

STRUCTURES DE DONNÉES ET ALGORITHMES FONDAMENTAUX

Algorithmique - Programmation – Langages 2
(APL2)

MODALITÉS DU COURS

- Volume d'enseignement : 3 CM, 7 TD, 9 TP
- Évaluation :
 - 1 note moyenne de TD
 - Test au début de chaque TD (hormis #1)
 - 10 min, 1 question cours, 1 exercice du TD précédent
 - 1 note moyenne de TP
 - Chaque TP est noté (hormis #1), à rendre pour le lendemain (8h)
 - 1 évaluation sur table (2h) : le 20 janvier
- Langage (TP) : C → les indices de tableaux commenceront à 0

OBJECTIFS

Structures de données :

stocker et manipuler plusieurs (un grand nombre) d'éléments d'un même type

1. Impact sur la performance
2. Algorithmes spécifiques
3. Implantation en C

Une structure de base : la liste chaînée

PLAN DU MODULE

CM#1 (rappels +)

Complexité (principe de mesure, ordre polynomial) ; Tableaux (rappels, impact sur la mémoire)

CM#2

Type abstrait de données (ADT, exemple de l'ensemble) ; Listes chaînées (pile, queue, liste circulaire, liste doublement chaînée ; algorithmes de base)

CM#3

Arbres binaires ; Tableaux associatifs ; Algorithmes de base

TD x7

Exercices sur ces notions

TP x9

Implantation en C ; Initiation au C

COÛT D'UN ALGORITHME

Un algorithme applique des opérations sur des données

Son exécution consomme donc des ressources

- En temps CPU pour les opérations
→ complexité en temps
- En mémoire pour le stockage des données
→ complexité en mémoire

EXEMPLE #1

Problème :

déterminer la parité d'un nombre entier n

Algorithme : (test sur le bit de poids faible)

```
Si  $n \& 1 = 0$ , alors  
    renvoie Vrai,  
sinon  
    renvoie Faux  
fin si
```

Complexité en temps :

1 opération et 1 test pour tout n (et renvoie ?) $\rightarrow k$ (=2 ou ...)
 k ne dépend pas de la valeur de n

EXEMPLE #2

Problème :

Calculer la factorielle d'un nombre entier n ($n! = n * (n-1) * (n-2) * \dots * 2 * 1$)

Algorithme :

Sortie : f : entier

$f \leftarrow 1$

pour i allant de 1 à n , faire

$f \leftarrow f * i$

fin pour

renvoie f

Complexité en temps :

n multiplications et n affectations (+ opérations sur $i \dots$) $\rightarrow k * n$ ($k=2, 3$ ou...)

PREMIÈRE CONCLUSION

La complexité peut dépendre de la valeur du paramètre d'un algorithme.

Le nombre exact d'opérations importe peu : ordre de grandeur (constant, linéaire...).

EXEMPLE #3

Problème :

Extraire le premier élément d'un ensemble indexé

Entrée : `tab[n]` : réels, `n` : entier

Algorithme :

Sortie : réel

`renvoie tab[0]`

Complexité en temps :

1 opération (?)

ne dépend pas de la longueur `n` (nombre de données)

EXEMPLE #4

Problème :

Extraire le plus grand élément d'un ensemble indexé

Entrée : `tab[n]` : réels, `n` : entier

Algorithme :

Sortie : `max` : réel ; **VI** : `i` entier

```
max ← tab[0]
```

```
pour i allant de 1 à n-1, faire
```

```
    si tab[i] > max, alors
```

```
        max ← tab[i]
```

```
    fin si
```

```
fin pour
```

```
renvoie max
```

Complexité en temps :

Potentiellement n tests et affectations (pire cas : ensemble trié de manière croissante)

DEUXIÈME CONCLUSION

La complexité peut dépendre du nombre de données passées en paramètre (taille de l'ensemble).

On s'intéresse ici à la complexité dans le pire cas.

POURQUOI MESURER LA COMPLEXITÉ ?

Dépendance à « n »

Valeur de paramètre ou nombre de données

Évaluer le comportement d'un algorithme

Comment évolue son temps d'exécution ?

Comparer deux algorithmes

Deux algorithmes résolvent le même problème : lequel choisir ? Lequel prend le plus de temps d'exécution

Impact sensible quand le temps devient important

ordre de grandeur pour des grandes valeurs de n

EXEMPLE #2 : DÉTAILS DU CALCUL

Problème :

Calculer la factorielle d'un nombre entier n

Algorithme :

Sortie : f : entier

```
1 f ← 1
n pour i allant de 1 à n, faire
  2 f ← f * i
  fin pour
renvoie f
```

Règles

- Commencer par « l'intérieur »



- Remonter vers « l'extérieur »



Complexité :

$$C(n) = 1 + n*2 = 2*n+1$$

$$\text{Mais : } n*2 = \underbrace{2+2+\dots+2}_{n \text{ fois}} = \sum_{i=1}^n 2$$

$$\text{Donc } C(n) = \sum_{i=1}^n 2+1$$

→ boucles du code ↔ sommes sur même intervalle

LIEN BOUCLE – SOMME : CAS CONSTANT

Cas général d'une boucle simple

Variables intermédiaires : a, b, i : entier

```
pour i allant de a à b, faire
  Traitement(i)
fin pour
```

Hypothèse : Traitement(i) demande P opérations (P constant)

Complexité :

$$C = \underbrace{P}_{i=a} + \underbrace{P}_{i=a+1} + \dots + \underbrace{P}_{i=b=a+b-a} = \sum_{i=a}^b P = P \sum_{i=a}^b 1 = P(b-a+1)$$

LIEN BOUCLE – SOMME : CAS VARIABLE

Cas général d'une boucle simple

Variables intermédiaires : a, b, i : entier

```
pour i allant de a à b, faire  
  Traitement(i)  
fin pour
```

Hypothèse : Traitement(i) demande i opérations

Complexité :

$$C = \underbrace{a}_{i=a} + \underbrace{(a+1)}_{i=a+1} + \dots + \underbrace{b}_{i=b=a+b-a} = \sum_{i=a}^b i$$
$$C = \sum_{i=a}^b i = \sum_{i=0}^b i - \sum_{i=0}^{a-1} i = \frac{(b+1)b}{2} - \frac{a(a-1)}{2}$$

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

Vérification : $a=5, b=7$

$$C = 5+6+7=18$$

$$= 28-10 = (8 \times 7) / 2 - (5 \times 4) / 2$$

EXEMPLE : SOMME PARTIELLE MAXIMALE

Énoncé :

Étant donné A_0, A_1, \dots, A_{n-1} réels (potentiellement négatifs), trouver la valeur maximale pour

$$\sum_{k=i}^j A_k = A_i + \dots + A_j$$

Exemple

Pour la séquence -2, 11, -4, 13, -5, -2

La réponse est 20 (=11-4+13)

Algorithme naïf

On considère chaque nombre à tour de rôle : début de séquence

On considère chaque nombre suivant à tour de rôle : fin de séquence

On calcule la somme entre le début et la fin + test si max

(Voir « *Data Structures and Algorithm Analysis in C* » de Mark Allen Weiss)

ALGORITHME #1

Entrée :

A[n] : Réels

Sortie :

max : Réel // valeur maximale

Variables intermédiaires :

part : Réel // somme partielle

i,j,k : entier // variables de boucles

```
max ← A[0]
```

```
pour i allant de 0 à n-1, faire
```

```
  pour j allant de i à n-1, faire
```

```
    part ← 0
```

```
    pour k allant de i à j, faire
```

```
      part ← part + A[k]
```

```
    fin pour
```

```
    si part > max alors
```

```
      max ← part
```

```
    fin si
```

```
  fin pour
```

```
fin pour
```

```
Renvoie max
```

$$\left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^j 2+3 \right) \right) + 1$$

ALGORITHME #1 : CALCUL

$$C(n) = \left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^j 2 + 3 \right) \right) + 1$$

$$\sum_{k=i}^j 2 = 2 \sum_{k=i}^j 1$$

$$= 2 \left(\sum_{k=0}^j 1 - \sum_{k=0}^{i-1} 1 \right)$$

$$= 2 \left((j+1) - i \right)$$

$$= 2(j-i+1)$$

$$C(n) = \left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (2(j-i+1)+3) \right) + 1$$

$$= \left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (2j-2i+2+3) \right) + 1$$

$$= \left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (2j-2i+5) \right) + 1$$

$$= \left(\sum_{i=0}^{n-1} \left(\sum_{j=i}^{n-1} 2j - \sum_{j=i}^{n-1} 2i + \sum_{j=i}^{n-1} 5 \right) \right) + 1$$

Rappels

$$\sum_{k=0}^{n-1} 1 = n$$

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

$$\sum_{k=0}^{n-1} k^2 = \frac{n(n-1)(2n-1)}{6}$$

ALGORITHME #1 : CALCUL

$$C(n) = \left(\sum_{i=0}^{n-1} \left(\sum_{j=i}^{n-1} 2j - \sum_{j=i}^{n-1} 2i + \sum_{j=i}^{n-1} 5 \right) \right) + 1$$

$$\sum_{j=i}^{n-1} 5 = 5 \sum_{j=i}^{n-1} 1 = 5(n-i) = 5n - 5i$$

$$\sum_{j=i}^{n-1} 2i = 2i \sum_{j=i}^{n-1} 1 = 2i(n-i) = 2ni - 2i^2$$

$$\sum_{j=i}^{n-1} 2j = 2 \sum_{j=i}^{n-1} j$$

$$= 2 \left(\sum_{j=0}^{n-1} j - \sum_{j=0}^{i-1} j \right)$$

$$= 2 \left(\frac{n(n-1)}{2} - \frac{i(i-1)}{2} \right)$$

$$= n^2 - n - i^2 + i$$

$$\sum_{j=i}^{n-1} 2j - \sum_{j=i}^{n-1} 2i + \sum_{j=i}^{n-1} 5 = n^2 - n - i^2 + i - (2ni - 2i^2) + 5n - 5i = i^2 - i(2n+4) + n^2 + 4n$$

$$C(n) = \left(\sum_{i=0}^{n-1} i^2 - (2n+4) \sum_{i=0}^{n-1} i + (n^2+4n) \sum_{i=0}^{n-1} 1 \right) + 1$$

Rappels

$$\sum_{k=0}^{n-1} 1 = n$$

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

$$\sum_{k=0}^{n-1} k^2 = \frac{n(n-1)(2n-1)}{6}$$

ALGORITHME #1 : CALCUL

$$C(n) = \left(\sum_{i=0}^{n-1} i^2 - (2n+4) \sum_{i=0}^{n-1} i + (n^2+4n) \sum_{i=0}^{n-1} 1 \right) + 1$$

$$\begin{aligned} C(n) &= \frac{n(n-1)(2n-1)}{6} - (2n+4) \frac{n(n-1)}{2} + (n^2+4n)n + 1 \\ &= \frac{n^3}{3} + \frac{5}{2}n^2 + \frac{13}{6}n + 1 \end{aligned}$$

Rappels

$$\sum_{k=0}^{n-1} 1 = n$$

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

$$\sum_{k=0}^{n-1} k^2 = \frac{n(n-1)(2n-1)}{6}$$

ALGORITHME #1

Entrée :

A[n] : Réels

Variables :

max, part : Réel

i, j, k : entier // variables de boucles

max ← A[0]

pour i allant de 0 à n-1, faire

pour j allant de i à n-1, faire

part ← 0

pour k allant de i à j, faire

part ← part + A[k]

fin pour

si part > max alors

max ← part

fin si

fin pour

fin pour

Renvoie max

$$C(n) = \frac{n^3}{3} + \frac{5}{2}n^2 + \frac{13}{6}n + 1$$

Séquence : -2, 11, -4, 13, -5, -2

i=2

j=2

k=2

part=-4

j=3

k=2

part=-4

k=3

part=-4+13=9

j=4

k=2

part=-4

k=3

part=-4+13=9

k=4

part=-4+13-5=9-5=4

j=5 ...

ALGORITHME #2

Entrée :

A[n] : Réels

Variables :

max, part : Réel

i, j, k : entier // variables de boucles

```
max ← A[0]
```

```
pour i allant de 0 à n-1, faire
```

```
  part ← 0
```

```
  pour j allant de i à n-1, faire
```

```
    part ← part + A[j]
```

```
    si part > max alors
```

```
      max ← part
```

```
    fin si
```

```
  fin pour
```

```
fin pour
```

Renvoie max

4 (pire cas)

4 (n-i) + 1

$$C_2(n) = \left(\sum_{i=0}^{n-1} 4(n-i) + 1 \right) + 1$$
$$= 2n^2 + 3n + 1$$

COMPARAISON

n	n=100	n=1000	n=10000	n=100000
$C(n)=n^3/3+5/2 n^2 +13/6 n + 1$	358551	335835501	333583355001	333358333550001
$C_2(n)=2n^2 + 3n + 1$	20301	2003001	200030001	20000300001
$C(n)/C_2(n)$	17,66	167,67	1667,67	16667,67
$(n^3/3)/(2n^2)=n/6$	16,67	166,67	1666,67	16666,67

Pour n grand :

$$\frac{C(n)}{C_2(n)} = \frac{n^3/3+5/2n^2+13/6n+1}{2n^2+3n+1} \approx \frac{n^3/3}{2n^2} = \frac{n}{6}$$

Et, quand $n \leftarrow n*10$, alors $C(n)/C_2(n) \leftarrow C(n)/C_2(n) * 10$

→ le rapport de complexité évolue de manière linéaire, proportionnelle à n, soit comme n^3/n^2

COMPLEXITÉ POLYNOMIALE : RÈGLES

Complexité polynomiale $f(n) = \sum_{i=0}^k a_i n^i$

Exemple : $C(n) = \frac{n^3}{3} + \frac{5}{2}n^2 - \frac{13}{6}n + 1$

On néglige tous les termes sauf le plus fort $f(n) \approx a_k n^k$

i.e de plus grande puissance

Exemple : $C(n) \approx \frac{n^3}{3}$

On néglige les facteurs multiplicatifs $f(n) \propto n^k$

Exemple : $C(n) \propto n^3$

Note : une complexité en nombre constant d'opérations sera considérée comme de complexité proportionnelle à 1


Notation en $O(\cdot)$

$$f(n) = O(n^k) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{n^k} = \text{constante non nulle}$$

$$C(n) = O(n^3)$$

CLASSES DE COMPLEXITÉ

Complexité
croissante



Fonction	Appellation
1	Constante
$\log n$	Logarithmique ou sous-linéaire
n	Linéaire
$n \log n$	Linéarithmique ou quasi-linéaire
n^2	Quadratique (polynomiale)
n^3	Cubique (polynomiale)
n^k	Polynomiale ($k > 3$)
2^n ou a^n ou $\exp(n)=e^n$	Exponentielle ($a > 1$)
$n!$	factorielle

DES ALGORITHMES EFFICACES, OU PAS

Avec un ordinateur exécutant 10^9 instructions par seconde

n=	20	40	60	100	300
n^2	1/2500 millisecondes	1/625 millisecondes	1/278 millisecondes	1/100 millisecondes	1/11 millisecondes
n^5	1/300 secondes	1/10 secondes	78/100 secondes	10 secondes	40,5 minutes
2^n	1/1000 secondes	18,3 minutes	36,5 années	$400 \cdot 10^9$ siècles	(72c) siècles
n^n	$3,3 \cdot 10^9$ années	(46c) siècles	(89c) siècles	(182c) siècles	(725c) siècles

Note : (Xc) → nombre à X chiffres

On situe le big bang à environ $13,8 \cdot 10^9$ années, soit (9c) siècles !

(Source : « Algorithmics, the spirit of computing », D. Harel)

RÈGLES DE CALCUL THÉORIQUE

Règle 1 (addition)

Si $c_1(n) = O(f(n))$ et $c_2(n) = O(g(n))$

Alors $(c_1 + c_2)(n) = \max\{O(f(n)), O(g(n))\} \rightarrow$ on garde la complexité dominante

Règle 2 (multiplication)

Si $c_1(n) = O(f(n))$ et $c_2(n) = O(g(n))$

alors $(c_1 * c_2)(n) = O(f(n) * g(n))$

Règle 3 (complexité polynomiale)

Si $c(n)$ est une fonction polynomiale de degré k

alors $c(n) = O(n^k)$

n^{k-1} est dominée par n^k pour tout k (voir tableau précédent)

Règle 4 (complexité logarithmique)

$\text{Log}^k n$ est dominée par n pour toute constante k

RÈGLES DE CALCUL PRATIQUE

Règle 5 (instruction consécutives)

La complexité de blocs d'instructions consécutifs est donnée par le bloc le plus complexe (cela correspond à une somme, voir aussi règle 1).

Règle 6 (boucles)

La complexité d'une boucle est égale à la complexité du bloc d'instructions internes fois le nombre d'exécutions de la boucle. (cela correspond à un produit, voir aussi règle 2).

Règle 7 (boucles imbriquées)

Les boucles imbriquées s'analysent de l'intérieur vers l'extérieur : chaque boucle multiplie la complexité par le nombre de répétitions qu'elle implique (cas particulier de la précédente).

Règle 8 (Si/Alors)

Dans une instruction

```
Si Condition Alors
```

```
    S1
```

```
Sinon
```

```
    S2
```

```
Fin si
```

La complexité est donnée par la complexité maximum entre `Condition`, `S1`, et `S2` (expression du pire cas)

FOCUS SUR LES BOUCLES

Regarder les boucles

Une boucle dont une borne dépend de n a un nombre d'exécutions en $O(n)$.
Une boucle dont les bornes sont fixes a un nombre d'exécutions en $O(1)$.

Exemples

Pour i allant de 0 à $n-1$ → $O(n)$

Pour j allant de 2 à 20 → $O(1)$

Pour k allant de i à n → $O(n)$

Pour l allant de 0 à i → $O(n)$ (si i peut varier jusque n)
→ $O(1)$ (si i ne peut varier que dans des bornes constantes (ex : 2 à 20))

ALGORITHME #2

Entrée :

A[n] : Réels

Variables :

max, part : Réel

i, j, k : entier // variables de boucles

```
max ← A[0] O(1)
```

```
pour i allant de 0 à n-1, faire
```

```
part ← 0 O(1)
```

```
pour j allant de i à n-1, faire
```

```
part ← part + A[j] O(1)
```

```
si part > max alors O(1) + O(1) = O(1)
```

```
max ← part O(1)
```

```
fin si
```

```
fin pour
```

```
fin pour
```

```
Renvoie max O(1)
```

$$O(n) + O(n) + O(n) + O(1) + O(1) = O(n^2)$$

Rappel : on avait trouvé
 $C(n) = 2n^2 + 3n + 1$

RAPPELS SUR LES TABLEAUX : IMPACT MÉMOIRE

Définition

Un tableau est une structure de données servant à stocker plusieurs éléments d'un **même type**, sur une **zone contiguë** de la mémoire (=plage mémoire).

Notation

nom[taille] : type

Exemples :

tab[10] : réel

data[20] : entier

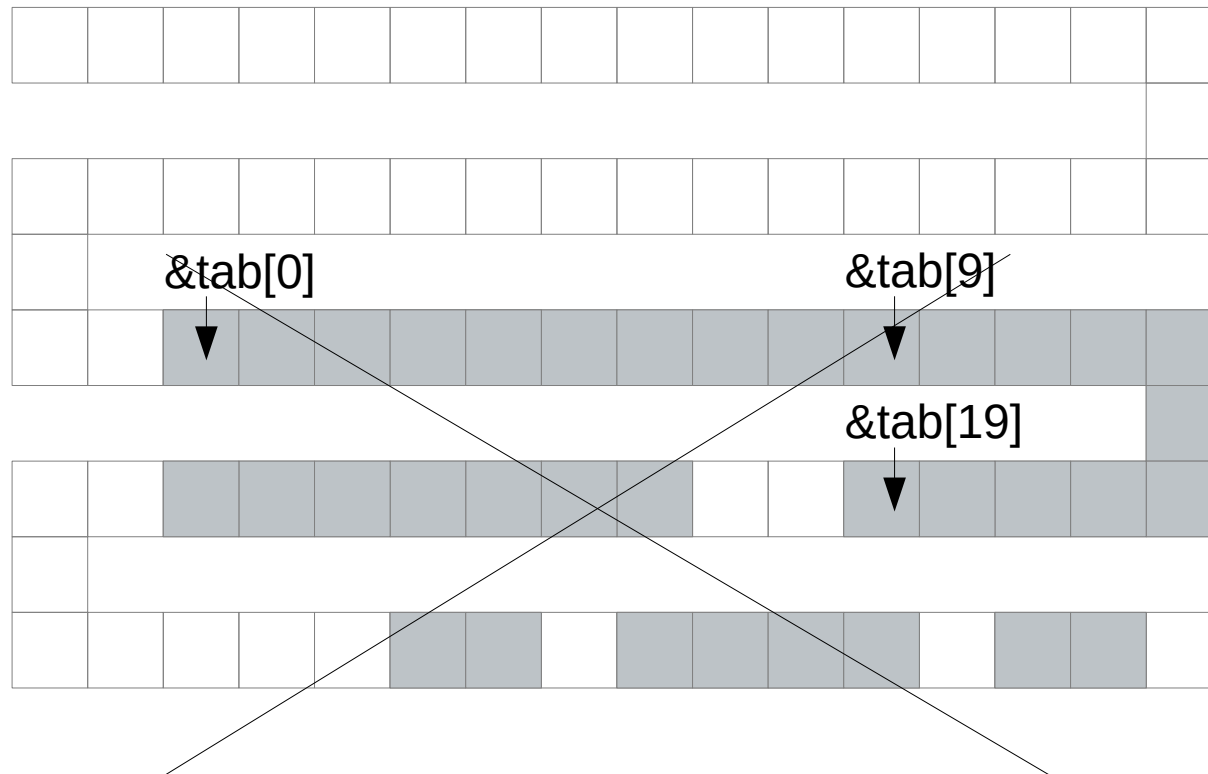
chaine[15] : caractère // chaîne de 14 caractères en C

Tableau[] : booléen // si la taille n'est pas connue a priori

Rappel : le C étant le langage employé dans ce cours, nous prenons comme convention d'indexer le premier élément du tableau par l'entier 0

TABLEAU=PLAGE MÉMOIRE

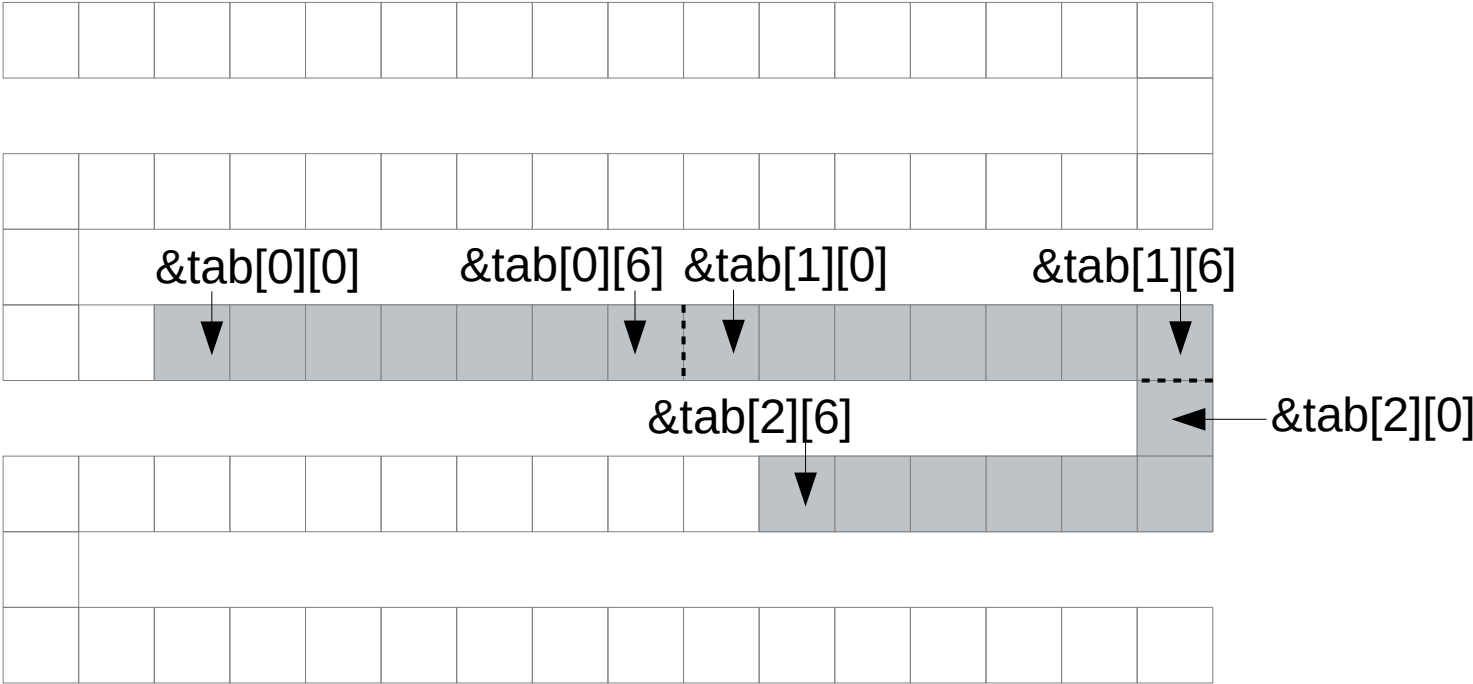
tab[20] : entiers



Note : dans ce schéma, une case = un entier = 4 octets
(en règle générale, une case mémoire = 1 octet)

TABLEAU MULTIDIMENSIONNEL

tab[3][7] : entiers



OPÉRATIONS ET COMPLEXITÉ

Accès à un élément (en lecture ou écriture)

Coût = calcul de l'adresse
= adresse tableau + indice * taille du type

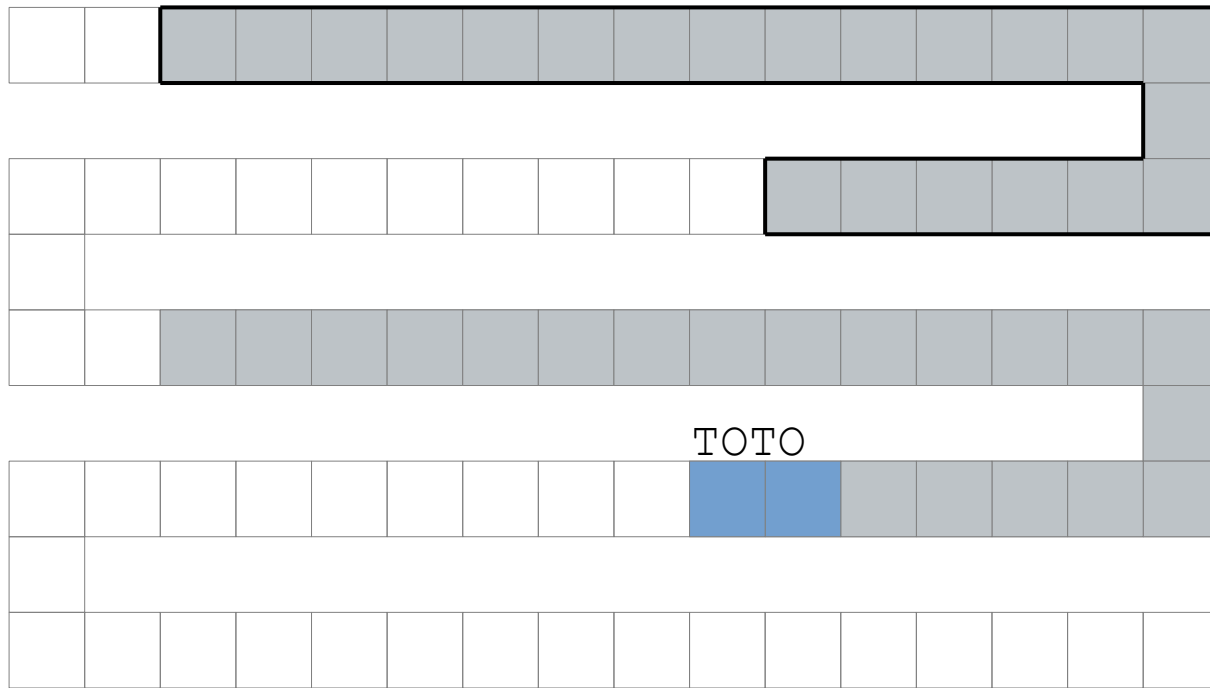
→ $O(1)$

Exemple : $\&\text{tab}[i] = \&\text{tab}[0] + i * \text{taille}(\text{entier})$

Note: pas exactement du C !!! (voir prochain TP)

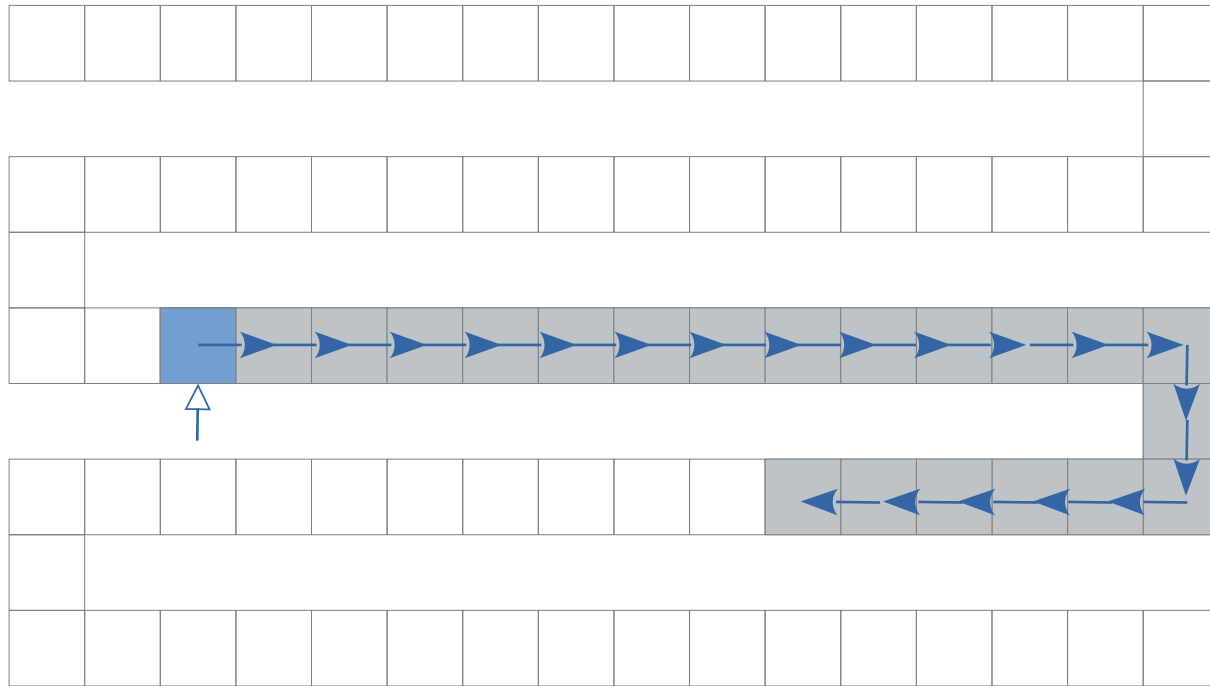
Ajout et retrait d'un élément ?

AJOUT D'UN ÉLÉMENT



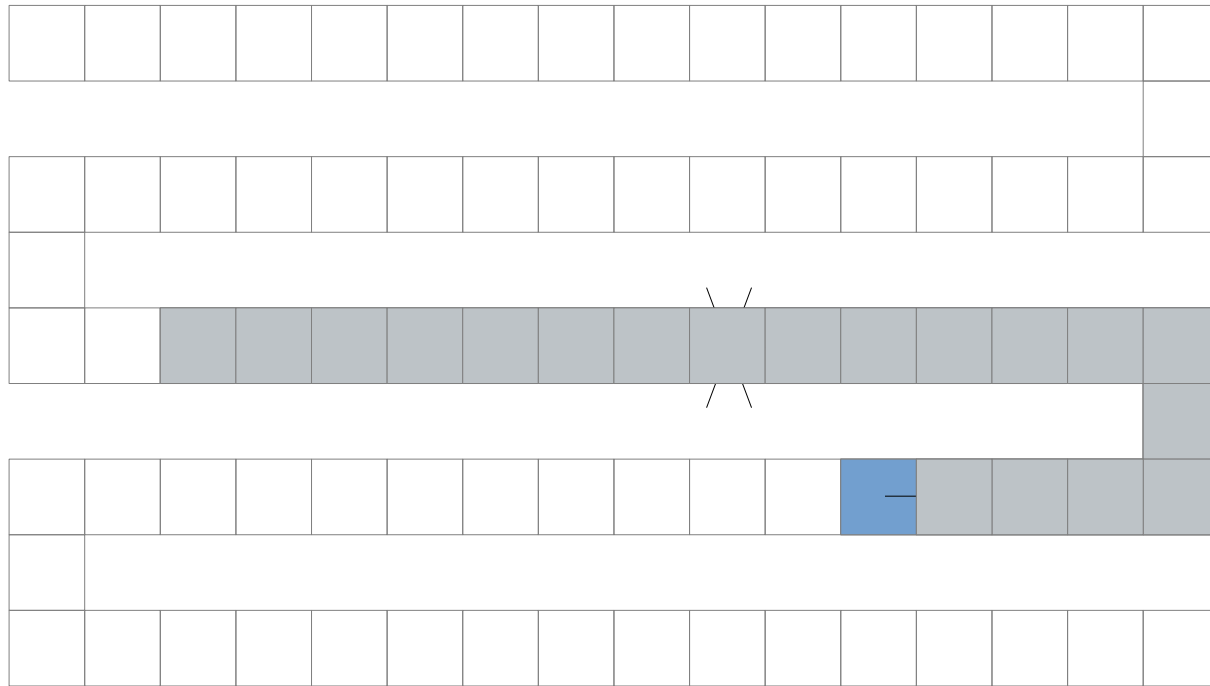
Pire cas : réallocation et recopie des n éléments $\rightarrow O(n)$

AJOUT D'UN ÉLÉMENT (V2)



Pire cas : ajout du premier élément = copie de n éléments $\rightarrow O(n)$

RETRAIT D'UN ÉLÉMENT



Pire cas : retrait du premier élément = copie de $n-1$ éléments $\rightarrow O(n)$

CONCLUSION

Chaque bloc d'instructions a une complexité

Se calcule avec des règles simples

Instruction élémentaire $\rightarrow O(1)$

Boucle dépendant de la taille de l'entrée (n) $\rightarrow O(n)$

2 boucles imbriquées dépendant de $n \rightarrow O(n^2)$

k boucles imbriquées dépendant de $n \rightarrow O(n^k)$

Tableaux

Occupent une zone contiguë en mémoire (même multidimensionnels)

Accès en $O(1)$

ajout/retrait en $O(n)$

CONCLUSION

Analyse à moduler avec les langages et architectures modernes

Allocation par « chunks » (blocs, voir *allocators*) :

allouer de la mémoire coûte cher et prend un temps variable → on alloue systématiquement des blocs de mémoire pour minimiser le nombre d'allocations

Mémoire bon marché

La mémoire étant bon marché à l'heure actuelle, une solution simple pour accélérer son code est souvent (et si possible) d'allouer un gros espace mémoire pour tout le programme

Cela reste des cas particuliers : La structure de données impacte la complexité

Il en existe d'autres que le tableau, dont les opérations sont de complexités différentes → objet des prochains cours