

Pimp A - info121A

Programmation IMPérative Avancée

*Burkhart Wolff
Frédéric Vernier*

1

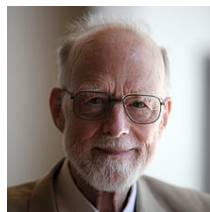
Programme

- *Tri*
- *Récurtivité*
- *Dichotomie*
- *Arbre*
- *Graphe*
- *Examen blanc*

2

Rappels 1/3

- *Tri $O(n^2)$*
 - *Sélection, insertion, tri à bulle*
- *Tri $O(n \cdot \log(n))$*
 - *tri par tas*
 - *tri rapide (Quick sort)*
 - *Tony Hoare*



3

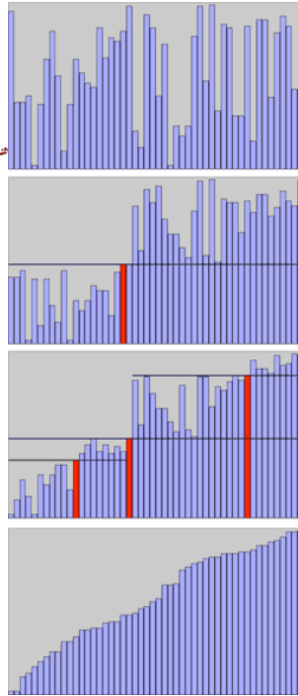
Rappels 2/3

- *Point de vue*
 - *suite d'éléments à trier*
 - *arbre binaire d'éléments*
- *Stratégies*
 - *Elitisme*
 - *“Come as you are”*

4

Quick sort

- Nouvelle stratégie “égocentrique”
 - choix d'un pivot moyen
 - partitionnement
 - **et on recommence 2 fois**
 - $2 \Rightarrow 4 \Rightarrow 8 \Rightarrow \dots$
 - tailles quelconques



5

Récursivité

- La boucle(for ou while) marche très mal
 - pourtant facile pour la logique humaine
- Nouvelle structure algorithmique
 - **La récursivité !**

6

Fonctions

- Définition / Appels
- Empilement de contextes

```
void tri(int n){
    int meilleur;
    ...;
}
void chercheMediane(int n){
    int mediane;
    tri(n);
}
void main(int argc, char*[] argv){
    tab[0]=...
    chercheMediane(n);
}
```

tri	n	meilleur
-----	---	----------

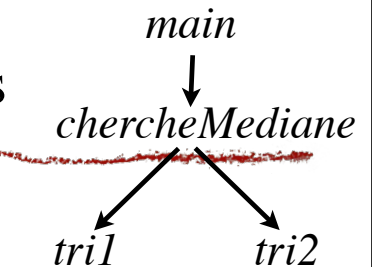
cherche Mediane	n	mediane
--------------------	---	---------

main	argc	argv
------	------	------

7

Fonctions

```
void tri1(int n){
    int meilleur;
}
void tri2(int n){
    int prochain;
}
void chercheMediane(int n){
    int mediane;
    tri1(n);
    tri2(n);
}
void main(int argc, char*[] argv){
    chercheMediane(n);
}
```



tri2	n	prochain
------	---	----------

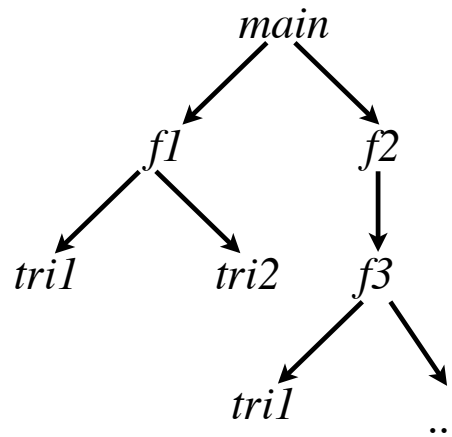
cherche Mediane	n	mediane
--------------------	---	---------

main	argc	argv
------	------	------

8

Pile d'appels

- *Parcours un arbre*
 - *profondeur d'abord*
 - *taille variable*



9

Récursivité

- *f appelle f*
 - *à l'infini (et au delà)*

```

void f(){
    f();
}
void main(int argc, char*[] argv){
    f();
}
  
```



10

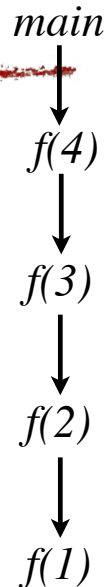
Stop !

- *Il faut que ça s'arrête !*

```

void f(int i){
    if (i>0)
        f(i-1);
}
void main(int argc, char*[] argv){
    f(4);
}
  
```

- *pas besoin de coder la limite*
 - *dans la fonction ou en param.*



11

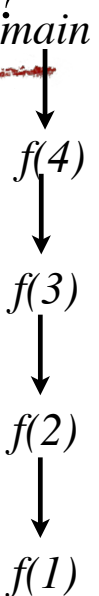
comment partager les données ?

- *passer les données en argument*

```

int T[];
void f(int tab[], int k){
    tab[k] = 0;
    if (k==0)
        f(tab, k-1);
}
void main(int argc, char*[] argv){
    f(T, 4);
}
  
```

f(tab, 0) ???
tab[k] = 0
donc i == 1 possible
donc appel de f(tab, 0)



12

Appel en chaine !

```
int T[];

void f(int tab[], int i){
    tab[i] = 0;
    if (i>0)
        f(tab, i-1);
}

void main(int argc, char*[] argv){
    f(T, 3);
}
```

0	0	0	0
---	---	---	---

- Comment aller dans l'autre sens ?

13

snese rtua'l snaD

*coder en dur la limite
dans la fonction récursive*

inversion

<pre>int T[] = {};</pre>		<pre>int T[] = {};</pre>
<pre>void f(int tab[], int i){ tab[i] = 0; if (i<3) f(tab, i+1); }</pre>		<pre>void f(int tab[], int i){ if (i>0) f(tab, i+1); tab[i] = 0; }</pre>
<pre>void main(int argc, ...){ f(T, 0); }</pre>		<pre>void main(int argc, ...){ f(T, 3); }</pre>

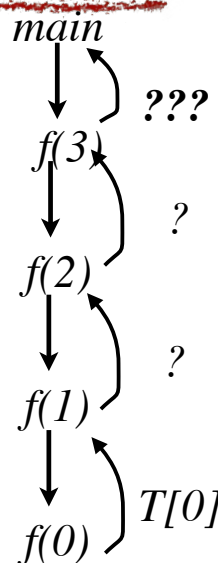
14

valeur de retour

- retourner = remonter dans l'arbre*

```
int f(int tab[], int i){
    if (i==0) return tab[i];
    else {
        int val = f(tab, i-1);
        if (tab[i]>val)
            return tab[i];
        else
            return val;
    }
}

void main(int argc, ...){
    cout << f(T, 3);
}
```



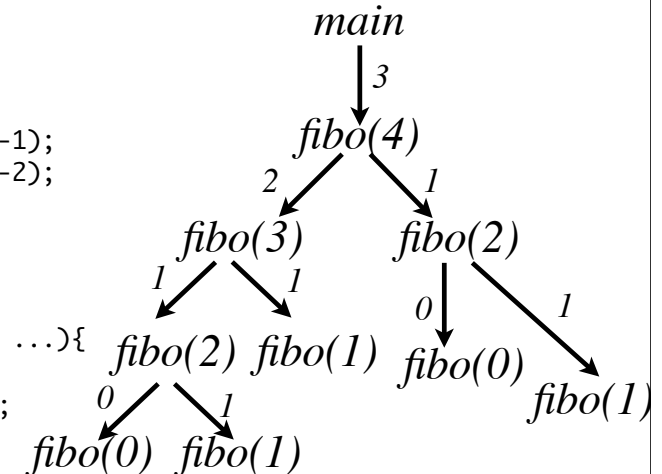
- Que fait ce code ?

15

double appel = arbre binaire

```
int fibo(int i){
    if (i==0 || i==1)
        return i;
    else {
        int v1 = fibo(i-1);
        int v2 = fibo(i-2);
        return v1+v2;
    }
}
```

```
void main(int argc, ...){
    int v = fibo(4);
    cout << v << endl;
}
```



16

+ un tableau de donnée

```
int T[] = {5, 0, 0, 9, 1, 2, 3, 0, 5, 7};
int f(int tab[], int i, int j){
    if (i==j && tab[i]==0)
        return 1;
    else if (i==j)
        return 0;
    else {
        int mid = (i+j)/2;
        return f(T, i, mid) + f(T, mid+1, j);
    }
}

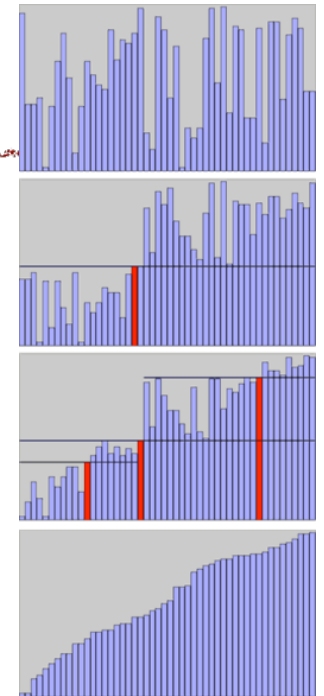
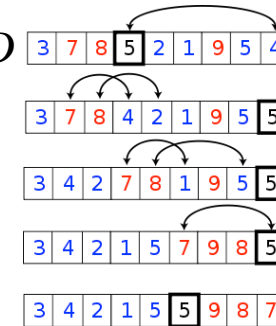
void main(int argc, ...){
    int v = f(T, 0, 9);
    cout << v << endl;
}
```

- *Que fait ce code ?*
- *Comment ?*

17

Quick Sort

- *Partitionnement*
 - “choix” d’un pivot
 - pivot -> fin
 - parcours GD



18

Implémentation

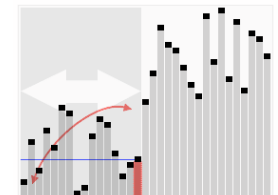
```
void partition (int debut, int fin) {

    int j=debut;
    for (int i=debut; i<fin; i++){
        if (diffButsLigue1_2010[i]<=diffButsLigue1_2010[fin]) {
            permute(i, j);
            j++;
        }
    }
}
```

19

Implémentation

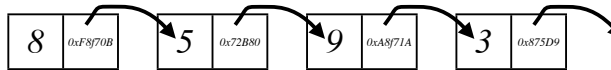
```
void tri_rapide (int debut, int fin) {
    if (debut<fin) {
        permute(debut, fin);
        int j=debut;
        for (int i=debut; i<fin; i++){
            if (diffButsLigue1_2010[i]<=diffButsLigue1_2010[fin]) {
                permute(i, j);
                j++;
            }
        }
        permute(j, fin);
        tri_rapide(debut, j-1);
        tri_rapide(j+1, fin);
    }
}
```



20

Tri fusion

- *Tri sur place*
 - *lent*
- *Tri avec mémoire supplémentaire*
 - *rapide*
- *Tri sur liste*
- *Récuratif*



21

Stratégie

- *Diviser pour régner*
- *Divide and conquer*
- *Divide ut imperes*



“Pour comprendre le principe de récursivité, il faut d'abord comprendre le principe de récursivité”

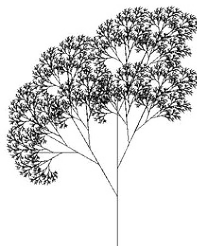
22

Récursivité

- *N'oubliez pas le cas de base !*
- *1 ou plusieurs appels récursifs ?*
- *Travail + appel récursif ou vide-versa*
- *Valeur de retour “propagée” BH*
- *N'oubliez pas le cas de base ! (2x)*

1	recursiveFunction (0)
2	printf (0)
3	recursiveFunction (0+1)
4	printf (1)
5	recursiveFunction (1+1)
6	printf (2)
7	recursiveFunction (2+1)
8	printf (3)
9	recursiveFunction (3+1)
10	printf (4)

1	recursiveFunction (0)
2	recursiveFunction (0+1)
3	recursiveFunction (1+1)
4	recursiveFunction (2+1)
5	recursiveFunction (3+1)
6	recursiveFunction (4)
7	printf (3)
8	printf (2)
9	printf (1)
10	printf (0)



23

Dessins récursifs

- *Nature*
 - *Broccoli, arbres, etc.*
- *Fractals*

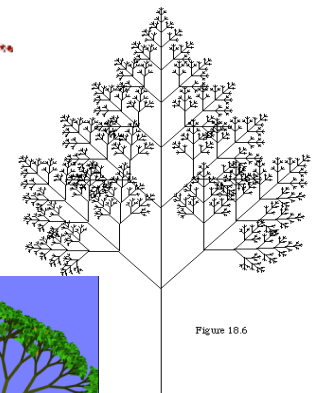
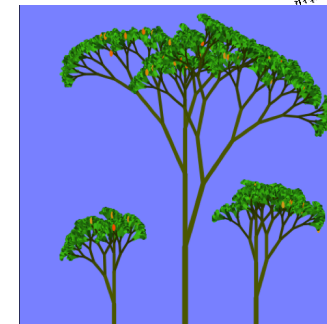
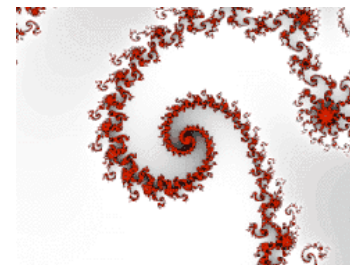
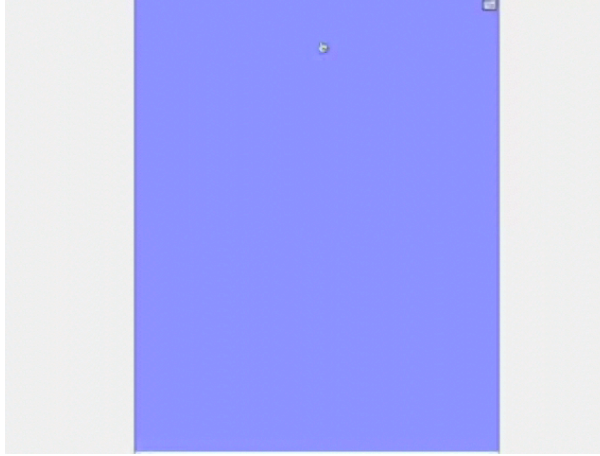


Figure 18.6

24

Nature

- *Hasard*
 - *Perlin / blanc*
- *Temps*
 - *Itératif*
 - *Paramètres*
 - *Récursion*



25

Conclusion 1/2

- *Récursion*
 - *Itération avec une pile*
 - *moins performant (lang. impératifs)*
- *Algorithmes hybrides*
 - *Récursion sur grand ensemble*
 - *cas de base = itération sur petit ensemble*

26

Conclusion 2/2

- *Récurtivité*
 - *Nouvelle approche de l'informatique*
 - *Invariant / cas de base*
 - *Pile / Arbre*
 - *... programmation fonctionnelle*
- *Mise en pratique*
 - *Pratiquez !*

27