

Chapitre 9

Tableaux

`std::array, std::vector,
std::span, tris simples`



Contenu du chapitre

1. `std::array<T,n>`
2. `std::vector<T>`
3. `std::span<T>`
4. Tableaux multidimensionnels
5. Tri à bulles
6. Tri par sélection
7. Tri par insertion
8. Comparaison des tris





- Pour stocker **plusieurs** éléments de **même type**, la méthode la plus simple consiste à les stocker dans un **tableau** (English : array)
- On accède aux **éléments** du tableau via un **indice**, qui indique leur **position** dans le tableau
- Le **premier** élément est à l'**indice 0** dans la plupart des langages
- Les **éléments** du tableau sont placés **consécutivement en mémoire**. Par exemple en C++, pour un tableau t d'éléments de type **int** :

```
for(int i = 0; i < 4; ++i) {  
    cout << &(t[i]) << endl;  
}  
cout << "sizeof(int) = " << sizeof(int);
```

0x142604410	+4
0x142604414	+4
0x142604418	+4
0x14260441c	+4
sizeof(int) = 4	



- Le C++ propose 4 méthodes pour créer un tableau de n entiers

```
vector<int> v(n);           // classe std::vector  
array<int,n> a;            // classe std::array  
int t[n];                  // tableau classique, «à la C»  
int* p = new int[n];        // tableau alloué dynamiquement
```

- std::array** est utilisé pour des tableaux dont la **taille** (nombre d'éléments) est **fixée à la compilation**
- std::vector** est utilisé pour des tableaux dont la taille n'est **connue qu'à l'exécution** ou dont la taille **peut varier**
- Les deux autres méthodes sont des abstractions de plus bas niveau. Nous les aborderons au chapitre 15 (allocation dynamique) et en PRG2



- La bibliothèque standard propose des **alternatives** aux tableaux pour stocker des données multiples : les **conteneurs**
- Ils stockent leurs éléments dans des **structures de données** plus complexes : tableau de tableaux, listes chainées, arbres binaires de recherche, tables de hachage, ...
- Ils offrent d'autres **compromis** entre coût en mémoire, vitesse d'exécution, fonctionnalités, ...
- Leur étude détaillée se fera lors du cours d'**Algorithmes et Structures de Données**

Container class templates

Sequence containers:

array C++	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++	Unordered Set (class template)
unordered_multiset C++	Unordered Multiset (class template)
unordered_map C++	Unordered Map (class template)
unordered_multimap C++	Unordered Multimap (class template)



1. std::array<T,n>



Déclaration



- La classe générique `std::array` est définie dans le header `<array>`
- Le type `T` et nombre `n` d'éléments font partie du type déclaré, selon la syntaxe

```
std::array<T,n> nom_de_variable;
```

- Exemples :

```
std::array<int, 10> a1;           // tableau de 10 int
std::array<int, 42> a2;           // tableau de 42 int
std::array<double, 5> a3;          // tableau de 5 double
std::array<std::string, 7> a4; // tableau de 7 std::string
std::array<std::array<double, 4>, 3> a5;
                                // tableau de 3 tableaux de 4 double
```

Déclaration sans initialisation



- Sans initialisation explicite, les éléments d'un `std::array<T,n>` déclaré en variable locale ont une valeur indéterminée si le type `T` ne s'initialise pas par défaut.

```
std::array<int, 10> a1;           // 10 entiers de valeurs indéterminées
std::array<std::string, 7> a2; // 7 std::string initialisées vides
```

- Le comportement est identique à celui de la déclaration d'un élément seul

```
int e1;                  // 1 entier de valeur indéterminée
std::string e2;        // 1 std::string initialisée vide
```

- Comme pour les variables simples, un `std::array<T,n>` alloué statiquement (variable locale statique ou globale) a ses éléments initialisés par défaut (à zéro pour les types numériques)



- On peut donner une valeur initiale aux éléments en utilisant un **agrégat** entre accolades contenant ces valeurs initiales

```
array<int, 4> a1 {3, 1, 4, 1};      // le tableau contient [3, 1, 4, 1]
array<int, 4> a2 = {3, 1, 4, 1};    // syntaxe alternative 1
array<int, 4> a3 ({3, 1, 4, 1});    // syntaxe alternative 2
```

- L'agrégat **peut contenir moins d'éléments** que le `std::array<T,n>`, les premiers éléments sont initialisés par l'agrégat, les suivants par défaut

```
array<int, 4> a3 {3, 1};           // le tableau contient [3, 1, 0, 0]
array<int, 4> a4 {};              // le tableau contient [0, 0, 0, 0]
```

- L'agrégat **ne peut pas contenir plus d'éléments** que le `std::array<T,n>`

```
array<int, 4> a5 {3, 1, 4, 1, 5}; // error: excess elements in struct initializer
```

- Si le compilateur peut **déduire le type** et la taille de l'initialisation, il n'est pas obligatoire (depuis C++17) de les spécifier.

```
array a6 {12.f, 3.f}; // a6 est de type array<float,2>
```



Initialisation par copie / affectation

- Enfin, on peut **initialiser** un `std::array<T, n>` en **copiant** le contenu d'un autre `std::array<T, n>` du **même type**, i.e. avec les mêmes type `T` et nombre d'éléments `n`.

```
array<int, 4> a1 {3, 1, 4, 5};  
array<int, 4> a2 = a1;           // a2 contient [3, 1, 4, 5]  
array<int, 5> a3 = a1;  
// error: no viable conversion from 'array<[...], 4>'  
// to 'array<[...], 5>'  
array<unsigned, 4> a4 = a1;  
// error: no viable conversion from 'array<int, [...]>'  
// to 'array<unsigned int, [...]>'
```

- Notons que l'on peut aussi copier dans un `std::array<T, n>` existant par **affectation**

```
array<int, 4> a5;           // a5 indéterminé  
a5 = a1;                   // a5 contient [3, 1, 4, 5]
```



- L'initialisation par agrégat vide permet de remplir un `std::array<T,n>` de valeurs par défaut, i.e. `T()`
- La méthode `.fill(val)` permet de remplir un `std::array<T,n>` existant avec la même valeur `val` pour ses `n` éléments

```
array<int, 4> a;           // a contient 4 entiers indéterminés
a.fill(42);                 // a contient [42, 42, 42, 42]
a = {};;                   // a contient [0, 0, 0, 0]
a.fill(-1);                // a contient [-1, -1, -1, -1]
a.fill(int());              // a contient [0, 0, 0, 0]
```

Accès aux éléments d'un std::array<T,n>



- Lire ou écrire un élément se fait via son indice **i**, de type **size_t** (non signé)
 - Avec l'opérateur **[i]**, qui ne vérifie pas que l'indice est correct
 - Avec la méthode **.at(i)**, qui lève une exception si **i >= n**.
 - Notons qu'un indice **i** négatif est $\geq n$ une fois converti en **size_t**
- Les premier et dernier éléments peuvent être accédés via les méthodes **.front()** et **.back()**

```
array<int, 3> a {3, 1, 4};  
cout << a[0] << a.at(0) << a.front();      // affiche 333  
cout << a[1] << a.at(1);                  // affiche 11  
cout << a[2] << a.at(2) << a.back();      // affiche 444  
cout << a[3];    // comportement indéterminé  
cout << a.at(3); // libc++abi: terminating due to uncaught exception  
                  // of type std::out_of_range: array::at
```



- La méthode `.size()` retourne le nombre d'éléments, de type `size_t`
- Une boucle sur tous les éléments par indice croissant s'écrit donc

```
array<int, 4> a {3, 1, 4, 1};

for(size_t i = 0; i < a.size(); ++i)
    cout << a[i] << ' ';
                                // affiche 3 1 4 1

for(size_t i = 0; i < a.size(); ++i)
    a[i] *= a[i];
                                // a contient [9, 1, 16, 1]

for(size_t i = 0; i < a.size(); ++i)
    cout << a.at(i) << ' ';
                                // affiche 9, 1, 16, 1
```



- Le premier élément à traiter est à l'indice `i = a.size()-1` et le dernier à traiter est à `i = 0`
- Le premier élément à ne pas traiter (`i = -1`) ... dépend du type choisi pour l'indice
 - Avec un type `unsigned`, la condition de continuation de la boucle reste `i < a.size()`. Une condition `i >= 0` serait toujours vraie et entraînerait une boucle infinie

```
array<int, 4> a {3, 1, 4, 1};  
for(size_t i = a.size()-1; i < a.size(); --i)  
    cout << a[i] << ' ';                                // affiche 1 4 1 3  
for(size_t i = a.size()-1; i >= 0; --i)  
    cout << a[i] << ' ';                            // Boucle infinie, comportement indéterminé
```

- Avec un type `signed`, la condition de continuation de la boucle `for` peut être soit `i >= 0` (plus intuitif), soit `i < a.size()` (par conversion implicite de `i` en `size_t`)

```
for(ptrdiff_t i = a.size()-1; i < a.size(); --i)  
    cout << a[i] << ' ';                                // affiche 1 4 1 3  
for(ptrdiff_t i = a.size()-1; i >= 0; --i)  
    cout << a[i] << ' ';                            // affiche 1 4 1 3
```

Indices signés ou non-signés ? -＼(ツ)／-



- La *Standard Template Library* d'Alexander Stepanov utilise le type non-signé `size_t` pour les indices des conteneurs
 - Avantages : une seul comparaison nécessaire pour s'assurer qu'un indice est dans l'intervalle $[0, n[$. Pas d'avertissement pour conversion implicite `signed` → `unsigned` dans des expressions telles que `i < a.size()` ou `a[i]`
 - Inconvénient : essayez d'écrire un parcours de l'intervalle d'indices $[a, b]$ par indice décroissant avec `a` qui peut être nul ou pas ...
- Les *C++ Core Guidelines* de Bjarne Stroustrup conseillent d'utiliser le type signé `gsl::index` défini par la *Guidelines Support Library* comme

```
using index = std::ptrdiff_t;
```

ES.107: Don't use `unsigned` for subscripts, prefer `gsl::index`



- Pour parcourir **tous les éléments** d'un conteneur par indice croissant mais **sans connaître l'indice associé à chaque élément**
- La méthode la plus **propre** et la plus **sûre** consiste à utiliser `for(:)`

```
array<double, 4> a {3., 1., 4., 2.};  
  
for (double e : a)  
    cout << e;      // affiche 3142
```

- Attention, ci-dessus `e` est une **copie** des éléments de `a`. Pour les modifier, il faut plutôt en prendre une **référence**.

```
for (double e : a)  
    e *= e;          // a est inchangé et contient [3., 1., 4., 2.]  
  
for (double & e : a)  
    e *= e;          // a contient [9., 1., 16., 4.]
```



- Quand les éléments du conteneur sont eux-mêmes de **type composé**, il faut plutôt choisir entre **référence** et **référence constante** pour éviter les copies inutiles

```
array<string, 4> a{"Hello", ", ", "world", "!\n"};  
  
for (const string & s : a)  
    cout << s;                                // affiche "Hello, world!\n"  
                                                // sans copie inutile de string  
  
for (string & s : a)  
    if (s == "world")  
        s = "class of PRG1";                  // modifie a[2]  
  
for (const string & s : a)  
    cout << s;                                // affiche "Hello, class of PRG1!\n"
```



- Tous les opérateurs de comparaison existent entre deux `std::array<T,n>` de même type : `==`, `!=`, `<`, `>`, `<=`, `>=`
- Deux `std::array<T,n>` sont égaux si tous les éléments de même indice le sont deux à deux
- La comparaison d'inégalité utilise l'**ordre lexicographique**, comme pour les `std::string`

```
array<int,4> a {1, 2, 3, 4};  
array<int,4> b {1, 2, 3, 4};  
array<int,4> c {1, 2, 4, 3};  
  
bool vrais = (a == b) and (a != c) and (a < c) and (c > a);  
bool faux = (a != b) or (a == c) or (a >= c) or (c <= a);
```

array<T, n> et const



- Il y a 2 emplacements et donc théoriquement 4 possibilités de spécifier la constance d'un array<T, n>

array <T, n>

const array <T, n>

array <**const** T, n>

const array <**const** T, n>

- En pratique, les 3 versions avec const se comportent exactement de la même manière, même si ce sont des types différents.
- Il est d'usage d'utiliser **const** array <T, n> et pas les versions avec **const** T pour spécifier un array dont le contenu ne change pas.

Passage en paramètre / en retour



- Un `std::array<T,n>` se comporte comme les autres types du C++
- On peut le passer par **référence**, par **référence constante**, voire par **valeur** si une copie est nécessaire
- On peut le retourner avec **return**

```
array<int, 5> a {1, 2, 3, 4, 5};  
// a = {1, 2, 3, 4, 5}  
  
long long s = somme(a);  
// s = 15  
  
somme_cumulee_1(a);  
// a = {1, 3, 6, 10, 15}  
  
array<int, 5> b = somme_cumulee_2(a);  
// b = {1, 4, 10, 20, 35}  
  
array<int, 5> c = somme_cumulee_3(a);  
// c = {1, 4, 10, 20, 35}  
// a inchangé
```

```
long long somme(const array<int,5> & a) {  
    long long s = 0;  
    for (int e : a) s += e;  
    return s;  
}  
  
void somme_cumulee_1(array<int,5> & a) {  
    for (size_t i = 1; i < a.size(); ++i)  
        a[i] += a[i-1];  
}  
  
array<int,5> somme_cumulee_2(const array<int,5> & a) {  
    array<int,5> b;  
    b.front() = a.front();  
    for (size_t i = 1; i < b.size(); ++i)  
        b[i] = b[i-1] + a[i];  
    return b;  
}  
  
array<int,5> somme_cumulee_3(array<int,5> a) {  
    for (size_t i = 1; i < a.size(); ++i)  
        a[i] += a[i-1];  
    return a;  
}
```



Passage en paramètre / en retour

- `std::array<T1,n1>` et `std::array<T2,n2>` étant des **types différents** dès que `T1 != T2` ou `n1 != n2`, il n'est pas possible à ce stade d'écrire des fonctions qui traitent des `std::array<T,n>` avec `T` et `n` quelconques.
- Nous résoudrons partiellement ce problème avec `std::span<T>` en fin de ce chapitre
- Nous résoudrons entièrement ce problème avec les **fonctions génériques** au chapitre 10

2. `std::vector<T>`



vector<T> vs array<T,n>



- std::array<T,n> souffre de limitations importantes
 - n doit être **constant**
 - n doit être calculable à la **compilation**
 - std::array<T,n1> et std::array<T,n2> sont deux **types différents**
- std::vector<T> permet de dépasser ces limitations
 - en **allouant dynamiquement** le bloc de mémoire où sont stockés les éléments du tableau
 - en le **réallouant** si nécessaire de manière transparente pour l'utilisateur
 - au prix d'un léger **surcoût en mémoire**, typiquement de $3 * \text{sizeof}(T^*)$, comme expliqué au cours d'ASD
- std::vector<T> offre
 - le **même interface** que std::array<T,n> (sauf la méthode `fill`)
 - des méthodes supplémentaires permettant de **redimensionner** le tableau et d'y **ajouter** ou en **supprimer** des éléments



- La classe générique `std::vector` est définie dans le header `<vector>`
- Le type `T` des éléments fait partie du type déclaré, selon la syntaxe

```
std::vector<T> nom_de_variable;
```

- Exemples :

```
std::vector<int> v1;           // tableau redimensionnable de 0 int
std::vector<double> v2;        // tableau redimensionnable de 0 double
std::vector<std::string> v3;    // tableau redimensionnable de 0 std::string
std::vector<std::array<double, 3>> v4;
                                // tableau redimensionnable de 0 tableaux de 3 double
```

- Ainsi initialisé par défaut, un `std::vector<T>` ne contient aucun élément

```
v.size() == 0 and v.empty() == true
```

Initialisation – agrégat et copie



- Comme pour `std::array<T, n>`, les initialisations par agrégat et par copie sont disponibles

```
vector<int> v1 {3, 1, 4, 2};           // v1 contient [3, 1, 4, 1]
vector<int> v2 = v1;                  // v2 contient [3, 1, 4, 1]
```

- Contrairement à `std::array<T, n>`, l'initialisation avec un agrégat partiel n'est pas possible, la taille du `std::vector<T>` étant déterminée par la taille de l'agrégat
- Il n'est pas obligatoire (depuis C++17) de spécifier le type `T` des éléments si celui-ci peut être déduit de l'initialisation

```
vector v3 = v1;                      // v3 de type vector<int>
vector v4 { 3u, 1u, 4u, 2u };        // v4 de type vector<unsigned>
```

Initialisation par remplissage



- Il est également possible d'initialiser un `std::vector<T>` avec une taille et une valeur commune pour tous les éléments
- Syntaxe :

```
std::vector<T> nom(taille, valeur);
```

- Si elle n'est pas spécifiée, `valeur = T();`
- Exemples :

```
vector<int> v1(4, 10);    // v1 contient [10, 10, 10, 10]
vector v2(2, 7u);         // v2 de type vector<unsigned> contient [7, 7]
vector<double> v3(3);    // v3 contient [0., 0., 0.]
vector<bool> v4(2);      // v4 contient [false, false]
```

resize(...) / clear()



- Les méthodes `resize` et `clear` modifient explicitement la taille du vector
- Syntaxe :

```
void resize(size_t taille, const T& val = T());  
void clear(); // équivalent à resize(0);
```

- Quand la taille demandée est plus petite que précédemment, les éléments finaux sont supprimés.
- Quand elle est plus grande, les éléments ajoutés en fin prennent la valeur `val` si elle est spécifiée, ou celle par défaut `T()` sinon
- Exemples : `vector<int> v {2, 3, 5}; // {2, 3, 5}`
`v.resize(5, 1); // {2, 3, 5, 1, 1}`
`v.resize(2); // {2, 3}`
`v.resize(4); // {2, 3, 0, 0}`
`v.clear(); // {}`

push_back(..) pop_back()



- Les méthodes `push_back` et `pop_back` modifient implicitement la taille du vector en ajoutant / supprimant un élément en position finale
- Syntaxe :

```
void push_back(const T& val);  
void pop_back();
```

- Exemples :

```
vector<int> v {2, 3, 5};      // {2, 3, 5}  
v.push_back(1);                // {2, 3, 5, 1}  
v.push_back(4);                // {2, 3, 5, 1, 4}  
v.pop_back();                 // {2, 3, 5, 1}
```

- Equivalence avec `resize` :

```
v.push_back(val);  
v.pop_back();
```

```
v.resize(v.size()+1, val);  
v.resize(v.size()-1);
```



Pour insérer un élément à une position donnée, il faut

- Incrémenter la taille du vecteur
- Déplacer tous les éléments de cette position à la fin du vecteur d'une position vers l'arrière
- Copier le nouvel élément à la place demandée

```
void insertion(vector<int>& v, size_t pos, int val) {
    if (pos > v.size()) // position d'insertion illégale
        return;

    v.resize(v.size()+1); // maintenant, v.size() >= 1
    for(size_t i = v.size()-1; i > pos; --i)
        v[i] = v[i-1];

    v[pos] = val;
}
```

- C'est une opération chère si pos est loin de la fin du vecteur



Pour supprimer un élément à une position donnée, il faut

- Déplacer tous les éléments qui suivent cette position d'une position vers le début
- Décrémenter la taille du vecteur

```
void suppression(vector<int>& v, size_t pos) {  
    if (pos >= v.size()) // position de suppression illégale  
        return;  
  
    for(size_t i = pos + 1; i < v.size(); ++i)  
        v[i-1] = v[i];  
  
    v.pop_back();  
}
```

- Noter que les positions d'insertion et de suppression licites ne sont pas les mêmes
- C'est une opération chère si pos est loin de la fin du vecteur

insert, erase, emplace, emplace_back, ...



- Insertion et suppression sont disponibles via les méthodes `insert` et `erase` de l'interface de `std::vector`, mais font appel à la notion d'**itérateur**. Ils permettent également d'insérer / supprimer une plage d'éléments. Nous les étudierons au chapitre 12. On y présentera aussi d'autres manières d'initialiser et de parcourir un tableau.
- `emplace` et `emplace_back` permettent de **construire** un élément en place dans le vecteur plutôt que de le copier avec `insert` ou `push_back`. Nous les étudierons au chapitre 15.

capacity()



- La **réallocation** de mémoire lors d'un changement de taille est une opération **coûteuse**. Il faut réserver un nouvelle zone mémoire capable de stocker tout le vecteur et y déplacer tous les éléments du vecteur.
- Pour être **efficace**, `std::vector<T>` ne **réalloue** jamais quand la taille diminue et **rarement** quand elle augmente. Pour cela, il distingue
 - la **taille** : nombre d'éléments stockés
 - la **capacité** : nombre d'éléments stockables dans la mémoire actuellement allouée.
- La plupart des `push_back` ont lieu avec `taille < capacité`, et consistent simplement à écrire dans le premier emplacement libre - à l'indice `v.size()` – puis à incrémenter la taille

```
vector<int> v;
size_t c = v.capacity();
cout << c << ' ';
for (int i = 0; i < 1000; ++i) {
    v.push_back(i);
    if (c != v.capacity())
        cout << (c = v.capacity()) << ' ';
}
```

0	1	2	4	8	16	32	64	128	256	512	1024
---	---	---	---	---	----	----	----	-----	-----	-----	------

reserve(), shrink_to_fit()



- Les méthodes `reserve` et `shrink_to_fit` permettent de gérer manuellement la capacité. Elles n'ont **aucun effet sur le résultat** de l'exécution du programme, mais peuvent en **améliorer l'efficacité en temps et en mémoire** si elles sont bien utilisées
- `void reserve(size_t c);`
demande que la capacité soit d'au moins `c`. Elle n'a aucun effet si `c` est déjà le cas. Pertinent si l'on sait a priori combien de `push_back` seront effectués
- `void shrink_to_fit();` demande que la capacité soit diminuée pour être égale à la taille actuelle. Le compilateur n'est pas obligé d'obéir

```
vector<int> v;
v.reserve(1000);
size_t c = v.capacity();
cout << c << ' ';
for (int i = 0; i < 1000; ++i) {
    v.push_back(i);
    if (c != v.capacity())
        cout << (c = v.capacity()) << ' ';
}
```

```
1000
```

vector<T> et const



- Il y a 2 emplacements et donc théoriquement 4 possibilités de spécifier la constance d'un vector<T>

`vector <T>`

`const vector <T>`

`vector <const T>`

`const vector <const T>`

- Mais en pratique, la nature redimensionnable du vecteur le rend **inutilisable avec des éléments de type `const T`**. La plupart des méthodes ne compilent pas.
- On utilise donc toujours `const vector <T>` pour spécifier un vector dont le contenu ne change pas.



3. std::span<T>



Pourquoi C++20 a introduit `span<T>` ?



- Pour rappel, le C++ propose 4 approches pour créer un tableau de n entiers stockés consécutivement en mémoire

```
vector<int> v(n);           // classe std::vector  
array<int,n> a;            // classe std::array  
int t[n];                  // tableau classique, «à la C»  
int* p = new int[n];        // tableau alloué dynamiquement
```

- Il était dommage de devoir écrire 4 fonctions quasi identiques qui traitent similairement ces 4 cas.
- `span<T>` résout ce problème offrant une « vue » commune des 4 cas, tout en ne stockant pas lui-même les éléments.
 - Stocke uniquement l'adresse dans l'objet « vu » de l'élément 0 et le nombre d'éléments
 - Fournit le même interface que `std::array<T,n>` pour lire / écrire les éléments, mais pas les opérateurs de comparaison

Déclaration et initialisation



- La syntaxe de base pour déclarer et initialiser un span est

```
span<T> nom (T* p, size_t n);
```

- Le type T étant déductible du type de p, on peut omettre <T>

```
span nom (T* p, size_t n);
```

- Enfin, on peut initialiser directement depuis un objet qui connaît son nombre d'éléments, i.e. avec un vector, un array ou un tableau C

- Aucun élément n'est copié lors de ces initialisations

```
vector<int> v {1, 2, 3, 4};  
span<int> s1(&(v[0]), v.size());  
span s2(&(v[0]), v.size());  
span s3(v.data(), v.size());  
span s4(v);
```

```
array<unsigned, 3> a {6u, 7u, 8u};  
span s5(a.data(), a.size());  
span s6(a);
```

```
double t[5] = {1., 2., 3., 4., 5.};  
span s7(t, 5);  
span s7(t);
```

```
short* p = new short[3];  
span s8(p, 3);
```

span<T> et const



- Il y a 2 emplacements et donc 4 possibilités de spécifier la constance d'un span

`span<T>`

`span<const T>`

`const span<T>`

`const span<const T>`

- const** devant span indique que l'on ne peut pas changer **quel tableau** le span « voit »
- const** devant T indique que **les éléments** du tableau vu ne peuvent être modifiés par le span
- Seul les `span<const T>` peuvent « voir » un `const vector<T>` ou un `const array<T,n>`

```
vector<int> v1;
span<int> s11(v1);
span<const int> s12(v1);

const vector<int> v2;
// span<int> s2(v2);
span<const int> s2c(v2);

s12 = v2; // change le tableau vu

const span<int> s13(v1);
// s13 = v1;

// les lignes commentées ne compilent pas
```

Passage en paramètre



- Le type `span` est **essentiellement utilisé pour le passage** de tableau en paramètre et rarement dans le code d'une fonction
- On le passe **par valeur**
- Si la fonction **ne modifie pas la valeur** des éléments, on utilise `span<const T>`
- Si la fonction **modifie la valeur** des éléments, on utilise `span<T>`
- La fonction **ne peut pas modifier le nombre** d'éléments

```
void display(span<const int> s) {
    cout << '[';
    for (int e : s)
        cout << e << ',';
    cout << "\b]\n";
}
```

```
void fill(span<int> s, int val) {
    for (int& e : s)
        e = val;
}
```

```
vector<int> v{1, 2, 3, 4};
const array<int, 3> a{5, 6, 7};
```

```
display(v);
display(a);
fill(v, 42);
display(v);
```

```
[1,2,3,4]
[5,6,7]
[42,42,42,42]
```

.subspan, .first, .last



- On peut définir un span qui ne voit **qu'une partie d'un tableau**, en partant de l'adresse d'un élément d'indice autre que 0, et/ou en spécifiant une taille inférieur à celle du tableau
- A partir d'un span s, la méthode **s.subspan(pos,len)** crée un span équivalent à `span(&s[pos],len)`
- **s.first(len)** est équivalent à `s.subspan(0,len)`
- **s.last(len)** est équivalent à `s.subspan(s.size()-len,len)`

```
vector<int> v{1, 2, 3, 4, 5, 6};
span s1(&v[1],2); // s1 voit {2, 3}
```

```
span s2 = span(v).subspan(1,2);
// s1 et s2 voient les même éléments
```

```
span s3(&v[0],3);
span s4 = span(v).first(3);
// s3 et s4 voient les même éléments
```

```
span s6(&v[v.size()-2],2);
span s7 = span(v).last(2);
// s6 et s7 voient les même éléments
```

4. Tableau multi-dimensionnels





- Un tableau peut contenir des éléments qui sont eux aussi des tableaux. On parle alors de tableau multi-dimensionnel.
- On peut combiner à choix std::vector et std::array
- Par exemple, voici 4 manières de déclarer et initialiser à 0 un tableau de **4 lignes et 3 colonnes** de double

```
array<array<double, 3>, 4> m1 {};  
  
vector<vector<double>> m2(4, vector<double>(3, 0.));  
  
vector<array<double, 3>> m3(4, array<double, 3>{ {} });  
  
array<vector<double>, 4> m4;  
m4.fill(vector<double>(3, 0.));
```

- Pour de petits tableaux de taille fixe, std::array est préférable
- Si le type des tableaux imbriqués est std::vector, le tableau multi-dimensionnel n'est pas nécessairement rectangulaire, chaque ligne pouvant avoir un nombre de colonnes différent

Accès aux éléments



- On accède aux éléments avec autant d'opérateurs [] (ou de méthodes .at()) qu'il y a de dimensions. Par ex.,

```
array<array<double, 3>, 4> m{};  
m[2][1] = 1;
```

- La signification des 2 indices est plus claire en observant que l'on peut aussi écrire la même opération en 2 lignes

```
array<double, 3>& ligne = m[2];  
ligne[1] = 1;
```

- Pour afficher tout le tableau 2d, on écrirait

```
for (size_t i_ligne = 0; i_ligne < m.size(); ++i_ligne) {  
    for (size_t i_colonne = 0; i_colonne < m[i_ligne].size(); ++i_colonne)  
        cout << m[i_ligne][i_colonne] << ' ';  
    cout << endl;  
}
```

0	0	0
0	0	0
0	1	0
0	0	0

Utilisation d'alias de type



- Le code peut éventuellement être rendu plus compréhensible en utilisant les alias de type avec `typedef` ou `using`

```
using Ligne3 = array<double, 3>;
using Matrice3x3 = array<Ligne3, 3>

Matrice3x3 matrice;

for (Ligne3& ligne : matrice) {
    for (double element : ligne)
        cout << element << ' ';
    cout << endl;
}
```

0	0	0
0	0	0
0	0	0



Note : Le standard C++23 introduira `std::mdspan`, un *span* qui « voit » un tableau 1d sous la forme d'une matrice à plusieurs dimensions, dont les éléments seront accessibles via un opérateur `[]` qui accepte des indices multidimensionnels

En septembre 2023, cet ajout n'est encore supporté que par le compilateur clang 17.



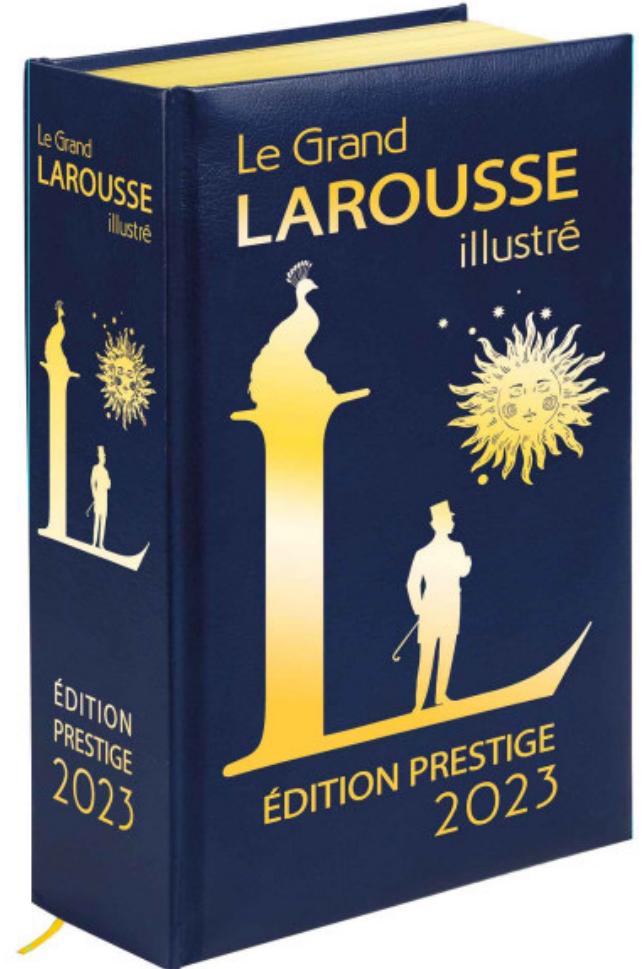
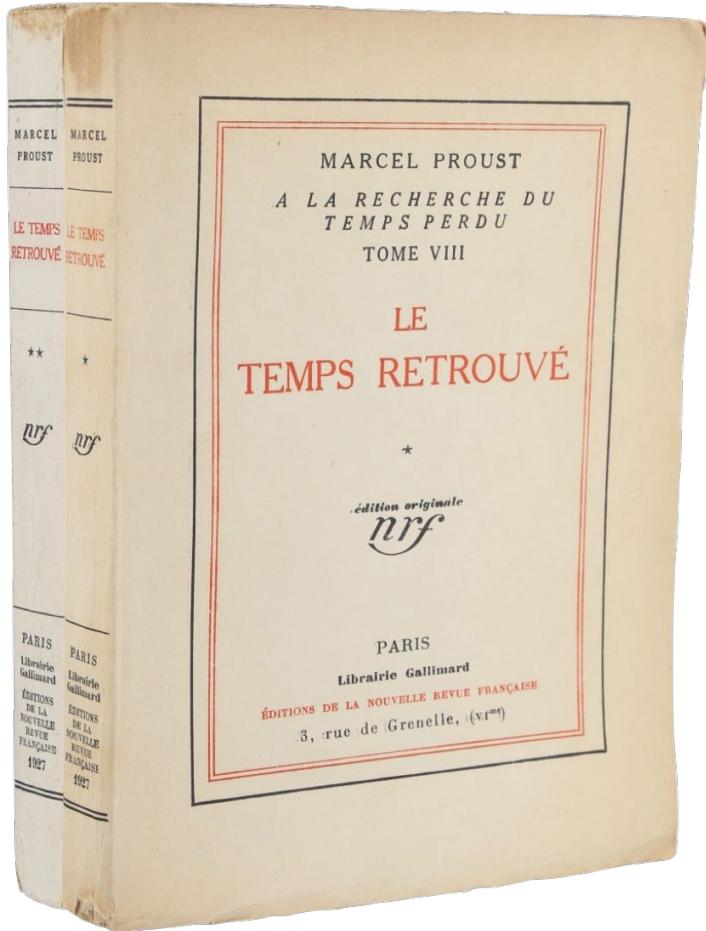
5. Tri à bulles



Pourquoi trier ?



- Le mot « léonardesque » apparait exactement une fois dans chacun de ces deux livres. Dans lequel le retrouverez-vous le plus vite ?



Comment trier ?



- La page wikipedia en anglais de chaque algorithme de tri ce termine par ces liens

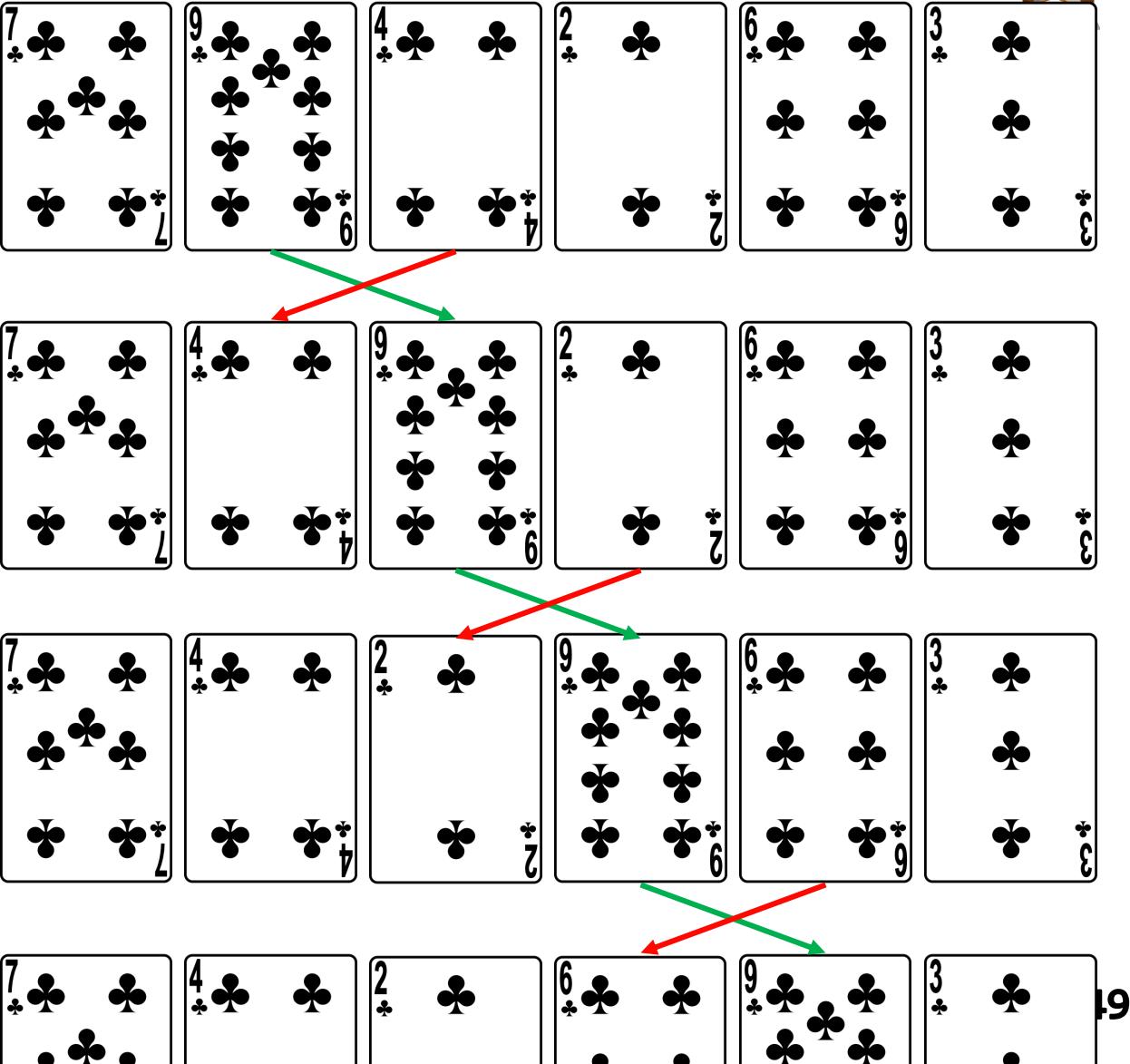
V · T · E	Sorting algorithms	[hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting · X + Y sorting · Transdichotomous model · Quantum sort	
Exchange sorts	Bubble sort · Cocktail shaker sort · Odd–even sort · Comb sort · Gnome sort · Proportion extend sort · Quicksort	
Selection sorts	Selection sort · Heapsort · Smootsort · Cartesian tree sort · Tournament sort · Cycle sort · Weak-heap sort	
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting	
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort	
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsor · Counting sort · Interpolation sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort	
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network · Samplesort	
Hybrid sorts	Block merge sort · Kirkpatrick–Reisch sort · Timsort · Introsort · Spreadsort · Merge-insertion sort	
Other	Topological sorting (Pre-topological order) · Pancake sorting · Spaghetti sort	
Impractical sorts	Stooge sort · Slowsort · Bogosort	

- En **PRG1**, nous étudions les tris à bulle, par sélection, et par insertion
- En **ASD**, vous étudierez les tris rapide, par tas, par arbre, fusion, comptage, et par base



Principe du tri à bulles

- Comparer l'une après l'autre toutes les paires d'éléments consécutifs
 - $(7 < 9) ? , (9 < 4) ? , \dots$
- Si une paire est dans le mauvais ordre, échanger les éléments
 - Échanger 4 et 9
- Répéter l'opération suffisamment de fois pour que le tableau soit trié





- Au cœur de l'algorithme, on échange 2 éléments du tableau.
- La fonction `echanger`
 - reçoit les lvalues à échanger par référence
 - utilise une variable temporaire `t` pour stocker la valeur de la lvalue dans laquelle on écrit en premier
- Il n'est pas nécessaire d'écrire notre propre fonction `echanger`, le header `<utility>` fournissant la fonction générique `std::swap(T&, T&)`

```
void echanger (int& a, int& b) {  
    int t = a; a = b; b = t;  
}  
  
int main() {  
    vector<int> v{7, 9, 4, 2, 6, 3};  
    echanger(v[1], v[2]);  
    for(int e : v) cout << e << ' ';  
}
```

7 4 9 2 6 3

```
#include <utility>  
using std::swap;  
  
int main() {  
    vector<int> v{ 7, 9, 4, 2, 6, 3};  
    swap(v[1],v[2]);  
    for(int e : v) cout << e << ' ';  
}
```

7 4 9 2 6 3



- Trier requiert une notion d'ordre.
- Traditionnellement - et dans la STL - on trie par défaut du plus petit au plus grand élément en utilisant l'opérateur <
- Le test pour savoir s'il faut changer l'ordre de la paire d'éléments consécutifs d'indices i et $i-1$ est

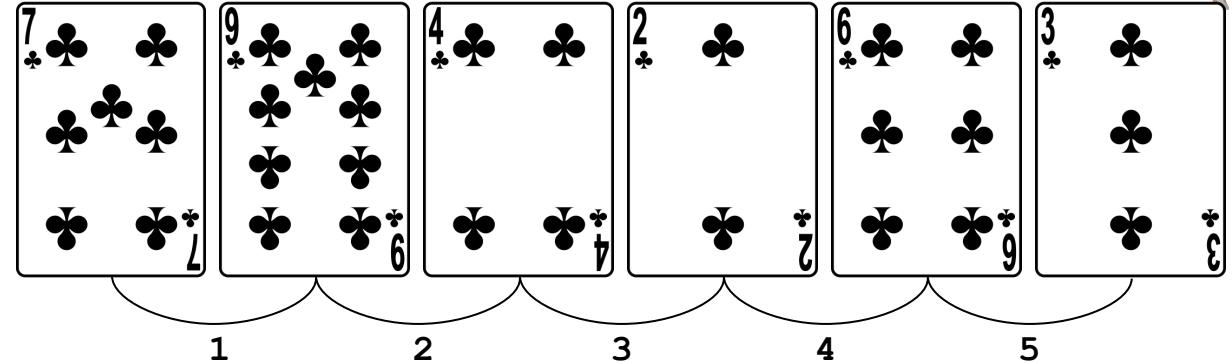
```
if (v[i] < v[i - 1])
    swap(v[i], v[i - 1]);
```

- On n'utilise pas l'opérateur \leq , car il est inutile – voir néfaste – d'échanger des éléments de valeurs égales.

Toutes les paires d'éléments consécutifs



- Pour un tableau de n éléments, on a $n-1$ paires d'éléments consécutifs



- On peut boucler soit
 - sur les paires d'indices $(i, i+1)$ avec i allant de `0` à `v.size()-2`
 - sur les paires d'indices $(i-1, i)$ avec i allant de `1` à `v.size()-1`

```
for (size_t i = 0; i < v.size() - 1; ++i)
    if (v[i + 1] < v[i])
        swap(v[i], v[i + 1]);
```

```
for (size_t i = 1; i < v.size(); ++i)
    if (v[i] < v[i - 1])
        swap(v[i], v[i - 1]);
```

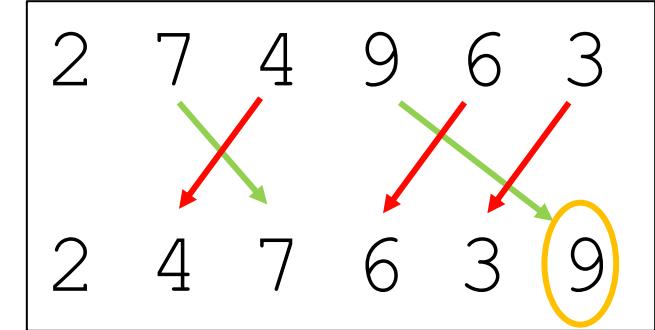
Effet d'une boucle sur toutes les paires



- Observons l'effet d'un passage sur toutes les paires

```
void afficher(span<const int> v) {
    for (int e: v) cout << e << ' '; cout << endl;
}

int main() {
    vector<int> v{ 2, 7, 4, 9, 6, 3};
    afficher(v);
    for (size_t i = 1; i < v.size(); ++i)
        if (v[i] < v[i - 1])
            swap(v[i], v[i - 1]);
    afficher(v);
}
```



- L'**élément le plus grand** se retrouve en position finale. Au passage suivant, il n'est donc pas nécessaire de le traiter, i.e. de traiter la paire finale
- Aucun élément ne **recule** de plus de une position. Dans le pire cas - minimum initialement placé en position finale - il faut $n-1$ passages en tout

Mise en oeuvre complète et exécution



```
void tri_a_bulle (span<int> v) {
    size_t end = v.size(); // dernière paire à traiter
    for (size_t j = 1; j < v.size(); ++j) {
        // v.size() - 1 boucles sur les paires d'éléments
        for (size_t i = 1; i < end; ++i)
            if (v[i] < v[i - 1]) {
                swap(v[i], v[i - 1]); afficher(v);
            }
        --end; // traiter une paire de moins à chaque passage
    }
}

int main() {
    vector<int> v{7, 9, 4, 2, 6, 3}; afficher(v); tri_a_bulle(v);
}
```

7	9	4	2	6	3
7	4	9	2	6	3
7	4	2	9	6	3
7	4	2	6	9	3
7	4	2	6	3	9
4	7	2	6	3	9
4	2	7	6	3	9
4	2	6	7	3	9
4	2	6	3	7	9
2	4	6	3	7	9
2	4	3	6	7	9
2	3	4	6	7	9

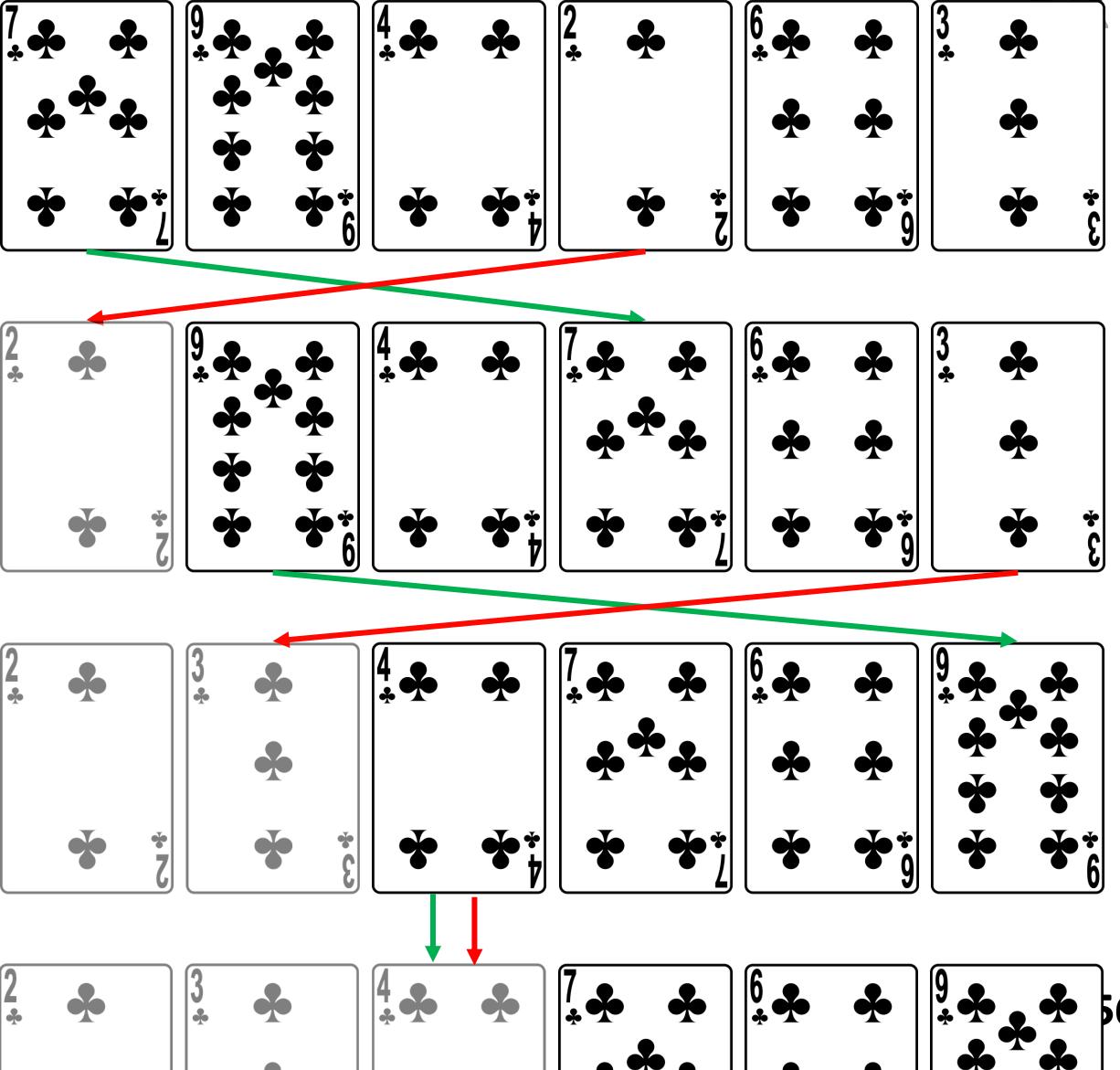
- La boucle extérieure peut aussi être un boucle `while(change)` avec la variable booléenne `change` initialisée à `false` et qui passe à `true` si un échange à lieu dans la boucle intérieure

6. Tri par sélection





- Trouver le plus petit élément du tableau
- Le placer en première position en l'échangeant avec l'élément qui y réside
- Répéter avec le reste du tableau jusqu'à ce que tout le tableau aie été trié



Trouver l'indice du plus petit élément



- Pour pouvoir l'échanger ensuite, il faut trouver l'**indice** `imin` du plus petit élément, pas juste sa valeur.
- Pour cela, on parcourt le tableau et on compare chaque élément au plus petit trouvé jusque là.

Note : Au chapitre 12, nous verrons que la librairie `<algorithm>` fournit une fonction générique `std::min_element` qui met en œuvre cette recherche avec des itérateurs

```
size_t indice_min (span<const int> v) {  
    size_t imin = 0;  
    for (size_t i = 1; i < v.size(); ++i)  
        if (v[i] < v[imin])  
            imin = i;  
    return imin;  
}  
  
int main() {  
    vector<int> v{7, 9, 4, 2, 6, 3};  
    size_t imin = indice_min(v);  
    cout << "v[" << imin << "] = "  
        << v[imin] << endl;  
}
```

v [3] = 2



- Boucle extérieure allant de l'indice 0 à l'avant dernier, cherchant le minimum de cet indice à la fin du tableau
- Il n'est pas nécessaire de traiter le sous-tableau final de 1 élément

```
void tri_par_selection (span<int> v) {  
    for (size_t i = 0; i < v.size()-1 ; ++i) {  
        size_t imin = i + indice_min(v.subspan(i));  
        swap(v[i], v[imin]); afficher(v);  
    }  
}  
  
int main() {  
    vector<int> v{7, 9, 4, 2, 6, 3}; afficher(v);  
    tri_par_selection(v);  
}
```

7	9	4	2	6	3
2	9	4	7	6	3
2	3	4	7	6	9
2	3	4	7	6	9
2	3	4	6	7	9
2	3	4	6	7	9

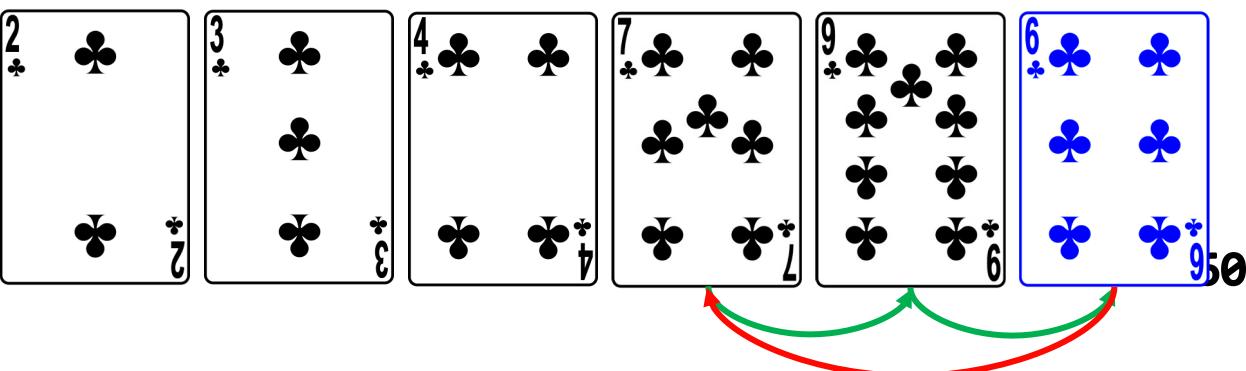
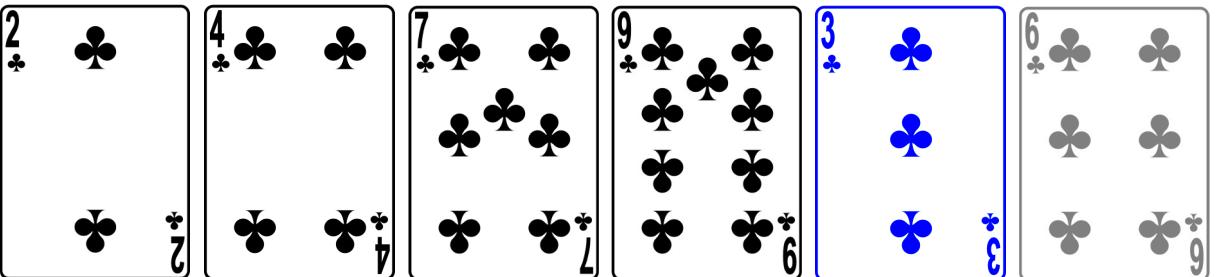
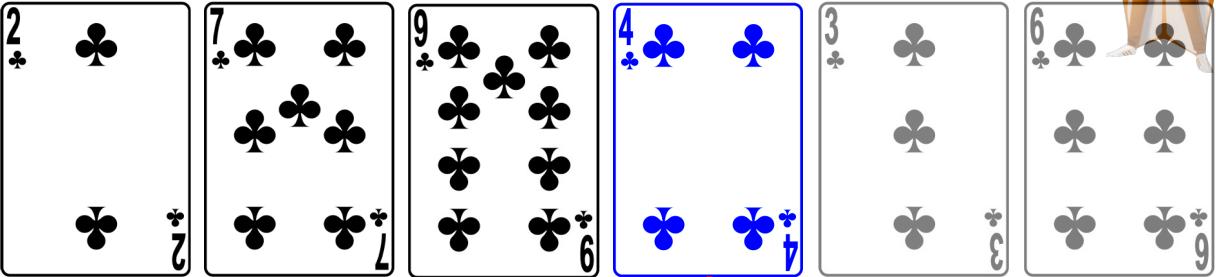


7. Tri par insertion





- Le début du tableau étant trié, on insère l'élément suivant dans la partie triée en
 - le plaçant dans une variable temporaire
 - déplaçant d'une position vers l'arrière les éléments qui le précèdent et sont strictement plus grands
 - écrivant la valeur temporairement sauvee dans l'emplacement libéré
- Commencer en insérant le 2^{ème} élément, finir en insérant le dernier





Insérer un élément

- Si l'on considère le sous-tableau de n éléments qui contient
 - les premiers éléments triés
 - l'élément à insérer en dernière position
- L'insertion ressemble à un swap

```
int t = a; a = b; b = t;
```

à plusieurs variables, i.e. à une permutation circulaire

```
int t = v[n-1]; v[n-1] = v[n-2];
v[n-2] = v[n-3]; ... v[n-k] = t;
```

- dont le nombre d'éléments impliqués k est déterminé en testant que $v[n-k]$ est le plus petit élément $> t$

```
void inserer_dernier_element(span<int> v) {
    if (v.size() < 2) return;
    int t = v.back();
    size_t i = v.size()-1;
    for (; i > 0 and t < v[i-1]; --i)
        v[i] = v[i-1];
    v[i] = t;
}

int main() {
    vector<int> v{2, 7, 9, 4, 3, 6};
    afficher(v);
    inserer_dernier_element(span(v).first(4));
    afficher(v);
    return 0;
}
```

2	7	9	4	3	6
2	4	7	9	3	6



- Pour trier tout le tableau, il faut insérer tous les éléments du 2^{ème} au dernier dans la partie triée du tableau qui les précède
- Il suffit d'appeler `inserer_dernier_element` avec des sous-tableaux de taille croissante

```
void tri_par_insertion(span<int> v) {
    for (size_t i = 2; i <= v.size(); ++i) {
        inserer_dernier_element(v.first(i));
        afficher(v);
    }
}

int main() {
    vector<int> v{9, 2, 7, 4, 3, 6}; afficher(v);
    tri_par_insertion(v);
}
```

9	2	7	4	3	6
2	9	7	4	3	6
2	7	9	4	3	6
2	4	7	9	3	6
2	3	4	7	9	6
2	3	4	6	7	9



8. Comparaison des tris



Comment comparer ?



Pour évaluer un algorithme de tri, on peut se baser sur divers critères

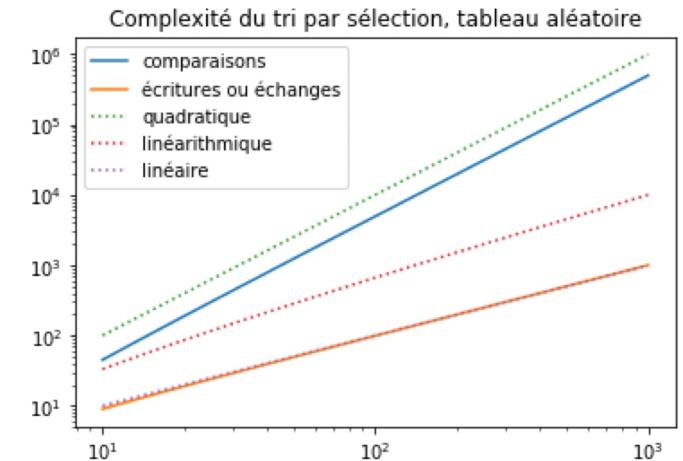
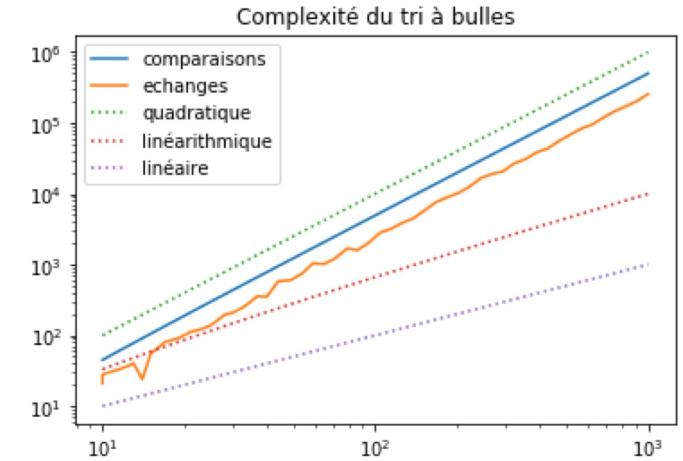
- **Complexité temporelle** : combien de temps pour trier n éléments ? Combien d'opérations élémentaires / comparaisons / écritures / ... ?
- Complexité spatiale : quelle quantité de mémoire est nécessaire ?
 - tous les tris présentés requièrent uniquement une variable tampon.
- **Stabilité** : les éléments égaux selon le critère de tri gardent-ils leur ordre relatif après tri ?

Note : Les tris les plus efficaces seront présentés au cours d'ASD

Complexités des deux tris par échange



- Le tri **à bulles** (avec une double boucle for) et **par sélection** font le même nombre de **comparaisons** : $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$
- Dans le tri **à bulles**, en moyenne une comparaison sur deux est suivie d'un **échange**, il y en a donc $\approx \frac{n^2}{4}$, mais cela dépend des données à trier
- Dans un tri **par sélection**, il y a exactement $n - 1$ échanges



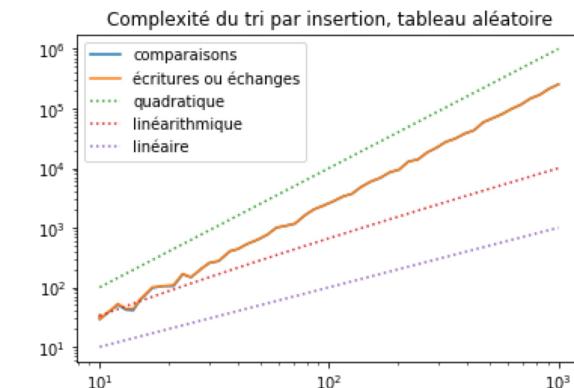
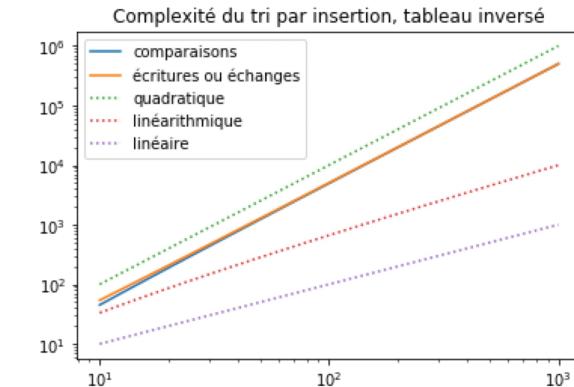
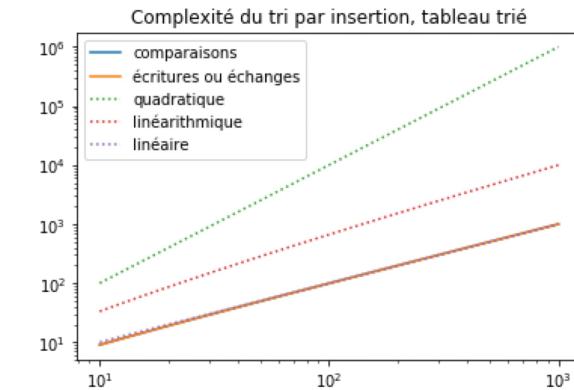


Complexité du tri par insertion

La complexité du tri par insertion varie selon les données à trier.

- Dans le **meilleur cas**, les données sont déjà triées. La boucle de la fonction `inserer_dernier_element` ne s'exécute jamais, et le nombre de comparaisons est de $n - 1$
- Dans le **pire cas**, le tableau est trié à l'envers et la boucle de `inserer_dernier_element` s'arrête via le test $i > 0$. Le nombre de comparaisons effectuées est $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$
- En **moyenne**, les insertions avancent d'environ la moitié du tableau et le nombre de comparaison est $\approx \frac{n^2}{4}$

Le tri par insertion est particulièrement **efficace quand les données sont presque triées**, ou quand le tableau est de petite taille. C'est le seul des trois qui est largement utilisé en pratique





- Si vous triez le tableau Excel ci-contre par note obtenue, les élèves ayant la même note restent ordonnés comme dans le tableau original.
- Excel utilise pour cela un **tri stable** : les éléments égaux selon le critère de tri conservent leur ordre relatif d'avant le tri
- C'est une propriété utile, qui permet de remplacer un tri selon un critère complexe par plusieurs tris successifs avec des critères plus simples

Elève	Note
Alice	A
Benoît	B
Chiara	A
Damien	C
Elena	A
Fabienne	C
Gaston	B
Hélène	A
Isabelle	C
Jonathan	B
Damien	C
Fabienne	C
Isabelle	C

Tri par note



- Le **tri à bulle** n'effectue d'échange que s'il y a une inégalité stricte entre les éléments. Les éléments égaux ne sont pas permutés. Le tri est **stable**

```
if (v[i] < v[i - 1]) swap(v[i], v[i - 1]);
```

- Le tri **par insertion** ne déplace un élément que s'il est strictement plus grand que celui à insérer. Le tri est **stable**

```
for (; i > 0 and t < v[i-1]; --i) v[i] = v[i-1];
```

- Le tri **par sélection** échange l'élément $v[i]$ uniquement parce qu'il est à l'emplacement i , indépendamment de sa valeur. Le tri est **instable**

```
swap(v[i], v[imin]);
```