Partie 9: Récursivité

Algorithme de la marche

Marcher n pas :

1. Pour i allant de 1 à n faire un pas
2. ???

Récursivité : définition

Un algorithme ou un programme est dit récursif si il est défini en faisant référence à lui-même.

Un algorithme récursif produit une fonction récursive.

Calculer a^n : faire n fois a*a*a...

```
int puissanceIter (int a,int n){
  int p=1;
  for (i=1;i<=n;i++)
    p=p*a;
  return (p);
}
```

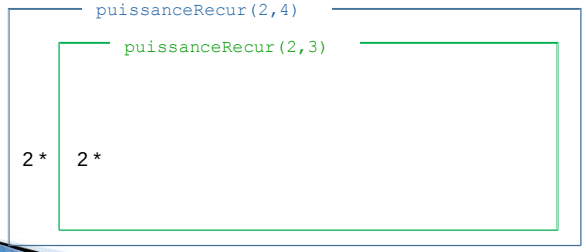
Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```


Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`



Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

Calculer a^n : a* a^{n-1}

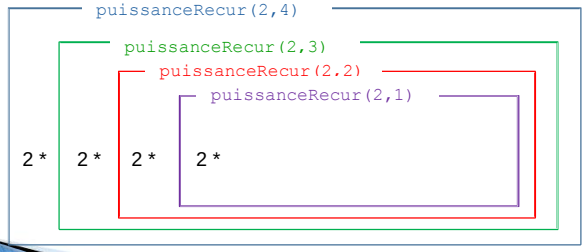
```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

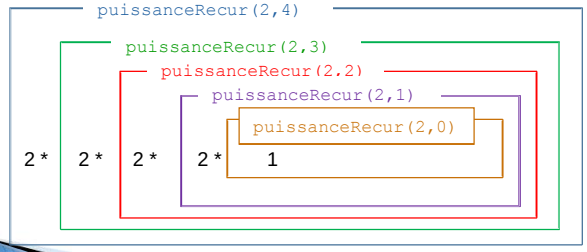
Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

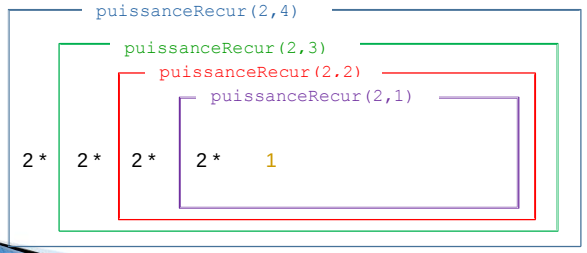
Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

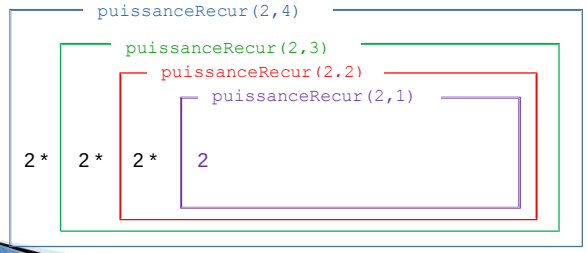
Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

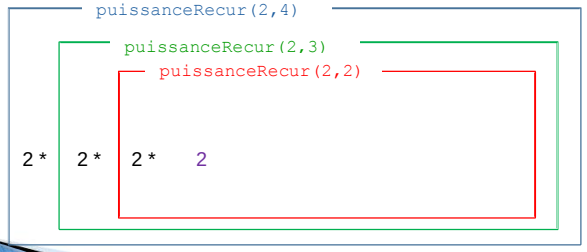
Calculer `puissanceRecur(2,4)`



Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

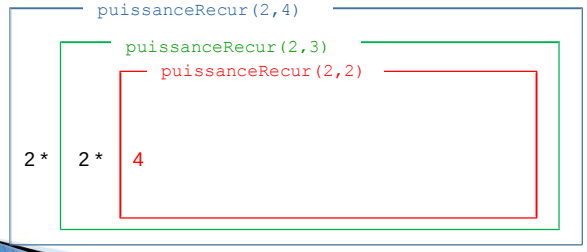
Calculer `puissanceRecur(2,4)`



Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

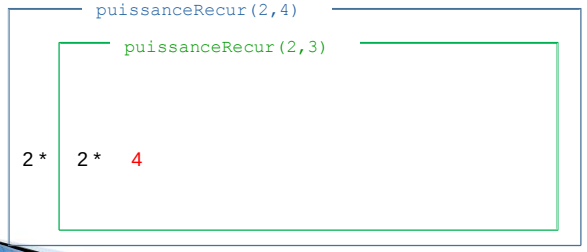
Calculer `puissanceRecur(2,4)`



Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

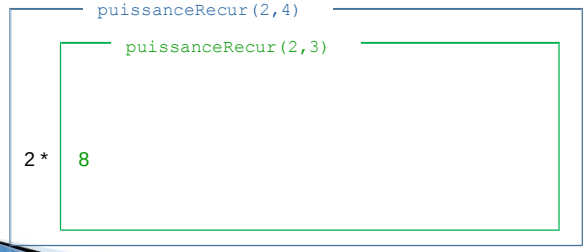
Calculer `puissanceRecur(2,4)`



Calculer a^n : a* a^{n-1}

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

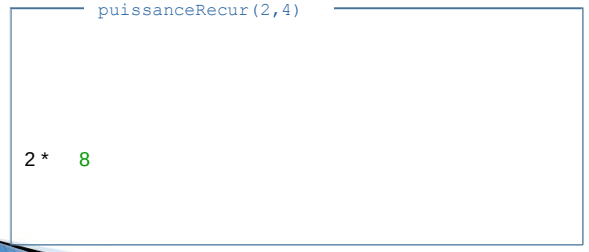
Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

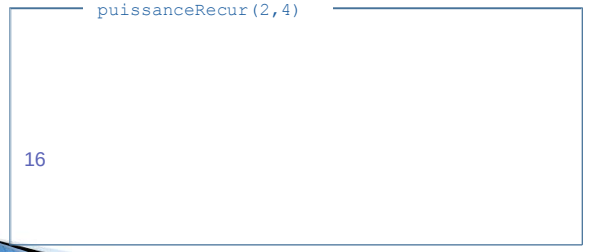
Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`



Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1);
  return (a*puissanceRecur(a,n-1));
}
```

`puissanceRecur(2,4)` ⟶ 16

Récursivité

- La récursivité consiste à remplacer une boucle par un appel à la fonction elle-même.

Chaque fois que l'on désire programmer une fonction récursive, on doit répondre aux questions suivantes :

- Comment le problème au rang n se déduit-il de la solution à un (des) rang(s) inférieur(s) ?
- Quelle est la condition d'arrêt de la récursivité ?

Calculer a^n : $a * a^{n-1}$

```
int puissanceRecur (int a, int n){
  if (n==0)
    return(1); /*condition d'arret */
  return (a*puissanceRecur(a,n-1));
}
```

Calculer `puissanceRecur(2,4)`

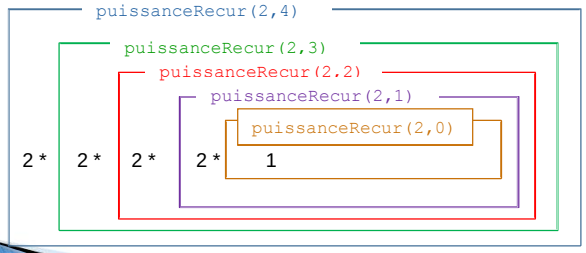
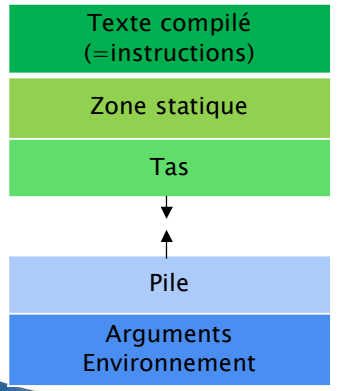


Schéma conceptuel de la mémoire



- La zone statique et le texte sont fixés avant le lancement
- Le arguments sont initialisés au lancement du programme
- La pile et le tas évoluent pendant l'exécution

Complexité : nombre d'appels

Soit :

$u_1 = 1$

$u_n = u_{n-1} + n$

Fonction itérative :

```
int suite(int n){
 int s=0;
 for (i=1;i<=n;i++)
    s=s+i;
 return(s);
}
```

Fonction récursive

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

Complexité en temps : *n pour les 2 versions.*

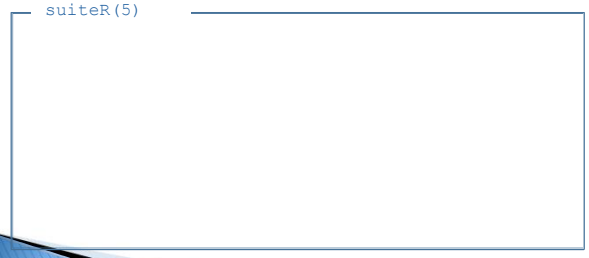
Complexité mémoire : *1 pour la version itérative MAIS n pour la version récursive.*

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

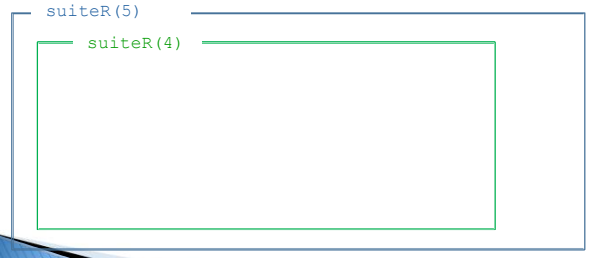
Calculer SuiteR(5)



Complexité : nombre d'appels

Somme des entiers :

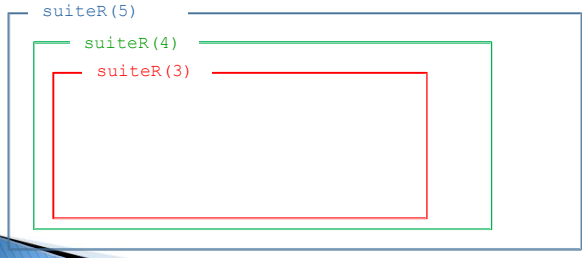
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

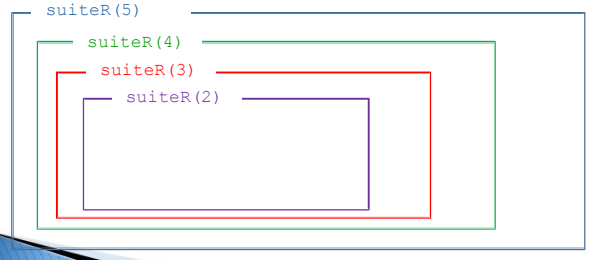
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

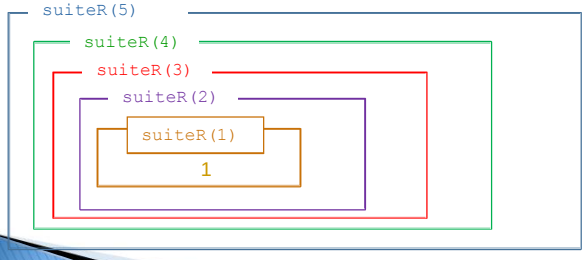
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

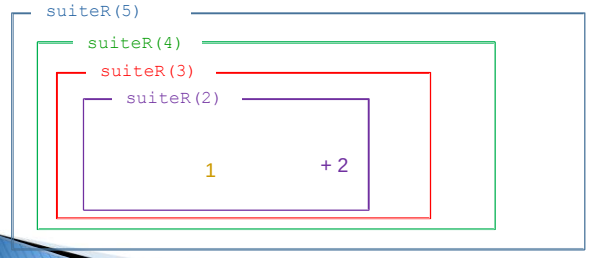
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

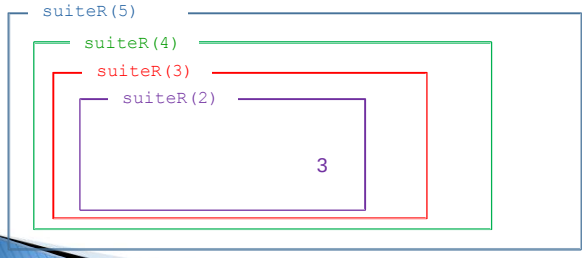
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

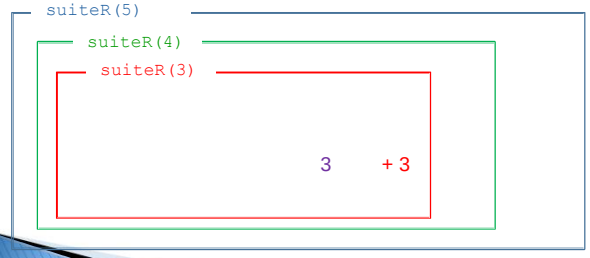
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

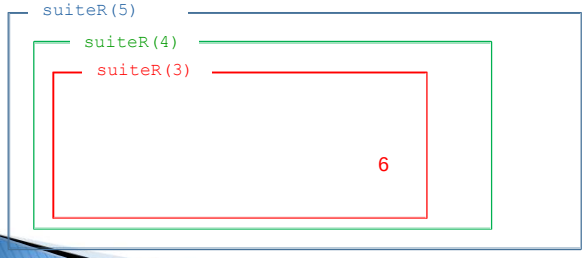
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

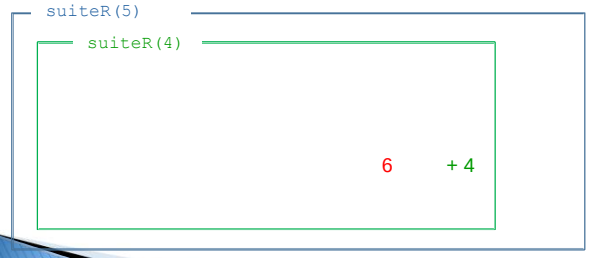
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

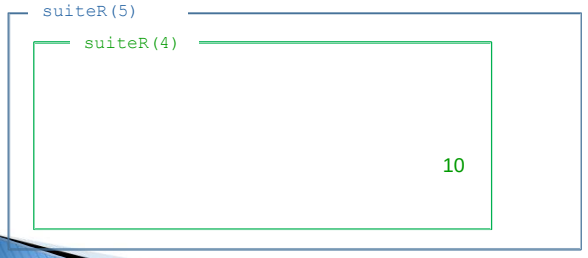
```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```



Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

Complexité : nombre d'appels

Somme des entiers :

```
int suiteR(int n){
  if (n==1)
    return(1);
  return(suiteR(n-1)+n);
}
```

suiteR(5) ⟶ 15

Complexité : Fibonacci

Soit :

$u_0=1$, $u_1=1$

$u_n= u_{n-1}+ u_{n-2}$ *pour* $n>2$

```
int fibo(int n){
  if(n<2)
    return(1);
 return(fibo(n-1)+fibo(n-2));
}
```

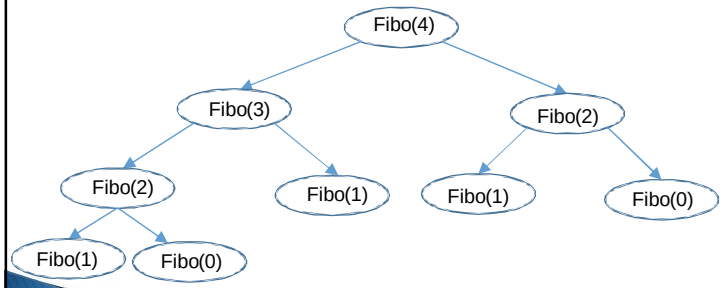
⚠ Complexité en $\Omega(\Phi^n)$ avec $\Phi \cong 1{,}61$ donc complexité exponentielle.
Il existe d'autres algorithmes en O(n) et même en $O(\log_2(n))$ pour faire le même calcul.

Complexité : Fibonacci

Soit :

$u_0=1$, $u_1=1$

$u_n= u_{n-1}+ u_{n-2}$ *pour* $n>2$



Exemple : factorielle

```
int
Factorielle(int n) {
   if (n <= 1)
     return 1;
   return (n * Factorielle(n-1));
}
int
main(void){
  return(Factorielle(3));
}
```

Exemple : Rechercher un élément dans un tableau itératif

```
int recherche(int t[]; int n; int x){
  int i = 0;
  while(i<n){
     if (t[i]==x)
       return(i);
     i++;
    }
 return(-1);
}
```


Exemple : Rechercher un élément dans un tableau récursif

```
int recherche(int t[]; int n; int x){
  if (n==0)
   return (-1);
  if (t[n-1]==x)
    return(n-1);
 return(recherche(t, n-1,x);
```