Algo Design 1 : Complexity notions

Pierre-Alain Fouque

Pierre-Alain.Fouque@univ-rennes1.fr

Algorithm : Motivation

- Different Approach of programming
  - Definition of the problem to solve
  - Searching for an algorithm
  - Complexity analysis
  - Implementation

Algorithm : definition

- **Algorithm** – Definition :
- Finite sequence of well-defined, computer-implementable instructions, to solve a class of problems or to perform a computation

(Wikipedia)

Algorithm : complexity

- Goal : mesure the inherent efficiency of an algorithm
  - As a function of the input size
  - Compute the important elementary operations
  - Asymptotic Measure
  - Worst / Average -case complexity
  - Time / Space complexity

Classical example: sorting

- Goal : sort an array of n integers
- Compute the number of comparisons
- Elementary algorithms :

  « order of » n^2 *comparisons*
- Advanced algorithms: $n^2 \Rightarrow n.log(n)$

Complexity Notations

- $f(n)=O(g(n))$ iff $0 \le f(n) \le c.g(n)$
- $f(n)=\Theta(g(n))$ iff $c.g(n) \le f(n) \le c'.g(n)$
- Examples :
  - $n^2+3n+1 = \Theta(n^2) = \Theta(50\ n^2+12345)$
  - $n/\ln(n) = O(n)$
  - $50\ n^{10} = O(n^{10,01})$
  - $2^n = O(\exp(n))$
  - $\exp(n) = O(n!)$

Hierarchy of functions

- One can establish a **hierarchy** between functions :

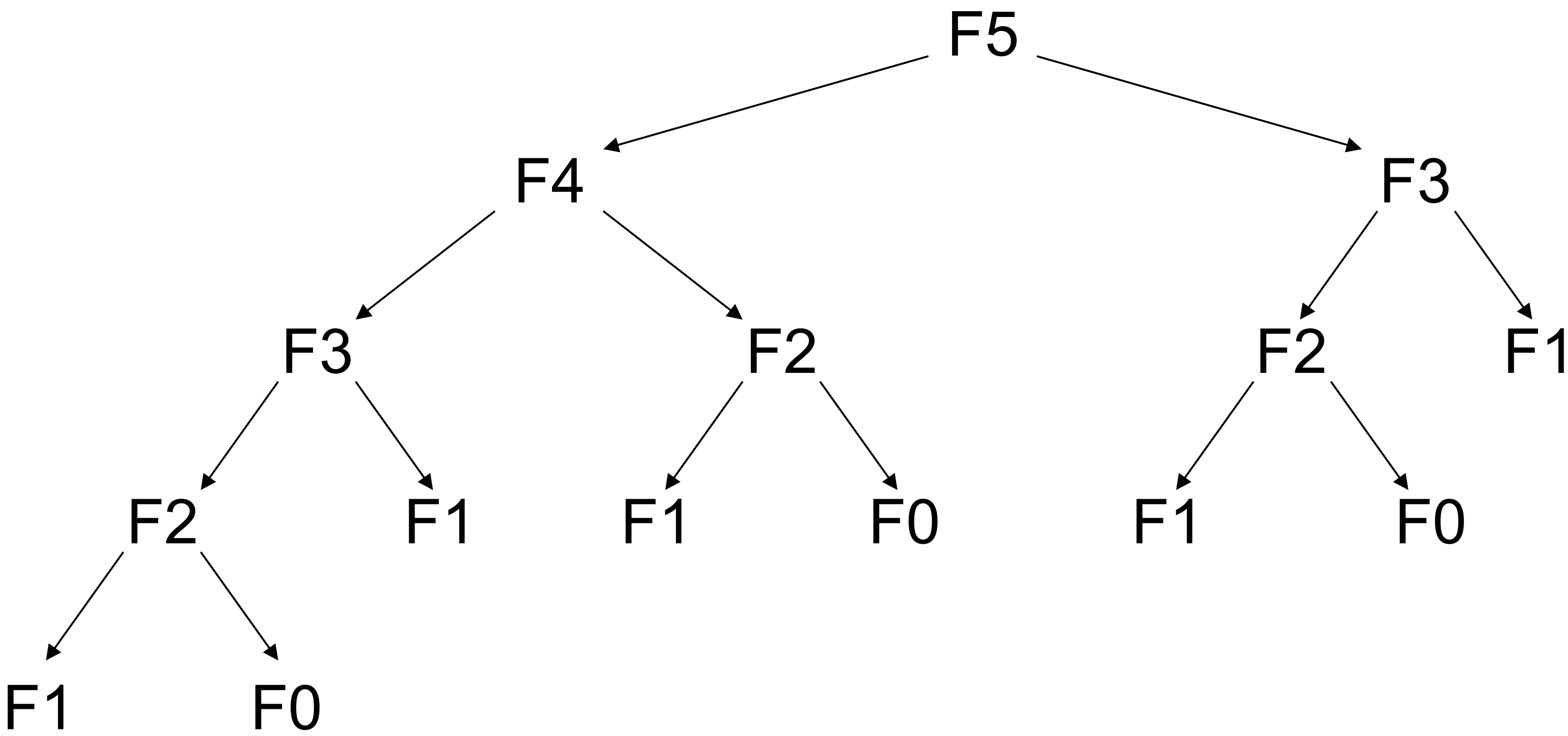
$$\log(n) << \sqrt{n} << n << n^2 << n^3 << 2^n << \exp(n) << n!$$

$\log(n)$	3.3	6.6	10
$\sqrt{n}$	3.1	10	32
n	10	100	1000
$n\log(n)$	33	664	10^4
n^2	100	10^4	10^6
n^3	10^3	10^6	10^9
2^n	10^3	10^{30}	10^{300}
$\exp(n)$	2×10^4	10^{43}	10^{434}
$n!$	3.6×10^6	10^{158}	10^{2568}

Fibonacci sequence

- $F_n = F_{n-1} + F_{n-2}$ if $n > 1$
- $F_0 = F_1 = 1$
- 1, 1, 2, 3, 5, 8, 13, 21, 34,…
  - ```
    int fibo1(int n)
    {
        if (n<=1) return 1;
        else return fibo1(n-1)+fibo1(n-2);
    }
    ```
- <u>Complexity</u> : $\Theta(\omega^n)$ where $\omega = (1+\sqrt{5})/2$
    ```

Fibonacci sequence



- Complexity : $\Theta(\omega^n)$ where $\omega = (1+\sqrt{5})/2$

Fibonacci sequence (2)

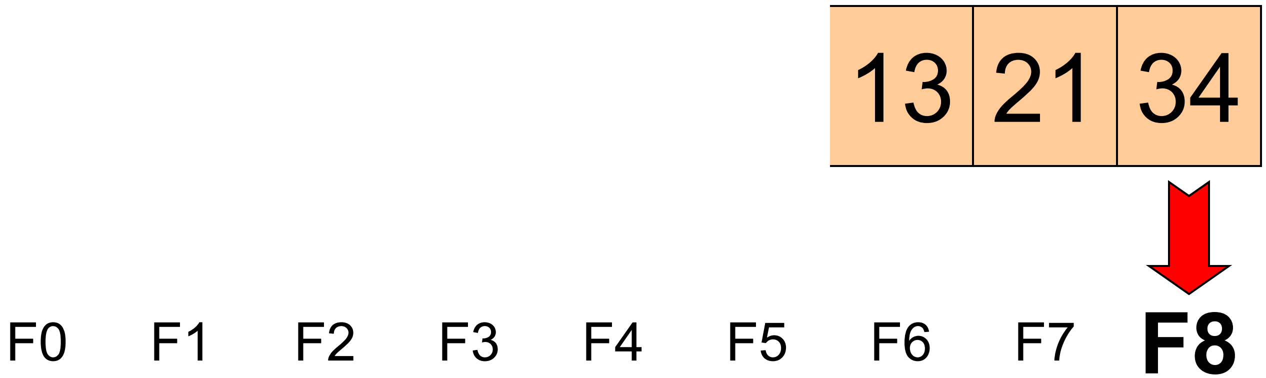
F0	F1	F2	F3	F4	F5	F6	F7	**F8**
1	1	2	3	5	8	13	21	34

- Complexity : $\Theta(n)$
- Space complexity : $\Theta(n)$

Fibonacci sequence (2)

- ```
  int fibo2(int n)
  { int *t;
    int i,res;
    t=(int *)malloc((n+1)*sizeof(int));
    t[0]=1; t[1]=1;
    for(i=2;i<=n;i++) t[i]=t[i-1]+t[i-2];
    res=t[n];
    free(t);
    return res;}
  ```
- Complexity : $\Theta(n)$
- Space Complexity: $\Theta(n)$
  ```

Fibonacci sequence (3)



- <u>Complexity</u> : Θ(n)
- <u>Space Complexity</u> : Θ(1) (constant)

Fibonacci sequence (3)

- ```
  int fibo3(int n)
  { int f1,f2,t,i;
    f1=1; f2=1;
    for(i=2;i<=n;i++)
      { t=f2;
        f2=f1+f2;
        f1=t;}
    return f2;}
  ```
- Complexity : Θ(n)
- Space Complexity : Θ(1) (constant)
  ```

Fibonacci sequence (4)

- $F_n \quad = 1 \times F_{n-1} + 1 \times F_{n-2}$
  $F_{n-1} = 1 \times F_{n-1} + 0 \times F_{n-2}$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \times \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

- We get back to compute a « matrix exponentiation»
- Complexity : Θ(log(n))
- Space Complexity : Θ(1) (constant)

Matching theory / practice

n	40	5.10^7	2.10^8	2.10^9
Fibo1(n)	31 s	Too large computation		
Fibo2(n)	0 s	18 s	Segmentation fault	
Fibo3(n)	0 s	4 s	19 s	3 min 15
Fibo4(n)	0 s	0 s	0 s	0 s

Classical Example : sorting

- Goal : sort an array of ***n*** integers
- Compute the number of comparisons
- Elementary Algorithms: $O(n^2)$
- Advanced Algorithms: $O(n.log(n))$

Insertion Sorting

- Sort an array A of n integers :
-
```
for i=2 to n do
    key=A[i]
    j=i-1
    while j>0 et A[j]>key do
        A[j+1]=A[j]
        j=j-1
    A[j+1]=key
```

Insertion Sorting (2)

```
• void InsertionSort(int *A, int n)
  {
    int i,j,key;
    for(i=1;i<n;i++)
      {
        key=A[i];
        j=i-1;
        while ((j>=0) && (A[j]>key))
          { A[j+1]=A[j];
            j=j-1;}
        A[j+1]=key;
      }
  }
```

Insertion Sorting (3)

- Different style … to be avoided !!!
- `Sort_ugly(int *A, int n){`

```
int i=1,j=0,key=*(A+1);
for(;i<n;A[j+1]=key,j=i++,key=A[i])
while ((j>=0) && (A[j]>key))
A[j+1]=A[j--];}
```

Insertion Sorting (4)

- Worst-case Complexity :

  table sorted upside down

$$C_n = \Theta(n^2)$$

- Average-case complexity :

$$C_n = \Theta(n^2)$$

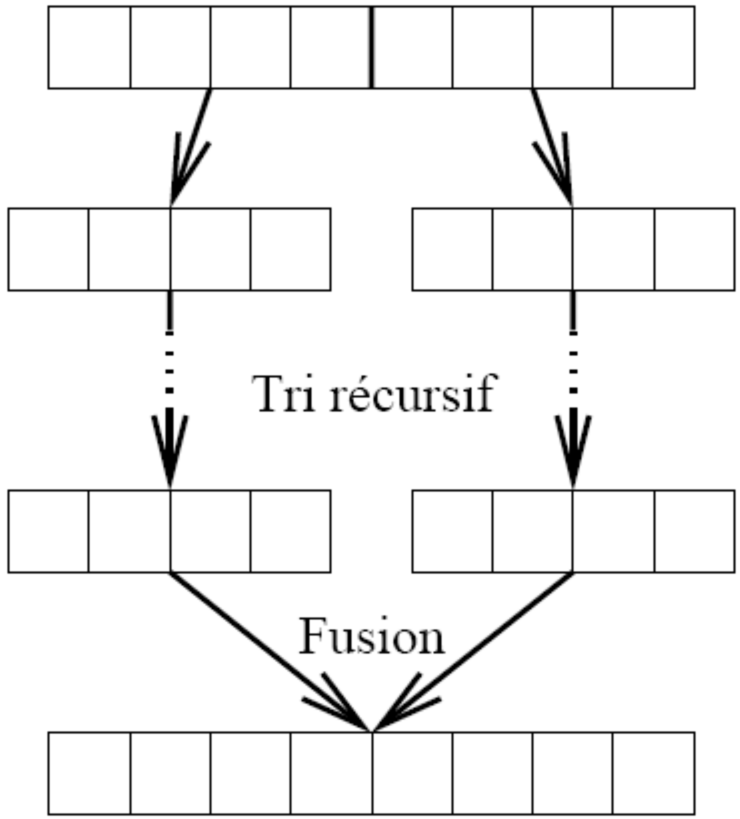
Merge Sort

- Sort an array A between the indices p & r :
- `if p<r then`

```
        q=(p+r)/2
        mergesort(A,p,q)
        mergesort(A,q+1,r)
        merge(A,p,q,r)
```

- Complexity…

Merge Sort



Quicksort

- Recursive sort based on partitionning
- `if p<r then`

```
        q=partition(A,p,r)
        quicksort(A,p,q-1)
        quicksort(A,q+1,r)
```

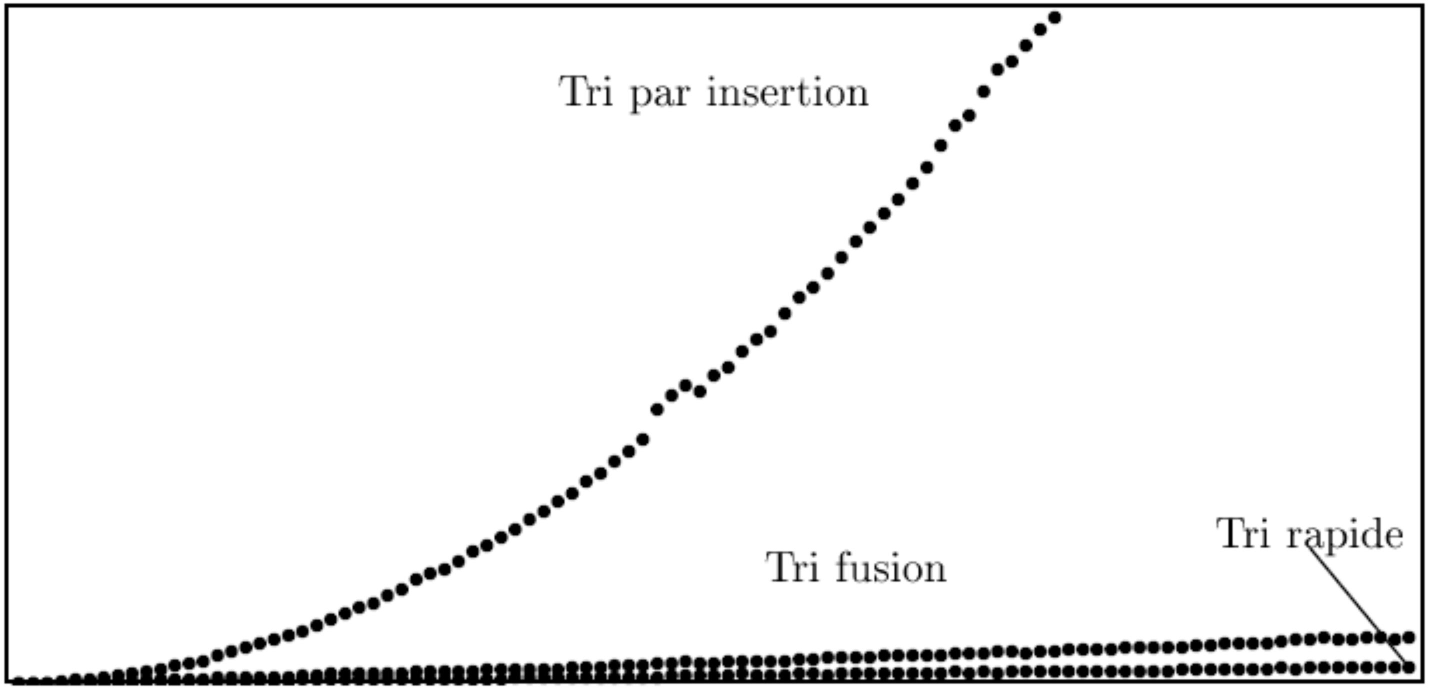
- Worst-case complexity : $C_n = \Theta(n^2)$
- Average-case Complexity : $C_n = \Theta(n.\log(n))$

Sort Problem

Algorithm	Worst-case	Average
Insertion Buble Sort	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n^2)$	$\Theta(n.\log(n))$
Merge Sort	$\Theta(n.\log(n))$	$\Theta(n.\log(n))$

Matching theory / practice

Computation Time

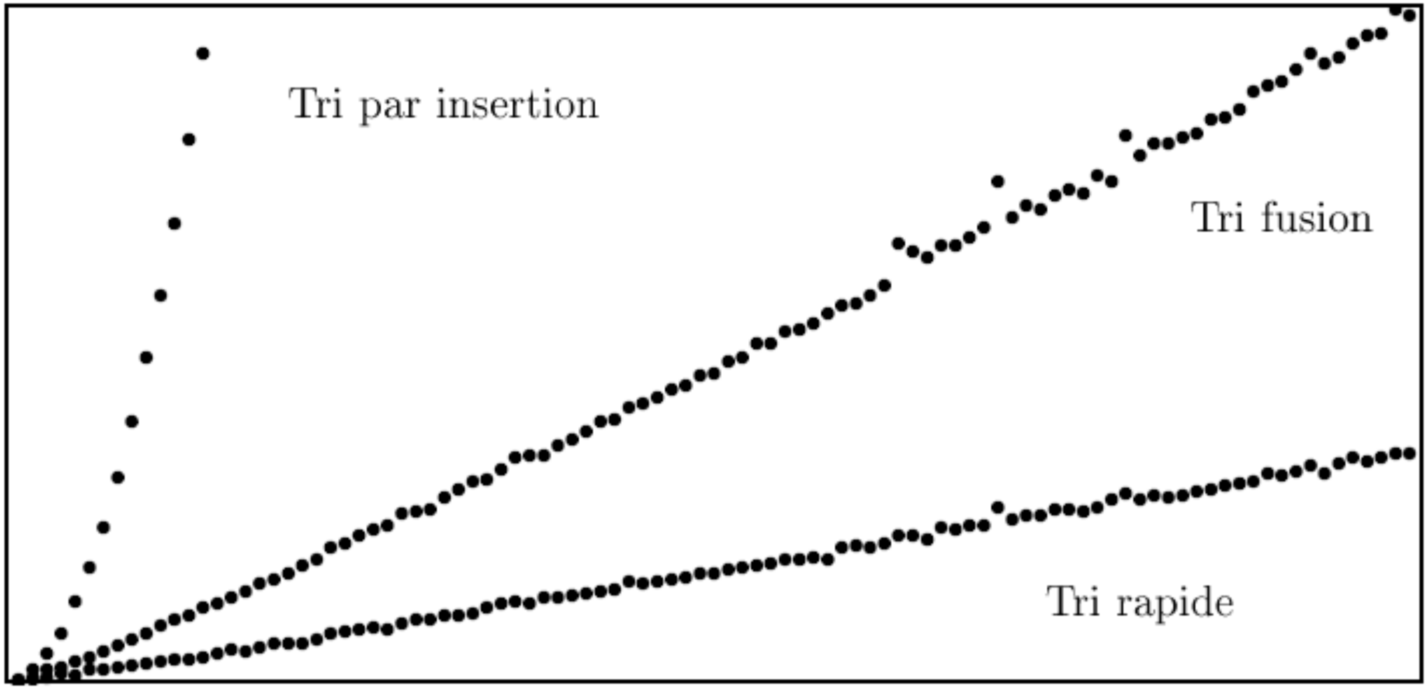


n=5000

FIG. 1.1 – *Complexité moyenne expérimentale*

Matching theory / practice

Computation Time

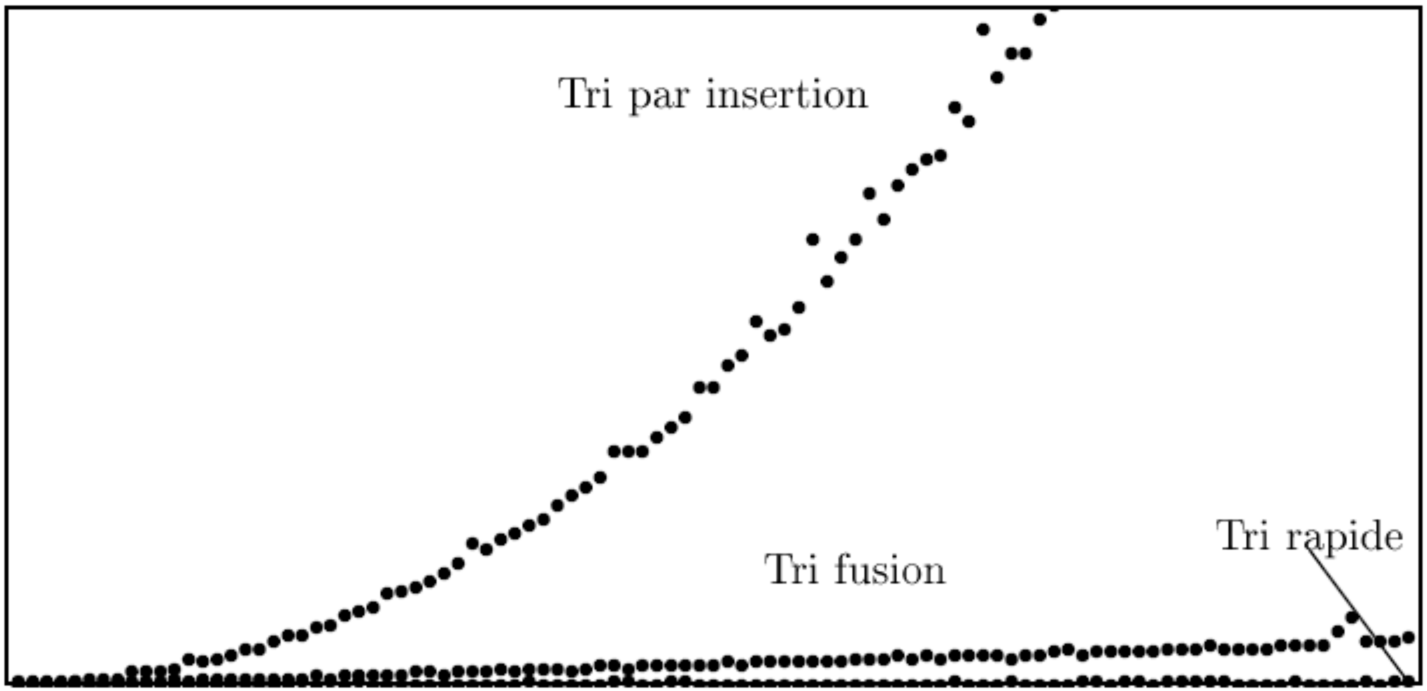


n=5000

FIG. 1.2 – *Complexité moyenne expérimentale (agrandissement)*

Matching theory / practice

Computation Time

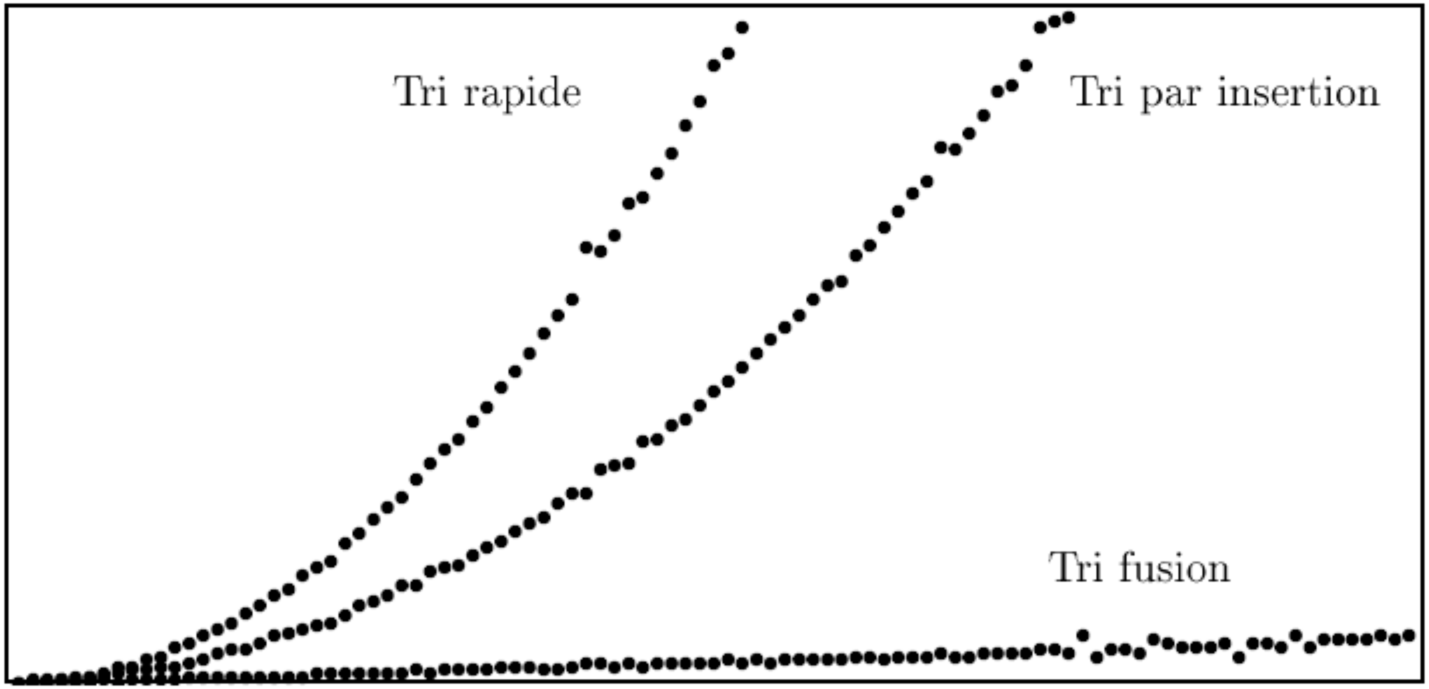


n=5000

FIG. 1.3 – *Complexité expérimentale dans le cas d'un tableau déjà trié*

Matching theory / practice

Computation Time



n=10000

FIG. 1.4 – *Complexité expérimentale dans le cas d'un tableau inversement trié*

Example: Matrix product

- <u>Goal</u> : multiply 2 n x n matrices
- Compute number of additions and multiplications between elements
- Elementary Algorithms : $O(n^3)$
- Advanced Algorithms : $O(n^{2,376})$!!!

Efficient Algorithms ?

- Average-case Complexity
- Worst-case Complexity
- Easy implementation
- Efficiency in practice
- Hybrid Algorithms

The End