Université Mohammed V de Rabat

ENSIAS

Pr. A. EL FAKER

STRUCTURES DE DONNÉES

Plan général

1. **Compléments**
   - *Complexité, programmation dynamique*
2. **Listes**
   - *Chaînage des données*
3. **Arbres binaires de recherche**
   - *Recherche logarithmique*
4. **Arbres AVL**
   - *Garder l'équilibre*
5. **Fichiers indexé et B-arbres**
   - *Organisation des données en BD*
6. **Backtracking et Problèmes combinatoires**
7. **Graphes**
   - *Algorithmes de recherche*

Leçon 1 : complément

- **Complexité temporelle**

- **Programmation dynamique**
  - **Allocation automatique et**
  - **Allocation programmée**

Université Mohammed V de Rabat

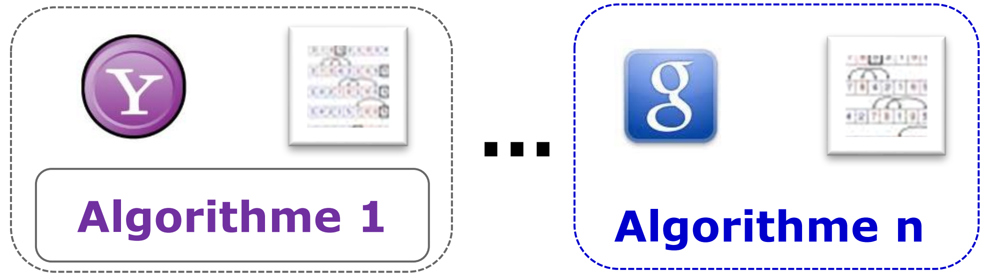
ENSIAS

Complément

Complexité temporelle

Complexité temporelle

- **Pour un même problème plusieurs algorithmes peuvent exister**



- **Quel est le « meilleur » algorithme**
  - **Cet algorithme est-il « faisable» ?**
  - **Combien de « temps » demande t-il ?**
  - **Combien d' « espace» demande t-il ?**

Complexité temporelle

A:	Mon algo. peut trier 10^6 nombres en 3s
B:	Mon algo. peut trier 10^6 nombres en 5s

Complexité temporelle

A:	Mon algo. peut trier 10^6 nombres en 3s
B:	Mon algo. peut trier 10^6 nombres en 5s
A:	Je viens de le tester sur mon Intel® Core™ i7
B:	Ce résultat date de mes années d'études (*1985*)

Complexité temporelle

➔ **Mesurer le temps en secondes ?**

mauvaise mesure ! !

(+) Utile en pratique

(−) Dépend de l'implémentation :

Langage/Compilateur/Processeur

A:	Mon algo. peut trier 10^6 nombres en 3s
B:	Mon algo. peut trier 10^6 nombres en 5s

Complexité temporelle

➔ **Mesurer le temps en secondes ?** **mauvaise mesure ! !**

➔ **Mesurer quoi ?**

- *Nombre d'opérations caractéristiques*
  - Nombre de multiplications en mathématiques
  - Nombre de comparaisons en recherche, … etc.

(+) *Dépend uniquement de l'algorithme*

Complexité temporelle

- **Mesurer le temps en secondes ?** **mauvaise mesure ! !**
- **Mesurer quoi ?**
  - *Nombre d'opérations caractéristiques*
- **En fonction de quoi ?**

$$\boxed{Complexité = f(n)}$$

n = taille des données

– Nombre d'élément dans un tableau liste

t

– Nombre de bits représentant une donnée …

1	1	0	0	1

Complexité temporelle

b^n : direct

```
1. p <- 1
2. Pour i=1,…,n répéter
   2.1. Multiplier p par b
3. Terminer avec réponse p
```

Efficacité ? Nombre de multiplications ~ n

```
1.  …
2.  n fois
   2.1 Une multiplication
3.  …
```

Complexité temporelle

b^n : direct

- Temps approximativement proportionnel à **n**
- On écrit : $T(n) = O(n)$

Complexité en temps est d'ordre **n**

- Même ordre de complexité que :

n+2, 3n+4, …

Complexité temporelle

b^n : rapide

Idée : Si on connait b^{500} on peut calculer $b^{1000} = b^{500} \times b^{500}$

avec **une multiplication** en plus

$$
\begin{aligned}
b^n &= 1 && \text{si } n = 0 \\
b^n &= b^{n/2} \times b^{n/2} && \text{si } n > 0 \text{ et } n \textit{ est paire} \\
b^n &= b \times b^{n/2} \times b^{n/2} && \text{si } n > 0 \text{ et } n \text{ est impaire}
\end{aligned}
$$

Complexité temporelle

b^n : rapide

*Complexité **dans le pire des cas**?*

$$b^n = 1 \quad \text{si } n = 0$$

$$b^n = b^{n/2} \times b^{n/2} \quad \text{si } n > 0 \text{ et } n \text{ est paire}$$

$$b^n = b \times b^{n/2} \times b^{n/2} \quad \text{si } n > 0 \text{ et } n \text{ est impaire}$$

$n/2$ entier

2 multiplications de plus (max)

$$T(n) \leq 2 + T\left(\frac{n}{2}\right)$$

$$T(n) \leq 2 + 2 + T\left(\frac{n}{2^2}\right)$$

$$\leq 2 + 2 + \cdots + 2 + T\left(\frac{n}{2^p}\right)$$

Il existe p tel que :

$$2^p \leq n < 2^{p+1}$$

$$1 \leq \frac{n}{2^p} < 2 \qquad p = \lfloor log_2(n) \rfloor$$

$$\boxed{T(n) \leq 2\lfloor log_2(n) \rfloor + 1}$$

Complexité temporelle

b^n : rapide

Complexité **dans le pire des cas**?

- *Au Maximum*
- *Négliger le terme de croissance lente*
- *Négliger le facteur constant*
- *Négliger ⌊ ⌋ : -0,5 en moyenne*

$2\lfloor \log_2(n) \rfloor + 1$ multiplications

$2\lfloor \log_2(n) \rfloor$

$\lfloor \log_2(n) \rfloor$

$\log_2(n)$

T(n)= O(*log* n)

Complexité temporelle

b^n : directe — Complexité en temps **linéaire**

$$b^n = 1 \quad \text{si } n = 0$$
$$b^n = b \times \ldots \times b \quad n \text{ fois}$$

T(n)= O(n)

b^n : rapide — Complexité en temps **logarithmique**

$$b^n = 1 \quad \text{si } n = 0$$
$$b^n = b^{n/2} \times b^{n/2} \quad \text{si } n > 0 \text{ et } n \text{ est paire}$$
$$b^n = b \times b^{n/2} \times b^{n/2} \quad \text{si } n > 0 \text{ et } n \text{ est impaire}$$

T(n)= O(log n)

n = 24 000

⇨ Directe : 24000 mult. ~180sec

⇨ Rapide : 14 mult.~0.2 sec

Complexité temporelle

QuickSort

Complexité ***dans le cas moyen*** *?*

```
quicksort(int *t,int debut,int fin){
   int i;
   if (fin>debut) {
      i = partition(debut,fin);
      quicksort(t,debut,i-1);
      quicksort(t,i+1,fin);
   }
}
```

Coût de la partition()

$$
\begin{aligned}
T(n) &= n + 2 * T(n/2) \\
&= n + 2(n/2 + 2T(n/2^2)) \\
&= n + n + 2^2T(n/2^2) \\
&= \underbrace{n+\ldots+n}_{p} + \underbrace{2^pT(n/2^p)}_{0} \\
&= np
\end{aligned}
$$

$$T(n) = O(\mathbf{n.\log(n)})$$

3	10	17	5	23	4	9	7	26	8	13	12	11

i

3	10	8	5	7	4	9	11	26	17	13	12	23

Complexités usuelles

Complexité		Algorithme	
$O(1)$	Temps **constant**	*Opérations base*	*Réalisable*
$O(\log n)$	Temps **logarithmique**	***Recherche Binaire***	*Réalisable*
$O(n)$	Temps **linéaire**	*Recherche Linéaire*	*Réalisable*
$O(n \log n)$	Temps **log linéaire**	***Tri rapide***	*Réalisable*
$O(n^2)$	Temps **quadratique**	*Tri ordinaire*	*Parfois irréalisable*
$O(n^3)$	Temps **cubique**	*MxN*	*Parfois irréalisable*
$O(2^n)$	Temps **exponentiel**	*Tous de Hanoi*	*Rarement réalisable*

Tours de Hanoï

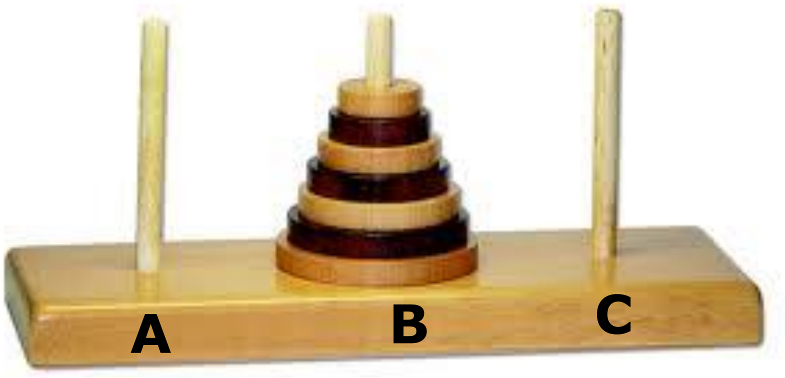
Université Mohammed V de Rabat

Récursivité : Tours de Hanoï

ENSIAS

Légende

***64** disques à déplacer*



3 tours

2 règles

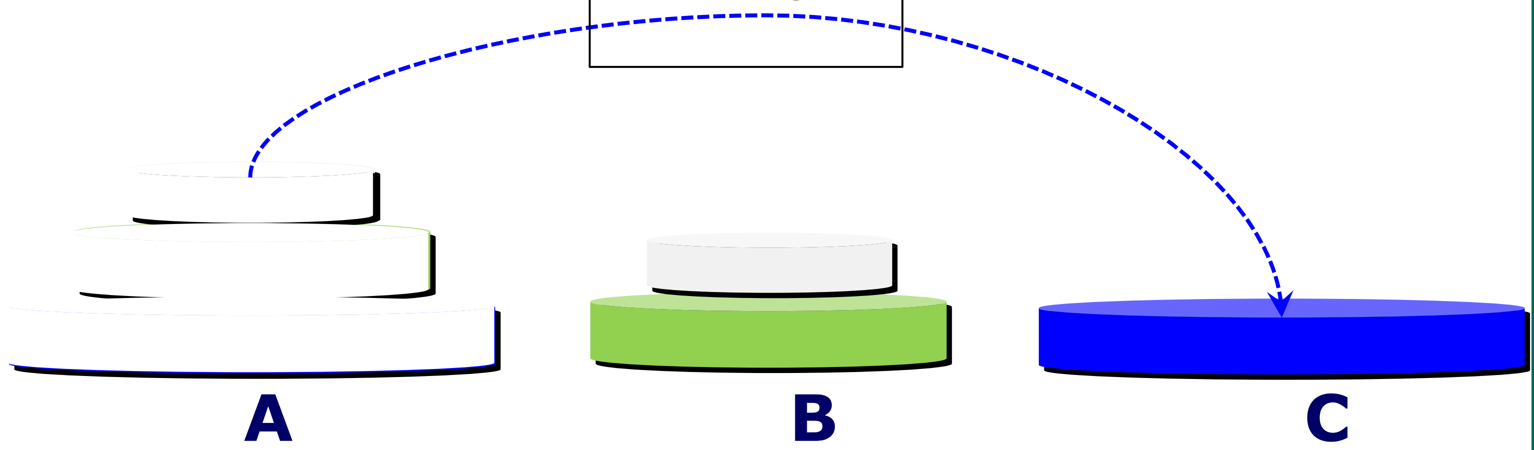
1 Ne déplacer qu'un disque à la fois

2 Un disque doit être empilé sur un autre de dimension supérieure

Récursivité : Tours de Hanoï

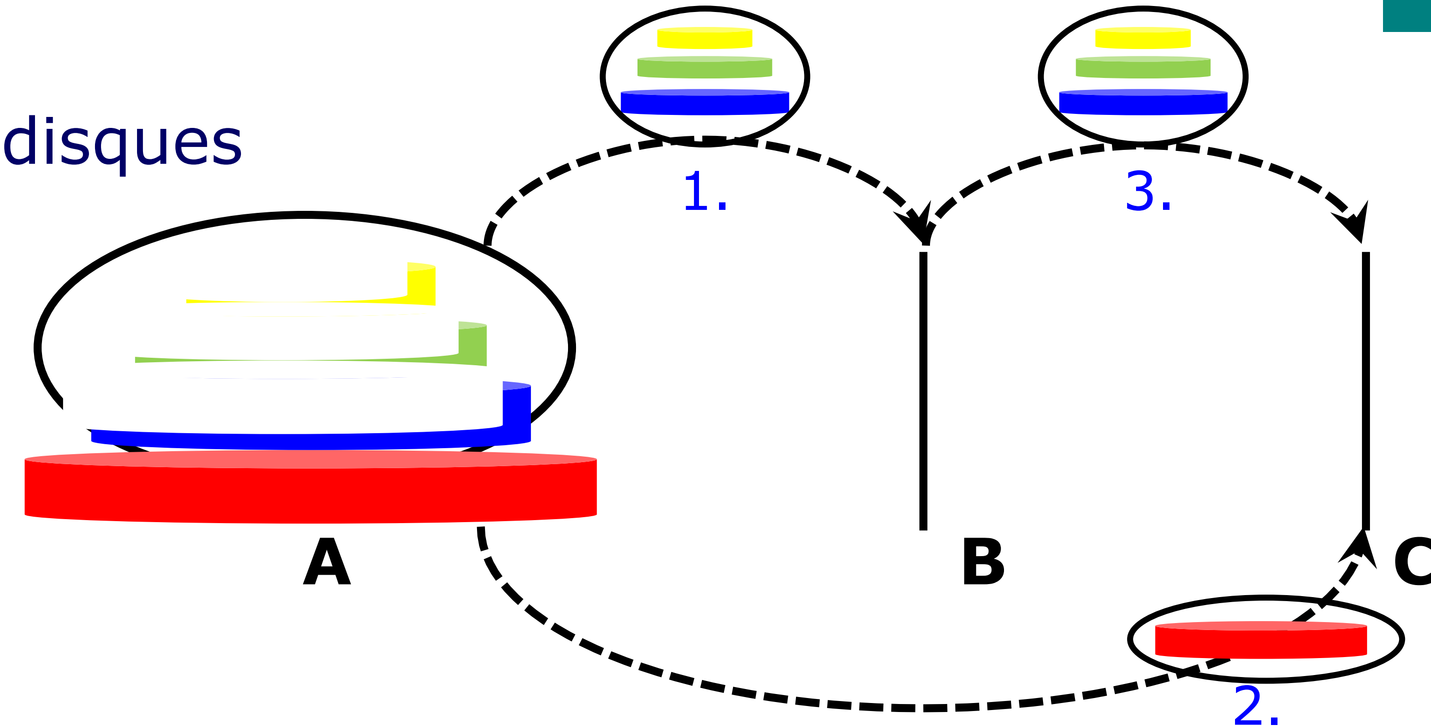
A->C
A->B
C->B
A->C
B->A
B->C
A->C

Avec 3 disques



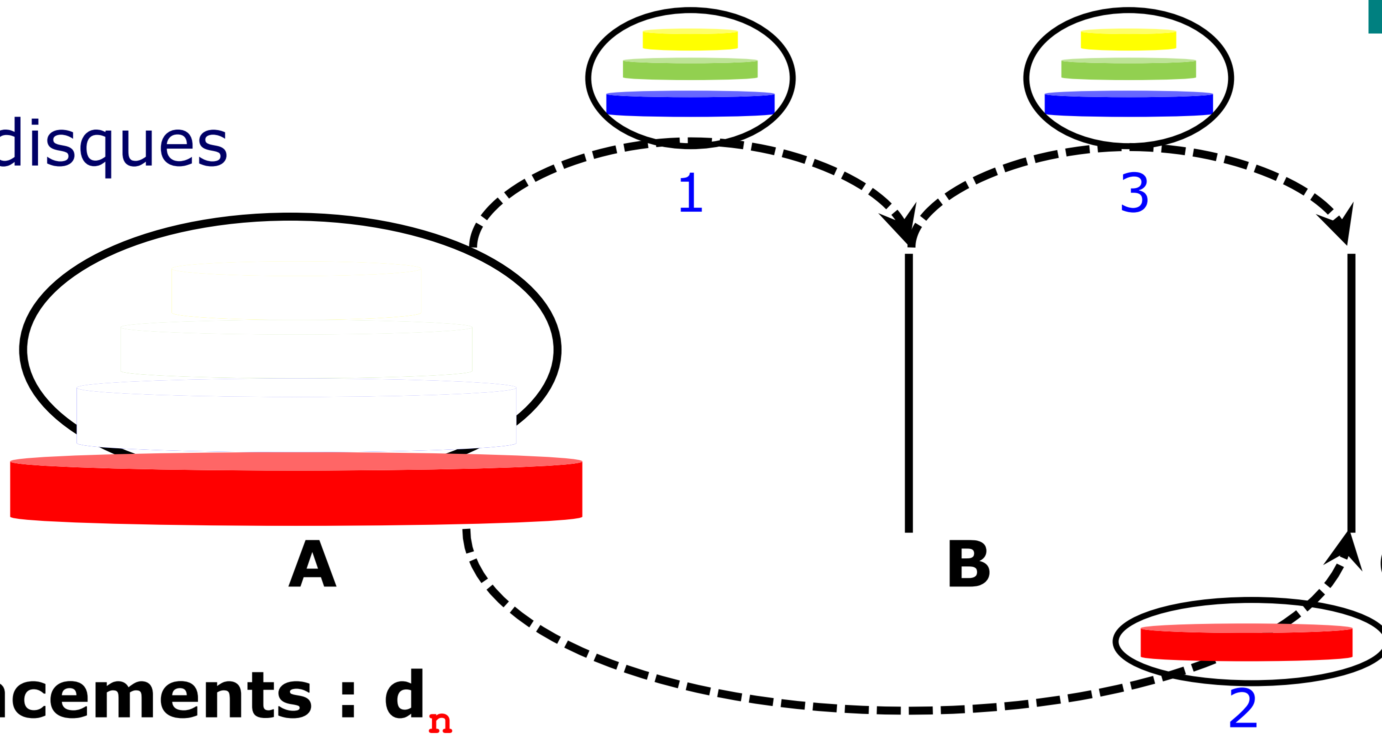
Récursivité : Tours de Hanoï

Avec n disques



1. déplacer **n-1** de **A** vers **B** (C)
2. déplacer **1** de **A** vers **C**
3. déplacer **n-1** de **B** vers **C** (A)

Avec n disques



→ **Déplacements : d_n**

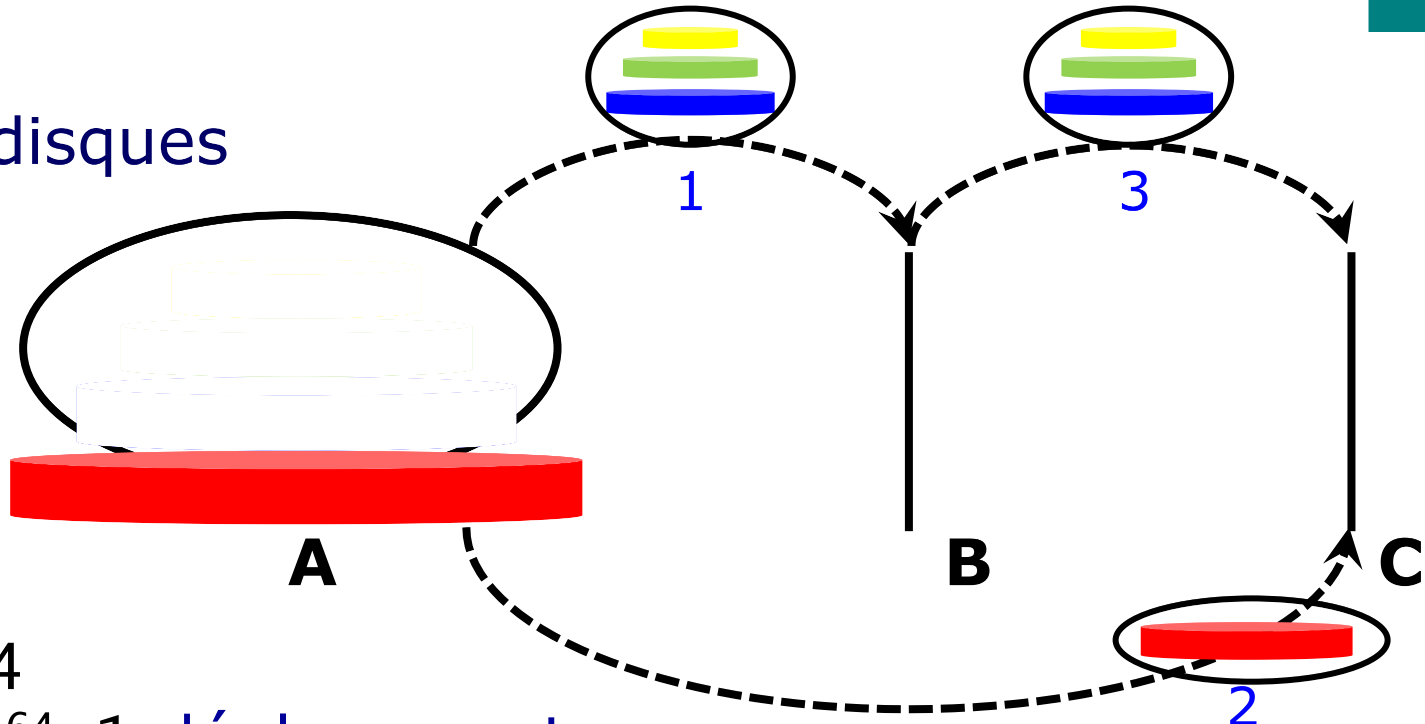
1. d_{n-1}
2. 1
3. d_{n-1}

$$\boxed{d_n = 2d_{n-1}+1 = 2^n-1}$$

• Déplacements ?

Récursivité : Tours de Hanoï

Avec n disques



→ n=64

2^{64}-1 déplacements

2^{64} secondes ≈ 585 milliards d'années

→ Âge de l'univers estimé à 15 milliards d'années

Il y a de la marge !!

Exercice

- Que calcule la fonction suivante ?

f(n)= <u>si</u> n<0 <u>alors</u> 0 <u>sinon</u> f(n-1)-f(n-1)

avec n entier

$$f \equiv 0$$

- Quelle est sa complexité ?

Exercice

- Que calcule la fonction suivante ?

```
f(n)= si n<0 alors 0 sinon f(n-1)-f(n-1)
avec n entier
```

f ≡ 0

- Quelle est sa complexité ?

$$T(n) = O(\mathbf{2}^{n})$$

Exercice

E ensemble de n nombres et s un entier cible

∃? deux éléments x et y de E tq x + y= s

- Complexité ? $O(n^2)$
- Proposez un algorithme en $O(n \log n)$

	0	1						n-2	n-1
E	56	13	9	24	12	6	8	7	4

s	32

Exercice

E ensemble de n nombres et s un entier cible

∃? deux éléments x et y de E tq x + y= s

1. trier E

0	1						n-2	n-1
4	6	7	8	9	12	13	24	56

S **32**

en O(**n log n**)

2. Recherche dans E :

Pour tout(y dans E)

chercher si s-y appartient à E (trié) par dichotomie :

en O(log n)

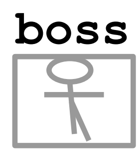
en O(**n log n**)

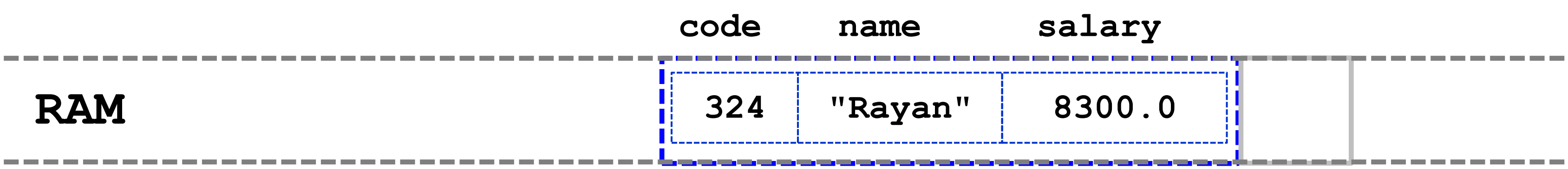
Programmation dynamique

Structures (rappel)

➔ Type utilisateur

```
employee boss;
```



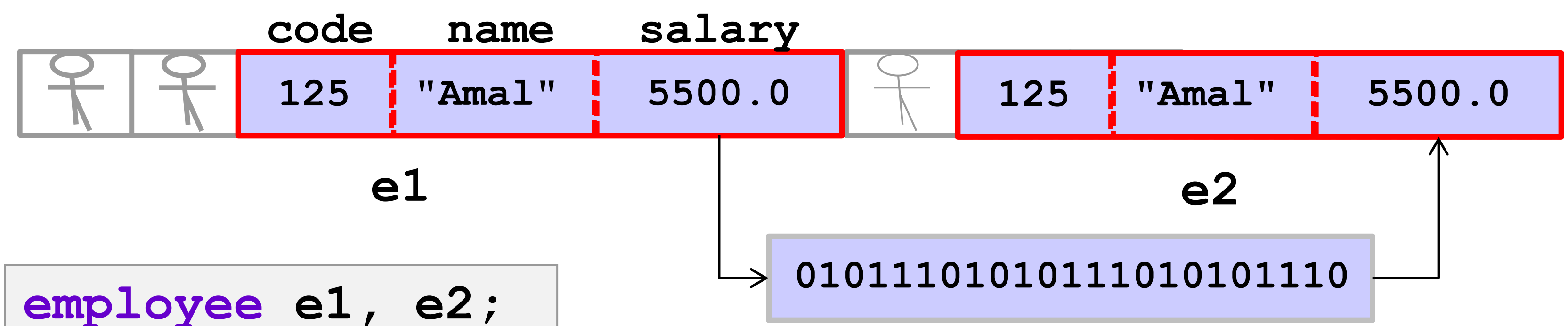



```
boss.salary = 9300.0;
```

```
typedef struct _E {
        int     code;
        char    name[25];
        float   salary;
} employee;
```

Structures : Affectation

➔ **L'affectation des structures : c'est possible**



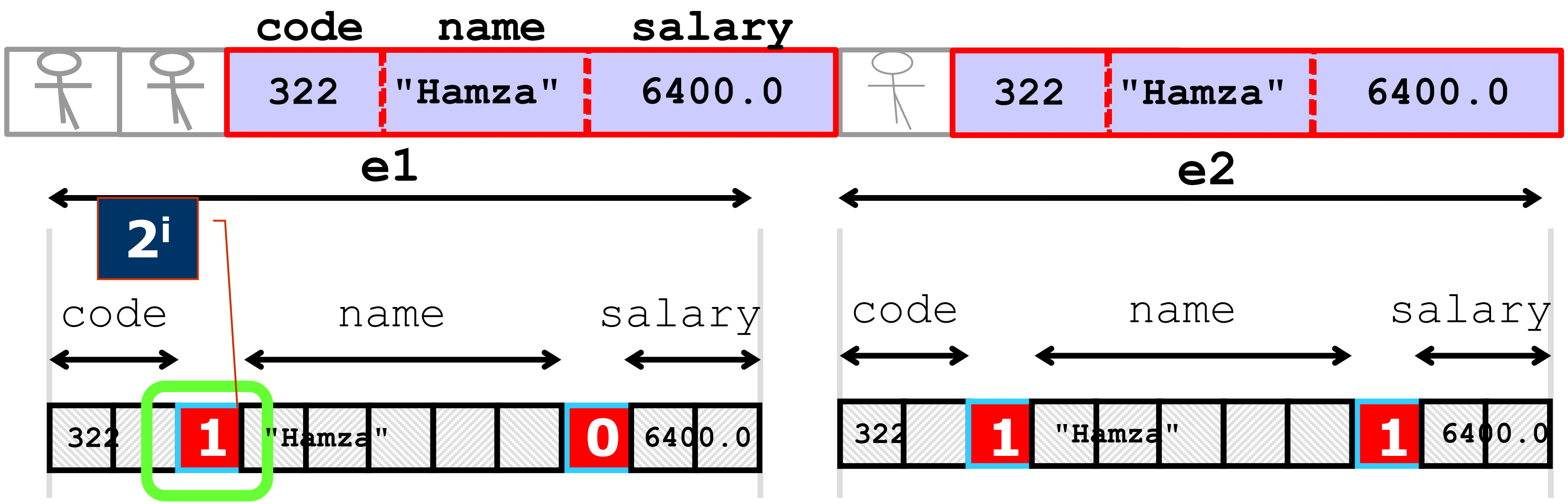

```c
employee e1, e2;
```

```c
e1.code = 125;
e1.salary = 5500.0;
strcpy(e1.name, "Amal");
```

```c
e2 = e1 ;                    // Copie binaire
```

Structures : Affectation

→ **La comparaison des structures : pas possible**




```
if(e1 == e2) { //erreur }
```

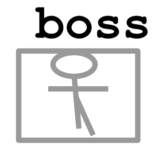
Contraintes d'alignement

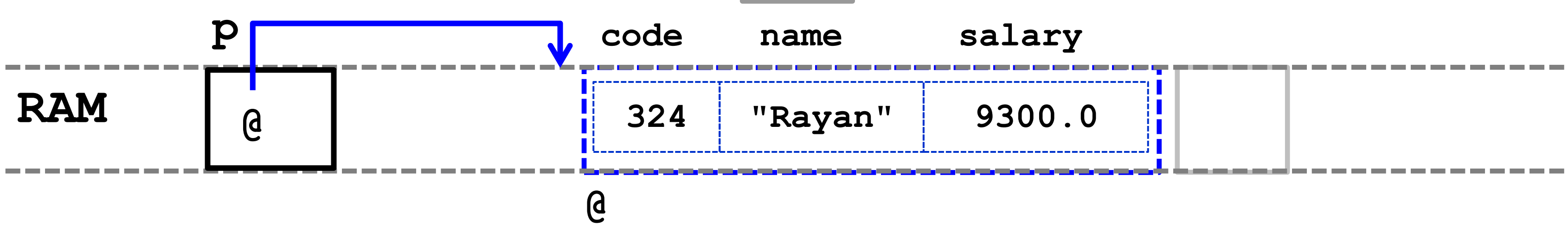
- Pour optimiser le temps d'accès mémoire
- Des compilateurs laissent des « trous »

Pointeurs de structures

➔ Accès aux champs par adresse

```
employee boss;
```






```
employee * p = & boss;
```

p pointe le début de la structure

Notation fléchée

```
(*p).salary = 9300.0;
```

```
p -> salary = 9300.0;
```

Augmenter le salaire

```
p -> salary = p -> salary + 2000.0;
```

Allocation automatique

- Au moment de **l'activation** d'une fonction

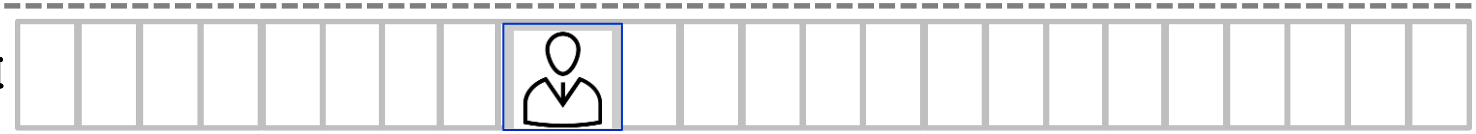
```
void function(){
  employee boss;
  //corps de la fonction
  return;
}
```

Mémoire allouée :

- *Par le **compilateur***
- *De manière **statique***

- A la **fin** de la fonction, la libération est automatique

boss

RAM 

Static Area **libre**

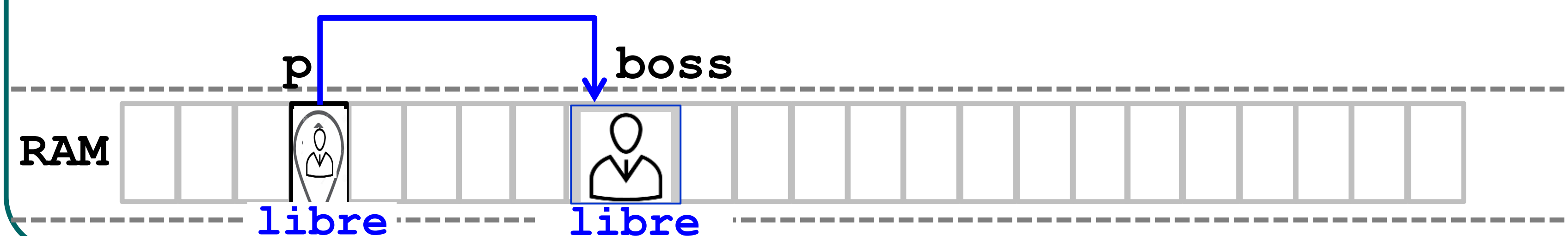
Allocation automatique

- Au moment de **l'activation** d'une fonction

```
void function(){
  employee * p;
  employee boss;
  p = &boss;
  //corps de la fonction
  return;
}
```

- **p** : pointeur
  Pas d'employé pointé
- **boss** : employé
  *Mémoire **allouée***
- **p** pointe l'employé
  boss

- A la **fin** de la fonction, la libération est automatique



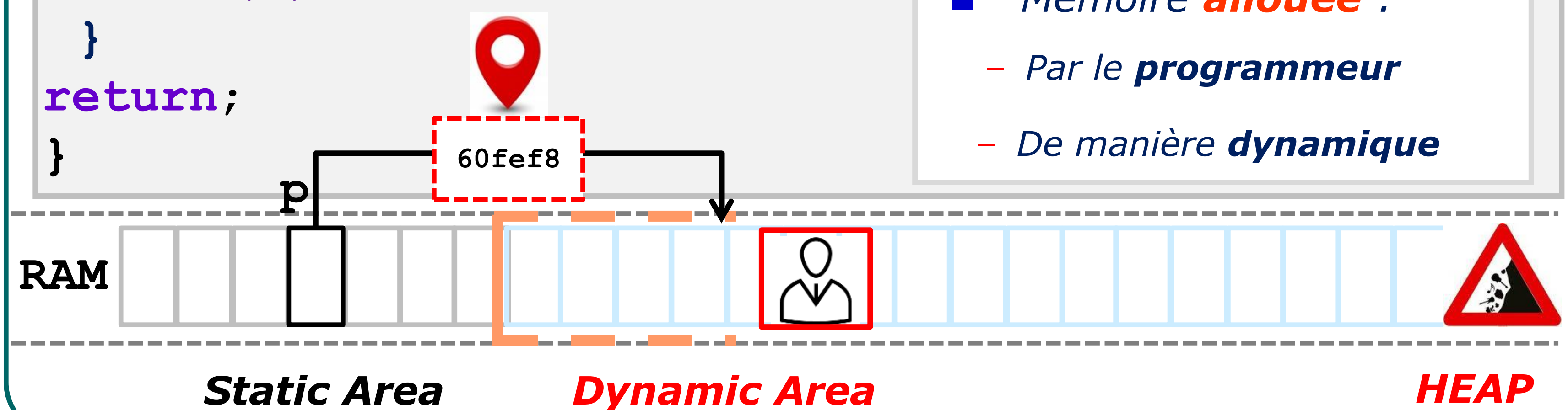
Allocation programmée

```c
#include <alloc.h> //(void  *) malloc(int size);
```

```c
void function(){
  employee * p ;
  p = malloc(sizeof(employee) );
  if(p == NULL) {
   printf(("allocation memoire impossible\n");
   exit(1);
  }
return;
}
```

- *Memory Allocation*
- *Pas besoin de* **boss**

- *Mémoire* ***allouée*** *:*
  - *Par le* ***programmeur***
  - *De manière* ***dynamique***

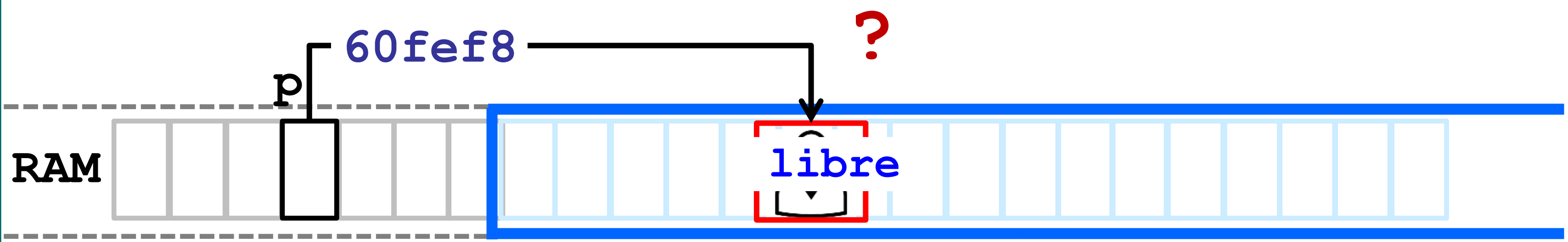


➔ Libération dynamique de la mémoire

```
void free(p);
```

Rend au système la zone pointée par `p`

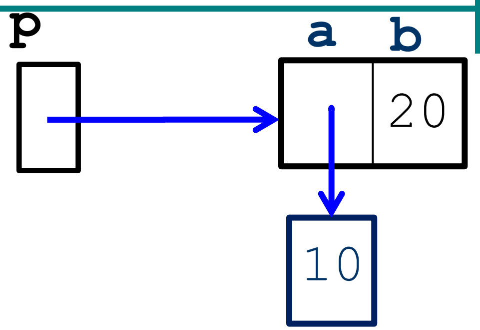
- *La zone devient libre*
- *`p` ne change pas de valeur (passage par valeur)*



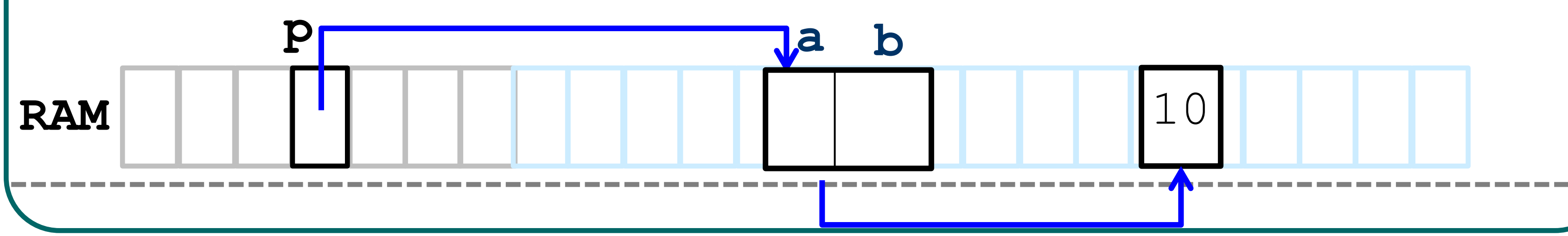
- *Pas de fonction* `free()` ***"automatique" ... GC***

Allocation programmée

Exercice 1

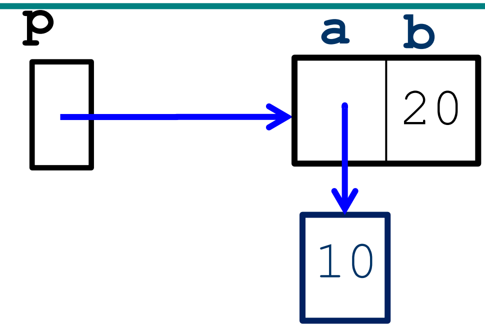


Ecrire les instructions nécessaires pour construire cette structure en mémoire



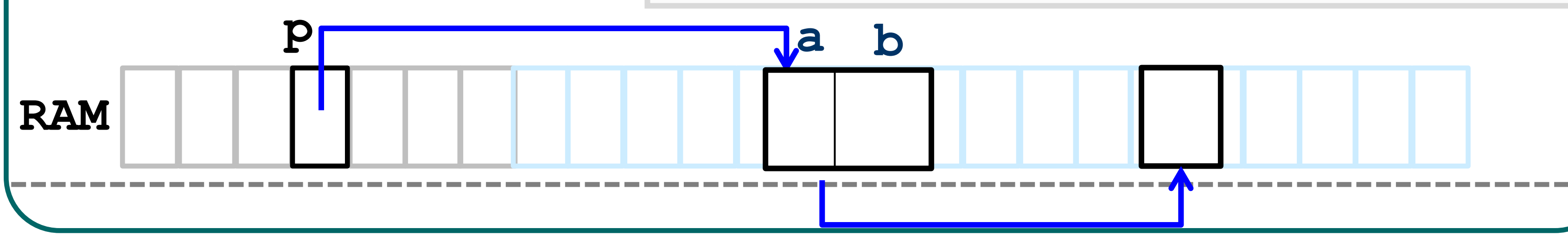
Allocation programmée

Etapes



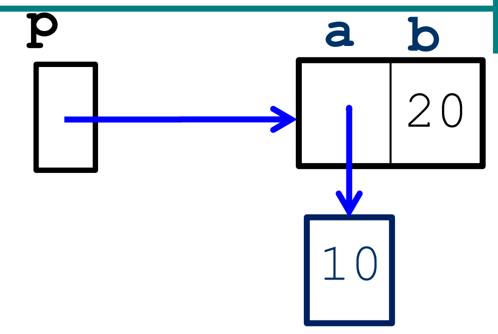
- Déclarer la <u>structure</u> et le <u>pointeur</u>

```
typedef struct{
    int * a;
    int   b;
} st;

st * p ;
```



Allocation programmée

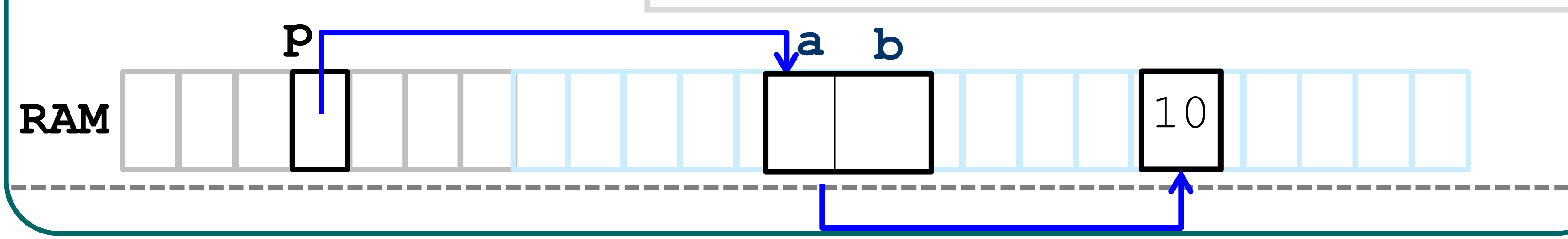
Etapes



- Déclarer la structure et le pointeur
- Allouer l'espace mémoire

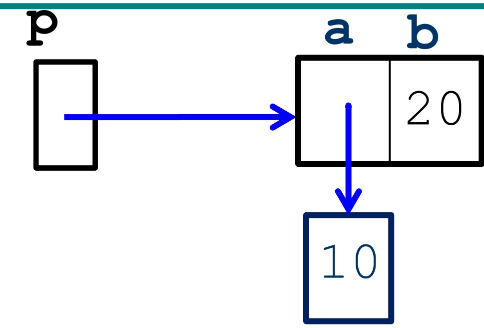
```
p = malloc(sizeof(st));

p -> a = malloc(sizeof(int));
```

Erreur sinon



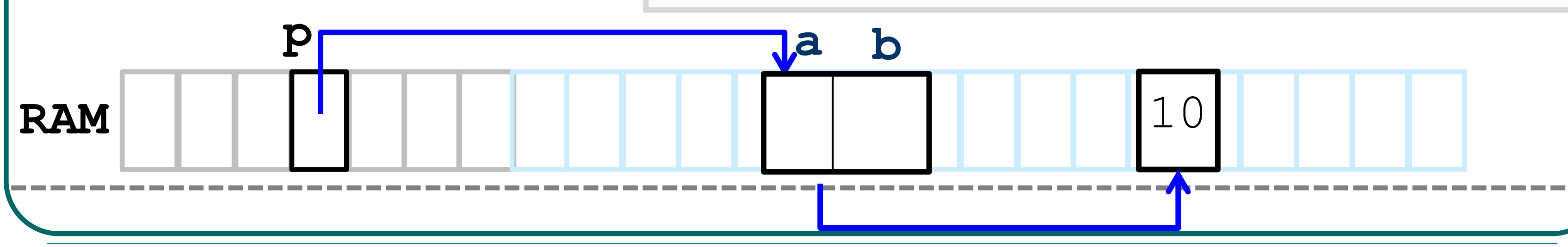
Allocation programmée

Etapes



- Déclarer la <u>structure</u> et le <u>pointeur</u>
- Allouer l'espace mémoire
- Remplir les champs

```
p -> b  = 20;

~~p -> a  = 10;~~

*(p -> a) = 10;
```

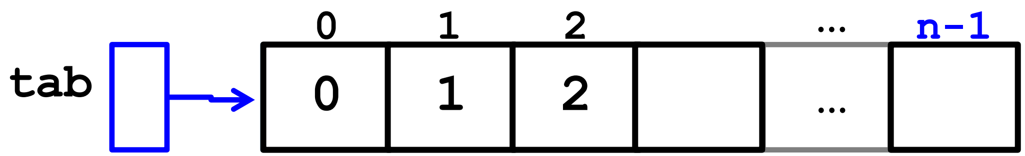


Allocation programmée

Exercice 2 : tableaux dynamiques

Remplir le tableau de taille n comme suit.

n est donnée par l'utilisateur.

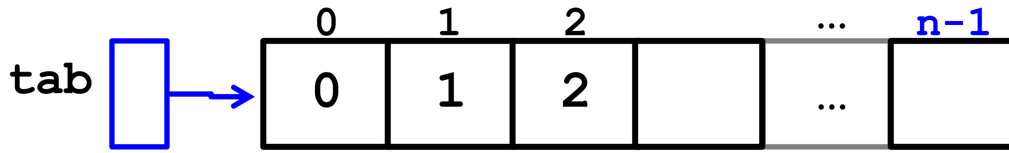


C:\develop\C\ProgDynamique2\bin\Debug\ProgDynamique2.exe

```
 Array size ? 10
 Successful memory allocation
 The elements of the array are: 0 1 2 3 4 5 6 7 8 9
Process returned 0 (0x0)   execution time : 10.678 s
Press any key to continue.
```

Allocation programmée

Exercice 2




```c
int main(void){
  int *tab; int i, n;
  printf(" Array size ? "); scanf("%d",&n);
  tab = malloc(n*sizeof(int));
```

C:\develop\C\ProgDynamique2\bin\Debug\ProgDynamique2.exe

```
 Array size ? 10
 Successful memory allocation
 The elements of the array are: 0 1 2 3 4 5 6 7 8 9
Process returned 0 (0x0)   execution time : 10.678 s
Press any key to continue.
```

Allocation programmée

Exercice 2

tab →

0	1	2		…	n-1
0	1	2		…	

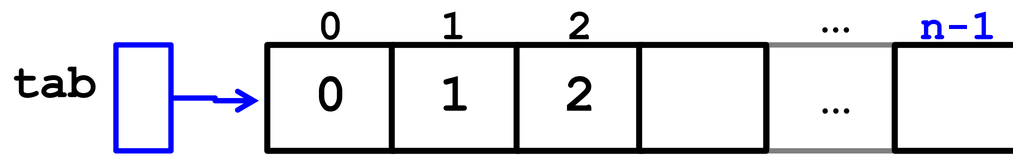
```c
int main(void){
  int *tab; int i, n;
  printf(" Array size ? "); scanf("%d",&n);
  tab = malloc(n*sizeof(int));
```

C:\develop\C\ProgDynamique2\bin\Debug\ProgDynamique2.exe

```
Array size ? 1000000000000
Error : Memory allocation

Process returned 0 (0x0)   execution time : 3.662 s
Press any key to continue.
```

Allocation programmée

Exercice 2




```c
int main(void){
  int *tab; int i, n;
  printf(" Array size ? "); scanf("%d",&n);
  tab = malloc(n*sizeof(int));
  if (tab == NULL) {
     printf(" Error : Memory allocation\n"); exit(0);
  } else printf(" Successful memory allocation \n");
   // Memory allocation was successful
  // Print the elements
```

Allocation programmée

Exercice 2

```
C:\develop\C\ProgDynamique2\bin\Debug\ProgDynamique2.exe
 Array size ? 10
 Successful memory allocation
 The elements of the array are: 0 1 2 3 4 5 6 7 8 9
Process returned 0 (0x0)   execution time : 10.678 s
Press any key to continue.
```

```c
int main(void){
   ...
  // Print the elements
  for(int i=0 ; i < n ; i++)tab[i] = i ;
  printf(" The elements of the array are: ");
  for(int i=0 ; i < n ; i++){
     printf("%d ", *(tab+i)); //or tab[i];
  }
  return 0;
}
```

STRUCTURES DE DONNÉES