TraiterNode (A.Racine);
}
```



```
Void TraiterNode (Node *Pere)
{  if (père != NULL)
    {  TraiterNode (Père->G);
        TraiterNode (Père->D);
    }
}
```

ABR : Ajouter une valeur

Algorithme

- Commencer par la racine
- Si (la valeur < valeur du père) aller à gauche
- Sinon aller à droite
- Quand on arrive à la terre ajouter un nœud et mettre la valeur dedans

```
void Ajouter (Arbre A, int X)
{  AjouterNode (A.Racine,X); }

Void AjouterNode (Node *Père, int X)
{  if (Père == NULL)
    {    Père = new Node;
        Père->V = X;
        Père->G=Père->D=NULL;
    } else
    if (X < Père->V)
        AjouterNode (Père-> G);
    else
        AjouterNode (Père->D);
}
```

ABR : Chercher une valeur

Algorithme

- Commencer par la racine
- Si (la valeur cherchée = valeur du père) c'est trouvé
- Sinon si la valeur < la valeur du père aller à droite sinon aller à gauche
- Si on arrive à la terre c'est que la valeur n'existe pas

```
int Chercher (Arbre A, int X)
{   return ChercherNode (A.Racine,X); }

intChercherNode (Node *Père, int X)
{   if (Père == NULL) return 0;
    if (X == Père->V) return 1;
    if (X < Père->V)
        return ChercherNode (Père-> G);
    else
        retrurn ChercherNode (Père->D);
}
```

ABR : Afficher les éléments triés

Algorithme

- Appliquer le parcours infixe
- Commencer à la racine
- Au niveau de chaque père, afficher d'abord le sous-arbre gauche, ensuite le père, ensuite le sous-arbre droit

```
int Trier (Arbre A)
{ return TrierNode (A.Racine); }

Int TrierNode (Node *Père)
{ if (Père != NULL) ;
  {   TrierNode (Père-> G);
      cout << Père->V;
      TrierNode (Père-> D);
  }
}
```

ABR : Supprimer un élément

Algorithme

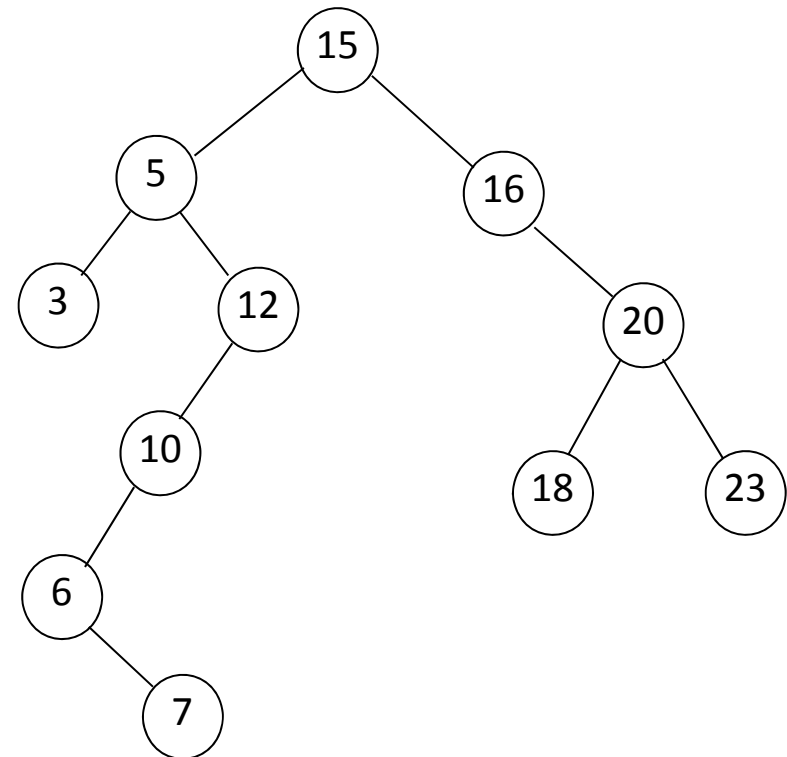
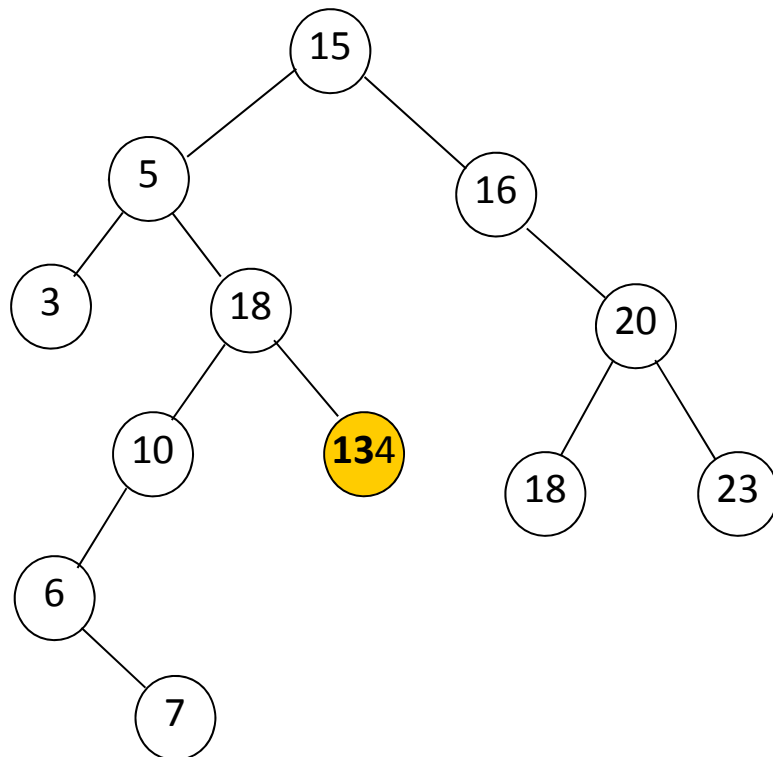
- Dans le cas d'un ABR acceptant la multiplicité des valeurs, la suppression d'un élément peut ne pas entraîner la suppression d'un nœud
- La suppression d'un nœud est délicate, il faut trouver une solution pour les fils de ce père

Trois cas:

- Père n'ayant pas de fils
 - On peut le supprimer
- Père ayant un seul fils
 - On peut le supprimer, son fils sera confié à son père
- Père ayant 2 fils
 - Le remplacer son plus petit descendant droit ou bien par son plus grand descendant gauche

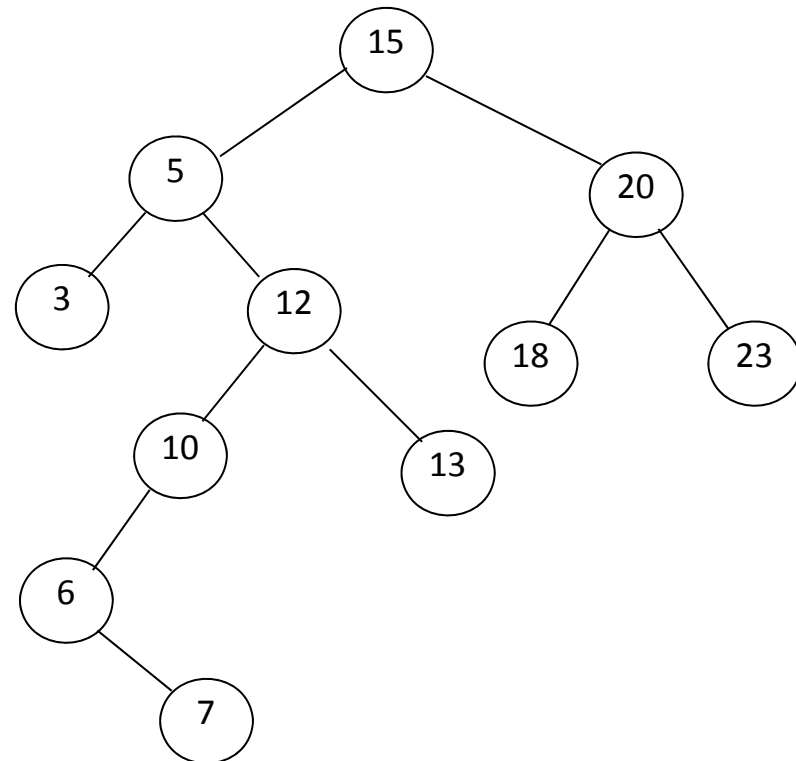
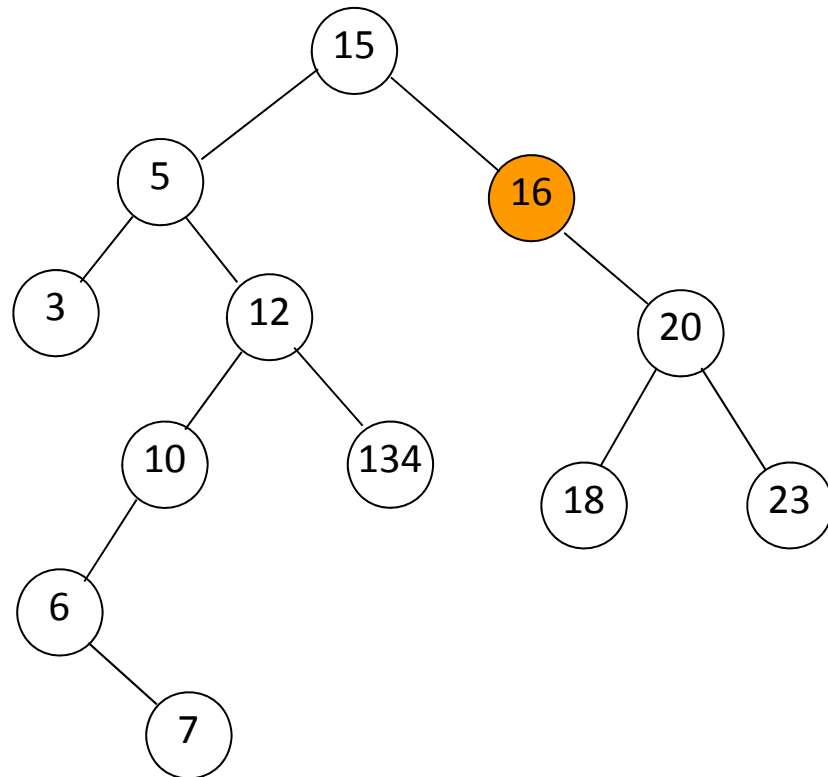
ABR : Supprimer un élément

1^{er} cas: 0 fils



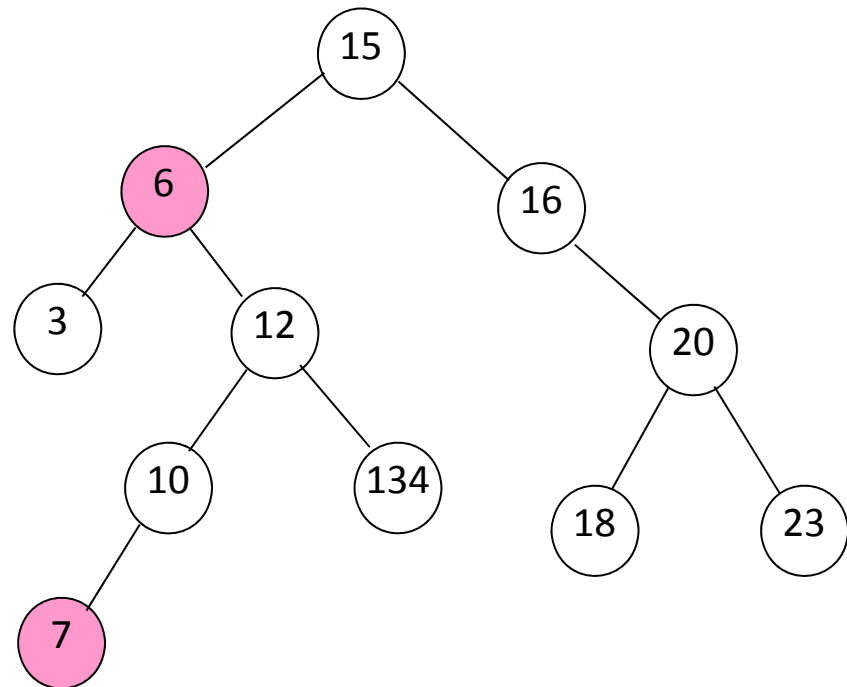
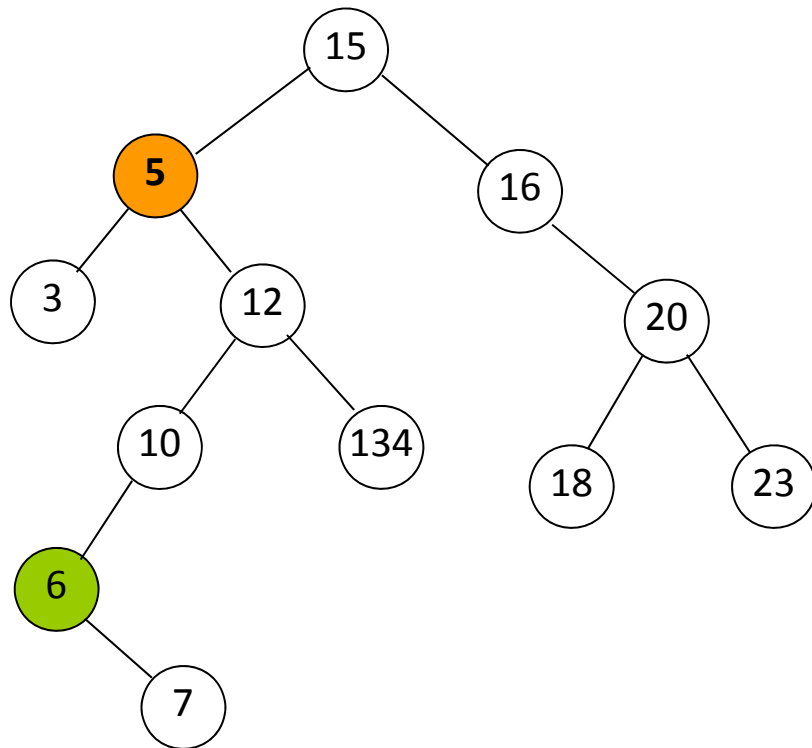
ABR : Supprimer un élément

1^{er} cas: 1 fils



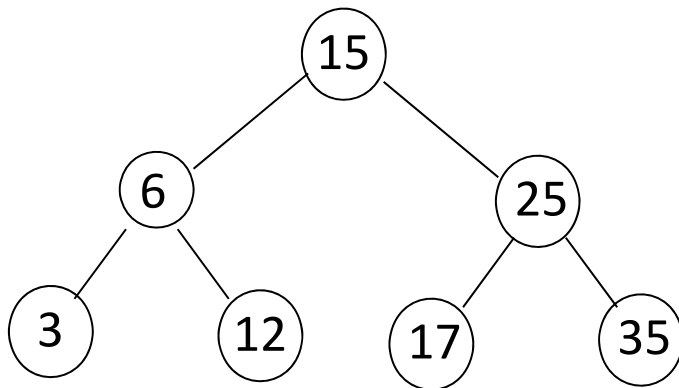
ABR : Supprimer un élément

1^{er} cas: 2 fils

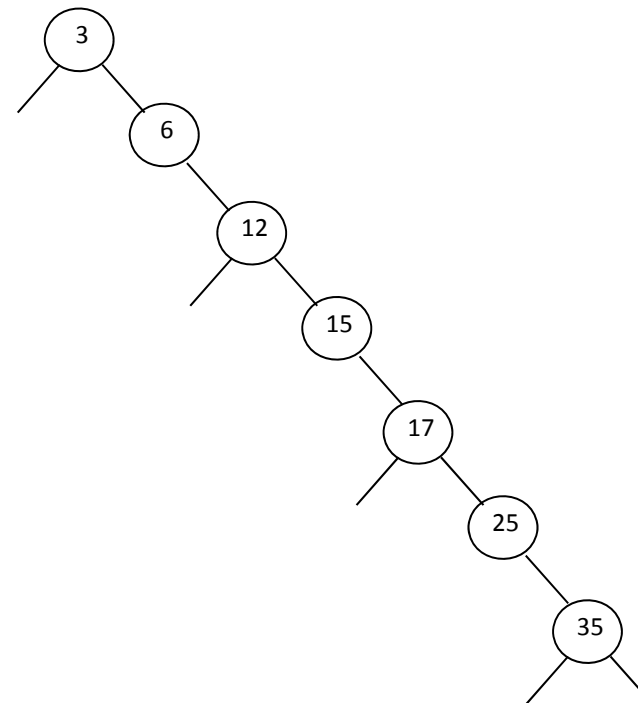


ABR: Complexité

Arbre équilibré: toutes les branches ont la même longueur
 $T(n) = \log(n)$

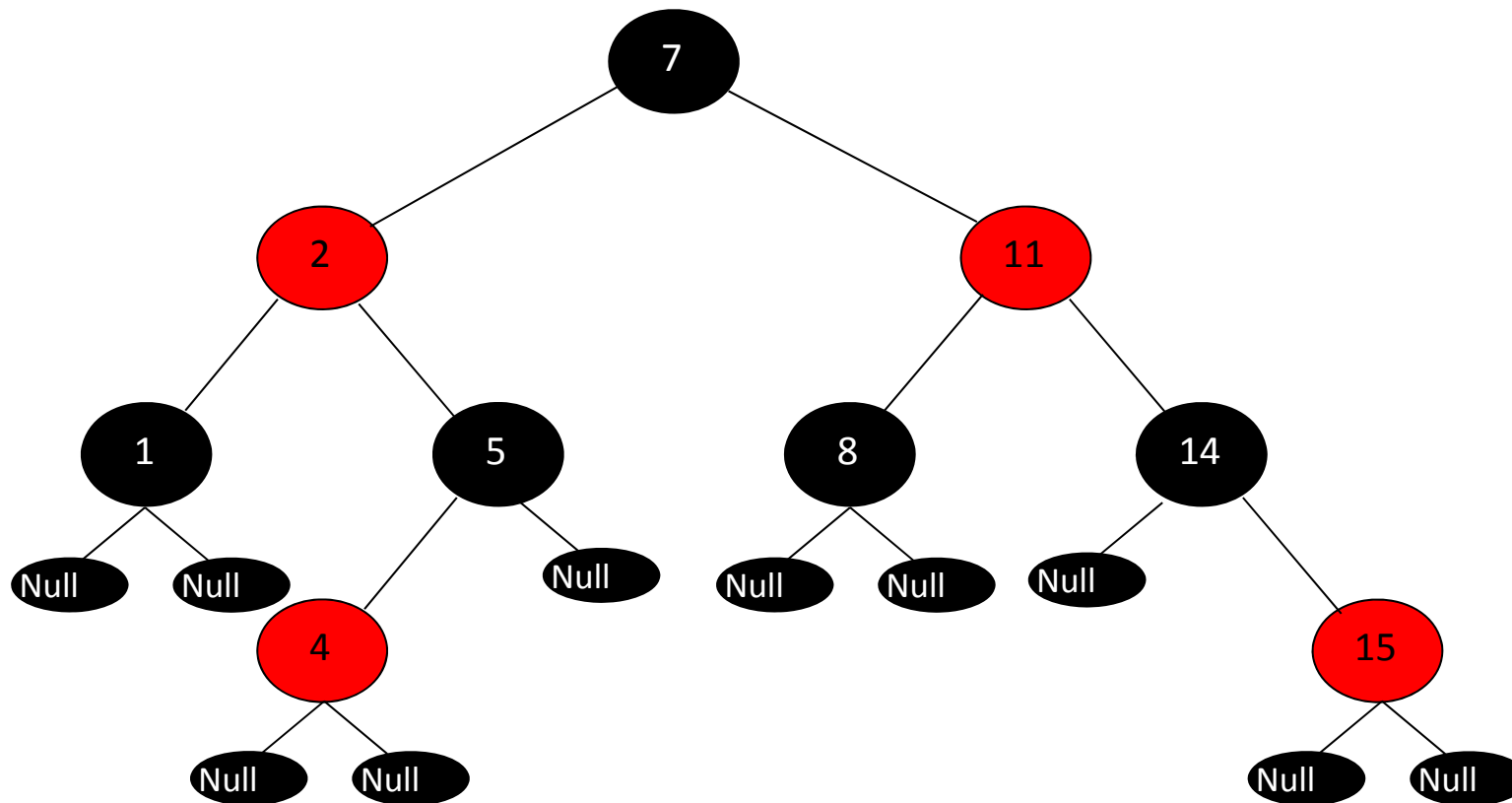


Arbre en une seule branche
 $T(n) = n$



Arbre « Rouge et Noir »

- On ne peut pas avoir un arbre parfaitement équilibré
- ➔ on se contente d'un arbre presque équilibré: les branches auront la même longueur (+/-) 1



Correction de sujets d'Examen

Examen (15 juin 2010)

Exercice-1 : (5 points) Considérer la fonction TRUC placée dans l'encadré-1.

- a- Quelle est la tâche réalisée par la fonction TRUC ? Expliquer votre réponse.
- b- Estimer la complexité de la fonction TRUC en nombre de comparaisons « $X[j+1] < X[j]$ »
- c- Donner une version récursive de la fonction TRUC.
- d- Calculer la complexité de la version récursive

```
void TRUC (int *X, int N)
{   int k, j, T;
    for (k = N-1 ; k >= 0 ; k = k -1)
    {   for (j= 0 ; j < k ; j++)
        if (X[j+1] < X[j] )
        {   T=X[j] ;
            X[j] = X[j+1];
            X[j+1] = T;
        }
    }
}
```

Examen (15 juin 2010)/corrigé

a- La fonction TRUC effectue un tri par ordre croissant du tableau X composé de N éléments par la méthode de « tri à bulles ». En effet, la for (j= effectue un parcours du tableau compare deux éléments adjacents et les permute si le 2^{ème} est inférieur au précédent

b- Complexité

$$T(N) = \sum_{k=N-1}^1 k = \frac{N(N-1)}{2}$$

En effet : la boucle « for (j= » effectue k comparaisons et la boucle « for (k » répète k de (N-1) à 1

c- Version récursive :

```
void TRUC (int *X, int N)
{   int k, j, T;
    If(N<2) return  // condition d'arrêt

    for (j= 0 ; j < N-1 ; j++)
    {   if (X[j+1] < X[j] )
        {   T=X[j] ;
            X[j] = X[j+1];
            X[j+1] = T;
        }
    }

    TRUC(X, N-1);
}
```

Examen (15 juin 2010)/corrigé

d- Complexité de la version réursive :

- En analysant la fonction on obtient une formule récurrente linéaire d'ordre 1.
- Pour la résoudre il faut appliquer la formule :
- Ce qui donne:

$$T(N) = \begin{cases} 0 & \text{si } N = 1 \\ T(N-1) + N - 1 & \text{sin on} \end{cases}$$

$$u_n = a^n \left(u_0 + \sum_{i=1}^n \frac{f(i)}{a^i} \right)$$

$$T(N) = 1^N (T(1) + \sum_{i=2}^N \frac{(i-1)}{1^i})$$

$$T(N) = \sum_{j=1}^{N-1} j = \frac{N(N-1)}{2}$$

Examen (15 juin 2010)/corrigé

Exercice-2 : (4 points)

a- Ecrire une fonction permettant de lire des entiers positifs et les afficher dans l'ordre inverse d'entrée (ie. Le dernier lu sera le premier affiché). Attention : l'utilisateur n'annonce pas le nombre d'entiers à lire. Un entier nul signifie la fin de la liste des entiers. Il ne faut utiliser ni un tableau ni une pile.

b- Estimer sa complexité en nombre d'instruction d'affichage

a- La fonction doit être réursive.
Pour lire et afficher à l'envers, il faut que la lecture se fasse avant l'appel récursif et l'affichage après l'appel récursif.

```
void Afficher ()  
{ int x ;  
  cin >> x  
  if (x != 0 ) Afficher () ;  
  cout << x ;  
}
```

b- Complexité :

La complexité doit être estimée en fonction de la taille de la donnée. Ici, cette taille n'est pas fournie explicitement. Nous allons supposer que le nombre d'entiers est N. L'appel à Afficher à l'intérieur de la fonction permettra de générer (N-1) entiers. Donc :

$$T(N) = \begin{cases} 1 & \text{si } N = 0 \\ T(N-1) + 1 \end{cases}$$

Avec N : le nombre d'éléments tapés

On applique la formule et on trouve :

$$T(N) = 1^N (T(0)) + \sum_{i=1}^N \frac{1}{1^i} = N + 1$$

Examen (15 juin 2010)/corrigé

Exercice-3 : (6 points): Soient deux entiers A et B strictement positifs. On voudrait écrire une fonction pour multiplier l'entier A par l'entier B en n'utilisant que des opérations d'addition. On remarque que $A*B = A+A+\dots+A$ (B fois).

a- On vous demande d'écrire une fonction récursive « `int Mul1 (int A, int B)` » permettant de calculer et de retourner le résultat de la multiplication de A par B . Calculer sa complexité $T(B)$ en nombre d'opérations d'addition après avoir établi une formule de récurrence.

$A*B = A+A+A+\dots+A$ (B fois)

→ $A*B = A*(B-1) + A$

```
int Mul1( int A, int B)
{
    if (B==1) return A ;
    Return ( Mul1(A, B-1) +A) ;
}
```

Complexité:

L'analyse de la fonction permet d'établir la formule récurrente de la complexité

$$T(B) = \begin{cases} 0 & \text{Si } B = 1 \\ T(B-1) + 1 \end{cases}$$

C'est une récurrence linéaire d'ordre 1. On applique la formule et on trouve:

$$T(B) = 1^B (T(1)) + \sum_{i=2}^B \frac{1}{1^i} = B - 1$$

Examen (15 juin 2010)/corrigé

Exercice-3 : (6 points)

c- Sachant que $B/2$ donne le quotient entier de B divisé par 2, trouver une relation entre $A*B$ et $A*(B/2)$

d- Ecrire une fonction « `int Mul2 (int A, int B)` » permettant de calculer et de retourner le résultat de la multiplication de A par B selon la démarche « diviser pour régner ». Estimer sa complexité $T(B)$ en nombre d'opérations d'addition après avoir établi une formule de récurrence.

C- La relation demandée est :

$$A * B = \begin{cases} (A * \frac{B}{2}) * 2 & \text{si } B \text{ est pair} \\ (A * \frac{B}{2}) * 2 + A & \text{si } B \text{ impair} \end{cases}$$

```
Int Mul2(int A, int B)
{
    int x;
    if (B==1 ) return A;
    x= Mul2 (A, B/2);
    if ( B % 2 == 0)
        return (x+x) ;
    else
        return (x+x+A);
}
```

Complexité

$$T(B) = \begin{cases} 0 & \text{si } B = 1 \\ T\left(\frac{B}{2}\right) + 2 & \end{cases}$$

$$T(B) = \Theta(B^0 \log(B)) = \Theta(\log(B))$$

Examen (15 juin 2010)/corrigé

Exercice-4 : (5 points): Etant donné un arbre binaire A composé de N nœuds.

Chaque nœud contient : X : une valeur entière, G : un pointeur sur le fils gauche et D : un pointeur sur le fils droit. La structure « arbre » contient un seul pointeur « racine » pointant sur le premier nœud de l'arbre (voir encadré-2). On suppose que l'arbre A est équilibrée (toutes les branches ont la même longueur).

- a- On suppose dans cette question que l'arbre binaire A n'est pas un ABR (il ne vérifie pas les conditions d'un arbre binaire de recherche). Ecrire une fonction « `int Maximum (arbre *A)` » permettant de chercher et de retourner la valeur maximale se trouvant dans l'arbre. Donner une formule récurrente donnant sa complexité $T(N)$ en nombre de comparaisons, puis estimer $T(N)$.
- b- On suppose dans cette question que A est un arbre binaire de recherche. Ecrire une fonction « `int Maximum (arbre *A)` » permettant de chercher et de retourner la valeur maximale se trouvant dans l'arbre. Estimer sa complexité $T(N)$ en nombre de comparaisons.

Examen (15 juin 2010)/corrigé

a- Si A n'est pas un ABR, il faudra parcourir tout l'arbre pour trouver le maximum. Au niveau de chaque famille, prendre le plus grand entre la valeur du père, le maximum de la sous-famille gauche et le maximum de la sous-famille droite.

Complexité: L'arbre étant supposé équilibré. Donc toutes les branches ont la même longueur. La complexité mesure le nombre de comparaisons en fonction du nombre de nœuds qu'on suppose égal à N

$$T(N) = \begin{cases} 0 & \text{si } N = 1 \\ 2T\left(\frac{N}{2}\right) + 1 & \text{sinon} \end{cases}$$

```
int Maximum (Arbre A)
{ return Max (A.Racine) ; }

int Max (Node *pere)
{ int Sup,SupGauche,SupDroit ;
  if (pere == NULL) return -9999;
  Sup =pere->X;
  SupGauche = Max(pere->G);
  SupDroit   = Max (pere -> D);
  if (SupGauche> Sup)
    Sup=SupGauche;
  if (SupDroit  > Sup)
    Sup=SupDroit;
  return Sup ;
}
```

$$T(N) = \Theta(N^{\log_b a}) = \Theta(N)$$

Examen (15 juin 2010)/corrigé

b- Si A est un ABR, la valeur maximale se trouve au bout de la branche droite. Il suffit de descendre dans l'arbre à l'aide d'une boucle while.

```
Int Maximum (Arbre A)
{
  Node *père ;
  Père = A.Racine ;
  While ( père -> D != NULL)
  {
    Père = père -> D ;
  }
  Return ( père -> X) ;
}
```

Complexité: Puisque nous avons supposé que l'arbre est équilibré, la complexité de cette fonction est égale à la longueur d'une branche qui est égale à la hauteur de l'arbre qui est $\log N$. N étant le nombre de nœuds de l'arbre



$$T(N) = \Theta(\log N)$$