

3.5 Tri optimal



Tri optimal



Peut-on faire sensiblement mieux que MergeSort et QuickSort ?

Proposition - Un algorithme de tri ...

sur n éléments

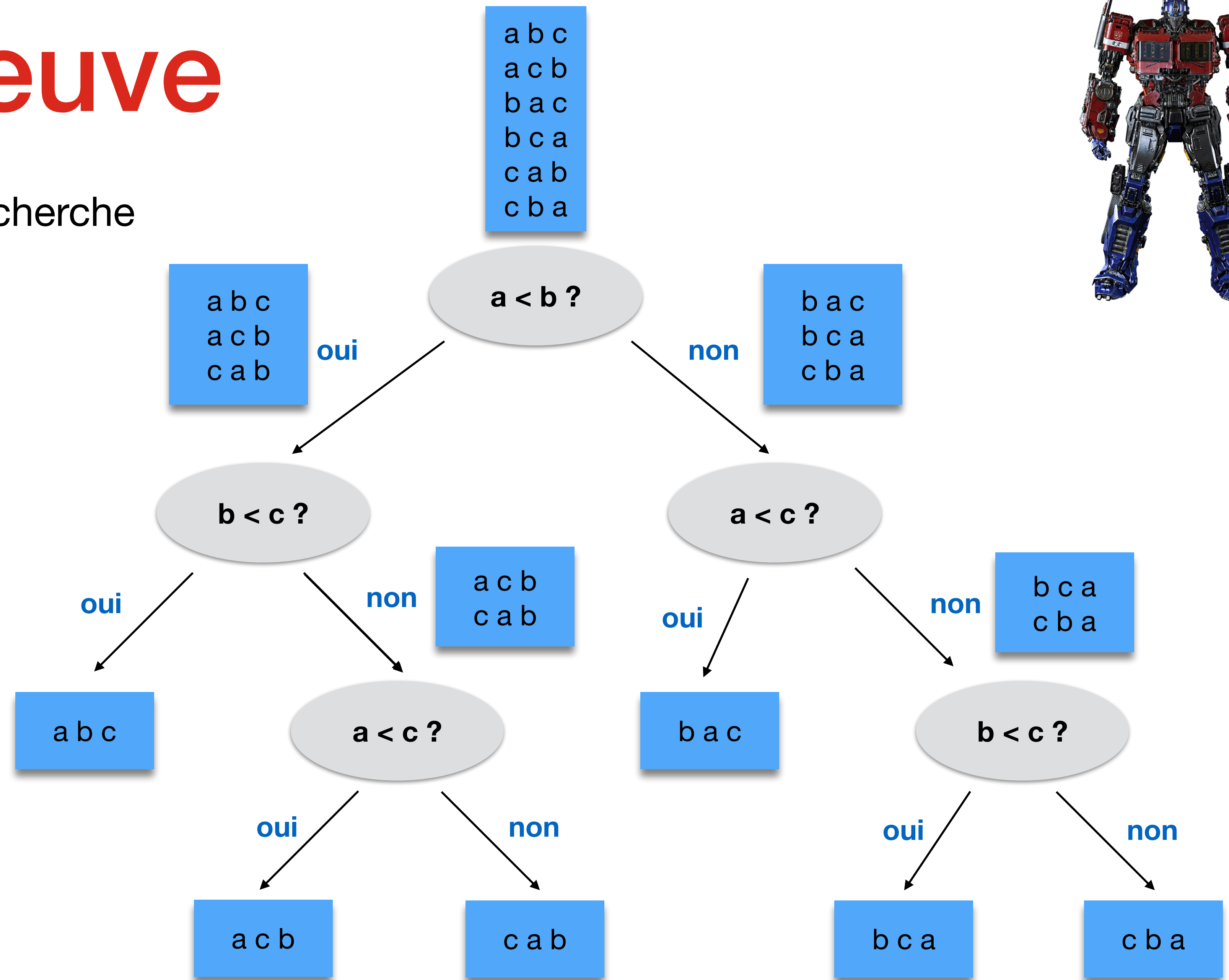
basé sur des comparaisons

a besoin de $O(n \log_2(n))$ comparaisons.

Preuve

On peut voir du tri comme une recherche de permutation : *celle ordonnée*.

- Il y a $n!$ ordres possibles pour n éléments
- Toute comparaison donne une information binaire. Dans le meilleur cas, cela réduit par 2 le nombre d'ordres possibles.
- On continue jusqu'à n'avoir plus qu'une permutation possible.
- Un arbre binaire de $n!$ feuilles a au moins une hauteur de $\log_2(n!) \approx n \cdot \log_2(n)$



Mieux qu'optimal ?

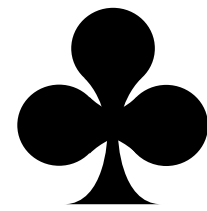
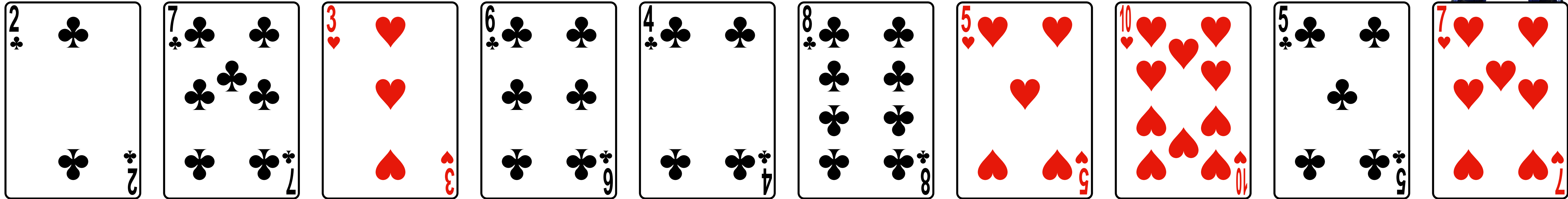


Mais on peut quand même parfois faire mieux que l'optimal...

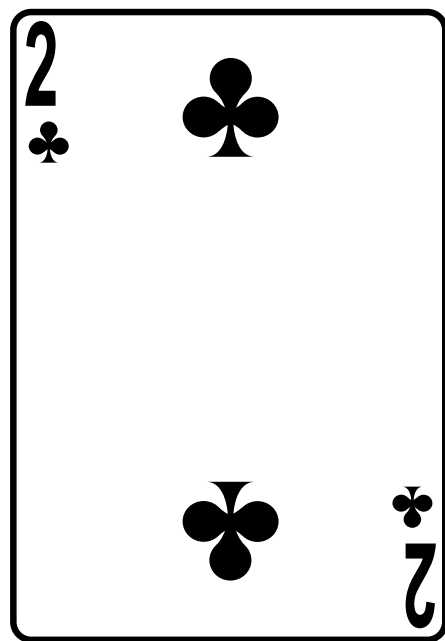
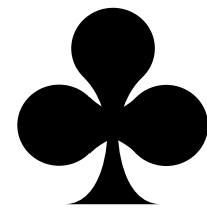
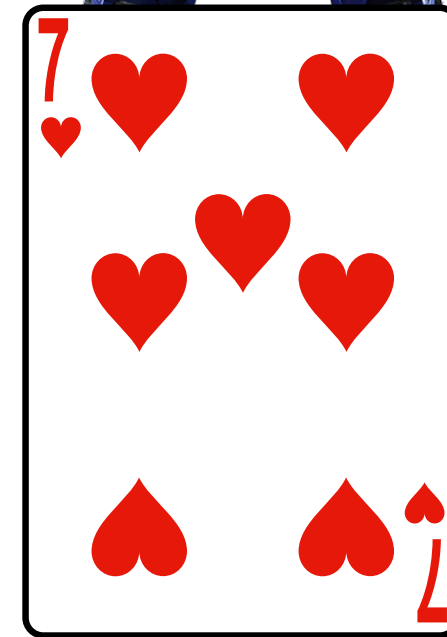
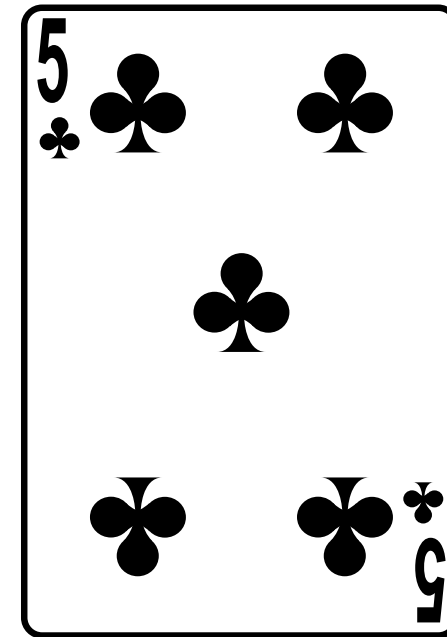
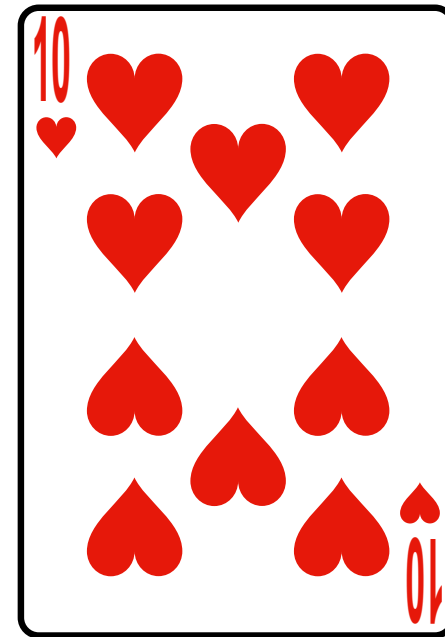
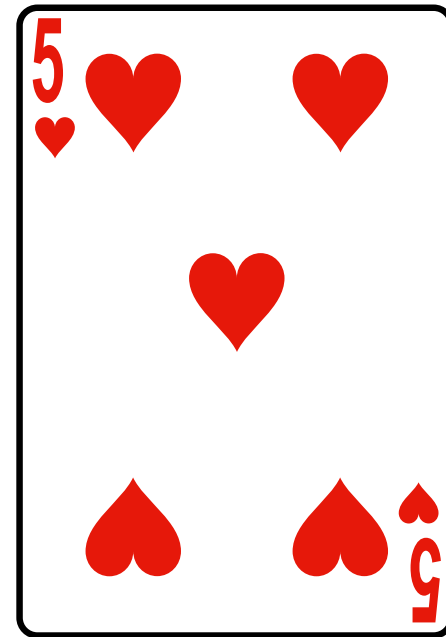
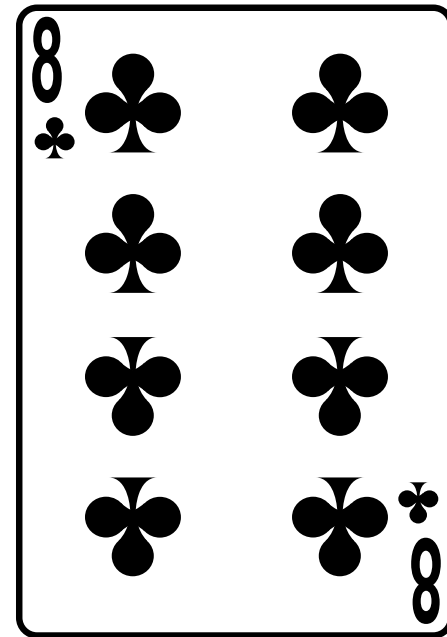
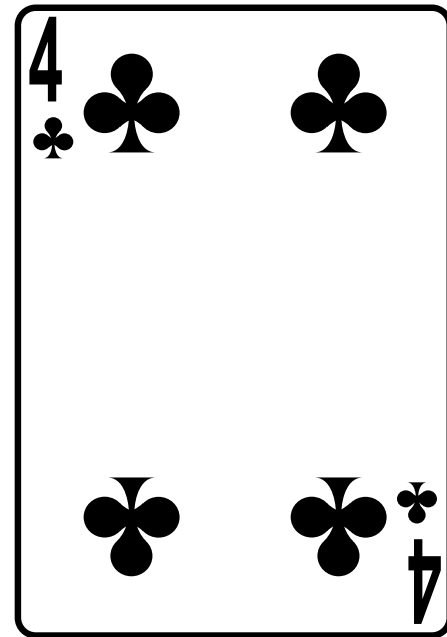
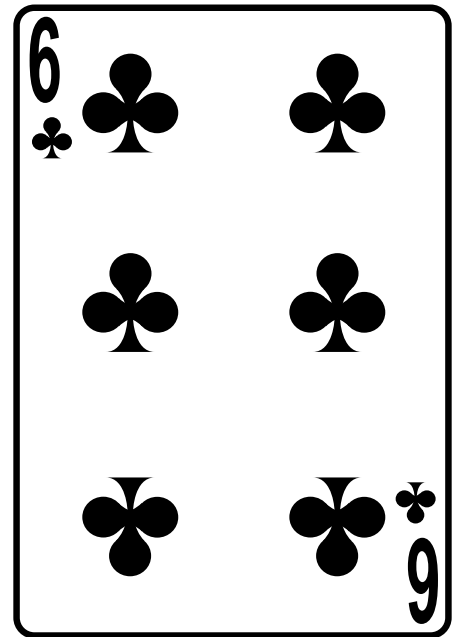
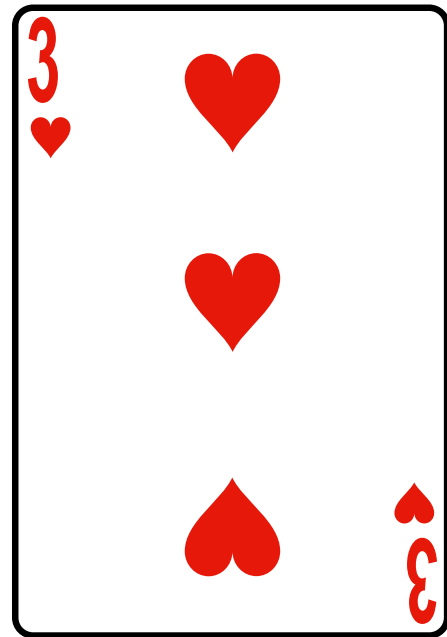
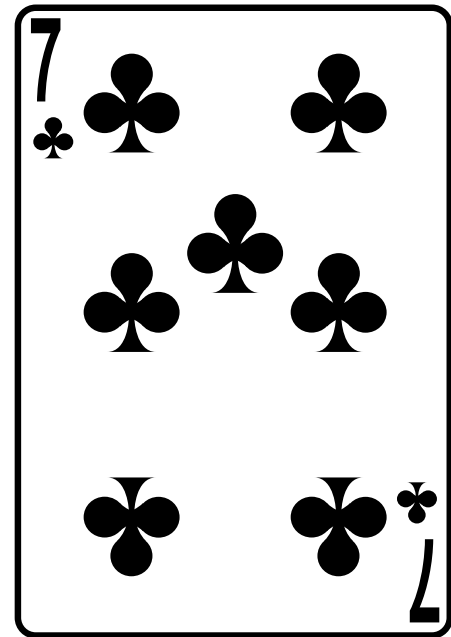
- Ne pas utiliser de comparaisons
- Utiliser des propriétés connues des données

Comment triezy vous un jeu de carte par couleur ?

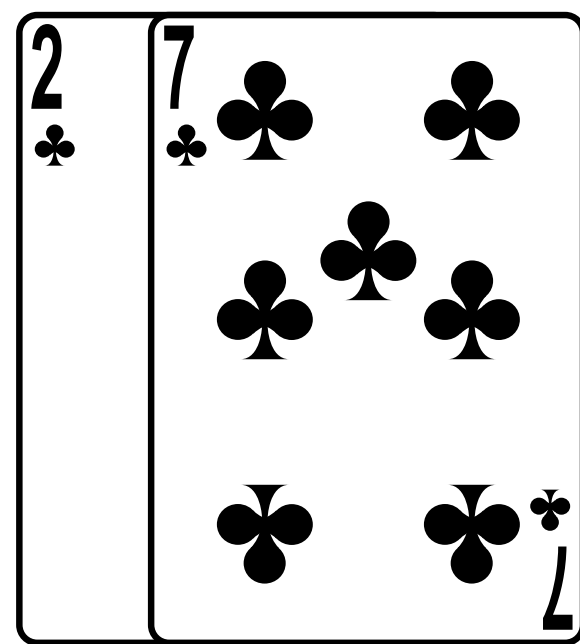
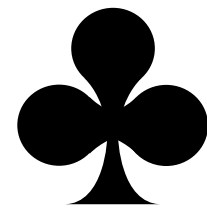
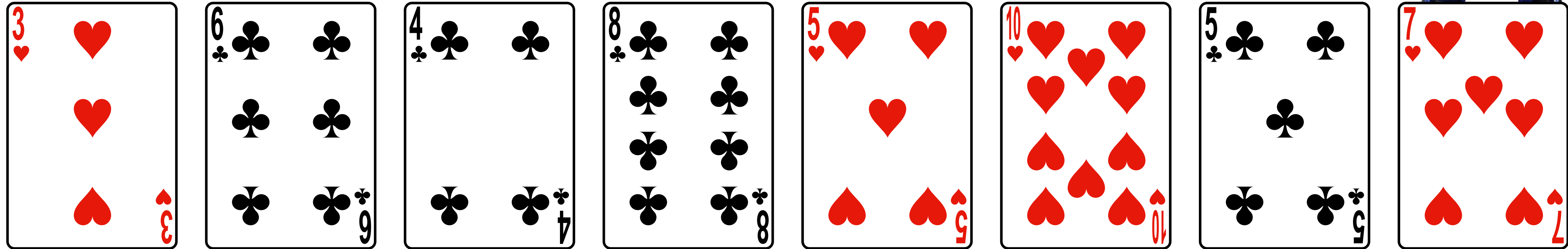
Trier sans comparer



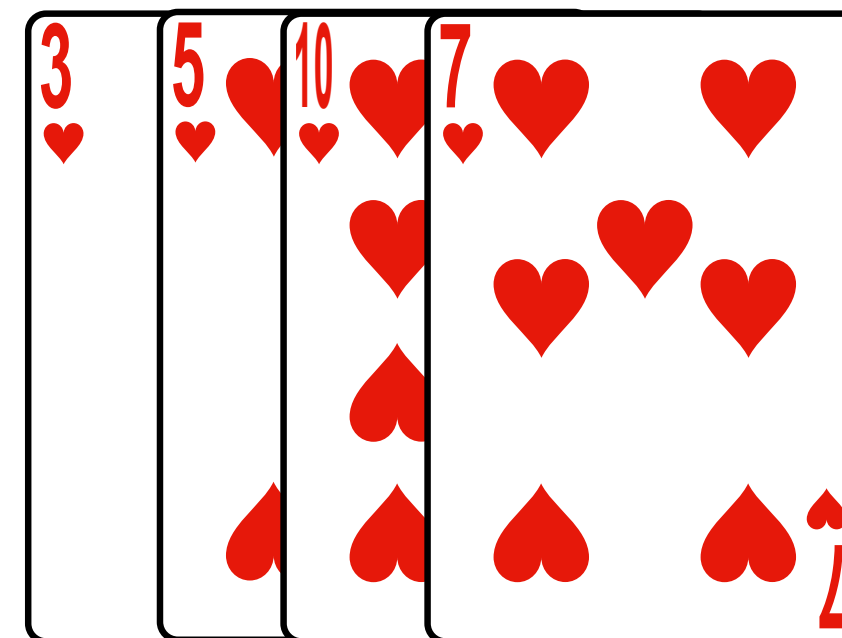
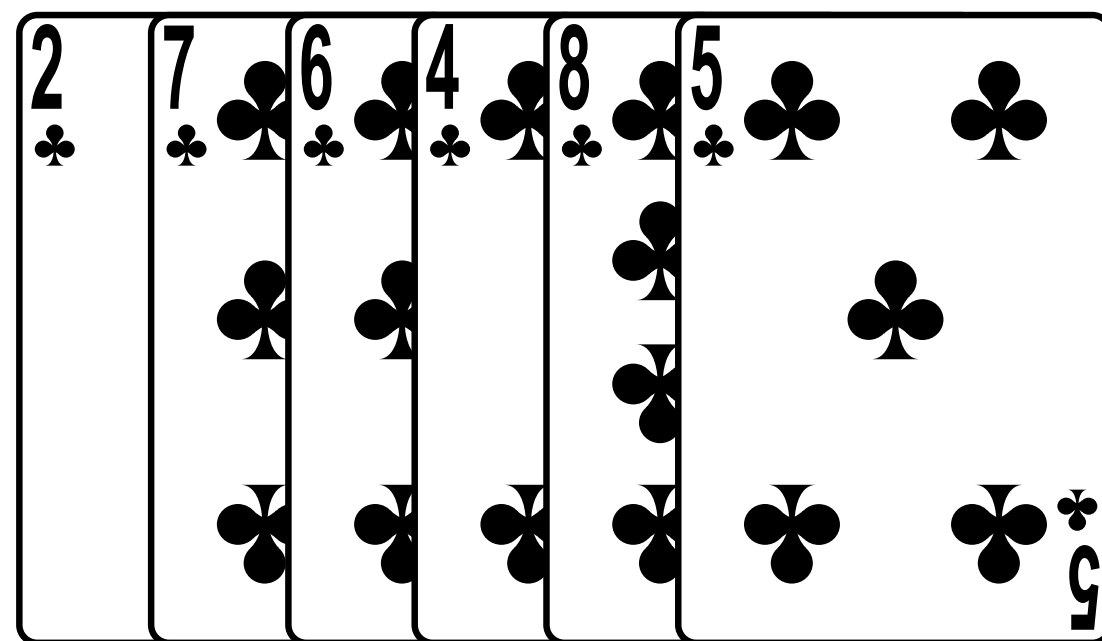
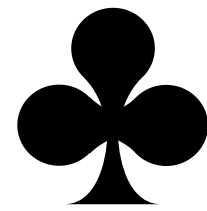
Trier sans comparer



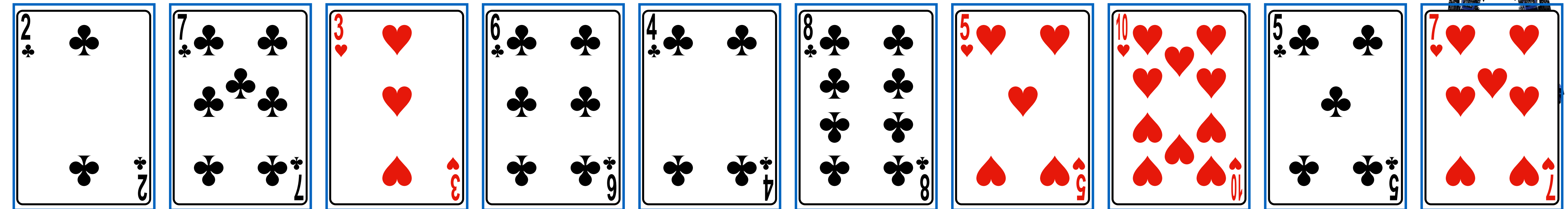
Trier sans comparer



Trier sans comparer

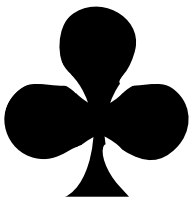


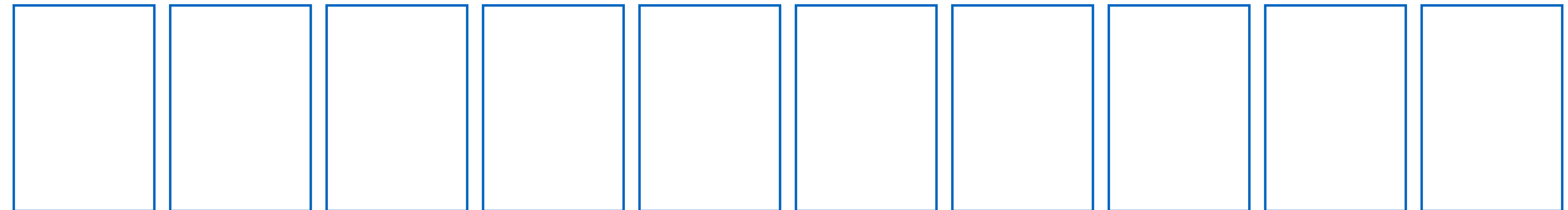
Tri comptage



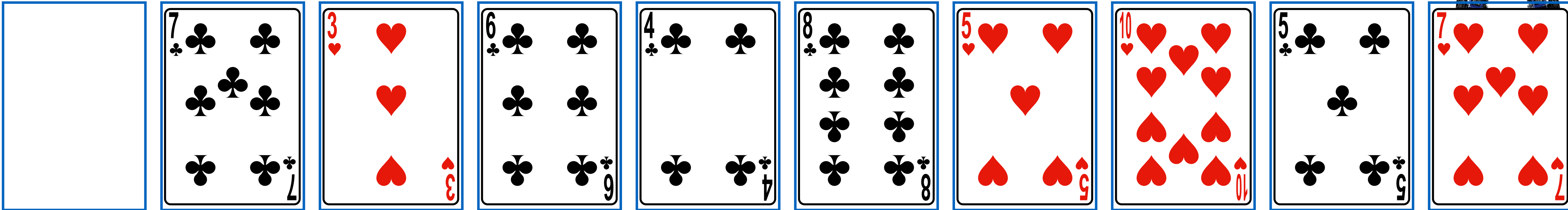
Compteurs:

*(Résultats d'un
comptage préliminaire)*

	6
	4

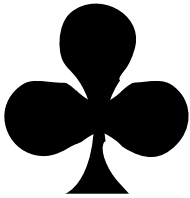


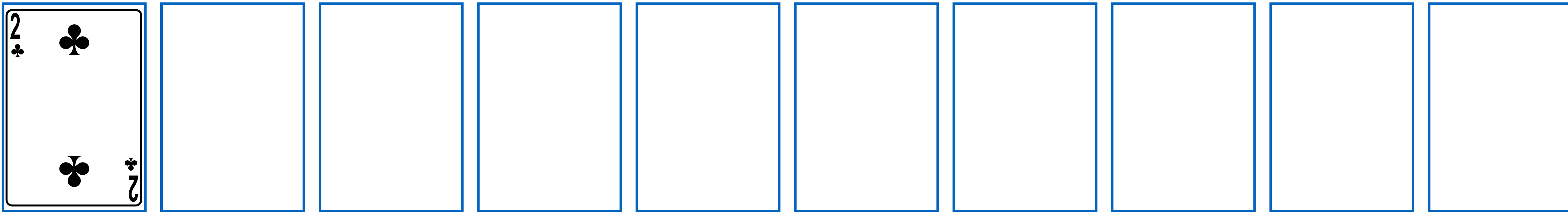
Tri comptage



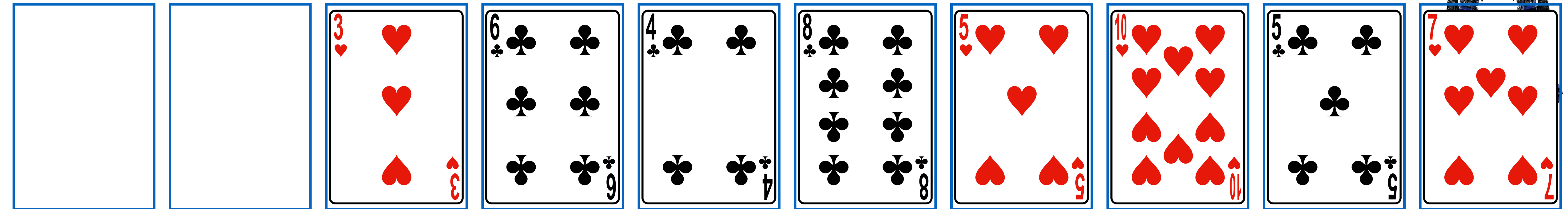
Compteurs:

*(Résultats d'un
comptage préliminaire)*

	6
	4

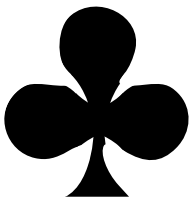


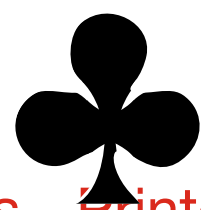
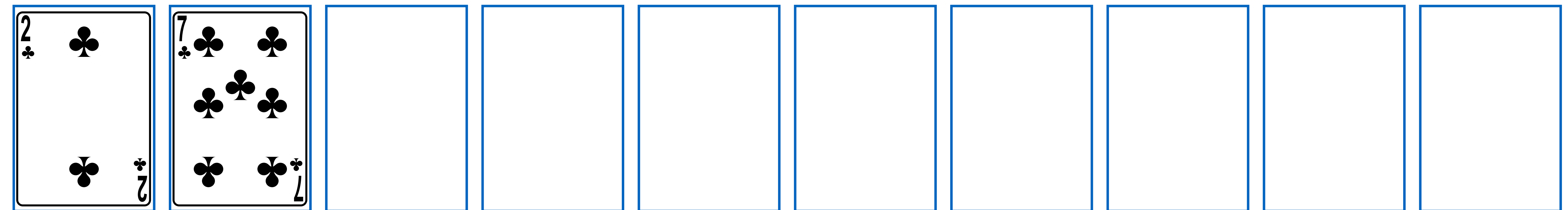
Tri comptage



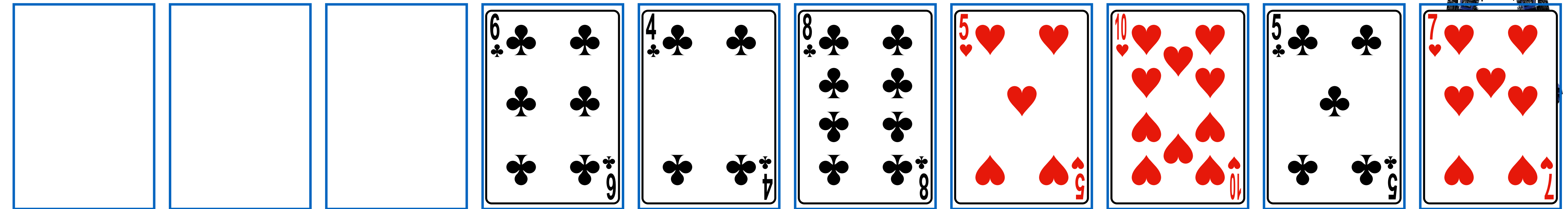
Compteurs:

*(Résultats d'un
comptage préliminaire)*

	6
	4

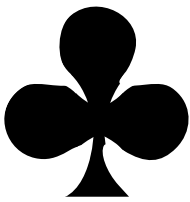


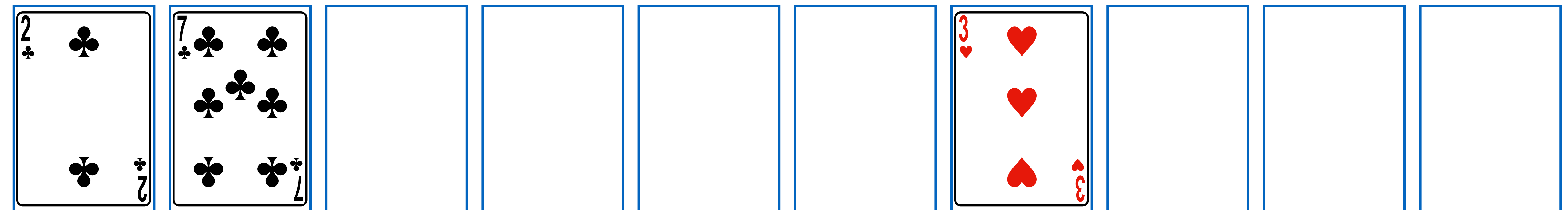
Tri comptage



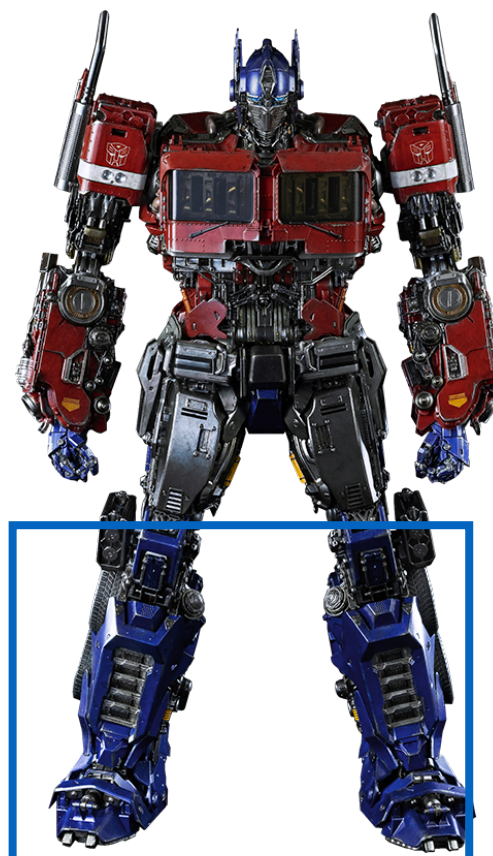
Compteurs:

*(Résultats d'un
comptage préliminaire)*

	6
	4



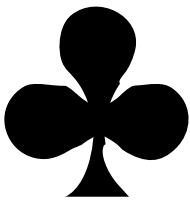
Tri comptage

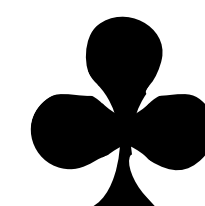
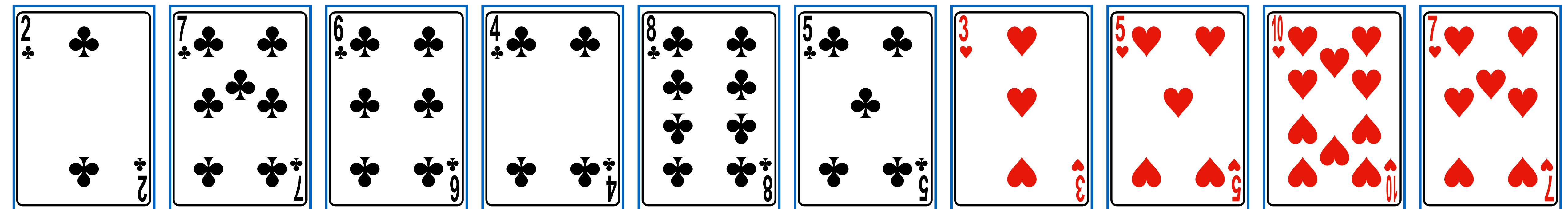


--	--	--	--	--	--	--	--	--	--

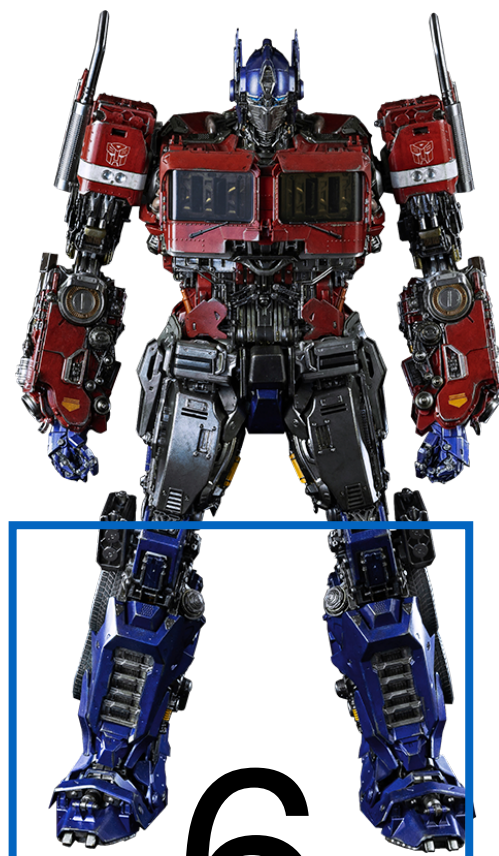
Compteurs:

*(Résultats d'un
comptage préliminaire)*

	6
	4



Tri comptage - généralisation



6

13

0

2

13

1

34

52

4

6

Compteurs:

(Résultats d'un
comptage préliminaire)

Pair

7

Impair

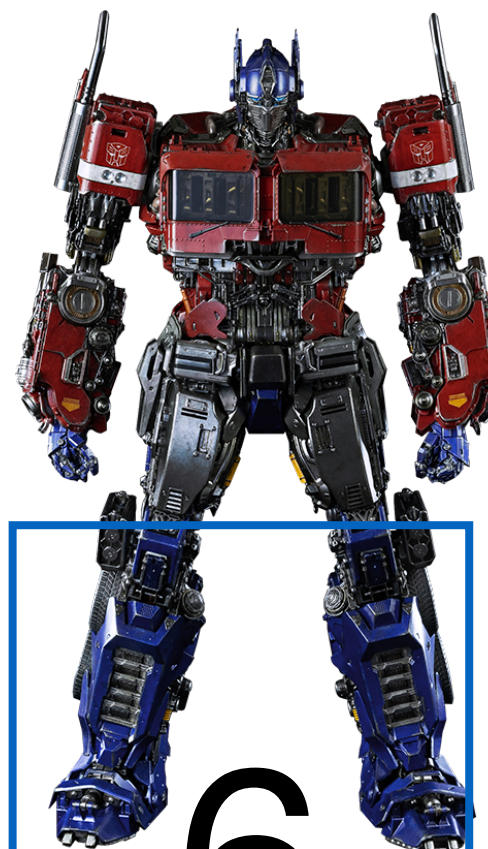
3

--	--	--	--	--	--	--	--	--	--

Pair

Impair

Tri comptage - généralisation



6

13

0

2

13

1

34

52

4

6

Compteurs:
(Résultats d'un
comptage préliminaire)

Termine par 1 1	Termine par 3 2	Termine par 5 0	Termine par 7 0	Termine par 9 0
Termine par 2 2	Termine par 4 2	Termine par 6 2	Termine par 8 0	Termine par 0 0

Termine par 1

Termine par 2

Termine par 3

Termine par 4

Termine par 5

Termine par 6

Termine par 7

Termine par 8

Termine par 9

Termine par 0

Tri comptage - généralisation



Définition d’une “classe”

(e.g. couleur de carte, parité, unité, etc)

Un compteur par classe

Un “index de placement” par classe

déplacé pour toujours être *l’index du prochain*

Tri comptage



Tri d'un tableau A de taille n ,
avec b classes distinctes,
données par la fonction $clé$.

- Stable
- Pas en place
 - tableau pour les compteurs
 - tableau pour la sortie
- Complexité $O(n+b)$ pour n éléments appartenant à b classes distinctes
 - 2 passages sur le tableau : $O(n)$
 - 1 passage sur les compteurs : $O(b)$

fonction TriComptage($A, n, b, clé$):

Comptage $\left[\begin{array}{l} C \leftarrow \text{tableau de } b \text{ compteurs à zéro} \\ \text{pour tout } e \text{ dans } A \\ \quad C[clé(e)] += 1 \end{array} \right.$

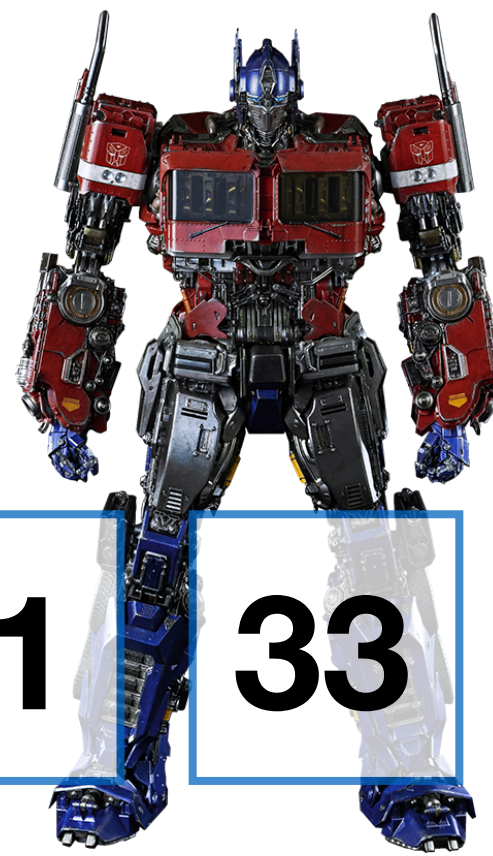
$\left[\begin{array}{l} idx \leftarrow 1 \\ \text{pour } i \text{ de } 1 \text{ à } b \\ \quad tmp \leftarrow C[i] \\ \quad C[i] \leftarrow idx \\ \quad idx += tmp \end{array} \right. \text{Calcul de l'index où} \\ \text{commencera chaque clé}$

$B = \text{tableau de même taille que } A$

$\left[\begin{array}{l} \text{pour tout } e \text{ dans } A \\ \quad B[C[clé(e)]] \leftarrow \text{déplacer } e \\ \quad C[clé(e)] += 1 \end{array} \right. \text{Tri par clé.}$

return B

Tri par base - radix sort

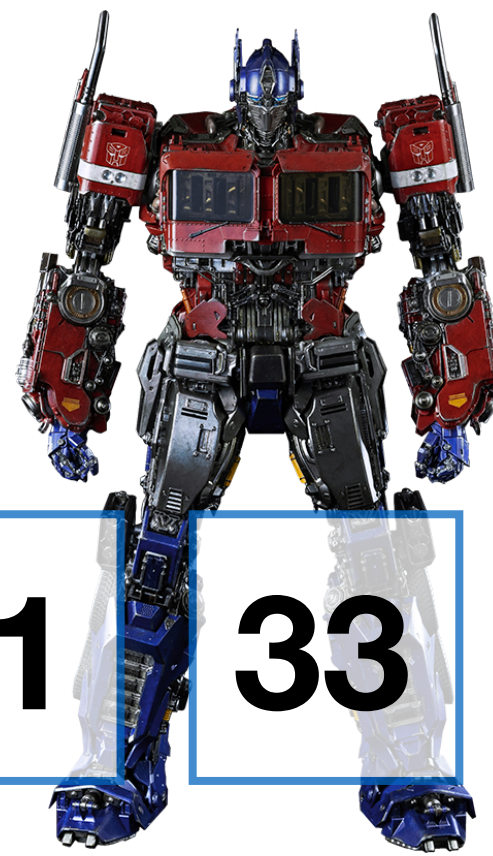


65	12	23	42	27	37	45	19	17	41	76	04	15	61	33
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

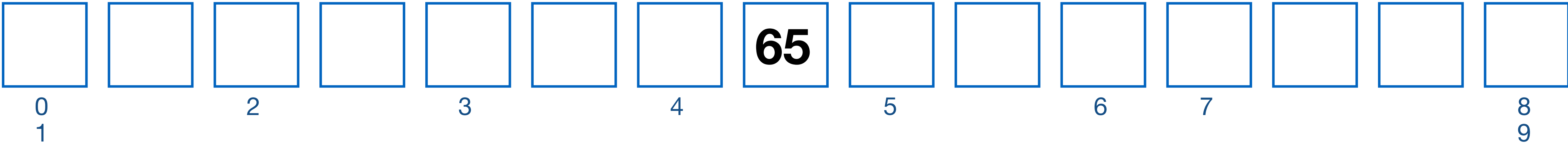
Tri comptage par unité

0		2		3		4	5			6	7			8
1														9

Tri par base - radix sort



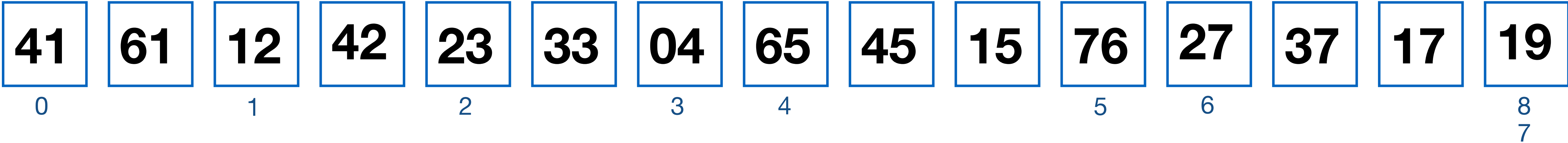
Tri comptage par unité



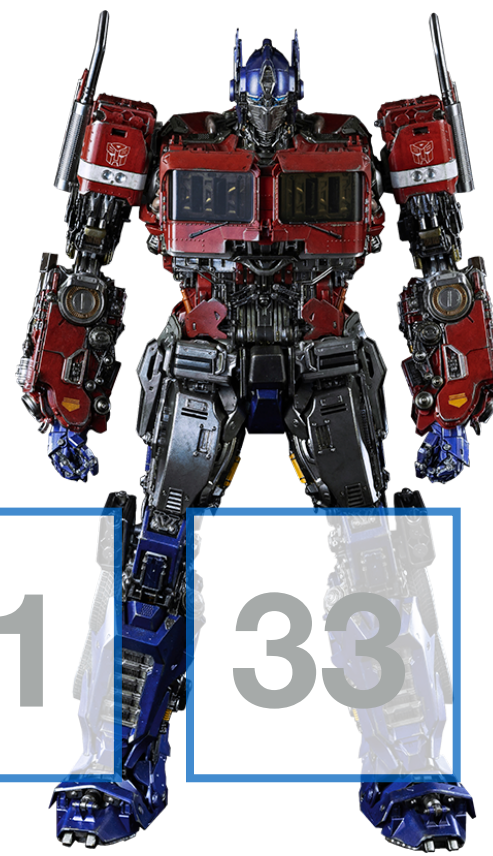
Tri par base - radix sort



Tri comptage par unité



Tri par base - radix sort



65 12 23 42 27 37 45 19 17 41 76 04 15 61 33

Tri comptage par unité

41 61 12 42 23 33 04 65 45 15 76 27 37 17 19

Tri comptage par dizaine

0-

1-

2-

3-

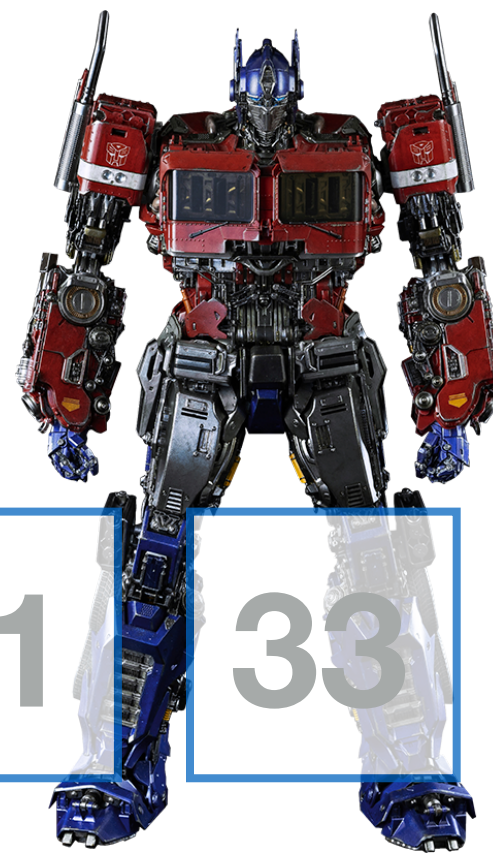
4-

5-

7-

6-

Tri par base - radix sort



65	12	23	42	27	37	45	19	17	41	76	04	15	61	33
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

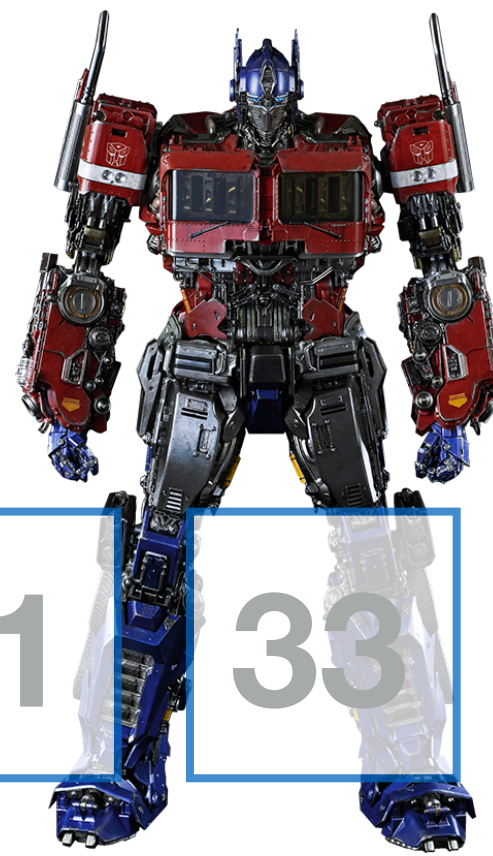
Tri comptage par unité

41	61	12	42	23	33	04	65	45	15	76	27	37	17	19
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Tri comptage par dizaine

									41					
0-	1-				2-		3-			4-		5- 6-		7-

Tri par base - radix sort



65 12 23 42 27 37 45 19 17 41 76 04 15 61 33

Tri comptage par unité

41 61 12 42 23 33 04 65 45 15 76 27 37 17 19

Tri comptage par dizaine

04 12 15 17 19 23 27 33 37 41 42 45 61 65 76

0-

1-

2-

3-

5-

4-

6-

Tri par base - radix sort



- Stable
- Pas en place
 - Un tableau pour les compteurs
 - Un tableau pour la sortie
- Complexité $O(d(n+b))$ pour n éléments pouvant prendre b^d valeurs :
 - d exécutions du tri comptage,
 - $O(n+b)$ par tri comptage.

Tri d'un tableau T contenant des nombres à d chiffres au plus.

fonction triParBase(T , d):

Pour i allant de d à 1

 Trier le tableau T

 avec un tri stable

 selon le i -ème chiffre

 (*par exemple, triComptage*)

Autres algorithmes de tri



Exchange sorts	Bubble sort - Cocktail sort - Odd–even sort - Comb sort - Gnome sort - Quicksort - Stooge sort - Bogosort
Selection sorts	Selection sort - Heapsort - Smoothsort - Cartesian tree sort - Tournament sort - Cycle 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 - Strand sort
Distribution sorts	American flag sort - Bead sort - Bucket sort - Burtsort - Counting sort - Pigeonhole sort - Proxmap sort - Radix sort - Flashsort
Concurrent sorts	Bitonic sorter - Batcher odd–even mergesort - Pairwise sorting network
Hybrid sorts	Block sort - Timsort - Introsort - Spreadsort - JSort
Other	Topological sorting - Pancake sorting - Spaghetti sort



<http://www.xkcd.com/1185/>

StackSort connects to StackOverflow, searches for 'Sort a list' and downloads and runs code snippets until the list is sorted

<https://gkoberger.github.io/stacksort/>

```
DEFINE HALFHEARTEDMERGESORT(LIST):
  IF LENGTH(LIST) < 2:
    RETURN LIST
  PIVOT = INT(LENGTH(LIST) / 2)
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
  // UMMMMM
  RETURN[A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
  // AN OPTIMIZED BOGOSORT
  // RUNS IN O(N LOG N)
  FOR N FROM 1 TO LOG(LENGTH(LIST)):
    SHUFFLE(LIST):
    IF ISSORTED(LIST):
      RETURN LIST
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
  OK SO YOU CHOOSE A PIVOT
  THEN DIVIDE THE LIST IN HALF
  FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO, WAIT, IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE EQUAL ONES GO INTO, UH
    THE SECOND LIST FROM BEFORE
  HANG ON, LET ME NAME THE LISTS
  THIS IS LIST A
  THE NEW ONE IS LIST B
  PUT THE BIG ONES INTO LIST B
  NOW TAKE THE SECOND LIST
  CALL IT LIST, UH, A2
  WHICH ONE WAS THE PIVOT IN?
  SCRATCH ALL THAT
  IT JUST RECURSIVELY CALLS ITSELF
  UNTIL BOTH LISTS ARE EMPTY
  RIGHT?
  NOT EMPTY, BUT YOU KNOW WHAT I MEAN
  AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
  IF ISSORTED(LIST):
    RETURN LIST
  FOR N FROM 1 TO 10000:
    PIVOT = RANDOM(0, LENGTH(LIST))
    LIST = LIST[PIVOT:] + LIST[:PIVOT]
    IF ISSORTED(LIST):
      RETURN LIST
  IF ISSORTED(LIST):
    RETURN LIST
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING
    RETURN LIST
  IF ISSORTED(LIST): // COME ON COME ON
    RETURN LIST
  // OH JEEZ
  // I'M GONNA BE IN SO MUCH TROUBLE
  LIST = [ ]
  SYSTEM("SHUTDOWN -H +5")
  SYSTEM("RM -RF ./")
  SYSTEM("RM -RF ~/*")
  SYSTEM("RM -RF /")
  SYSTEM("RD /S /Q C:\*") // PORTABILITY
  RETURN [1, 2, 3, 4, 5]
```

3.6. Tris en C et C++





THE
C
PROGRAMMING
LANGUAGE



En C, la librairie "stdlib.h" fournit la fonction `qsort`, dont le prototype est

```
void qsort (void* base,  
           size_t num,  
           size_t size,  
           int (*comp)(const void*, const void*));
```

- `void *base` : adresse du premier élément du tableau
- `size_t num` : nombre d'éléments à trier
- `size_t size` : taille d'un élément du tableau
- `int (*comp) (void const *a, void const *b)` : adresse de la fonction de comparaison, fournie par l'utilisateur.

qsort - Comparaison



Ecrivons une fonction comparant deux entiers, de prototype

```
int plus_petit (const void * a, const void * b);
```

Elle doit

- caster les pointeurs `void*` en pointeurs `int*`
- comparer les deux valeurs entières
- retourner un entier
 - `<0` si `*a` est plus petit que `*b`
 - `>0` si `*b` est plus petit que `*a`
 - `0` s'ils sont égaux selon le critère choisi

qsort - Comparaison



```
int plus_petit (const void * a, const void * b)
{
    return ( *(const int*)a - *(const int*)b );
}
```

Elle doit

- caster les pointeurs `void*` en pointeurs `int*`
- comparer les deux valeurs entières
- retourner un entier
 - `<0` si `*a` est plus petit que `*b`
 - `>0` si `*b` est plus petit que `*a`
 - `0` s'ils sont égaux selon le critère choisi

qsort - Utilisation



Soit le tableau d'entiers

```
int values[] = { 40, 10, 100, 90, 20, 25 };  
int N = sizeof(values)/sizeof(int);
```

Pour le trier entièrement, il suffit d'écrire

```
#include "stdlib.h"  
qsort (values, N, sizeof(int), plus_petit);
```

Ce qui donne le tableau trié suivant

```
{ 10, 20, 25, 40, 90, 100 }
```

qsort - Propriétés



Le langage C ne fournit pas de garantie sur la complexité de `qsort`, mais en pratique

- `qsort` est le diminutif de Quick Sort, l'algorithme de tri rapide
- Le tri a donc une complexité moyenne $\Theta(n \log n)$
- Il peut avoir une complexité quadratique $\Theta(n^2)$ dans le pire des cas
- Il n'est pas stable



`std::sort`

`std::stable_sort`

`std::nth_element`

`std::partial_sort`

`std::partial_sort_copy`

`std::partition`

`std::partition_copy`

`std::partition_point`

`std::is_sorted`

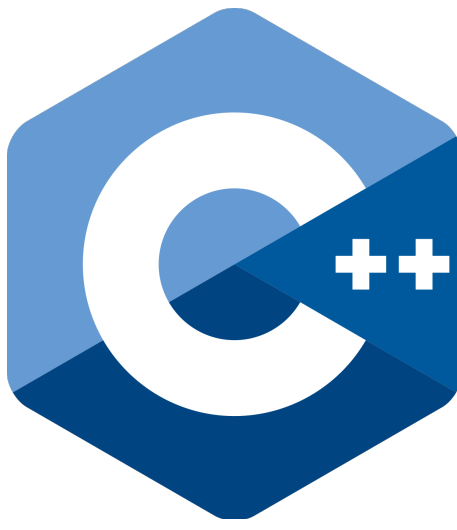
`std::is_sorted_until`

`std::is_partitioned`

`std::merge`

`std::inplace_merge`

`std::stable_partition`

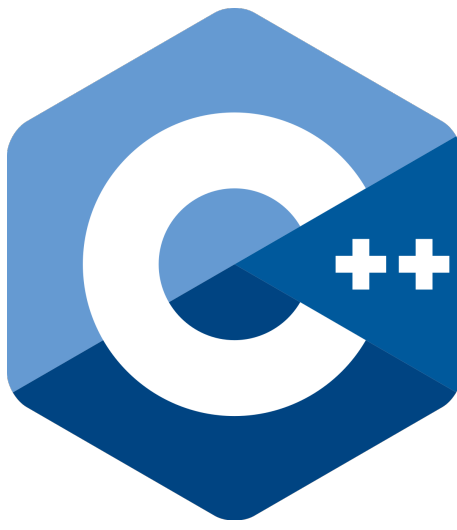


En C++, la librairie `<algorithm>` fournit la fonction `std::sort`, dont les prototypes sont

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first,
           RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);
```

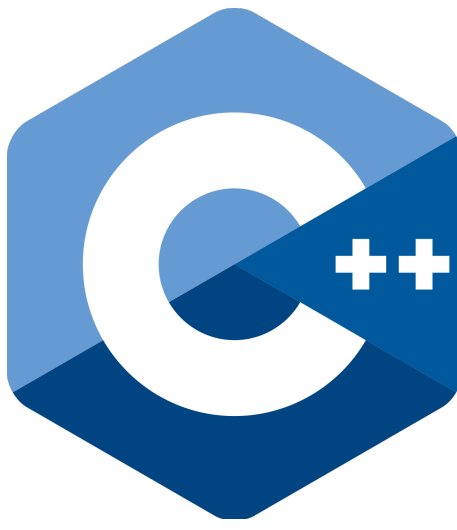
std::sort - interface



```
template <class RandomAccessIterator, class Compare>  
void sort (RandomAccessIterator first,  
           RandomAccessIterator last,  
           Compare comp);
```

- `first` : itérateur vers le premier élément à trier
- `last` : itérateur vers l'élément qui suit le dernier à trier. Ensemble, ils définissent une séquence `[first, last[`.
- `comp` : fonction de comparaison. Prend deux éléments en paramètres et retourne un `bool` qui vaut vrai si le premier est plus petit que le second. Si elle n'est pas spécifiée, le tri utilise l'opérateur `<` du type trié

std::sort - utilisation



Soit le vecteur v dont on veut trier les 4 premiers éléments

```
#include <vector>
std::vector<int> v{ 32,71,12,45,26,80,53,33 };
```

```
#include <algorithm>
std::sort( v.begin(), v.begin() + 4 );
```

{ 12, 32, 45, 71, 26, 80, 53, 33 }

Pour trier tout le vecteur, on écrit

```
std::sort( v.begin(), v.end() );
```

{ 12, 26, 32, 33, 45, 53, 71, 80 }

std::sort - fonction en paramètre



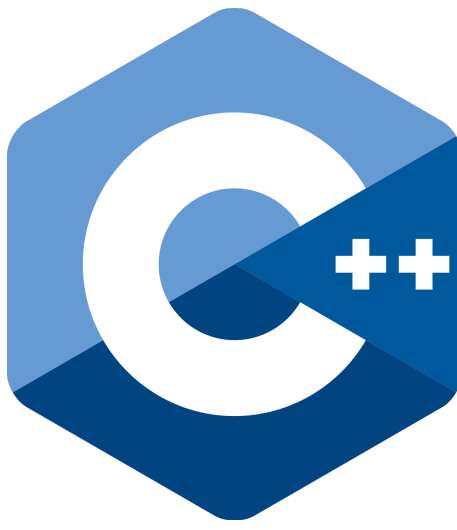
La version avec comparaison générique peut prendre en paramètre une fonction,

```
bool plus_grand (int i, int j) { return (i>j); }
```

```
std::sort(v.begin(), v.end(), plus_grand);
```

```
{ 80, 71, 53, 45, 33, 32, 26, 12 }
```

std::sort - foncteur en paramètre



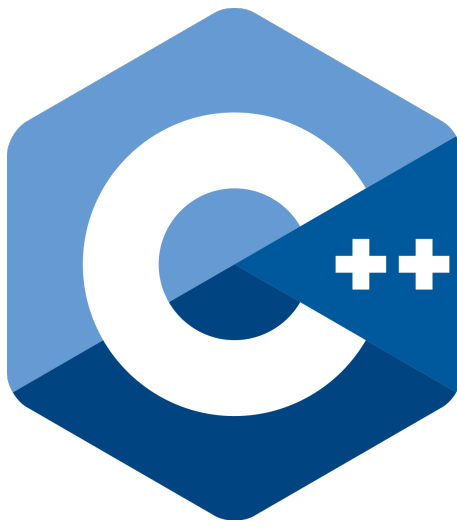
un objet fonction, ou foncteur, c'est à dire une structure ou une classe qui définit l'opérateur `operator()`,

```
struct plus_petit_modulo {  
    int base;  
    bool operator() (int i, int j) { return (i%base < j%base );}  
};
```

```
auto compare_dernier_chiffre = plus_petit_modulo{10};  
std::sort( v.begin(), v.end(), compare_dernier_chiffre );  
v
```

```
{ 80, 71, 32, 12, 53, 33, 45, 26 }
```

std::sort - foncteur en paramètre



un objet fonction, ou foncteur, c'est à dire une structure ou une classe qui définit l'opérateur `operator()`,

```
struct plus_petit_modulo {  
    int base;  
    bool operator() (int i, int j) { return (i%base < j%base );}  
};
```

```
auto compare_parity = plus_petit_modulo{2};  
std::sort( v.begin(), v.end(), compare_parity );  
v
```

```
{ 80, 32, 12, 26, 71, 53, 33, 45 }
```

std::sort - lambda en paramètre



ou une [expression lambda](#) (C++11), i.e. un objet fonction anonyme capable de capturer des variables dans la portée.

```
int B = 2;  
std::sort( v.begin(), v.end(),  
           [&B](int i, int j) { return i%B < j%B; } );  
v
```

```
{ 80, 32, 12, 26, 71, 53, 33, 45 }
```

`std::sort` - Propriétés



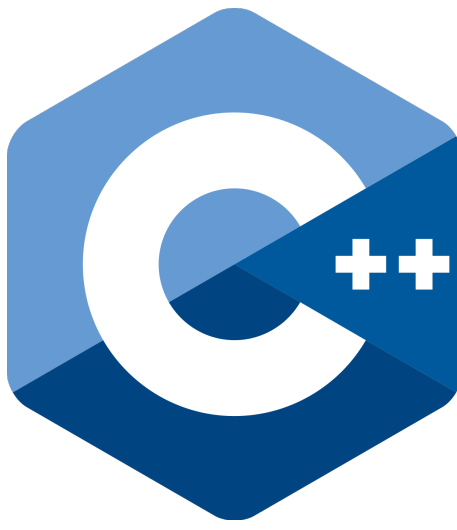
La librairie standard garantit une **complexité moyenne** linéarithmique $\Theta(n \log n)$.

En pratique, `std::sort` est toujours une variation de l'algorithme de **tri rapide**, éventuellement avec un tri par insertion pour les partitions les plus petites.

Donc,

- il n'est pas stable
- c'est le plus rapide des tris proposés par la STL
- on ne peut exclure une complexité quadratique dans le pire des cas

Le tri rapide étant un tri par échange, il est important que la fonction `swap(T, T)` soit efficace pour le type `T` à trier.



La librairie `<algorithm>` fournit également la fonction `std::stable_sort`, dont les prototypes sont

```
template <class RandomAccessIterator>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last );
```

```
template <class RandomAccessIterator, class Compare>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last,
                  Compare comp );
```

std::stable_sort - Utilisation

Trions par exemple en ne tenant compte que des parties entières avec la fonction

```
bool partie_entiere_plus_petite (double i, double j)
{ return (int(i) < int(j)); }
```

```
std::vector<double> v2 {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.
58};
std::stable_sort(v2.begin(), v2.end(), partie_entiere_plus_petite);
v2
```

```
{ 1.41, 1.73, 1.32, 1.62, 2.72, 2.58, 3.14, 4.67 }
```

Inégalité stricte !



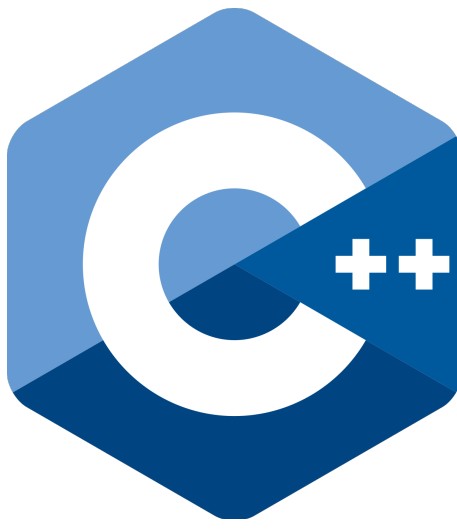
Mais attention, il est indispensable que la fonction de tri mette en oeuvre une inégalité stricte. Sinon la stabilité n'est pas garantie.

```
bool partie_entiere_plus_petite_ou_egale (double i, double j)
{ return (int(i) <= int(j)); }
```

```
std::vector<double> v3 {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.
58};
std::stable_sort(v3.begin(), v3.end(), partie_entiere_plus_petite_ou
_egale);
v3
```

```
{ 1.62, 1.32, 1.73, 1.41, 2.58, 2.72, 3.14, 4.67 }
```

`std::stable_sort` - Propriétés



La librairie standard garantit

- une complexité temporelle linéarithmique $\Theta(n \log n)$, si il y a assez de mémoire.
- une complexité temporelle $\Theta(n \log^2 n)$ si le tri fusion doit être réalisé en place.

En pratique, `std::stable_sort` est toujours une variation de l'algorithme de **tri par fusion**.

Donc,

- il est stable
- il est moins rapide `std::sort`
- les complexités sont également garanties dans le pire des cas

Le tri par fusion effectant de nombreux déplacements, il est important que l'affectation `T = std::move(T)` soit efficace pour le type `T` à trier.



La librairie `<algorithm>` fournit aussi une fonction de sélection rapide sous le nom de `nth_element`, dont le prototype est

```
template <class RandomAccessIterator>
void nth_element (RandomAccessIterator first,
                  RandomAccessIterator nth,
                  RandomAccessIterator last);
```

où les éléments à traiter sont dans l'intervalle `[first, last[` et la valeur de `n` est donnée par `n = nth - first`.

std::nth_element - effets de la
fonction

```
template <class RandomAccessIterator>
void nth_element (RandomAccessIterator first,
                  RandomAccessIterator nth,
                  RandomAccessIterator last);
```

La fonction met le n^{ieme} élément à sa place,

mais a aussi pour effet de partitionner `[first, last[` autour de sa valeur.

- `[first, nth[` ne contient que des éléments $\leq *nth$
- `[nth, last[` ne contient que des éléments $\geq *nth$

std::nth_element - Comparaison générique



Il existe évidemment aussi une version avec fonction de comparaison générique.

```
template <class RandomAccessIterator, class Compare>  
void nth_element (RandomAccessIterator first,  
                  RandomAccessIterator nth,  
                  RandomAccessIterator last,  
                  Compare comp);
```

std::nth_element - Utilisation



Trouvons le 4^{ième} élément (d'indice 3) du tableau v4 et partitionnons le autour de cette valeur

```
std::vector<int> v4{ 3,5,2,6,8,1,7,4};  
std::nth_element(v4.begin(), v4.begin()+3, v4.end());
```

```
{ 3, 1, 2, 4, 6, 5, 7, 8 }
```

std::nth_element - Propriétés



La STL garantit une complexité **moyenne** linéaire $\Theta(n)$ pour `n = last - first`.

Cela correspond à la complexité de l'algorithme de sélection rapide.



std::partial_sort

Il est également possible de ne trier qu'une partie d'un tableau avec la fonction `partial_sort` de la librairie `<algorithm>`. Le prototype en est

```
template <class RandomAccessIterator>
void partial_sort (RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last );
```

- `[first, last[` est l'intervalle des valeurs à trier.
- `middle` doit être dans cet intervalle.



```
template <class RandomAccessIterator>
void partial_sort (RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last );
```

En sortie, l'intervalle `[first, middle[`

- contient les `middle-first` plus petites valeurs
- est trié

l'intervalle `[middle, last[`

- ne contient que des valeurs plus grandes
- dans un ordre quelconque.

std::partial_sort - Utilisation



Trions les 4 plus petits éléments du tableau v5.

```
std::vector<int> v5{ 4,3,6,2,7,1,8,5};  
std::partial_sort(v5.begin(), v5.begin()+4, v5.end());
```

```
{ 1, 2, 3, 4, 7, 6, 8, 5 }
```



La fonction de tri peut également être générique

```
template <class RandomAccessIterator, class Compare>  
void partial_sort (RandomAccessIterator first,  
                  RandomAccessIterator middle,  
                  RandomAccessIterator last,  
                  Compare comp );
```



Trions les 3 éléments du tableau v6 de plus grande valeur absolue.

```
bool plus_grande_valeur_absolue(double a, double b) {  
    return abs(a) > abs(b);  
}
```

```
std::vector<double> v6{ 4, -3, -6, 2, -7, -1, 8, 5};  
std::partial_sort(v6.begin(), v6.begin()+3, v6.end(), plus_grande_valeur_absolue);
```

```
{ 8, -7, -6, 2, -3, -1, 4, 5 }
```

`std::partial_sort` - Propriétés



La librairie standard garantit une complexité $\Theta(n \log m)$ avec

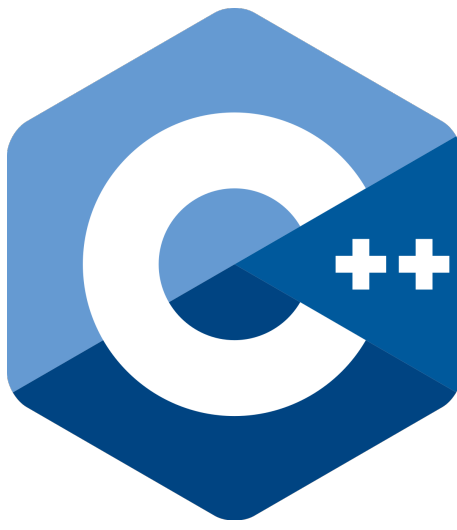
- `n = last - first`
- `m = middle - first.`

Il existe plusieurs possibilités pour mettre en oeuvre ce tri partiel.

- La complexité garantie suggère
 - l'utilisation d'un tas de m éléments
 - dans lequel on insère les autres $n - m$ éléments
 - en supprimant le maximum entre chaque insertion.
- Une approche alternative consiste à
 - utiliser la sélection rapide pour trouver le m^{ieme} élément
 - ce qui partitionne aussi le tableau de sorte que les m plus petits sont à gauche
 - utiliser le tri rapide sur ces m plus petits éléments

Cette approche alternative a une complexité $\Theta(n + m \log m)$ en moyenne, ce qui est meilleur en moyenne, mais éventuellement moins bon dans le pire des cas.

Et bien plus encore ...



`std::sort`

`std::partition`

`std::is_partitioned`

`std::partition_copy`

`std::merge`

`std::stable_sort`

`std::partition_point`

`std::inplace_merge`

`std::nth_element`

`std::is_sorted`

`std::stable_partition`

`std::is_sorted_until`

`std::partial_sort`

`std::partial_sort_copy`