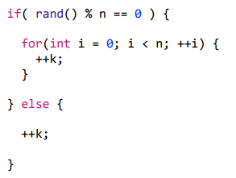
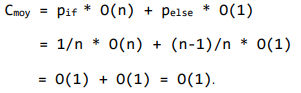


En cas d’enchaînement, il faut prendre en compte les différentes parties du code.

Meilleur cas : O(1) (instruction else)

Pire cas : O(n) (instruction if)

Cas moyen :



f = O(g) 🡪 f croit au plus aussi vite que g

f = o(g) 🡪 f croit strictement plus lentement que g.

f = Ω(g) 🡪 f croit au moins aussi vite que g (inverse de grand O), ce qui veut dire g = O(f).

f = ϴ(g) 🡪 f et g ont le même ordre de grandeur.

**Règles :**

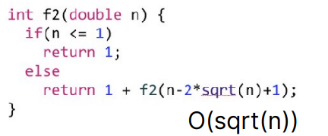
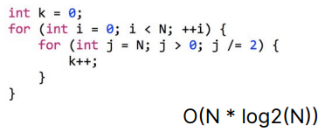
1. Si une fonction est la somme de plusieurs termes, si l’un d’eux croit plus vite que les autres, on ne garde que lui et on ignore les autres.
2. Si une fonction est le produit de plusieurs facteurs, on peut ignorer tout facteur constant.

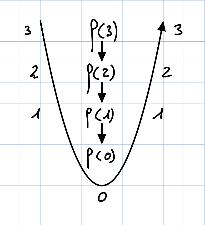
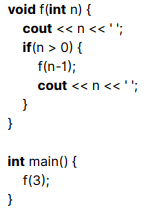
**Rappels :**

-

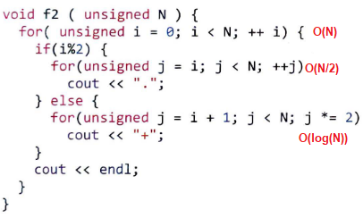
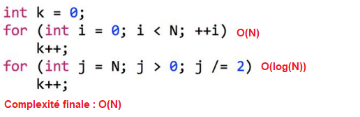
-

-





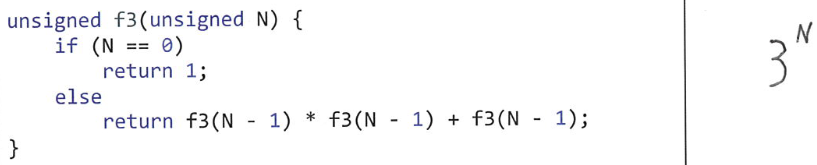
**ATTENTION :** Si les boucles ne sont pas imbriquées, on prend la complexité la plus élevée.



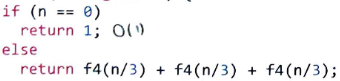
O(N^2)

**NE PAS OUBLIER**: Des codes à la suite les complexités s’additionnent, mais les codes qui s’imbriquent, elles se multiplient.

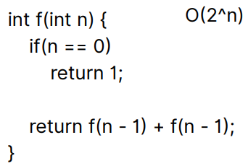
Si on appelle x fois la fonction et que le paramètre diminue de 1 à chaque fois on .

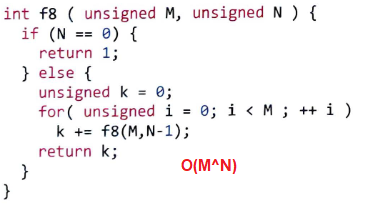


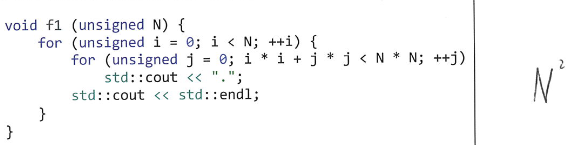
Un second exemple pourrait être d’appeler x fois la fonction et de diviser par 3 à chaque fois, dans ce cas

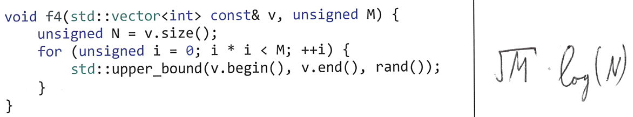


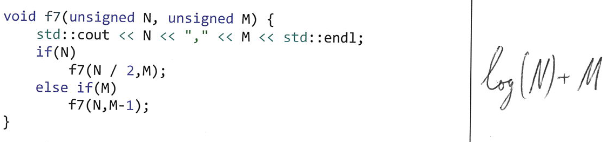
Encore un exemple du même style

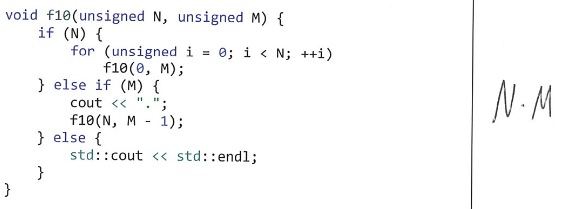












**Fonctions de la STL :**

* std::sort :
* std::stable\_sort : si assez de mémoire
* std::partial\_sort : où et
* std::nth\_element :
* std::find :
* std::generate : , n étant la taille du vecteur à remplir
* std::upper/lower\_bound :
* std::equal\_range :
* std::binary\_search : :
* std::distance :
* std::max/min\_element :
* std::swap :

**Si on a des appels à (f-1) et (f-2), la complexité est équivalent à**

lower\_bound : Retourne un itérateur du premier élément qui n’est pas plus petit que la valeur passée (plus grand ou égal).

upper\_bound : Retourne un itérateur du premier élément qui est plus grand que la valeur

binary\_search : Cherche un élément dans une plage donnée, true si trouvé sinon false.

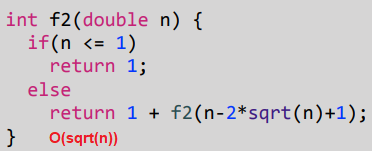
equal\_range : Retourne une plage contenant les valeurs à chercher (dans une liste triée) ce sont deux itérateurs, un du début de la plage (first) et un de la fin (second).

Paramètres de récursivité : Paramètres changeant à chaque exécution

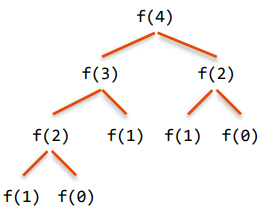
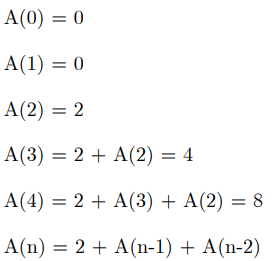
Cas trivial : Termine la fonction

Cas général : Effectue un nouvel appel récursif

La complexité de la fonction étant constante, on calcule via le nombre d’appels récursifs. Le mieux reste d’essayer avec un nombre pas trop grand.



Fibonacci est un très bon exemple d’algorithme récursif et qui permet de détailler simplement la manière de les comprend. Pour rappel Fibonacci indique que pour calculer . On va donc dessiner un arbre des appels récursifs. De cette manière, on peut établir combien il nous faut d’appel pour chaque niveau.



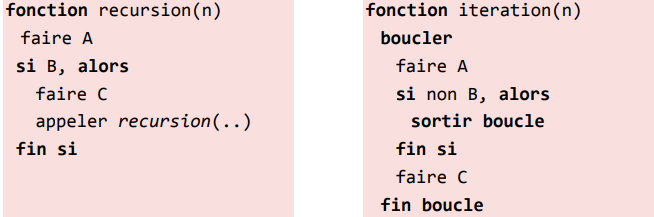
Les fonctions de types Fibonacci appelant l’avant dernier et l’avant avant dernier pallier ont une complexité particulière en . C’est le cas pour le pire cas de l’algorithme d’Euclide (a % b = a - b).

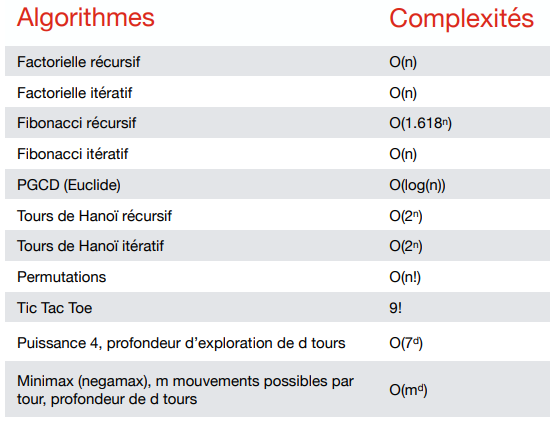
Encore un exemple si on a 4 lettres à placer dans n’importe quel ordre. La complexité sera de , donc on a 4 lettres placées en première place, puis 3 à mettre, etc…

L’algorithme **MiniMax** a pour but de minimiser les pertes maximums. C’est-à-dire qu’il assigne à chaque coup possible un certain score et le coup ayant le meilleur score est celui choisi. La complexité de ce type d’algorithme est assez difficile à établir. Mais en général, le joueur peut choisir entre M coups, on explore alors N coups successifs. Ce qui fait un algorithme en . **NegaMax** c’est une version où le score d’une position pour un joueur est l’inverse pour l’autre.

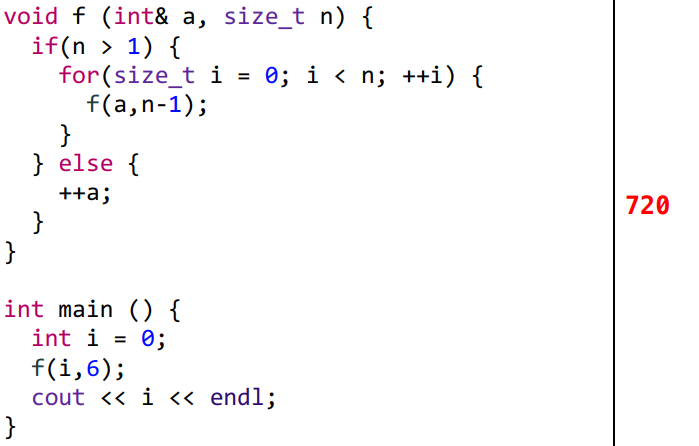
**L’élagage alpha-beta** c’est d’arrêter la recherche de minimum beta s’il est plus petit que le max (alpha) au niveau au-dessus. Et arrêter la recherche de maximum alpha s’il est déjà plus grand le minimum (beta) au niveau au-dessus. Cela permet de couper les branches de l’arbre qui ne donnent pas un résultat satisfaisant. Permet de doubler la profondeur d’exploration et dans le pire cas, il n’apporte aucune amélioration.

Fibonacci 🡪 Récursif et Itératif . Il faut faire attention à ne pas dépasser la pile d’appels, ce qui peut arriver notamment avec la somme de N premiers entiers qui récursivement explose la pile. Récursion finale, si la dernière opération est un appel récursif alors on peut le faire itérativement.

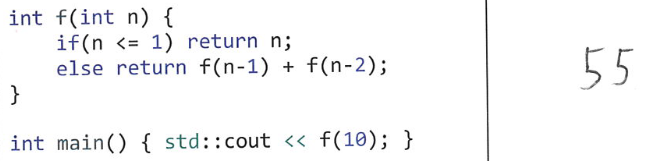




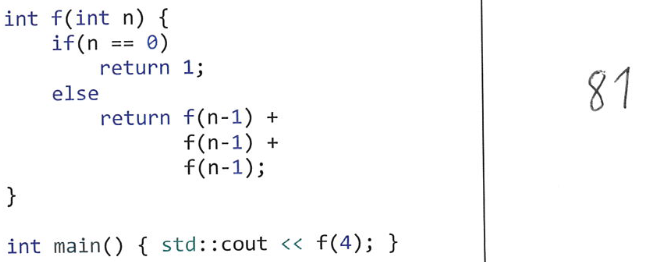
Exemple d’algorithme de factorielle :

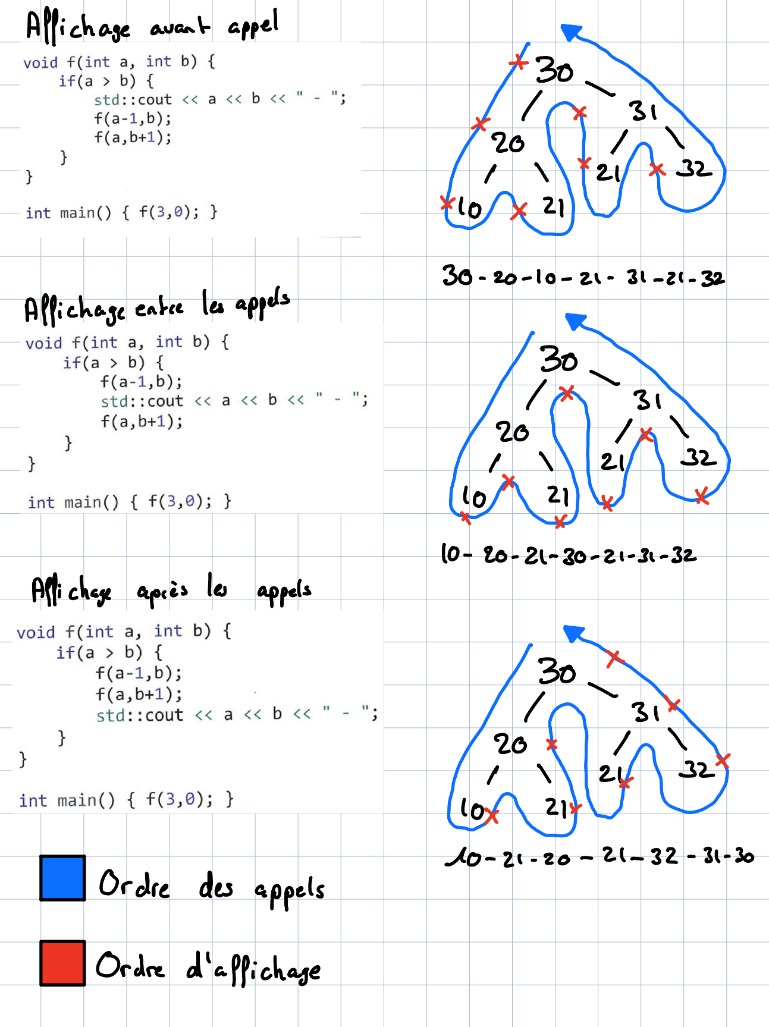


Exemple d’algorithme Fibonacci :



Algorithme s’appelant plusieurs fois, , ici n = 4 :



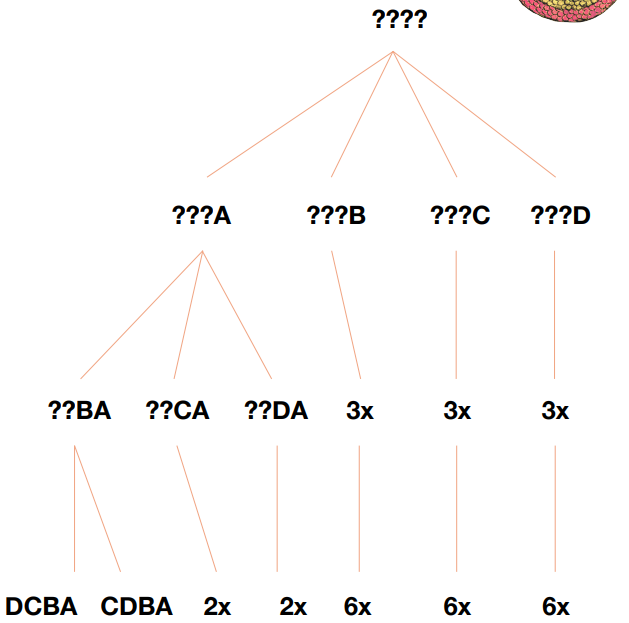
Arbre des appels récursifs :

**Fibonacci :** Appel à (f-1) et (f-2)

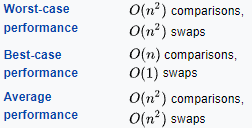
**Euclide**: Appel à f(b, a%b)

**Hanoï**: Appel 2 fois à f(n-1)

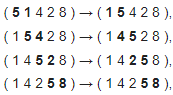
**Permutations :** (n) appel n-fois (n-1), etc…



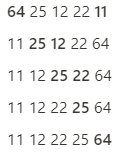
**Tri à bulles : comparaisons. Complexité de 0 à**  échanges dans le pire cas. Stable s’il n’échange pas les éléments égaux. Il faut le voir comme des bulles qui remontent à la surface, il parcourt le tableau du début vers la fin et amène à la fin l’élément le plus grand. Il fait cela en boucle jusqu’à avoir parcouru la taille du tableau. Dans un tri intermédiaire, on retrouve alors toujours les plus grands éléments en fin.

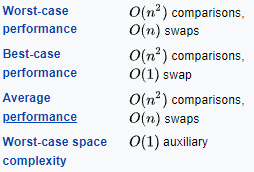


Exemple d’une première passe

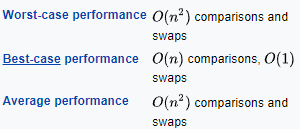


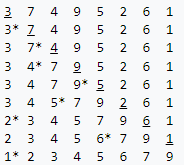
**Tri par sélection :**  **comparaisons. Complexité en**échanges peu importe l’ordre des entrées. Non-stable. Cherche le minimum et le met à la position courant via un échange. Ramène au début les éléments les plus petits.

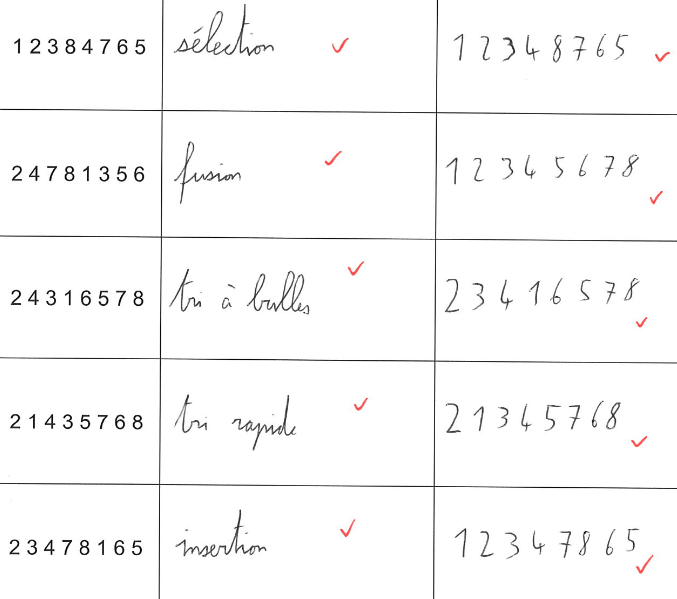




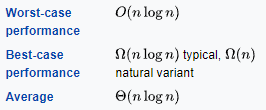
**Tri par insertion :** **Meilleur cas : pour tout et pire cas/cas moyen .** A chaque passe, on le prend l’élément suivant et on le compare à tous les éléments de gauche. S’il est plus petit on le décale, sinon on le laisse à sa place. On retrouve tous les éléments triés vers le début, mais pas forcément les plus petits (ce qui serait le cas pour un sélection), voir exemple avec le 1.

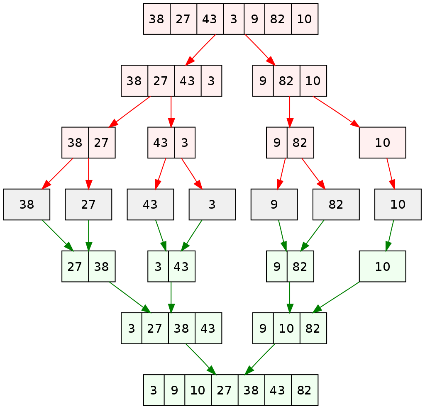




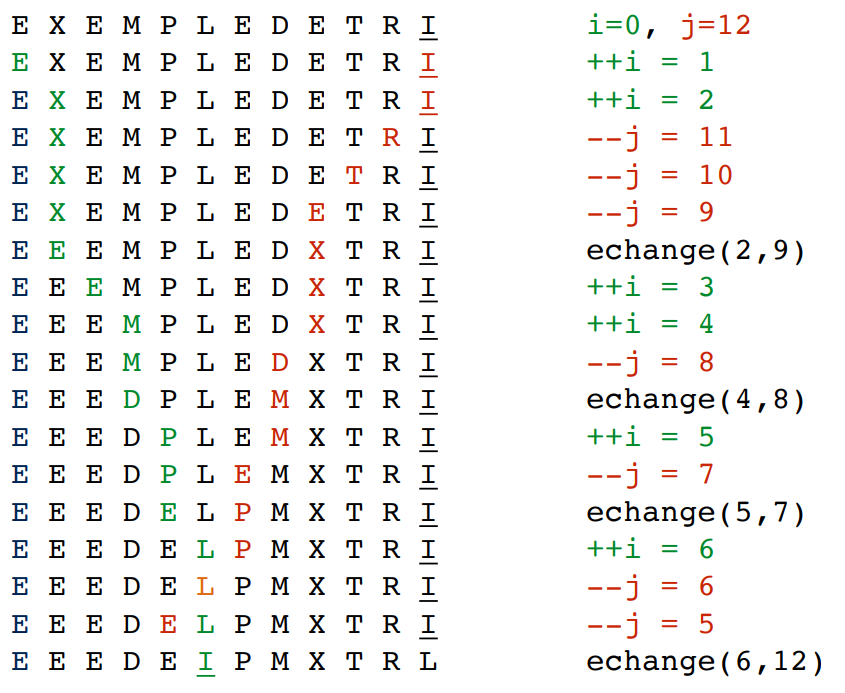
****

**Tri par fusion :** Algorithme récursif dans lequel on essaye de trier un tableau en le séparant en tableaux de 1 élément trié. Une fois les tableaux séparés, on les fusionne au fur et à mesure. Les petits tableaux sont dans l’ordre, il suffit de comparer à chaque fois le premier élément de chaque tableau et de le mettre dans le tableau de destination. La complexité est linéarithmique. Il n’est pas en place, mais il est stable. Pas adapter aux petits tableaux.

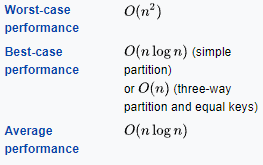




**Tri rapide :** On choisit un pivot et on le met en fin de tableau, puis on met à gauche tous les éléments plus petits et à droite les plus grand. Il est stable et en place. Pour se faire, on a deux itérateurs qui s’arrêtent lorsque un élément est à la mauvaise place et les inverse. La manipulation s’arrête lorsqu’ils se croisent. **Si les valeurs sont égales on inverse !!!**

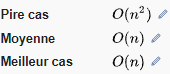


Dans le meilleur cas le pivot c’est la médiane des éléments à partitionner, on divise alors la partition par deux, complexité en (pareil en moyenne). Dans le pire cas, c’est une valeur extrême et il faut faire n partitions. La complexité est alors en . Pas adapter pour les petits tableaux, on fait alors un tri hybride avec un tri par insertion.

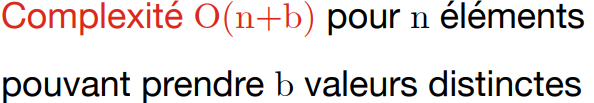


Si on utilise une comparaison non-stricte et que beaucoup d’éléments se répètent, la complexité est en . échanges si tableau trié.

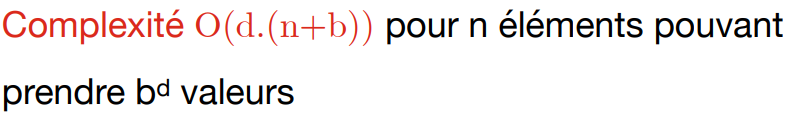
**Sélection rapide :** Permet de trouver le kième plus petit élément dans un tableau. On utilise le même algorithme de partition qu’un tri rapide, mais ne continue que du côté pertinent.



**Tri comptage :** Utilise les propriétés des données (typiquement avec les cartes). Il va compter chaque type de données et laisse de la place dans un nouveau tableau pour les placer. Il utilise 2 tableaux pour stocker les éléments (source, destination) et un tableau pour les compteurs. Stable, pas en place.

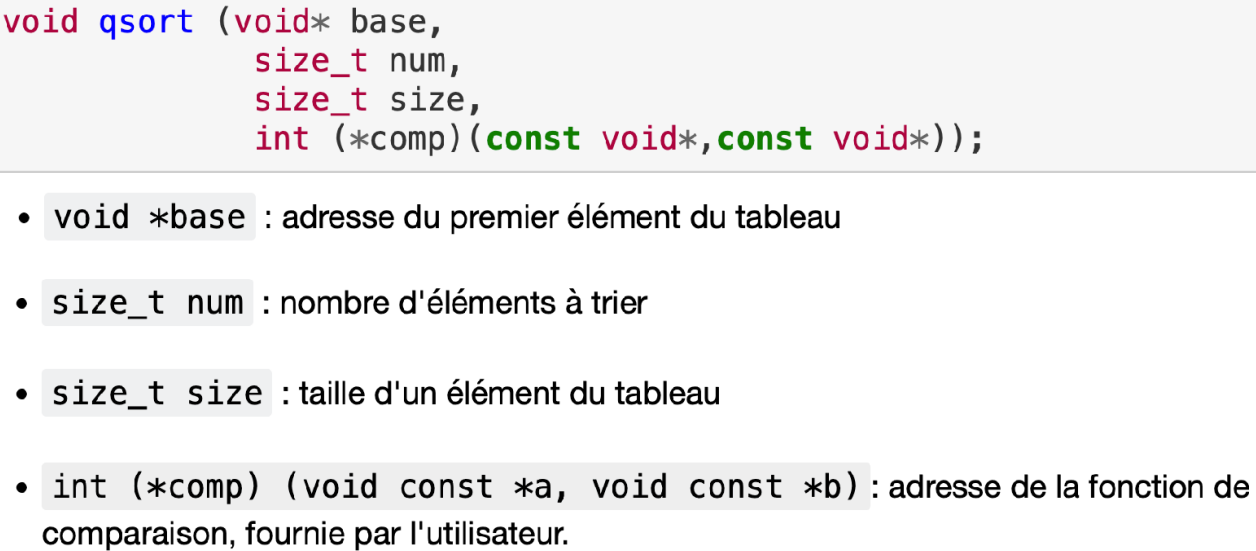


Tri par base : On trie les tableaux chiffres par chiffres (unités, dizaines, etc…). Il est stable et pas en place.



La caractéristique que l’on veut utiliser pour regrouper les cartes c’est la dernière que l’on doit trier. Ex : Avoir un jeu de cartes triés avec des groupes d’enseignes et dans ces groupes triés par numéro. On fait d’abord les numéros, puis les enseignes. Ex.2 : Pour des chiffres, on fera d’abord les unités, puis les dizaines.

**std::qsort :** Il n’y a pas de garantie sur la complexité de l’algorithme. En pratique on est sur du linéarithmique ou quadratique dans le pire cas. Pas stable.

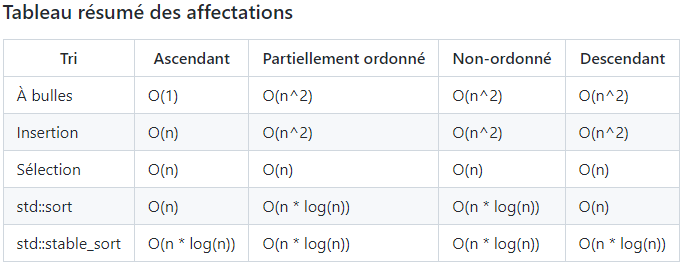
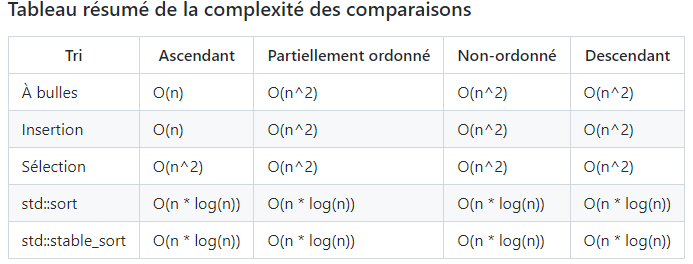


**std::sort :** Complexité linéarithmique, variation du **quicksort**, pas stable.

**std::stable\_sort** : (si on utilise une fonction de comparaison personnalisée, il faut avoir une inégalité stricte : < ou >). C’est une variation du **tri fusion**, donc il faut que std::move soit implémenté pour le type.

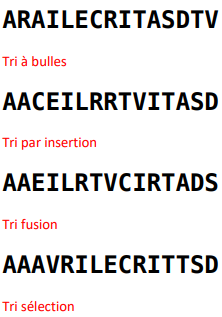
**std::nth\_element** **:** Prend l’élément les itérateurs d’un tableau et au milieu le pilier pour partitionner et fait une partition (même complexité que la sélection rapide).

**std::partial\_sort :**  Trie les éléments du tableau donné, mais seulement jusqu’à un nombre d’élément voulu (représenté par un itérateur).



**Tips détecté les tris :**

* Tri à bulles : Les plus grandes valeurs se retrouvent à la fin.
* Tri par insertion : Début organisé, mais la fin ne l’est pas encore (on peut voir des petites valeurs).
* Tri par sélection : Début organisé et il n’y pas de plus petites valeurs autre part qu’au début.
* Tri par fusion : On retrouve des morceaux triés dans certains partie des éléments. Séparer en deux à chaque fois.
* Tri de Shell : Le tri n’a aucun sens.
* Tri rapide : Il faut essayer de trouver quel était le pivot et selon quoi le tableau est plus ou moins mélanger.

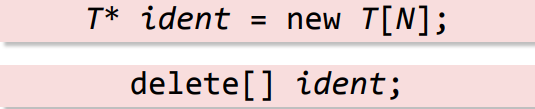
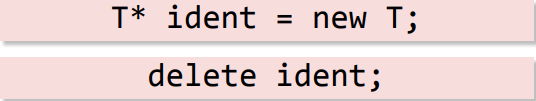


Les variables **globales** sont construites et allouées avant le main() et détruites et libérées après le main.

Les variables **statiques** sont construites lors du premier passage et détruites et libérées après le main().

Les variables **automatiques** sont construites lors de leur exécution et détruite lors de la sortie du bloc.

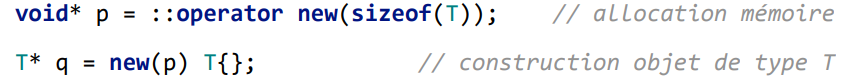
Variables **dynamiques** sont construites au passage de l’exécution sur « new » et elles sont détruites et libérées au passage sur « delete ».



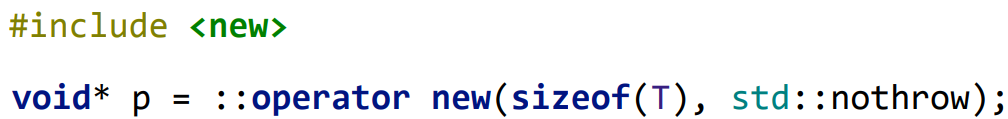
On détruit dans le sens inverse de la construction. Le premier élément créé est le dernier élément détruit.

Le mot clé « new » alloue et construit, mais il est possible de séparer les deux.

cette ligne devient alors les deux lignes ci-dessous

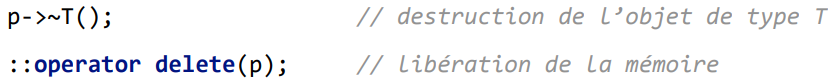


Peut lever 2 types d’exception, soit std::bad\_alloc ou une exception levée à la construction et relayée. On peut les éviter, pour la mauvaise allocation et ce qui permet de retourner un « nullptr ». Impossible pour les exceptions de T().



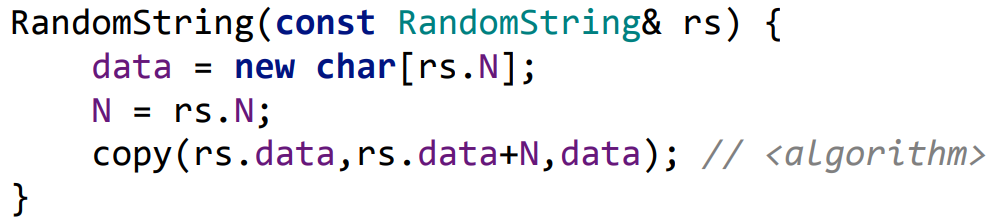
La même manipulation est possible lors de la destruction et libération avec « delete ». « Delete » ne lève pas d’exceptions.

 cette ligne devient alors les deux lignes ci-dessous.

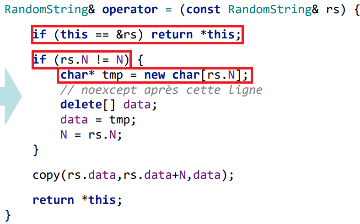


Fuites mémoires : Perdre l’adresse de l’objet à détruire, perdre le pointeur en sortant du scope ou une exception est levée. Pour les éviter, il faut encapsuler les allocations dynamiques et détruire dans les destructeurs.

Sans constructeurs de copie explicite, le langage copie l’adresse, donc lors de la destruction l’objet est effacé 2 fois. Il faut en faire un qui copie les données.



Il faut aussi fournir un opérateur d’affectation. Celui-ci doit copier et libérer les ressources précédentes. Par contre, il faut gérer certains cas limite comme l’auto-affectation, l’affectation des mêmes données (optimisation) ou encore la levée d’une exception.

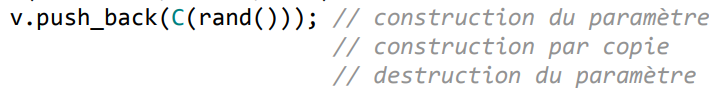


Il faut implémenter std::swap si on souhaite faire des tris (tri à bulles en a besoin, par contre insertion et fusion non). C’est un simple échange des adresses mémoire.

Il faut aussi implémenter tout le reste des opérateurs. Comme l’opérateur d’affichage et de comparaison pour la réalisation d’un tri à bulles.

std::push\_back appelle le constructeur std::move et std::emplace\_back non.

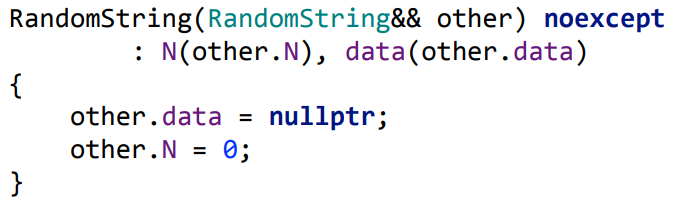
Si on std::push\_back et qu’il n’y a pas assez de place, la taille double et on recopie tous les éléments et on n’oublie pas de détruire derrière.





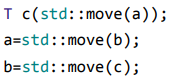
Le mieux est d’utiliser std::swap pour tous les types simples de notre classe.

Il faut implémenter std::move si on souhaite réaliser un tri par insertion générique. Le but étant de déplacer les ressources de a à b, mais en laissant a dans un état valide. std::move cast les paramètres en rvalue référence (T&&). Il va donc voler les ressources de la source et les rendre valide.



**Rappels :**

* **vector<T> v(N)** : Construit N objets de type T
* **push\_back** : Il realloue de la mémoire s’il n’y a pas assez de place en doublant la capacité. Il appelle le constructeur de copie pour mettre les objets dans le nouveau vecteur. Il fini par détruire les anciens éléments
* **push\_back**(T(…)) : Il créer un objet avec le constructeur normal, puis appelle le constructeur de déplacement et détruit le temp.
* **emplace\_back**(arguments de la classe) : Construit directement dans le vecteur. Pas de move, appelle constructeur basique. Juste un appel au constructeur normal (en place).
* **resize(n, val) :** 
  + n < taille : On détruit les éléments
  + n > taille : On crée un objet val. avec le constructeur et on fait des copies avec le constructeur de copie. Si pas de valeur précisée, alors on crée un objet avec le constructeur par défaut et on copie.
  + n > capacité : On realloue comme push\_back
* **swap**: Fait une construction avec le constructeur de déplacement et fait deux affectations avec l’opérateur d’affectation par déplacement.



* **v[i] = T(…)** : Construit l’objet normalement et fait une affectation par déplacement, puis détruit.
* Si on initialise un objet avec un std::move, ça appelle le constructeur de déplacement. Et si on fait une affectation avec, ça appelle l’affectation par déplacement. Un move permet d’éviter une destruction.
* **Si on construit un objet à l’aide d’un objet temporaire, le constructeur ne construit qu’un seul objet.**
* **A la fin d’un bloc on détruit tous les objets existants. Dans le cas d’un vecteur, on détruit tous les objets dans ce vecteur un à un.**

