Chapitre 3

RÉCURSIVITÉ ET COMPLEXITÉ DES ALGORITHMES RÉCURSIFS

Plan

- Principe de la récursivité
- Forme Itérative / forme récursive
- Récursivité terminale
- Calcul de la complexité des algorithmes récursifs

Définition

- Algorithme récursif :
  - un algorithme qui s'appelle lui-même
- Tout algorithme récursif doit :
  - avoir au moins un cas qui ne comporte pas d'appel récursif : **cas de base (ou condition d'arrêt)** (par exemple lorsque le tableau possède 0 éléments)
  - définir les bons cas de base :
    - ils doivent être atteignables quelque soit l'exemple en entrée
    - (par exemple en s'assurant que le tableau dans l'appel récursif est plus petit que celui en entrée)
- **Exemple :** Le calcul de la factorielle de N.

  N!= N*(N-1)*(N-2)*…*2*1 , on peut écrire ainsi N!= N*(N-1)!

Itératif vs récursif

1 Solution itérative

```
fonction fact(n : entier) : entier
   i, f : entier
   f ← 1
   pour i de 2 à n faire
          f ← f *i
   finpour
   return f
fin
```

2 Solution récursive

```
fonction fact(n : entier) : entier
    si (n=1) ou (n=0) alors
                return 1
              sinon
                return n * fact(n-1)
     finsi
fin
```

Cas de base

Appel récursif

Appel de `fact(5)` récursif

- *Phase de descente récursive*

  Appel à **`fact(5)`**

  Appel à **`fact(4)`**

  Appel à **`fact(3)`**

  Appel à **`fact(2)`**

  Appel à **`fact(1)`**

  Appel à **`fact(0)`**

- *Condition terminale*
  - Retour de la valeur 1

- *Phase de remontée* (après l'appel à fact(n-1) on multiplie par n : la récursivité n'est pas terminale)

  Retour de la valeur 1

  Retour de la valeur 2

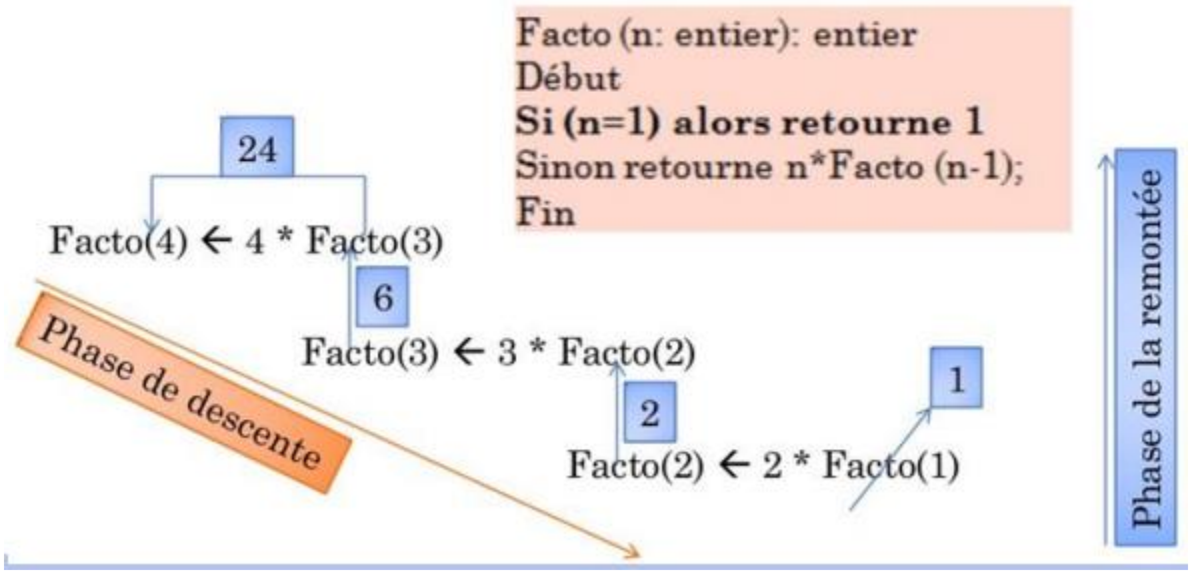
  Retour de la valeur 6

  Retour de la valeur 24

  Retour de la valeur 120

Exécution d’un appel récursif

- L’exécution d’un appel récursif passe par deux phases :
  - la phase de descente
  - la phase de remontée
- Lors de la **phase de descente**, chaque appel récursif fait à son tour un appel récursif
- En arrivant à la condition d’arrêt, on commence la **phase de remontée** qui se poursuit jusqu’à ce que l’appel initial soit terminé, ce qui termine le processus récursif



Types de récursivité

- On distingue plusieurs types de récursivité :
  - Récursivité **simple** : c'est une fonction qui contient dans son corps un seul appel récursif

    Exemple : Fonction factorielle
  - Récursivité **multiple** : c'est une fonction qui contient plus qu'un appel récursif dans son corps

    Exemple: Calcul du nombre de combinaisons en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1, & si\ p = 1\ ou\ p = n \\ C_{n-1}^{p} + C_{n-1}^{p-1}, & sinon \end{cases}$$

```
Fonction combinaison(n : entier, p : entier) : entier
        si (p=1) ou (p=n) alors return 1
                sinon
                    return combinaison (n-1, p) + combinaison(n-1, p-1)
        finsi
fin
```

Types de récursivité

- Récursivité **mutuelle** : Des fonctions sont dites mutuelles récursives si elles dépendent les unes des autres

  Exemple : La définition de la parité

$$pair(n) = \begin{cases} vrai, & si\ n = 0 \\ impair(n-1), & sinon \end{cases} \qquad impair(n) = \begin{cases} faux, & si\ n = 0 \\ pair(n-1), & sinon \end{cases}$$

- Récursivité **imbriquée** : Lorsque un appel récursif est réalisé à l'intérieur d'un autre appel récursif

  Exemple : La fonction d'Ackermann

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0\ et\ n = 0 \\ A\big(m-1, A(m,n-1)\big), & sinon \end{cases}$$

```
Ackermann(m : entier, n : entier) : entier
  si(m=0) alors return n + 1 finsi
  si (m>0) et (n=0) alors
       return Ackermann(m-1, 1)
    sinon
      return
     Ackermann(m-1, Ackermann(m, n-1))
   finsi
fin
```

Récursivité terminale

- Un algorithme est **récursif terminal** si l'appel récursif est la toute dernière instruction réalisée

  -ce n'est pas le cas dans fact : après l'appel récursif il faut faire une multiplication :

  return n * (fact (n-1))

  – tous les résultats des appels à fact(n-1), fact(n-2), etc. doivent être stockés dans une pile

  fact(5) ← fact(4) ← fact(3) ← fact(2) ← fact(1)

  120 *5 **24** *4 **6** *3 **2** *2 **1**

  – c'est le cas dans somme :

```
Fonction somme(n : entier, r : entier) : entier
  si (n<=1) alors return n + r
            sinon
               return somme (n - 1, r + n)
    finsi
fin
```

  – l'addition r + n est faite avant l'appel pas de stockage de résultats intermédiaires, les appels successifs sont vus comme des égalités

  som(5,0) = som(4,5) = som(3,9) = som(2,12) = som(1,14) = 15

Terminale

```
void f(int n){
  if(n==0) System.out.println("Hello");
  else f(n-1);
}
```

Non terminale

```
void f(int n) {
  if(n>0) f(n-1);
  System.out.println("Hello");
}
```

Comment transformer une récursivité non terminale ?

- Ajouter un paramètre qui stocke le résultat intermédiaire
- Ce paramètre est appelé **paramètre d'accumulation**
- Avantages :
  - théoriquement plus de nécessité de garder en mémoire la pile d'appels récursifs
  - ces programmes peuvent être écrits facilement de manière itérative

```
fonction factoriel(n : entier, resultat : entier) : entier
    si (n=1) alors
        return resultat
    sinon
        return factoriel(n-1, resultat *n)
  finsi
fin
```

Calcul de complexité

- La complexité d'un algorithme récursif se calcule par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche

- Il y a 2 types d'équations de récurrence :

type 1: $C(n) = a*C(n-b) + f(n)$

type 2: $C(n) = a*C(n/b) + f(n)$: diviser pour régner

Avec pour chacun **un cas de base : C(.) =** … exemple $C(0)= 1$

Calcul de complexité de type 1 : Exemple 1

- **Exemple 1 :** La fonction factorielle (avec C(n) la complexité d'un appel à Factoriel(n))

```
fonction factoriel(n : entier) : entier                C(n) dépend de C(n-1)
    si (n=1) alors          1  (opération logique : test)
            return 1        0
        sinon
            return n * factoriel(n-1)   [C(n-1)] + 2 (les opérations arithmétiques – et *)]
    finsi
Fin
```

C(n) = C(n-1) + 3

$$C(n) = \begin{cases} 1, & n = 1 \\ C(n-1) + 3, & sinon \end{cases}$$

Calcul de complexité de type 1 : Exemple 1

- Pour calculer la solution générale de cette équation, on peut procéder par substitution :

$C(n) = 3 + C(n-1)$

$= 3 + [3 + C(n-2)]$

$= 3 + [3 + (3 + C(n-3))]$

$= \ldots$

$= 3*i + C(n-i)$

$= \ldots$

$= 3*(n-1) + C(n-(n-1))$ pour $i = n-1$

$= 3*(n-1) + C(1) = 3*(n-1) + 1 = 3*n - 2$

$C(n) = 3n-2$

➔ **$C(n) = O(n)$**

Calcul de complexité de type 1 : Exemple 1

- Pour calculer la solution générale de cette équation, on peut aussi procéder par somme d'équations :
- $C(n) = \cancel{C(n-1)}+3$

$\cancel{C(n-1)} = \cancel{C(n-2)}+3$

$\cancel{C(n-2)} = \cancel{C(n-3)}+3$

…

$\cancel{C(2)} = \cancel{C(1)}+3$

$\cancel{C(1)} = 1$

$C(n) = 1 + 3*(n-1)$

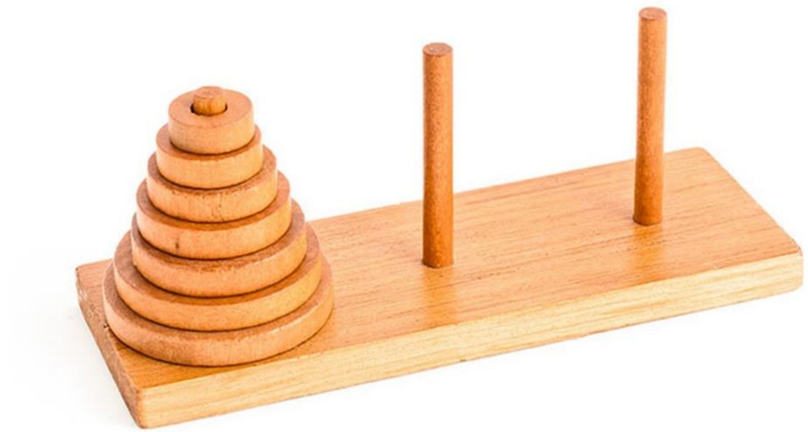
$C(n) = 3n-2$

➔ **$C(n) = O(n)$**

Calcul de complexité de type 1 : Exemple2

Exemple 2 :   Tour de Hanoï

Déplacer n rondelles d'un piquet à un autre avec la condition :
Tout au long des déplacements, on ne met jamais une rondelle plus grande sur une rondelle plus petite



Solution : fonction récursive

Déplacer n (>0) rondelles du piquet 1 au piquet 3 revient à :

- Déplacer n-1 rondelles (les plus petites) du piquet 1 au piquet 2
- Déplacer la rondelle la plus grande du piquet 2 au piquet 3
- Déplacer n-1 rondelles (les plus petites) du piquet 2 au piquet 3

$$C(n) = \begin{cases} 2*C(n-1) + 1 & \text{si } n>0 \\ 0 & \text{si } n=0 \end{cases}$$

Calcul de complexité de type 1 : Exemple 2

Exemple 2 : $C(n) = 2*C(n-1) + 1$ si $n>0$

0 si $n=0$

- Pour calculer la solution générale de cette équation, on peut procéder par somme d'équation :
- $2^{0}*\ C(n) = 2^{0}*\ \cancel{2*C(n-1)} + 2^{0}*\ 1$

$\cancel{2^{1}*\ C(n-1)} = \cancel{2*\ 2*C(n-2)} + 2^{1}*1$

$\cancel{2^{2}*\ C(n-2)} = \cancel{2^{2}*\ 2*C(n-3)} + 2^{2}*\ 1$

$\cancel{2^{3}*\ C(n-3)} = \cancel{2^{3}*\ 2*C(n-4)} + 2^{3}*\ 1$

…

$\cancel{2^{n-1}*\ C(1)} = \cancel{2^{n-2}*\ 2*C(2)} + 2^{n-2}*\ 1$

$\cancel{2^{n-}*\ C(0)} = 2^{n-2}*\ 0$

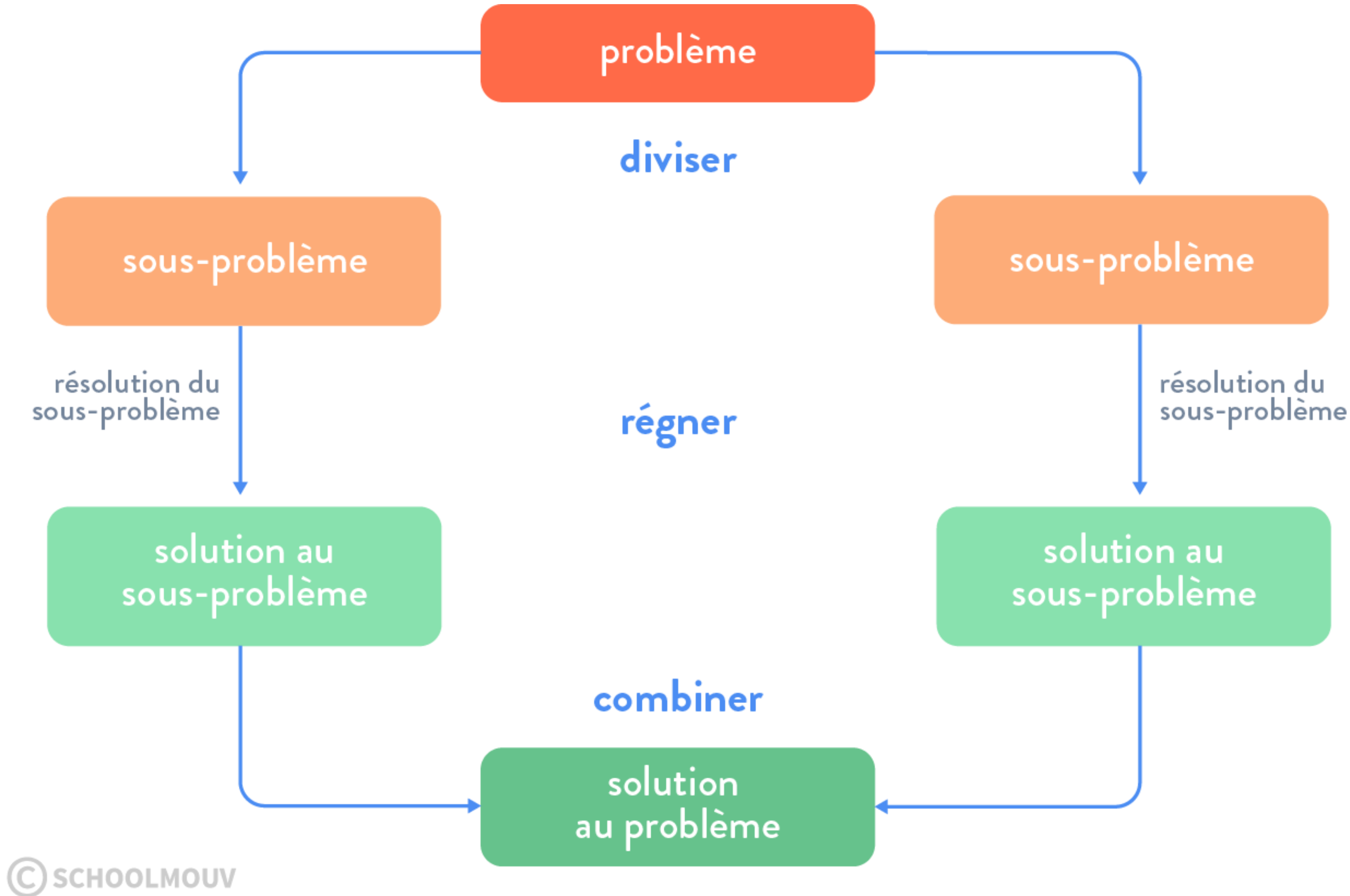
- $C(n) = 2^{n-2}*0 + 1*\sum_{i=0}^{n-1} 2^{i}$
- $= 0+1*(2^{n}-1)$
- $C(n) = 2^{n}-1$

Paradigme "diviser pour régner"

Principe:

- Spécifier la solution du problème en fonction de la (ou des) solution(s) d’un (ou de plusieurs) sous-problème(s) plus simple(s) à traiter de façon récursive
- Le paradigme "diviser pour régner " parcourt trois étapes à chaque appel récursif à savoir :
  - **Diviser** : le problème en un certain nombre de sous-problèmes de taille moindre
  - **Régner** : sur les sous-problèmes en les résolvant d’une façon récursive ou directement si la taille d’un sous-problème est assez réduite
  - **Combiner** : les solutions des sous-problèmes en une solution globale pour le problème initial

Paradigme "diviser pour régner"



Remarques

- La plupart des traitements itératifs simples sont facilement traduisibles sous forme récursive (exemple du `for`)
- L'inverse est faux
- Il arrive même qu'un problème ait une solution récursive triviale alors qu'il est très difficile d'en trouver une solution itérative

Calcul de la complexité d'un algorithme "diviser pour régner"

- Le temps d'exécution d'un algorithme "diviser pour régner" se décompose suivant les trois étapes du paradigme de base
  - **Diviser** : le problème en **a** sous-problèmes chacun de taille **1/b** de la taille du problème initial. Soit **D(n)** le temps nécessaire à la division du problèmes en sous-problèmes
  - **Régner** : soit **a*C(n/b)** le temps de résolution des a sous-problèmes
  - **Combiner** : soit **Comb(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes
- Finalement le temps d'exécution global de l'algorithme est :

$$C(n) = a*C(n/b) + D(n) + Comb(n)$$

- Soit la fonction f(n) qui regroupe D(n) et Comb(n).
- C(n) est alors définie :

$$\mathbf{C(n) = a*C(n/b) + f(n)}$$

Equation de partition

- Le "théorème général" permet de connaître le comportement asymptotique des équations de récurrence qui s'écrivent sous la forme suivante :

$$C(n) = a*C(n/b) + f(n)$$

- Pour l'utiliser il faut définir le comportement asymptotique de la fonction f(n) par rapport à $n^{log_b a}$

Théorème de résolution de la récurrence

Pour $f(n) = O(n^k)$, la relation de récurrence se définit comme suit :

$$\mathbf{C(n) = a*C(n/b) + O(n^k)}$$

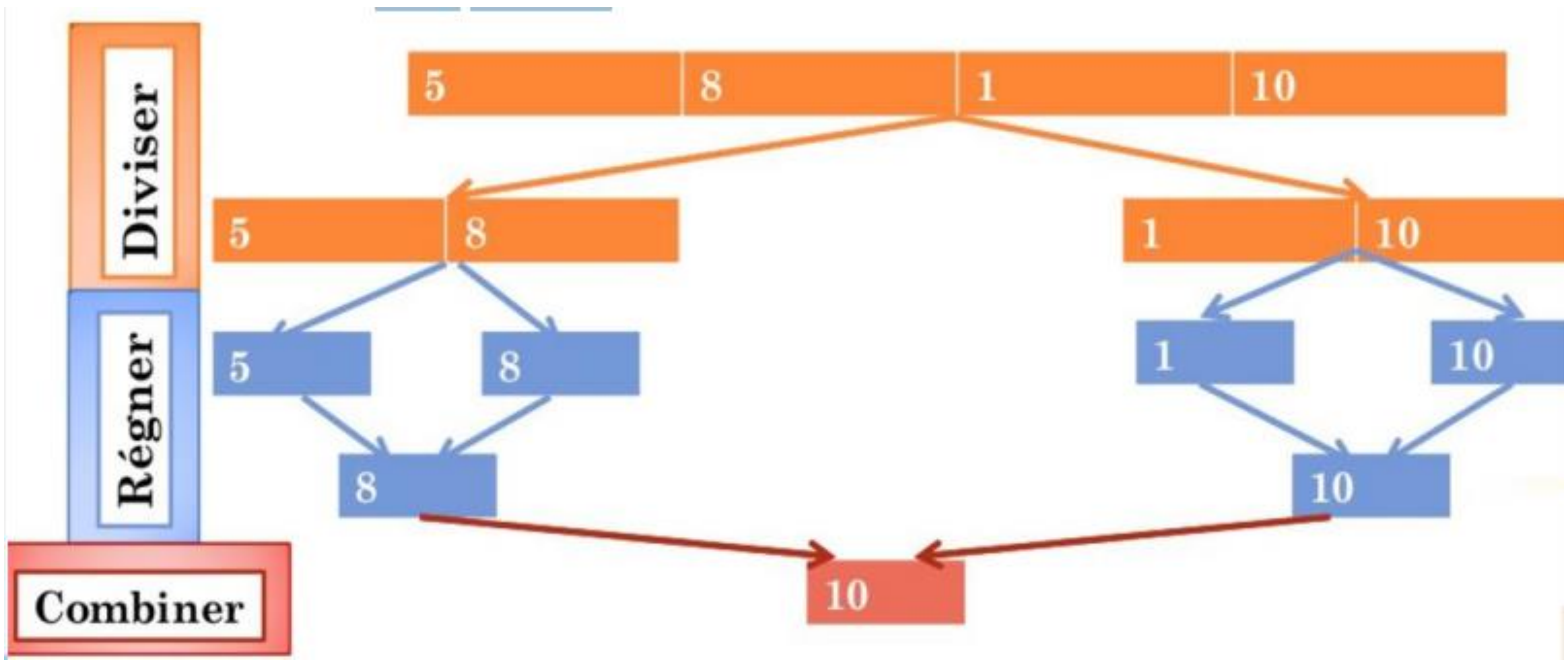
Ainsi il faut comparer $\mathbf{n^k}$ à $n^{log_b a}$ ou particulièrement il faut comparer les puissances de n : k et $log_b a$

Le "théorème général" permet d'obtenir une expression de la forme $log_b a$ complexité de C(n) comme suit :

1. Si $a > b^k$ ➔ $C(n) = O(n^{log_b a})$
2. Si $a = b^k$ ➔ $C(n) = O(n^k log_b n)$
3. Si $a < b^k$ ➔ $C(n) = O(f(n)) = O(n^k)$

Recherche du maximum

- Soit Tab un tableau à n éléments, écrire une fonction récursive permettant de rechercher l'indice du maximum dans Tab en utilisant le paradigme "diviser pour régner"



Recherche du maximum

```
Fonction maximum(tab : tableau d'entier, deb : entier, fin : entier) :
entier
    si (deb = fin) alors return deb
        sinon
            /* division du problème en 2 sous-problèmes */
            m ← (deb + fin)/2
            /* régner sur le 1 sous-problèmes */
             k1 ← maximum(tab, deb, m)
            /* régner sur le 2 sous-problèmes */
             k2 ← maximum(tab, m+1, fin)
            /* combiner les solutions */
             si (tab[k1] > tab[k2]) alors return k1
                        sinon  return k2
             finsi
    finsi
fin
```

C(n)

C(n/2)

C(n/2)

$$C(n) = 2*C(n/2) + c$$

Complexité recherche du maximum

D'après le théorème du paradigme "diviser pour régner ", on a:

$$C(n) = a*C(n/b) + O(n^k)$$

1. Si $a > b^k$ ➔ $C(n) = O(n^{log_b a})$
2. Si $a = b^k$ ➔ $C(n) = O(n^k log_b n)$
3. Si $a < b^k$ ➔ $C(n) = O(f(n)) = O(n^k)$

- Complexité de l'algorithme de recherche de maximum

$$C(n) = 2*C(n/2) + c$$

$$a = 2,\ b = 2,\ k = 0 \rightarrow a = 2 > b^k = 2^0 = 1$$

$$\rightarrow C(n) = O(n^{log_b a})$$

$$log_2 2 = 1$$

$$\rightarrow C(n) = O(n^1) = O(n)$$

Recherche binaire ou dichotomique

- Hypothèse : Liste triée par ordre croissant
- A chaque itération, on compare l’élément recherché (x) à l’élément du milieu du tableau
- Si l’élément du milieu est supérieur à x alors on recherche l’élément dans la sous liste inférieure sinon on la recherche dans la sous liste supérieure
- Exemple : n=20 x=50

							↤	↦												
Index	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Elément	06	08	09	13	18	37	37	59	60	60	61	68	70	80	80	83	83	89	91	96
								> 50	> 50											

Recherche dichotomique (binaire)

```
Fonction recherche_binaire(L : tableau d'entier, inf : entier, sup : entier,
x : entier) : booléen
si (inf <= sup) alors
    mil ← (inf + sup)div 2
    si (x = L[mil]) alors  return vrai
          sinon
                si (L[mil] > x) alors
                        return recherche_binaire(L, inf, mil, x)
                     sinon
                        return recherche_binaire(L, mil+1, sup, x)
                finsi
     finsi
    sinon return faux
fin
```

C(n)

C(n/2)

C(n) = C(n/2) + c

Complexité recherche dichotomique

D'après le théorème du paradigme "diviser pour régner", on a:

$C(n) = a*C(n/b) + O(n^k)$

1. Si $a > b^k$ ➔ $C(n) = O(n^{log_b a})$
2. Si $a = b^k$ ➔ $C(n) = O(n^k log_b n)$
3. Si $a < b^k$ ➔ $C(n) = O(f(n)) = O(n^k)$

Complexité de l'algorithme de la recherche dichotomique

$C(n) = C(n/2) + c$

$a = 1, b = 2, k = 0 \rightarrow a = b^k$

➔$C(n) = O(n^k log_b n) = O(n^0 log_2 n) = O(log_2 n)$

Algorithme du tri fusion

```
Fonction Tri_fusion (L : tableau d'entier, deb ; entier, fin :entier) : tableau
d'entier
   L1, L2: tableau d'entier[(fin-deb + 1)/2]
   si (deb > fin) alors
          retourner L[max(deb, fin)]
      sinon
         mil ← (deb + fin) div 2
          L1  ← Tri_fusion(L, deb, mil)
          L2  ← Tri_fusion(L, mil+1, fin)
          Retourner Fusion_triée(L1, L2, mil-deb+1, fin-mil)
    finsi
fin
```

C(n)

C(n/2)

C(n/2)

Cf(n) = n

C(n) = 2*C(n/2) + n

Complexité tri fusion

D'après le théorème du paradigme "diviser pour régner ", on a:

$$C(n) = a*C(n/b) + O(n^k)$$

1. Si $a > b^k$ ➔ $C(n) = O(n^{\log_b a})$
2. Si $a = b^k$ ➔ $C(n) = O(n^k \log_b n)$
3. Si $a < b^k$ ➔ $C(n) = O(f(n)) = O(n^k)$

Complexité de l'algorithme du tri fusion

$$C(n) = 2*C(n/2) + n$$

$$a = 2,\ b = 2,\ k = 1 \rightarrow a = b^k$$

$$\rightarrow C(n) = O(n^k \log_b n) = O(n^1 \log_2 n) = O(n \log_2 n)$$

Chapitre 3

RÉCURSIVITÉ ET COMPLEXITÉ DES ALGORITHMES RÉCURSIFS