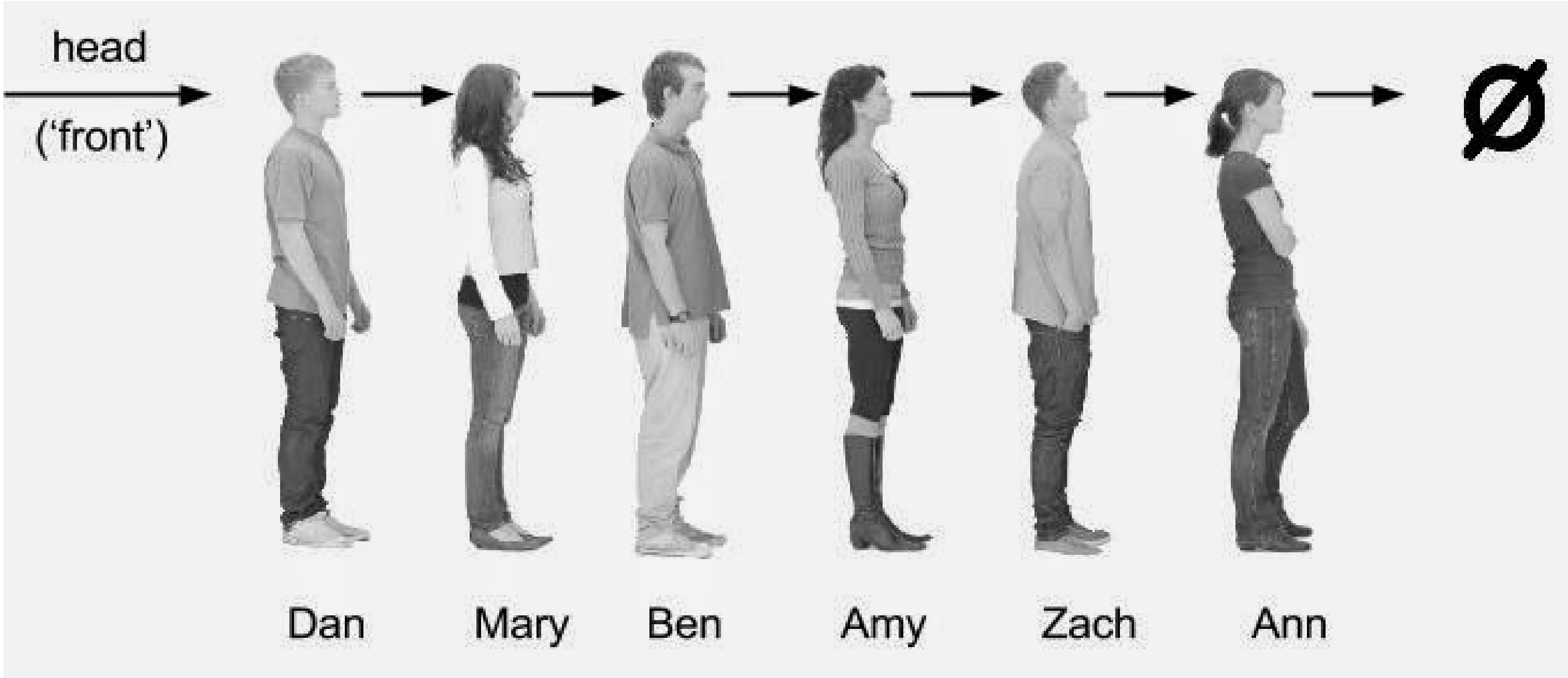


STRUCTURES DE DONNÉES ET ALGORITHMES FONDAMENTAUX

Initiation au développement (DEV)

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

Deux structures de base :

- **le tableau (structure à accès direct)**
- **la liste chaînée (structure à accès séquentiel)**

De quoi va-t-on parler aujourd'hui ?

Complexité

- Complexité en temps vs espace
- Quelques exemples
- Principe de mesure (ordre polynomial)

Type

- Type abstrait de données
- Liste itérative

Première structure de données : la table (aka le tableau)

- Définition (rappels)
- Opérations de base
- Impact sur la mémoire

COÛT D'UN ALGORITHME

Un algorithme applique des <u>opérations</u> sur des <u>données</u>

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

Coût d'un algorithme

Un algorithme applique des <u>opérations</u> sur des <u>données</u>

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
  renvoyer Vrai
sinon
  renvoyer Faux
fin si
```

Complexité en temps :

1 opération et 1 test pour tout n (et renvoie ?) → k (=2 ou ...)

k ne dépend pas de la valeur de n

Exemple #1

Problème :

déterminer la parité d'un nombre entier n

Algorithme : (test sur le bit de poids faible)

```
Si n&1 = 0 alors
  renvoyer Vrai
sinon
  renvoyer Faux
fin si
```

Complexité en temps :

1 opération et 1 test pour tout n (et renvoie ?) → k (=2 ou ...)

k ne dépend pas de la valeur de n

Exemple #1

Problème :

déterminer la parité d'un nombre entier n

Algorithme : (test sur le bit de poids faible)

```
Si n&1 = 0 alors
   renvoyer Vrai
sinon
   renvoyer Faux
fin si
```

Complexité en temps :

1 opération et 1 test pour tout n (et renvoie ?) → 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)*...*2*1)

Algorithme :

```
Sortie : f : entier
f ← 1
pour i allant de 1 à n faire
   f ← f * i
fin pour
renvoyer f
```

Complexité en temps :

n multiplications et n affectations (+ opérations sur **i**...) → 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
renvoyer tab[1]
```

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[1]
pour i allant de 2 à n, faire
   si tab[i] > max, alors
        max ← tab[i]
   fin si
fin pour
renvoyer 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
   pour i allant de 1 à n, faire
n    2 f ← f * i
   fin pour
   renvoyer f
```

Règles

- Commencer par « l'intérieur »
- Remonter vers « l'extérieur »

Complexité :

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

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

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

→ boucles du code ↔ sommes sur même intervalle

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 :

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

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 :

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

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

$$\boxed{\sum_{k=1}^{n} k = \frac{n(n+1)}{2}}$$

Vérification : a=5, b=7

C =5+6+7=18

=28-10 = (7x8)/2 – (4x5)/2

Exemple : somme partielle maximale

Énoncé :

Étant donné $A_1, A_2, \dots, A_n$ réels (potentiellement négatifs), trouver la valeur maximale pour

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

Exemple

Pour la séquence -2, $\boxed{11, -4, 13}$, -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

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

```
Entrée :
     A[n] : Réels
Sortie :
     max : Réel // valeur maximale
Variables intermédiaires :
     part : Réel // somme partielle
     i,j,k : entier // variables de boucles
max ← A[1]
pour i allant de 1 à n faire
     pour j allant de i à n 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
Renvoyer max
```

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

$$\begin{aligned}
\sum_{k=i}^{j} 2 &= 2 \sum_{k=i}^{j} 1 \\
&= 2\left(\sum_{k=1}^{j} 1 - \sum_{k=1}^{i-1} 1\right) \\
&= 2\left(j - (i-1)\right) \\
&= 2\left(j - i + 1\right)
\end{aligned}$$

$$\begin{aligned}
C(n) &= \left(\sum_{i=1}^{n} \sum_{j=i}^{n} \left(\boxed{2(j-i+1)} + 3\right)\right) + 1 \\
&= \left(\sum_{i=1}^{n} \sum_{j=i}^{n} (2j - 2i + 2 + 3)\right) + 1 \\
&= \left(\sum_{i=1}^{n} \sum_{j=i}^{n} (2j - 2i + 5)\right) + 1 \\
&= \left(\sum_{i=1}^{n} \left(\sum_{j=i}^{n} 2j - \sum_{j=i}^{n} 2i + \sum_{j=i}^{n} 5\right)\right) + 1
\end{aligned}$$

Rappels

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

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

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

Algorithme #1 : calcul

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

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

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

$$\begin{aligned}\sum_{j=i}^{n}2j &=2\sum_{j=i}^{n}j\\ &=2\left(\sum_{j=1}^{n}j-\sum_{j=1}^{i-1}j\right)\\ &=2\left(\frac{n(n+1)}{2}-\frac{(i-1)i}{2}\right)\\ &=n^2+n-i^2+i\end{aligned}$$

$$\sum_{j=i}^{n}2j-\sum_{j=i}^{n}2i+\sum_{j=i}^{n}5=\boxed{n^2+n-i^2+i}-\boxed{(2ni-2i^2+2i)}+\boxed{5n-5i+5}=i^2-(2n+6)i+(n^2+6n+5)$$

$$C(n)=\left(\sum_{i=1}^{n}i^2-(2n+6)\sum_{i=1}^{n}i+(n^2+6n+5)\sum_{i=1}^{n}1\right)+1$$

Rappels

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

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

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

ALGORITHME #1 : CALCUL

$$C(n) = \left(\sum_{i=1}^{n} i^2 - (2n+6) \sum_{i=1}^{n} i + (n^2+6n+5) \sum_{i=1}^{n} 1 \right) + 1$$

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

Rappels

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

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

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

Algorithme #1

```
Entrée :
    A[n] : Réels
Variables :
    max, part : Réel
    i,j,k : entier // variables de boucles
max ← A[1]
pour i allant de 1 à n faire
    pour j allant de i à n 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
Renvoyer max
```

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

Séquence : -2, 11, $\boxed{-4,}$ $\boxed{13,}$ $\boxed{-5,}$ $\boxed{-2}$

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[1]
pour i allant de 1 à n faire
   part ← 0
   pour j allant de i à n faire
      part ← part + A[j]
      si part > max alors
         max ← part
      fin si
   fin pour
fin pour
Renvoyer max
```

4 (pire cas)

4 (n-i+1) +1

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

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
$C2(n)=2n^2+3n+1$	20301	2003001	200030001	20000300001
C(n)/C2(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 ← n*10, alors C(n)/C2(n) ← C(n)/C2(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(.)

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

$$C(n)=O(n^3) \text{ avec } 1/3 \text{ comme constante}$$

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	$400 . 10^9$ siècles	(72c) siècles
n^n	$3,3 . 10^9$ années	(46c) siècles	(89c) siècles	(182c) siècles	(725c) siècles

Note : (Xc) → nombre à X chiffres

On situe le big bang à environ $13,8 . 10^9$ 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 $(c_1+c_2)(n) = \max\{O(f(n)),O(g(n))\}$ → 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[1]                              O(1)
pour i allant de 1 à n faire
    part ← 0                            O(1)
    pour j allant de i à n faire
        part ← part + A[j]              O(1)
        si part > max alors             O(1) + O(1) = O(n²)
            max ← part                  O(1)
        fin si
    fin pour
fin pour
Renvoyer max                            O(1)
```

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

De quoi va-t-on Parler aujourd'hui ?

Complexité

- Complexité en temps vs espace
- Quelques exemples
- Principe de mesure (ordre polynomial)

Type

- Type abstrait de données
- Liste itérative

Première structure de données : la table (aka le tableau)

- Définition (rappels)
- Opérations de base
- Impact sur la mémoire

PRENONS UN PEU DE RECUL

Pourquoi une structure de données ?

Stockage et manipulation de nombreuses données

Notion générique d’ensemble

- Accéder à un élément
- Ajouter un élément
- Supprimer un élément
- Tester l’appartenance d’un élément
- Définir l’ensemble vide
- Compter le nombre d’éléments

… (on peut ajouter les opérations d’union, intersection, sous-ensemble...)

→ **Conception centrée sur les valeurs et les opérations :**

Type Abstrait de Données

TYPE ABSTRAIT DE DONNÉES

Changement de focalisation

- Implantation (représentation) → opérations

Deux éléments de définition

- Signature :
  - nom (sorte), sortes utilisées, ensemble de valeurs valides, prototype des opérations
- Préconditions/axiomes
  - Définissent le comportement des opérations

BOOLÉEN COMMME TAD (PROPOSITION)

Signature

Sorte : Booléen

Utilise : (nil)

Opérations

Vrai	: → Booléen
Faux	: → Booléen
¬ (not)	: Booléen → Booléen
∧ (and)	: Booléen x Booléen → Booléen
∨ (or)	: Booléen x Booléen → Booléen

Préconditions/axiomes

$\neg$ Vrai = Faux

$\neg\neg b = b$

$(b=\text{Vrai} \vee b=\text{Faux}) = \text{Vrai}$

$(b \wedge \text{Faux}) = \text{Faux}$

$(b \vee \text{Vrai}) = \text{Vrai}$

$\neg (a \wedge b) = \neg a \vee \neg b$

Commutativité

$a \wedge b = b \wedge a$

$a \vee b = b \vee a$

Distributivité

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

Représentation ? Peut être quelconque (0/1, ‘V’/’F’, « VRAI »/« FAUX », 3/-18...)

UTILISATION DU TAD

Rappel des opérations/valeurs

Vrai, Faux, ¬ (not), ˄ (and), ˅ (or)

Définition de nouvelles fonctions (algorithmes)

```
Fonction nand
  Entrées : a, b : Booléen
  Sortie :  Booléen
  Début
    Renvoyer ¬(a ∧ b)
  Fin
```

```
Fonction xor
  Entrées : a, b : Booléen
  Sortie :  Booléen
  Début
    Renvoyer (a ∨ b) ∧ ¬(a ∧ b)
  Fin
```

```
¬(a ∧ b) = not(and(a,b))
```

```
(a ∨ b) ∧ ¬(a ∧ b)=
            et(ou(a,b),not(et(a,b))
```

INTÉRÊT D'UN TAD

Séparer algorithme et programmation

- Algorithme décrit en utilisant les opérations définies pour le TAD
- Programmation implante ces opérations sur une structure de données, puis implante l'algorithme

Conception modulaire de programmes

- Workflow : Algorithme → opérations → TAD → structure de données
- Écrire plein de petites fonctions est bien
- Écrire plein de petites fonctions est bien (répétition voulue)
- Bon pour
  - → conception et réalisation de tests unitaires
  - → maintenance du code
  - → réutilisation du code
  - → identification et correction de bugs
  - → structuration de la documentation
  - → ...

EXEMPLE IMPLANTATION BOOLÉEN

```
Fonction Vrai
  Sortie : Booléen
  Début
    Renvoyer -18
  Fin


Fonction Faux
  Sortie : Booléen
  Début
    Renvoyer 3
  Fin


Fonction not
  Entrée : b : Booléen
  Sortie : Booléen
  Début
    Si b = Faux() alors
      Renvoyer Vrai()
    Sinon
      Renvoyer Faux()
    FinSi
  Fin
```

```
Fonction and
  Entrée : a,b : Booléen
  Sortie : Booléen
  Début
    Si a=Faux() alors
      Renvoyer Faux()
    Sinon Si b=Faux() alors
      Renvoyer Faux()
    Sinon
      Renvoyer Vrai()
    Finsi
  Fin

Fonction or
  Entrée : a,b : Booléen
  Sortie : Booléen
  Début
    Si a=Vrai() alors
      Renvoyer Vrai()
    Sinon Si b=Vrai() alors
      Renvoyer Vrai()
    Sinon
      Renvoyer Faux()
    FinSi
  Fin
```

TAD ENSEMBLE

Signature

Sorte : Ensemble
Utilise : Élément, Booléen, Entier
Opérations :

ensemble_vide	: → Ensemble
EstVide	: Ensemble → Booléen
Ajouter	: Ensemble x Élément → Ensemble
Supprimer	: Ensemble x Élément → Ensemble
EstDans	: Ensemble x Élément → Booléen
Taille	: Ensemble → Entier

Axiomes (E:Élément, S:Ensemble)

- EstVide(ensemble_vide) = Vrai
- Si (EstVide(S) = Vrai) alors S=ensemble_vide
- Si Taille(S) > 0 alors EstVide(S) = Faux
  Sinon EstVide(S)=Vrai
- EstDans(ensemble_vide, E)=Faux
- EstDans(Ajouter(S,E),E) = Vrai
- Si EstDans(S,E)=Faux alors
  Taille(Ajouter(S,E)) = Taille(S)+1
- Si EstDans(S,E)=Vrai alors
  Taille(Supprimer(S,E)) = Taille(S)-1
- Si EstDans(S,E)=Faux alors
  Taille(Supprimer(S,E)) = Taille(S)

Cas sans répétition (cas avec répétition = multi-ensemble)

- EstDans(Supprimer(S,E),E) = Faux
- Si EstDans(S,E)=Vrai alors
  Taille(Ajouter(S,E)) = Taille(S)

TAD ENSEMBLE

Signature

Sorte : Ensemble
Utilise : Élément, Booléen, Entier
Opérations :

ensemble_vide	: → Ensemble
EstVide	: Ensemble → Booléen
Ajouter	: Ensemble x Élément → Ensemble
Supprimer	: Ensemble x Élément → Ensemble
EstDans	: Ensemble x Élément → Booléen
Taille	: Ensemble → Entier

Problème

Cette définition ne permet pas d'accéder à un élément !

Par exemple : comment lister les éléments pour les afficher ? (fonction Afficher(Ensemble))

TAD LISTE ITÉRATIVE

Liste

Ensemble d'éléments rangés

Rangé ≠ trié → Rangé = chaque élément a un rang

Liste itérative : rang=entier

Signature

Sorte : ListeIter

Utilise : Élément, Booléen, Entier

Opérations :

liste_vide	: → ListeIter
EstVide	: ListeIter → Booléen
Ajouter	: ListerIter x Entier x Élément → Ensemble
Supprimer	: ListeIter x Entier → ListeIter
EstDans	: ListeIter x Élément → Booléen
Taille	: ListeIter → Entier
Contenu	: ListeIter x Entier → Élément

```
Procédure Afficher
  Entrée : L : ListeIter
  Variables : i,T : entier
  Début
    T ← Taille(L)
    Pour i allant de 1 à T faire
      Afficher(Contenu(L,i))
    FinPour
  Fin
```

STRUCTURE DE DONNÉES : LE TABLEAU

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

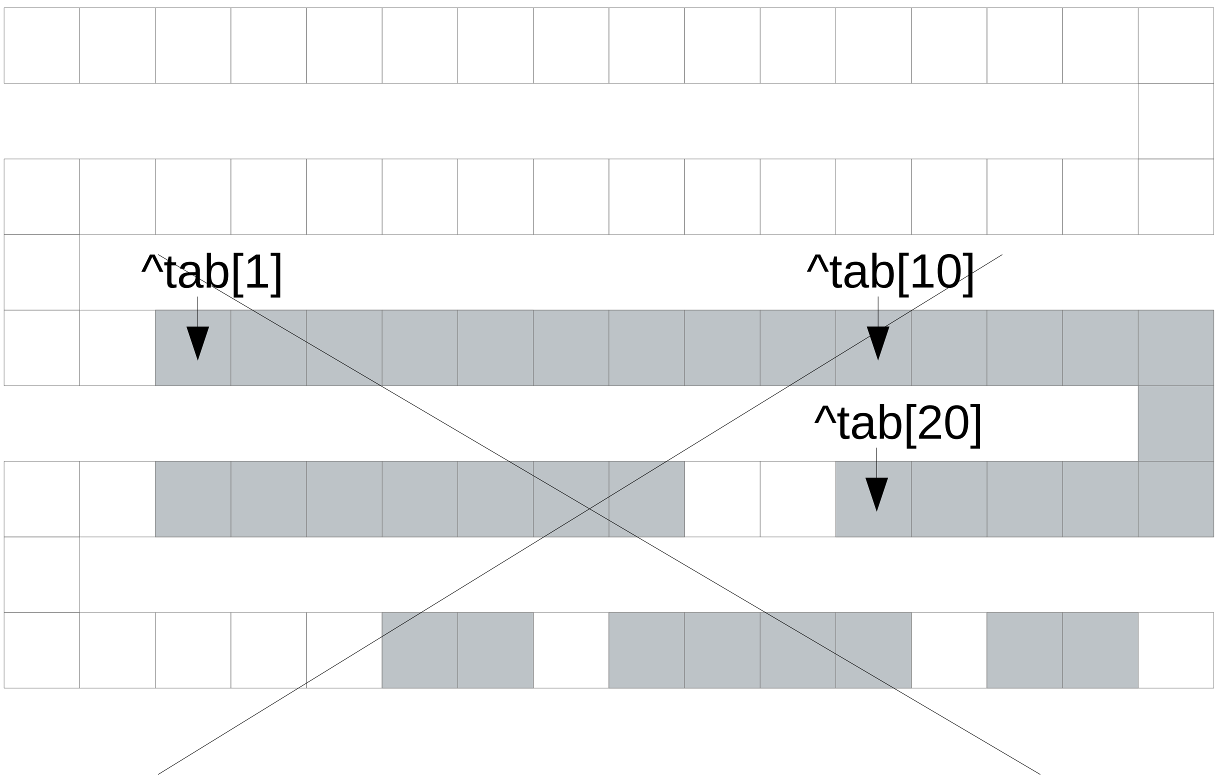
nom[taille] : type

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 : nous prenons comme convention d'indexer le premier élément du tableau par l'entier 1. Un tableau à N éléments sera donc indexé de 1 à N inclus.

TABLEAU=PLAGE MÉMOIRE

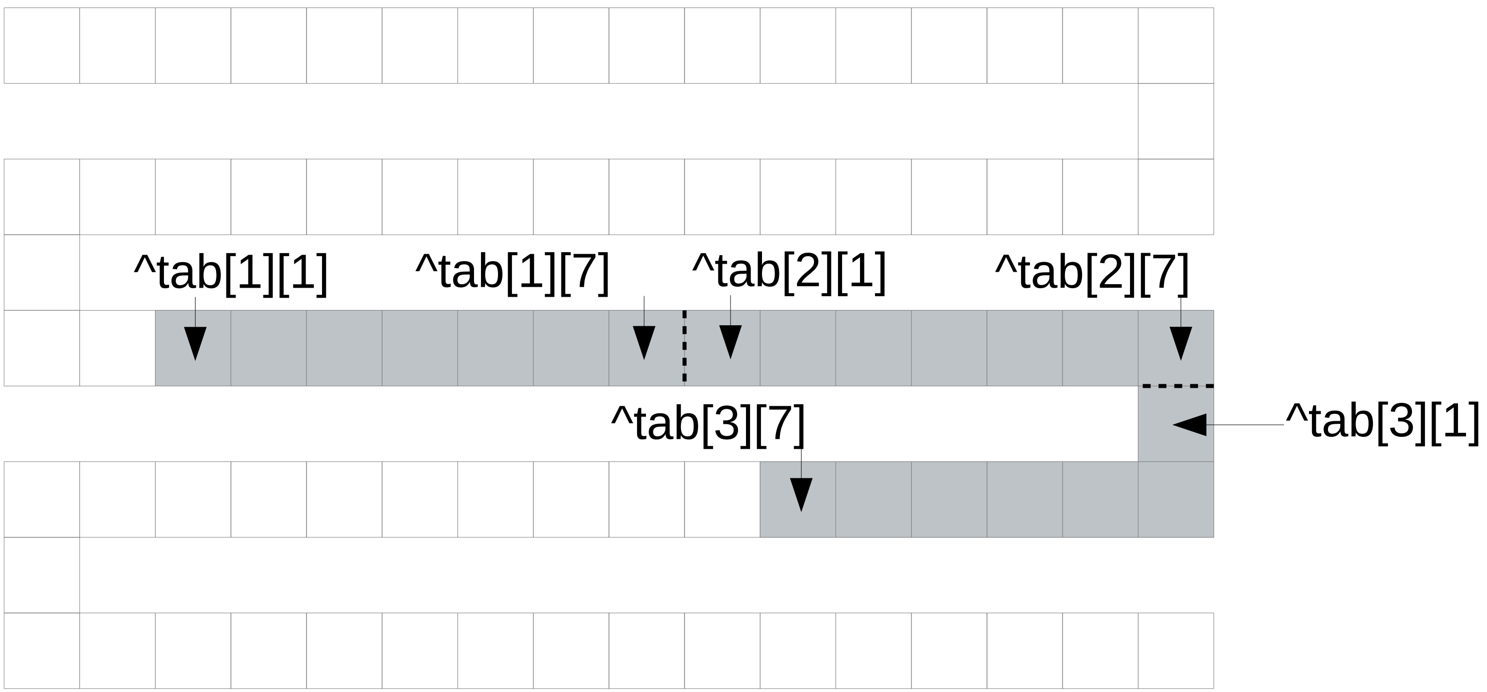
tab[20] : entiers



Notes :
- dans ce schéma, une case = un entier = 4 octets
(en règle générale, une case mémoire = 1 octet)
- *^ renvoie l'adresse d'une variable (comme & en C)*

Tableau multidimensionnel

tab[3][7] : entiers



Opérations et complexité

Accès à un élément (en lecture ou écriture : Contenu)

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

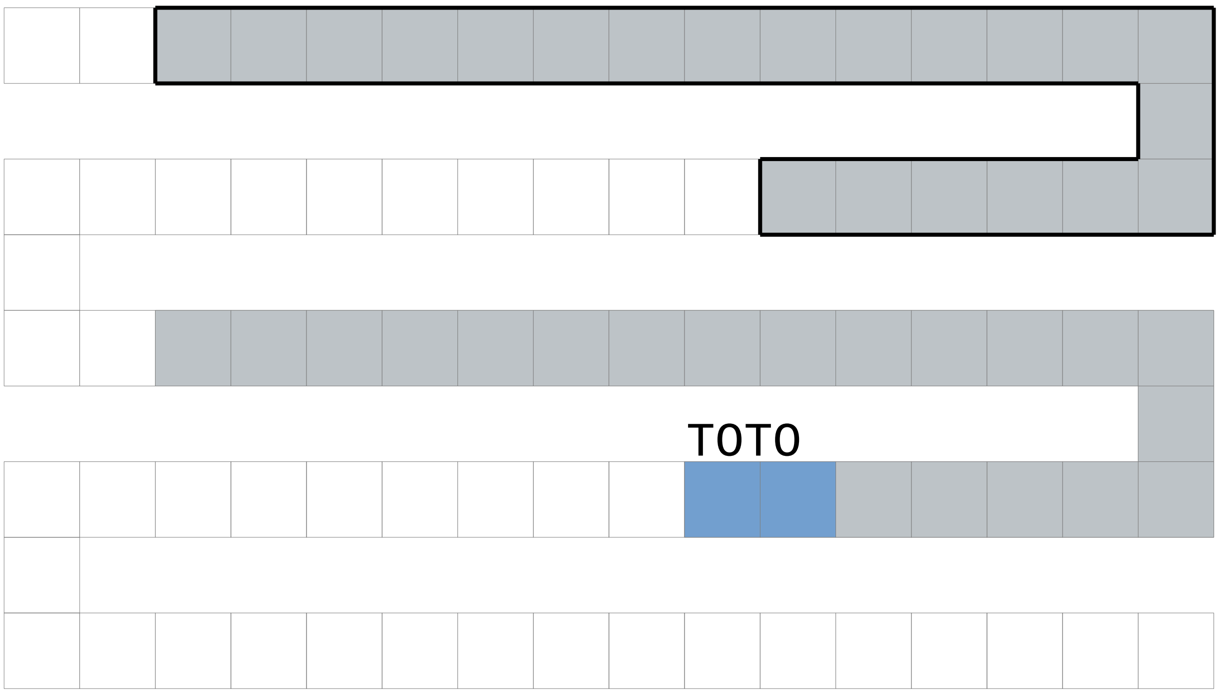
→ O(1)

Exemple : ^tab[i] = ^tab[1] + (i-1) * taille(entier)

Note: La formule est légèrement différente en C (voir TP sur les pointeurs)

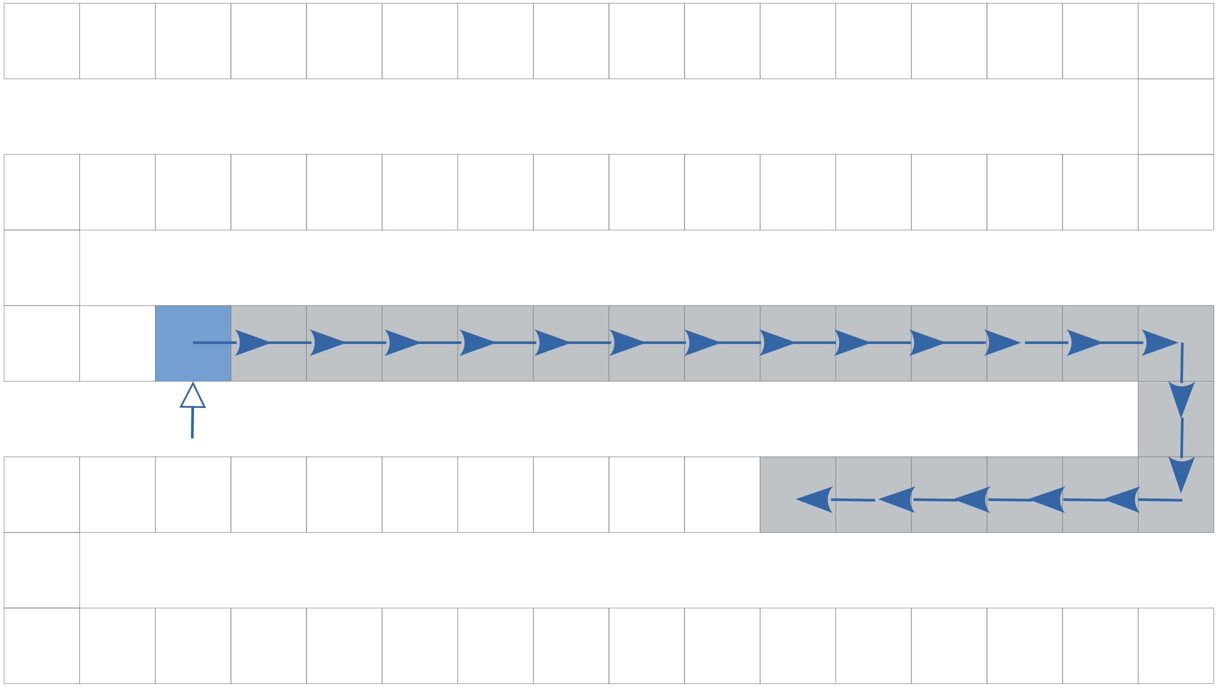
Ajout et suppression d'un élément ?

Ajout d'un élément (Ajouter)



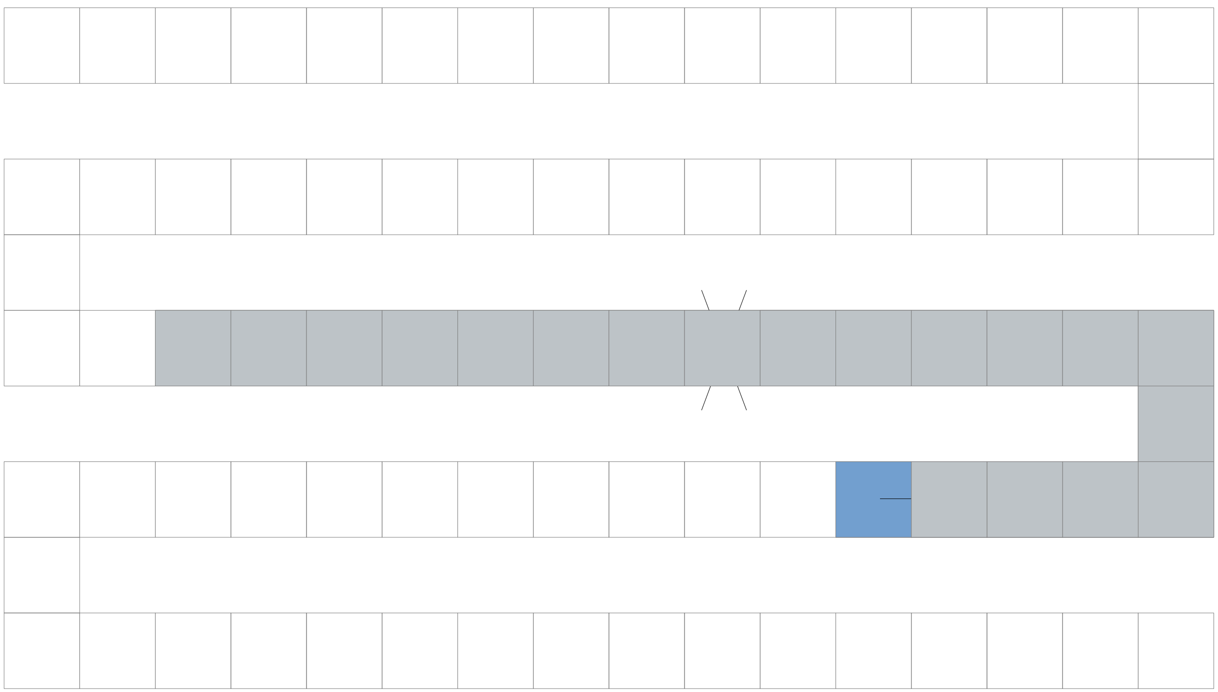
Pire cas : réallocation et recopie des n éléments → O(n)

AJOUT D’UN ÉLÉMENT (AJOUTER V2)



Pire cas : ajout du premier élément = copie de n éléments → O(n)

Suppression d'un élément (Supprimer)



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

Conclusion

Chaque bloc d'instructions a une complexité

- Se calcule avec des règles simples
- Instruction élémentaire → $O(1)$
- Boucle dépendant de la taille de l'entrée (n) → $O(n)$
- 2 boucles imbriquées dépendant de n → $O(n^2)$
- k boucles imbriquées dépendant de n → $O(n^k)$

Tableaux

- Occupent une zone contiguë en mémoire (même multidimensionnels)
- Accès en $O(1)$
- ajout/suppression 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