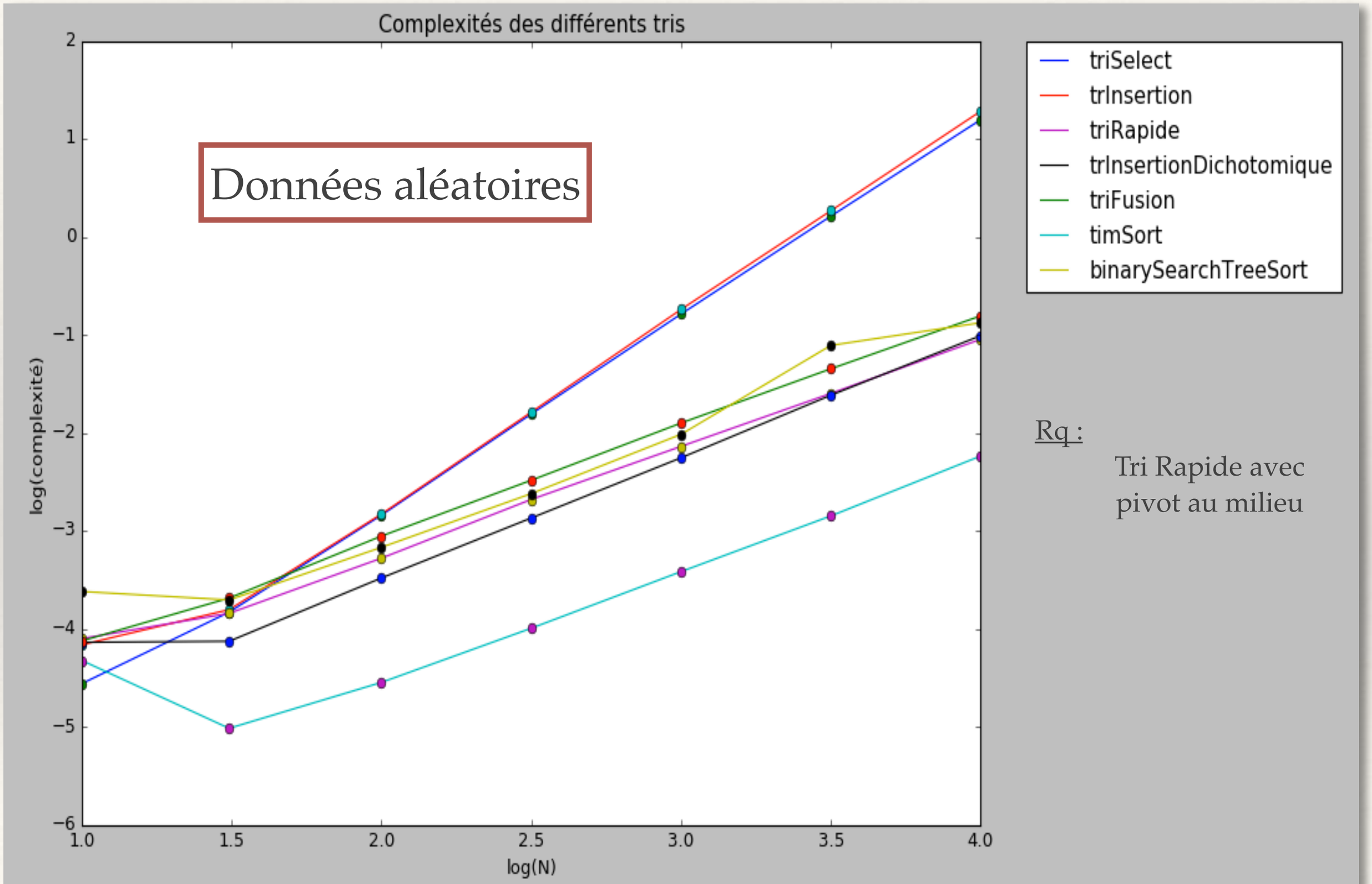
INFORMATIQUE

Les Tris de base :

tri par sélection
tri par insertion

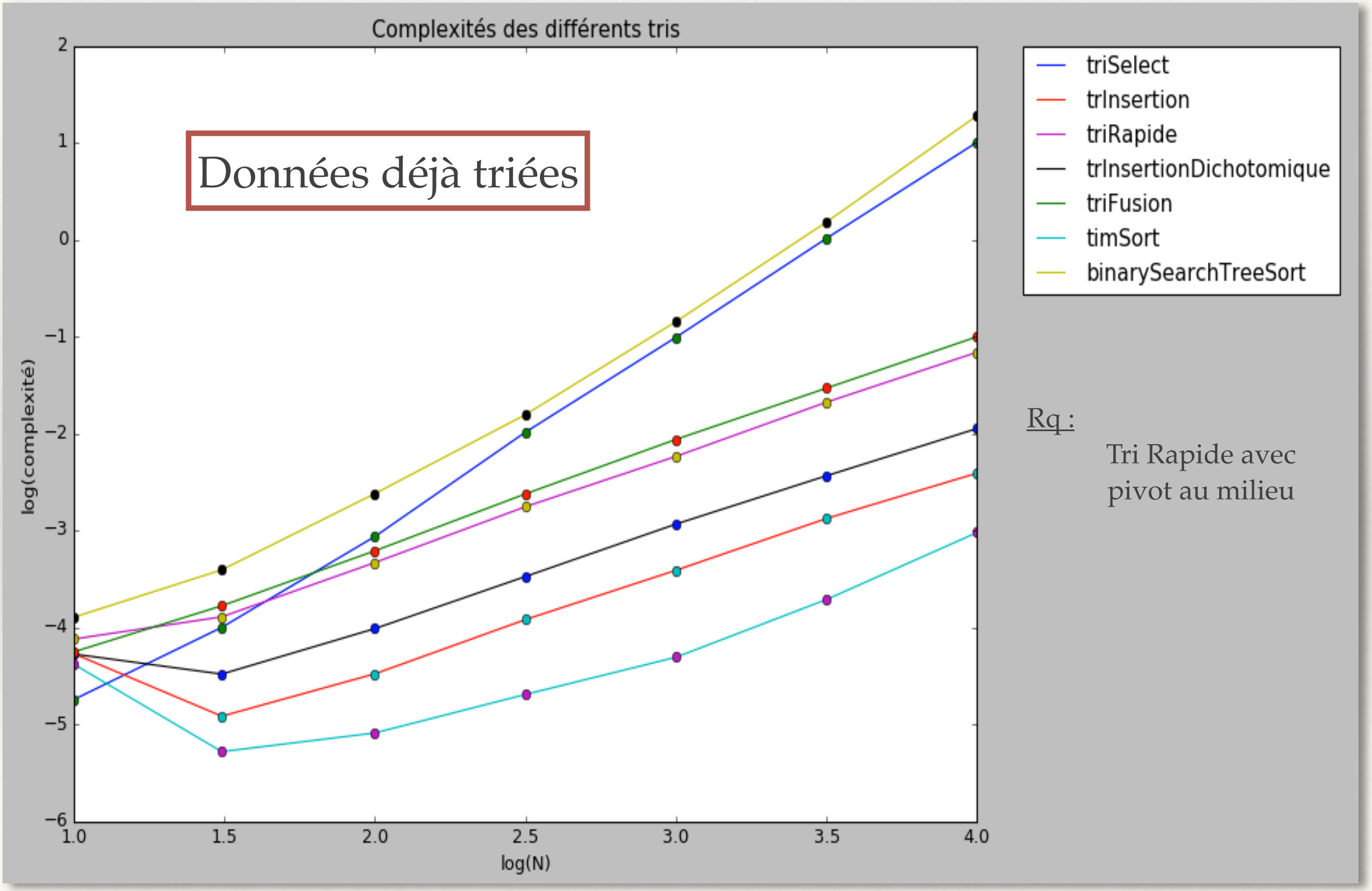
Benchmark test des tris de base



Rq :

Tri Rapide avec pivot au milieu

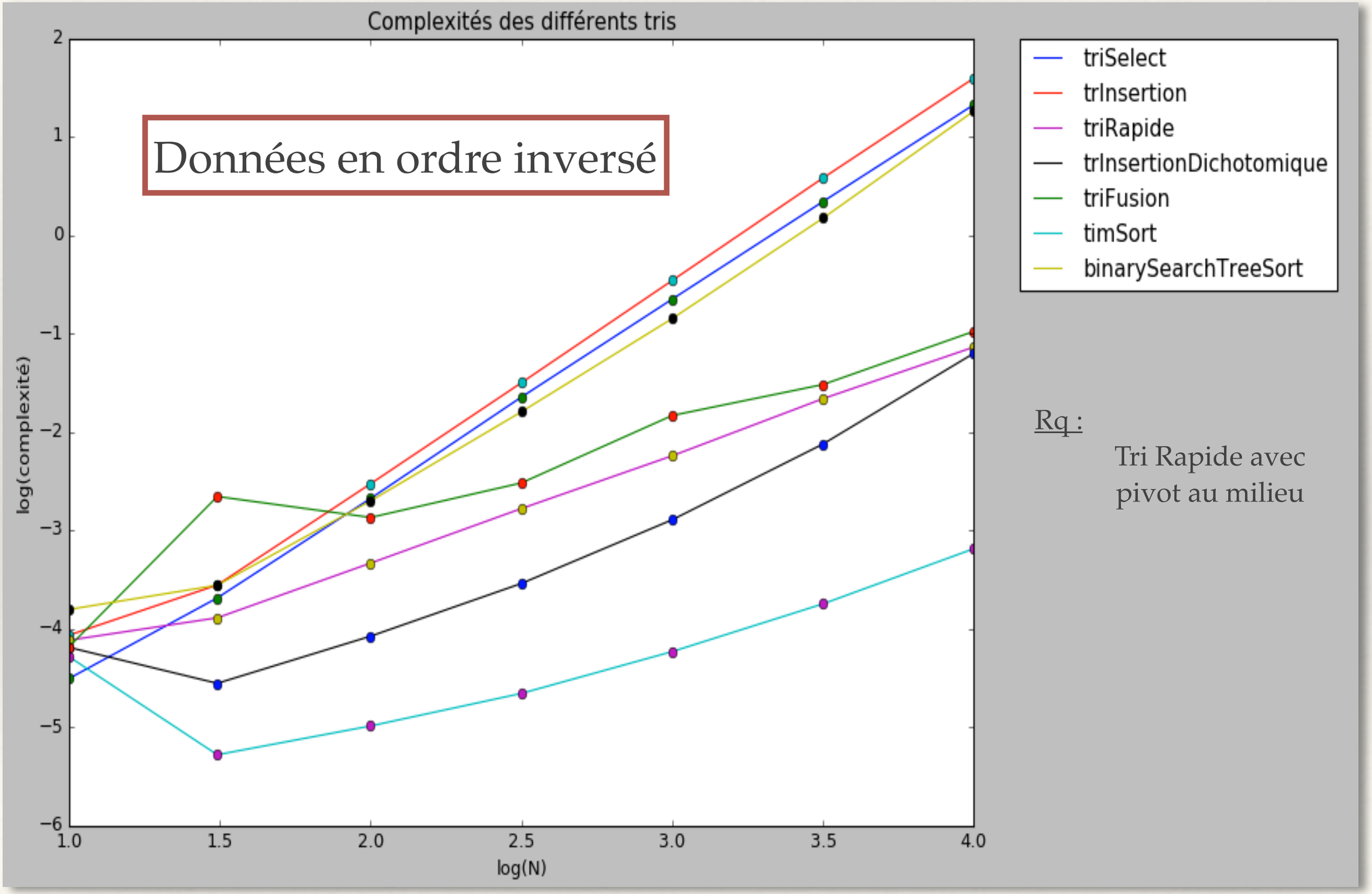
Benchmark test des tris de base



Rq :

Tri Rapide avec pivot au milieu

Benchmark test des tris de base

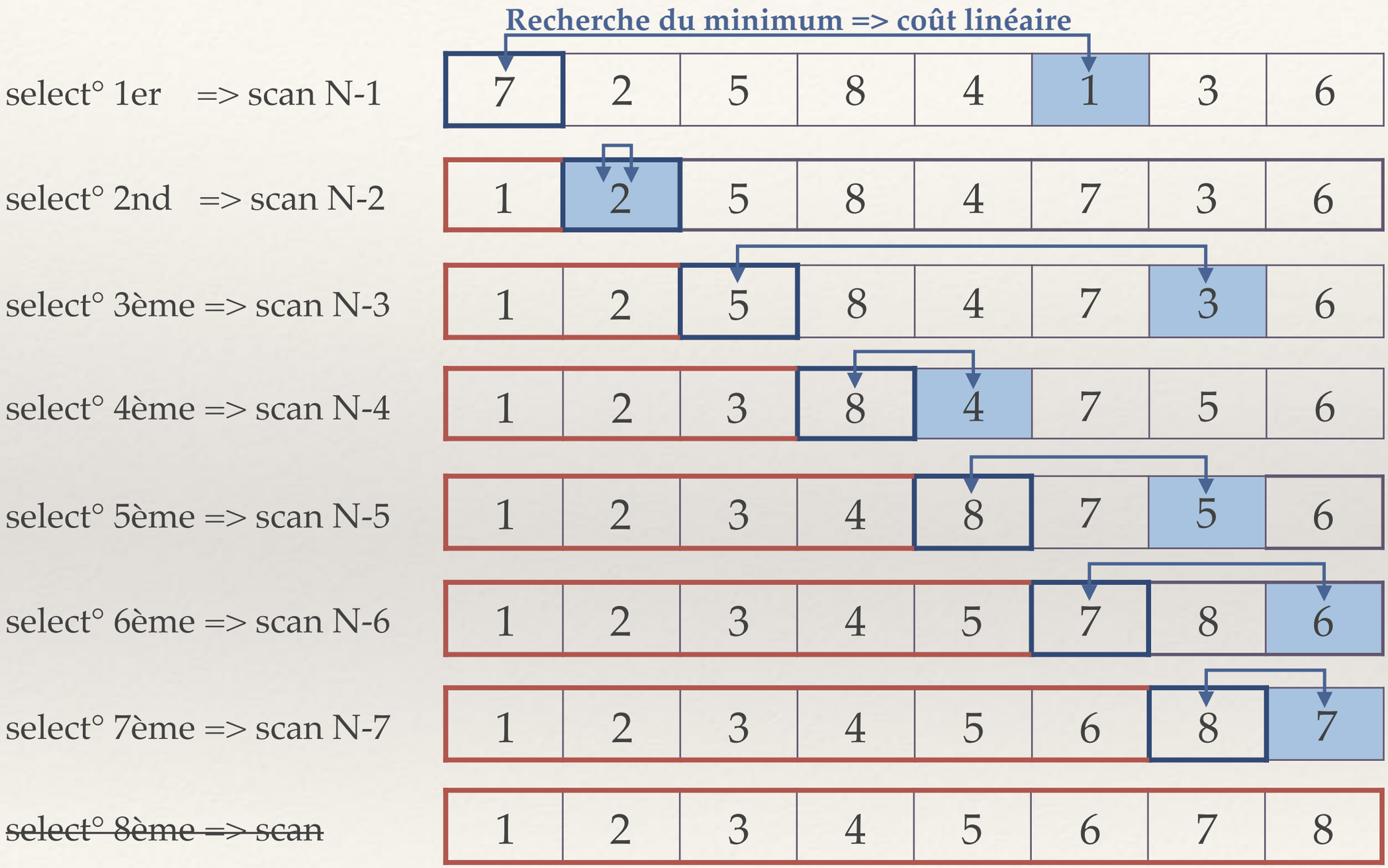


Rq :

Tri Rapide avec pivot au milieu

1 - Le Tri par Sélection

L'idée du tri par sélection est qu'à chaque itération **on sélectionne** le plus petit élément parmi ceux restants à la droite des éléments de gauche préalablement triés.



28 comparaisons et 7 permutations

Recherche du minimum => coût linéaire de N-1

select° 1er => scan N-1

select° 2nd => scan N-2

select° 3ème => scan N-3

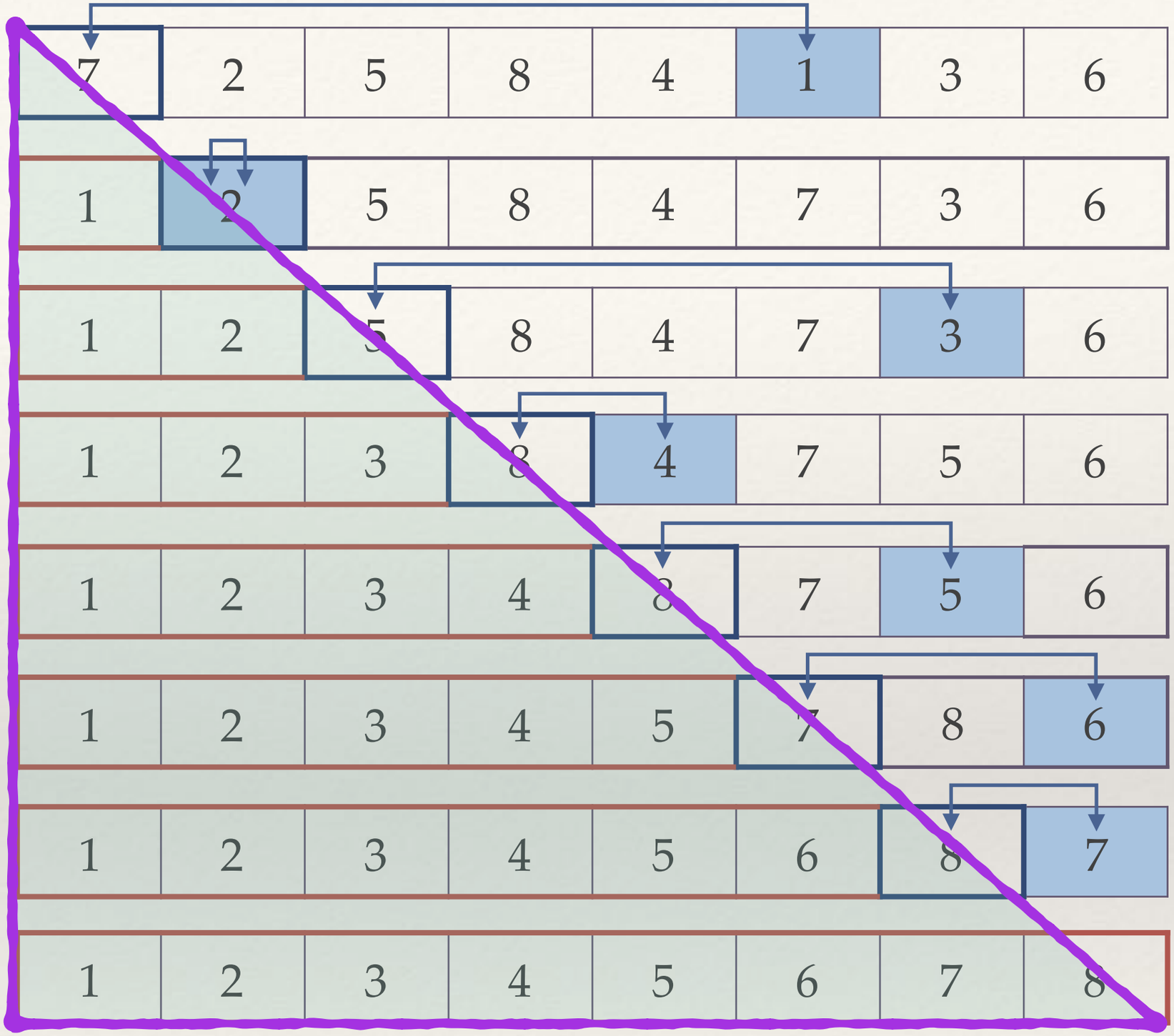
select° 4ème => scan N-4

select° 5ème => scan N-5

select° 6ème => scan N-6

select° 7ème => scan N-7

~~select° 8ème => scan~~



Signature du tri par sélection : rien ne change dans la diagonale inférieure

Travail à faire :

Proposer un codage pour le tri par sélection.

- **Etablir la terminaison.**
- **Evaluer la complexité [pire et meilleur des cas].**
- **Démontrer l'algorithme. [chercher un invariant]**

Codage

L'idée est simple :

- on cherche le plus petit => C'est le premier !
- puis le plus petit parmi ceux restants
- etc ….

```
def triSelect(L):

    for i in range(len(L)):                  #soit i une position

        for k in range( i+1, len(L) ):       #on passe en revue tous les suivants

            if ( L[k] < L[i] ):              #on sélectionne le plus petit
                L[k], L[i] = L[i], L[k]      #pour remplacer le i-ème : SWAP !

    return L
```

Complexité

Dans tous les cas… c'est la même chose !

1ère boucle FOR : taille N

2ème boucle FOR : taille de N-1 à 1

- Toujours 1 comparaison
- 0 ou 1 inversion

<u>Toujours :</u>

N-1 + N-2 + N-3 + … + 3 + 2 + 1 = **N(N-1)/2 itérations**

soit **N(N-1)/2 comparaisons dans tous les cas**

$$\mathbb{C} = O(N^2)$$

dans tous les cas

Dans le meilleur des cas : aucune inversion (tableau déjà trié)

Dans le pire des cas : N(N-1)/2 inversions (tableau en ordre décroissant)

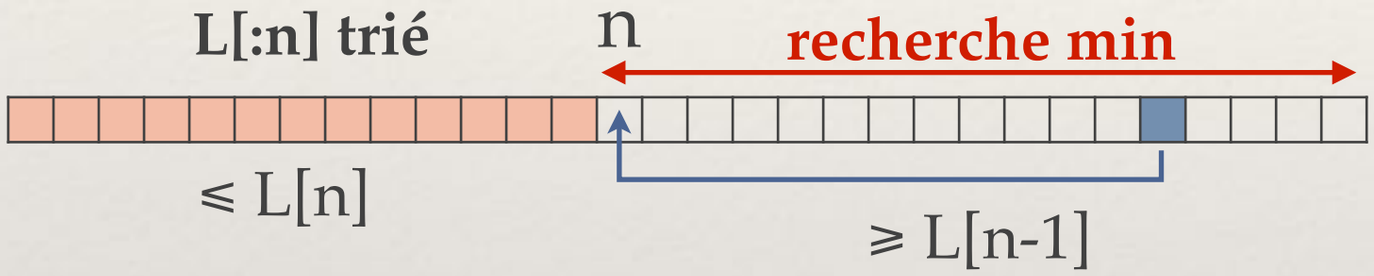
Terminaison : Double **boucle FOR** => #itération déterminé => Termine

Preuve :

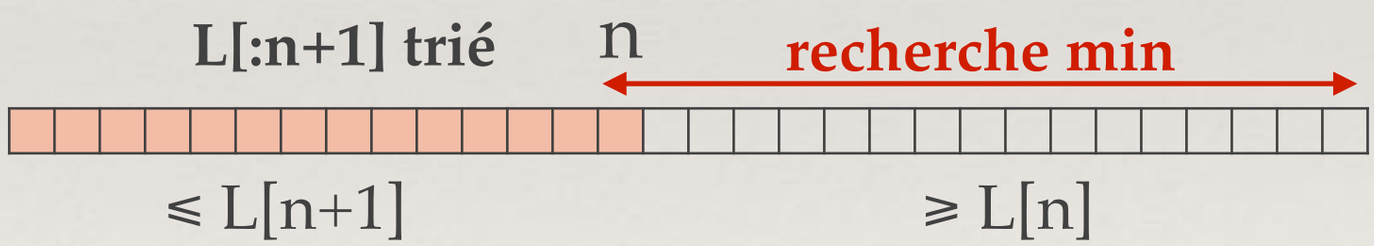
Invariant : => L[:n] tableau de taille n trié à l'étape n

L[:1] est trié car il n'y a qu'un élément.
De plus c'est le plus petit, car la 1ère étape est une recherche de minimum de L.

Soit **L[:n] trié à l'étape n** :



i = n lors de l'étape n+1

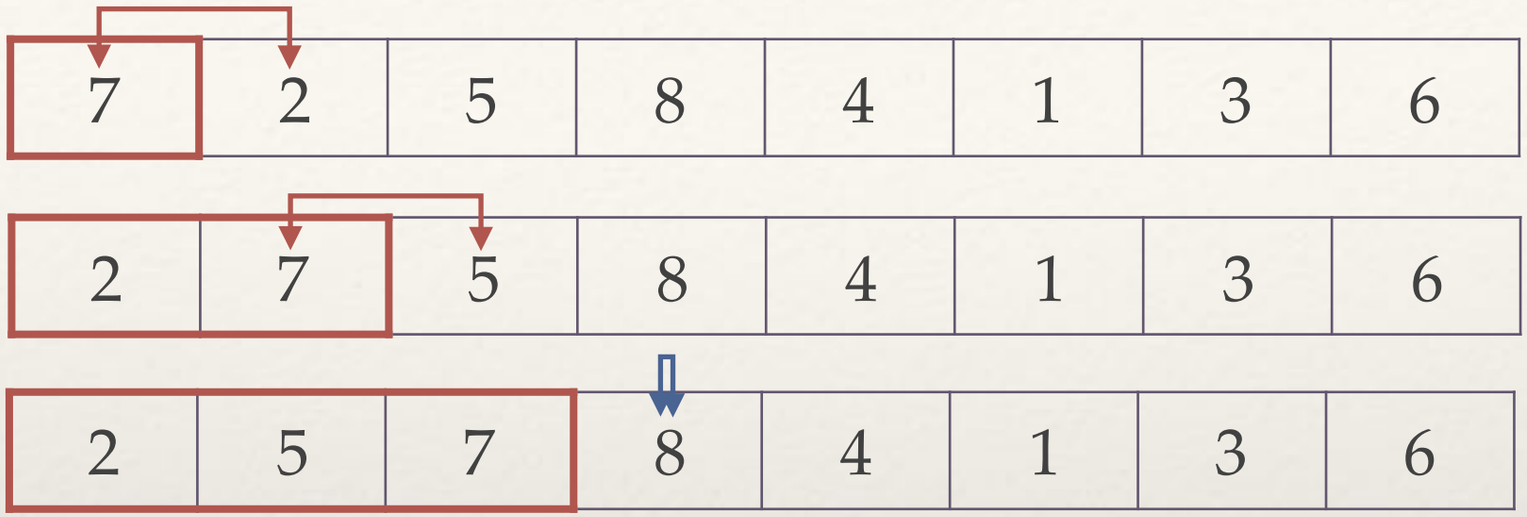


L[i] est remplacé par tout élément plus petit de L[i+1:] que l'on passe tous en revue.
donc **L[:n+1] est bien trié à l'étape n+1** ce qui prouve l'algorithme par récurrence.

C'est un algorithme Naïf !

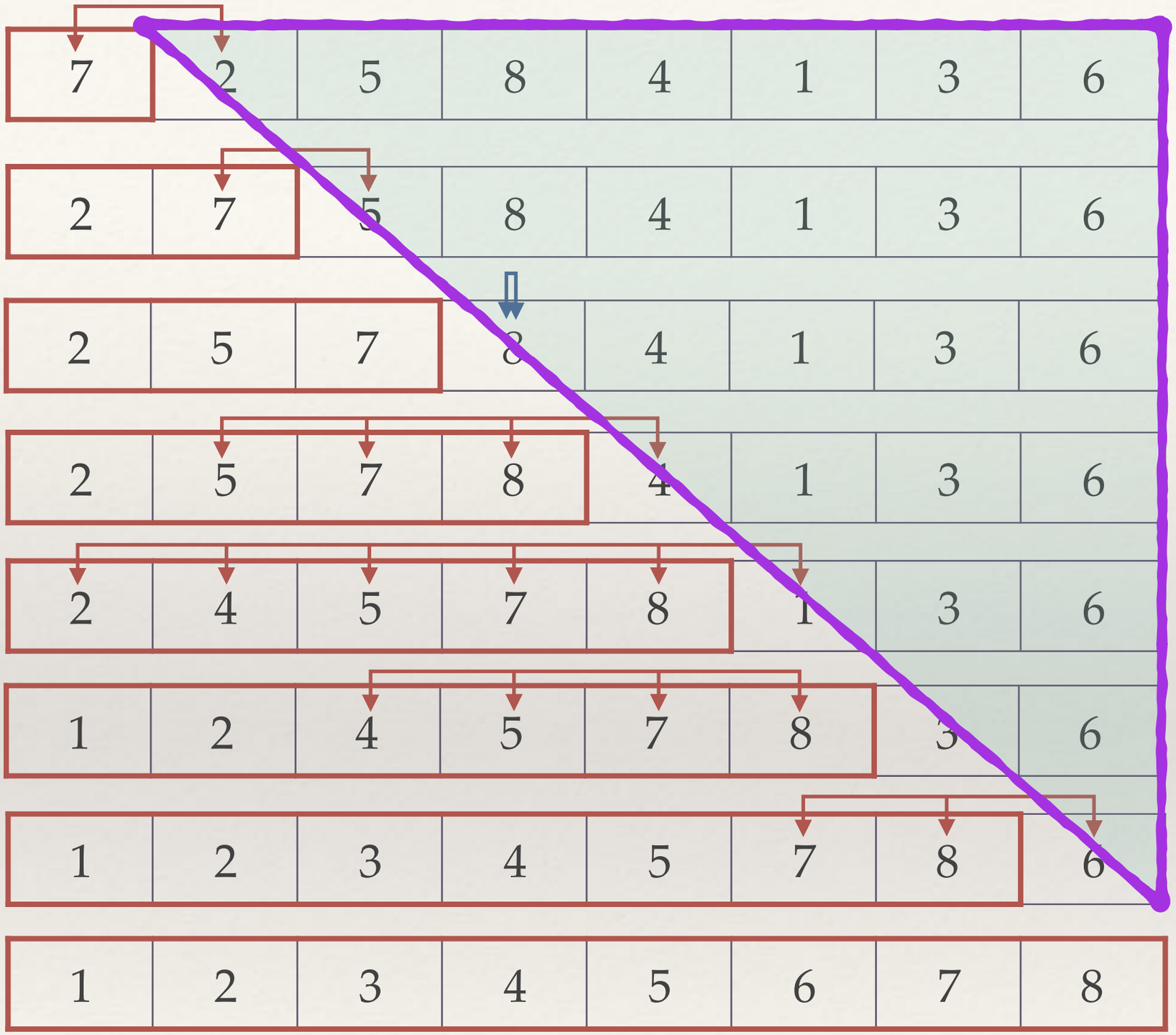
2 - Le Tri par Insertion

L'idée du tri par insertion est qu'à chaque itération on insère l'élément suivant dans la partie gauche du tableau préalablement triée en le glissant autant que possible à gauche.



20 comparaisons et 15 permutations

Signature du tri par insertion : rien ne change dans la diagonale supérieure



Travail à faire :

Proposer un codage pour le tri par insertion.

- **Etablir la terminaison.**
- **Evaluer la complexité [pire et meilleur des cas].**
- **Démontrer l'algorithme. [chercher un invariant]**

Codage

L'idée est simple :
On veut maintenir les i premiers éléments du tableau triés !
=> pour **insérer** le i+1 ème on le fait glisser vers la gauche jusqu'à ce qu'il soit bloqué par un élément plus petit que lui

```
def triInsertion(L):

    for i in range(len(L)-1):                    #soit i une position

        j=i+1       #aff                         #je prend le suivant

        while( j > 0 and L[j-1] > L[j] ):  #comp  #tant que celui à sa gauche est plus grand

            ( L[j-1], L[j]) = (L[j], L[j-1] )  #inv  #je les permute SWAP !
             j = j - 1     #aff                      #glisse à gauche

return L
```

Complexité

Meilleur des cas / Pire des cas / en moyenne

1ère boucle FOR : taille N - 1

- 1 affectation : on prend l'élément suivant

2ème boucle WHILE : taille de 0 à j < N-1 selon les cas

- 0 ou 1 inversion SWAP
- 0 ou 1 affectation

<u>Selon les cas le nombre d'opérations réalisées n'est pas le même :</u>

meilleur des cas

N - 1 < # **affectations** < N-1 + N(N-1)/2

N - 1 < # **comparaisons** < N(N-1)/2

0 < # **inversions** < N(N-1)/2

pire des cas

Linéaire

O(N) < complexité < O(N^2)

Quadratique

Le meilleur des cas : —> est obtenu si le tableau est déjà trié (pas de while)

Le pire des cas : —> est obtenu si il est trié en ordre décroissant (tjrs while)

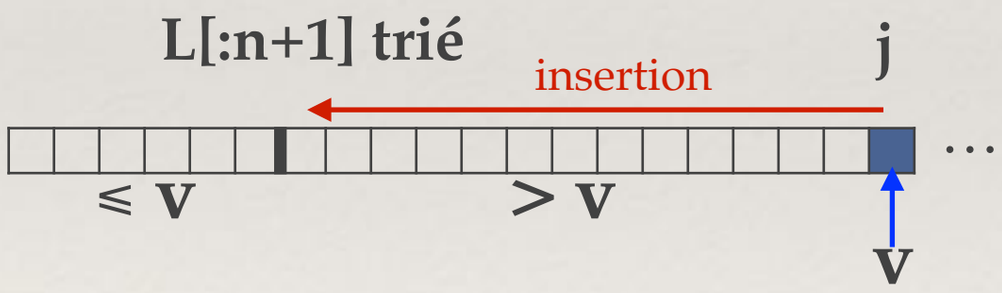
En moyenne : on doit faire toutes les boucles FOR mais le glissement d'insertion ne parcourra que la moitié de la partie triée [valeur moyenne] la complexité sera donc **N(N-1)/4 ~ O(N^2)**

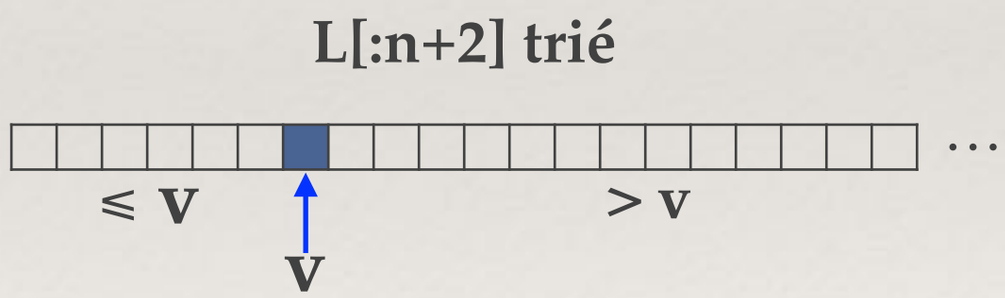
Terminaison : On peut majorer le nombre de passage dans la boucle while par N(N-1)/2. Donc **l'algorithme termine** !

Rq : - il y a un nombre connu de boucle FOR
- dans chaque boucle while : j est une suite décroissante bornée par le bas en 0

Preuve : **Invariant :** => après n itérations n+1 éléments triés à gauche

- **L[0] est trié** car il n'y a qu'un élément (mais ici ce n'est pas forcément le plus petit)
- Soit **L[:n+1] trié à l'étape n** : n éléments triés et j est le n+1 ème élément soit **v = L[j]**



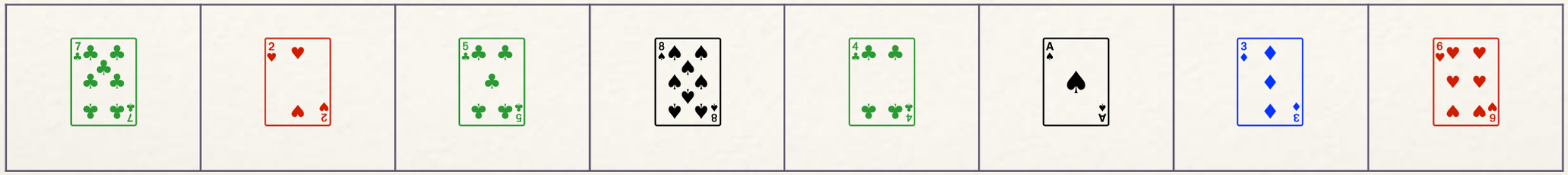


- on obtient **L[:n+2] trié à l'étape n+1** ce qui prouve l'algorithme par récurrence

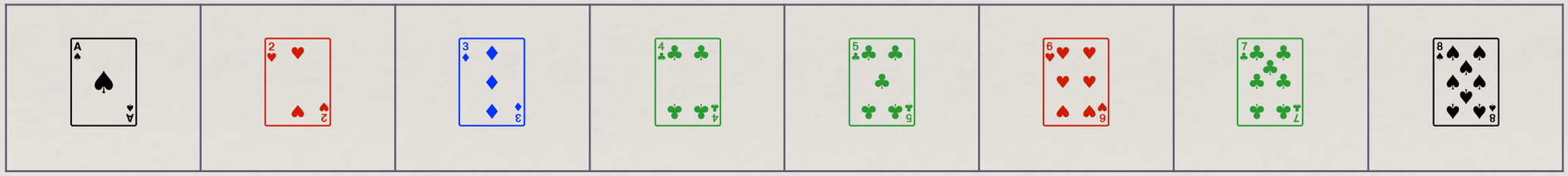
Rq : On voit bien qu'en soi l'insertion représente un coût moyen linéaire

Stabilité du tri par insertion

On considère un tableau dont les entrées, toutes uniques ont deux attributs :
Cartes à jouer (valeur, couleur) :

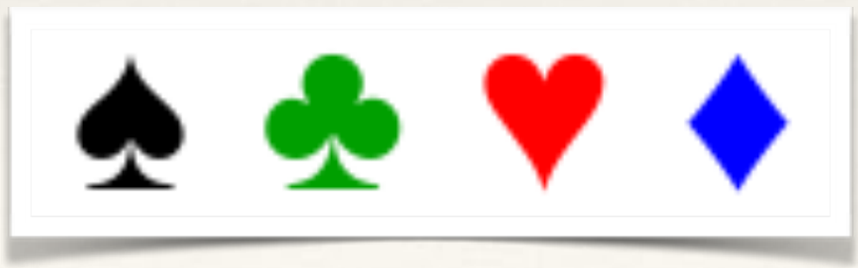


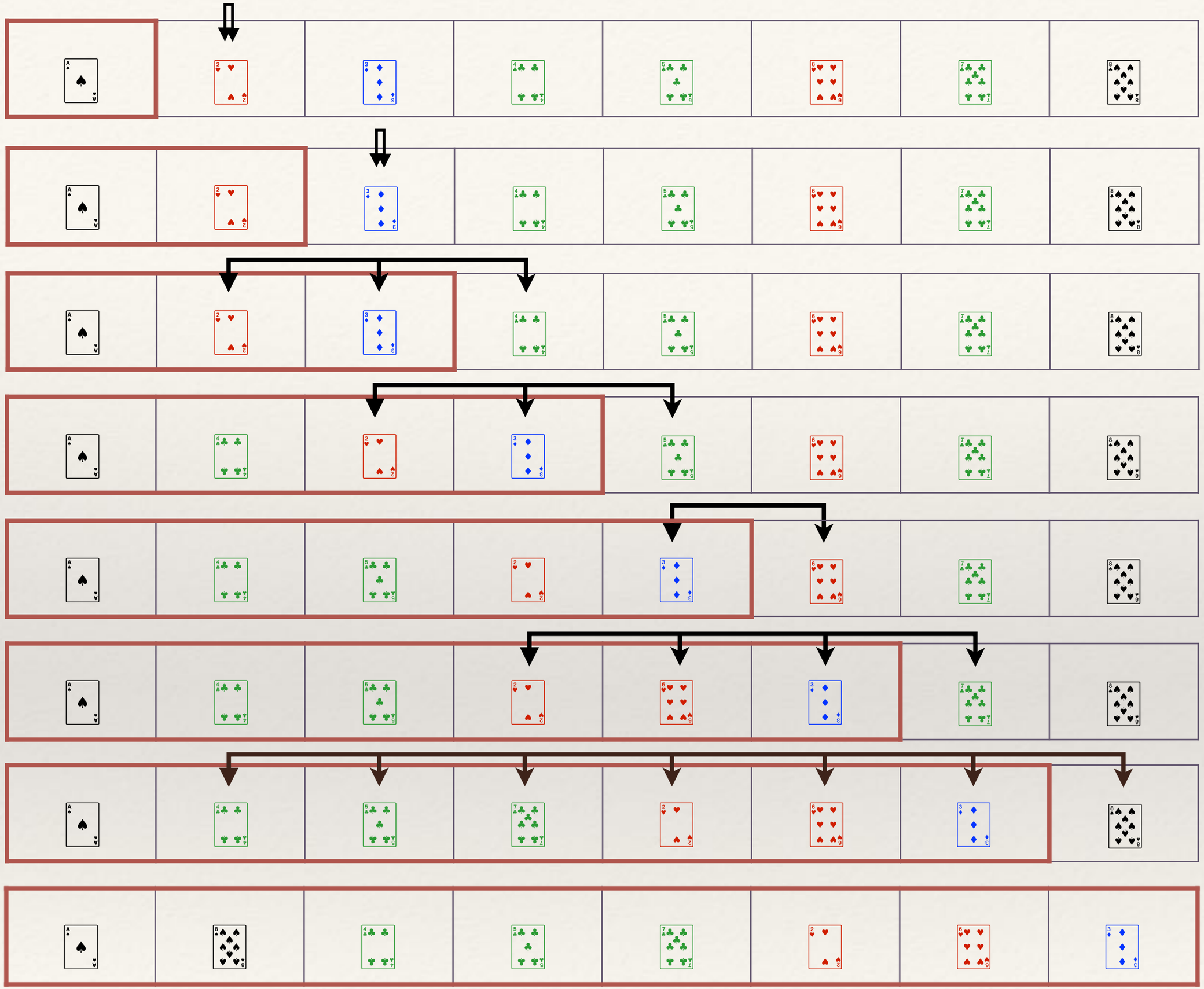
Premier tri par valeurs



C’est la même chose que précédemment, puisqu’on ignore les couleurs.

On souhaite désormais trier à nouveau les cartes mais cette fois selon l’ordre des couleurs, soit :





Que remarque-t-on ?

On parle de stabilité du tri lorsque le tri selon un attribut maintient l'ordre selon un autre attribut sur lequel on a déjà fait le tri.

Pourquoi est-ce si formidable ?

Youh Houuuhh !!!!

Ex : Si je trie mes chansons
- par ordre alphabétique,
- puis par album :

Je verrai dans chaque album mes chansons par ordre alphabétique !!!

En pratique une base de donnée peut avoir des millions d'entrées, et certains attributs (âge, sexe, nationalité) sont très redondants.

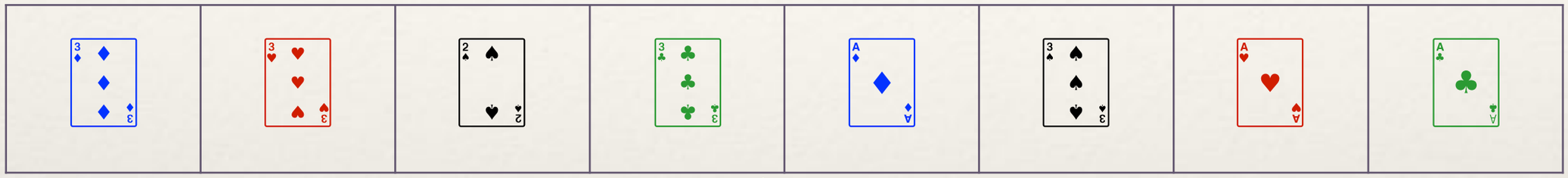
Or un tableau même partiellement trié se trie beaucoup plus vite par insertion !

Quelle peut-être la conséquence de la stabilité sur la complexité en temps ?

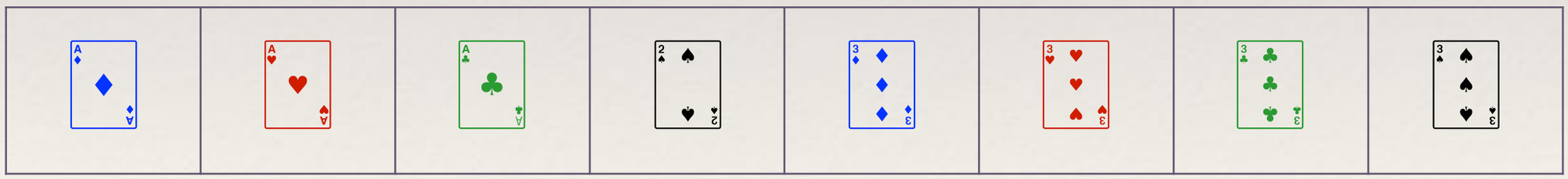
Tableau avec redondances des valeurs

[Complexité en temps du tri par insertion]

Pour illustrer les choses on réduit notre espace à [3 valeurs & 4 couleurs]



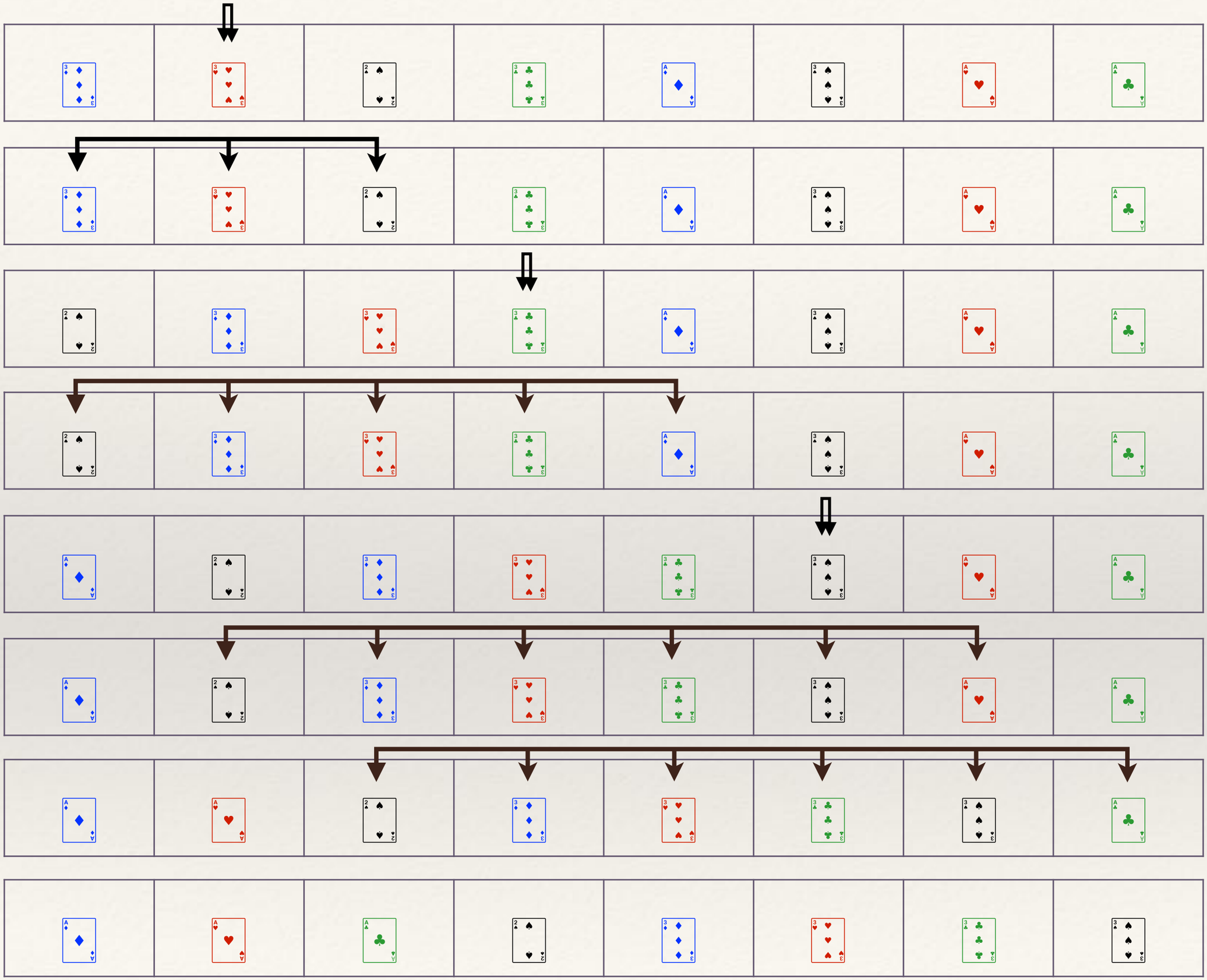
Premier tri par valeurs



16 permutations !

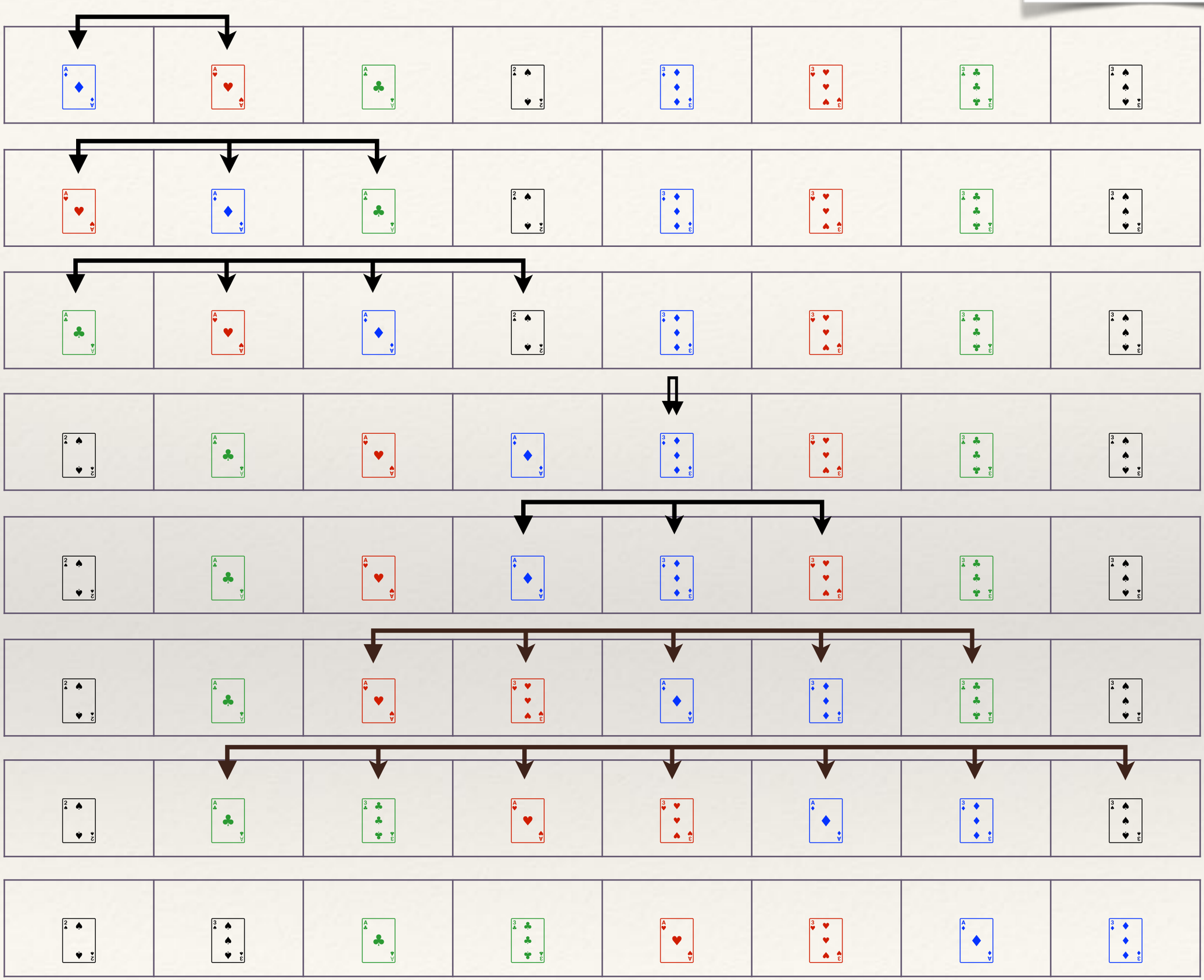
Rq : les (valeurs, couleurs) initiales ont elles été choisies au hasard ?

Premier tri par valeurs



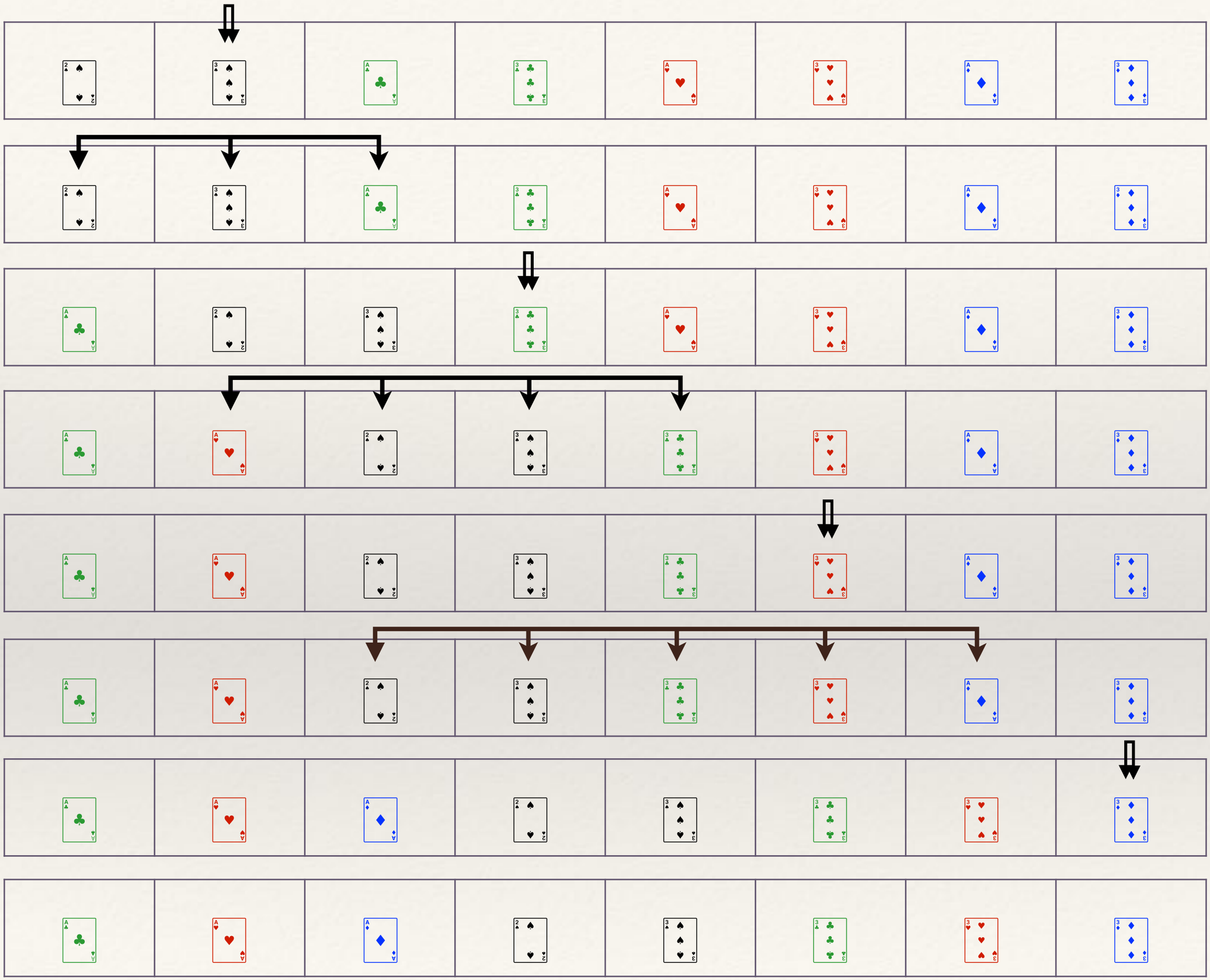
16 permutations !

Second tri par couleurs



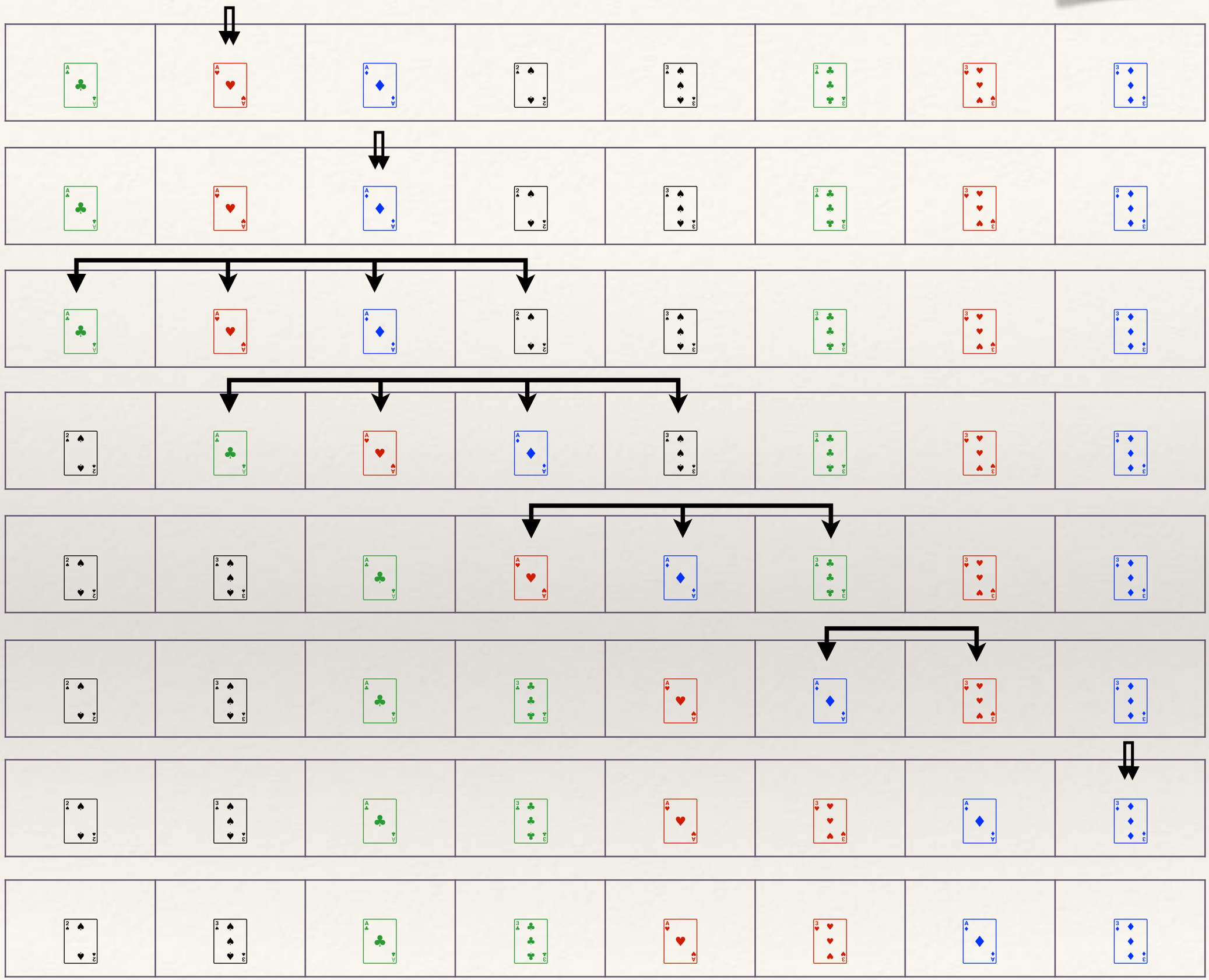
18 permutations !!

On tri à nouveau par valeurs



9 permutations seulement !!

On tri à nouveau par couleurs



9 permutations seulement !!

Inutiles de continuer car les conditions initiales sont ensuite toujours les mêmes…

On pourrait en revanche envisager plus d'attributs distincts mais redondants, ainsi qu'une quantité massive de données :

ex : un million d'utilisateurs qui ont tous entre 1 et 100 ans
- chaque âge est redondant environ 10.000 fois.
- l'ordre de grandeur est le même pour la nationalité.
- deux sexes possibles…. etc

L'idée d'une amélioration des performances grâce à un tableau partiellement trié est reprise dans le tri Shell «**Shell-sort** » qui est une altération du tri par insertion.

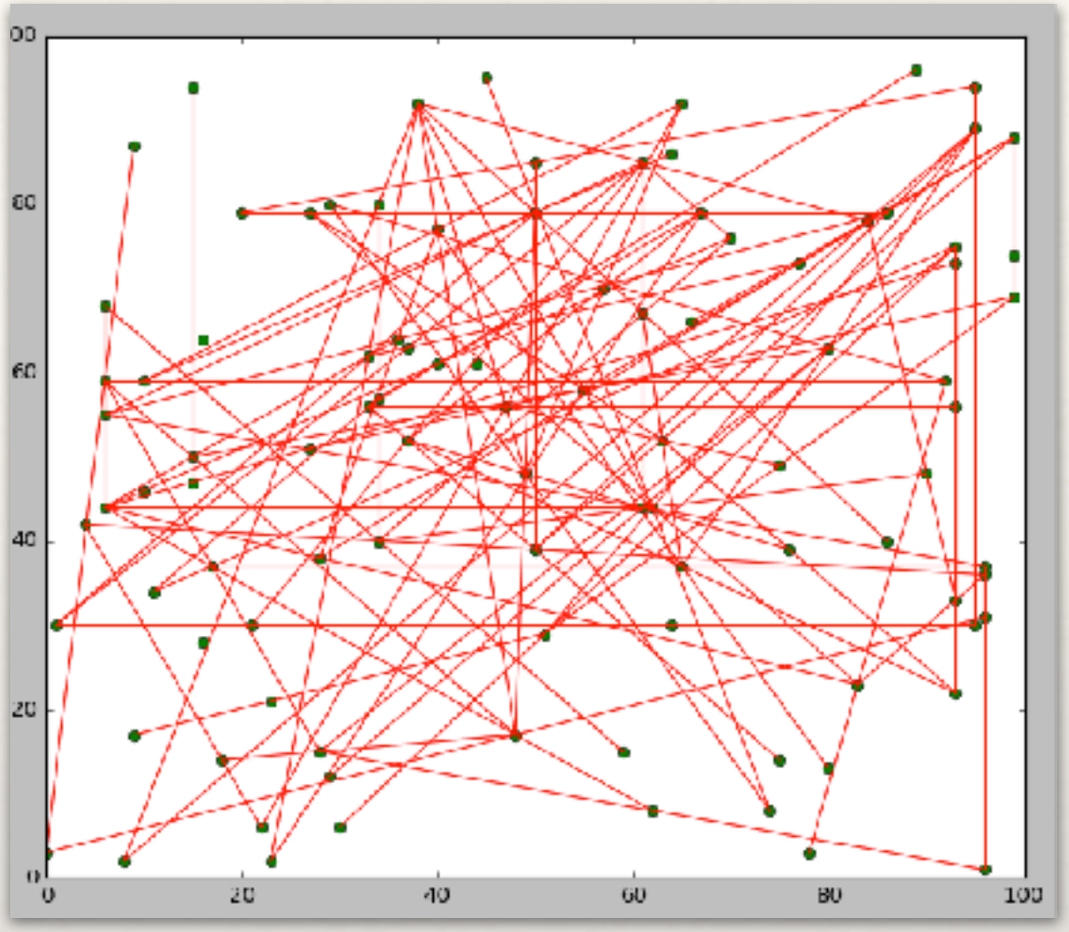
Conclusion :

On voit que **la stabilité du tri par insertion peut considérablement en améliorer les performances**. C'est d'autant plus intéressant que ce tri est linéaire si le tableau est déjà trié ou trié par morceaux.

Application de la stabilité du tri

Recherche optimisée d'alignements :

$$\mathbb{C} = O(N^2.\log(N))$$



liste d'alignements vide O(1)

Faire une boucle sur tous les points : N - fois

- Trier les N-1 points restants selon l'ordre lexicographique (x, y) O(N.Log(N))
- Calculer les angles[modulo pi]/horizontale de tous les points [0, pi] O(N)
- Trier les points (**tri stable !**) selon les angles O(N.Log(N))
- Balayer la liste des angles O(N)
  - identifier chaque alignement
  - si le point de référence est l'extrémité basse
    - on garde l'alignement O(1)
    - on garde en mémoire ses extrémités pour le tracé

98 points aléatoires : 93 alignements [81 de 3 points / 10 de 4 points / 2 de 5 points]