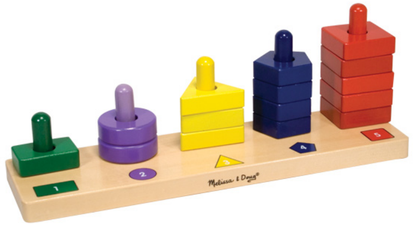
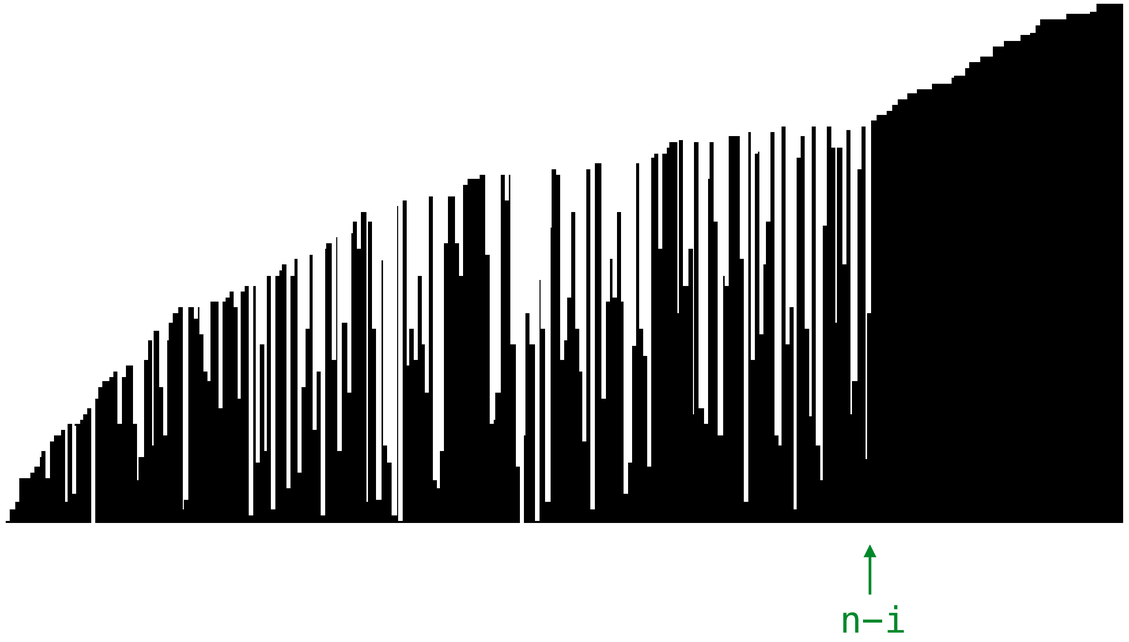
Chapitre 3 : Algorithmes de tri

3.1. Tris simples (Rappels)

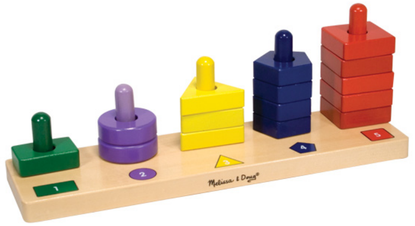
Tri à bulles



```
fonction BubbleSort(A,n)
        (tableau A de n éléments)

  pour i de 1 à n-1 boucler
    pour j de 1 à n-i boucler
      si A(j+1) < A(j), alors
        permuter A(j) et A(j+1)
      fin si
    fin pour j
  fin pour i
```

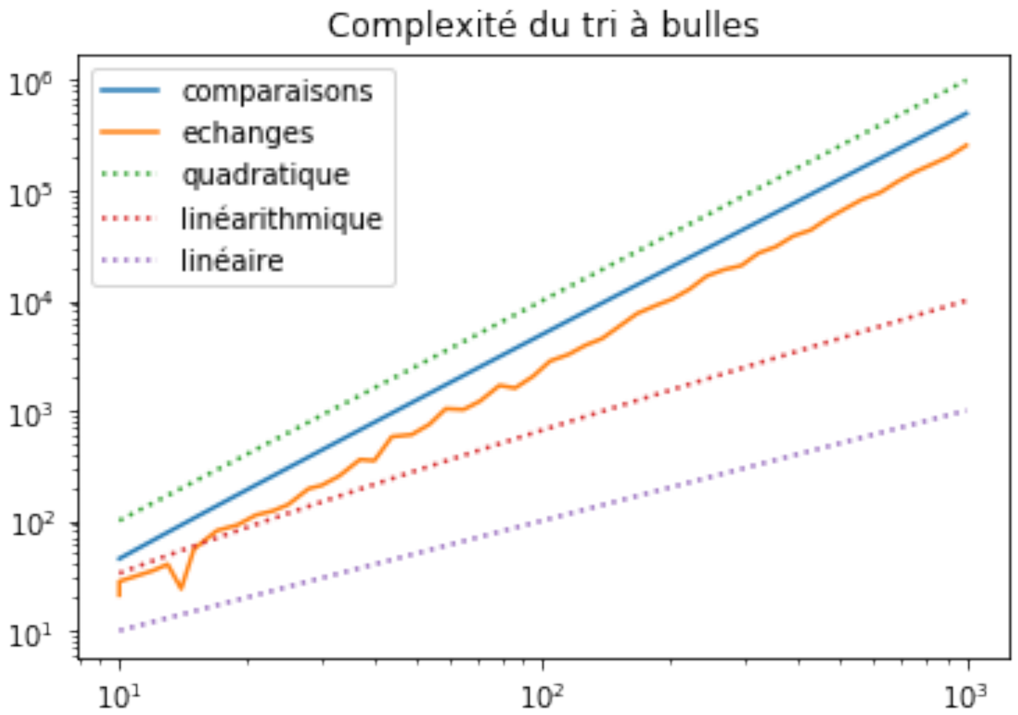


Tri à bulles



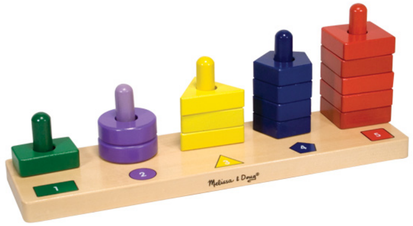
```
pour i de 1 à n-1 boucler
  pour j de 1 à n-i boucler
```

- $O(n^2)$ comparaisons
- De 0 à $O(n^2)$ échanges selon l'ordre des éléments à trier



N	Comp.	Ech.
10	45	21
19	171	91
37	666	358
71	2485	1212
138	9453	4502
268	35778	19041
517	133386	67990
1000	499500	256151

Stabilité



- Que se passe-t-il quand 2 éléments du tableau sont équivalents du point du vue de critère de tri ?

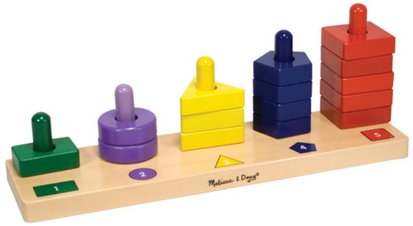
Ordre alphabétique des prénoms

```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Catherine (4 ans)
Bonnet Christine (22 ans)
Aubert Denis (33 ans)
```

Ordre alphabétique des noms

```
Aubert Alexandre (43 ans)
Aubert Beatrice (22 ans)
Aubert Carole (27 ans)
Aubert Denis (33 ans)
Bonnet Anne (27 ans)
Bonnet Benoit (21 ans)
Bonnet Christine (22 ans)
Caron Alain (27 ans)
Caron Brigitte (22 ans)
Caron Catherine (4 ans)
```

Eléments



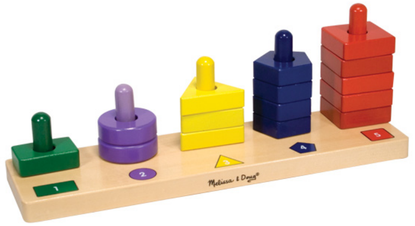
- Personnes dont on connaît nom, prénom, âge, …

```
struct Personne { string nom; string prenom; unsigned age; };

ostream& operator<<(ostream& o, Personne const& p) {
    return o << p.prenom << " " << p.nom << " (" << p.age << ")";
}
```

- Triables selon différents critères

```
bool comparer_noms (const Personne& p, const Personne& q) {
    return p.nom < q.nom;
}

bool comparer_prenoms (const Personne& p, const Personne& q) {
    return p.prenom < q.prenom;
}
```

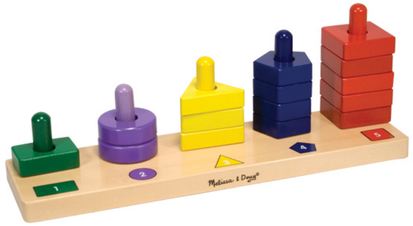
Tri à bulles générique



- Fonction de comparaison reçue en paramètre
- L'appel à cette fonction remplace `operator<`

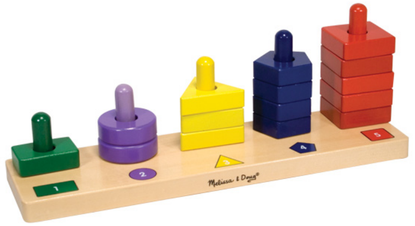
```
template<class T, class Compare>
void bubbleSort(vector<T>& v, Compare compare) {

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

Code appelant



```
int main() {
    vector<Personne> gens =
            { {"Aubert","Beatrice", 22},
              {"Caron","Alain", 27},
              {"Bonnet","Christine", 22},
              {"Bonnet","Anne", 27},
              {"Aubert","Alexandre", 43},
              {"Caron","Catherine", 4},
              {"Bonnet","Benoit", 21},
              {"Aubert","Denis", 33},
              {"Aubert","Carole", 27},
              {"Caron","Brigitte", 22}};

    bubbleSort(gens, comparer_prenoms );
    cout << gens << endl;

    bubbleSort(gens, comparer_noms );
    cout << gens << endl;
}
```

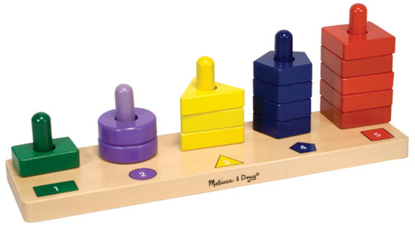
Code appelant



```
int main() {
    vector<Personne> gens =
            { {"Aubert","Beatrice", 22},
              {"Caron","Alain", 27},
              {"Bonnet","Christine", 22},
              {"Bonnet","Anne", 27},
              {"Aubert","Alexandre", 43},
              {"Caron","Catherine", 4},
              {"Bonnet","Benoit", 21},
              {"Aubert","Denis", 33},
              {"Aubert","Carole", 27},
              {"Caron","Brigitte", 22}};

    bubbleSort(gens, comparer_prenoms );
    cout << gens << endl;

    bubbleSort(gens, comparer_noms );
    cout << gens << endl;
}
```

```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Catherine (4 ans)
Bonnet Christine (22 ans)
Aubert Denis (33 ans)
```

Code appelant

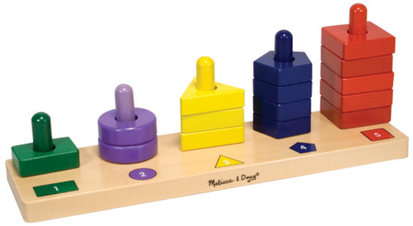


```
int main() {
    vector<Personne> gens =
            { {"Aubert","Beatrice", 22},
              {"Caron","Alain", 27},
              {"Bonnet","Christine", 22},
              {"Bonnet","Anne", 27},
              {"Aubert","Alexandre", 43},
              {"Caron","Catherine", 4},
              {"Bonnet","Benoit", 21},
              {"Aubert","Denis", 33},
              {"Aubert","Carole", 27},
              {"Caron","Brigitte", 22}};

    bubbleSort(gens, comparer_prenoms );
    cout << gens << endl;

    bubbleSort(gens, comparer_noms );
    cout << gens << endl;
}
```

```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Cath
Bonnet Chr
Aubert Den
```

```
Aubert Alexandre (43 ans)
Aubert Beatrice (22 ans)
Aubert Carole (27 ans)
Aubert Denis (33 ans)
Bonnet Anne (27 ans)
Bonnet Benoit (21 ans)
Bonnet Christine (22 ans)
Caron Alain (27 ans)
Caron Brigitte (22 ans)
Caron Catherine (4 ans)
```

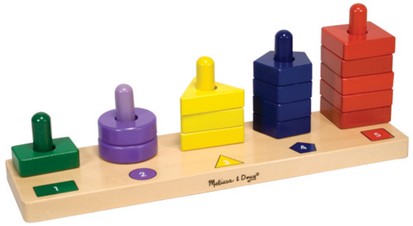
Attention ! < ou <= ?



```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom < q.nom;
}
```

```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom <= q.nom;
}
```

Attention ! < ou <= ?

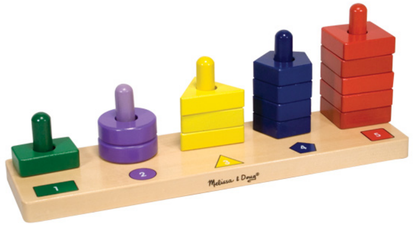


```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom < q.nom;
}
```

```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Catherine (4 ans)
Bonnet Christine (22 ans)
Aubert Denis (33 ans)
```

```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom <= q.nom;
}
```

Attention ! < ou <= ?



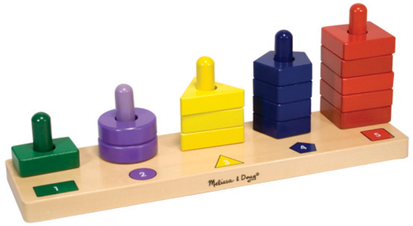
```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom < q.nom;
}
```

```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Catherine (4 ans)
Bonnet Christine (22 ans)
Aubert Denis (33 ans)
```

```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom <= q.nom;
}
```

```
Aubert Denis (33 ans)
Aubert Beatrice (22 ans)
Aubert Alexandre (43 ans)
Aubert Carole (27 ans)
Bonnet Christine (22 ans)
Bonnet Benoit (21 ans)
Bonnet Anne (27 ans)
Caron Catherine (4 ans)
Caron Brigitte (22 ans)
Caron Alain (27 ans)
```

Attention ! < ou <= ?



```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom < q.nom;
}
```

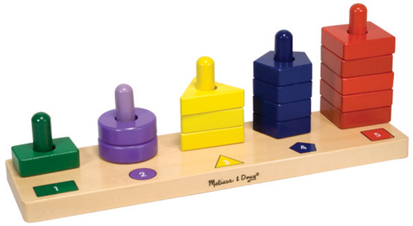
```
Caron Alain (27 ans)
Aubert Alexandre (43 ans)
Bonnet Anne (27 ans)
Aubert Beatrice (22 ans)
Bonnet Benoit (21 ans)
Caron Brigitte (22 ans)
Aubert Carole (27 ans)
Caron Catherine (4 ans)
Bonnet Christine (22 ans)
Aubert Denis (33 ans)
```

```
bool comparer_noms (const Personne& p,
                    const Personne& q) {
    return p.nom <= q.nom;
}
```

```
Aubert Denis (33 ans)
Aubert Beatrice (22 ans)
Aubert Alexandre (43 ans)
Aubert Carole (27 ans)
Bonnet Christine (22 ans)
Bonnet Benoit (21 ans)
Bonnet Anne (27 ans)
Caron Catherine (4 ans)
Caron Brigitte (22 ans)
Caron Alain (27 ans)
```

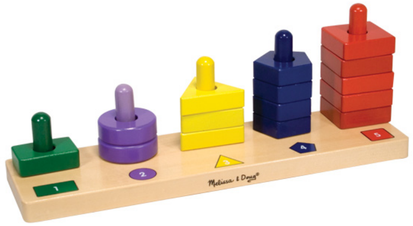
- Le tri à bulle n’est stable que s’il n’échange pas les éléments égaux

Tri par sélection

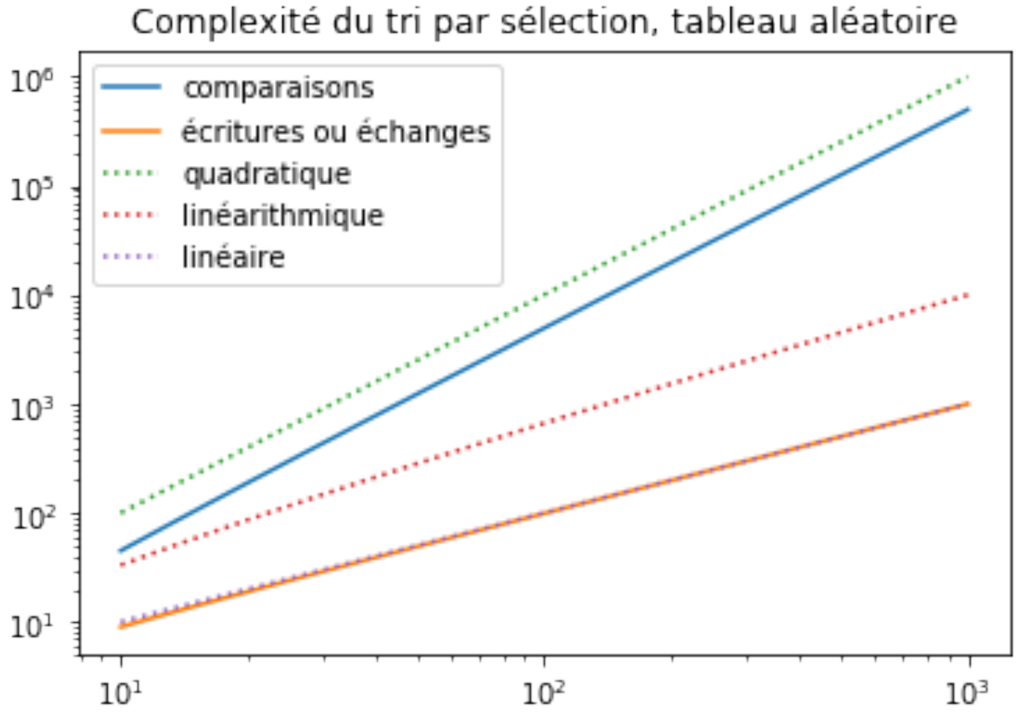


```
fonction SelectionSort(A,n)

  pour i de 1 à n-1 boucler

    imin ← i
    pour j de i+1 à n boucler
      si A(j) < A(imin), alors
        imin ← j
      fin si
    fin pour j

    permuter A(i) et A(imin)

  fin pour i
```

Tri par sélection

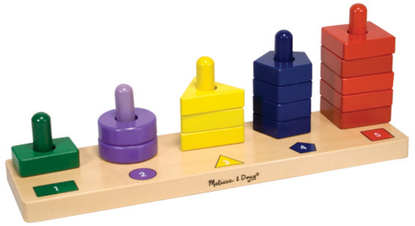


- $\mathrm{O}(\mathrm{n}^2)$ comparaisons, $\mathrm{O}(\mathrm{n})$ échanges, quelle que soit l'ordre d'entrée

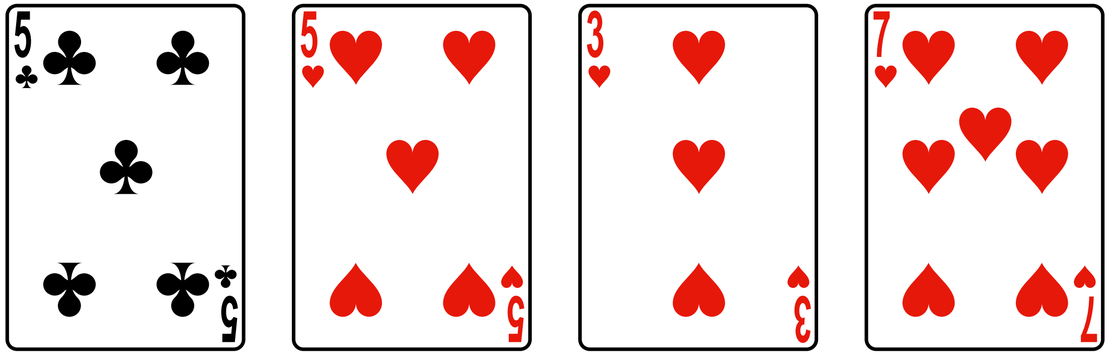


N	Comp.	Ecr.
10	45	9
19	171	18
37	666	36
71	2485	70
138	9453	137
268	35778	267
517	133386	516
1000	499500	999

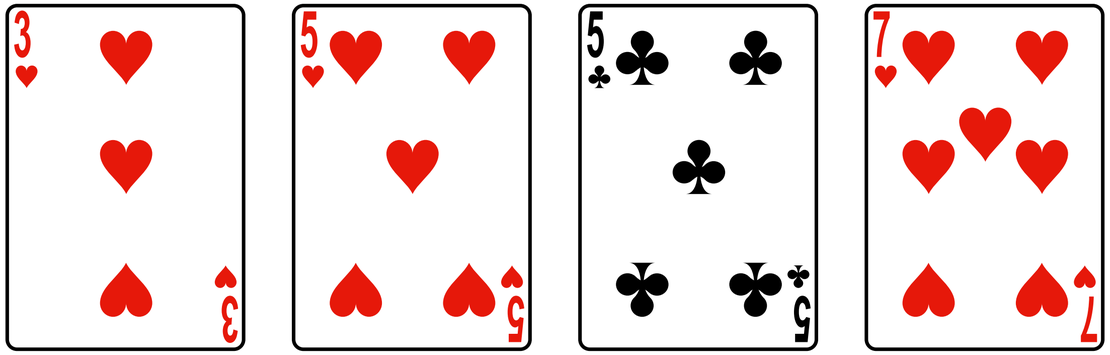
Tri par sélection non stable



- Cartes triées par couleur que l'on veut maintenant trier par valeur

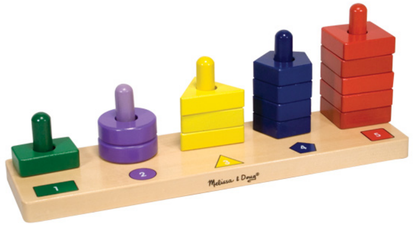


- 1ère étape, on trouve le minimum 3♥ et on échange avec 5♣

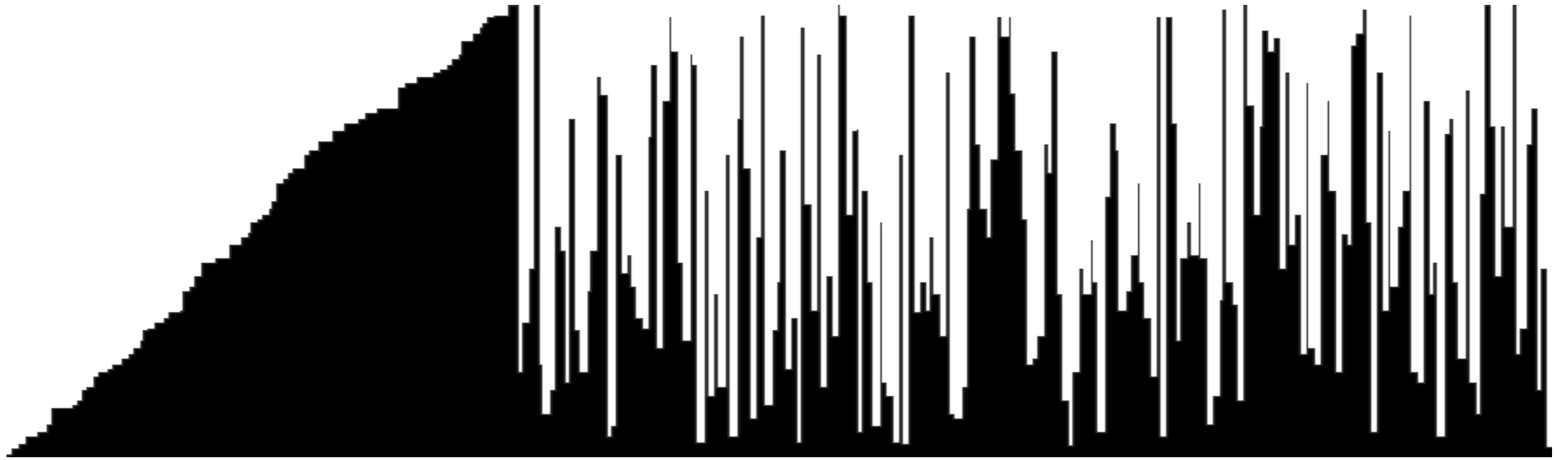


- Résultat final. Les « 5 » ne sont plus triés par couleur

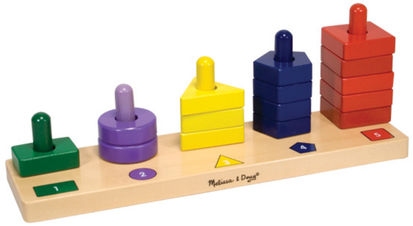
Tri par insertion



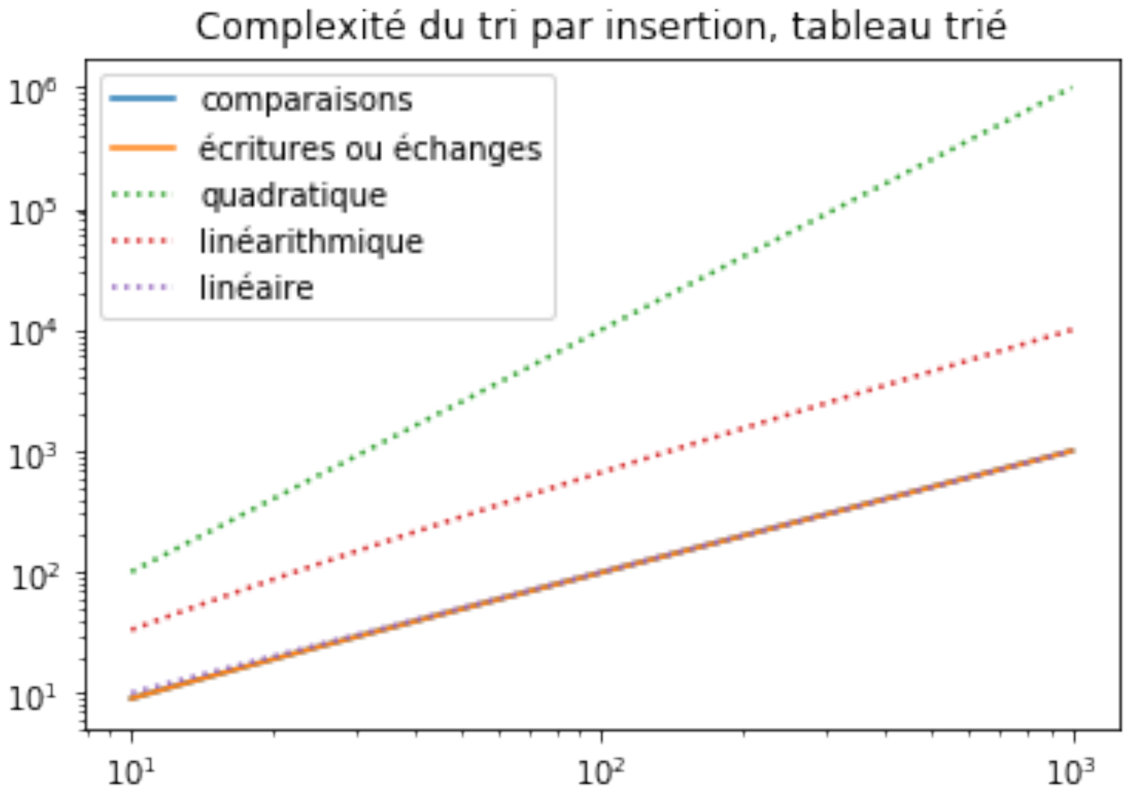
```
fonction InsertionSort(A,n)

  pour i de 2 à n boucler
    tmp ← A(i)
    j ← i
    tant que j−1 ≥ 1 et A(j−1) > tmp boucler
      A(j) ← A(j−1)
      décrémenter j de 1
    fin tant que
    A(j) ← tmp
  fin pour i
```



Insertion- Meilleur cas

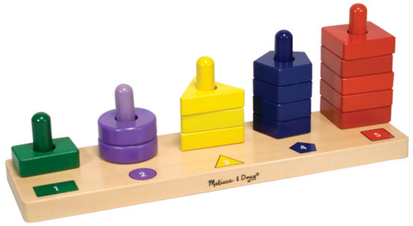


- Entrée déjà triée
- Complexité linéaire $\mathrm{O}(\mathrm{n})$



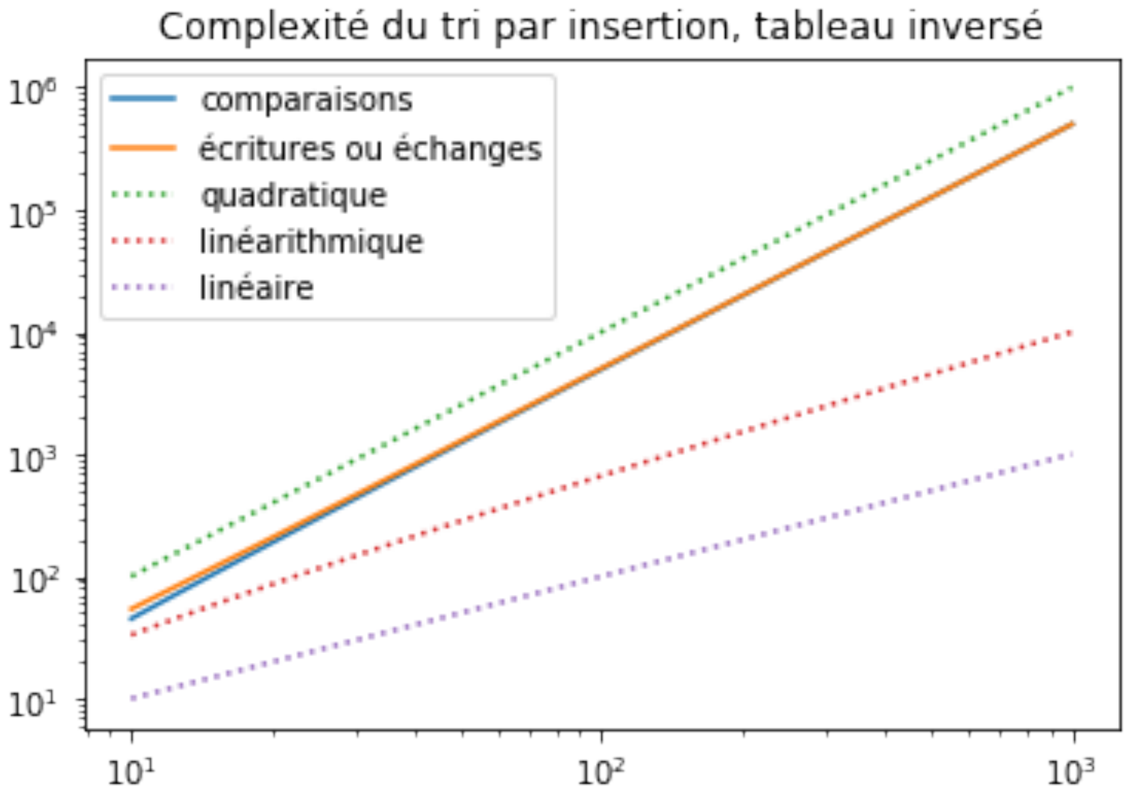
N	Comp.	Ecr.
10	9	9
19	18	18
37	36	36
71	70	70
138	137	137
268	267	267
517	516	516
1000	999	999

Insertion - pire cas



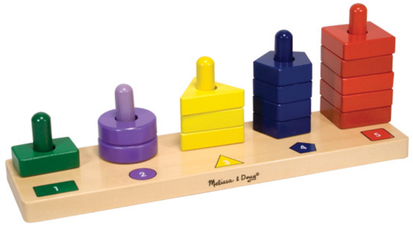
- Entrée triée à l'envers

- Complexité quadratique $\mathrm{O}(\mathrm{n}^2)$

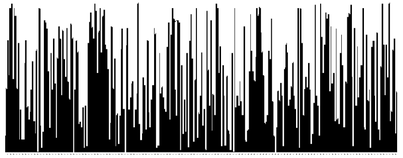


N	Comp.	Ecr.
10	45	54
19	171	189
37	666	702
71	2485	2555
138	9453	9590
268	35778	36045
517	133386	133902
1000	499500	500499

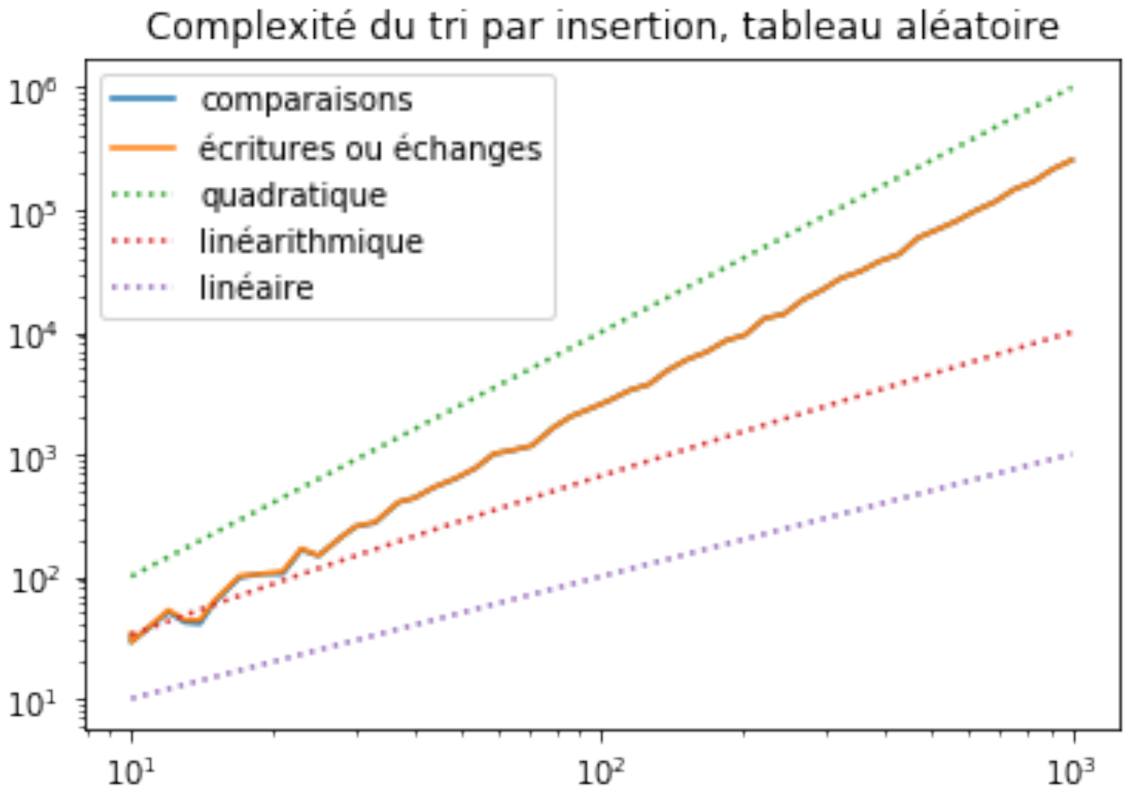
Complexité - cas moyen - O(n²)



- Entrée aléatoire

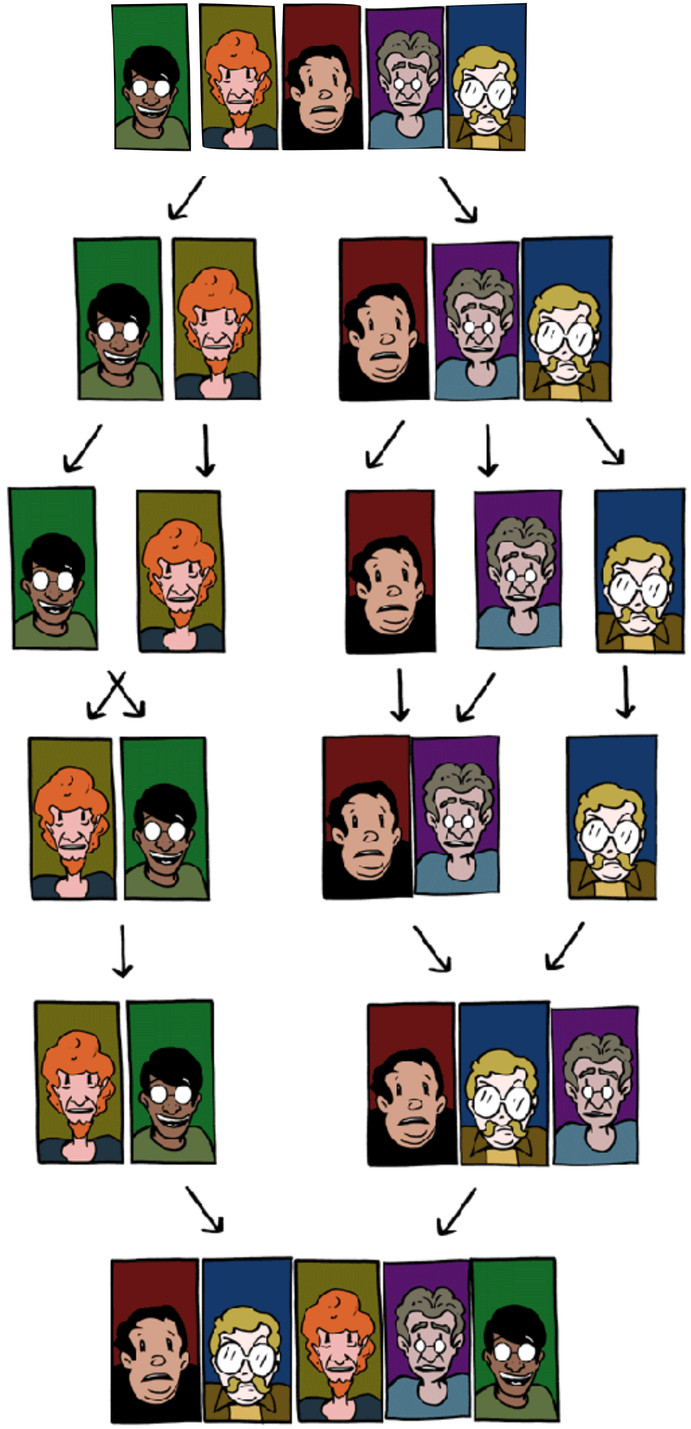


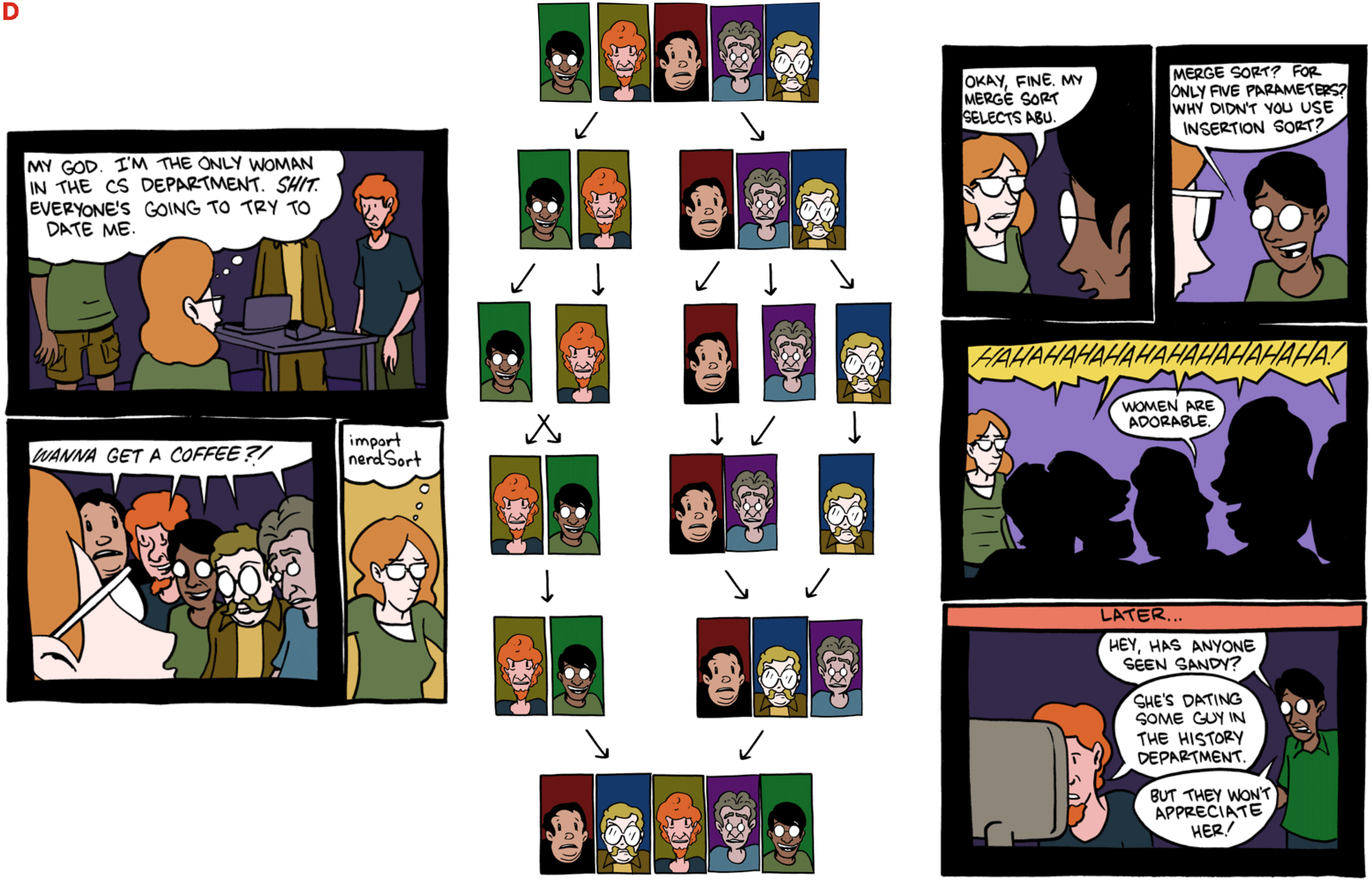
- Complexité quadratique $\mathrm{O}(\mathrm{n}^2)$



N	Comp.	Ecr.
10	33	34
19	105	105
37	403	406
71	1167	1171
138	4828	4831
268	18327	18333
517	68577	68582
1000	254774	254779

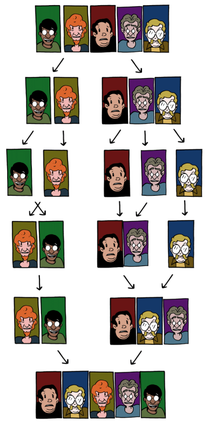
3.2. Tri par fusion





http://www.smbc-comics.com/?id=1989

Tri par fusion



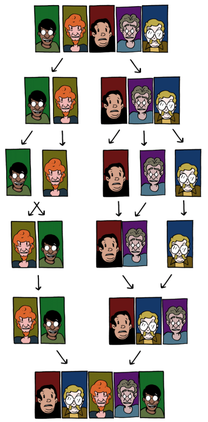
- Approche récursive
- Cas trivial :

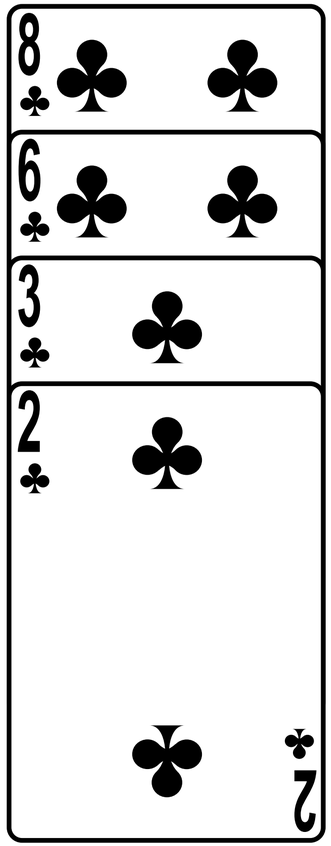
  Comment trier un tableau de 1 élément ?

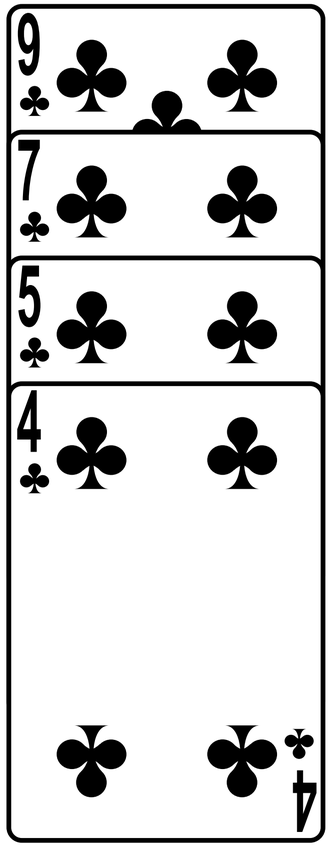
- Cas général :

  Est-ce que savoir trier un tableau de n/2 éléments m'aide à trier un tableau de n éléments ?

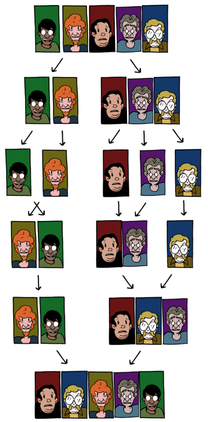
Fusionner deux tableaux triés

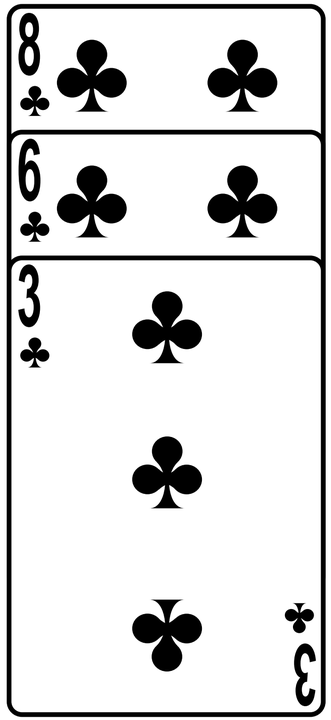


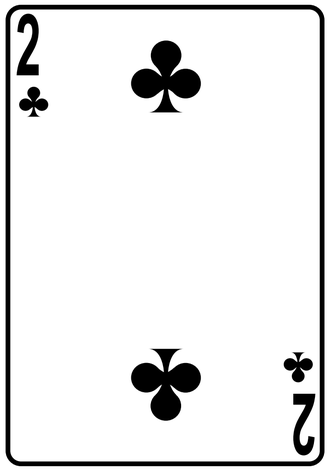


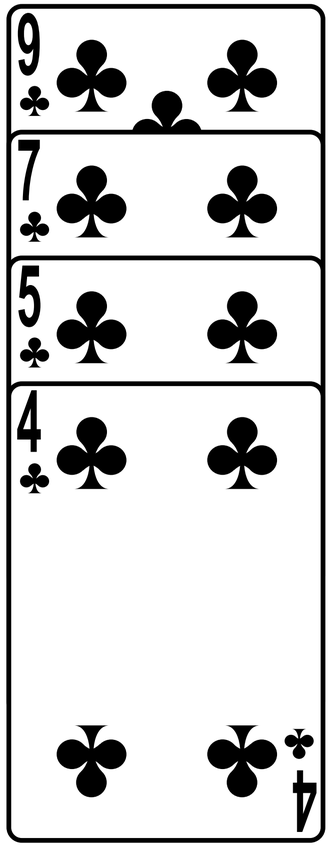


Fusionner deux tableaux triés

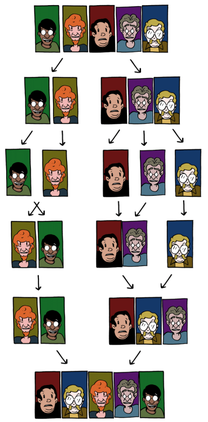


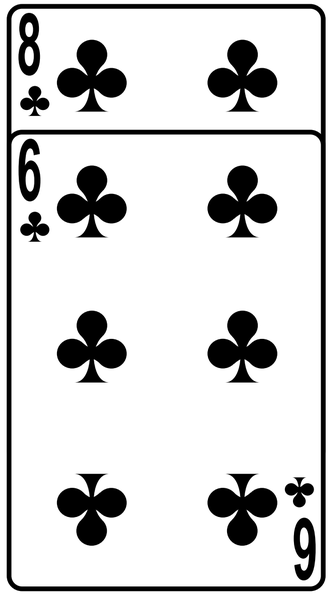


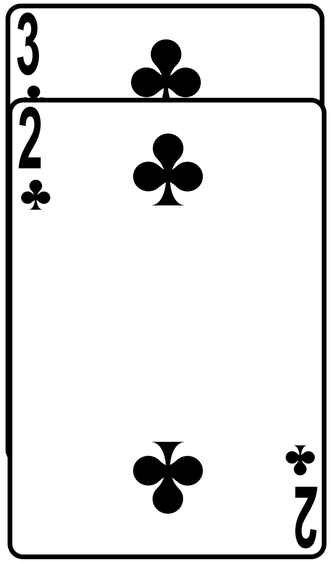


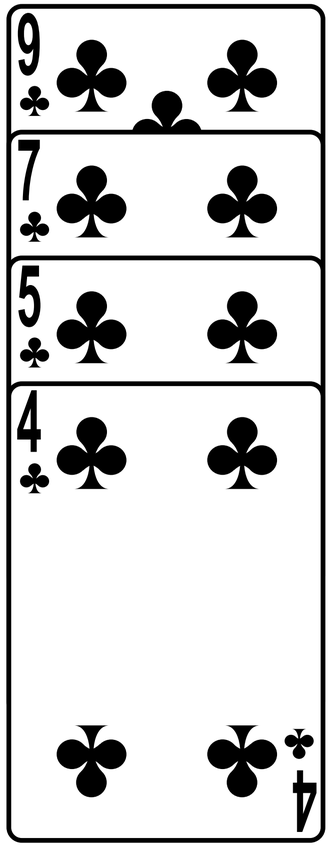


Fusionner deux tableaux triés

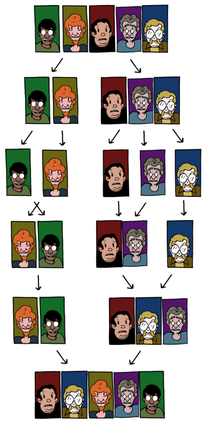


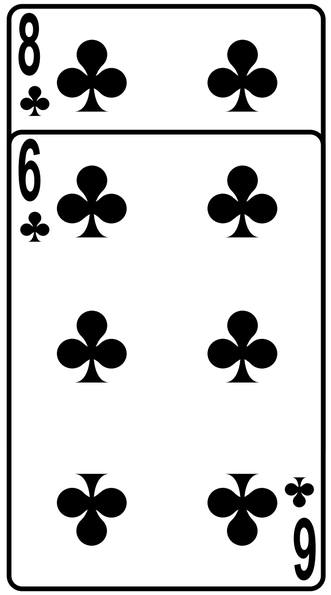


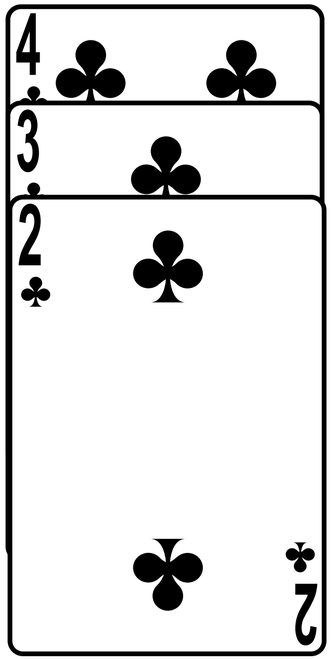


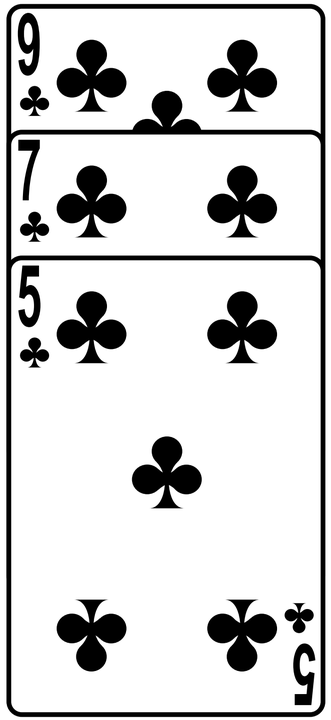


Fusionner deux tableaux triés

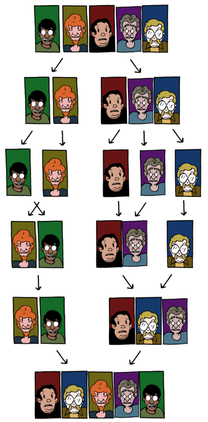


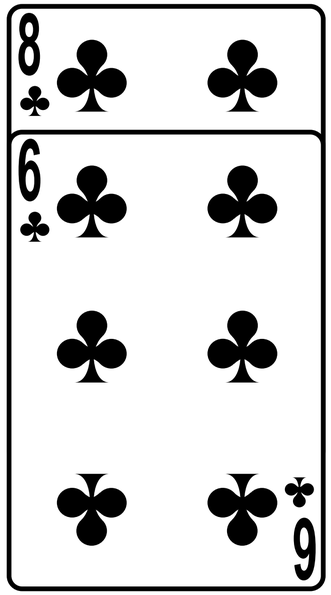


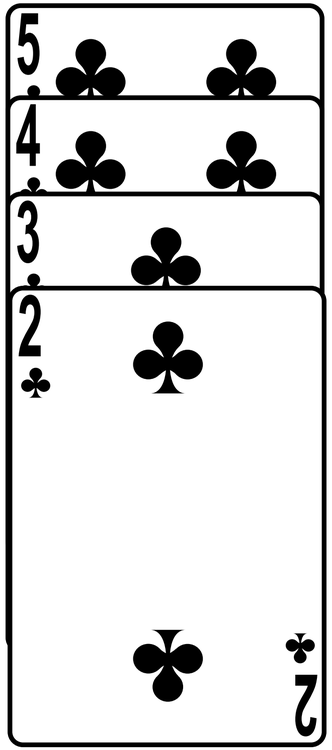


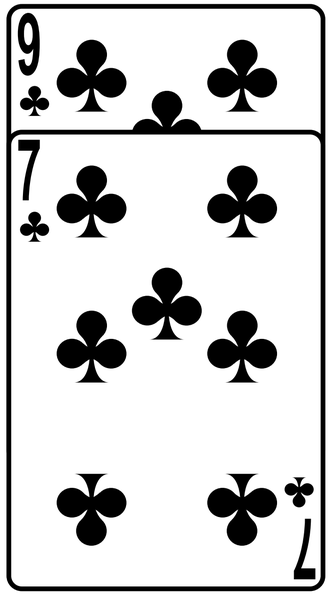


Fusionner deux tableaux triés

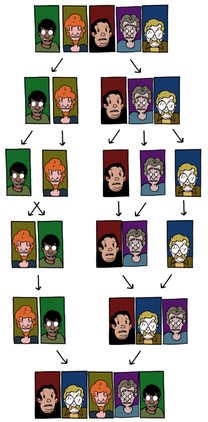


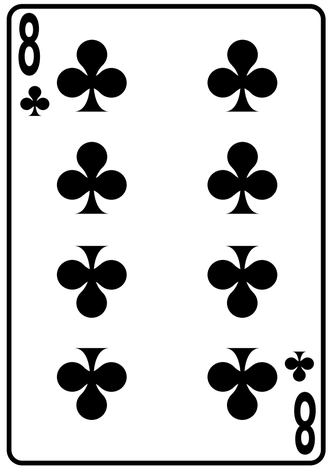


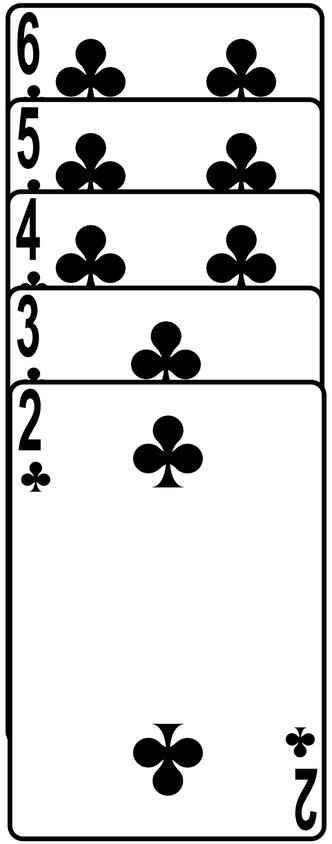


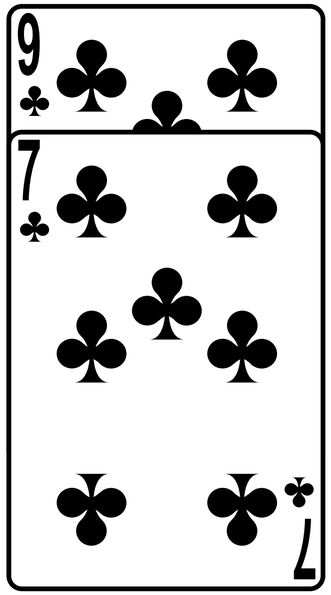


Fusionner deux tableaux triés

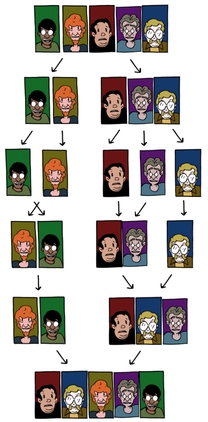


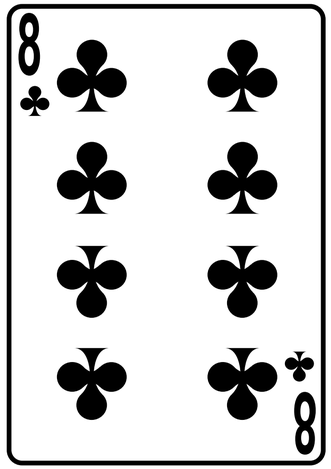


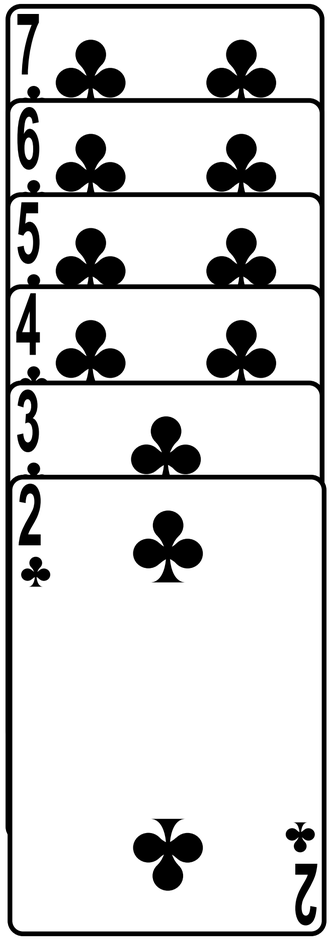


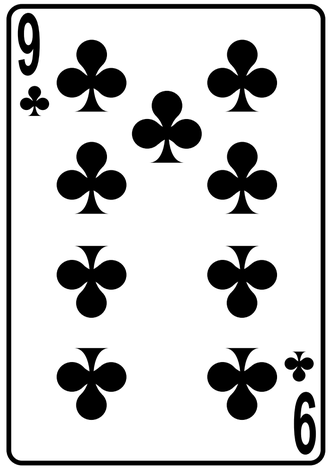


Fusionner deux tableaux triés

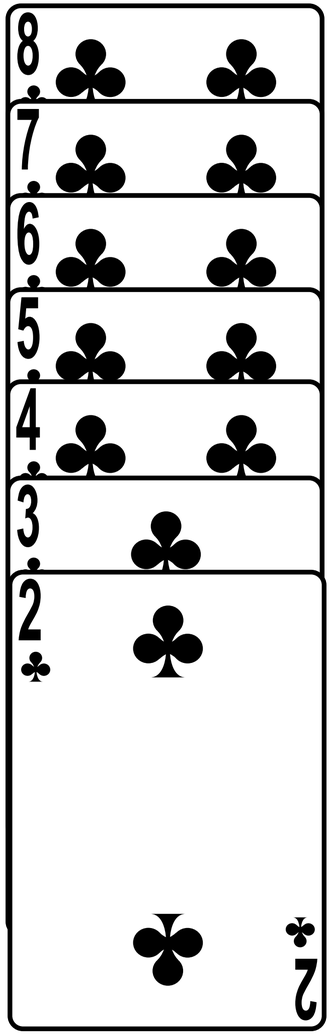


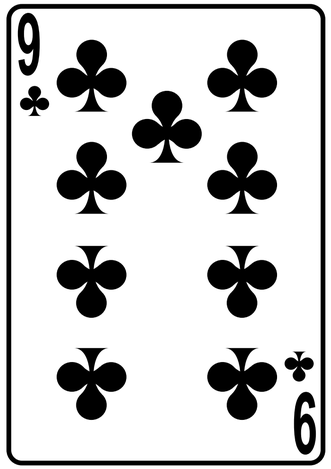




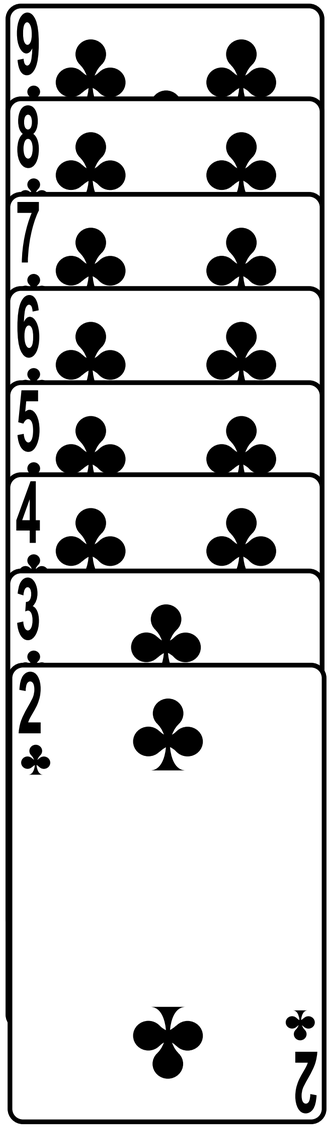


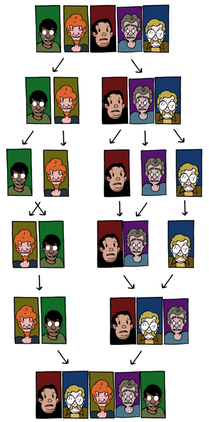
Fusionner deux tableaux triés



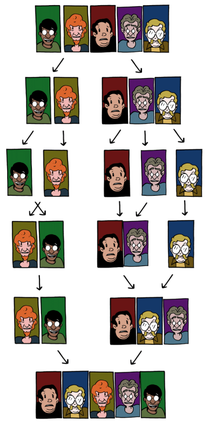


Fusionner deux tableaux triés

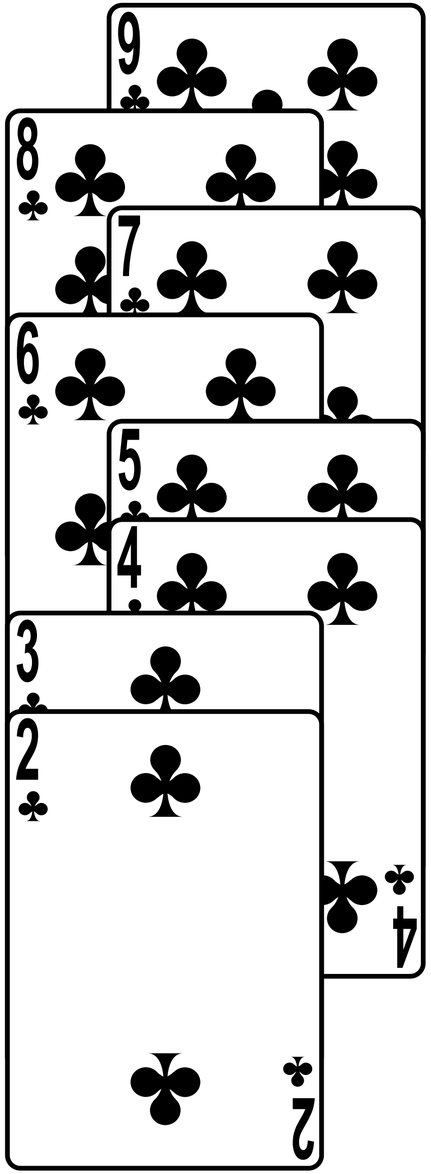




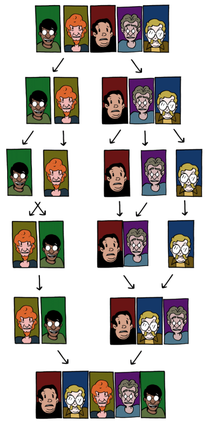
Fusionner deux tableaux triés



- Entrée : 2 tableaux triés par ordre croissant
- Sortie : 1 tableau fusionné trié par ordre croissant
- Algorithme :
  - Comparer les premiers (et donc plus petits) éléments des 2 tableaux
  - Déplacer le plus petit des 2 dans le tableau fusionné
  - Répéter jusqu'à ce que les 2 tableaux soient vides



Algorithme de fusion



Entrée :

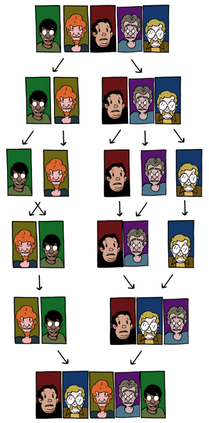
- Tableau A
- les éléments p à q du tableau A sont triés
- les éléments q+1 à r du tableau A sont triés

Sortie :

- Tableau A modifié
- les éléments p à r du tableau A sont triés.

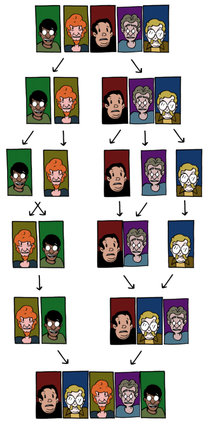
```
fonction Fusionner(A,p,q,r)
  L ← copie du tableau A de p à q
  R ← copie du tableau A de q+1 à r
  insérer une sentinelle ∞ en fin de L et R

  i ← 1, j ← 1
  pour k de p à r boucler
    si L(i) ≤ R(j), alors
      A(k) ← L(i)
      incrémenter i
    sinon
      A(k) ← R(j)
      incrémenter j
    fin si
  fin pour
```

Récursion



- Entrée : un tableau non trié
- Sortie : un tableau trié
- Algorithme :
  - Cas trivial : tableau de 0 ou 1 élément
  - Cas général : tableau de plusieurs éléments
    - Diviser le tableau en 2 parts les plus égales possibles
    - *Trier (récursivement)* les deux sous-tableaux
    - Fusionner les tableaux triés (en sortie de récursion)

Fonction récursive et appel à fusion



Entrée

- Tableau `A`
- L'intervalle `[lo,hi]` des indices à trier

Sortie

- Tableau `A` trié des indices `lo` à `hi`

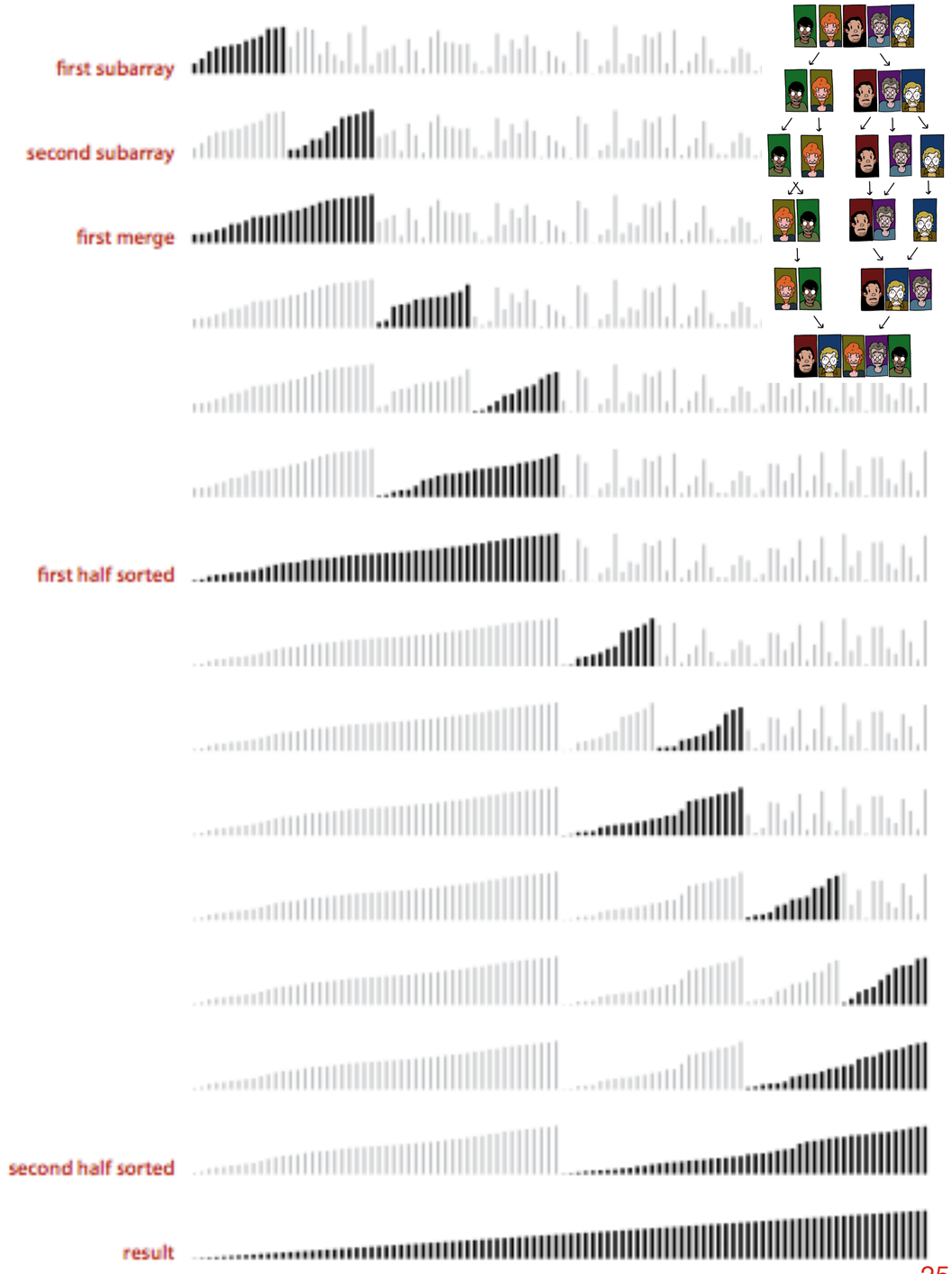
Pour trier tout le tableau `A` de taille `n`, on appelle `TriFusion(A,1,n)`

```
fonction TriFusion(A,lo,hi)

  si hi <= lo, alors
    retourner
  fin si

  mid ← lo + (hi-lo)/2

  TriFusion(A,lo,mid)
  TriFusion(A,mid+1,hi)

  Fusionner(A,lo,mid,hi)
```

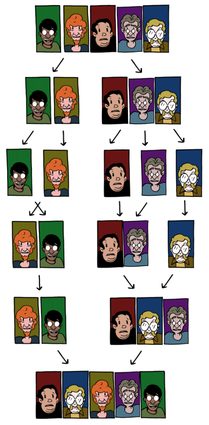
Visualisation

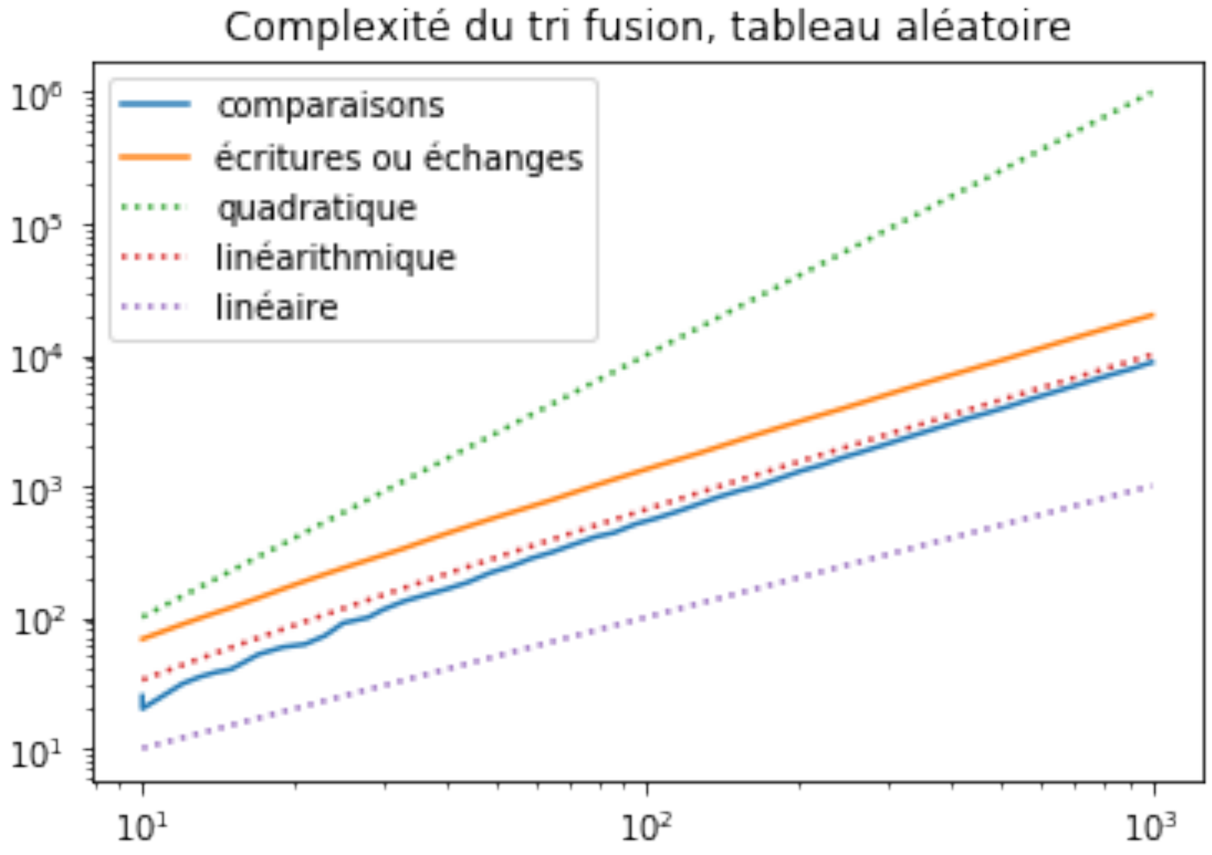
- Visualisation des étapes successives du tri par fusion

- On ne visualise pas les tris de sous-tableaux de moins d'une dizaine d'éléments.



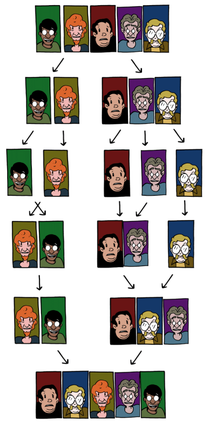
Complexité linéarithmique





N	Comp.	Ecr.
10	25	68
19	59	164
37	150	390
71	354	880
138	818	1972
268	1826	4336
517	3992	9326
1000	8731	19952

Complexité linéarithmique

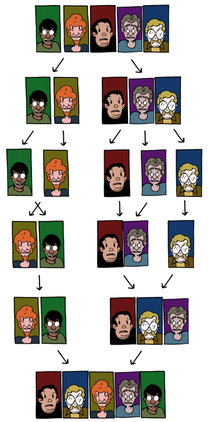


$$C(1) = 0$$

$$C(n) = n + 2 \cdot C(\frac{n}{2})$$

= 1 fusion + 2 appels récursifs

Complexité linéarithmique

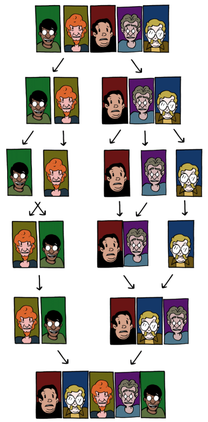


$$C(1) = 0$$

$$C(n) = n + 2 \cdot C(\frac{n}{2})$$ = 1 fusion + 2 appels récursifs

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot C(\frac{n}{4})$$

Complexité linéarithmique



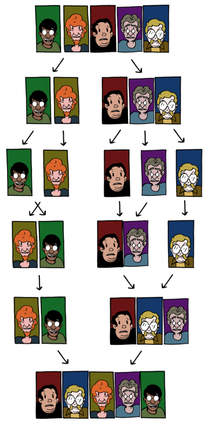
$$C(1) = 0$$

$$C(n) = n + 2 \cdot C(\frac{n}{2})$$ = 1 fusion + 2 appels récursifs

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot C(\frac{n}{4})$$

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot C(\frac{n}{8})$$

Complexité linéarithmique



$$C(1) = 0$$

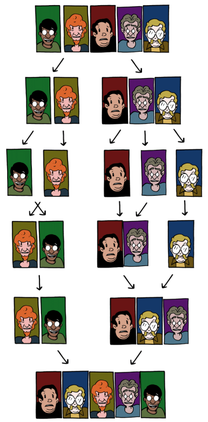
$$C(n) = n + 2 \cdot C(\frac{n}{2})$$ = 1 fusion + 2 appels récursifs

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot C(\frac{n}{4})$$

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot C(\frac{n}{8})$$

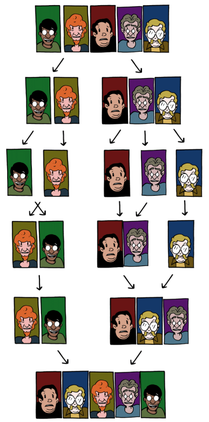
$$C(n) = n \cdot \log_2 n$$

Autres caractéristiques



- Complexité spatiale (mémoire)
  - Besoin d’un tableau auxiliaire de même taille que l’original

Autres caractéristiques

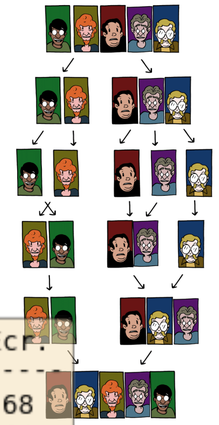


- Complexité spatiale (mémoire)
  - Besoin d'un tableau auxiliaire de même taille que l'original

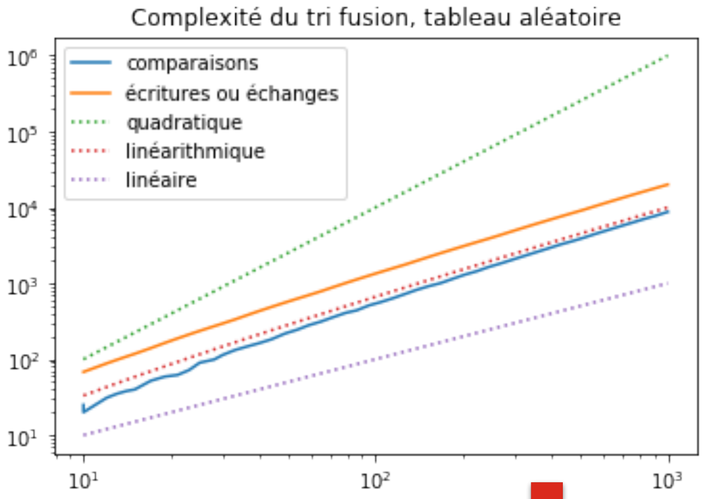
- Stabilité
  - Si `L(i)` et `R(j)` sont égaux
  - On choisit `L(i)`
  - Ce qui maintient l'ordre relatif

```
si L(i) ≤ R(j), alors
  A(k) ← L(i)
  incrémenter i
sinon
  A(k) ← R(j)
  incrémenter j
fin si
```

Minimizer le nombre d'écritures

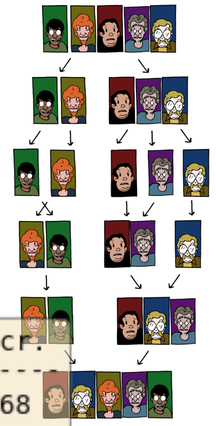


- Fusionner copie 2x, de A à LR, puis de LR à A

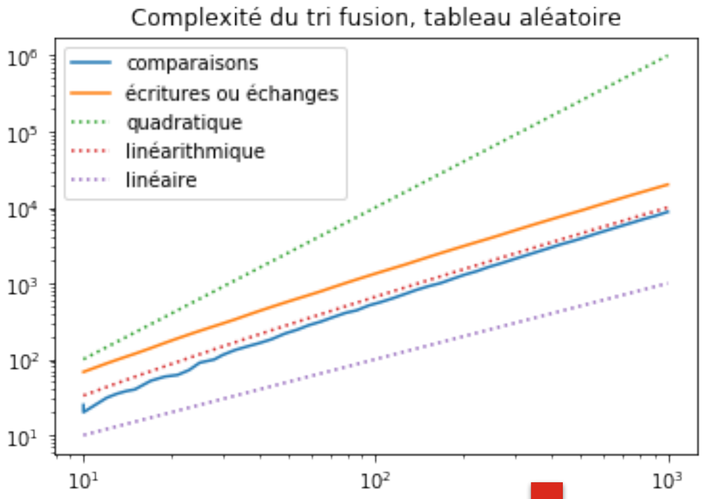


N	Comp.	Ecr.
10	25	68
19	59	164
37	150	390
71	354	880
138	818	1972
268	1826	4336
517	3992	9326
1000	8731	19952

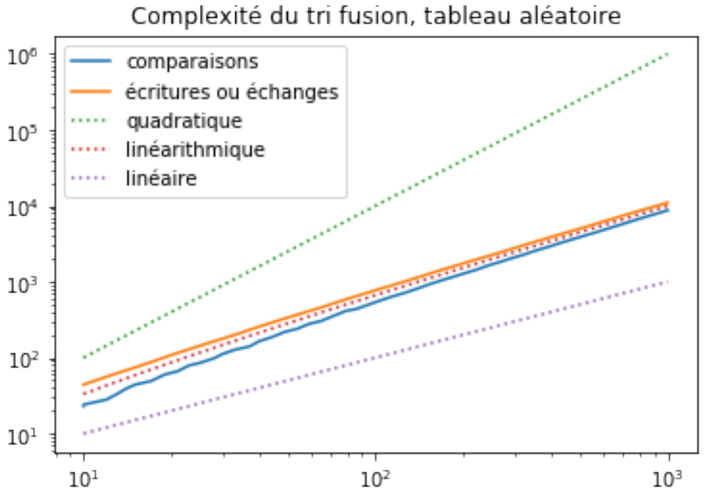
Minimizer le nombre d'écritures



- Fusionner copie 2x, de A à LR, puis de LR à A
- Pour ne faire qu'une copie
  - Allouer une seule fois un tableau T de même taille que A
  - Alterner les rôles de A et T d'une récursion à l'autre

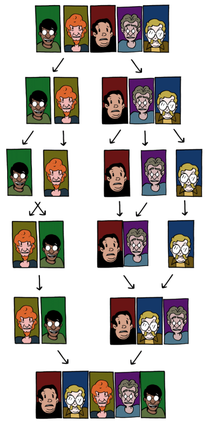


N	Comp.	Ecr.
10	25	68
19	59	164
37	150	390
71	354	880
138	818	1972
268	1826	4336
517	3992	9326
1000	8731	19952



N	Comp.	Ecr.
10	23	44
19	60	101
37	140	232
71	348	511
138	802	1124
268	1830	2436
517	4013	5180
1000	8721	10976

Autres optimisations possibles

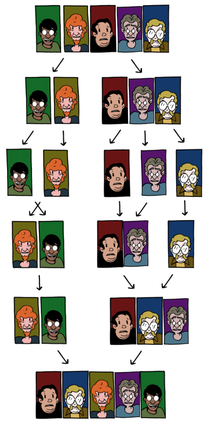


- La récursion est chère pour les petits tableaux.
  - Insertion sort est alors plus rapide.
  - Des méthodes hybrides (ex. TimSort) mélangent tris par fusion et par insertion
- Une mise en oeuvre itérative (bottom up) est également possible
- Une mise en oeuvre en place est possible, mais au prix d'une complexité en temps en $O(n \log^2(n))$

Demo

TriFusion(1,12)

E X E M P L E D E T R I



Noir - concernés par l'appel à TriFusion

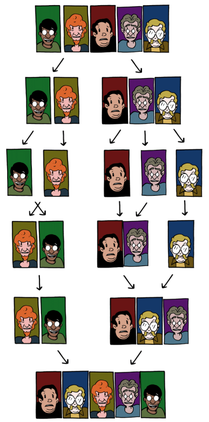
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

```
TriFusion(1,12)        E X E M P L E D E T R I
TriFusion(1,6)         E X E M P L E D E T R I
```



Noir - concernés par l'appel à TriFusion

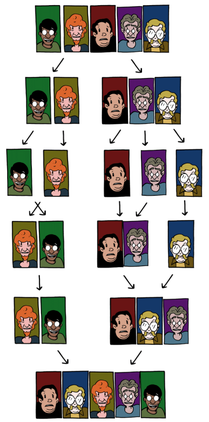
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

`TriFusion(1,12)`	E X E M P L E D E T R I
`TriFusion(1,6)`	E X E M P L E D E T R I
`TriFusion(1,3)`	E X E M P L E D E T R I



Noir - concernés par l'appel à TriFusion

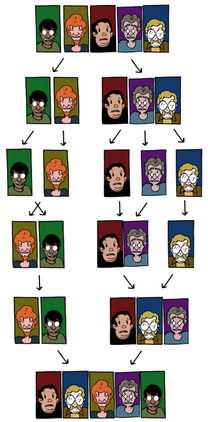
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

`TriFusion(1,12)`	E X E M P L E D E T R I
`TriFusion(1,6)`	E X E M P L E D E T R I
`TriFusion(1,3)`	E X E M P L E D E T R I
`TriFusion(1,2) ...`	E X E M P L E D E T R I



Noir - concernés par l'appel à TriFusion

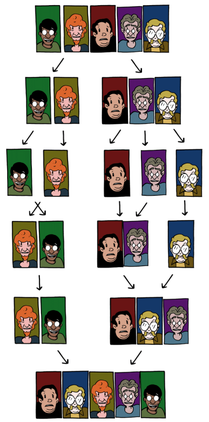
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I



Noir - concernés par l'appel à TriFusion

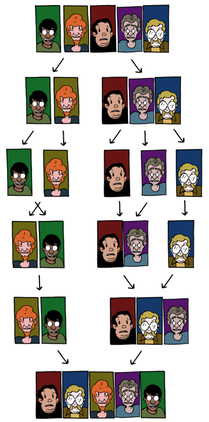
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I



Noir - concernés par l'appel à TriFusion

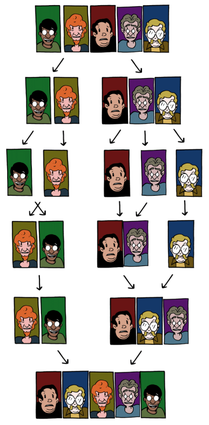
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I



Noir - concernés par l'appel à TriFusion

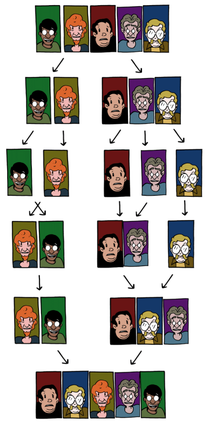
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I



Noir - concernés par l'appel à TriFusion

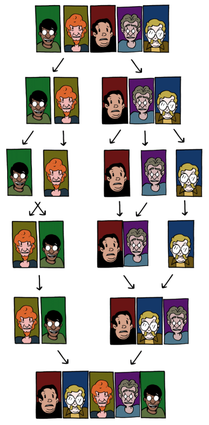
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I



Noir - concernés par l'appel à TriFusion

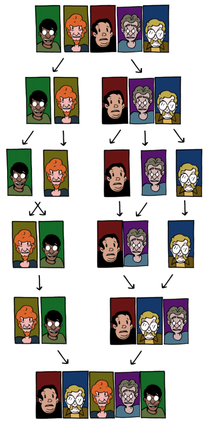
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I



Noir - concernés par l'appel à TriFusion

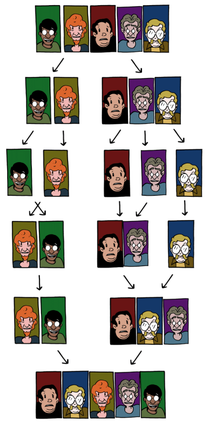
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I



Noir - concernés par l'appel à TriFusion

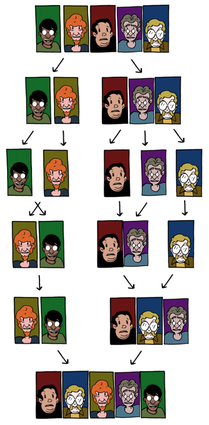
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I



Noir - concernés par l'appel à TriFusion

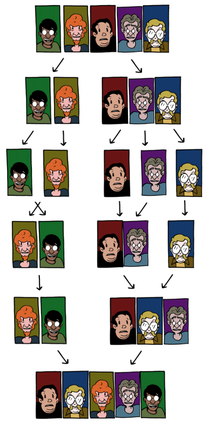
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I



Noir - concernés par l'appel à TriFusion

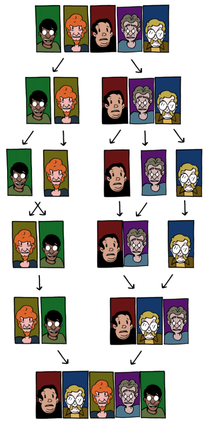
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I



Noir - concernés par l'appel à TriFusion

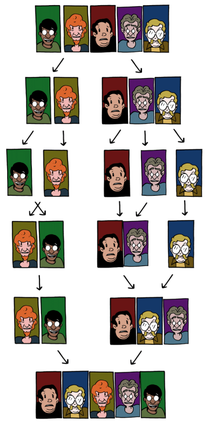
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I



Noir - concernés par l'appel à TriFusion

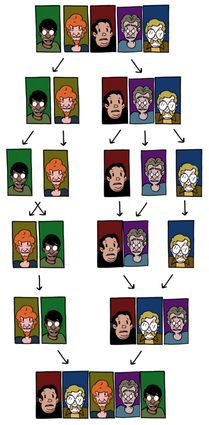
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I



Noir - concernés par l'appel à TriFusion

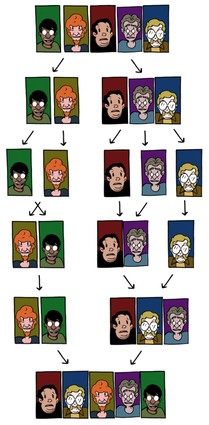
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I



Noir - concernés par l'appel à TriFusion

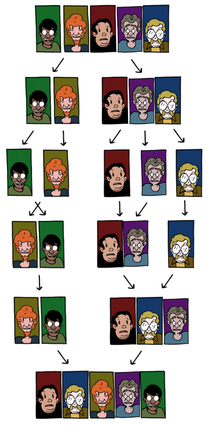
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I
TriFusion(10,11) ...	E E L M P X D E E T R I



Noir - concernés par l'appel à TriFusion

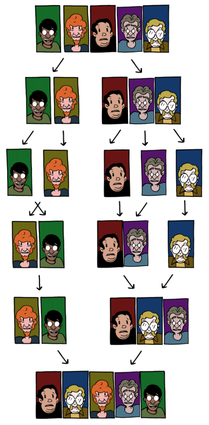
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I
TriFusion(10,11) ...	E E L M P X D E E T R I
Fusionner(11,11,12)	E E L M P X D E E R T I



Noir - concernés par l'appel à TriFusion

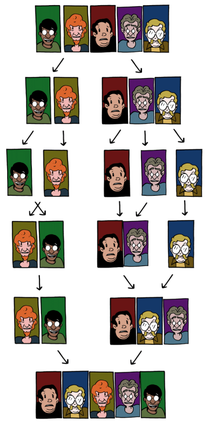
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I
TriFusion(10,11) ...	E E L M P X D E E T R I
Fusionner(11,11,12)	E E L M P X D E E R T I
Fusionner(10,11,12)	E E L M P X D E E I R T



Noir - concernés par l'appel à TriFusion

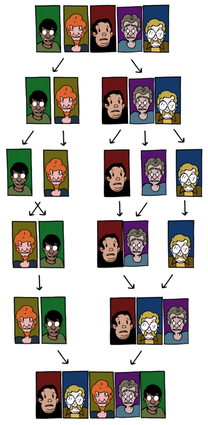
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I
TriFusion(10,11) ...	E E L M P X D E E T R I
Fusionner(11,11,12)	E E L M P X D E E R T I
Fusionner(10,11,12)	E E L M P X D E E I R T
Fusionner(7,9,12)	E E L M P X D E E I R T



Noir - concernés par l'appel à TriFusion

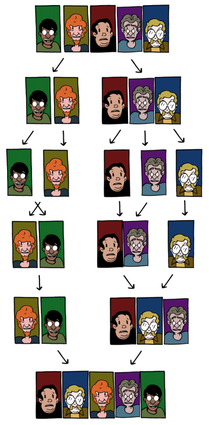
Bleu - fusionnés

Gris - pas concernés par la procédure en cours

Point (.) - appel à TriFusion sur un tableau de taille 1

Demo

TriFusion(1,12)	E X E M P L E D E T R I
TriFusion(1,6)	E X E M P L E D E T R I
TriFusion(1,3)	E X E M P L E D E T R I
TriFusion(1,2) ...	E X E M P L E D E T R I
Fusionner(1,1,2)	E X E M P L E D E T R I
Fusionner(1,2,3)	E E X M P L E D E T R I
TriFusion(4,6)	E E X M P L E D E T R I
TriFusion(4,5) ...	E E X M P L E D E T R I
Fusionner(4,4,5)	E E X M P L E D E T R I
Fusionner(4,5,6)	E E X L M P E D E T R I
Fusionner(1,3,6)	E E L M P X E D E T R I
TriFusion(7,12)	E E L M P X E D E T R I
TriFusion(7,9)	E E L M P X E D E T R I
TriFusion(7,8) ...	E E L M P X E D E T R I
Fusionner(7,7,8)	E E L M P X D E E T R I
Fusionner(7,8,9)	E E L M P X D E E T R I
TriFusion(10,12)	E E L M P X D E E T R I
TriFusion(10,11) ...	E E L M P X D E E T R I
Fusionner(11,11,12)	E E L M P X D E E R T I
Fusionner(10,11,12)	E E L M P X D E E I R T
Fusionner(7,9,12)	E E L M P X D E E I R T
Fusionner(1,6,12)	D E E E E I L M P R T X



Noir - concernés par l'appel à TriFusion

Bleu - fusionnés

Gris - pas concernés par la procédure en cours

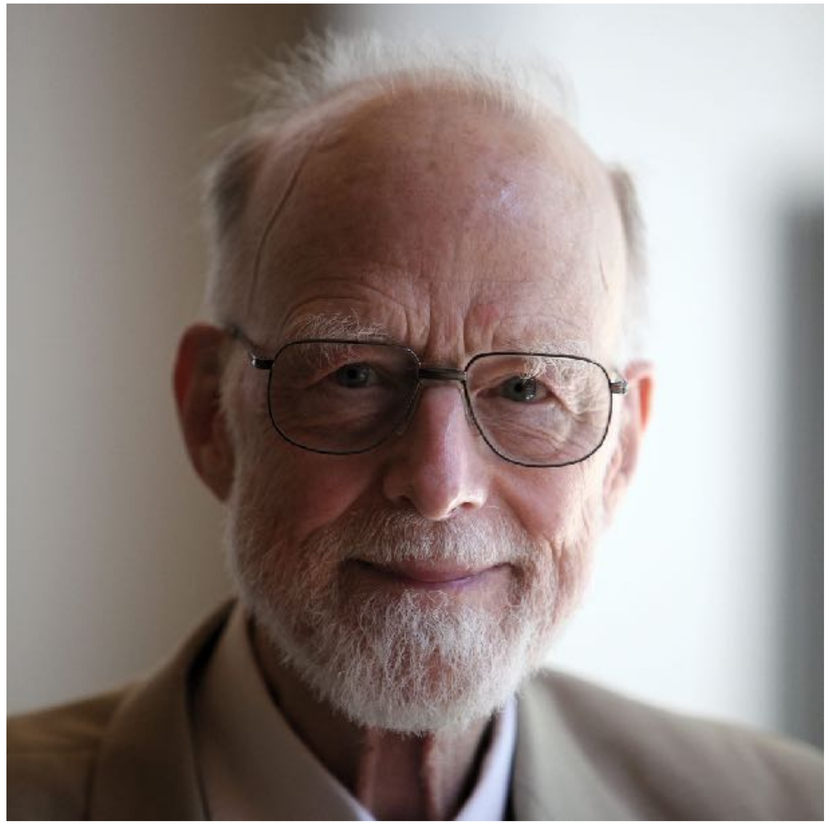
Point (.) - appel à TriFusion sur un tableau de taille 1

3.3 Tri Rapide

Tri rapide - Quicksort

- Développé par Tony Hoare (*Communications of the ACM,* 1959), étudiant invité à Moscou (URSS) pour travailler sur un logiciel de traduction du russe à l'anglais.

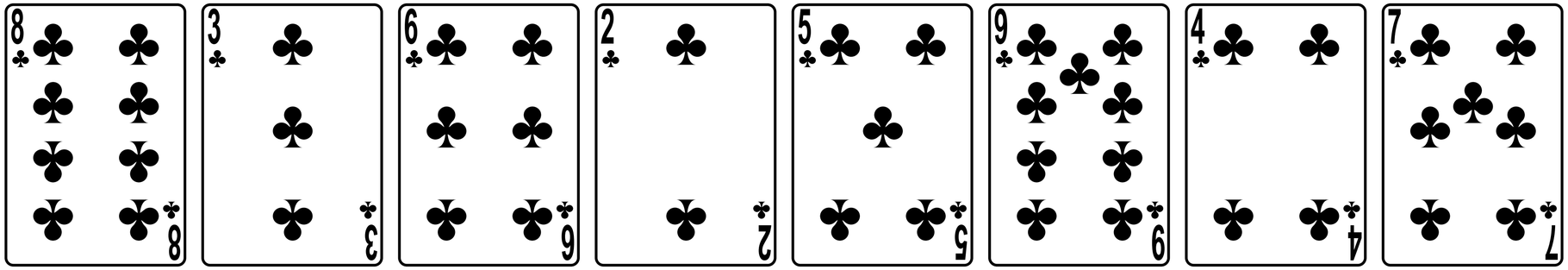
- Comme pour le tri fusion, l'approche récursive, mais …
  - On manipule avant les appels récursifs
  - La taille / position des 2 sous-tableaux à trier récursivement dépend des données



Sir Charles Antony Richard Hoare

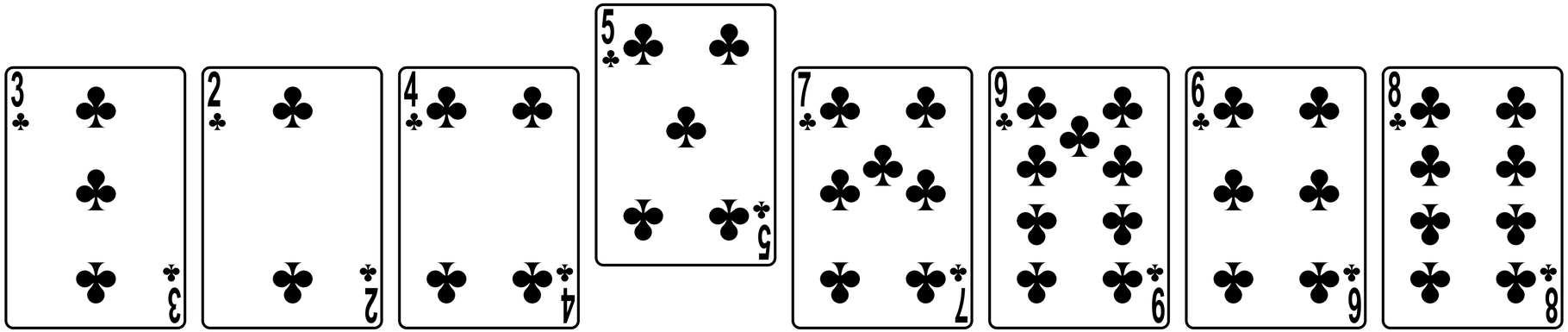
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



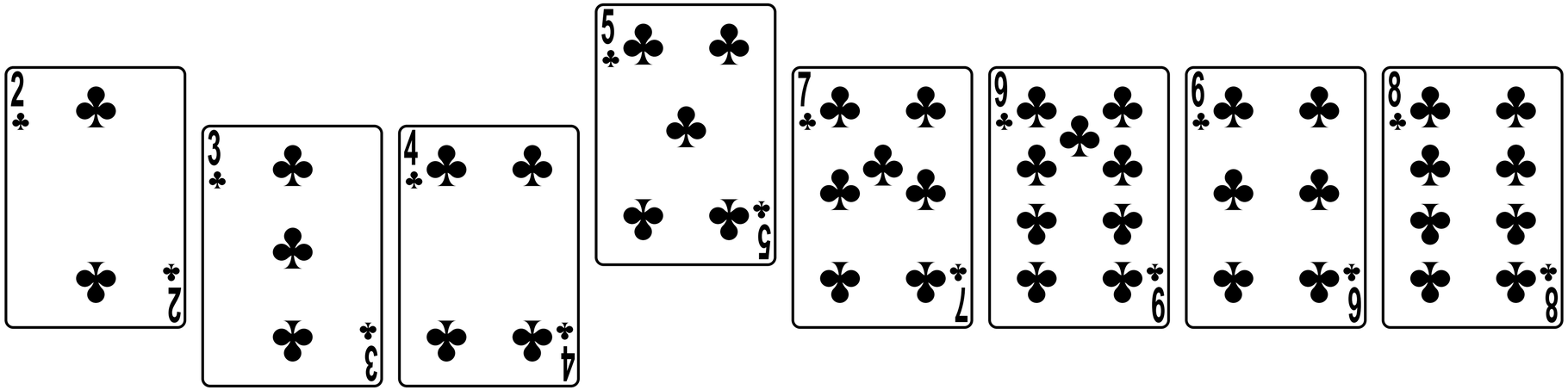
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



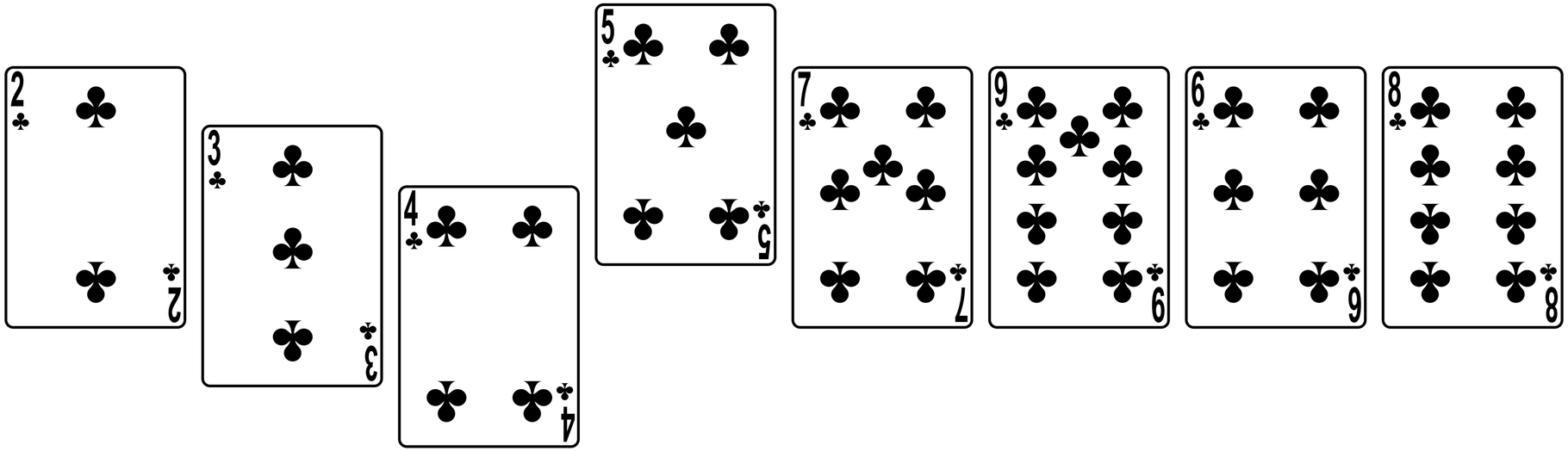
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



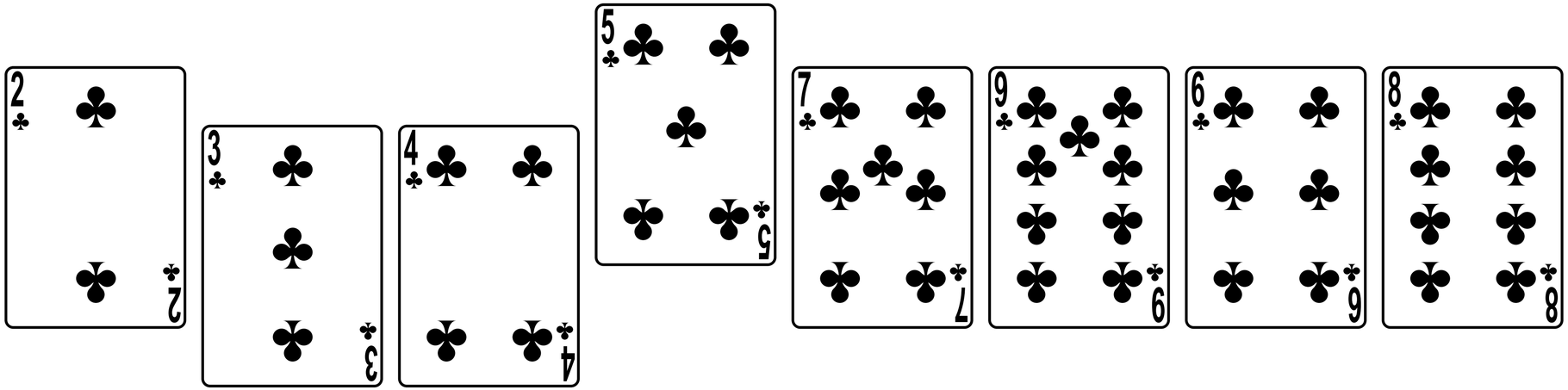
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



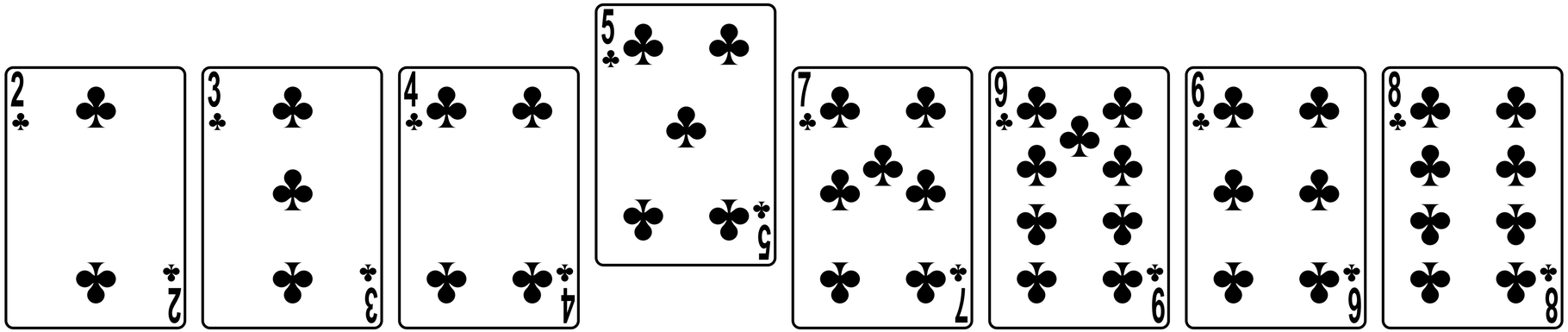
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



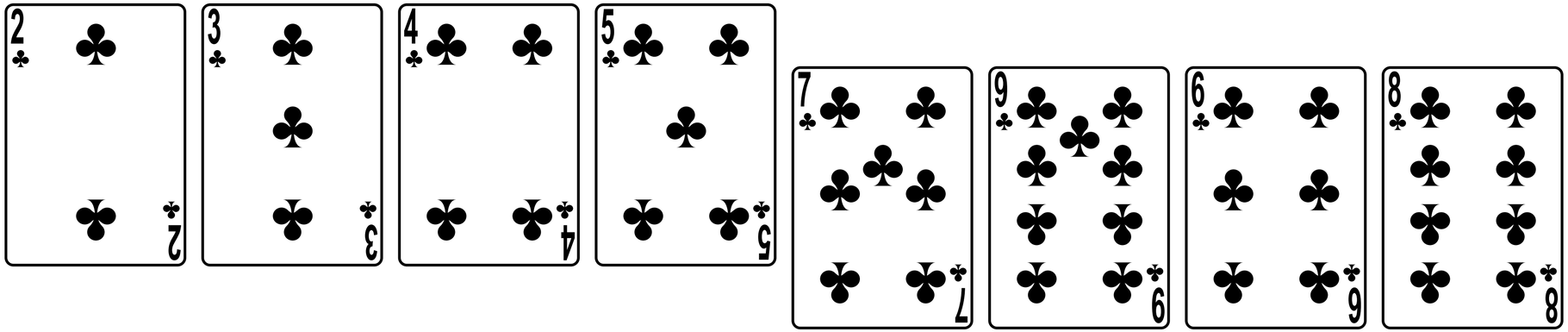
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



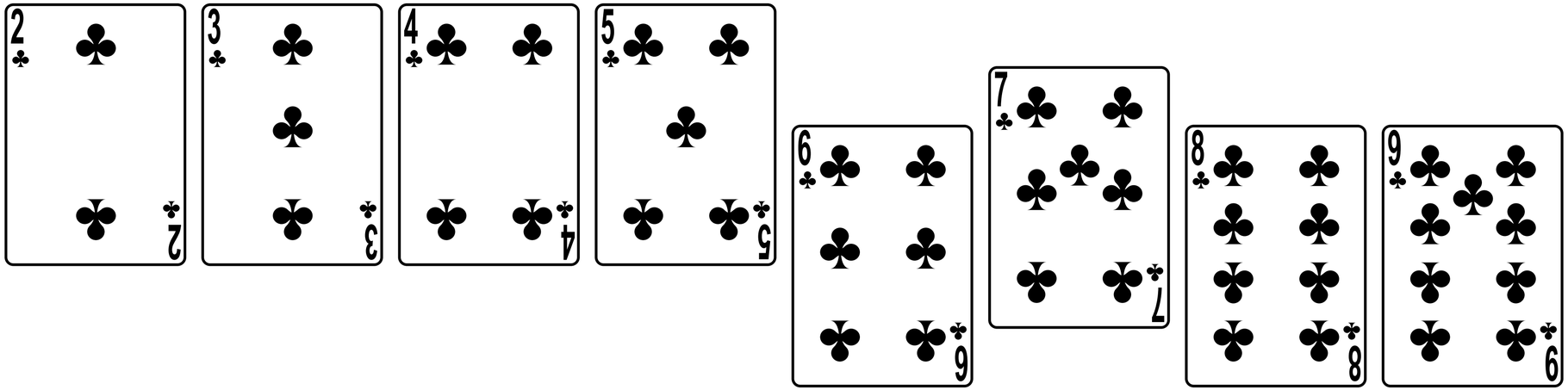
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



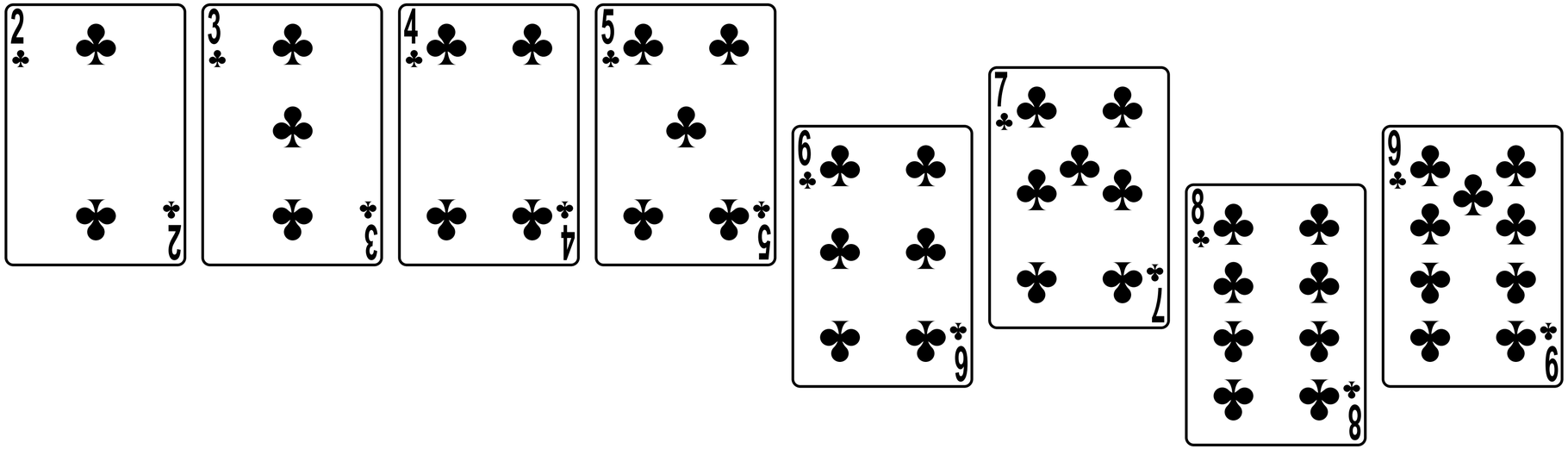
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



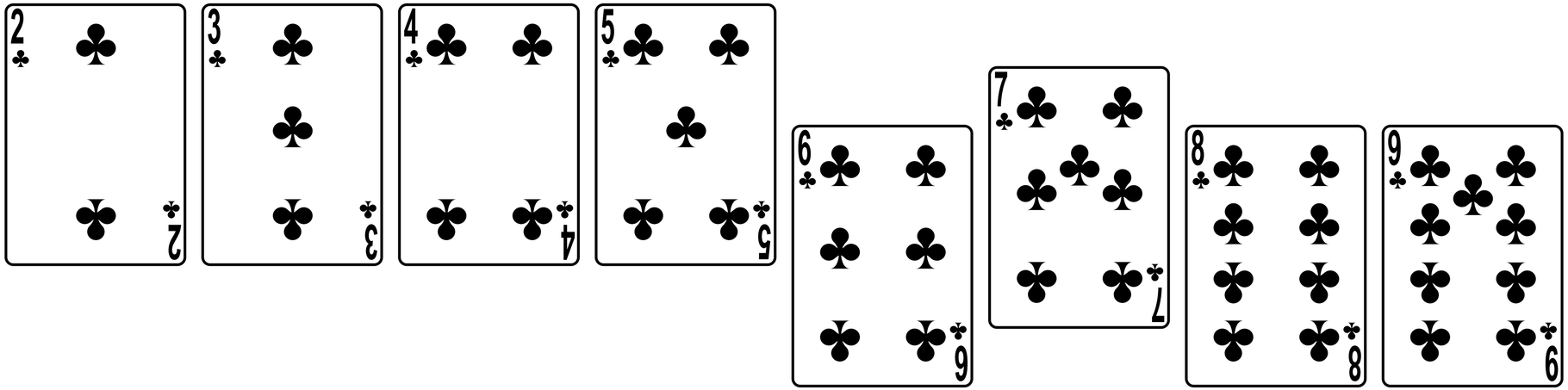
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



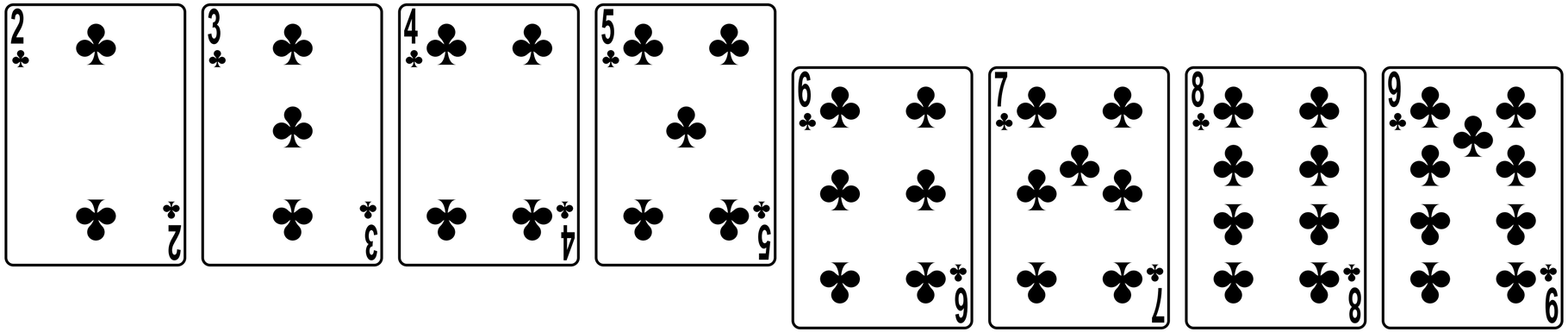
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p



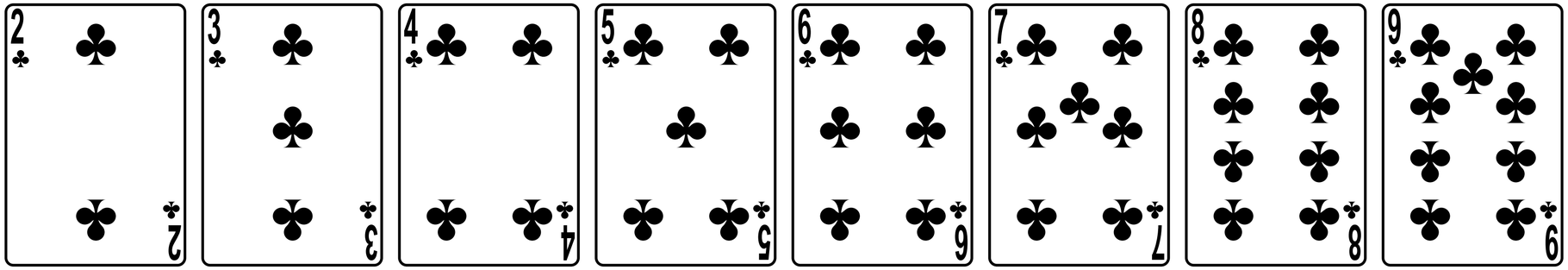
Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p

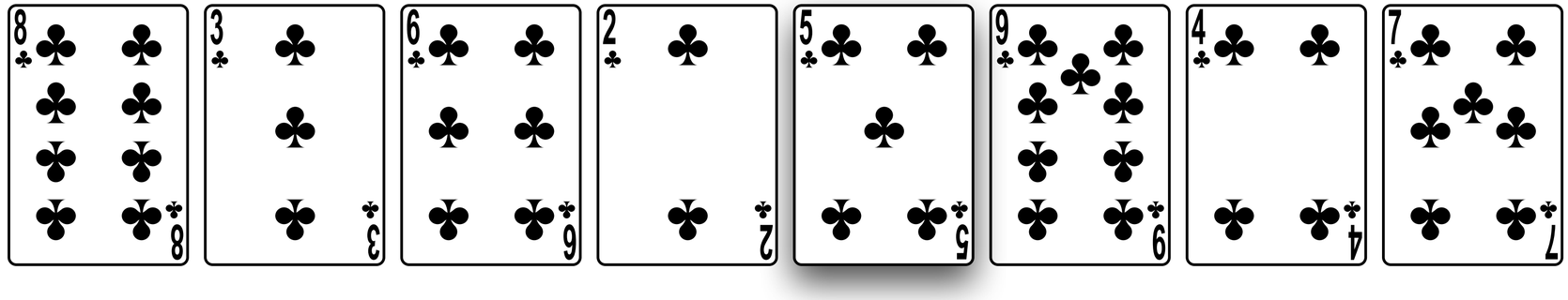


Partitions récursives

- Choisir un pivot p
- Placer les éléments e ≤ p à gauche et les e ≥ p à droite
- Trier récursivement les sous-tableaux à gauche et droite de p

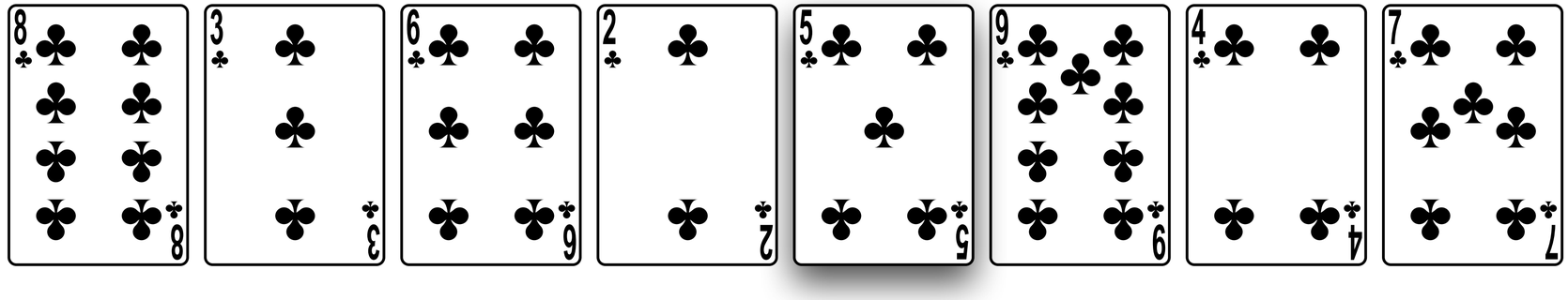


Partition en place



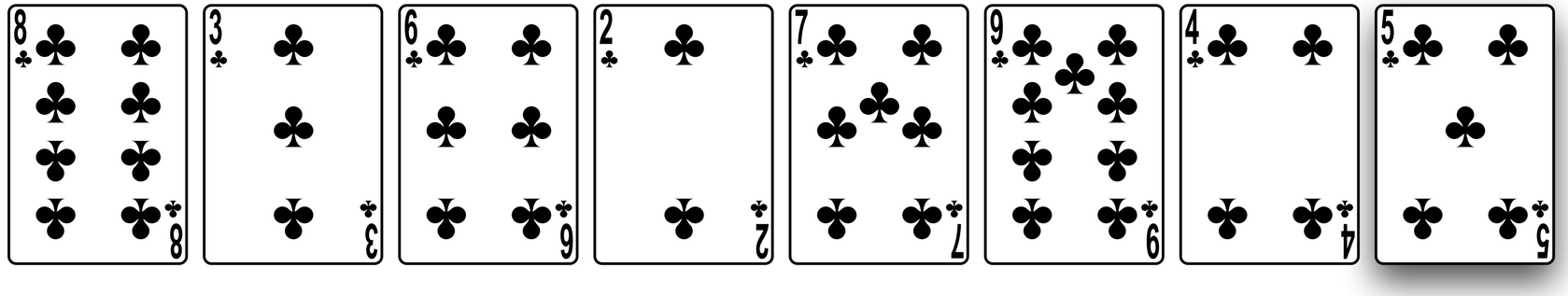
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

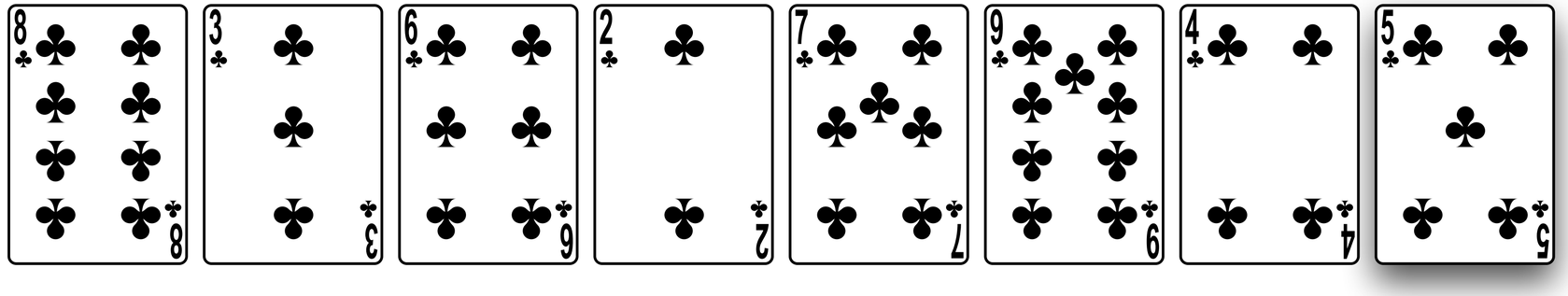
Partition en place



- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

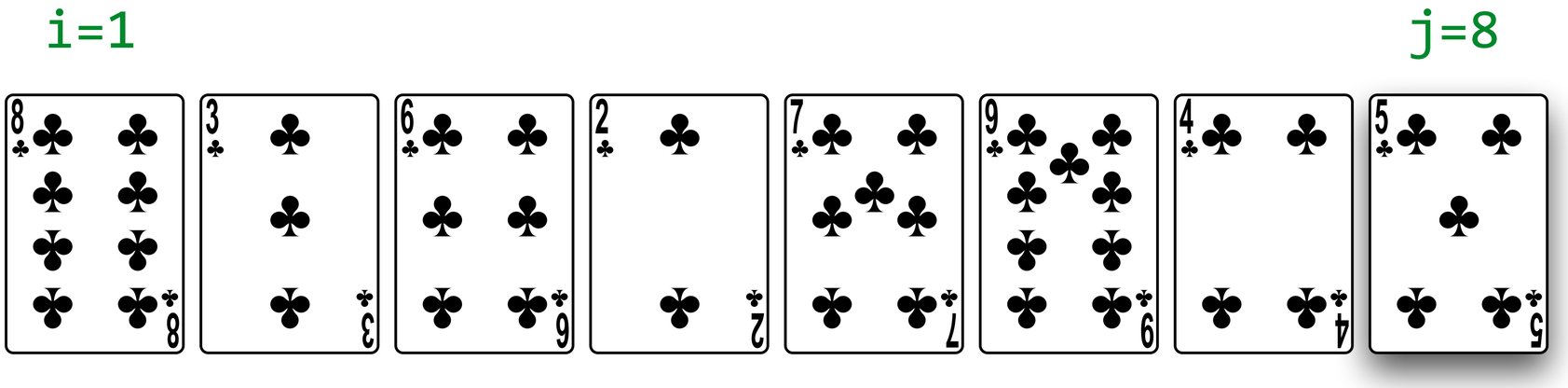
Partition en place

`i=0` `j=8`



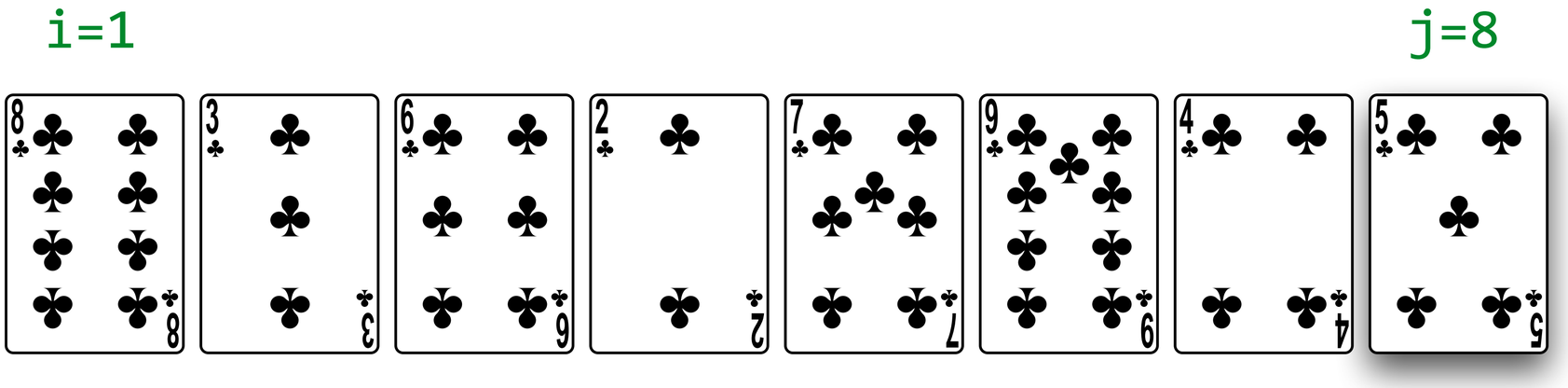
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



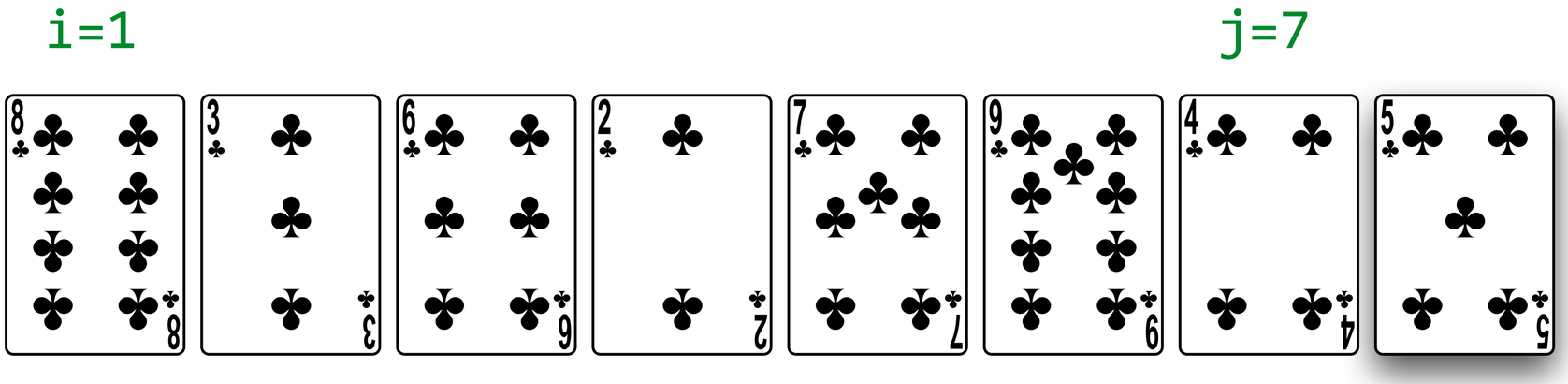
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



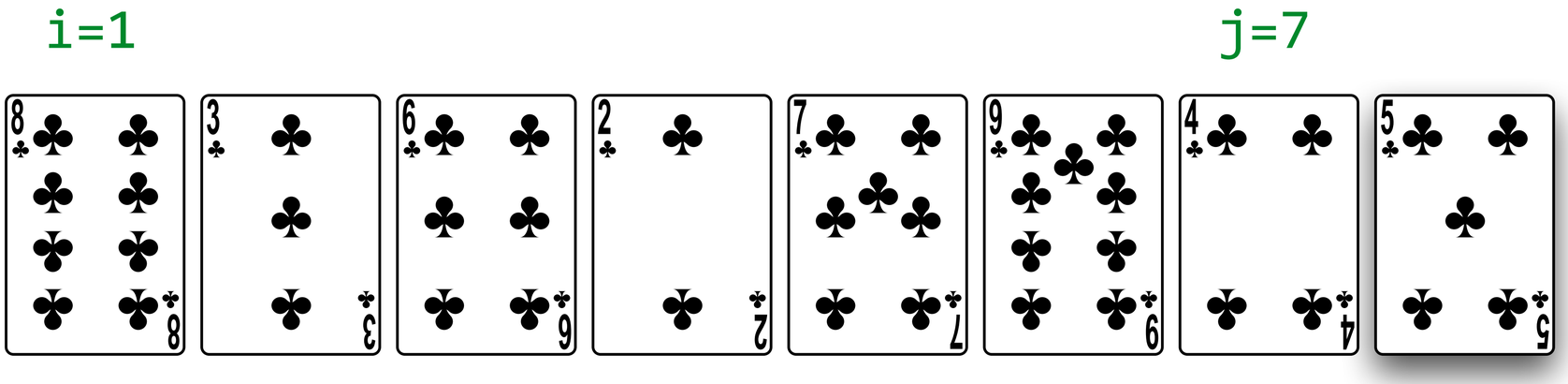
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



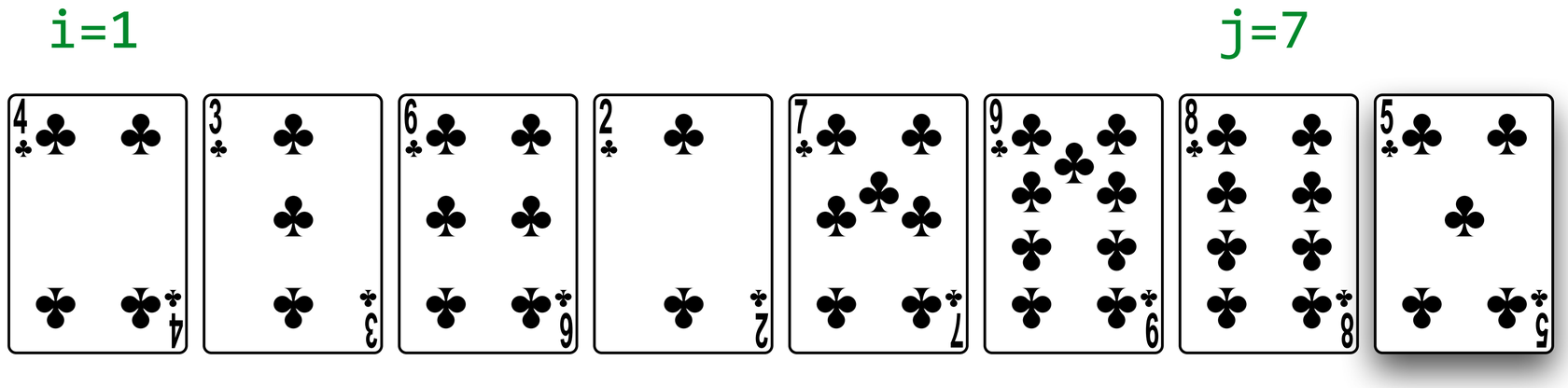
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



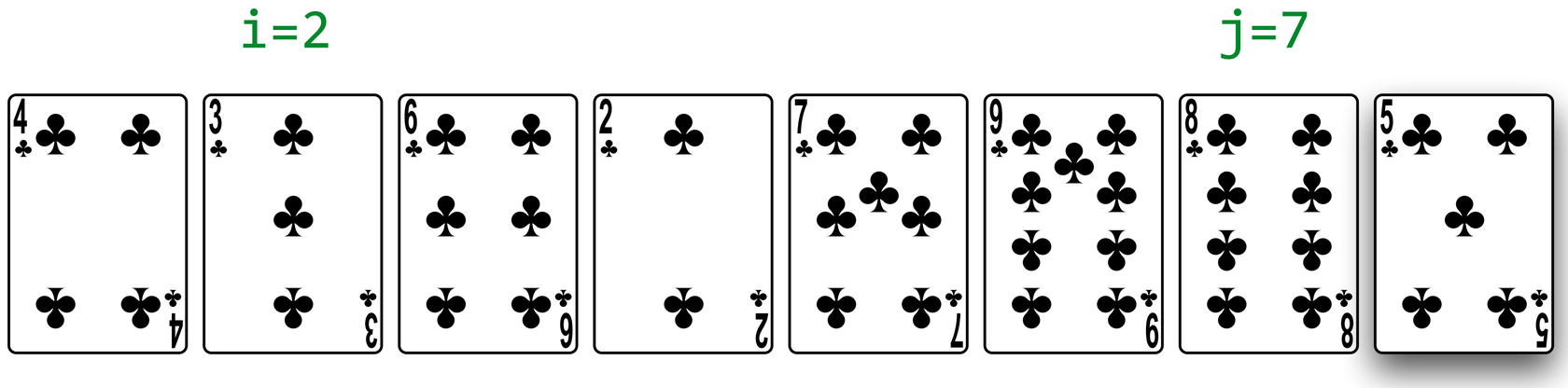
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



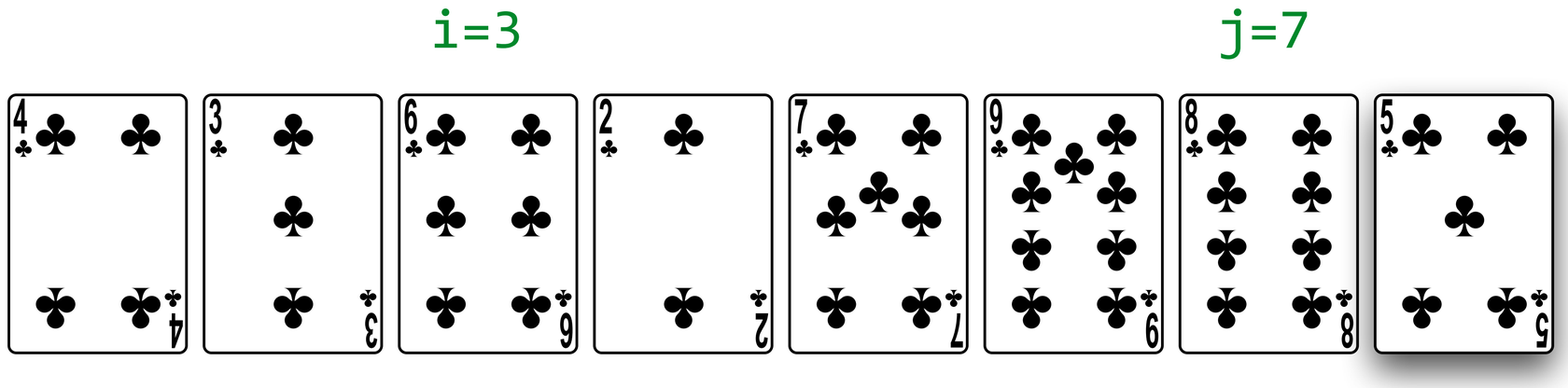
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



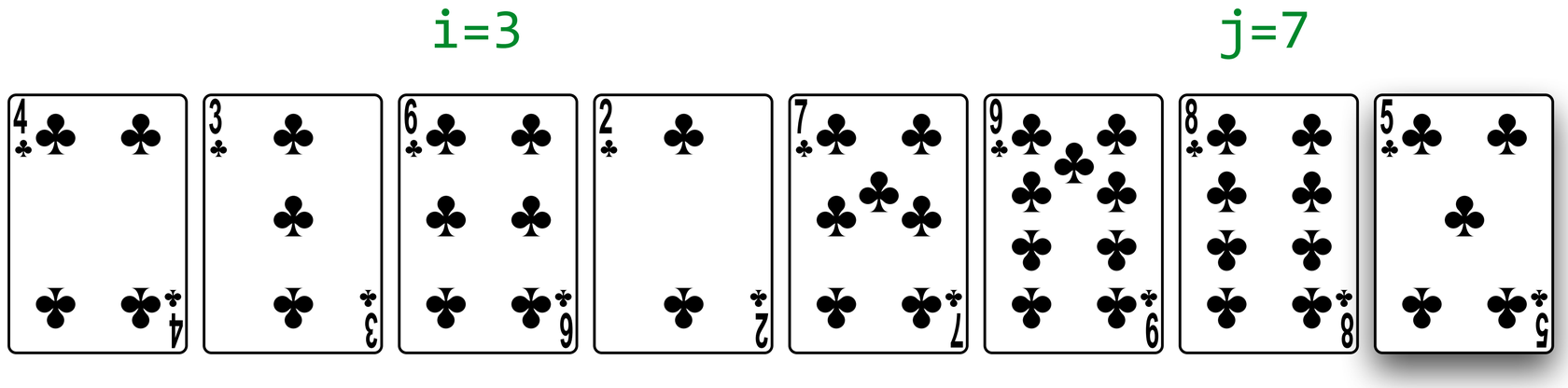
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



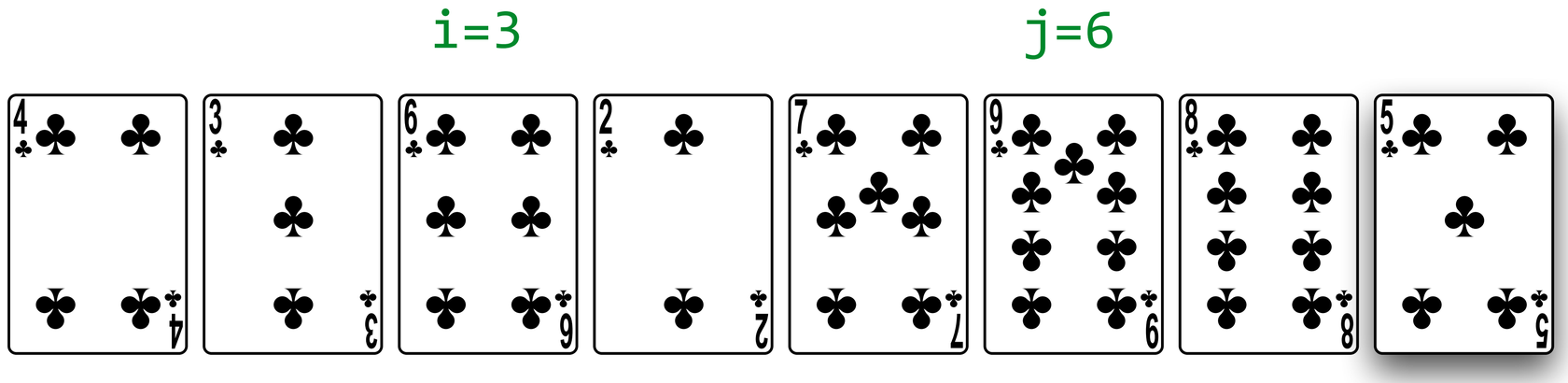
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



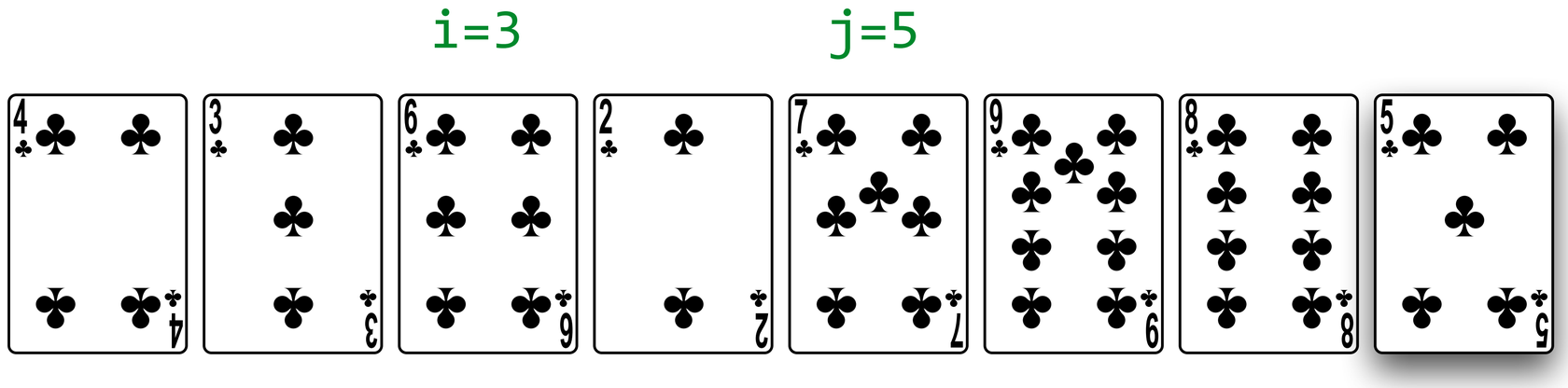
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



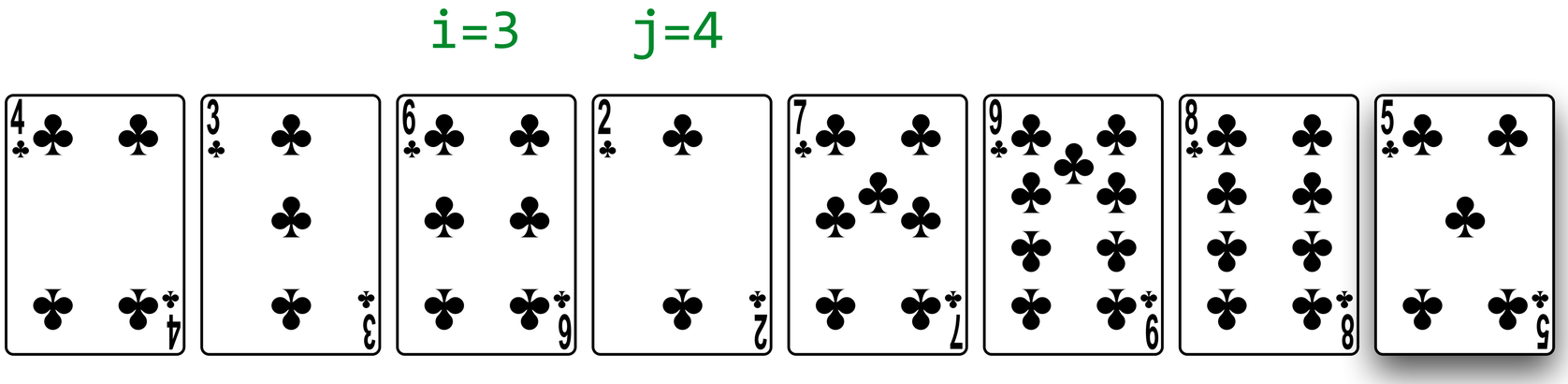
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



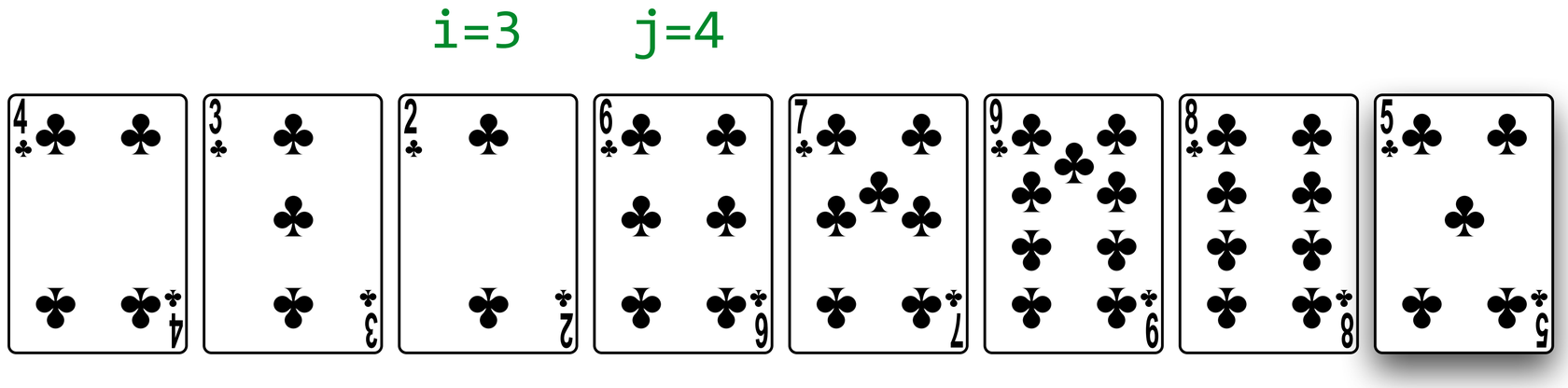
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place

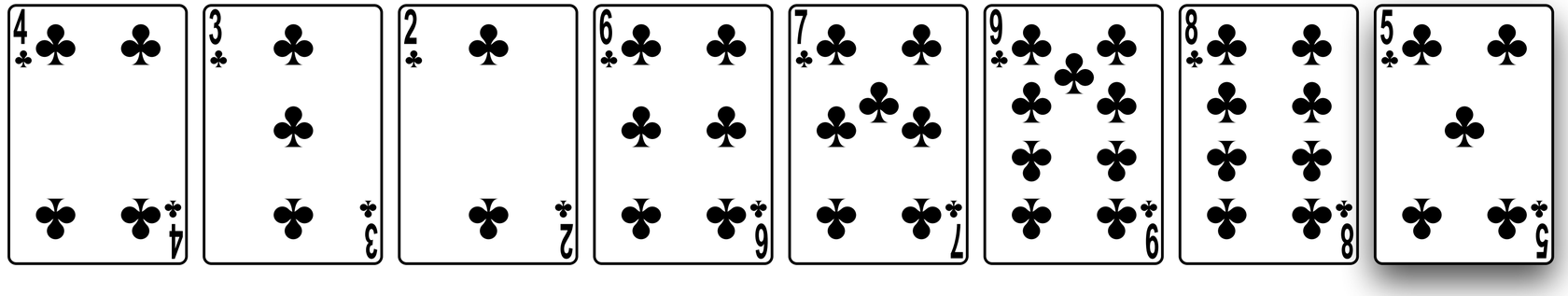


- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place

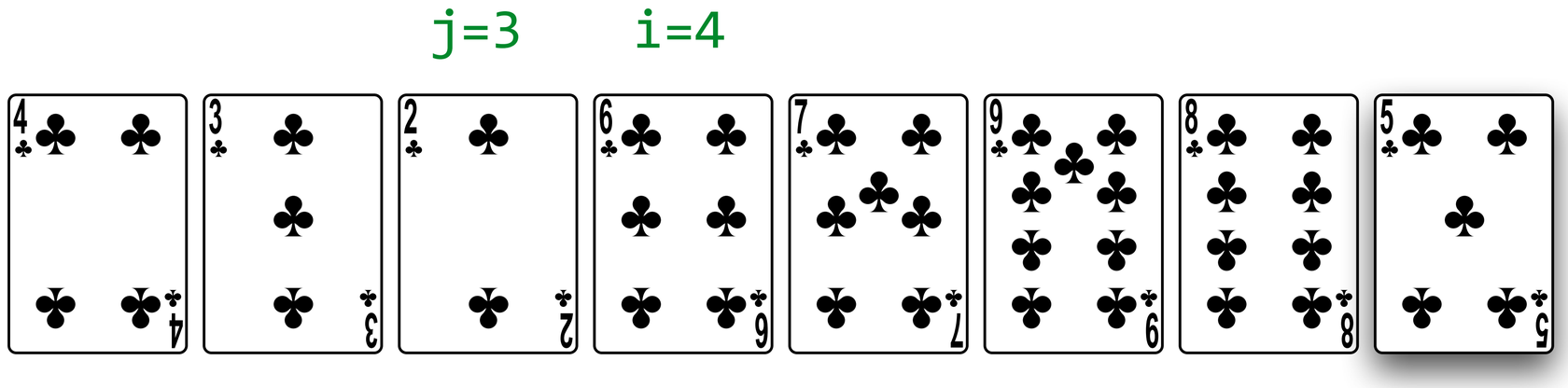
`i=4`

`j=4`



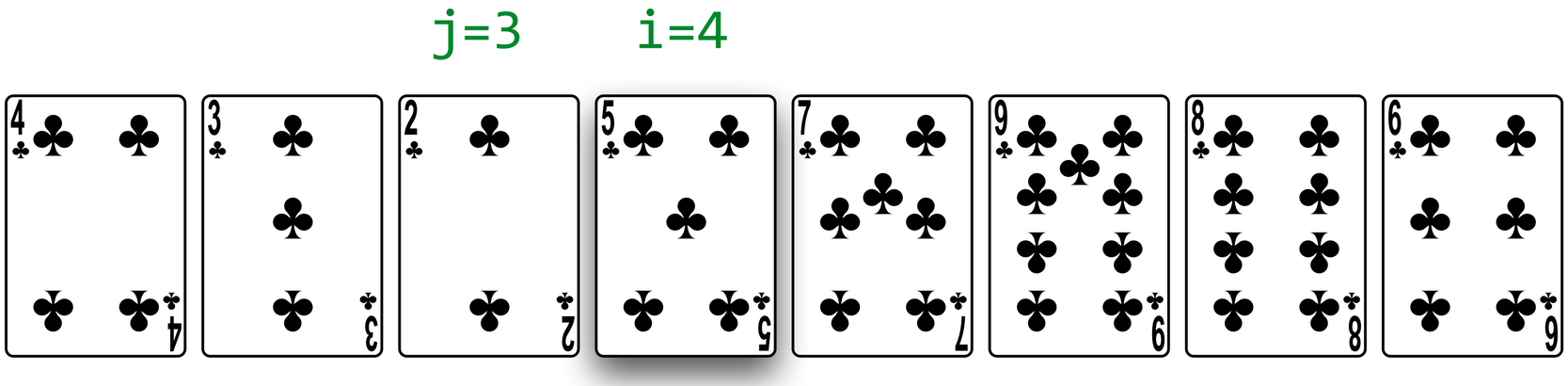
- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Partition en place



- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

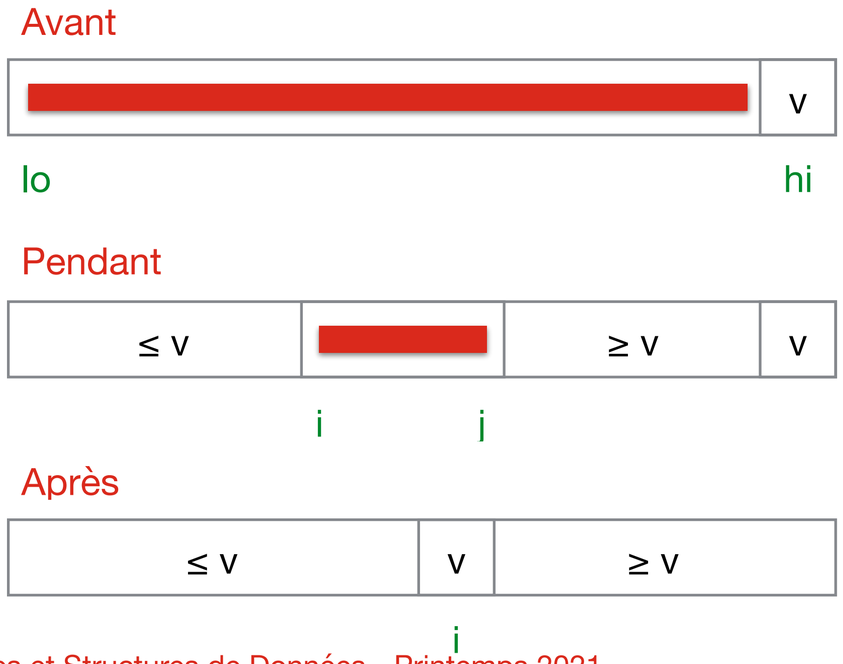
Partition en place



- Placer `p` en fin de tableau `A`
- Les indices `i` et `j` parcourent le tableau en sens inverses
- Échanger `A(i)` et `A(j)` pour maintenir `A(i) ≤ p` et `A(j) ≥ p`
- Jusqu'à ce que `i` et `j` se croisent.
- Placer `p` entre les deux partitions

Tri rapide: algorithme de partition

- Effectue la partition des éléments du tableau `A` de l'indice `lo` à l'indice `hi` en utilisant `A(hi)` comme pivot.
- Retourne la position du pivot dans le tableau partitionné.




```
fonction Partition(A,lo,hi)
  i ← lo-1, j ← hi

  boucler
    répéter incrémenter i
    tant que A(i) < A(hi)

    répéter décrémenter j
    tant que j>lo et A(hi) < A(j)

    si i ≥ j, alors sortir boucle
    fin si

    permuter A(i) et A(j)
  fin boucler

  permuter A(i) et A(hi)
  retourner i
```

Tri rapide: algorithme récursif

- Effectue le tri rapide des éléments du tableau A de l'indice lo à l'indice hi (compris).
- Choisit un pivot, le place en position hi, et appelle la fonction partition.
- S'appelle récursivement sur les partitions droite et gauche.

```
fonction TriRapide(A,lo,hi)
  si lo < hi, alors
    p ← choisir l'élément pivot
    permuter A(hi) et A(p)
    i ← Partition(A,lo,hi)

    TriRapide(A,lo,i-1)
    TriRapide(A,i+1,hi)
  fin si
```

Partition avec pivot I

E X E M P L E D E T R I i=0, j=12

Partition avec pivot I

```
E X E M P L E D E T R I        i=0, j=12
E X E M P L E D E T R I        ++i = 1
```

Partition avec pivot I

```
E X E M P L E D E T R I        i=0, j=12
E X E M P L E D E T R I        ++i = 1
E X E M P L E D E T R I        ++i = 2
```

Partition avec pivot I

```
E X E M P L E D E T R I        i=0, j=12
E X E M P L E D E T R I        ++i = 1
E X E M P L E D E T R I        ++i = 2
E X E M P L E D E T R I        --j = 11
```

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7
E E E D E L P M X T R I	echange(5,7)

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7
E E E D E L P M X T R I	echange(5,7)
E E E D E L P M X T R I	++i = 6

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7
E E E D E L P M X T R I	echange(5,7)
E E E D E L P M X T R I	++i = 6
E E E D E L P M X T R I	--j = 6

Partition avec pivot I

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7
E E E D E L P M X T R I	echange(5,7)
E E E D E L P M X T R I	++i = 6
E E E D E L P M X T R I	--j = 6
E E E D E L P M X T R I	--j = 5

Partition avec pivot I

E	X	E	M	P	L	E	D	E	T	R	I	i=0, j=12
E	X	E	M	P	L	E	D	E	T	R	I	++i = 1
E	X	E	M	P	L	E	D	E	T	R	I	++i = 2
E	X	E	M	P	L	E	D	E	T	R	I	--j = 11
E	X	E	M	P	L	E	D	E	T	R	I	--j = 10
E	X	E	M	P	L	E	D	E	T	R	I	--j = 9
E	E	E	M	P	L	E	D	X	T	R	I	echange(2,9)
E	E	E	M	P	L	E	D	X	T	R	I	++i = 3
E	E	E	M	P	L	E	D	X	T	R	I	++i = 4
E	E	E	M	P	L	E	D	X	T	R	I	--j = 8
E	E	E	D	P	L	E	M	X	T	R	I	echange(4,8)
E	E	E	D	P	L	E	M	X	T	R	I	++i = 5
E	E	E	D	P	L	E	M	X	T	R	I	--j = 7
E	E	E	D	E	L	P	M	X	T	R	I	echange(5,7)
E	E	E	D	E	L	P	M	X	T	R	I	++i = 6
E	E	E	D	E	L	P	M	X	T	R	I	--j = 6
E	E	E	D	E	L	P	M	X	T	R	I	--j = 5
E	E	E	D	E	I	P	M	X	T	R	L	echange(6,12)

Appels récursifs

lo i hi	1	2	3	4	5	6	7	8	9	10	11	12
	E	X	E	M	P	L	E	D	E	T	R	I

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X
10	10	11	D	E	E	E	E	I	L	M	P	R	T	X

Appels récursifs

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X
10	10	11	D	E	E	E	E	I	L	M	P	R	T	X
11		11	D	E	E	E	E	I	L	M	P	R	T	X

Appels récursifs

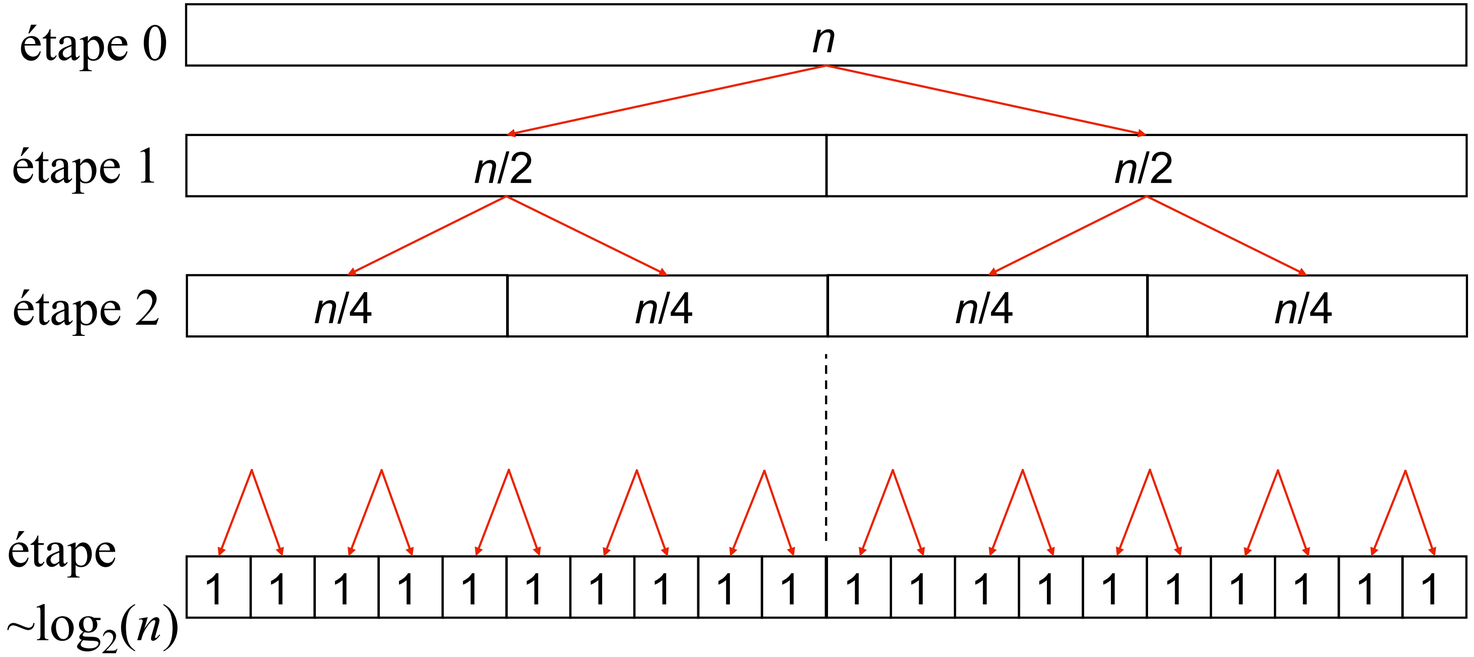
lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X
10	10	11	D	E	E	E	E	I	L	M	P	R	T	X
11		11	D	E	E	E	E	I	L	M	P	R	T	X

Appels récursifs

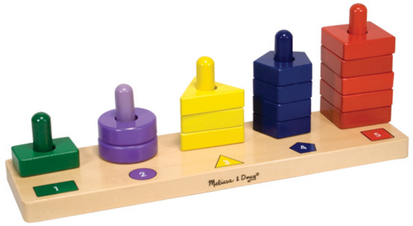
lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X
10	10	11	D	E	E	E	E	I	L	M	P	R	T	X
11		11	D	E	E	E	E	I	L	M	P	R	T	X
			D	E	E	E	E	I	L	M	P	R	T	X

Complexité : meilleur cas

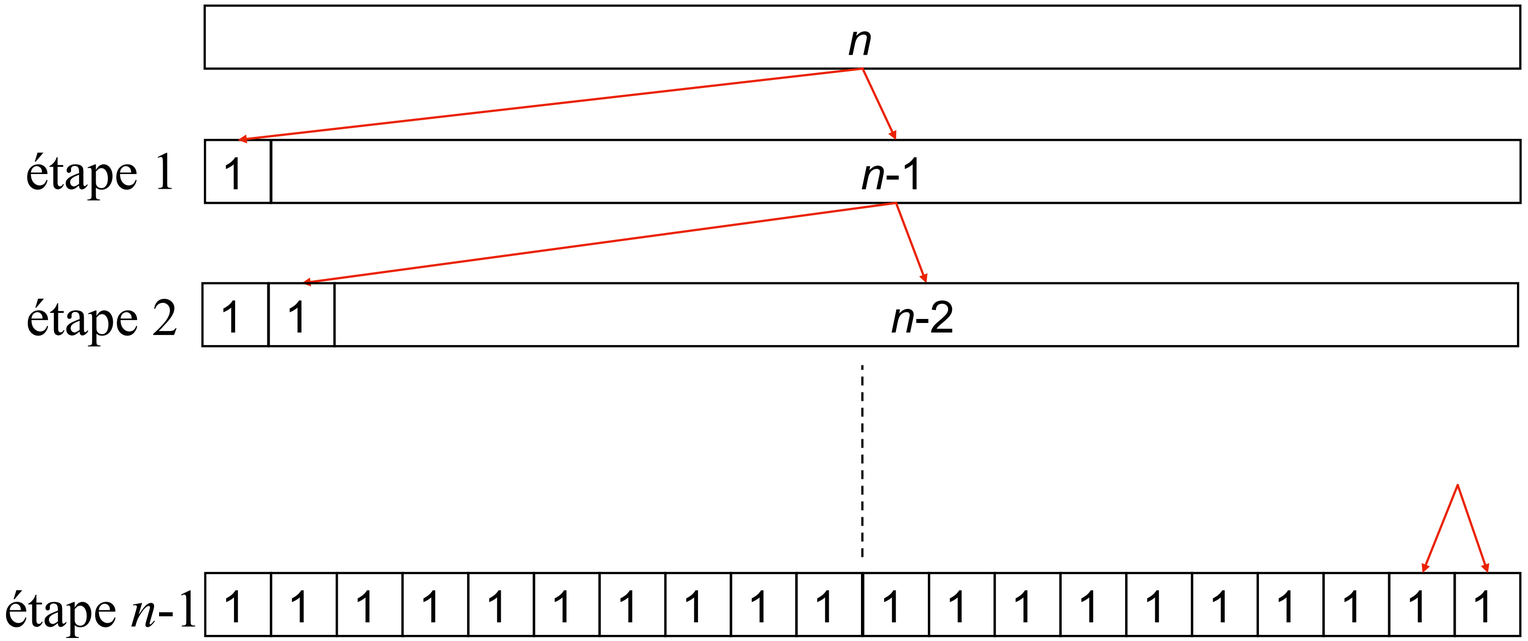
- Pivot = médiane des éléments à partitionner
- Partition du tableau en 2 parts égales
- Profondeur de récursion : $\log_2(n)$
- Complexité en $O(n.\log(n))$
- Comme pour le tri par fusion



Complexité dans le pire cas

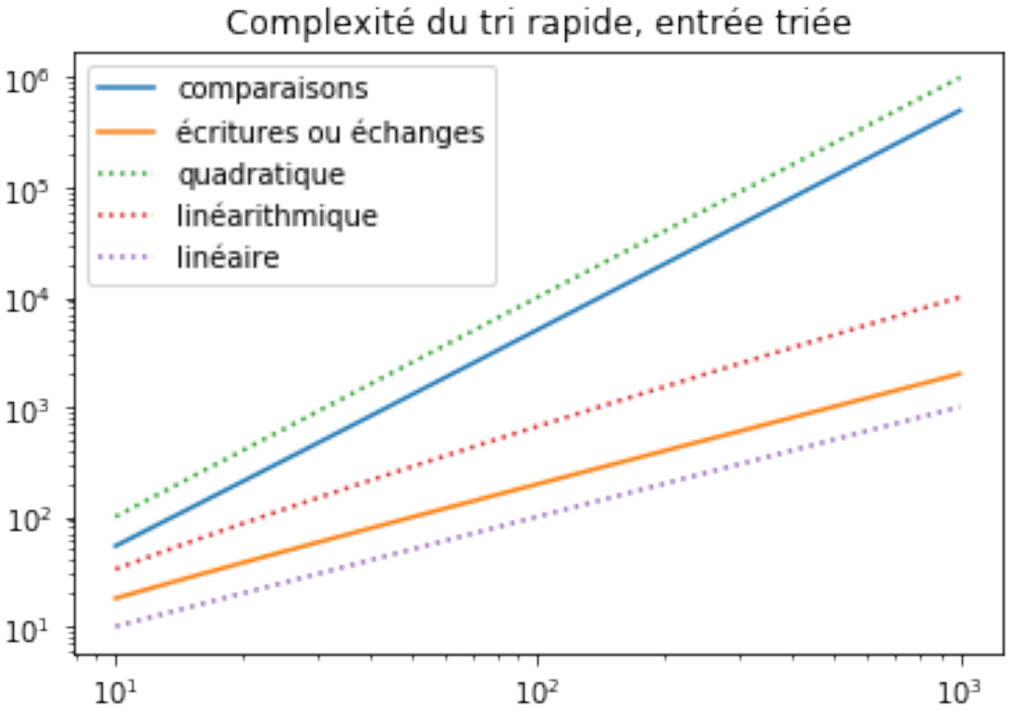


- Pivot = minimum ou maximum des éléments à partitionner
- Partition du tableau avec un côté vide et l'autre a n-1 éléments
- Profondeur de récursion : n-1
- Complexité en $O(n^2)$
- comme pour les tris



Le pire cas est-il courant ?

- Cela dépend du choix du pivot
- Si pivot = dernier élément, une entrée triée est un pire cas



N	Comp.	Ecr.
10	54	18
19	189	36
37	702	72
71	2555	140
138	9590	274
268	36045	534
517	133902	1032
1000	500499	1998

Comment choisir le pivot?

Comment choisir le pivot?

- A(hi)

Comment choisir le pivot?

- A(hi)
- Pire cas si A est trié

Comment choisir le pivot?

- A(hi)
- A(lo)

- Pire cas si A est trié

Comment choisir le pivot?

- A(hi)
- A(lo)

- Pire cas si A est trié
- Idem

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)

- Pire cas si A est trié
- Idem

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?
- Pas si mal…

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?
- Médiane (A(i) pour tous les i)

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?
- Pas si mal…

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?
- Médiane (A(i) pour tous les i)

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?
- Pas si mal…
- Trop cher, complexité O(n)

Comment choisir le pivot?

- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?
- Médiane (A(i) pour tous les i)
- Médiane (A(hi), A(lo), A(mid))

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?
- Pas si mal…
- Trop cher, complexité O(n)

Comment choisir le pivot?

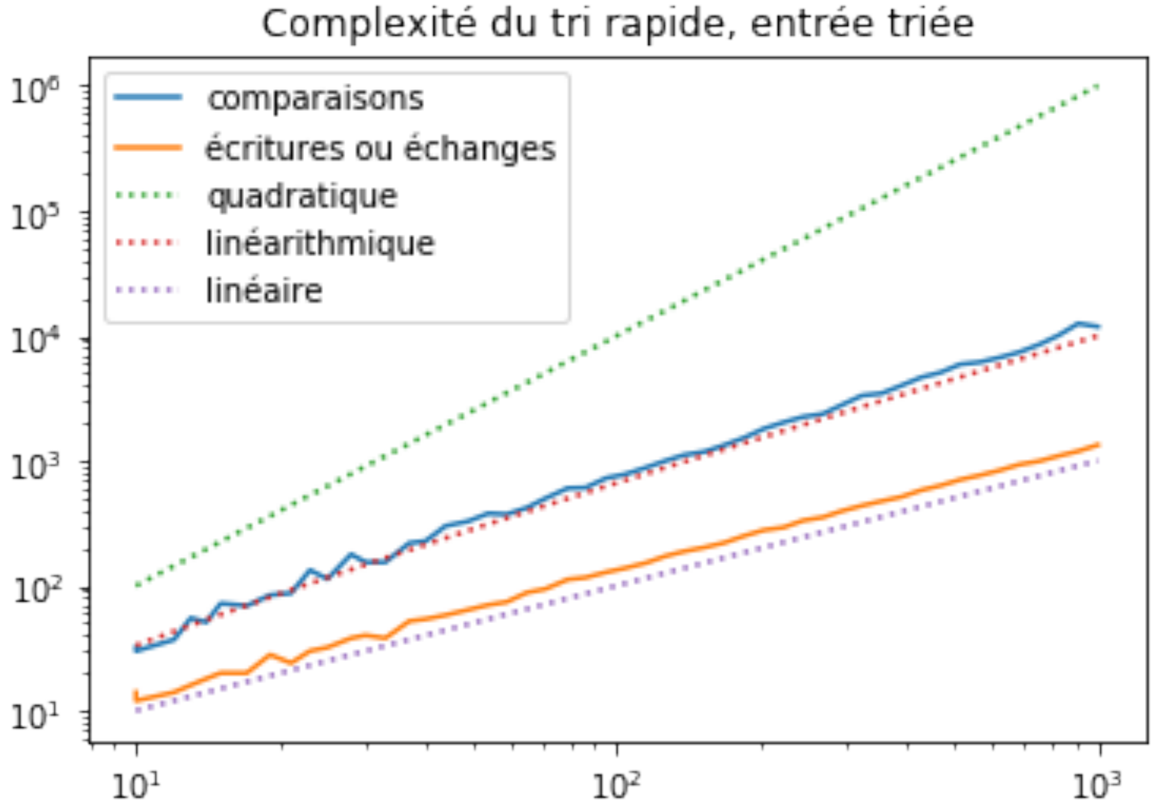
- A(hi)
- A(lo)
- A(mid = lo + (hi-lo)/2)
- A(random) ?
- Médiane (A(i) pour tous les i)
- Médiane (A(hi), A(lo), A(mid))

- Pire cas si A est trié
- Idem
- Idéal pour A trié, mais sinon ?
- Pas si mal…
- Trop cher, complexité O(n)
- Bonne idée

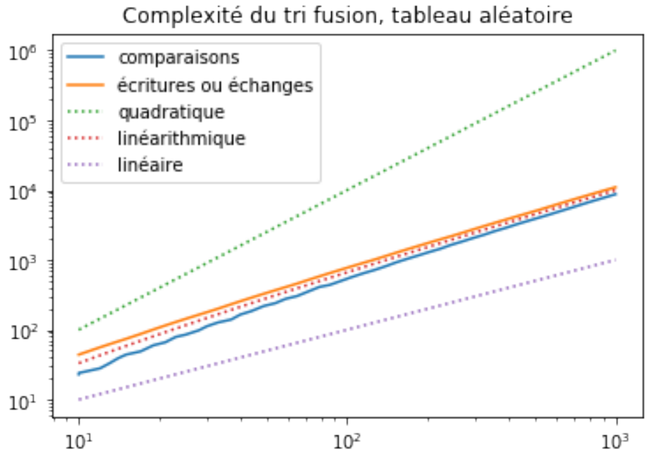
Comment choisir le pivot?

- A(hi) — Pire cas si A est trié
- A(lo) — Idem
- A(mid = lo + (hi-lo)/2) — Idéal pour A trié, mais sinon ?
- A(random) ? — Pas si mal…
- Médiane (A(i) pour tous les i) — Trop cher, complexité O(n)
- Médiane (A(hi), A(lo), A(mid)) — Bonne idée
- Médiane (Médiane(…), Médiane(…), Médiane(…))

Pivot aléatoire



N	Comp.	Ecr.
10	32	14
19	84	28
37	219	52
71	499	94
138	1110	188
268	2362	352
517	5895	704
1000	11811	1336



N	Comp.	Ecr.
10	23	44
19	60	101
37	140	232
71	348	511
138	802	1124
268	1830	2436
517	4013	5180
1000	8721	10976

- Complexité linéarithmique $\mathrm{O}(\mathrm{n.log(n)})$
- Plus de comparaisons, moins d'écritures que le tri fusion

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour $N>1$,

$$C_N = (N+1)+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{C_1 + C_{N-2}}{N}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour $N>1$,

partition

$$C_N = (N+1)+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{C_1 + C_{N-2}}{N}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour $N>1$,

$$C_N = \underbrace{(N+1)}_{\text{partition}}+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{\overbrace{C_1}^{\text{gauche}} + \overbrace{C_{N-2}}^{\text{droite}}}{N}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour $N>1$,

$$C_N = \underbrace{(N+1)}_{\text{partition}}+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{\overbrace{C_1}^{\text{gauche}} + \overbrace{C_{N-2}}^{\text{droite}}}{\underbrace{N}_{\text{probabilité}}}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour $N>1$,

$$C_N = \underbrace{(N+1)}_{\text{partition}}+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{\overbrace{C_1}^{\text{gauche}} + \overbrace{C_{N-2}}^{\text{droite}}}{\underbrace{N}_{\text{probabilité}}}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

à partir de cette relation, on peut calculer $NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$

Complexité moyenne, pivot aléatoire

Proposition - Le nombre moyen de comparaisons C_N pour trier rapidement un tableau de N éléments distincts est environ $2N \ln N$.

Preuve - C_N satisfait la récurrence $C_0=C_1=0$, et pour N>1,

$$C_N = \underbrace{(N+1)}_{\text{partition}}+\left(\frac{C_0 + C_{N-1}}{N}\right)+\left(\frac{\overbrace{C_1}^{\text{gauche}} + \overbrace{C_{N-2}}^{\text{droite}}}{\underbrace{N}_{\text{probabilité}}}\right)+\ldots+\left(\frac{C_{N-1} + C_0}{N}\right)$$

à partir de cette relation, on peut calculer $NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$

en divisant par N(N+1), on trouve $\dfrac{C_N}{N+1} = \dfrac{C_{N-1}}{N} + \dfrac{2}{N+1}$

Tri rapide: cas moyen (suite)

- En appliquant cette équation de manière répétée, on trouve

$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}
$$

Tri rapide: cas moyen (suite)

- En appliquant cette équation de manière répétée, on trouve

$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}
$$

- Ce qui donne, en approximant la somme par une intégrale,

$$
\begin{aligned}
C_N &= 2(N+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1}\right) \\
&\approx 2(N+1)\int_3^{N+1} \frac{1}{x}dx \\
&\approx 2(N+1)\ln N \\
&\approx 1.39 N \lg N
\end{aligned}
$$

Optimisation : tri hybride

- Pour des petits tableaux, le tri par insertion est plus rapide en évitant les appels récursifs. On a donc intérêt à arrêter la récursion avant d'atteindre des tableaux de 1 élément.

```
si hi ≤ lo, alors
  retourner
fin si
```

→

```
si hi ≤ lo + cuttof, alors
  retourner TriInsertion(A,lo,hi)
fin si
```

Optimisation : tri hybride

- Pour des petits tableaux, le tri par insertion est plus rapide en évitant les appels récursifs. On a donc intérêt à arrêter la récursion avant d'atteindre des tableaux de 1 élément.

```
si hi ≤ lo, alors
  retourner
fin si
```

→

```
si hi ≤ lo + cuttof, alors
  retourner TriInsertion(A,lo,hi)
fin si
```

- On peut aussi arrêter la récursion sans appeler `TriInsertion`, et puis appeler `TriInsertion` une seule fois à la fin.

Opt. : récursion terminale

- Avec un bon choix de pivot, le pire des cas est improbable, mais …
  - dans le pire cas, profondeur de récursion de n-1
  - risque de saturer la pile d'appels

```
fonction TriRapide(A,lo,hi)
  si lo < hi, alors
    p ← choisir l'élément pivot
    permuter A(hi) et A(p)
    i ← Partition(A,lo,hi)

    TriRapide(A,lo,i-1)
    TriRapide(A,i+1,hi)
  fin si
```

Opt. : récursion terminale

- Avec un bon choix de pivot, le pire des cas est improbable, mais …
  - dans le pire cas, profondeur de récursion de $n-1$
  - risque de saturer la pile d'appels
- Le deuxième appel récursif est en récursion terminale
  - On peut le remplacer par une itération
  - Pour éviter une récursion trop profonde dans le pire cas,
    - l'appel récursif restant traite la plus petite partition
    - l'itération traite la plus grande partition

```
fonction TriRapide(A,lo,hi)
  si lo < hi, alors
    p ← choisir l'élément pivot
    permuter A(hi) et A(p)
    i ← Partition(A,lo,hi)

    TriRapide(A,lo,i-1)
    TriRapide(A,i+1,hi)
  fin si
```

Semi-récursif

```
fonction TriRapide(A,lo,hi)
  si lo < hi, alors
    p ← choisir l'élément pivot
    permuter A(hi) et A(p)
    i ← Partition(A,lo,hi)

    TriRapide(A,lo,i-1)
    TriRapide(A,i+1,hi)
  fin si
```

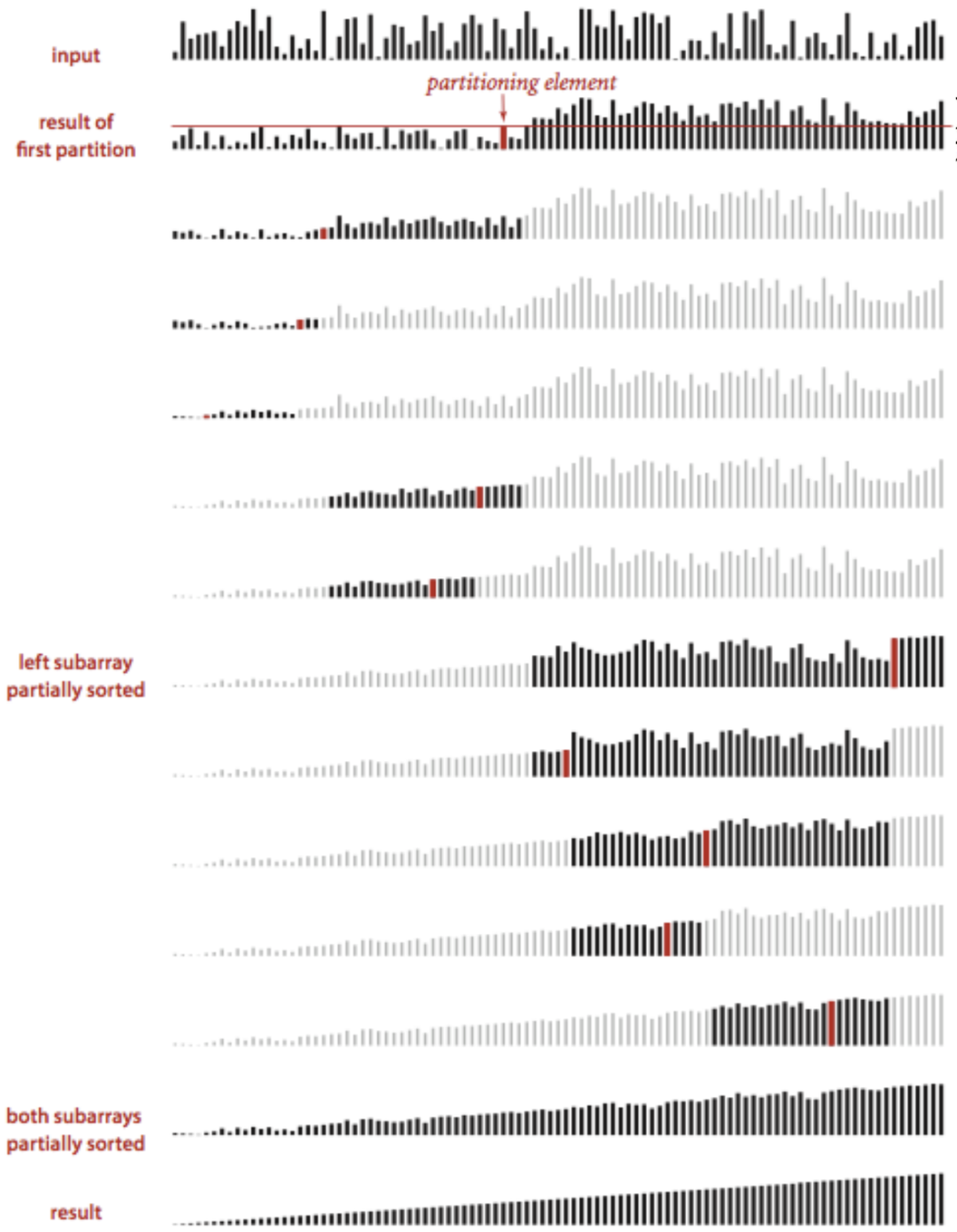
```
fonction TriRapide(A,lo,hi)
  tant que lo < hi
    p ← choisir l'élément pivot
    permuter A(hi) et A(p)
    i ← Partition(A,lo,hi)

    si i-lo < hi-i, alors
      TriRapide(A,lo,i-1)
      lo ← i+1
    sinon,
      TriRapide(A,i+1,hi)
      hi ← i-1
    fin si
  fin tant que
```

Visualisation

- Pivot = médiane de 3 éléments
- Pas de partition si

```
hi - lo < cutoff
```

- Le résultat partiellement trié sera par insertion



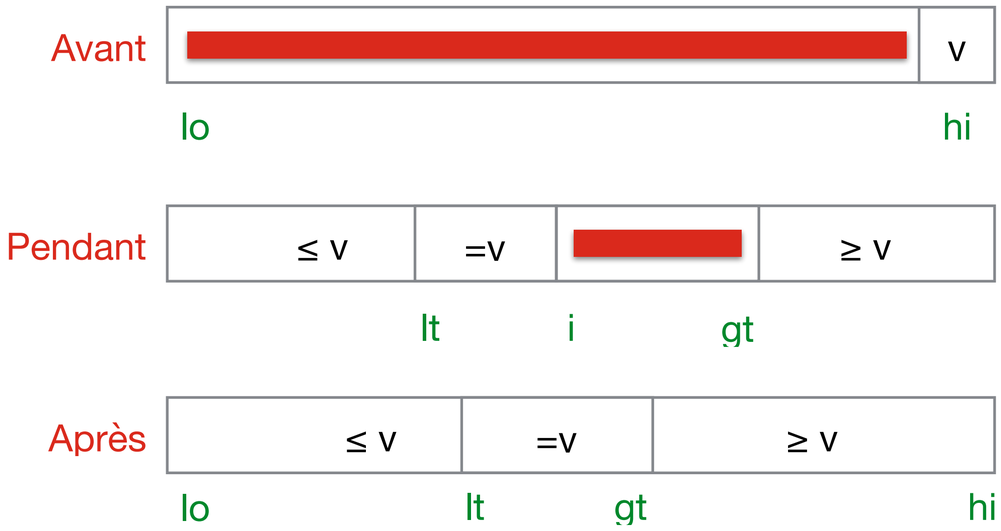
Propriétés

- Complexité au pire : $O(n^2)$
  - Extrêmement rare avec un bon choix du pivot
- Complexité en moyenne : $O(n \log(n))$
- Temps de calcul en pratique : plus rapide que le tri fusion malgré $1{,}39$ fois plus de comparaisons, car moins d'écritures
- Mémoire : s'effectue en place
- Stabilité : pas stable car la partition n'est pas stable
- Utilisation : Java sort (types primitifs), C qsort, C++ std::sort, …

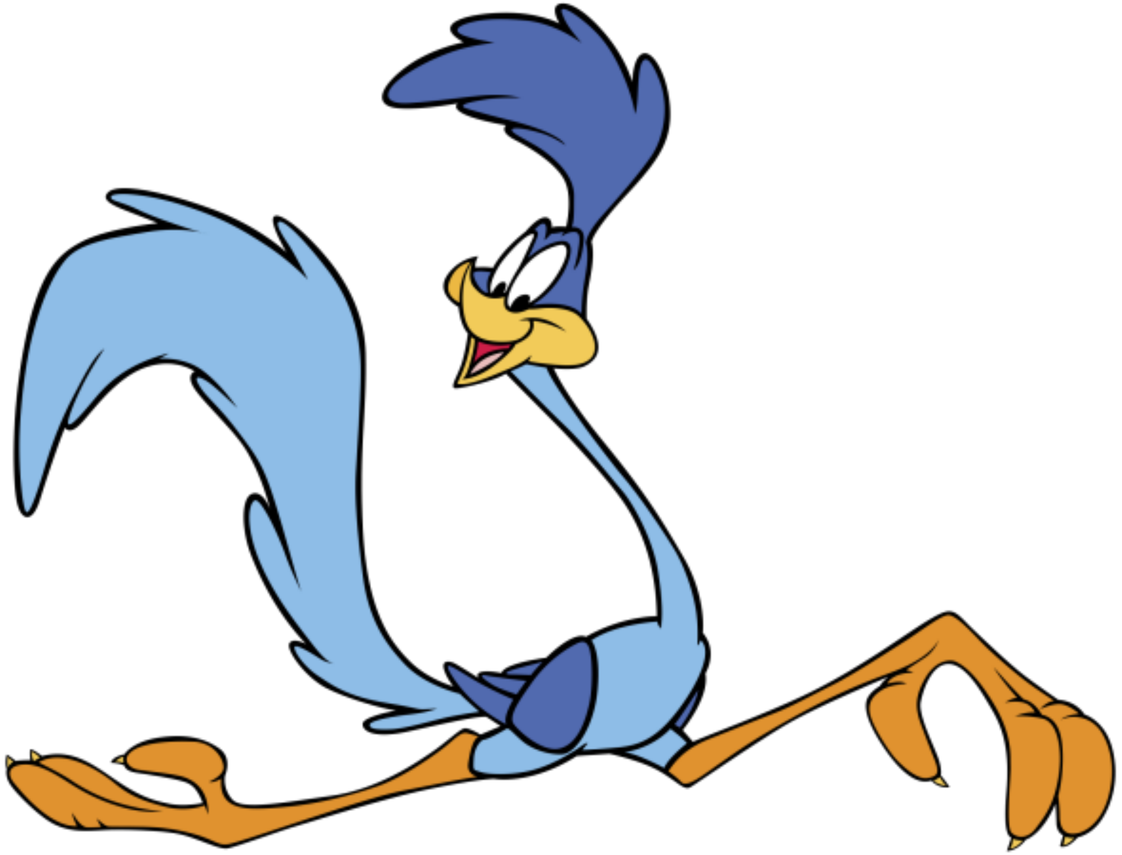
Eléments répétitifs

- Le tri rapide mal mis en oeuvre est sensible à la présence d'éléments répétés dans le tableau.
- Si on utilise ≤ plutôt que < dans la partition, l'algorithme devient $O(n^2)$ en présence d'éléments répétés.
- la plupart des mises en oeuvre de la fonction `qsort()` de C avaient ce défaut jusqu'à sa découverte en 1991.
- On peut le résoudre plus proprement en partitionnant en 3

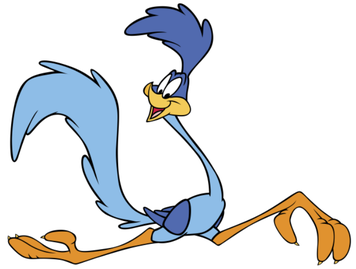
```
répéter incrémenter i
tant que A(i) < A(hi)

répéter décrémenter j
tant que j>lo et A(hi) < A(j)
```



3.4. Sélection rapide



QuickSelect



But - trouver le $k^{\text{ième}}$ plus petit élément dans un tableau de N éléments

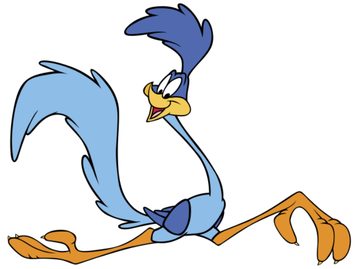
Exemples - min ($k=1$), max ($k=N$), médiane ($k=N/2$)

Quelle complexité ?

- au plus $O(N \log_2 N)$ en triant le tableau
- au plus $O(N)$ pour les cas simples comme $k=1$, $k=N$, $\ldots$
- au moins $O(N)$, il faut bien observer une fois chaque élément

Le problème général est-il aussi difficile que le tri ou aussi simple que la recherche du min ou du max?

Algorithme

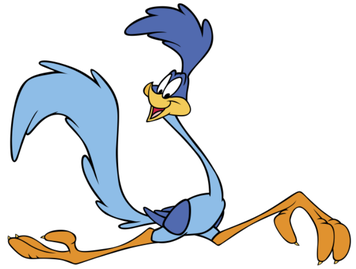


- Effectue la sélection rapide du $k^{\text{ième}}$ plus petit élément du tableau `A` de `n` éléments
- Utilise la même fonction de partition que le tri rapide
- Ne continue que du côté pertinent, déterminé en comparant `i` et `k`

- On ignore ici le problème du choix du pivot.

```
fonction SelectionRapide(A,n,k)
  lo ← 1
  hi ← n

  tant que hi > lo
   i ← partition(A,lo,hi)
   si i < k, alors
    lo ← i+1
   sinon, si i > k, alors
    hi ← i-1
   sinon  (i = k)
    retourner A(k)
  fin tant que

  retourner A(k)
```

Complexité



Proposition - QuickSelect a une complexité moyenne linéaire $\mathrm{O}(\mathrm{n})$

Preuve -

- Intuitivement, chaque partition divise le tableau en deux. Le nombre de comparaisons est donc

$$N + N/2 + N/4 + \ldots + 1 \sim 2N$$

- Plus formellement, il faut faire une analyse similaire à QuickSort

Par contre, la complexité dans le pire cas est $\mathrm{O}(\mathrm{n}^2)$. Mais elle est improbable pour n grand et avec un choix aléatoire de pivot.

3.5 Tri optimal

Tri optimal

Peut-on faire sensiblement mieux que MergeSort et QuickSort ?

Proposition - Un algorithme de tri …

basé sur des comparaisons

a besoin de $\mathrm{O}(\mathrm{n}\ \log_2(\mathrm{n}))$ comparaisons

Pour trier n éléments

Preuve

Preuve

- Il y a n! ordres possibles pour n éléments

Preuve

a b c
a c b
b a c
b c a
c a b
c b a

- Il y a n! ordres possibles pour n éléments

Preuve

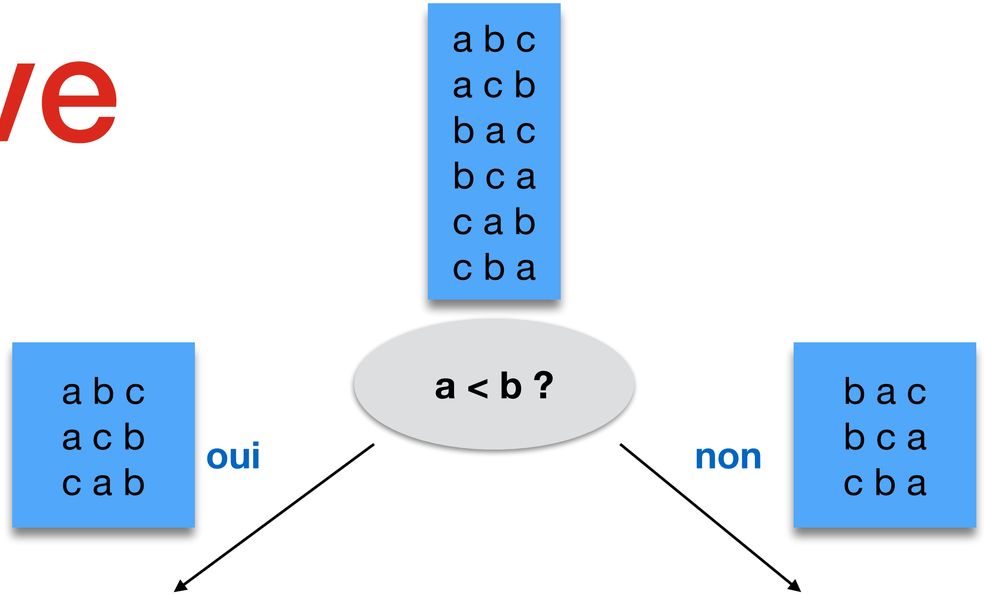
a b c
a c b
b a c
b c a
c a b
c b a

- Il y a n! ordres possibles pour n éléments

- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales

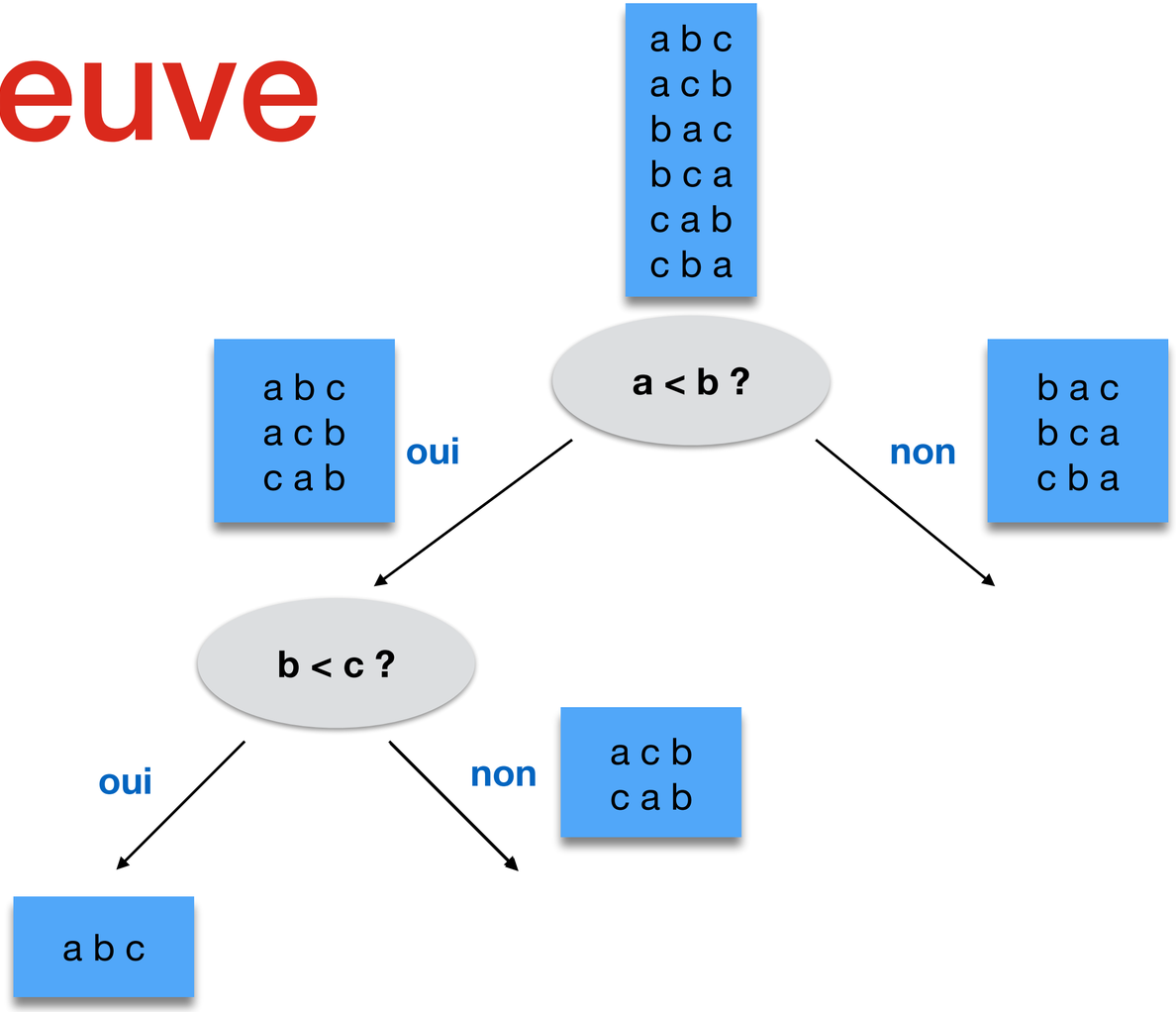
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales



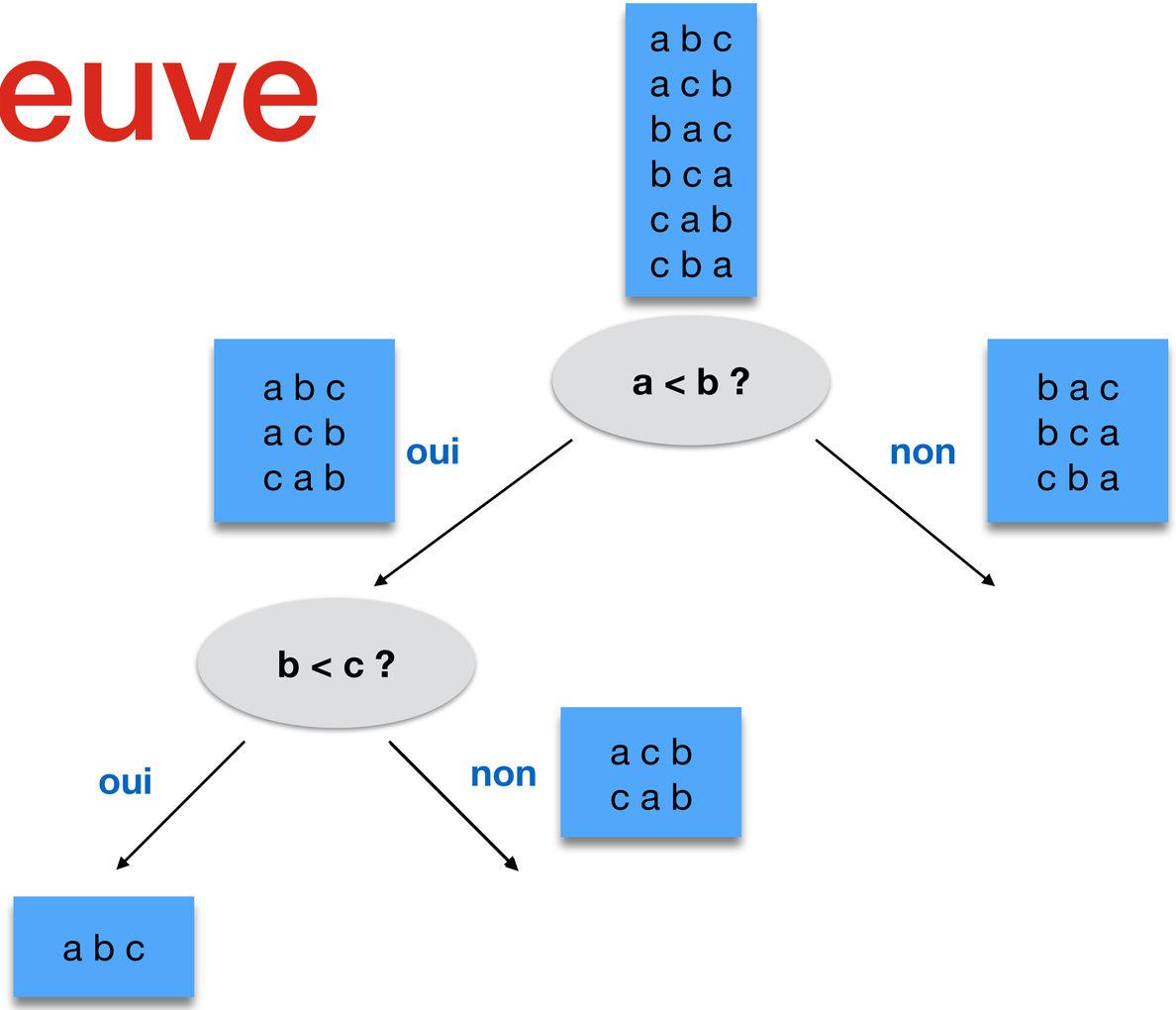
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales



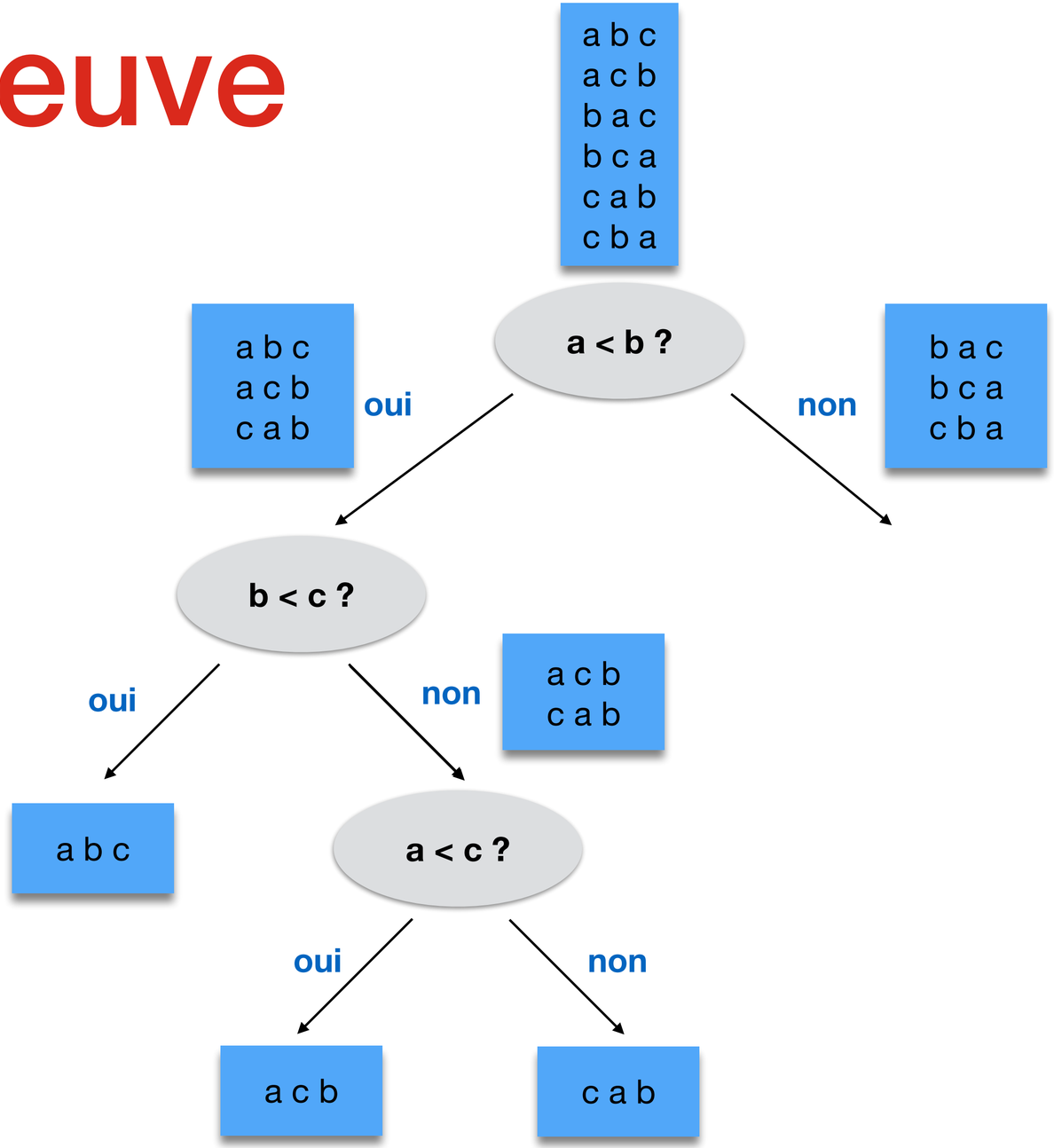
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales
- Trier, c'est comparer jusqu'à n'avoir une permutation dans chaque feuille de l'arbre de décision



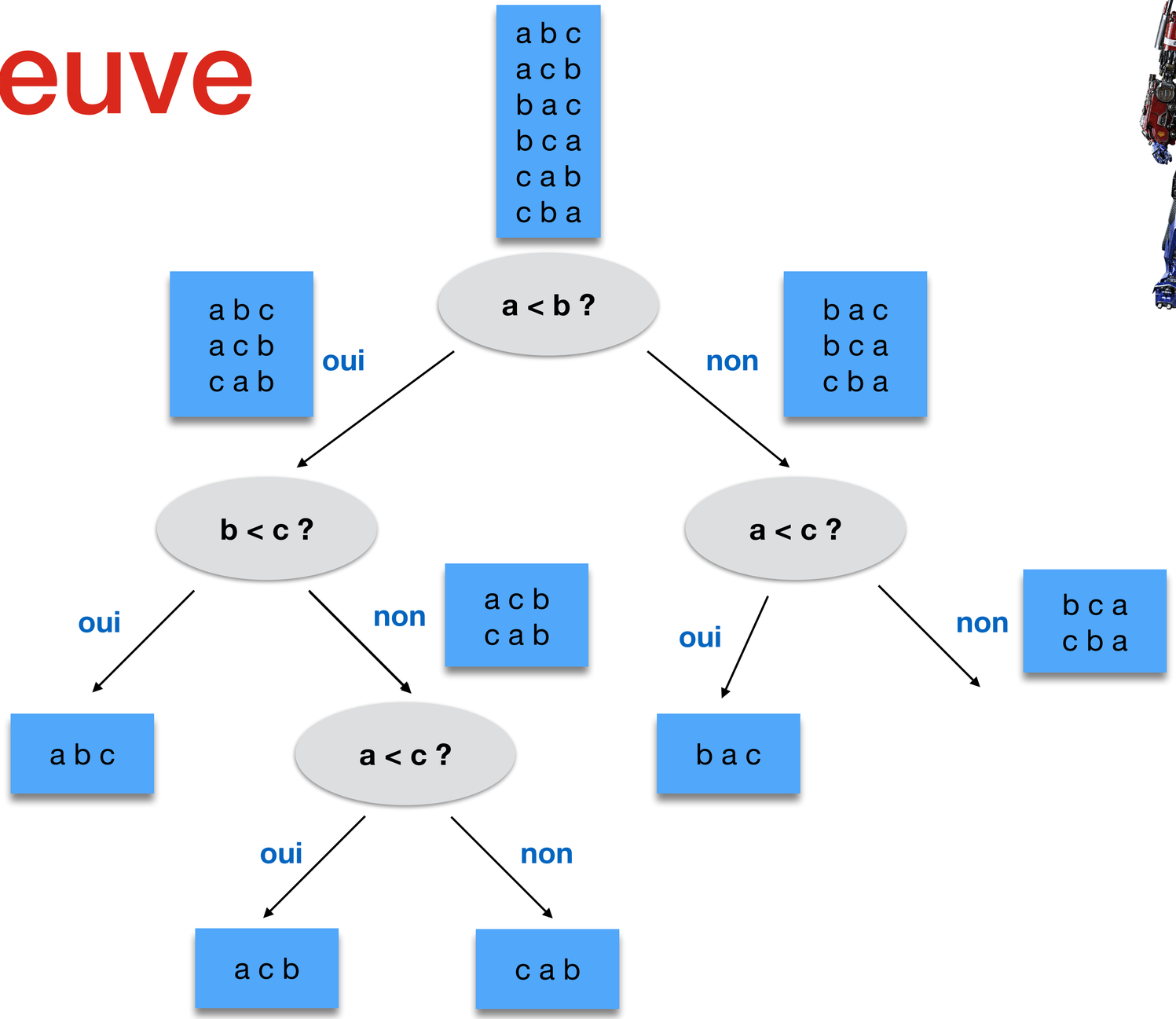
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales
- Trier, c'est comparer jusqu'à n'avoir une permutation dans chaque feuille de l'arbre de décision



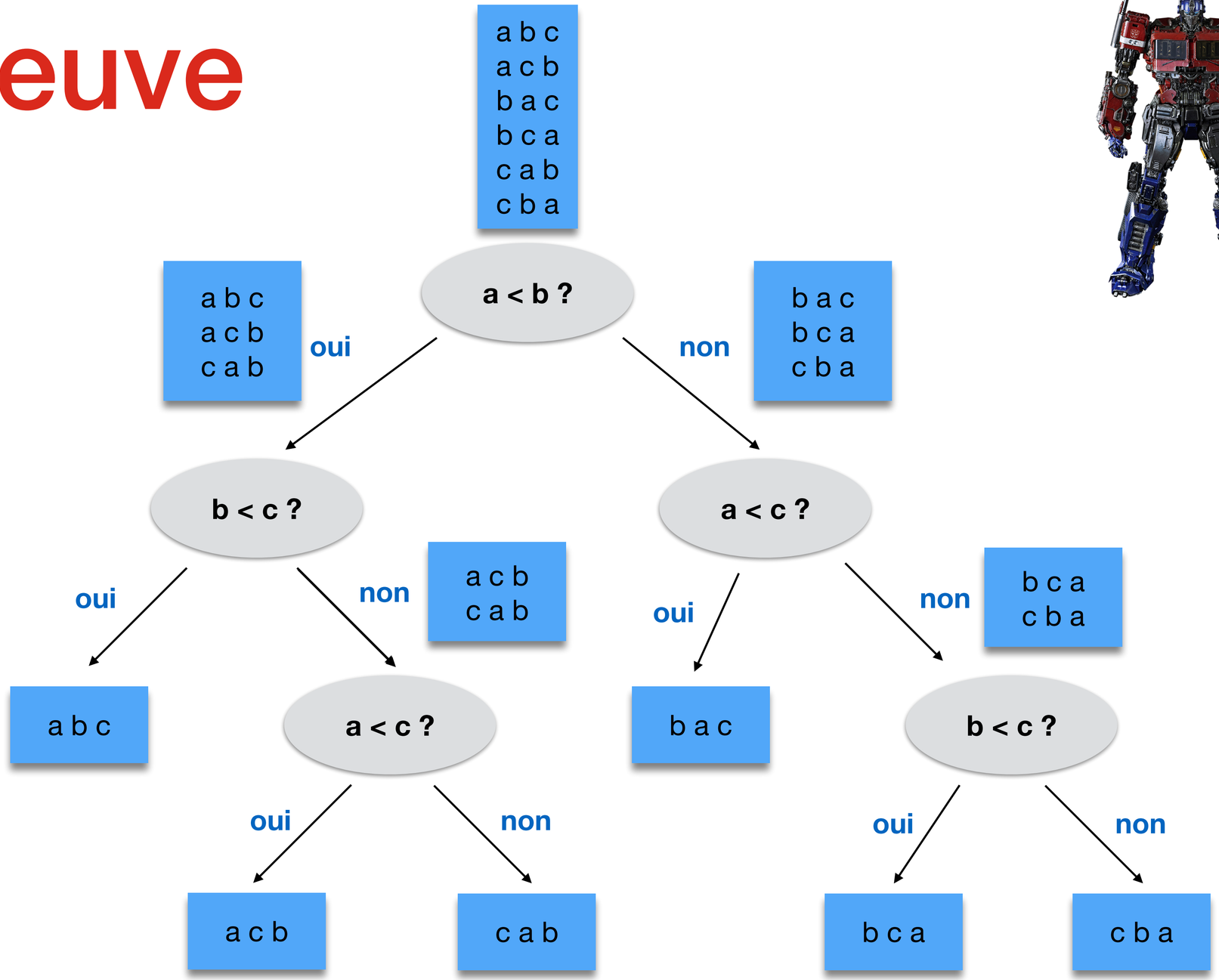
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales
- Trier, c'est comparer jusqu'à n'avoir une permutation dans chaque feuille de l'arbre de décision



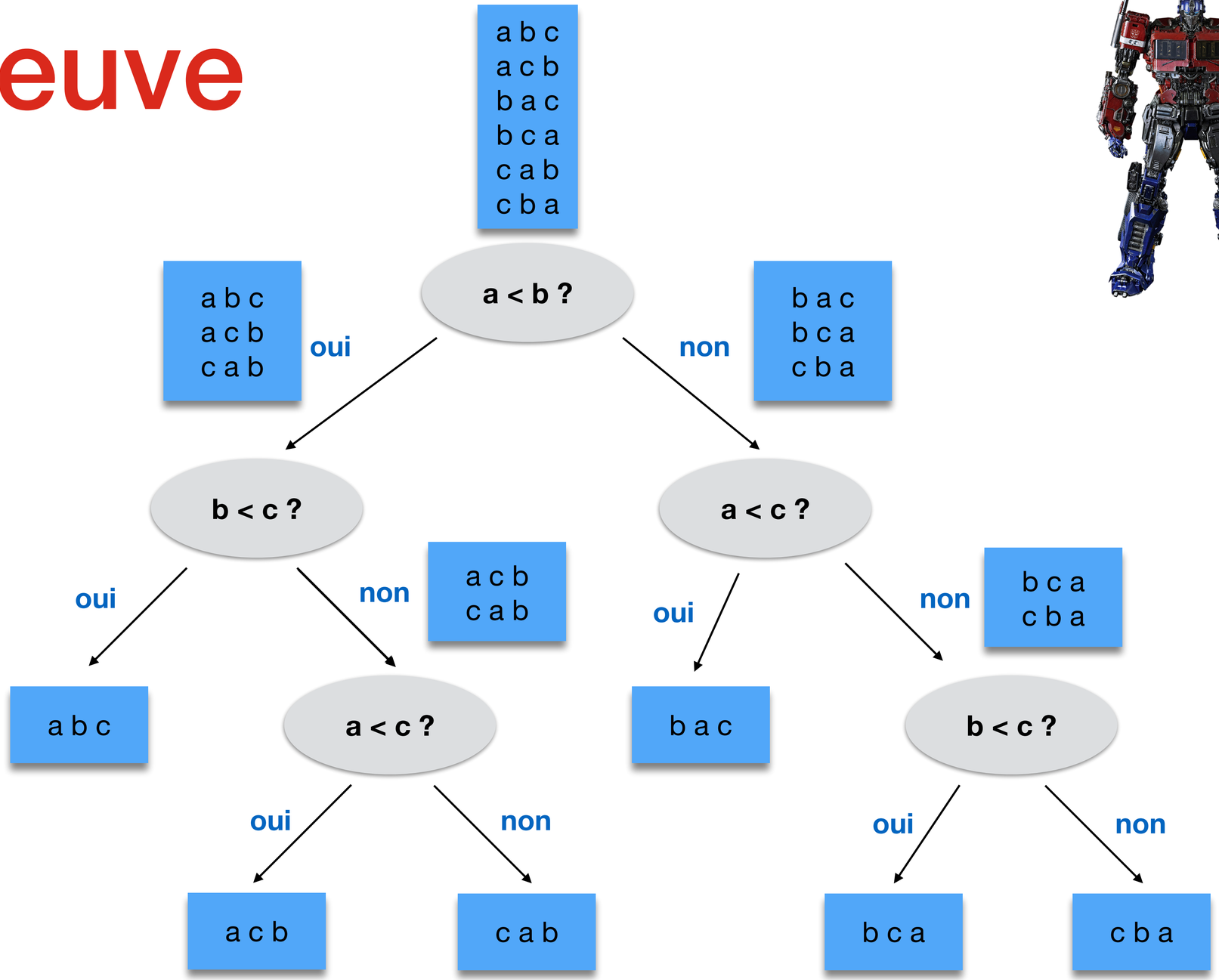
Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales
- Trier, c'est comparer jusqu'à n'avoir une permutation dans chaque feuille de l'arbre de décision



Preuve

- Il y a n! ordres possibles pour n éléments
- 1 comparaison permet de séparer l'ensemble des permutations en 2 parties, au mieux en 2 parties égales
- Trier, c'est comparer jusqu'à n'avoir une permutation dans chaque feuille de l'arbre de décision
- 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

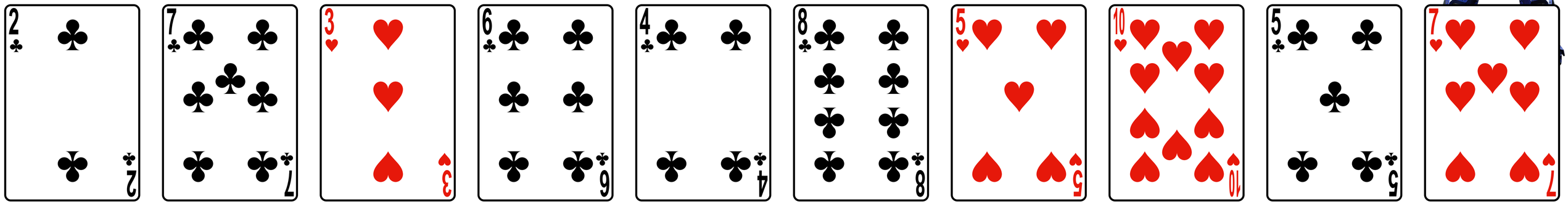
Mieux qu’optimal ?

Mais on peut quand même parfois faire mieux que l’optimal…

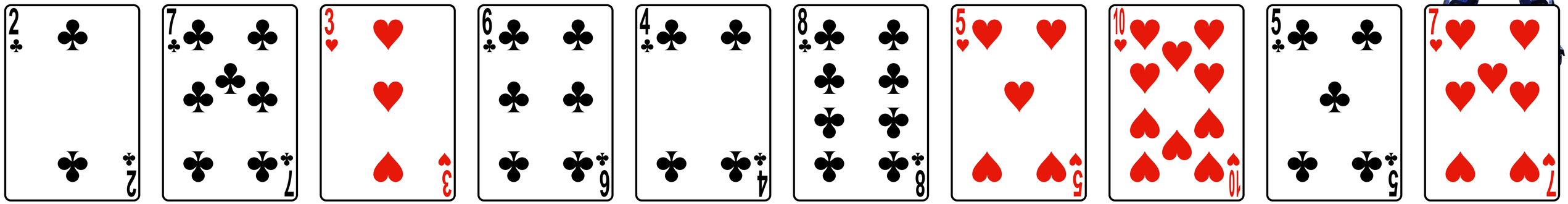
- Ne pas utiliser de comparaisons
- Utiliser des propriétés connues des données

Comment trier vous un jeu de carte par couleur ?

Trier sans comparer

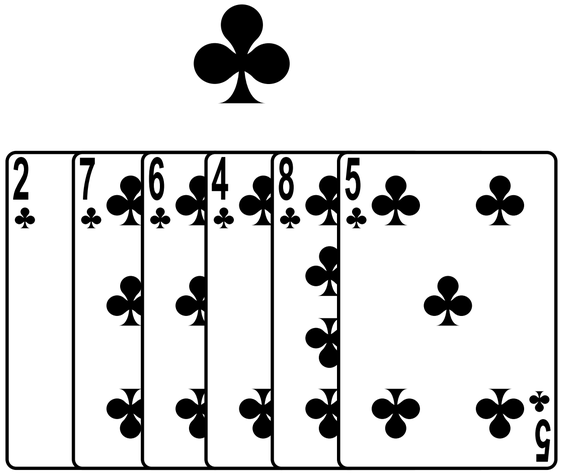


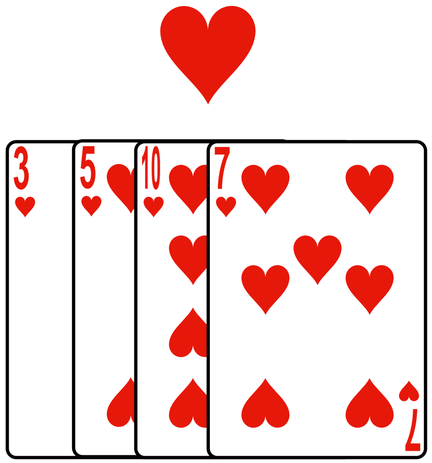
Trier sans comparer



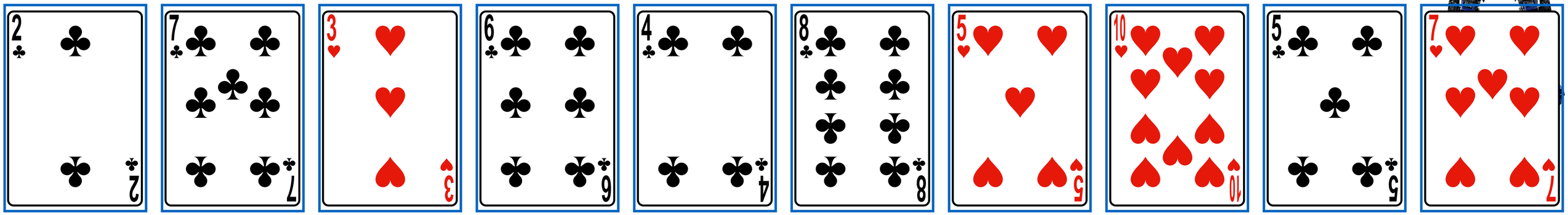
♣ ♥

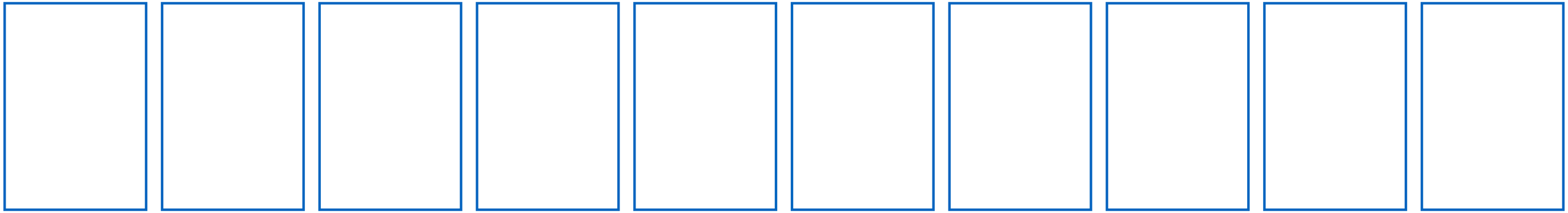
Trier sans comparer



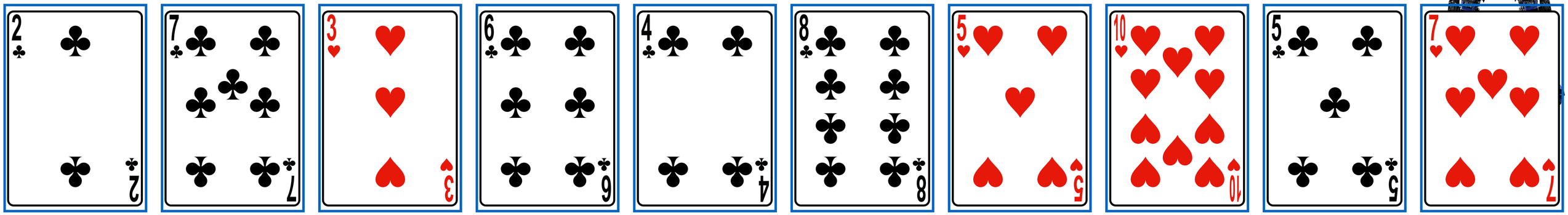


Tri comptage



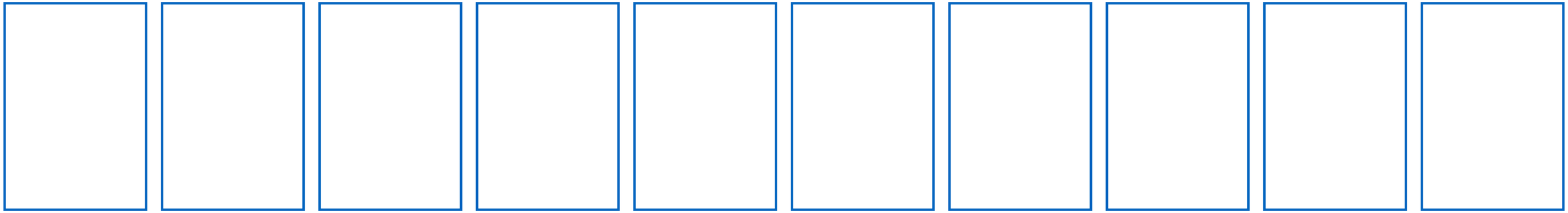


Tri comptage

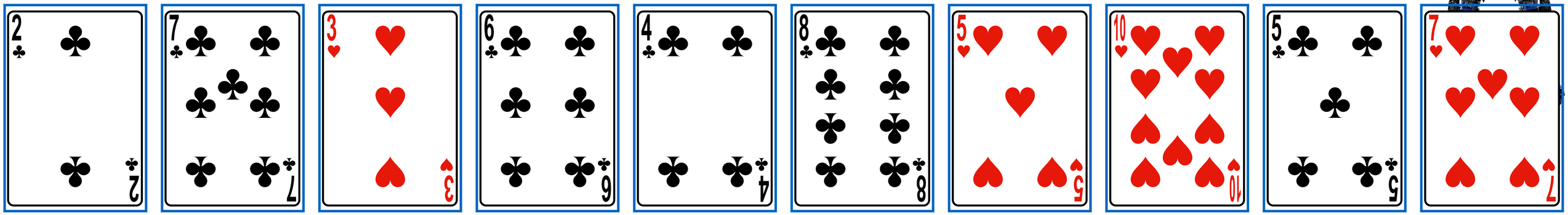


Compteurs:

♣	6
♥	4

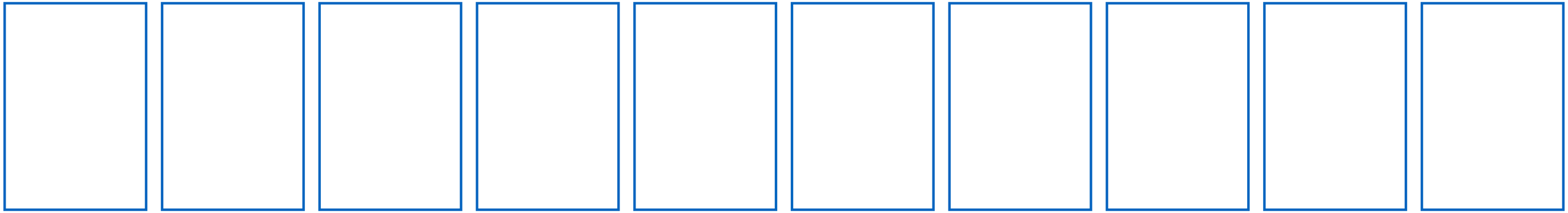


Tri comptage

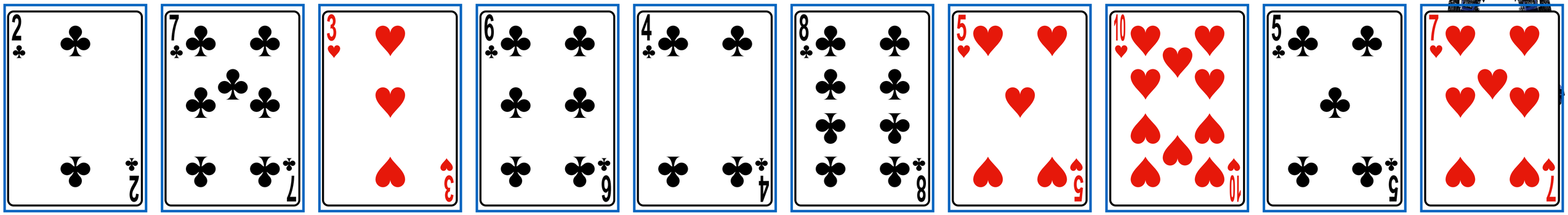


Compteurs:

♣	6
♥	4

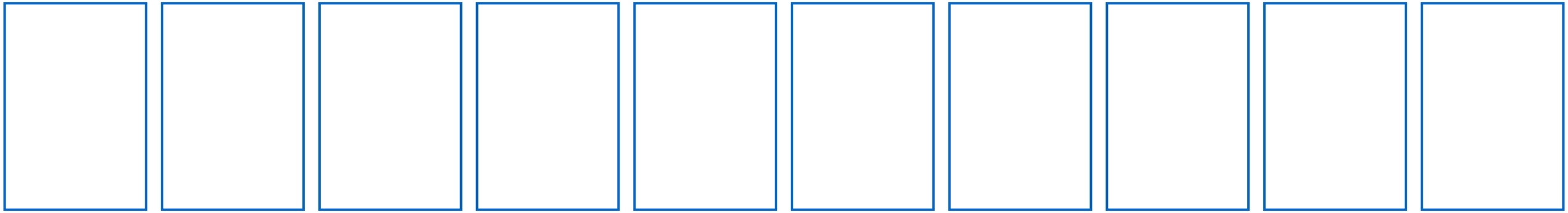


Tri comptage

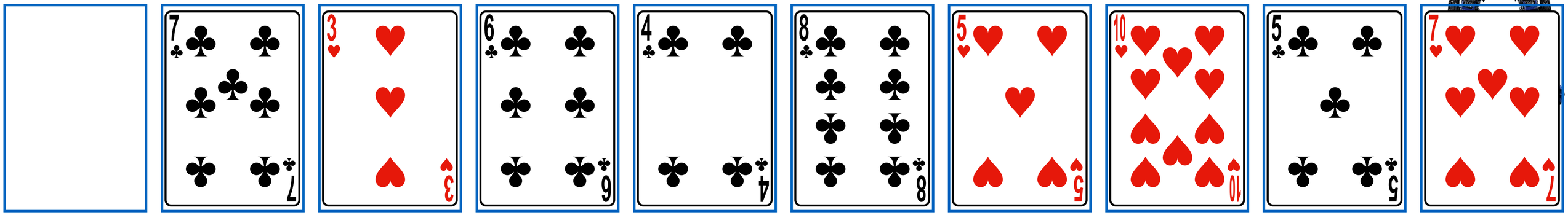


Compteurs:

♣	6
♥	4

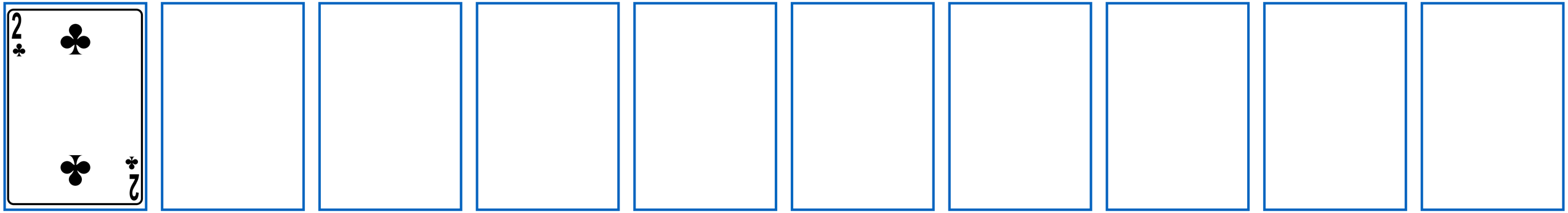


Tri comptage

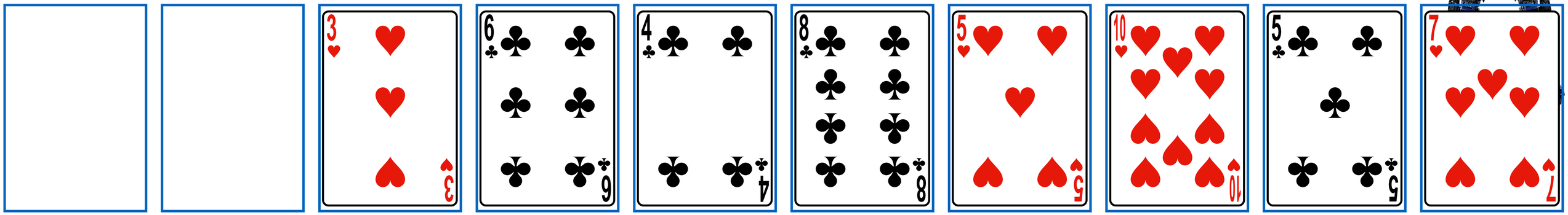


Compteurs:

♣	6
♥	4

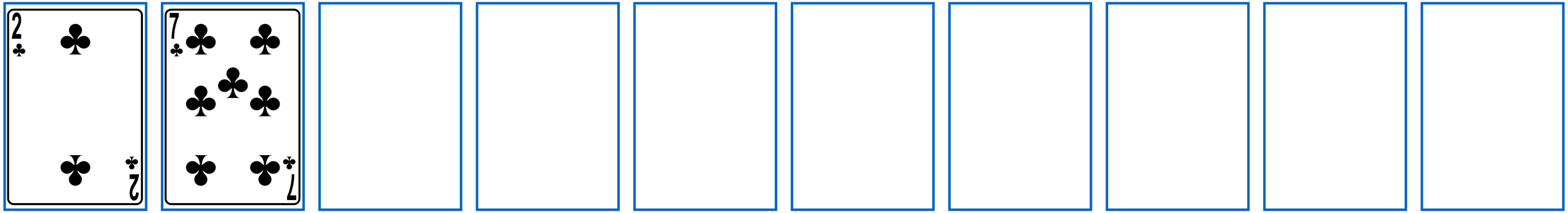


Tri comptage

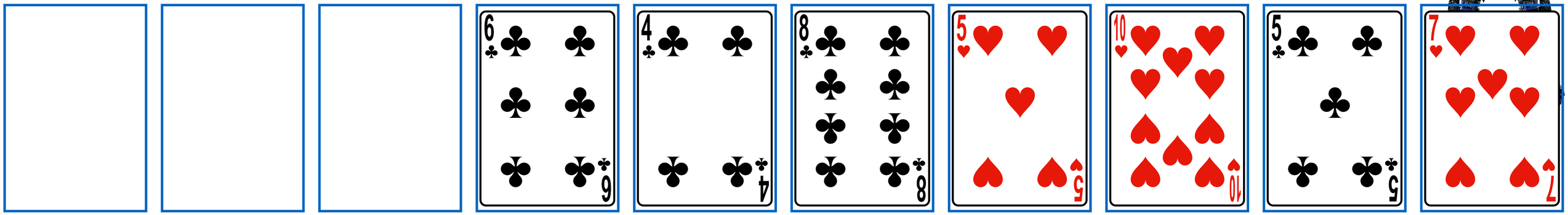


Compteurs:

♣	6
♥	4

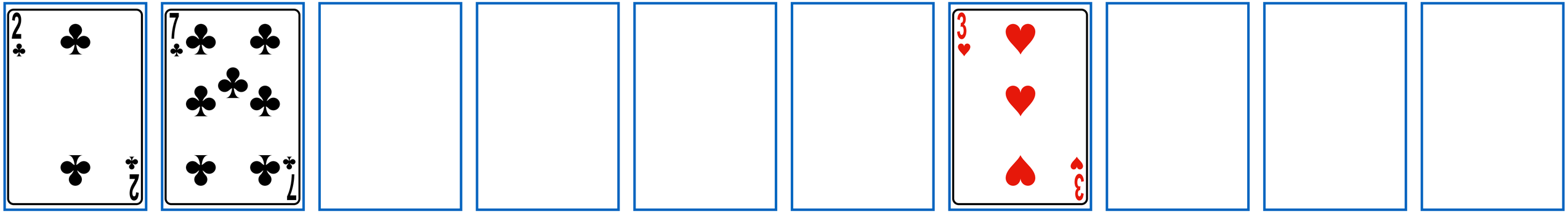


Tri comptage

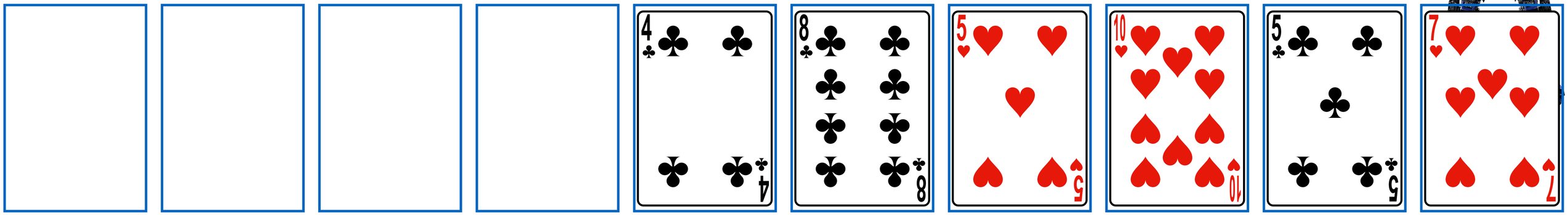


Compteurs:

♣	6
♥	4

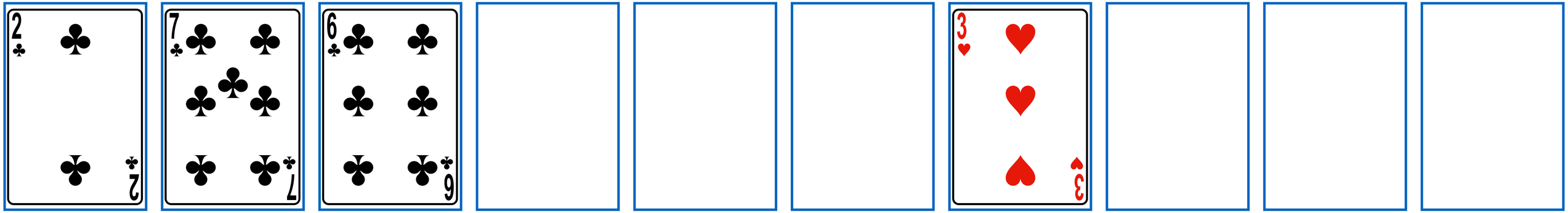


Tri comptage

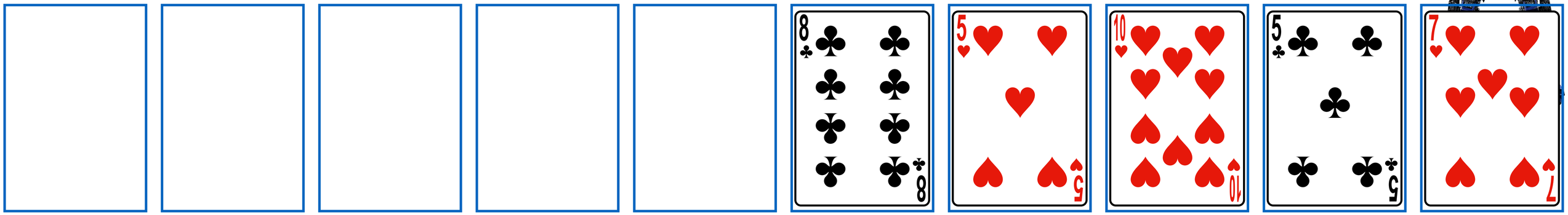


Compteurs:

♣	6
♥	4

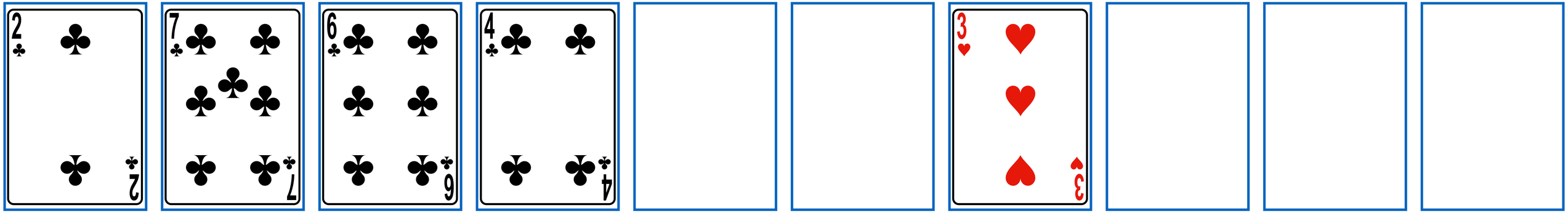


Tri comptage

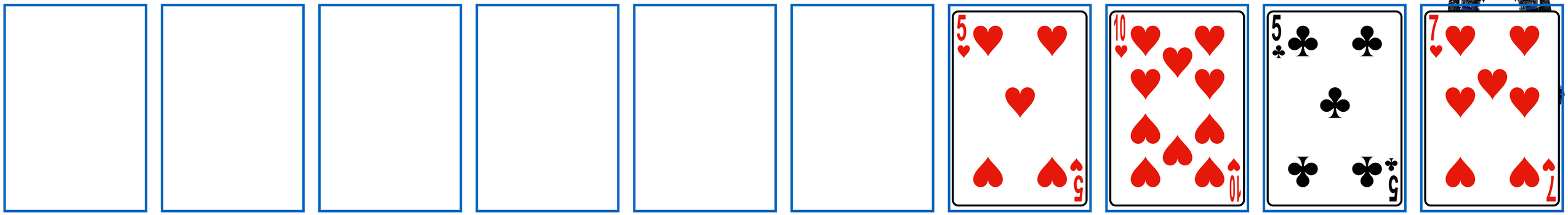


Compteurs:

♣	6
♥	4

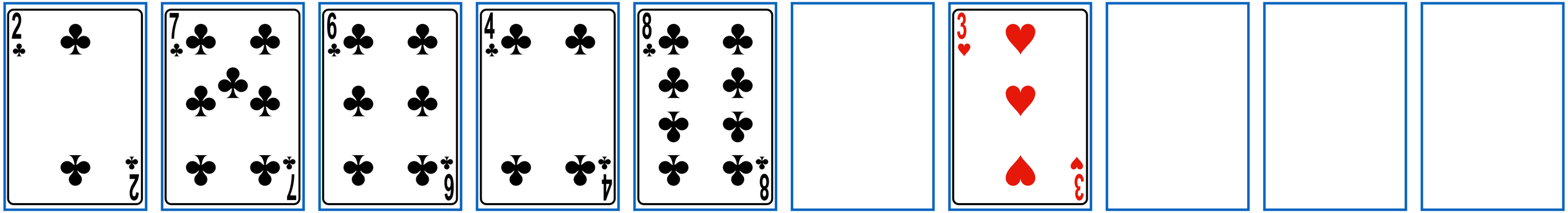


Tri comptage



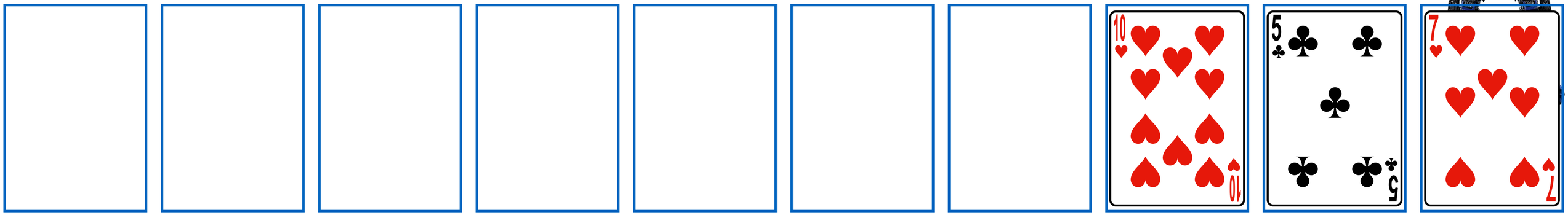
Compteurs:

♣	6
♥	4



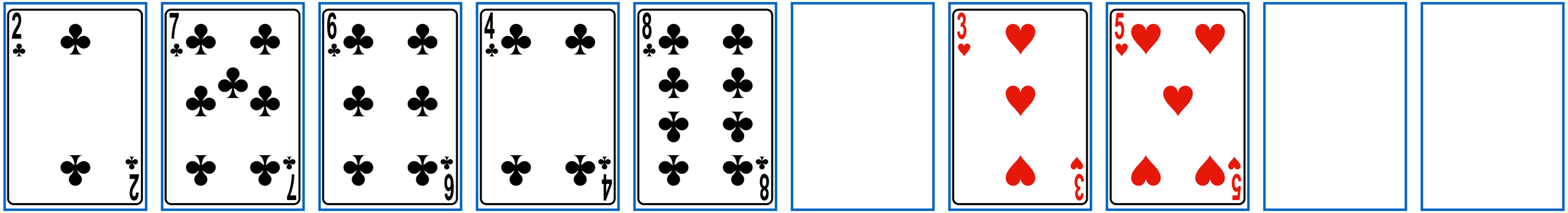
♣ ♥

Tri comptage

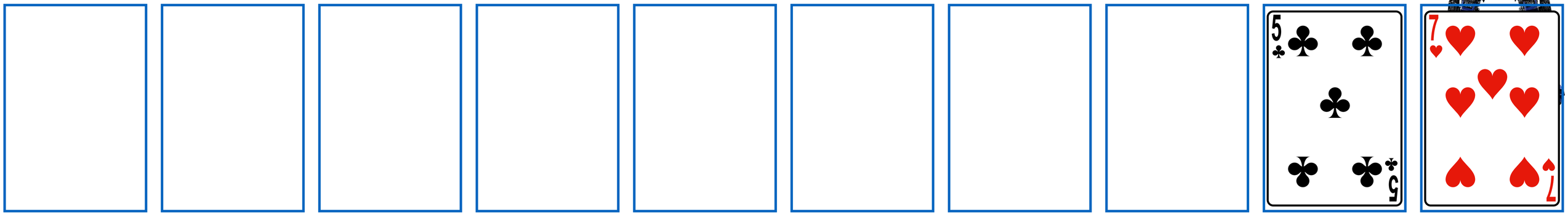


Compteurs:

♣	6
♥	4

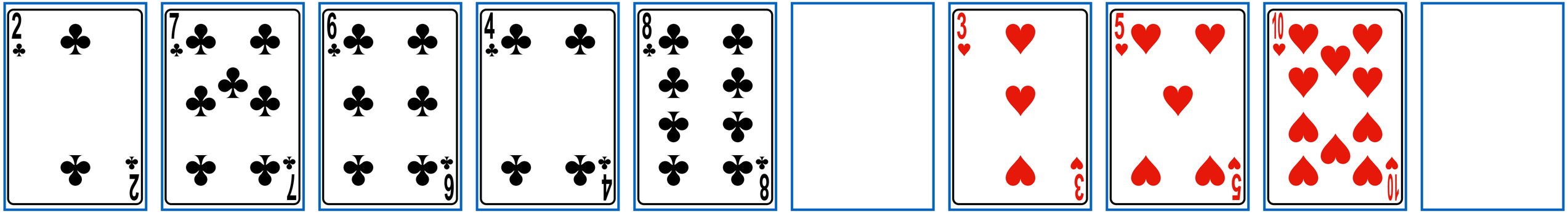


Tri comptage

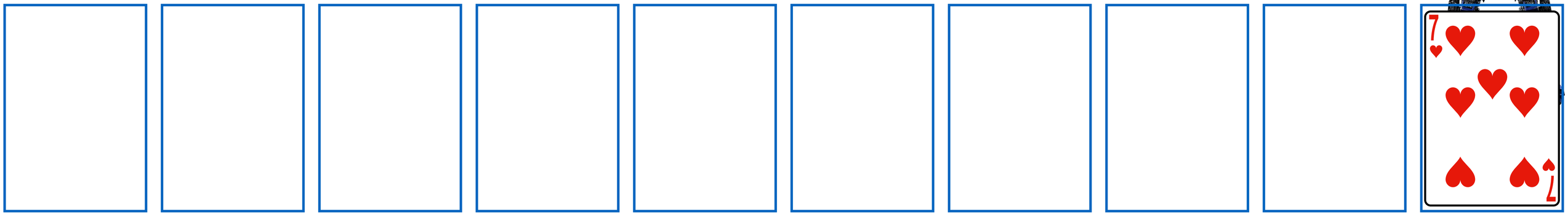


Compteurs:

♣	6
♥	4

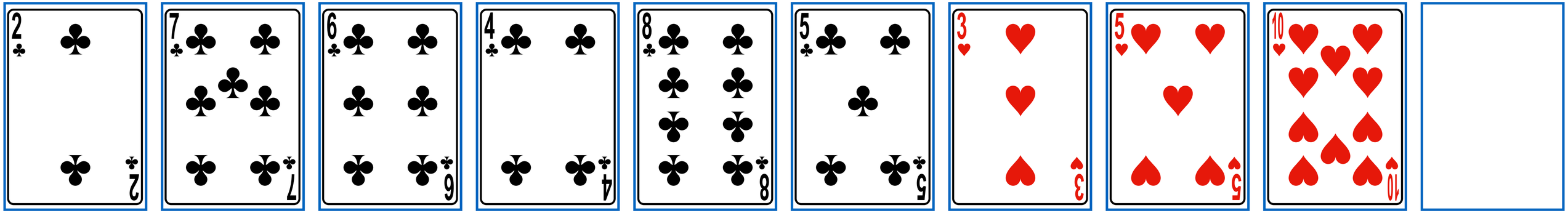


Tri comptage

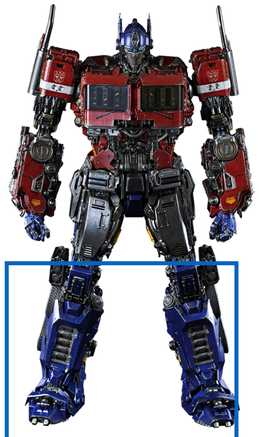


Compteurs:

♣	6
♥	4

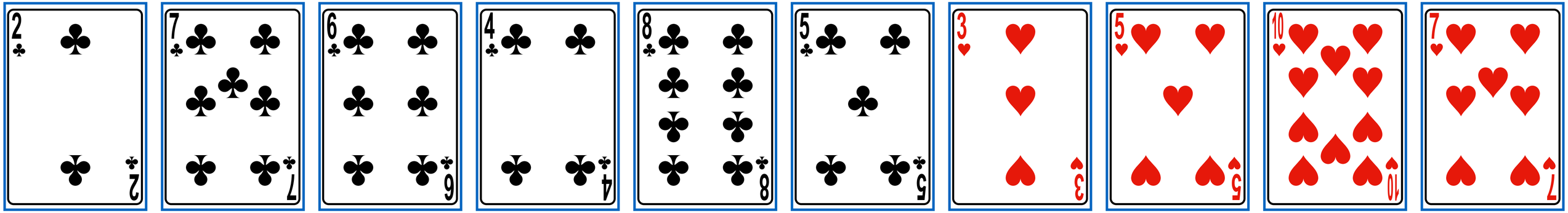


Tri comptage



Compteurs:

♣	6
♥	4



Tri comptage

```
fonction TriComptage(A,n,b,clé):

 C ← tableau de b compteurs à zéro
 pour tout e dans A
  C[clé(e)] += 1

 idx ← 1
 pour i de 1 à b
  tmp ← C[i]
  C[i] ← idx
  idx += tmp

 B = tableau de même taille que A

 pour tout e dans A
  B[C[clé(e)]] ← déplacer e
  C[clé(e)] += 1

 return B
```

Tri comptage

- Stable

```
fonction TriComptage(A,n,b,clé):

 C ← tableau de b compteurs à zéro
 pour tout e dans A
  C[clé(e)] += 1

 idx ← 1
 pour i de 1 à b
  tmp ← C[i]
  C[i] ← idx
  idx += tmp

 B = tableau de même taille que A

 pour tout e dans A
  B[C[clé(e)]] ← déplacer e
  C[clé(e)] += 1

 return B
```

Tri comptage

- Stable
- Pas en place :
  - tableau pour les compteurs
  - tableau pour la sortie

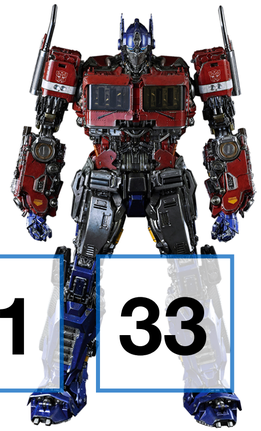
```
fonction TriComptage(A,n,b,clé):

 C ← tableau de b compteurs à zéro
 pour tout e dans A
  C[clé(e)] += 1

 idx ← 1
 pour i de 1 à b
  tmp ← C[i]
  C[i] ← idx
  idx += tmp

 B = tableau de même taille que A

 pour tout e dans A
  B[C[clé(e)]] ← déplacer e
  C[clé(e)] += 1

 return B
```

Tri comptage

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

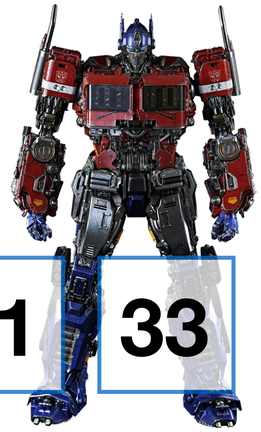
```
fonction TriComptage(A,n,b,clé):

 C ← tableau de b compteurs à zéro
 pour tout e dans A
  C[clé(e)] += 1

 idx ← 1
 pour i de 1 à b
  tmp ← C[i]
  C[i] ← idx
  idx += tmp

 B = tableau de même taille que A

 pour tout e dans A
  B[C[clé(e)]] ← déplacer e
  C[clé(e)] += 1

 return B
```

Tri par base - radix sort



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

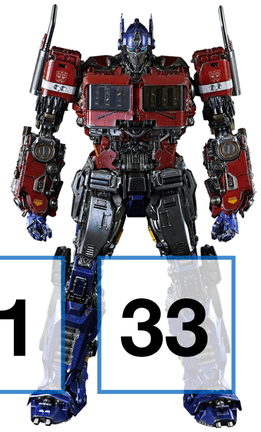
Tri par base - radix sort



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

Tri comptage par unité

Tri par base - radix sort

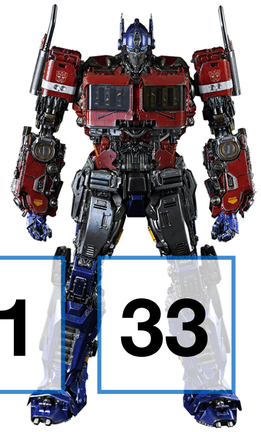


65	12	23	42	27	37	45	19	17	41	76	04	15	61	33

Tri comptage par unité

0 1		2		3		4	5			6	7			8 9

Tri par base - radix sort



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

Tri comptage par unité

							65							
0 1		2		3		4		5		6	7			8 9

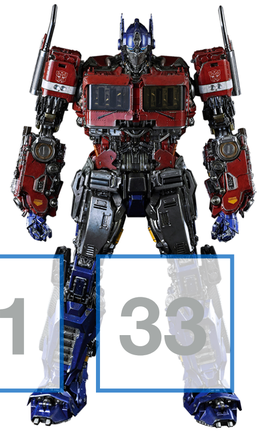
Tri par base - radix sort

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
0		1		2		3	4			5	6			8 7

Tri par base - radix sort



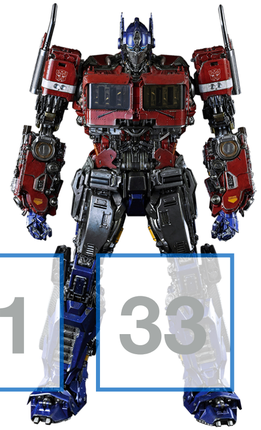
65	12	23	42	27	37	45	19	17	41	76	04	15	61	33

Tri comptage par unité

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

Tri comptage par dizaine

Tri par base - radix sort



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

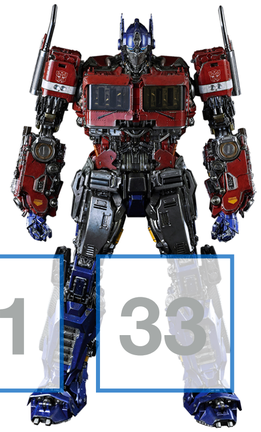
Tri comptage par unité

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

Tri comptage par dizaine

0x	1x				2x		3x		4x			5x 6x		7x

Tri par base - radix sort



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

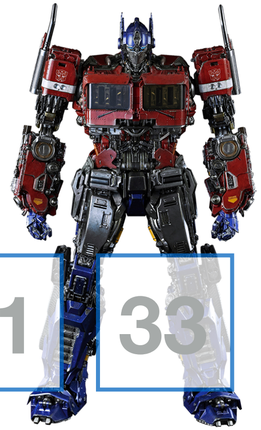
Tri comptage par unité

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

Tri comptage par dizaine

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

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
	0x				1x		2x		3x			5x 4x		6x

Tri par base - radix sort

```
fonction triParBase(T, d):
   Pour i allant de d à 1
       Trier avec un tri
       stable le tableau T
       selon le i-ème chiffre
```

Tri par base - radix sort

- Stable

```
fonction triParBase(T, d):
   Pour i allant de d à 1
       Trier avec un tri
       stable le tableau T
       selon le i-ème chiffre
```

Tri par base - radix sort

- Stable
- Pas en place
  - Un tableau pour les compteurs
  - Un tableau pour la sortie

```
fonction triParBase(T, d):
   Pour i allant de d à 1
       Trier avec un tri
       stable le tableau T
       selon le i-ème chiffre
```

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
  - 2.d passages sur le tableau : O(d.n)
  - d passages sur les compteurs : O(d.b)

```
fonction triParBase(T, d):
   Pour i allant de d à 1
       Trier avec un tri
       stable le tableau T
       selon le i-ème chiffre
```

Autres algorithmes de tri

http://en.wikipedia.org/wiki/Sorting_algorithm

Autres algorithmes de tri

Exchange sorts	Bubble sort - Cocktail sort - Odd–even sort - Comb sort - Gnome sort - Quicksort - Stooge sort - Bogosort

http://en.wikipedia.org/wiki/Sorting_algorithm

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

http://en.wikipedia.org/wiki/Sorting_algorithm

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

http://en.wikipedia.org/wiki/Sorting_algorithm

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

http://en.wikipedia.org/wiki/Sorting_algorithm

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 - Burstsort - Counting sort - Pigeonhole sort - Proxmap sort - Radix sort - Flashsort

http://en.wikipedia.org/wiki/Sorting_algorithm

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 - Burstsort - Counting sort - Pigeonhole sort - Proxmap sort - Radix sort - Flashsort
Concurrent sorts	Bitonic sorter - Batcher odd–even mergesort - Pairwise sorting network

http://en.wikipedia.org/wiki/Sorting_algorithm

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 - Burstsort - 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

http://en.wikipedia.org/wiki/Sorting_algorithm

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 - Burstsort - 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://en.wikipedia.org/wiki/Sorting_algorithm

ou encore … INEFFECTIVE SORTS

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

```
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]
```

ou encore … INEFFECTIVE SORTS

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

```
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]
```

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

ou encore … INEFFECTIVE SORTS

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

```
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]
```

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/

3.6. Tris en C et C++

THE C PROGRAMMING LANGUAGE

qsort

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*));
```

qsort

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

qsort

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

qsort

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

qsort

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);
```

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*`

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

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);
```

- ^

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);
```

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

`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 $\Theta(n \log n)$

`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

`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::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
```

std::sort

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);
```

std::sort - interface

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

- `first` : itérateur vers le premier élément à trier

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[`.

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 };
```

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 }
```

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 );}
};
```

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_parite = plus_petit_modulo{2};
std::sort( v.begin(), v.end(), compare_parite );
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 $\Theta(n \log n)$.

`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.

`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

`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.

std::stable_sort

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::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.

`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**.

`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

`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.

std::nth_element

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 ≤ `*nth`
- `[nth,last[` ne contient que des éléments ≥ `*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^{ieme} 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** linéraire $\Theta(n)$ 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 );
```

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.

std::partial_sort - effets de la fonction

```
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 }
```

std::partial_sort - version générique

La fonction de tri peut également être générique

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

std::partial_sort - version générique

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_va
leur_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`.

`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.

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 paritionne 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::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
```