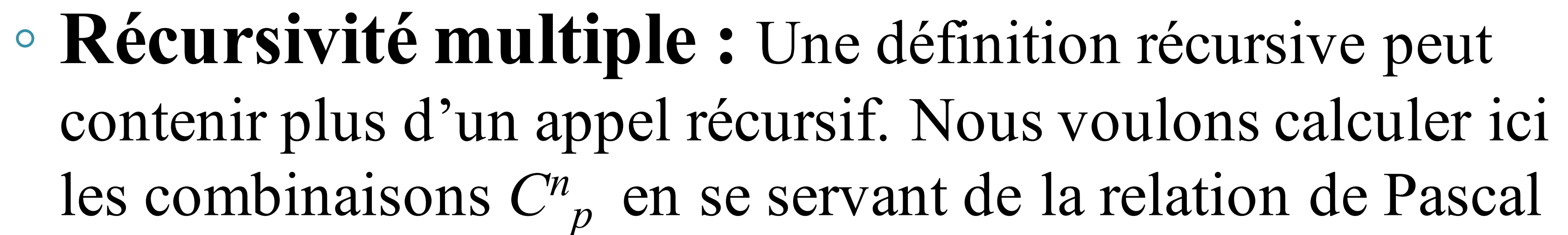
La Récursivité, Complexité

Récursivité

- **Définition :** *Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.*
- **Récursivité simple**

  La fonction puissance $x \mapsto x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$



- **Récursivité multiple :** Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons $C^n{}_p$ en se servant de la relation de Pascal

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

Itération et Récursivité : Exemple

- Calcul de factoriel :

Version itérative

```
Fonction factor (n:entier):entier
Variables  res : entier
Debut
     res ←n;
     Tantque (n > 1)
        res ← res* (n−1);
        n←  n−1;
     FinTantQue
Retourne(res);
Fin
```

Itération et Récursivité : Exemple

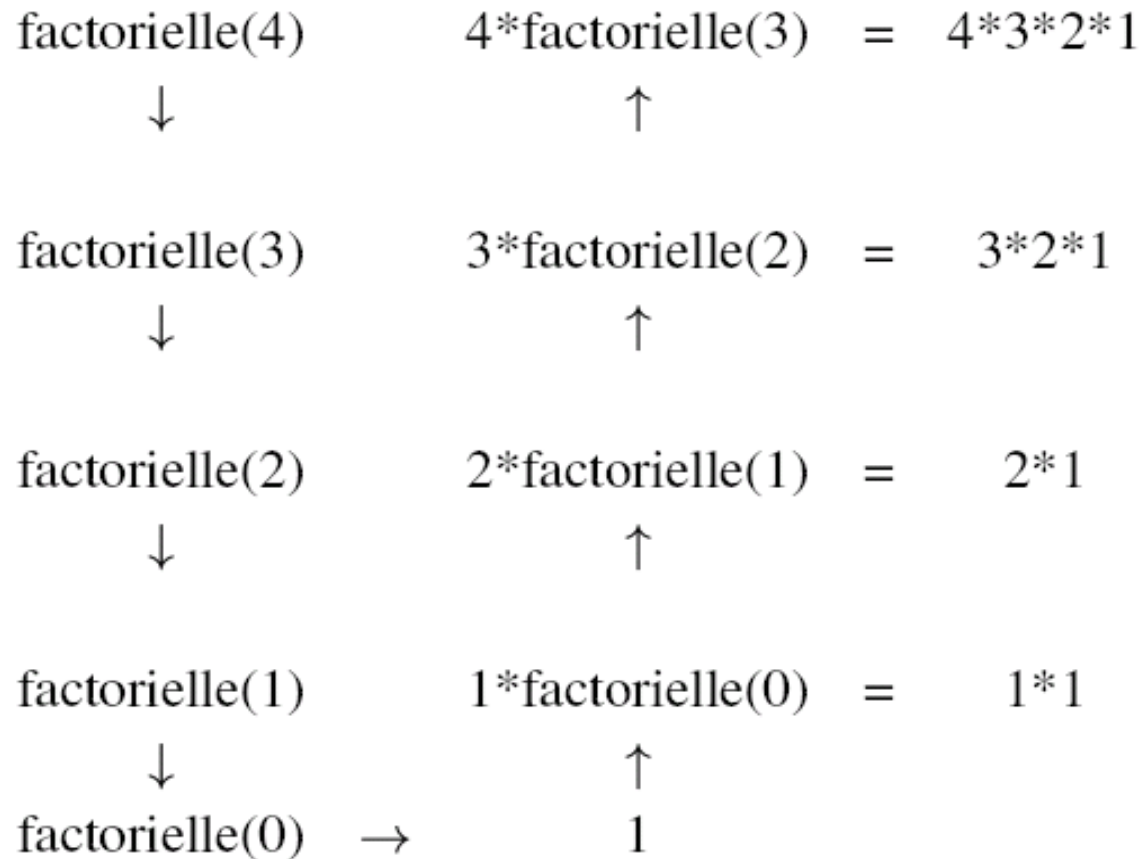
```
Fonction   fact (n : entier ) : entier
     Si (n=0)  alors
   retourne (1);
     Sinon
   retourne (n*fact(n-1));
     Finsi
FinFonction
```

Version récursive

- Un module (fonction ou procédure) peut s'appeler lui-même: on dit que c'est un module **récursif**
- Tout module récursif doit posséder un cas limite (cas trivial) qui arrête la récursivité

Liste des appels de la fonction factoriel (cas du n=4)

Version récursive



Récursivité : observations

- Une fonction est définie récursivement lorsque la valeur de la fonction en un point x est définie par rapport à sa valeur en un point strictement « plus petit »
- De ce fait, le calcul se fait de proche en proche, jusqu'à atteindre le plus petit élément pour lequel il faut une valeur immédiate (c'est-à-dire non récursive)
- Le corps d'une fonction récursive doit toujours exprimer un choix, soit par une expression conditionnelle, soit par une définition par cas
- Au moins un cas terminal rend une valeur qui n'utilise pas la fonction définie

Fonctions récursives : exercice

- La version itérative qui calcule la somme des n premier nombres entier :

Fonction SomPremEnt(n: entier):entier

Variables s,i: entiers

Debut

s←0;

Pour i allant de 1 à n Faire

s ←s+i;

FinPour

Retourne(s);

FinFonction

Écrivez une fonction récursive?

Fonctions récursives : exercice

- La version récursive

```
Fonction SomPremEnt(n: entier):entier
Variables s: entier
Debut
Si  (n=0) alors
   s ←0;
Sinon
    s ←n+SomPremEnt(n-1);
Finsi
Retourne(s);
FinFonction
```

Fonctions récursives : exercice

- Ecrivez une fonction itérative (puis récursive) qui calcule le terme n de la suite de Fibonacci définie par :

$U(0)=U(1)=1$

$U_k=U_{k-1}+U_{k-2}$

Fonctions récursives : exercice (suite)

- Une fonction itérative pour le calcul de la suite de Fibonacci :

```
Fonction Fib (n : entier ) : entier
        Variables i, AvantDernier, Dernier, Nouveau : entier
  AvantDernier ←1;
  Dernier ←1;

Si (n=1 OU n=0) alors
        Nouveau← 1;
 Sinon
      Pour i allant de 2 à n
      Nouveau← Dernier+ AvantDernier;
      AvantDernier ←Dernier;
      Dernier ←Nouveau;
      FinPour
  Finsi
retourne (Nouveau);
FinFonction
```

Fonctions récursives : exercice

- Une fonction récursive qui calcule le terme n de la suite de Fibonacci définie par : $U(0)=U(1)=1$
  $U(n)=U(n-1)+U(n-2)$

```
Fonction Fib (n : entier ) : entier
    Variable res : entier
        Si (n=1 OU n=0) alors
            res ←1;
        Sinon
            res ← Fib(n-1)+Fib(n-2);
        Finsi
    retourne (res);
FinFonction
```

`Remarque: la solution récursive est plus facile à écrire

complexité d'un algorithme

Notion de complexité d'un algorithme

- Pour évaluer l'**efficacité** d'un algorithme, on calcule sa **complexité**
- Mesurer la **complexité** revient à quantifier le **temps** d'exécution et l'espace **mémoire** nécessaire
- Le temps d'exécution est proportionnel au **nombre des opérations** effectuées. Pour mesurer la complexité en temps, on met en évidence certaines opérations fondamentales, puis on les compte
- Le nombre d'opérations dépend généralement du **nombre de données** à traiter. Ainsi, la complexité est une fonction de la taille des données. On s'intéresse souvent à son **ordre de grandeur** asymptotique
- En général, on s'intéresse à la **complexité** dans **le pire des cas** et à la **complexité moyenne**

Exemple 1: Recherche séquentielle

- Recherche de la valeur x dans un tableau T de N éléments :

```
Variables i: entier,
          Trouve : booleean
…
 i←0 ;Trouve ← Faux;
TantQue  (i < N)  ET (Trouve=Faux) Faire
        Si (T[i]=x) alors
            Trouve ← Vrai;
        Sinon
            i←i+1;
        FinSi
FinTantQue
Si (Trouve=Vrai) alors            // c'est équivalent à écrire Si Trouvé=Vrai alors
                    Ecrire ("x appartient au tableau");
Sinon               Ecrire ("x n'appartient pas au tableau");
FinSi
```

Recherche séquentielle : complexité

- Pour évaluer l'efficacité de l'algorithme de recherche séquentielle, on va calculer sa complexité dans le pire des cas. Pour cela on va compter le nombre de tests effectués

- Le pire des cas pour cet algorithme correspond au cas où x n'est pas dans le tableau T

- Si x n'est pas dans le tableau, on effectue 3N tests : on répète N fois les tests (i < N), (Trouvé=Faux) et (T[i]=x)

- La **complexité** dans le pire des cas est **d'ordre N**, (on note **O(N)**)

- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1ms	1s	16mn40s

Exemple 2: Recherche dichotomique

- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique

- **Principe :** diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur x à chaque étape de la recherche. Pour cela on compare x avec T[milieu] :

  - Si x < T[milieu], il suffit de chercher x dans la 1ère moitié du tableau entre (T[0] et T[milieu-1])

  - Si x > T[milieu], il suffit de chercher x dans la 2ème moitié du tableau entre (T[milieu+1] et T[N-1])

algorithme

```
 inf←1 ; sup←N; Trouve ← Faux;
TantQue (inf <=sup)  ET (Trouve=Faux) Faire
      milieu←(inf+sup) /2;
      Si (x=T[milieu]) alors
                        Trouve ← Vrai;
      Sinon
              Si (x>T[milieu]) alors
                        inf←milieu+1;
              Sinon   sup←milieu-1;
              FinSi
    FinSi
FinTantQue
Si (Trouve=vrai) alors    Ecrire ("x appartient au tableau");
Sinon                     Ecrire ("x n'appartient pas au tableau »);
FinSi
```

Exemple d'exécution

- Considérons le tableau T :

4	6	10	15	17	18	24	27	30

- Si la valeur cherché est 20 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	5	5	6
sup	8	8	5	5
milieu	4	6	5	

- Si la valeur cherché est 10 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	0	2
sup	8	3	3
milieu	4	1	2

Recherche dichotomique : complexité

- La complexité dans le pire des cas est d'ordre $\log_2 N$

- L'écart de performances entre la recherche séquentielle et la recherche dichotomique est considérable pour les grandes valeurs de N

  - Exemple: au lieu de N=1milion ≈2^{20} opérations à effectuer avec une recherche séquentielle il suffit de 20 opérations avec une recherche dichotomique