IV. Récursivité et complexité algorithmique

1. Traitement récursif
2. Forme d'un algorithme récursif
3. Application : recherche dichotomique
4. Choix entre la récursivité et l'itération
5. Exercices

1. Traitement récursif

Un sous-programme est dit *récursif* lorsqu'il contient une instruction d'appel de sous-programme à lui-même.

Exemple 1 : La factorielle

$$\mathbf{fact(n)} = \begin{cases} \mathbf{1} & \mathbf{si\ n = 0} \\ \mathbf{n * fact(n-1)} & \mathbf{sinon} \end{cases}$$

$0! = 1$
$1! = 1 * 1$
$6! = 6 * 5 * 4 * 3 * 2 * 1 * 1$
$n! = n * (n-1) * (n-2) * \ldots * 1$
$n! = n * (n-1)!$

```
int fact(int n){
       if (n == 0){ return 1; }
       else { return n * fact(n - 1); }
}
```

Exemple 2 : Nombres de Fibonacci

$$\begin{cases} Fibo(0) = Fibo(1) = 1 \\ Fibo(n) = Fibo(n-1) + Fibo(n-2) & si \quad n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	...
Fibo(n)	1	1	2	3	5	8	13	21	...

```
int fibo(int n){
       if (n == 0 or n == 1){ return 1; }
       else { return fibo(n - 1) + fibo(n - 2); }
}
```

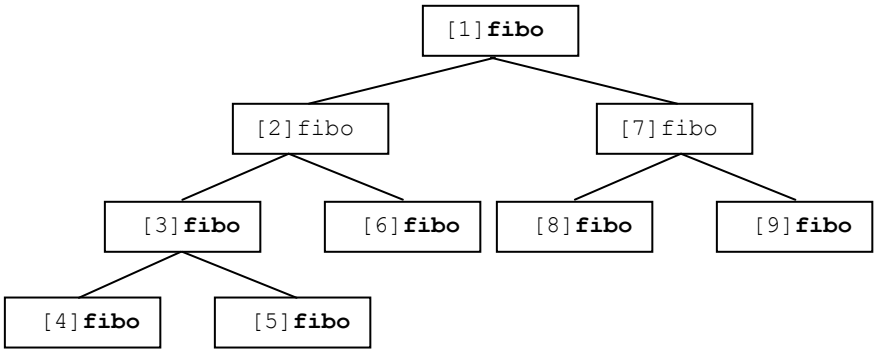
Autres écritures

```
long fibo(int n){
       if (n == 0 or n == 1){ return 1; }
       else { return fibo(n - 1) + fibo(n - 2); }
}
```

```
long fibo(int n){
       if (n == 0 or n == 1){ return 1; }
       return fibo(n - 1) + fibo(n - 2);
}
```

Ordre d'exécution : fonction `fibo()`

`n = 4`



Exercice 1 : Les deux premiers nombres de la séquence de Fibonacci peuvent être soit 1 et 1, soit 0 et 1.

```
Fibo(0) = 1;          ou          Fibo(0) = 0;
Fibo(1) = 1;                      Fibo(1) = 1;
```

Modifier le sous-programme précédent pour qu'il corresponde au deuxième cas (i.e., Fibo(0) = 0; Fibo(1) = 1).

Exercice 2 : écrire la version itérative de la fonction fibo().

2. Forme d'un algorithme récursif

Un sous-programme récursif doit comporter au moins un cas trivial, c'est à dire, un cas sans appel récursif. En d'autres termes, un sous-programme récursif doit avoir la structure suivante :

a/
```
if (condition){ <appel-récursif> }
else { <actions sans appel-récursif> }
```

b/
```
while (condition){
       <appel-récursif>
       .
       .
       .
}
```

c/ ...

3. Application : recherche dichotomique

Soit un tableau `a` ordonné d'entiers, on veut réaliser un algorithme rapide qui teste si une valeur donnée appartient à `a`.

Complexité de la recherche linéaire : `O(n)`
Complexité de la recherche dichotomique : `O(log2 n)`

En d'autres termes, si on applique la recherche dichotomique, le nombre d'opérations nécessaires pour trouver un élément dans un tableau de `n` cases est `log2(n)`.

Notation Big-O

Constant	Logarithmic	Linear	n log n	Quadratic	Polynomial (other than n^2)	exponential
O(1)	O(log n)	O(n)	O(nlog n)	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $(a > 1)$

```
bool binarySearch(int a[], int value, int start, int stop){
       if (start > stop){ return false; }

       int mid = (start + stop) / 2;

       if (a[mid] == value){ return true; }

       if (a[mid] > value){ return binarySearch(a, value, start, mid - 1); }
       else { return binarySearch(a, value, mid + 1, stop); }
}
```

ou

```
bool binarySearch(int a[], int value, int start, int stop){
       if (start > stop){ return false; }

       int mid = (start + stop) / 2;

       if (a[mid] == value){ return true; }
       if (a[mid] > value){ return binarySearch(a, value, start, mid - 1); };
       return binarySearch(a, value, mid + 1, stop);
}
```

Exercice : Modifier le sous-programme ci-dessus pour qu'il renvoie (-1) si l'élément n'existe pas dans le tableau `a`, sinon il renvoie l'indice de cet élément.

```
int binarySearch(int a[], int value, int start, int stop){
       if (start > stop){ return -1; }

       int mid = (start + stop) / 2;

       if (a[mid] == value){ return mid; }

       if (a[mid] > value){ return binarySearch(a, value, start, mid - 1); };

       return binarySearch(a, value, mid + 1, stop);
}
```

4. Choix entre la récursivité et l'itération

L'utilisation de la récursivité peut être très coûteuse en espace mémoire. En général, la récursivité doit être évitée lorsqu'il est facile de trouver une solution itérative. Ces cas sont fréquents quand le problème peut être énoncé en termes de relations récurrentes (e.g., factorielle, nombres de Fibonacci, etc.). Réciproquement, la récursivité est le meilleur procédé lorsqu'une méthode itérative est trop difficile à élaborer ou lorsque les structures de données sous-jacentes sont également récursives.

Exemple

```
// Program_1
int fact(int n){
	if (n == 0){ return 1; }
	else { return n * fact(n - 1); }
}

int main(){
	int i;
	cin >> i;
	int k = fact(i);
}
```

```
// Program_2
int fact(int n){
	int f = 1;
	for (int i = 2; i <= n; i++){ f = f * i; }

	return f;
}

int main(){
	int i;
	cin >> i;
	int k = fact(i);
}
```

Remarque : Les gains réalisés en utilisant le programme itératif au lieu du programme récursif sont doubles:

- Gain mémoire : dans le programme itératif (`program_2`), la taille mémoire nécessaire est réduite et fixe, alors que dans le programme récursif (`program_1`), elle évolue proportionnellement à la variable `i`.
- Gain du temps d'exécution : l'activation et la fin d'activation d'une fonction nécessite un certain travail (allocation de place sur la pile, mise à jour des pointeurs, etc.) qui augmente beaucoup le temps d'exécution du programme.

5. Exercices

1. Tours de Hanoï.