Algorithmique avancée

Cours n°1 : Tris, Fibonacci.

Licence 1 Informatique, Université Paris 8, 2022-2023

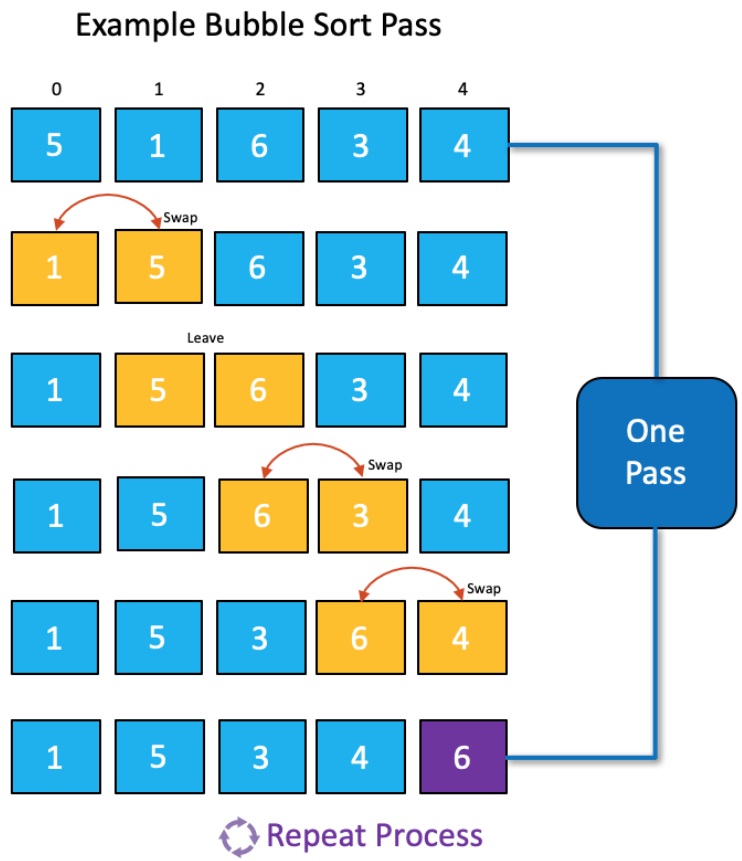
25 Septembre 2023

Organisation et évaluation

- 3h de cours puis 1h30 de TP en salle machine **A160** les lundis de 15h à 19h30.
- **Objectif du cours :** introduire et estimer la notion de complexité par l'exemple. Divers problèmes seront étudiés, et pour chacun plusieurs méthodes de résolution seront proposées et comparées.
  1. Suite de Fibonacci et coefficients binomiaux.
  2. Graphes et plus courts chemins.
  3. Droites discrètes.
  4. Algorithmes de compression et décompression d'images.
  5. Algorithmes d'IA pour les jeux.
- Note basée sur un projet individuel (projet, rapport, soutenance) dont les sujets vous seront proposés vers la Semaine 6.
- **Prérequis :** Savoir programmer en C (au cas où, reprendre le cours de *Programmation impérative* du L1,S1).

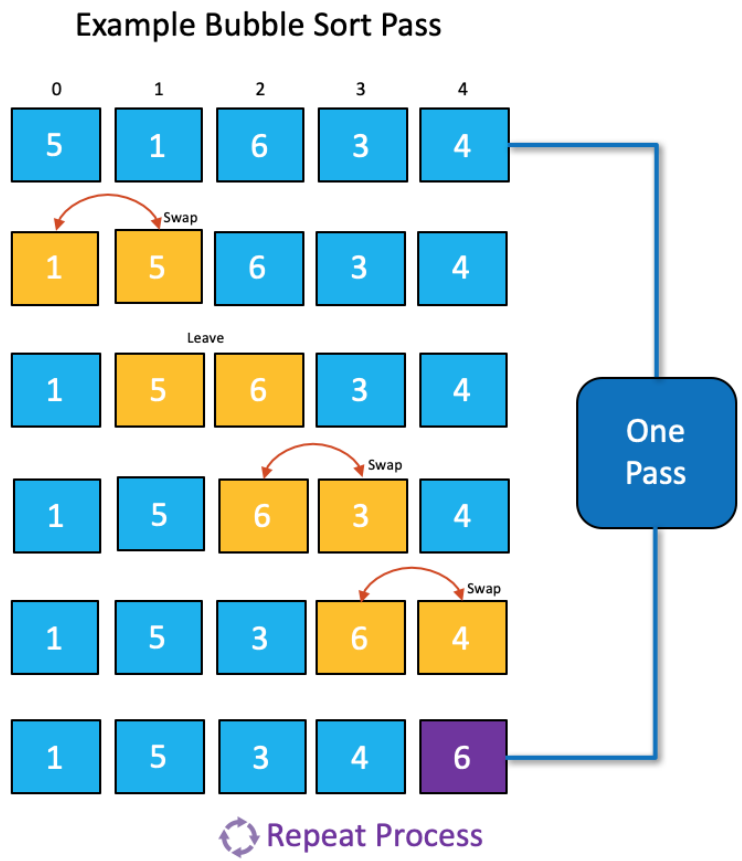
Algorithmes de tri : le tri à bulles

- Le **tri à bulles** est un algorithme de tri parcourant une liste en comparant les éléments deux à deux, et en permutant $L[i]$ et $L[i+1]$ si $L[i] > L[i+1]$, etc. jusqu'à ce que la liste soit triée.



Algorithmes de tri : le tri à bulles

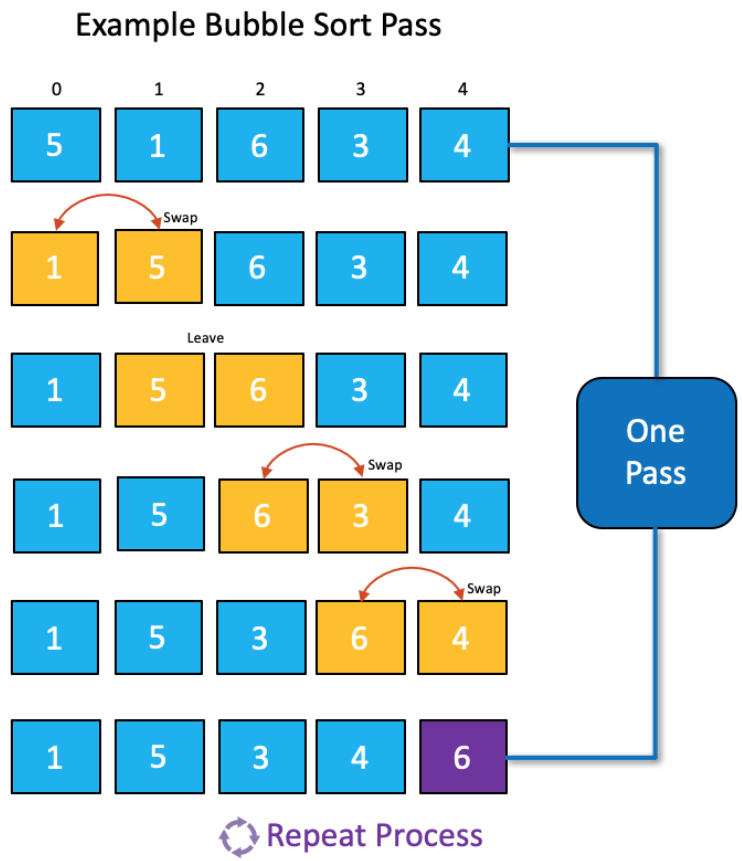
- Le **tri à bulles** est un algorithme de tri parcourant une liste en comparant les éléments deux à deux, et en permutant $L[i]$ et $L[i+1]$ si $L[i] > L[i+1]$, etc. jusqu'à ce que la liste soit triée.



- Complexité : $O(n^2)$ (**quadratique**).

Algorithmes de tri : le tri à bulles

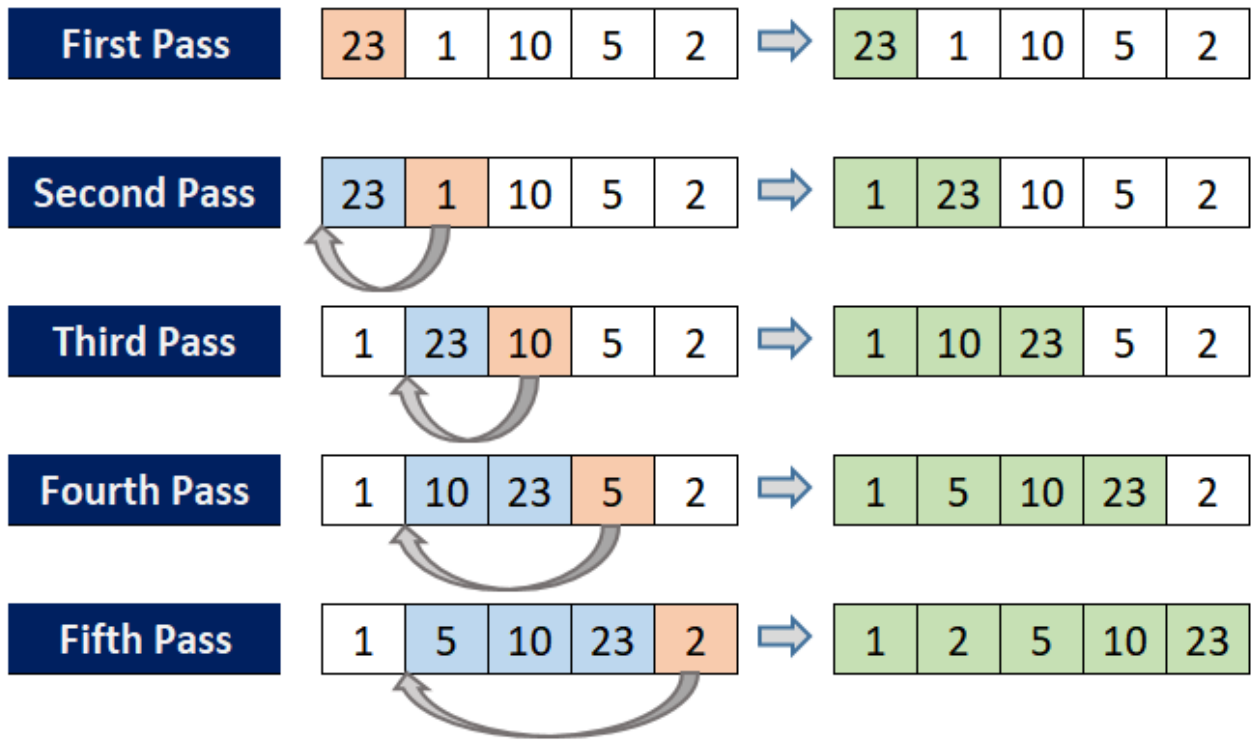
- Le **tri à bulles** est un algorithme de tri parcourant une liste en comparant les éléments deux à deux, et en permutant $L[i]$ et $L[i+1]$ si $L[i] > L[i+1]$, etc. jusqu'à ce que la liste soit triée.



- Complexité : $O(n^2)$ (**quadratique**).
- Meilleur des cas (liste triée): $n-1$ comparaisons; Pire des cas (liste inversement triée): $\frac{n(n-1)}{2}$ comparaisons.

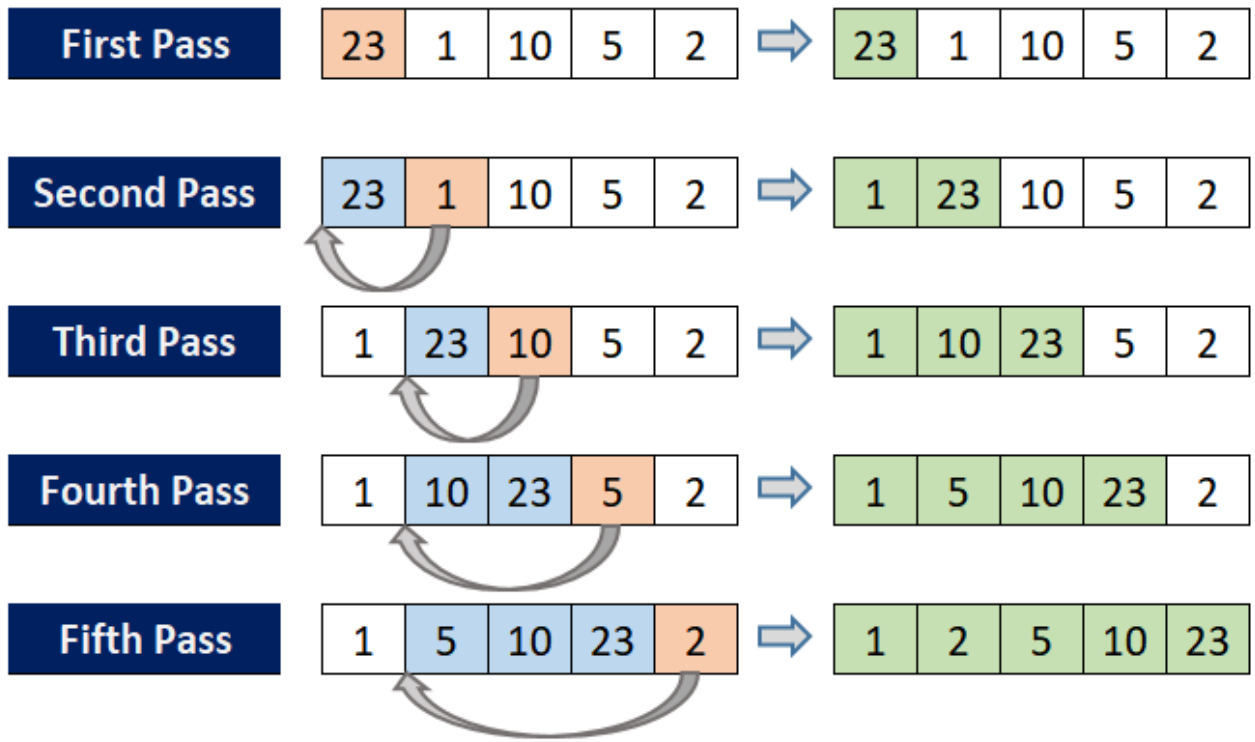
Algorithmes de tri : le tri par insertion

- Le **tri par insertion** est un tri consistant à parcourir une liste, et à insérer les éléments parcourus un par un « à la bonne place ».



Algorithmes de tri : le tri par insertion

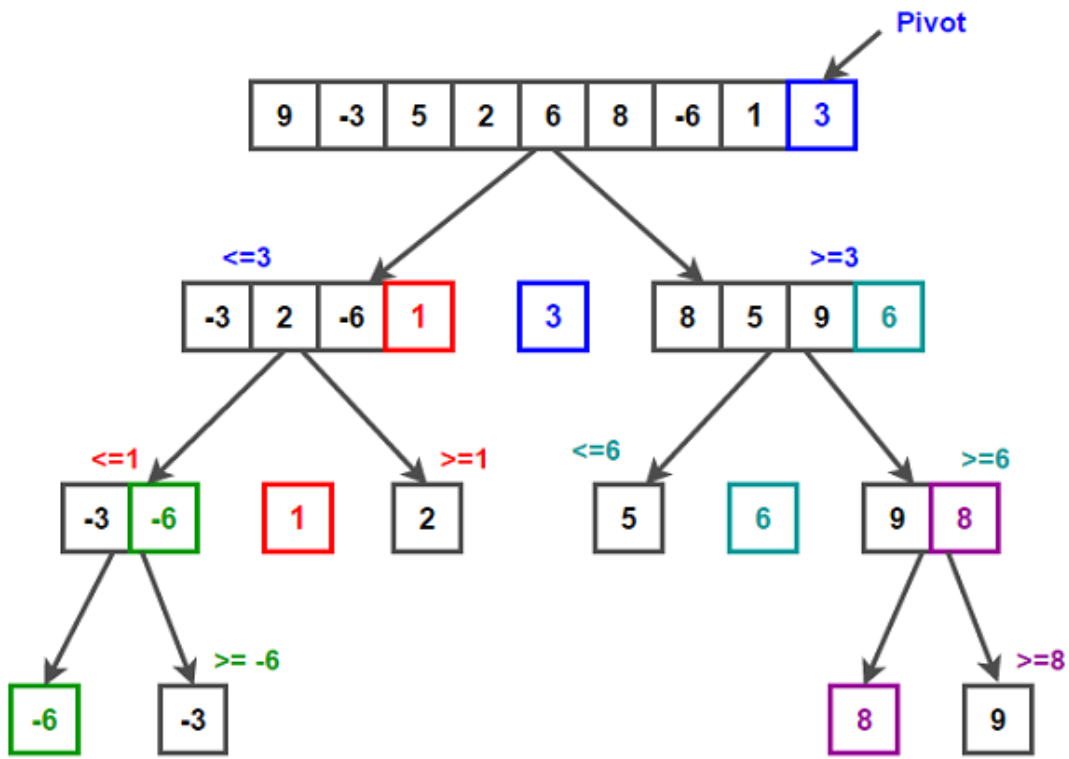
- Le **tri par insertion** est un tri consistant à parcourir une liste, et à insérer les éléments parcourus un par un « à la bonne place ».



- Complexité: $O(n^2)$ (**quadratique**).
- Meilleur des cas (liste triée) : n comparaisons; Pire des cas (liste inversement triée): $\frac{n(n-1)}{2}$ comparaisons.

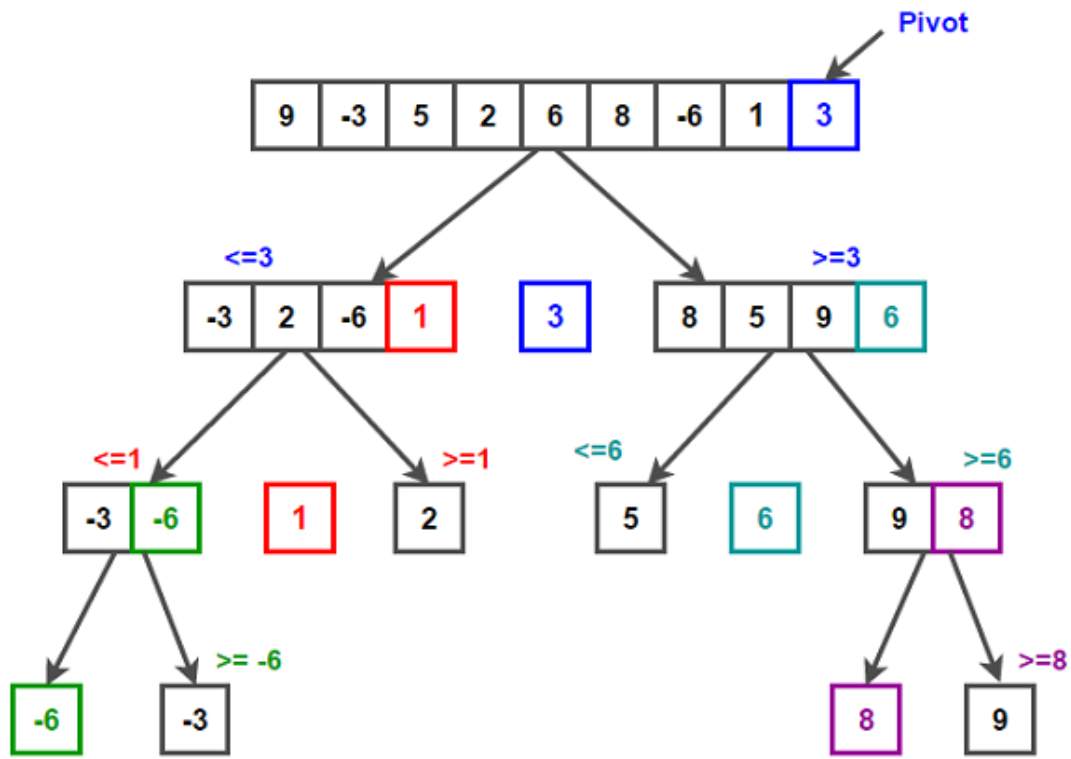
Algorithmes de tri : le tri rapide

- Le **tri rapide** (ou **quick sort**) est un algorithme utilisant le paradigme « Diviser pour régner » dont le principe est d'ordonner une liste en cherchant dans celui-ci un élément « pivot » autour duquel réorganiser les éléments selon les valeurs inférieures et supérieures.



Algorithmes de tri : le tri rapide

- Le **tri rapide** (ou **quick sort**) est un algorithme utilisant le paradigme « Diviser pour régner » dont le principe est d'ordonner une liste en cherchant dans celui-ci un élément « pivot » autour duquel réorganiser les éléments selon les valeurs inférieures et supérieures.



- Complexité en moyenne : $O(n\log(n))$.
- Cependant, cela dépend fortement du choix du pivot. Dans le pire des cas: $O(n^2)$.

Algorithmes de tri : le tri fusion

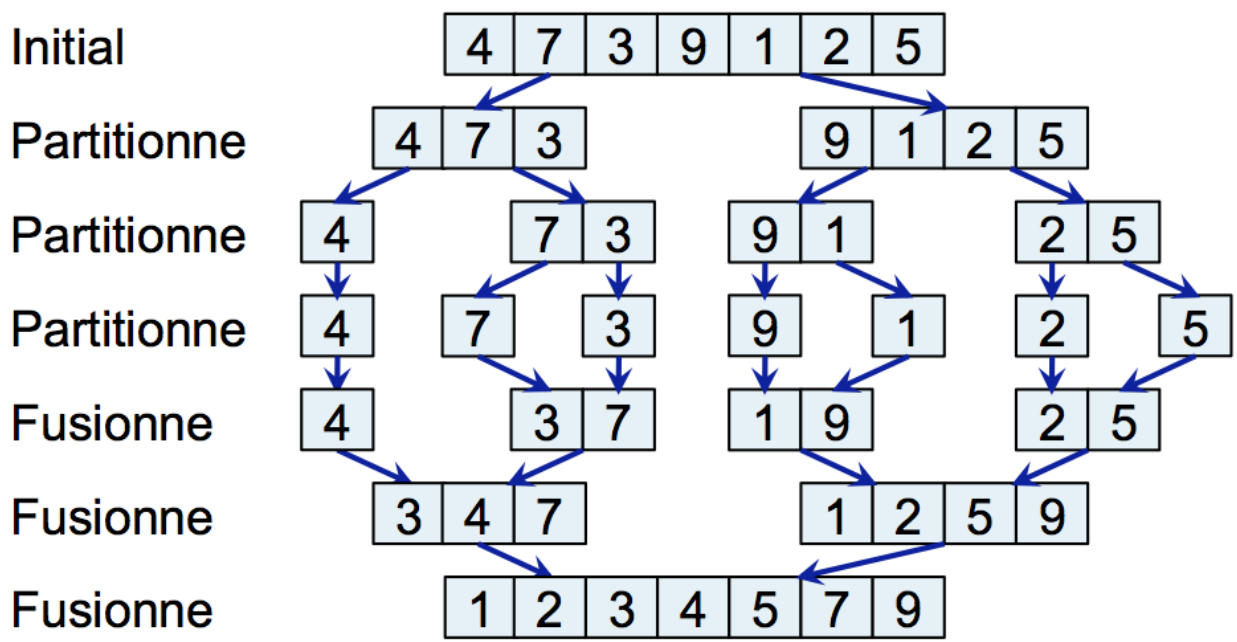
- Le **tri fusion** est un algorithme de tri utilisant le paradigme du « Diviser pour régner », consistant à diviser la liste en deux sous-listes de taille équivalente puis effectuer un appel-récursif sur les deux sous-listes, et fusionner les résultats.

Algorithmes de tri : le tri fusion

- Le **tri fusion** est un algorithme de tri utilisant le paradigme du « Diviser pour régner », consistant à diviser la liste en deux sous-listes de taille équivalente puis effectuer un appel-récursif sur les deux sous-listes, et fusionner les résultats.

- Complexité: $O(n\log(n))$.

- Dans le pire des cas, elle reste en $O(n\log(n))$.



Test des algorithmes de tri

- Pour des listes de taille respectives 100, 1000 et 10000 constituées d'entiers entre 0 et 100 générés aléatoirement :

```
Tri bulle       taille 100      temps 28
Insertion       taille 100      temps 8
TFusion         taille 100      temps 8
Quicksort       taille 100      temps 7
```

```
Tri bulle       taille 1000     temps 2754
Insertion       taille 1000     temps 698
TFusion         taille 1000     temps 116
Quicksort       taille 1000     temps 84
```

```
Tri bulle       taille 10000    temps 258315
Insertion       taille 10000    temps 51143
TFusion         taille 10000    temps 1140
Quicksort       taille 10000    temps 753
```

Test des algorithmes de tri

- Pour des listes de taille respectives 100, 1000 et 10000 constituées d'entiers entre 0 et 100 générés aléatoirement :

```
Tri bulle       taille 100      temps 28
Insertion       taille 100      temps 8
TFusion         taille 100      temps 8
Quicksort       taille 100      temps 7
```

```
Tri bulle       taille 1000     temps 2754
Insertion       taille 1000     temps 698
TFusion         taille 1000     temps 116
Quicksort       taille 1000     temps 84
```

```
Tri bulle       taille 10000    temps 258315
Insertion       taille 10000    temps 51143
TFusion         taille 10000    temps 1140
Quicksort       taille 10000    temps 753
```

- Reprenons maintenant ce test pour le choix d'une liste de taille 1000, mais déjà triée:

```
Tri bulle       taille 1000     temps 3
Insertion       taille 1000     temps 3
TFusion         taille 1000     temps 49
Quicksort       taille 1000     temps 1025
```

- Pour une liste de taille 1000 triée dans l'ordre décroissant:

```
Tri bulle       taille 1000     temps 2681
Insertion       taille 1000     temps 1134
TFusion         taille 1000     temps 51
Quicksort       taille 1000     temps 1012
```

Suite de Fibonacci

- On rappelle que la suite de Fibonacci est définie par

$$F(0) = 1, F(1) = 1 \text{ et pour } n \geq 2,\ F(n) = F(n-1) + F(n-2).$$

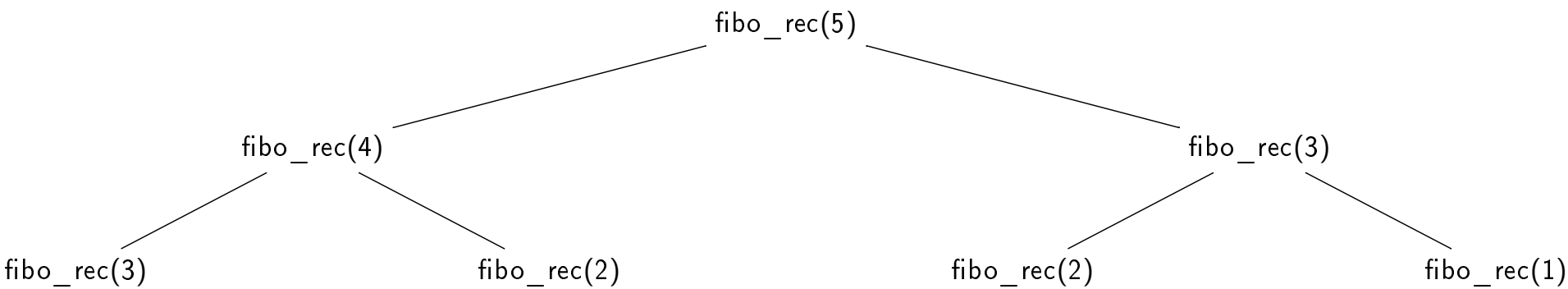
- Dans ce cours, nous allons voir plusieurs algorithmes différentes permettant de calculer le nombre $F(n)$.

- **Première méthode : Récursivité lourde**

```
int fibo_rec(int n){
    int res=0;
    if (n==0 || n==1){
        return 1;
    }
    else{
        res=fibo_rec(n-1) + fibo_rec(n-2);
        return res;
    }
}
```

Suite de Fibonacci : fonction récursive lourde

- Évaluons l'efficacité de cet algorithme. Par exemple, pour calculer **fibo_rec(5)**, voici les calculs réalisés:



- Pour aller de l'ordre jusqu'aux feuilles, on effectue de l'ordre de 2^n appels récursifs. On a donc une complexité **exponentielle**. En pratique, on n'utilisera cet algorithme pour afficher des valeurs de $F(n)$ que pour $n \leq 40$.

Suite de Fibonacci : fonction itérative

- Voici une fonction itérative, où on stocke dans a et b les valeurs initiales, puis ensuite on attribue à une variable c la somme $a + b$, et le nouveau b devient a, le nouveau a devient c.

```
int fibo_ite(int n){
    int a=1;
    int b=1;
    int c=1;
    for (int i = 1; i < n; i++) {
        c = a + b;
        a= b;
        b = c;
    }
    return c;
}
```

Suite de Fibonacci : fonction itérative

- Voici une fonction itérative, où on stocke dans a et b les valeurs initiales, puis ensuite on attribue à une variable c la somme $a + b$, et le nouveau b devient a, le nouveau a devient c.

```
int fibo_ite(int n){
    int a=1;
    int b=1;
    int c=1;
    for (int i = 1; i < n; i++) {
        c = a + b;
        a= b;
        b = c;
    }
    return c;
}
```

- Complexité : $O(n)$.

Suite de Fibonacci : fonction vectorielle

- Voici une fonction vectorielle, où on crée un tableau de taille n, que l'on remplit au fur et à mesure avec une boucle *for* en additionnant les deux valeurs précédentes.

```
int fibo_vect(int n){
    int *f = malloc((n+1)*sizeof(int));
    f[0] = 1;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
       {
            f[i] = f[i-1] + f[i-2];
       }
    return f[n];
}
```

- Complexité : $O(n)$.

Suite de Fibonacci : récursivité terminale

- Voici une fonction avec une récursivité terminale :

```
int fibo_rect_aux(int n, int a, int b){
  if (n==1){
    return a;
  }
  else return fibo_rect_aux(n-1,a+b,a);
}

int fibo_rect(int n){
  return fibo_rect_aux(n,1,1);
}
```

- Complexité : $O(n)$.

Suite de Fibonacci : fonction matricielle

- Cette fonction utilise la remarque suivante :

$$\begin{cases} f_n = f_{n-1} + f_{n-2} \\ f_{n-1} = f_{n-1} \end{cases} \quad \Leftrightarrow \quad \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix}$$

Suite de Fibonacci : fonction matricielle

- Cette fonction utilise la remarque suivante :

$$\begin{cases} f_n = f_{n-1} + f_{n-2} \\ f_{n-1} = f_{n-1} \end{cases} \quad \Leftrightarrow \quad \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix}$$

d'où l'on obtient par récurrence directe que

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}$$

et donc il suffit de calculer les puissances de la matrice

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

par exemple par **exponentiation rapide**.

Suite de Fibonacci : fonction matricielle

- Exponentiation rapide de matrices 2×2:

```
void puissance(int mat[2][2], int res[2][2], int n){
  int b[2][2], aux[2][2];
  if (n==1){
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
        res[i][j] = mat[i][j];
      }
    }
  }
  else if (n%2 ==0){
    multiplyMatrices(mat,mat,b);
    puissance(b,res,n/2);
  }
  else {
    multiplyMatrices(mat,mat,b);
    puissance(b,res,n/2);
    multiplyMatrices(mat,res,aux);
    copy(aux,res);
  }
}
```

- Complexité: $O(\log(n))$, donc a priori meilleur que tous les algos vus jusqu'ici.

Suite de Fibonacci : fonction logarithmique

- **Meilleure implémentation de la méthode matricielle :** on crée une structure de **paire** constituée de deux *long long int*.

```
paire fiblog (int n) {
    paire mi, res;
    ullint fi;
    int i;

    if (n < 2){
        res.fun = (ullint) 1;
        res.fdeux = (ullint) 1;
        return res;
    }
    i = n >> 1;
    mi = fiblog(i);
    if (n & 0x01) {
        res.fdeux = mi.fun * mi.fun + mi.fdeux * mi.fdeux;
        res.fun = (mi.fun + mi.fdeux + mi.fdeux) * mi.fun;
        return res;
    }
    res.fun = mi.fun * mi.fun + mi.fdeux * mi.fdeux;
    res.fdeux = (mi.fun + mi.fun - mi.fdeux) * mi.fdeux;
    return res;
}
```

- Complexité: $O(\log(n))$.

Suite de Fibonacci : mémorisation dans un fichier

- Voici une fonction dans lequel on écrit dans un fichier *fibo.txt* les valeurs déjà calculées, puis on effectue le calcul de Fibonacci si et seulement si la valeur n'est pas déjà calculée.

- Une fonction **import**, qui lire les valeurs déjà calculées, jusqu'au plus petit entre l'indice jusqu'au quel on a déjà calculé dans le fichier et le *n* en paramètre :

```
int import(int* tab, int n){
    FILE *fichier = fopen("fibo.txt","r");
    int max;
    if (fichier != NULL){
        fscanf(fichier,"%d",&max);
        int l = minimum(max,n);
        for(int i=0; i<l; i++){
            fscanf(fichier,"%d",&(tab[i]));
        }
        fclose(fichier);
    }
    return max;
}
```

Suite de Fibonacci : mémorisation dans un fichier 2

- Une fonction **export**, qui écrit dans le fichier en ajoutant les valeurs des `tab[i]` pour i de 0 à s inclus :

```
void export(int *tab,int s){
    FILE *fichier= fopen("fibo.txt","w");
    if (fichier != NULL){
        fprintf(fichier,"%d \n",s);
        for (int i=0; i<= s; i++){
            fprintf(fichier,"%d \n",tab[i]);
        }
        fclose(fichier);
    }
}
```

- La fonction **fib_aux**, qui calcule les coefficients de Fibonacci lorsque nécessaire :

```
int fib_aux(int *tab, int n){
    if (tab[n]!=0)
        return tab[n];
    else{
        int a=fib_aux(tab,n-1);
        int b=fib_aux(tab,n-2);
        tab[n]=a+b;
        return tab[n];
    }
}
```

Suite de Fibonacci : mémorisation dans un fichier 3

- La fonction **fib_memo**, qui fait le travail final :

```
int fib_incr(int n){
    int *tab = malloc((n+1)*sizeof(int));
    int max=import(tab,n+1);
    if (max >= n)
        return tab[n];
    else{
        int res=fib_aux(tab,n);
        export(tab,n);
        return res;
    }
}
```

- Si `max` renvoyé par import est plus grand que le n, on a déjà calculé la valeur et il suffit de renvoyer l'élément `tab[n]` qui correspond à $F(n)$. Sinon, il faut continuer à écrire dans le fichier les éléments du tableau non calculés, en appelant la fonction **fib_aux** pour calculer les termes manquants.