Programmation C et Structures de données (Prog 2)

Cours 1

Récursivité et pile d'appels

Pierre Fouilhoux & Christophe Tollu
pierre.fouilhoux@lipn.fr et ct@lipn.univ-paris13.fr

30 janvier 2024

Fonctions récursives

Définitions

- Une **fonction récursive** est une fonction qui, dans sa définition, contient un appel à elle-même.
- Un **appel récursif** est un appel réalisé alors que l'exécution d'un appel précédent de la même fonction n'est pas achevé.

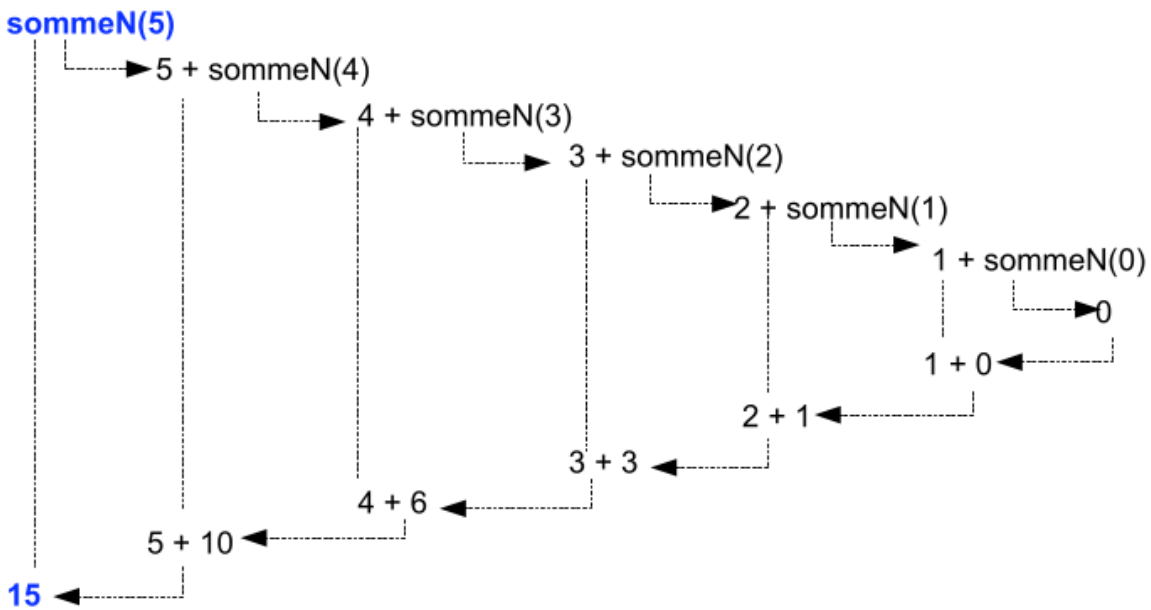
Exemple : somme des **n** premiers entiers

```
1 int sommeN (unsigned int n)
2 {
3   unsigned int res;
4   if (n == 0)
5     return 0;
6   else
7     return n + sommeN(n-1);
8 }
```

Déroulés des appels récursifs

Appel de **sommeN(5)**

```
1 int sommeN (unsigned int n)
2 {
3   unsigned int res;
4   if (n == 0)
5     return 0;
6   else
7     return n + sommeN(n-1);
8 }
```



Déroulés des appels récursifs

Remarques concernant les appels itérés :

- Lors d'un appel à une fonction (récursive ou non), la fonction où a été fait l'appel se met en attente de la fin de la fonction appelée.
- Le cas d'une fonction récursive n'est pas particulier : chaque appel à une fonction (récursive ou non) est totalement indépendant des autres !
- Si la fonction appelée retourne une valeur, c'est bien à la fin de l'appel que l'on obtient cette valeur :
  dans l'exemple précédent, la somme est effectuée après l'évaluation de la valeur de la fonction.

Principe générale de la récursivité

Principe générale de la récursivité

- Décomposer un problème en un problème plus “petit”
- Pour la récursion sur les entiers :
  à chaque appel récursif, on réduit la valeur de cet entier
- Pour la récursion sur les tableaux :
  à chaque appel, on réduit la plage des cases à étudier du tableau

Mise en œuvre de la récursivité

Comment coder récursivement ?

- Une fonction récursive est composée d'au moins
  - un **cas de base** : cas où le calcul fait par la fonction est simple et ne nécessite pas d'appel récursif
  - un **cas général** : cas où le calcul est fait en utilisant (au moins) un résultat d'un appel récursif.

Attention

- Lors du cas général, la valeur d'au moins un des paramètres effectifs doit changer
- Le cas général doit toujours "converger" vers un cas de base en un nombre fini d'appels récursifs.

Récursivité : terminaison

Voici un cas où cela ne converge pas :

Empilement (potentiellement) infini des appels récursifs

```
1 int f(int n)
2 { return f(n-1);}
```

Mais ce n'est pas si simple à repérer ! L'exemple suivant dest plus compliqué à analyser

Fonction de Ackerman

```
1 unsigned int ack (int m, int n)
2 {
3   if (m == 0)
4     return n + 1;
5   if (n == 0)
6     return ack(m-1, 1);
7   return ack(m-1, ack(m, n-1));
8 }
```

Mais on peut prouver théoriquement que cette fonction a toujours une fin : (on parle de la terminaison d'un algorithme).

Exemple du calcul du nombre factoriel

Essayons de constuire une fonction récursive pour le nombre factoriel.
On sait que :

- $0! = 1$
- $n! = 1 * 2 * 3 * \ldots * n \quad$ si $n \geq 1$

On peut voir que :

- $n! = n * (n-1)! \quad$ si $n \geq 1$
  C'est-à-dire que le nombre factoriel de n est exactement n fois le nombre factoriel de $n-1$ C'est le cas général.
- $n! = 1 \quad$ si $n = 0$: c'est un cas de base
- $n! = 1 \quad$ si $n = 1$: c'est un autre cas de base (non nécessaire)

Calcul de la factorielle d'un entier positif

```
1 unsigned factorielleRec (unsigned n)
2 {
3   if ( (n == 0) || (n == 1) ) return 1;
4   return n * factorielleRec(n-1);
5 }
```

Exemple du calcul du nombre factoriel

Essayons de constuire une fonction récursive pour le nombre factoriel.
On sait que :

- $0! = 1$
- $n! = 1 * 2 * 3 * \ldots * n \quad$ si $n \geq 1$

On peut voir que :

- $n! = n * (n-1)! \quad$ si $n \geq 1$
  C'est-à-dire que le nombre factoriel de n est exactement n fois le nombre factoriel de $n-1$ C'est le cas général.
- $n! = 1 \quad$ si $n = 0$: c'est un cas de base
- $n! = 1 \quad$ si $n = 1$: c'est un autre cas de base (non nécessaire)

Calcul de la factorielle d'un entier positif

```
1 unsigned factorielleRec (unsigned n)
2 {
3   if ( (n == 0) || (n == 1) ) return 1;
4   return n * factorielleRec(n-1);
5 }
```

Exemple du calcul du nombre factoriel

Essayons de constuire une fonction récursive pour le nombre factoriel.
On sait que :

- $0! = 1$
- $n! = 1 * 2 * 3 * \ldots * n \quad$ si $n \geq 1$

On peut voir que :

- $n! = n * (n-1)! \quad$ si $n \geq 1$
  C'est-à-dire que le nombre factoriel de n est exactement n fois le nombre factoriel de $n-1$ C'est le cas général.
- $n! = 1 \quad$ si $n = 0$: c'est un cas de base
- $n! = 1 \quad$ si $n = 1$: c'est un autre cas de base (non nécessaire)

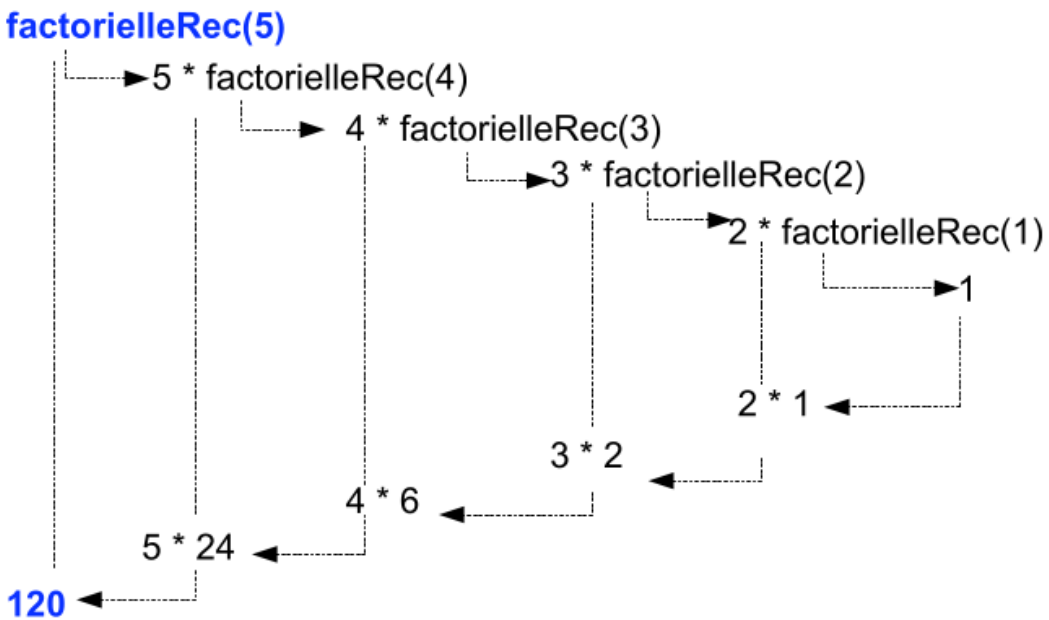
Calcul de la factorielle d'un entier positif

```
1 unsigned factorielleRec (unsigned n)
2 {
3   if ( (n == 0) || (n == 1) ) return 1;
4   return n * factorielleRec(n-1);
5 }
```

Exemple du calcul du nombre factoriel

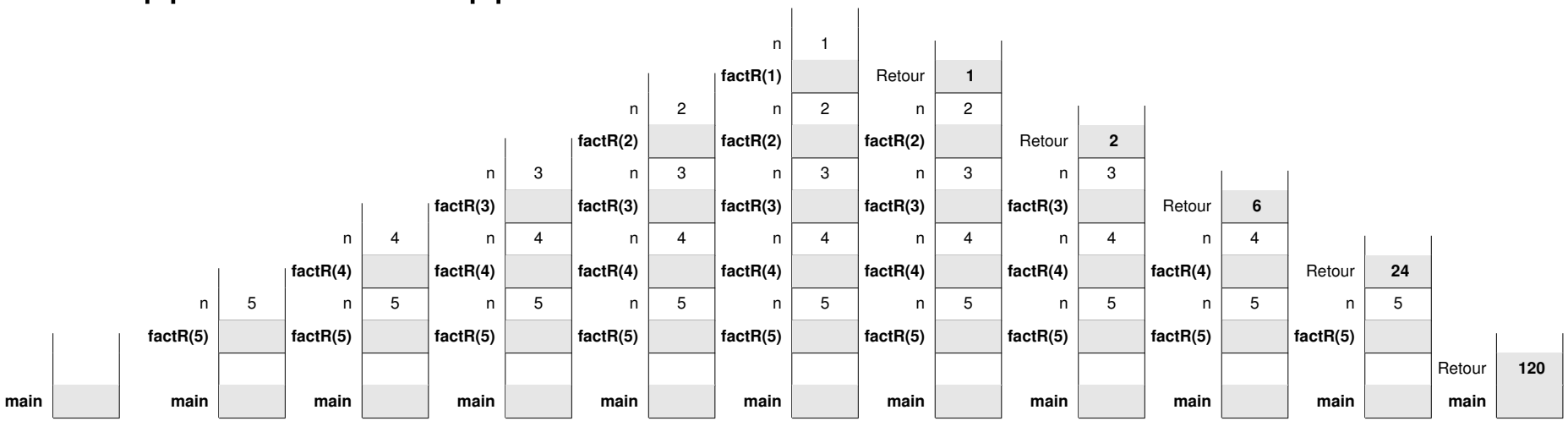
Calcul de la factorielle d'un entier positif

```
1 unsigned factorielleRec (unsigned n)
2 {
3   if ( (n == 0) || (n == 1) ) return 1;
4   return n * factorielleRec(n-1);
5 }
```



Récursivité et efficacité : récursivité terminale

Pile d'appels lors des appels récursifs :

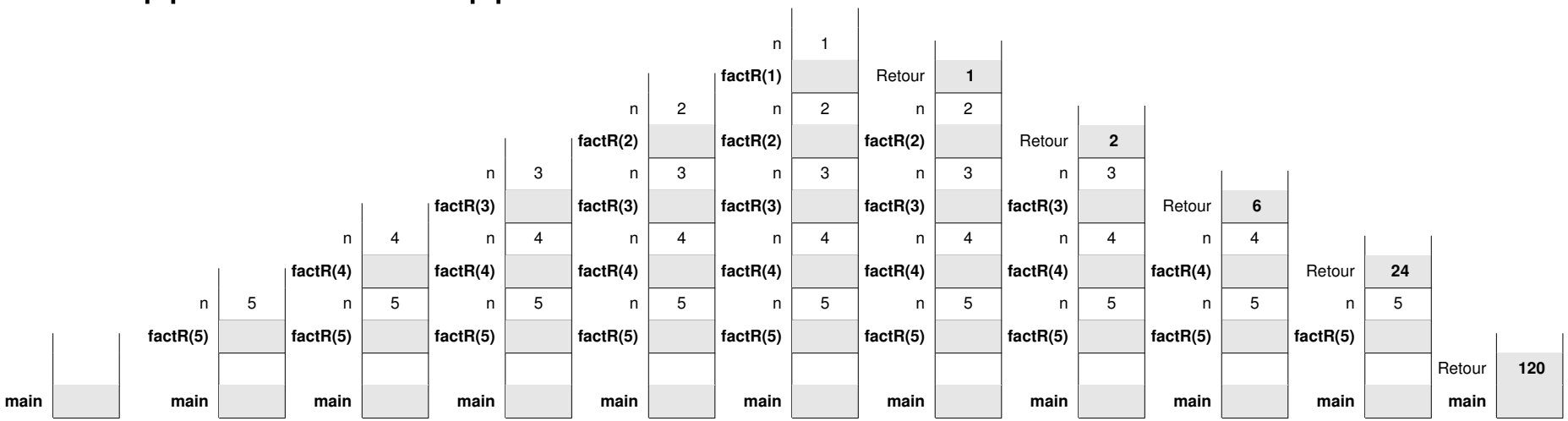


Occupation mémoire

On peut remarquer que la pile sera, au moment où elle sera au plus haut, d'une taille proportionnelle à n.

Récursivité et efficacité : récursivité terminale

Pile d'appels lors des appels récursifs :



Occupation mémoire

On peut remarquer que la pile sera, au moment où elle sera au plus haut, d'une taille proportionnelle à n.

Récursivité et tableaux

Comment afficher récursivement les cases d'un tableau ?

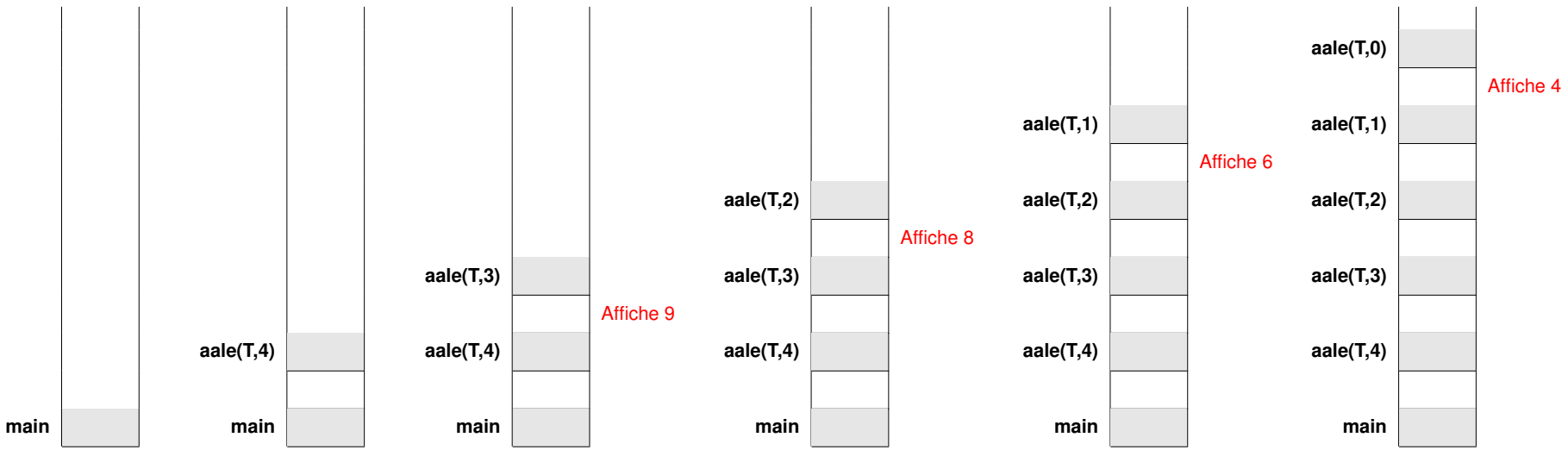
Affichage d'un tableau

```
1 void affiche_a_l_envers(int T[], int taille){
2   if (taille >0){
3     printf("%d ", T[taille -1]);
4     affiche_a_l_envers(T, taille -1);
5   }
6 }
```

Cette fonction affiche la dernière case du tableau avant les appels :
le tableau sera affiché à l'envers.

Affichage récursif

Pile d'appels lors des appels récursifs de affiche_a_l_envers (aale) :
pour $T = [4, 6, 8, 9]$



puis les fonctions se terminent une après l'autre (sans rien faire).

Récursivité et tableaux

Comment afficher récursivement les cases d'un tableau ?

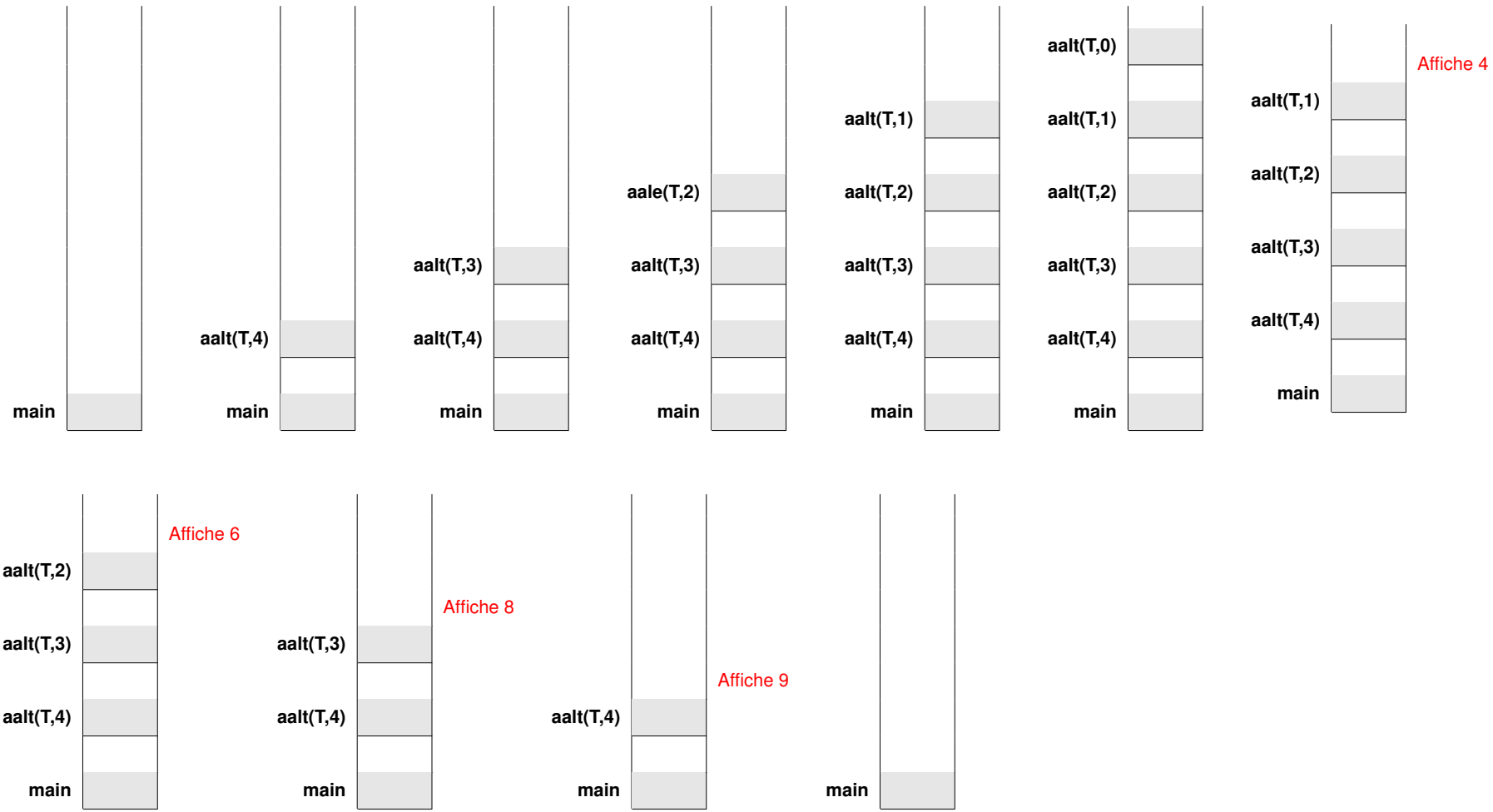
Affichage d'un tableau

```
1 void affiche_a_l_endroit(int T[], int taille){
2   if (taille >0){
3     affiche_a_l_endroit(T, taille -1);
4     printf("%d ", T[taille -1]);
5   }
6 }
```

La première fonction affiche les cases de 0 à taille-1 :
En effet, les fonctions pour un tableau de 4 cases sont appelées dans l'ordre :
pour 4, puis pour 3, puis pour 2, puis pour 1, puis pour 0
sans avoir rien affiché encore.
Puis l'appel pour la valeur 0 n'affiche rien.
Puis l'appel pour 1 affichera la case 0
Puis l'appel pour 2 affichera la case 1
..
Le tableau s'affiche dans l'ordre !

Affichage récursif

Pile d'appels lors des appels récursifs de affiche_a_l_endroit (aalt) : pour $T = [4, 6, 8, 9]$



Récursivité et tableaux

Recherche dans un tableau de flottants **trié**

```
1  /* Renvoie l'indice d'une valeur du tableau /'egale a x */
2  /* si une telle valeur existe, renvoie -1 sinon */
3  /* ATTENTION PRE-REQUIS: le tableau doit \^etre tri/'e */
4
5  int recherche_tab_dichotomique(float x, float * tab, int deb, int fin)
6  {
7    int milieu = (deb + fin)/2;
8    if (fin < deb)
9      return -1;
0    if (x < tab[milieu])
1      return recherche_tab(x, tab, deb, milieu - 1);
2    if (x > tab[milieu])
3        return recherche_tab(x, tab, milieu + 1, fin);
4    return milieu;
5  }
```

L'appel initial pourrait être...

```
1  /* En supposant que t est un tableau de flottants */
2  /* de taille au moins n et dont les valeurs sont rangees dans l'ordre croissant */
3  int result = recherche_tab(-0.75, t, 0, n-1);
```

Zones de mémoire

Avertissement liminaire

- La **présentation de la mémoire** qui suit est **schématique**. Elle ne vise qu'à faciliter la compréhension des pointeurs (et plus tard de l'allocation dynamique).
- Pour plus de détails sur l'organisation de la mémoire, les modes d'allocation, la correspondance adresses virtuelles – adresses physiques, etc., il faudra attendre un cours sur l'**architecture des machines** et les **systèmes d'exploitation**.

En première approximation...

la mémoire peut être représentée comme **un (long) tableau d'octets segmenté en plusieurs zones**.

Zones de mémoire

Variables locales, paramètres d'entrée des fonctions, adresse de retour

Allocation dynamique

Variables globales et statiques

Instructions des fonctions

Zone texte et zone de données

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions du programme.

Au lancement de l'exécution du programme

- Les (instructions des) fonctions sont chargées en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en **lecture seule**.
- Sa taille est déterminée avant l'exécution.

Zone texte et zone de données

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions du programme.

Au lancement de l'exécution du programme

- Les (instructions des) fonctions sont chargées en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en **lecture seule**.
- Sa taille est déterminée avant l'exécution.

Zone texte et zone de données

Rappel

Un fichier exécutable contient déjà :

- Une zone correspondant aux variables globales et statiques.
- Une zone correspondant à l'ensemble des fonctions du programme.

Au lancement de l'exécution du programme

- Les (instructions des) fonctions sont chargées en mémoire dans la **zone texte**.
- Chaque instruction de chaque fonction possède une adresse propre en mémoire.

Zone texte

- Cette zone est en **lecture seule**.
- Sa taille est déterminée avant l'exécution.

Zone texte et zone de données

Au lancement de l'exécution du programme

- La zone de données de l'exécutable est recopiée dans la mémoire.
- Tous les objets globaux et statiques ont une adresse fixée avant l'exécution.

Zone de données

- Cette zone peut être en **lecture seule ou en lecture–écriture**.
- La taille de cette zone (et de chaque objet global ou statique) est déterminée avant l'exécution..

Accessibilité

Les variables dans la zone de données et les fonctions dans la zone texte sont **accessibles depuis tout point du programme** (ou depuis tout point d'un segment correspondant à un fichier objet pour les objets statiques).

Zone texte et zone de données

Au lancement de l'exécution du programme

- La zone de données de l'exécutable est recopiée dans la mémoire.
- Tous les objets globaux et statiques ont une adresse fixée avant l'exécution.

Zone de données

- Cette zone peut être en **lecture seule ou en lecture–écriture**.
- La taille de cette zone (et de chaque objet global ou statique) est déterminée avant l'exécution..

Accessibilité

Les variables dans la zone de données et les fonctions dans la zone texte sont **accessibles depuis tout point du programme** (ou depuis tout point d'un segment correspondant à un fichier objet pour les objets statiques).

Zone texte et zone de données

Au lancement de l'exécution du programme

- La zone de données de l'exécutable est recopiée dans la mémoire.
- Tous les objets globaux et statiques ont une adresse fixée avant l'exécution.

Zone de données

- Cette zone peut être en **lecture seule ou en lecture–écriture**.
- La taille de cette zone (et de chaque objet global ou statique) est déterminée avant l'exécution..

Accessibilité

Les variables dans la zone de données et les fonctions dans la zone texte sont **accessibles depuis tout point du programme** (ou depuis tout point d'un segment correspondant à un fichier objet pour les objets statiques).

Pile d'appel

Caractérisation

La pile d'appel permet de :

- stocker les **variables locales**,
- stocker les **paramètres d'entrée** des fonctions,
- sauvegarder l'adresse de retour, la taille du segment actif de la pile (ou le *frame pointeur*), ...

Précision

- Dans ce cours on représente les appels de fonctions avec une **zone grisée** dans la pile.
- Cette zone grisée contient, entre autres, les informations permettant, à la fin de l'exécution d'une fonction, de **dépiler correctement** en réactivant la zone (*frame*) de l'appel précédent.
- Le nombre et la nature des informations sauvegardées dans cette zone, donc sa taille, **dépendent de l'architecture**.

Les limites de la pile d'appel

La taille de la pile

- La pile a une taille limitée.
- À chaque appel de fonction, on ajoute des informations dans la pile.
- Si un programme effectue trop d'appels imbriqués, la pile déborde.

Autre exemple : la suite de fibonnacci

La suite de Fibonacci :

$$u_n = \begin{cases} u_{n-1} + u_{n-2} & si \quad n \geq 2 \\ 1 & si \quad n = 1 \\ 0 & si \quad n = 0 \end{cases}$$

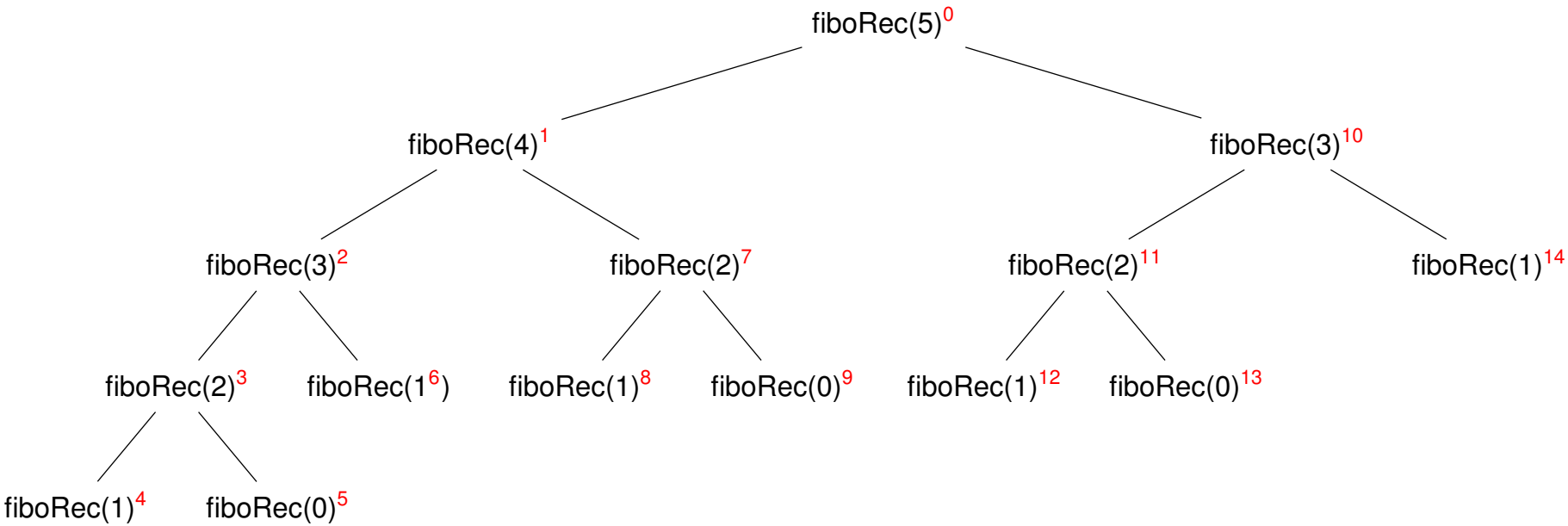
a une écriture mathématique naturellement récursive.

Calcul du n^{ème} terme de la suite de Fibonacci

```
1 unsigned int fib (unsigned int n)
2 {
3   if (n == 0 || n == 1)
4       return n;
5   return fib(n-1) + fib(n-2);
6 }
```

Autre exemple : la suite de fibonnacci

Arborescence des appels récursifs



L'exposant en rouge désigne l'ordre dans lequel les appels sont réalisés.

L'arborscence représente quelle fonction appelle quelle fonction.

Récursivité et efficacité

```
1 unsigned int fib (unsigned int n)
2 {
3   if (n == 0 || n == 1)
4       return n;
5   return fib(n-1) + fib(n-2);
6 }
```

Appels récursifs redondants

- Pour exécuter l'appel f(20), on exécutera plus de trois mille trois cents appels f(2) !
- En plus d'être lent, cela peut faire "sauter" la mémoire :
  "Out of memory"
- Certains systèmes ont aussi un nombre maximum d'appels ou de taille mémoire pour une pile :
  "Stack overflow"

Récursivité et efficacité

Récursif vs itératif

- Il existe des techniques pour remédier à la redondance des appels dans une fonction récursive.
- Il ne faut donc pas croire que l'exécution d'une fonction récursive est toujours nettement plus coûteuse que celle d'une fonction itérative effectuant la même tâche !
- En outre, l'écriture d'une fonction récursive est, dans de nombreuses applications, beaucoup plus « naturelle » . . .

Récursivité et efficacité

Exercice vu en TD : version itérative $\Theta(n^2)$)

```
1 item max_diff_tab (double t[], int deb, int fin) {
2   int i, j;
3   double res = 0;
4   for (i = deb; i <= fin; ++i)
5     for (j = i+1; j <= fin; ++j)
6       if (t[j]-t[i] > res)
7         res = t[j]-t[i];
8   return res;
9 }
```

Exercice vu en TD (version récursive $\Theta(n \log_2 n)$)

```
1 item max_diff_tab_rec (double t[], int deb, int fin) {
2   int m = (deb+fin)/2;
3   double max_fin_min_deb, max_diff_deb, max_diff_fin;
4   if (fin <= deb)
5     return 0;
6   max_diff_deb = max_diff_tab_rec(t, deb, m);
7   max_diff_fin = max_diff_tab_rec(t, m+1, fin);
8   max_fin_min_deb = max_tab(t, m+1, fin) - min_tab(t, deb, m);
9   return max_trois(max_diff_deb, max_diff_fin, max_fin_min_deb);
0 }
```

Première technique : la mémoïsation

La mémoïsation

- Utiliser un tableau (global ou passé en paramètre) qui stocke les valeurs déjà rencontrées.
  Le tableau est indicé par le même paramètre que la fonction.
- Le tableau est initialisé à une valeur impossible à rencontrer lors du calcul (par exemple -1 ou 0).
- Lors de l'appel récursif, on ajoute un test pour savoir si la valeur a été ou non déjà calculée auparavant.
  Si c'est le cas, on retourne cette valeur :
  Et si ce n'est pas le cas, on met à jour le tableau avec la valeur calculée.
  Ainsi aucun appel récursif en doublon !

Récursivité et efficacité : mémoïsation

Fonction récursive classique

```
1 unsigned int fib (unsigned int n)
2 {
3   if (n == 0 || n == 1)
4       return n;
5   return fib(n-1) + fib(n-2);
6 }
```

Dans cette fonction récursive :

- les appels récursifs ont déjà été exécutés quelques instants auparavant et on ne s'en sert pas
- on va stocker les valeurs obtenues pour des entrées **n** déjà rencontrées dans un tableau **tab_fib**.
- on initialise les cases de **tab_fib** à -1 (car ce n'est pas une valeur possible pour cette suite)

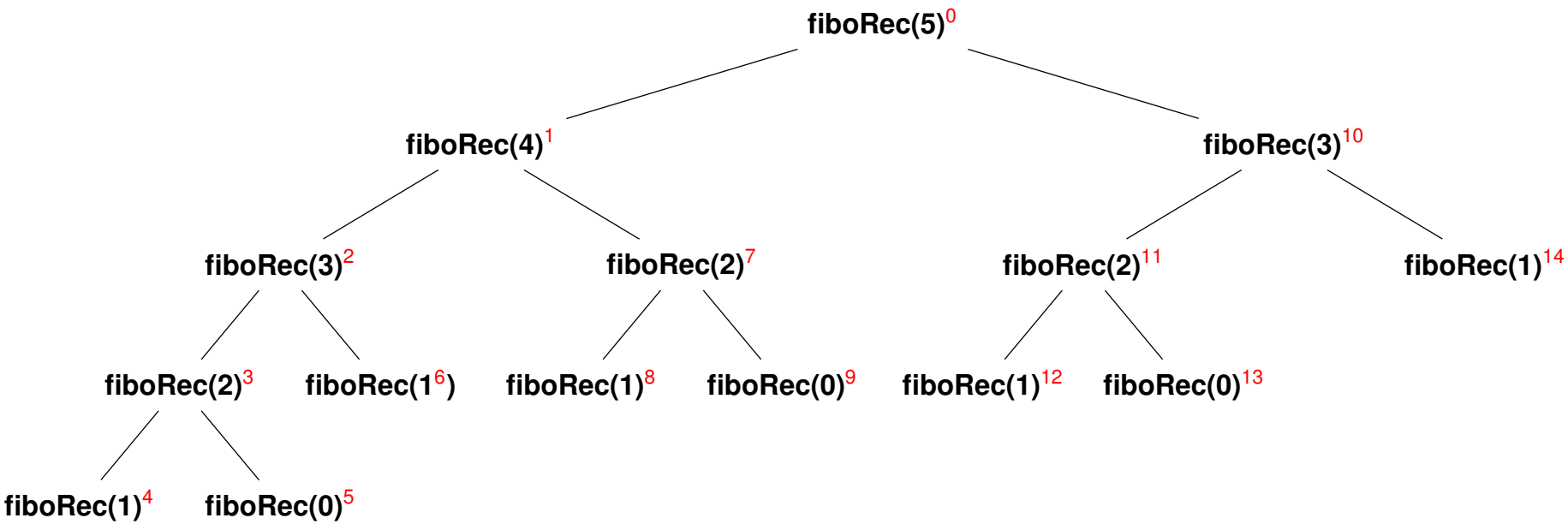
Récursivité et efficacité : mémoïsation

Fibonacci « mémoïsé »

```
1  #define N 50
2  unsigned tab_fib[N] = {-1}; // Tableau de stockage des valeurs memoisee
3                              // La valeur -1 sert \'a encoder une valeur inconnue
4
5  unsigned fib_mem(unsigned tab_fib[], unsigned n)
6  {
7    if (n == 0 || n == 1){
8      tab_fib[n] = n;
9      return n;
0    }
1
2    if (tab_fib[n] == -1) // si valeur inconnue, on fait le cacul
3      tab_fib[n] = fib_mem(fib_tab, n-1) + fib_mem(fib_tab, n-2);
4
5    return tab_fib[n]; // dans tous les cas, on retourne la valeur
6  }
```

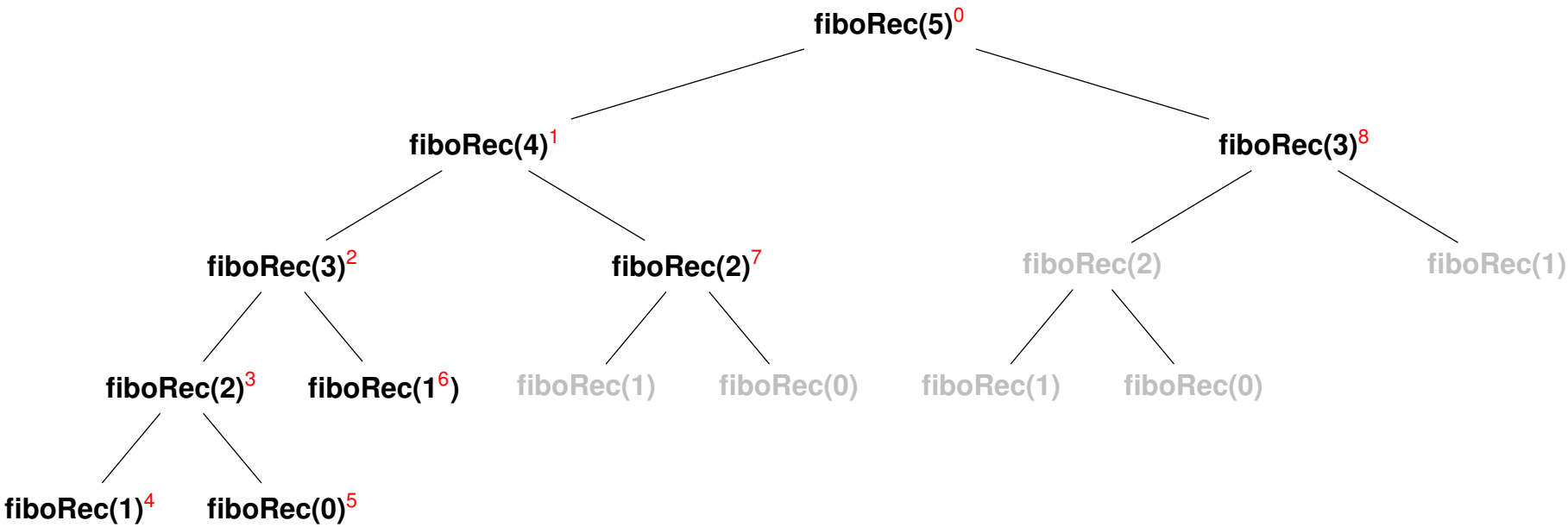
Autre exemple : la suite de fibonnacci

Version classique



Autre exemple : la suite de fibonnacci

Version mémoïsée



Récursivité terminale / non terminale

Définitions

- Un appel récursif est **non terminal** s'il est utilisé comme sous-expression stricte d'une expression évaluée dans l'appel "parent" de la (même) fonction. [cf. fonction `somme_premiers_entiers`]
- Un appel récursif est **terminal** si son résultat est celui de l'appel "parent" de la (même) fonction [cf. fonction `pgcd`] .
- Une fonction récursive est **(à récursivité) terminale** si tous ses appels récursifs sont terminaux.

Exemple de code récursif naturellement terminal

Exemple (fonction récursive terminale)

```
1 unsigned int pgcd (unsigned int m, unsigned int n)
2 { /* on suppose que m et n sont strictements positifs */
3   if (m % n == 0 ) /* si n divise m */
4     return n;
5   else
6     return pgcd(n, m % n);
7 }
```

On peut voir que l'appel récursif ne contient uniquement que l'appel récursif : l'appel intial renverra exactement le retour de l'appel du cas de base !

Suspendre les fonctions pendant les appels récursifs successifs est inutile.

Exemple de code récursif naturellement terminal

Exemple (fonction récursive terminale)

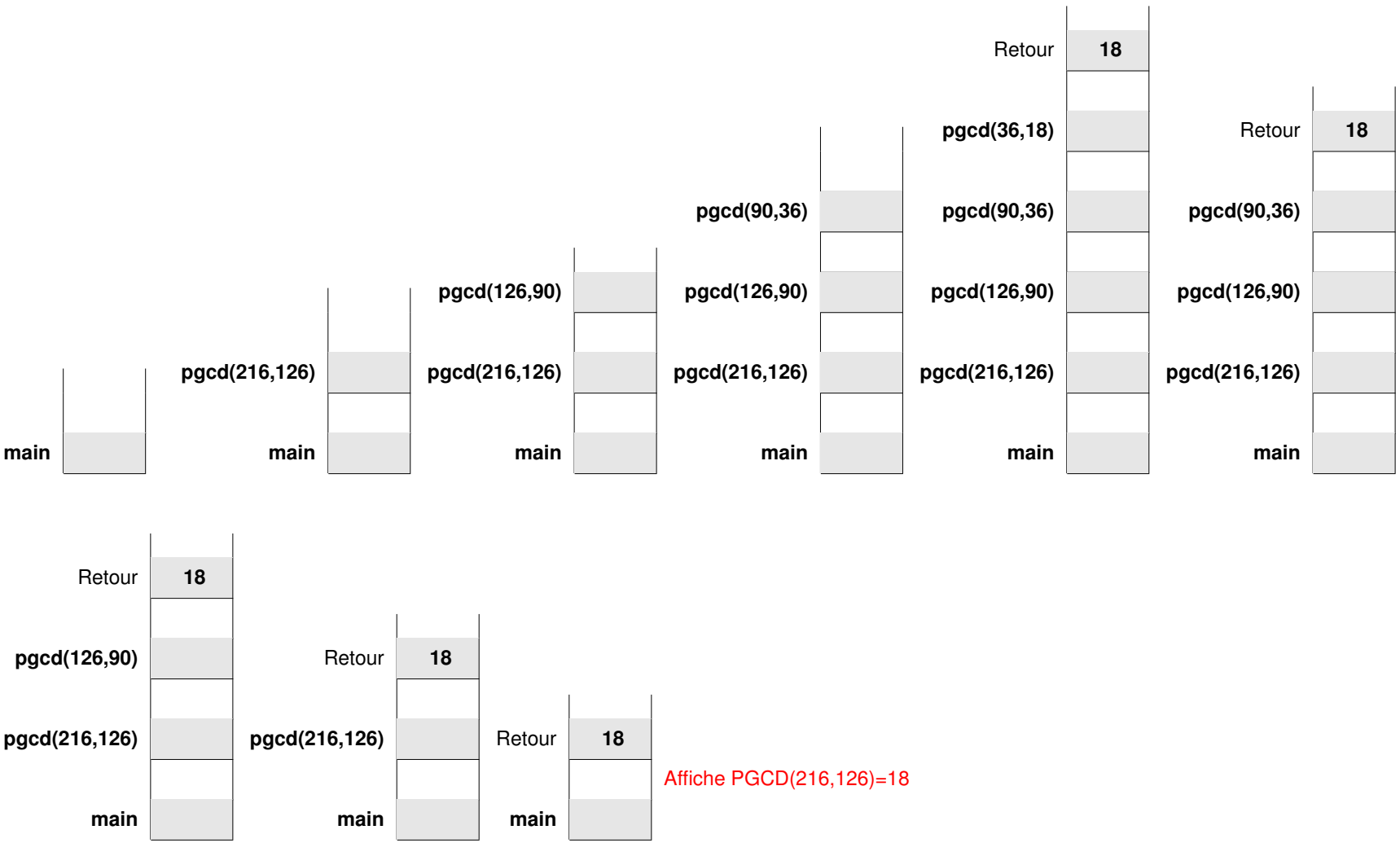
```
1 unsigned int pgcd (unsigned int m, unsigned int n)
2 { /* on suppose que m et n sont strictements positifs */
3   if (m % n == 0 ) /* si n divise m */
4     return n;
5   else
6     return pgcd(n, m % n);
7 }
```

On peut voir que l'appel récursif ne contient uniquement que l'appel récursif : l'appel intial renverra exactement le retour de l'appel du cas de base !

Suspendre les fonctions pendant les appels récursifs successifs est inutile.

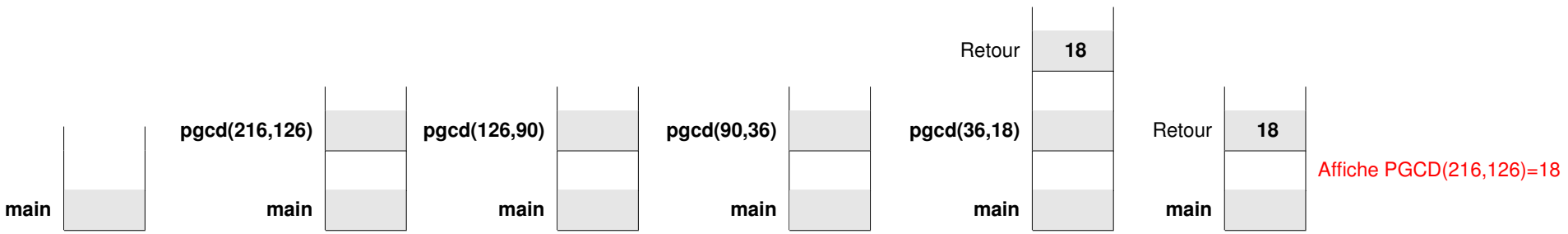
PGCD : récursivité classique

Pile d'appels lors des appels récursifs classiques :



PGCD : récursivité terminale

Pile d'appels lors des appels récursifs terminaux :



Récursivité terminale / non terminale

Optimisation possible

- Dans le cas d'une **récursivité terminale**, il n'est (théoriquement) **pas nécessaire d'empiler les appels** : l'appel récursif "remplace" simplement l'appel parent sur la pile (en passant les nouvelles valeurs des arguments et en conservant l'adresse de retour de l'appel initial).
- Le compilateur, dans certains contextes d'optimisation, peut donc **traiter une fonction récursive comme une fonction itérative** et optimiser la gestion de la pile (*tail-call optimization*).
- L'option -O2 du compilateur gcc active ce niveau d'optimisation (**sans aucune garantie** que les fonctions récursives qui peuvent être ainsi "dérécursivées" le seront effectivement).

Récursivité et efficacité : récursivité terminale

Comment écrire une version terminale

- Répérer la valeur retournée par la fonction et la mettre comme un deuxième paramètre de la fonction
- Ce deuxième paramètre sert d'accumulateur des valeurs calculées à chaque appel
- Le cas de base retourne alors la valeur "portée" par le deuxième paramètre
- Le cas général effectue le calcul et "l'envoie" à l'appel récursif par le deuxième paramètre.

Factorielle non terminale et terminale

Factorielle non terminale

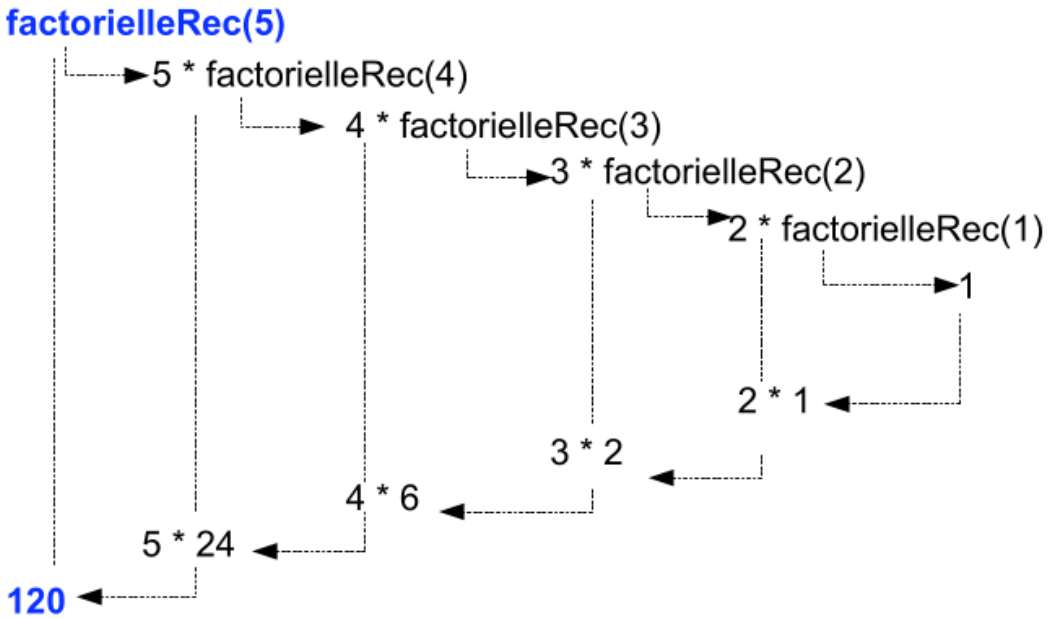
```
1 unsigned factorielleRec (unsigned n)
2 {
3   if ( (n == 0) || (n == 1) ) return 1;
4   return n * factorielleRec(n-1);
5 }
```

Factorielle terminale

```
1 unsigned factorielleTer2 (unsigned n, unsigned acc)
2 {
3   if (n <= 1)
4     return acc;
5   return factorielleTer2(n-1, n*acc);
6 }
```

Factorielle non terminale et terminale

Factorielle non terminale



Factorielle terminale

factorielleTer2(5,1) = factorielleTer2(4,5) = factorielleTer2(3,20)
= factorielleTer2(2,60) = factorielleTer2(1,120) = 120

(On peut remarquer que les calculs sont faits “dans un ordre inverse” : 5*4*3*2*1)

Factorielle non terminale et terminale

On peut noter que le deuxième paramètre est “technique”

- il doit toujours être initialisé à 1
- il est une souce d’erreur si on le propose à un utilisateur

On ajoute une fonction dite “interface” qui cache à l’utilisateur ce deuxième paramètre : son but unique est d’appeler la fonction récursive terminale avec ses 2 paramètres.

“Interface” pour factorielle terminale

```
1 unsigned factorielleTer(unsigned n)
2 {
3   return factorielleTer2(n, 1);
4 }
```

Récursivité et efficacité : récursivité terminale

Si la fonction a deux appels récursifs de paramètres n-1 et n-2

- Il faut un paramètres pour le calcul (comme dans l'exemple précédent)
- Et un deuxième paramètre pour faire "passer" la sauvegarde du calcul précédent à l'appel récursif
- Ainsi un des paramètre est la valeur n et l'autre n+1 : on initialise en appelant avec les deux valeurs initiales 0 et 1

Récursivité et efficacité : récursivité terminale

Fibonacci non terminale

```
1 unsigned int fib (unsigned int n)
2 {
3   if (n == 0 || n == 1)
4     return n;
5   return fib(n-1) + fib(n-2);
6 }
```

Fibonacci récursif terminal

```
1  unsigned fib2_ter (unsigned n, unsigned a, unsigned b)
2  {
3    if (n == 0)
4      return a;
5    if (n == 1)
6      return b;
7    return fib2_ter(n-1, b, a+b);
8  }
9
10 unsigned fib_ter (unsigned n)
11 {
12   return fib2_ter(n, 0, 1);
13 }
```

Appels récursifs pour Fibonnaci terminal

Appels pour Fibonacci terminal

fib_ter(7)	= fib2_ter(7,0,1)	= fib2_ter(6,1,1)
	= fib2_ter(5,1,2)	= fib2_ter(4,2,3)
	= fib2_ter(3,3,5)	= fib2_ter(2,5,8)
	= fib2_ter(1,8,13)	= 13

On peut remarquer que

- le premier paramètre est un compteur, notons le **cpt**, remontant les appels récursifs jusq'au cas de base
- le deuxième paramètre contient le terme fib(7-cpt)
- le troisième paramètre contient le terme fib(7-cpt+1)
- donc le cas de base où **cpt=1** a en troisième paramètre le terme fic(cpt).

Conclusion

Conlusion

- Programmer récursivement
  - est souvent très naturel et simple
  - donne des fonctions qui sont parfois plus rapides que la première idée impérative
- Mais coder récursivement peut faire exploser la mémoire.
  Pour réparer cela
  - on peut mémoïser ou rendre terminale une fonction récursive
  - cela n'accélère pas la vitesse du programme (ou très peu) mais permet de ne pas retenir inutilement en mémoire des appels de fonctions.