La complexité

Temps d’exécution

Tout algorithme a le devoir de s'arrêter un jour.
Il est souhaitable que ce jour ne soit pas trop lointain.

Lorsqu'un algorithme est proposé, on donnera :

- une "preuve de correction" ;
- une estimation de son temps d'exécution.

Exemple

```cpp
#include <iostream>
using namespace std;

int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemple

```cpp
#include <iostream>
using namespace std;

int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

```cpp
int main() {
    int n;
    cout << "Combien de nombres voulez-vous entrer? ";
    cin >> n;
    if (n <= 0) {
        cout << "Veuillez entrer une taille de tableau valide." << endl;
        return 1;
    }
    //Allocation dynamique du tableau avec new
    int* tableau = new int[n];

    cout << "Entrez " << n << " nombres:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> tableau[i];
    }
```

Exemple

```cpp
int main() {
    int n;
    cout << "Combien de nombres voulez-vous entrer? ";
    cin >> n;
    if (n <= 0) {
        cout << "Veuillez entrer une taille de tableau valide." << endl;
        return 1;
    }
    //Allocation dynamique du tableau avec new
    int* tableau = new int[n];

    cout << "Entrez " << n << " nombres:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> tableau[i];
    }
    int valeurMin = trouverMinimum(tableau, n);
    cout << "La valeur minimale est : " << valeurMin << endl;
    //Libération de la mémoire avec delete[]
    delete[] tableau;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Nombre d’étapes

Le temps de calcul va dépendre de la taille de l'entrée. Plus le nombre de valeurs sera grand, plus le tri sera long
C'est cette fonction qu'il s'agit de quantifier.
→ On pourrait compter précisément le temps de calcul de cet algorithme, c'est-à-dire le nombre de cycles du processeur.
Mais :

- on ne sait pas comment ce programme sera compilé ;
- le temps de calcul dépend du langage (et du compilateur / interpréteur) ;
- le temps dépend aussi du processeur.

Nous sommes donc contraints de procéder par approximation.

Cycle du processeur

- Le processeur est un CI cadencé au rythme d'une horloge interne qui envoie des impulsions
- La fréquence d'horloge correspond à un nombre d'instruction par seconde, en Hertz
- Le temps s'écoulant entre chaque battement constitue un cycle processeur, durant lequel le processeur exécute une action élémentaire permettant l'exécution d'une partie d'instruction.
- Chaque instruction correspond à un ensemble de micro-commandes, regroupées dans des micro-instructions. Chaque micro-instruction est exécutée durant un cycle de processeur. L'ensemble des micro-instructions correspondant à une instruction donnée forme un micro-programme.
- Ces micro-programmes sont écrits par le concepteur de processeur : Il est donc impossible de rentrer dans ce détail→ on va procéder par approximation.

Nombre d’étapes

En admettant que chaque instruction s'exécute en un temps fini (à peu près semblable).
On note n la longueur du tableau à trier et :

$$S(n)=\sum_{i=1}^{n} i=\frac{n.(n+1)}{2}$$

```java
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(1)=\sum_{i=1}^{1} i=\frac{1.(1+1)}{2}=1 \qquad S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

$$S(2)=\sum_{i=1}^{2} i=\frac{2.(2+1)}{2}=3 \qquad S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Nombre d'étapes

Exemples :

```
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

array[3,2,5]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //1 comparaison
si 5 < **2** //*1 affectation*
return **2** ;

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Bilan : 4 affectations + 2 comparaisons = 6 opérations

Nombre d'étap

```java
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Array[3,2,5,1]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 5 < 2 //1 comparaison
I =3 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1** ;//*1 affectation*
return **1** ;

Bilan : 6 affectations + 3 comparaisons = 9 opérations

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

array[3,2,5]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 5 < **2** //1 comparaison
return **2** ;

Bilan : 4 affectations + 2 comparaisons = 6 opérations

Nombre d'étap

```java
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Array[3,2,5,1]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 5 < 2 //1 comparaison
I =3 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1 ;**//*1 affectation*
return **1** ;

Bilan : 6 affectations + 3 comparaisons = 9 opérations

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

array[3,2,5]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 5 < **2** //1 comparaison
return **2** ;

Bilan : 4 affectations + 2 comparaisons = 6 opérations

=> Étude du pire des cas !!!!

=> avec l'Étude du pire des cas !!!!

Nombre d'étap

```java
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Array[5,3,2,1]
min = 5 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 3 < 5 //1 comparaison
min = **3** ;//*1 affectation*
I =2 ; //*1 affectation*
si 2 < **3** //1 comparaison
min = **2** ;//*1 affectation*
I =3 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1 ;**//*1 affectation*
return **1** ;

Bilan : 7 affectations + 3 comparaisons = 10 opérations

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

array[3,2,1]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1** ;//*1 affectation*
return **1** ;

Bilan : 5 affectations + 2 comparaisons = 7 opérations

=> avec l'Étude du pire des cas !!!!

Nombre d'étap

```
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Array[5,3,2,1]
min = 5 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 3 < 5 //1 comparaison
min = **3** ;//*1 affectation*
I =2 ; //*1 affectation*
si 2 < **3** //1 comparaison
min = **2** ;//*1 affectation*
I =3 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1 ;**//*1 affectation*
return **1** ;

Bilan : 7 affectations + 3 comparaisons = 10 opérations

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

array[3,2,1]
min = 3 ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < 3 //1 comparaison
min = **2** ;//*1 affectation*
I =2 ; //*1 affectation*
si 1 < **2** //1 comparaison
min = **1** ;//*1 affectation*
return **1** ;

Bilan : 5 affectations + 2 comparaisons = 7 opérations

=> Étude du meilleur des cas !!!!

=> avec l'Étude du meilleur des cas !!!!

Nombre d'étap

```java
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Exemples :

$$S(4)=\sum_{i=1}^{4} i=\frac{4.(4+1)}{2}=10$$

Array[1,2,3,4]
min = **1** ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < **1** //1 comparaison
I =2 ; //*1 affectation*
si 3 < **1** //1 comparaison
I =3 ; //*1 affectation*
si 4 < **1** //1 comparaison
return **1** ;

Bilan : 4 affectations + 3 comparaisons = 7 opérations

$$S(3)=\sum_{i=1}^{3} i=\frac{3.(3+1)}{2}=6$$

Array[1,2,3]
min = **1** ; //*1 affectation*
i = 1 ; //*1 affectation*
si 2 < **1** //1 comparaison
I =2 ; //*1 affectation*
si 3 < **1** //1 comparaison
return **1** ;

Bilan : 3 affectations + 2 comparaisons = 5 opérations

Résumé de la complexité

Choisir ce que l'on va compter
- unité de comparaison des algorithmes
- par exemple le nombre de comparaisons

Ce qui importe est l'ordre de grandeur de la complexité
- **constant**, log n, linéaire, n*log n, n2, 2n
- on s'intéressera essentiellement au nombre de fois où l'on parcourt une structure de donnée (liste, arbre,….)

Calcul de la complexité

Choisir ce que l'on va compter

- unité de comparaison des algorithmes
- par exemple le nombre de comparaisons

Ce qui importe est l'ordre de grandeur de la complexité

- **constant**, log n, linéaire, n*log n, n2, 2n
- on s'intéressera essentiellement au nombre de fois où l'on parcourt une structure de donnée (liste, arbre,….)

```
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

Calcul de la complexité

```
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

La fonction trouverMinimum prend un tableau arr et sa taille taille en tant qu'arguments et renvoie la valeur minimale du tableau.

→ La complexité de cette fonction peut être déterminée comme suit :

1) L'affectation int min = arr[0]; est une opération O(1), aussi appelée opération constante.
2) La boucle for parcourt le tableau depuis le second élément jusqu'à la fin.
   Dans le pire des cas, elle exécute sa boucle interne n−1 fois (où n est la taille du tableau).
   Chaque itération de cette boucle a une complexité de O(1) car les opérations à l'intérieur de la boucle (la comparaison et éventuellement l'affectation) sont des opérations constantes.

Donc, la complexité de la boucle est de:
O(1)+O(1)+...+O(1)O(1)+O(1)+...+O(1) *(n-1 fois)*
=O(n−1) ≈ O(n)
La complexité totale de la fonction trouverMinimum est donc O(n) c'est à dire, la complexité est constante.

Calcul de la complexité

```
int trouverMinimum(int array[], int taille) {
    int min = array[0];

    for (int i = 1; i < taille; i++) {
        if (arr[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

#1

#taille-1
#taille-1
0 à taille-1

Soit 1+ 3 x (taille-1)

Nombre d’étapes

Ce dernier calcul ne représente rien de bien pertinent.

Une quantication précise du temps de calcul est illusoire.

Dans le cas d'un algorithme de tri, il est légitime d'étudier le nombre de comparaisons
(on étudie l'algorithme plutôt que ses implémentations dans des langages de programmation).

Une approche plus solide consistera donc à donner un ordre de grandeur sur le nombre d'opérations.

Quels cas analyser ?

Parmi les entrées de taille n :

- On étudie en général la complexité dans le pire des cas.
- On pourrait également étudier la complexité dans le meilleur des cas (mais cette information ne garantit rien).
- L'étude de la complexité en moyenne peut s'avérer pertinente, mais elle est souvent difficile (en moyenne sur une certaine distribution des entrées de taille n).

Notre exemple de tri s'effectue dans tous les cas en O (n).
En fait, la complexité est équivalente, à un facteur près, à n.

On note alors : Θ(n)

Fonctions usuelles

En analyse, nous retrouvons principalement les fonctions suivantes données par ordre de croissance :

- la fonction constante : O (1)
- la fonction logarithmique : O (log(n))
- la fonction linéaire : O (n)
- la fonction linéarithmique ou quasi-linéaire : O (n.log(n))
- la fonction quadratique : O (n^2)
- les fonctions polynomiales : O (n^c), où c est une constante
- les fonctions exponentielles : O (c^n), où c est une constante
- la fonction factorielle : O (n!)

La récursivité

La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

Suite de Fibonacci

```cpp
#include <iostream>
using namespace std;

// Fonction récursive pour calculer le n-ième terme de la suite de Fibonacci
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cout << "Entrez le rang du terme de la suite de Fibonacci que vous souhaitez obtenir: ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    } else {
        cout << "Le terme de rang " << n << " de la suite de Fibonacci est: " << fibonacci(n) << endl;
    }
    return 0;
}
```

La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

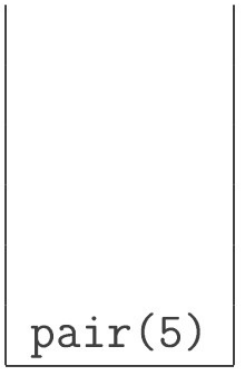
La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, pair (5)



La récursivité

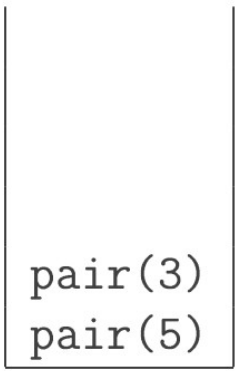
Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, `pair` (5)

appel n°2 : $n_2 = 3$, `pair` (3)



La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

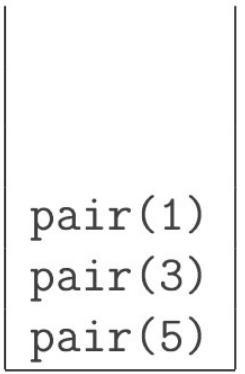
```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, pair (5)

appel n°2 : $n_2 = 3$, pair (3)

appel n°3 : $n_3 = 1$, pair (1)



La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

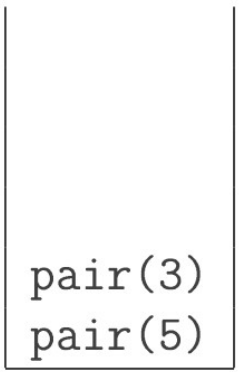
Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, pair (5)

appel n°2 : $n_2 = 3$, pair (3)

appel n°3 : $n_3 = 1$, pair (1)

retour n°3 : $n_3 = 1$, Faux



La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

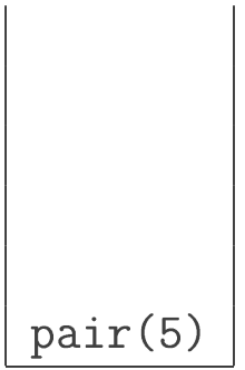
appel n°1 : $n_1 = 5$, pair (5)

appel n°2 : $n_2 = 3$, pair (3)

appel n°3 : $n_3 = 1$, pair (1)

retour n°3 : $n_3 = 1$, Faux

retour n°2 : $n_2 = 3$, Faux



La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, pair (5)

appel n°2 : $n_2 = 3$, pair (3)

appel n°3 : $n_3 = 1$, pair (1)

retour n°3 : $n_3 = 1$, Faux

retour n°2 : $n_2 = 3$, Faux

retour n°1 : $n_1 = 5$, Faux

La récursivité

Une fonction est récursive lorsqu'elle s'appelle elle-même.

```cpp
#include <iostream>
using namespace std;

bool Estpair(int n) {
    if (n > 1) {
        return Estpair(n-2);
    }
    if (n==1)   {
        return false;
    }
    return true;
}

int main() {
    int n;
    cout << "Entrez le nombre à tester ";
    cin >> n;
    if (n < 0) {
        cout << "Veuillez entrer un nombre positif." << endl;
    }
    else {
        if (Estpair(n)) {
            cout << "Le nombre " << n << " est pair" << endl;
        }
        else {
            cout << "Le nombre " << n << " est impair" << endl;
        }
    }
    return 0;
    }
```

Exemple d'exécution avec n = 5.

appel n°1 : $n_1 = 5$, pair (5)

appel n°2 : $n_2 = 3$, pair (3)

appel n°3 : $n_3 = 1$, pair (1)

retour n°3 : $n_3 = 1$, Faux

retour n°2 : $n_2 = 3$, Faux

retour n°1 : $n_1 = 5$, Faux

Le dernier entrant est le premier sorti.
Les appels successifs sont stockés dans une **pile LIFO** (Last In, First Out).
Chaque appel exécute au maximum 2 opérations. L'étude de la complexité revient à compter le nombre d'appels en fonction du paramètre n.

La récursivité

Techniquement, comment est implémentée la récursivité ?
La variable n déclarée dans la fonction est une variable locale.
Chaque appel créé donc une nouvelle variable n (différente des autres), c'est-à-dire qu'elle occupe une zone mémoire spécifique
.
En fait, c'est comme si chaque appel créait une nouvelle instance de la fonction (avec toutes les nouvelles variables locales).
Ce sont chacune de ces instances qui sont référencées dans la pile d'appel.

Exercices

- Écrire une version récursive d'un algorithme Factorielle.
  En entrée : n un entier,
  En sortie : n ! = 1 x 2 x 3 x …. x n
- Écrire un algorithme optimisé qui recherche l'existence d'un nombre dans une liste triée.
  En entrée : une liste triée (ev. écrire un algorithme qui teste que la liste est bien triée)
  En sortie : Vrai ou Faux