Structures de Données & Algorithmes II
-
Introduction & Tris

Pascal Mérindol (CM, TD, TP)

merindol@unistra.fr
http://www-r2.u-strasbg.fr/~merindol

Ce cours est construit sur la base de plusieurs supports pédagogiques parmi lesquels les cours de D. Bechmann et J.F Dufourd.
Gif <- @Wikipédia. L'usage de ce support ne peut être qu'académique.

Organisation générale

- **Volume horaire (6 ECTS)**
  - 20h CM (mardi 8-10h)
  - 22h TD (mardi 10-12h)
  - 12h TP (lundi 15h30-17h30)
- **Supports (Planning & CM, TD, TP, Projet)**
  - https://robinet.u-strasbg.fr/enseignement/Sda
  - https://moodle2.unistra.fr/
  - + notes au tableau...
- **MECC**
  - Deux devoirs sur table (1h le 17/11 -20%- et 2h avec convocation le 8/12 - 50%)
  - Un projet de dev (13/10 -> 19/12)

Contenu

- **Les tris (6-8h)**
  - introduction : animations, images et intuitions.
  - complexité : borne inférieure théorique.
  - tri par sélection : simple, tri par tas (heapsort).
  - tri par insertion : simple, dichotomique.
  - tri rapide : quicksort.

- **Arbres & Forets (12-14h)**
- **Les graphes (14-16h)**
- **Les tables (6-8h)**

Références

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

…

* ***Spécification algébrique, algorithmique et programmation*** : Jean-François Dufourd, Dominique Bechmann, Yves Bertrand.

* ***The Art of Computer Programming, Volume 3, Sorting and Searching*** : Donald Knuth.

* ***Introduction to Algorithms*** : Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.

Motivations ?

- **D.E. Knuth dans The Art of Computer Programming**
  - Tome 3 : «Sorting & Searching»
    - *Computer manufacturers estimate that over 25% of the running time on their computers is currently being spent on sorting (. . .) We may conclude that*
      - *(i) there are many important applications of sorting, or*
      - *(ii) many people sort when they shouldn't, or*
      - *(iii) inefficient sorting algorithms are in common use.*
    - *The real truth probably involves some of all three alternatives. In any event we can see that sorting is worthy of serious study, as a practical matter.*

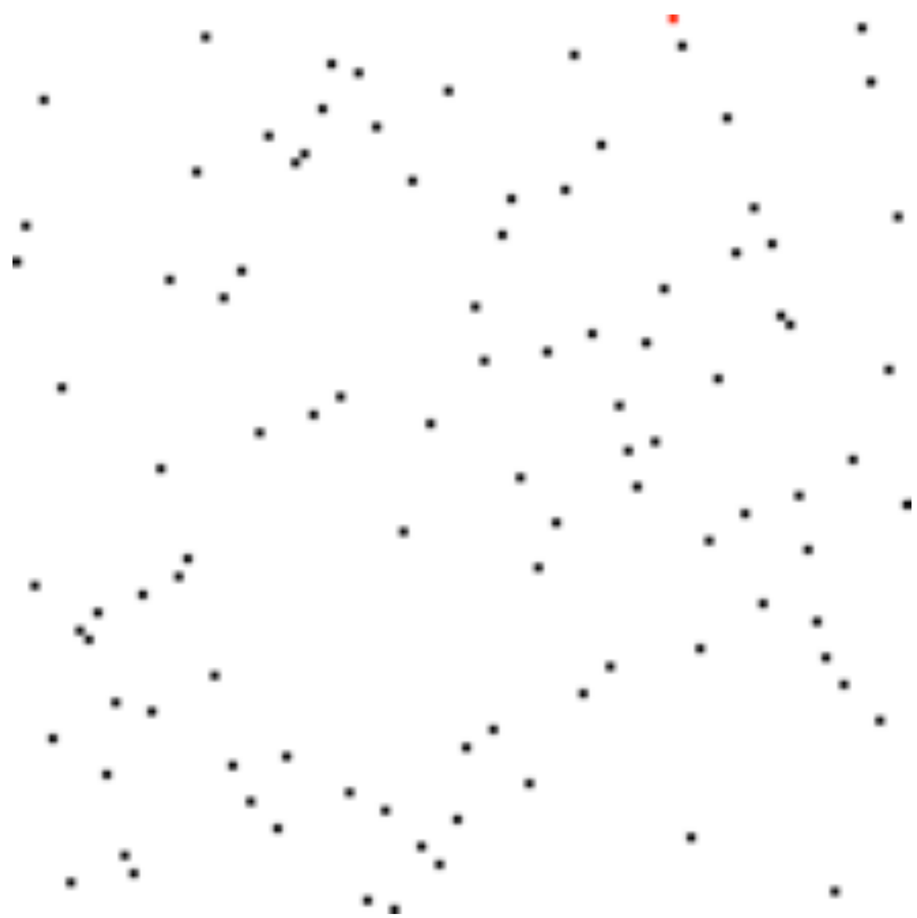
Les tris internes (triés par méthode)

- **Par sélection**
  - exemple efficace : en utilisant une file de priorité en tas (i.e. Heapsort)
  - sélection par transposition : Bubble sort & Shaker ("joli" mais pas efficace)
- **Par transposition** (i.e. par échange e.g. Comb sort, Bubble sort)
    - par transposition-partitionnement (i.e. Quicksort) : efficace en moyenne !
- **Par insertion**
  - séquentielle (Insertion sort) ou dichotomique (Binary search insertion sort)
  - implémentation efficace avec un BST type AVL (B-Tree Sort)
- **Par fusion** (e.g. Mergesort)
  - diviser pour régner (approche algorithmique ~ Quicksort)
- **Par distribution : sans comparaisons** e.g. Counting sort, Bucket sort, etc**.**
- **... & Techniques hybrides** e.g. Timsort, Introsort, etc.

Tri par sélection : illustration

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

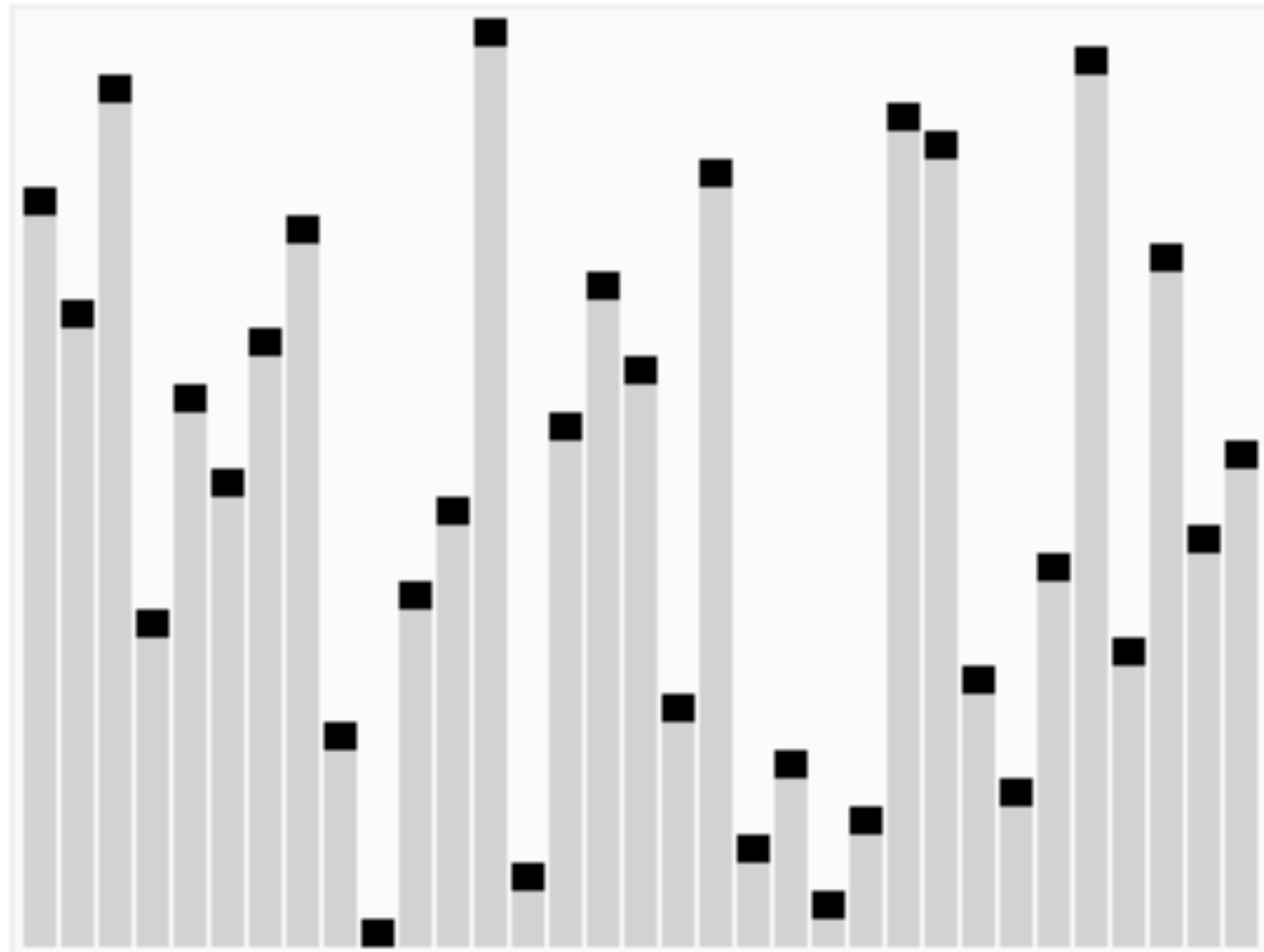
- Que comprenez vous de ces animations ?
  - qui est quoi ?



- **Méthode la plus «naturelle pour le dev»**
  - mais complexité quadratique

Heapsort

10	4	8	5	12	2	6	11	3	9	7	1

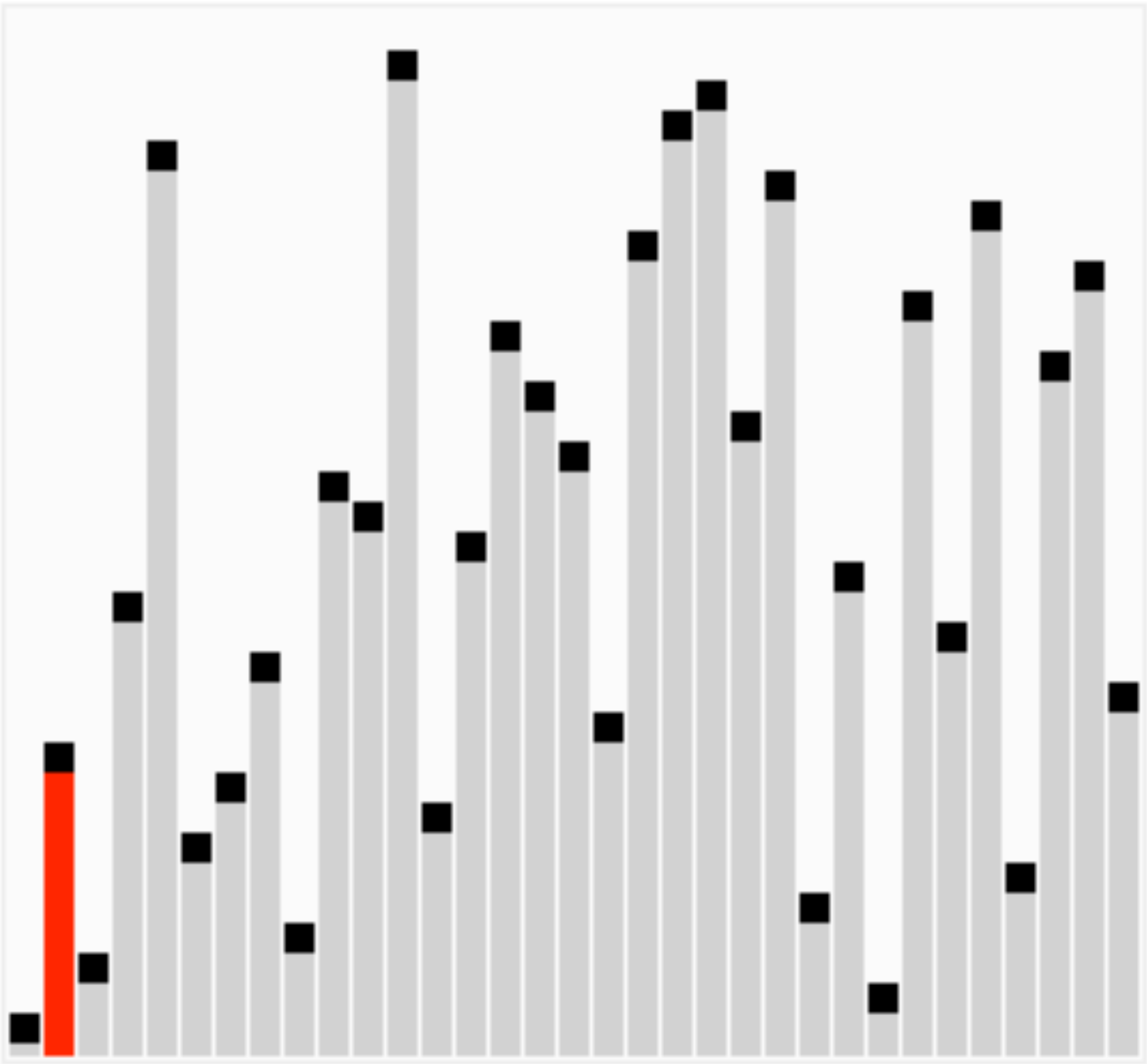


- **Méthode efficace dans le pire cas**
  - cas moyen ~ pire cas O(nlogn)
  - au lieu de O(n^2) avec une méthode par sélection classique

Bubble sort

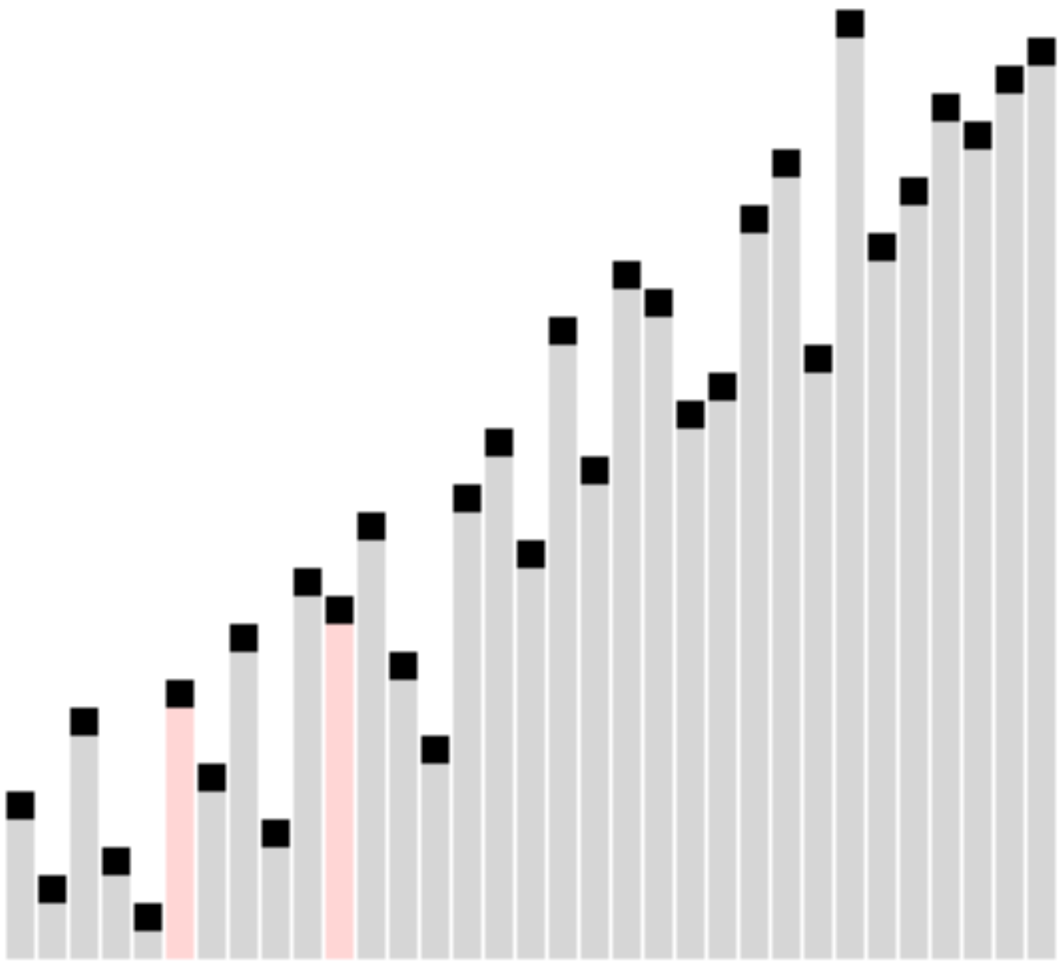
- **Méthode peu efficace dans tous les cas**
  - O(n^2) en moyenne et au pire - O(n) avec bcp de chances...
  - très simple mais long... : échange des valeurs consécutives (gap de 1)

Shaker (ou Cocktail) sort



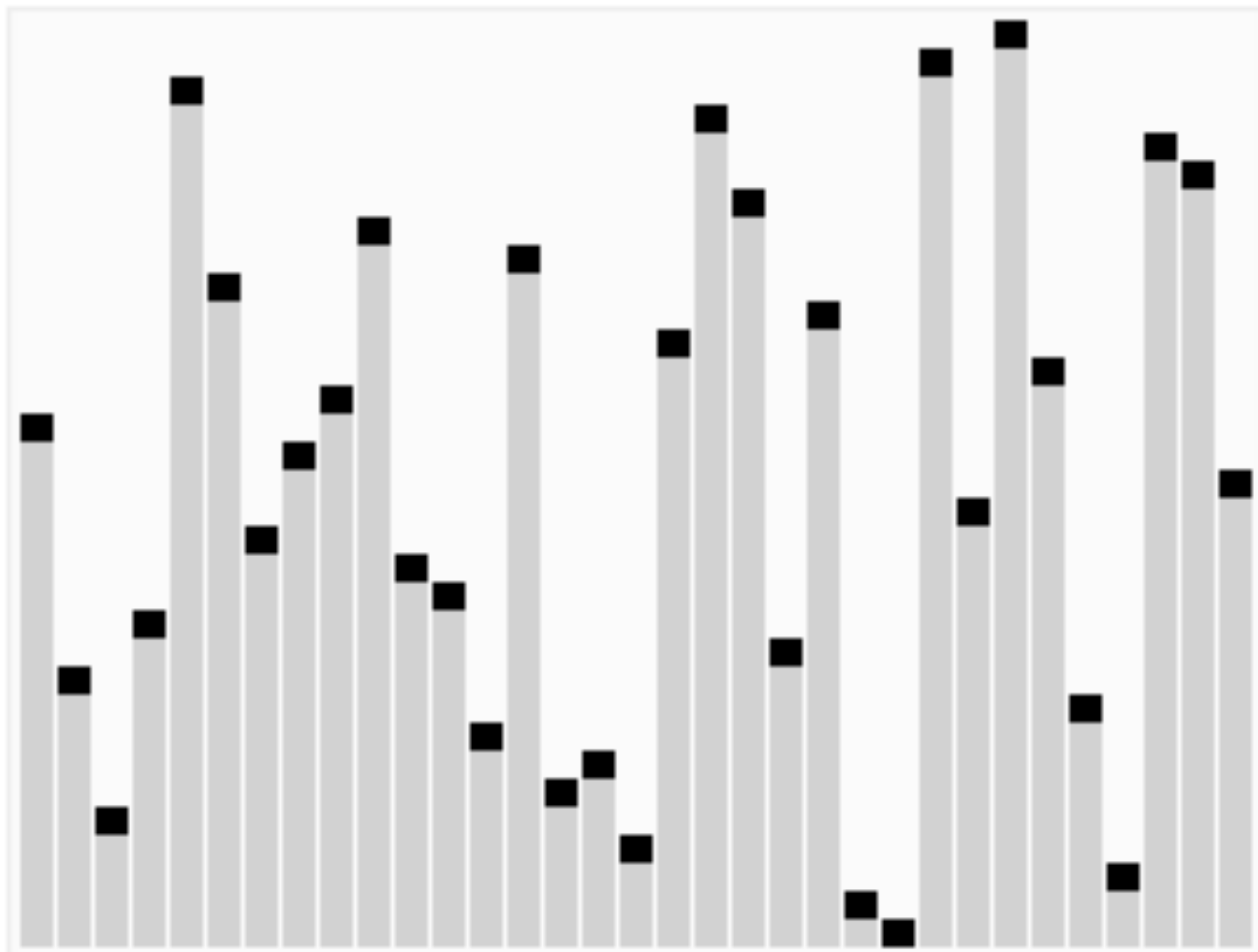
- **Légère amélioration de Bubble**
  - Aller / retour pour booster les perfs…
  - …mais aucun interêt pratique pour ces méthodes ~ insertion séquentielle naive :(

Comb sort



- **Méthode par transposition (échange) avec gap dynamique**
  - meilleures perfs en moyenne que ses cousins Bubble et Shaker !
    - NB : cette méthode n'est pas dans la catégorie «par sélection»
  - ...mais reste mauvais dans les pires cas :(

Quick sort

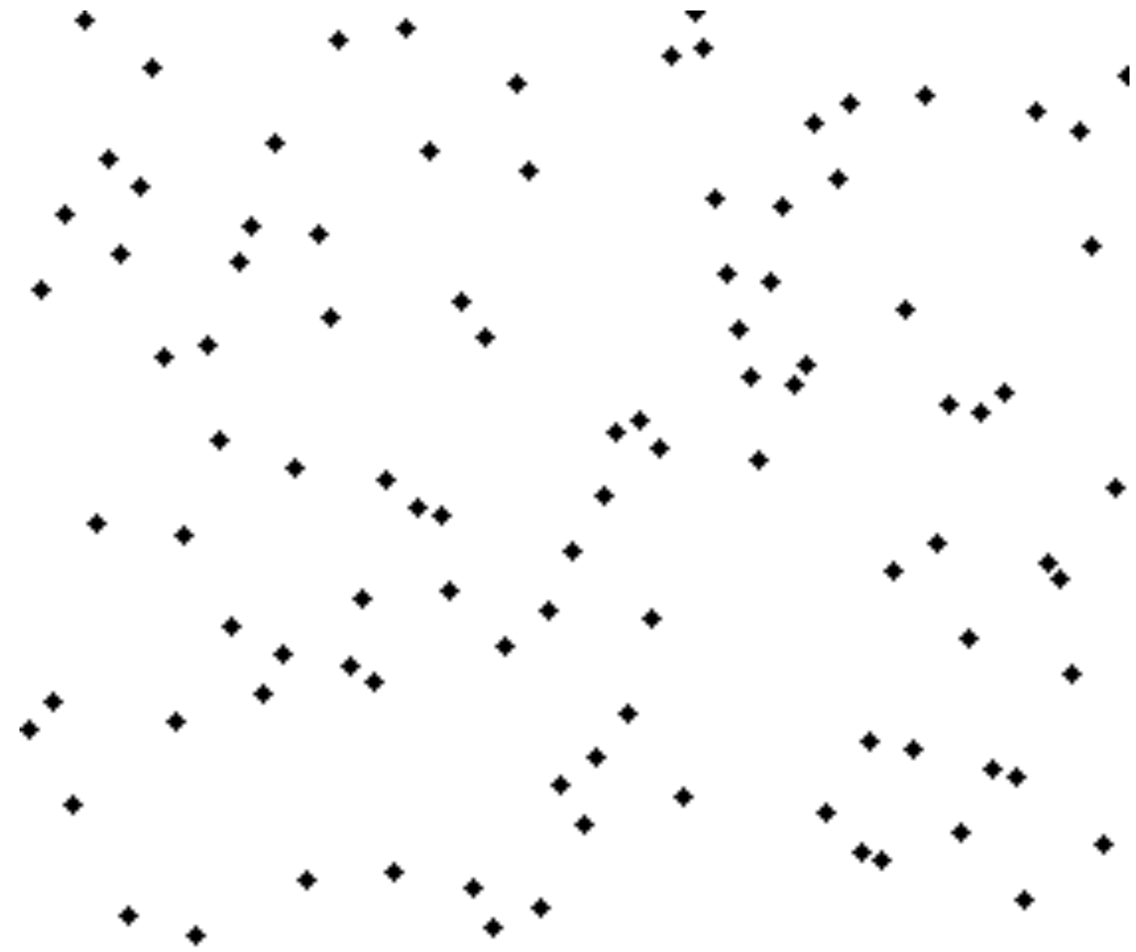


- **Méthode efficace en moyenne**

UNIVERSITÉ DE STRASBOURG

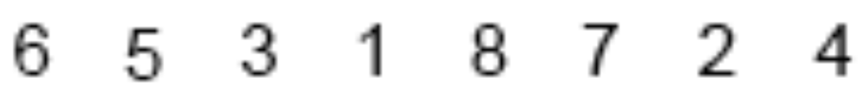
Insertion sort

6 5 3 1 8 7 2 4



- **Méthode peu efficace avec des structures naives**
  - comparable à heapsort en théorie avec un BST équilibré
  - pratique pour les petits ensembles...(et «naturelle humainement»)

Merge sort



- **Méthode simple et efficace**
  - comparable à Heapsort et B-Tree sort en théorie
  - approche diviser pour mieux régner...
    - se parallélise bien :)

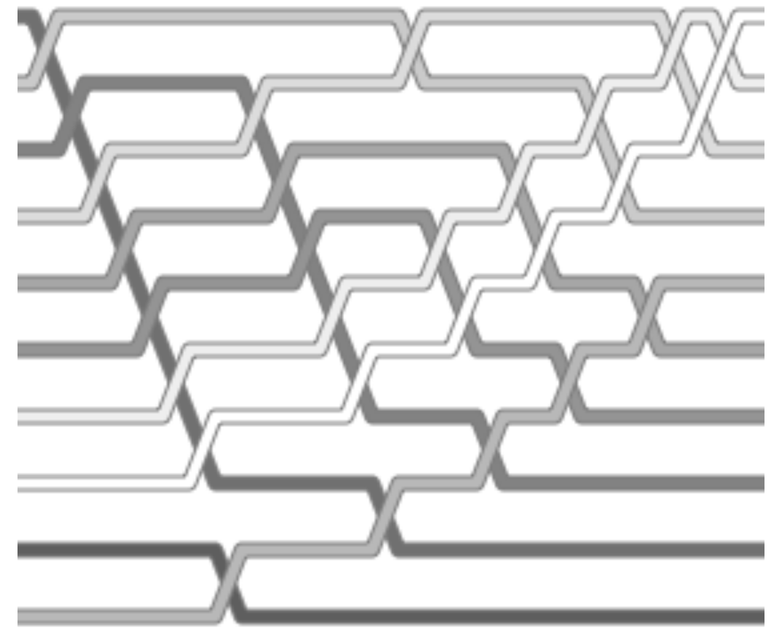
Plus d’anims ici :

- **http://www.sorting-algorithms.com/**

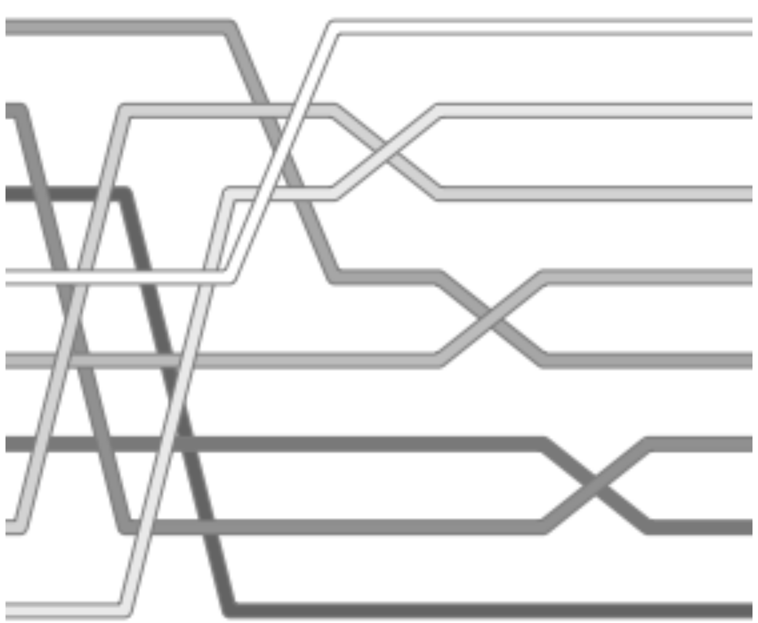
- **http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html**

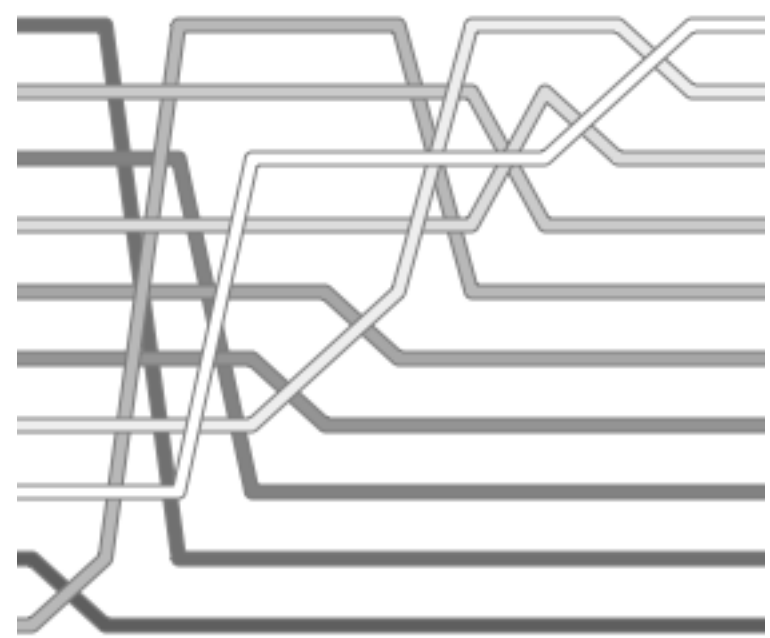
  - => google image : sorting anim gif

Encore ? en mode statique cette fois ..?



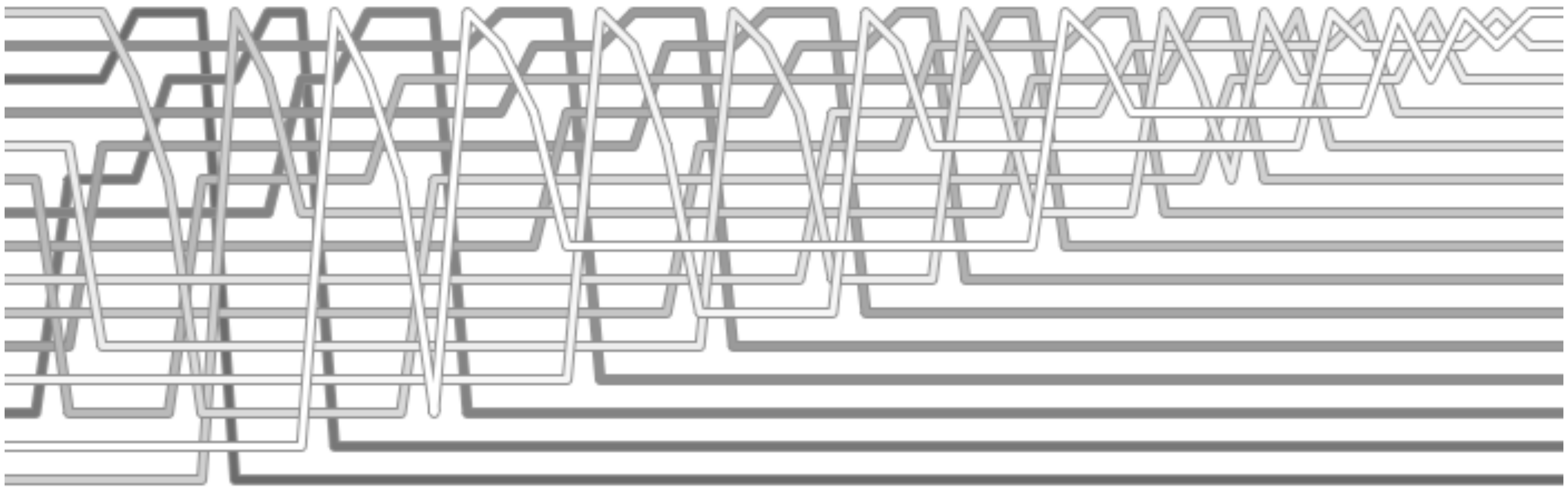
Qui est quoi ?

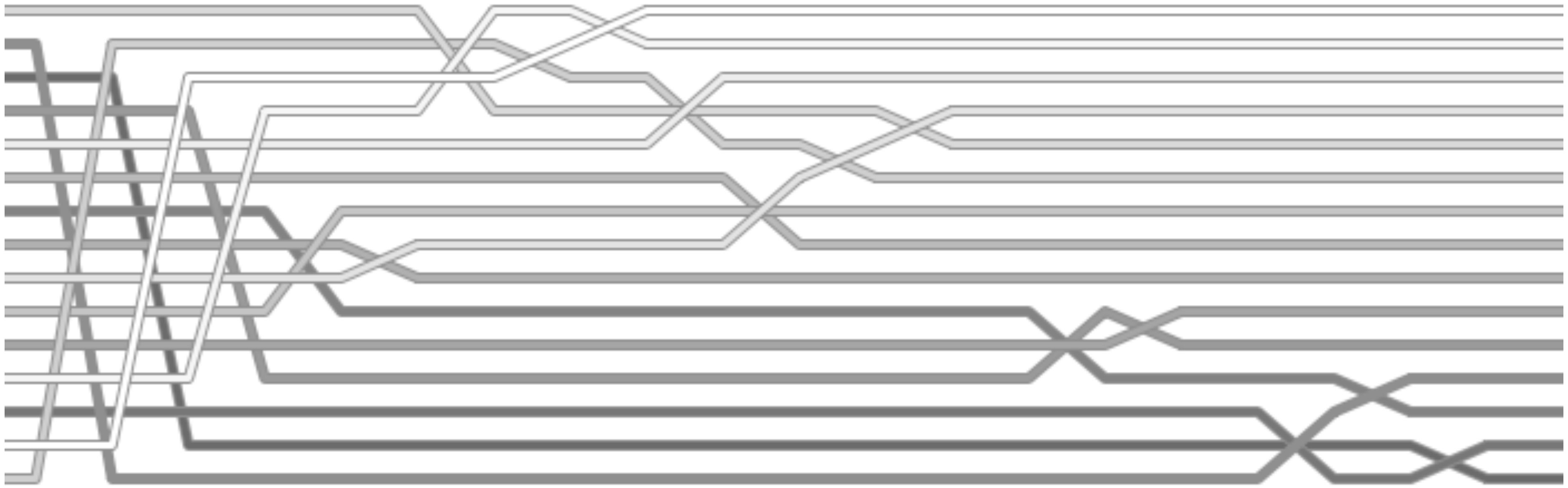




Mieux que les anims ?

http://corte.si/posts/code/visualisingsorting/





Après l’intuition, les définitions et notions de complexité...

- **Dans ce cours : tri interne par <u>comparaisons</u>**
- **Soit une sorte S d’objets à trier munie d’un préordre total _≤_ large, avec une version stricte _<_ et un opérateur d’équivalence _~_**
  - spéc **PREORDT**
  - opérations
    - **_ ≤_, _<_, _~_ : S S -> Booléen**
  - axiomes x, y, z : S
    - $x \leq x = \text{vrai}$ // réflexivité
    - $x \leq z \supset (x \leq y \wedge y \leq z) = \text{vrai}$ // transitivité
    - $x < y = x \leq y \wedge \neg\, y \leq x$ // préordre strict
    - $x \sim y = x \leq y \wedge y \leq x$ // équivalence
    - $x \leq y \vee y \leq z = \text{vrai}$ // préordre total

Listes vs. Tables

- **Pour obtenir de bonnes performances, on privilégie les tables aux listes**
  - accès direct aux éléments (représentation contiguë)

- **Soit une table t avec comme index les entiers 0...n-1 et des valeurs dans S**
  - opération tri : Table -> Table
    - t' = tri(t)
  - t' est une permutation de t tel que :
    - $0 \leq i < j \leq n-1$ implique $t'[i] \leq t'[j]$
      - ce serait l'inverse pour un tri décroissant...

- **Stabilité : le tri est stable si, pour toutes tables t, tri(t) conserve l'ordre relatif entre les éléments équivalents dans t (instable sinon).**

Compléxités

http://bigocheatsheet.com/

- **Rappels : «ordre de grandeur»**
  - O borne supérieure asymptotique (majorant)
    - f est dominé asympotiquement par g
  - Ω borne inférieure asymptotique (minorant)
    - f domine asympotiquement g
  - Θ borne d'encadrement asympotique
    - f et g ont le même ordre de grandeur asymptotique

Définitions formelles ?

Borne inférieure de complexité

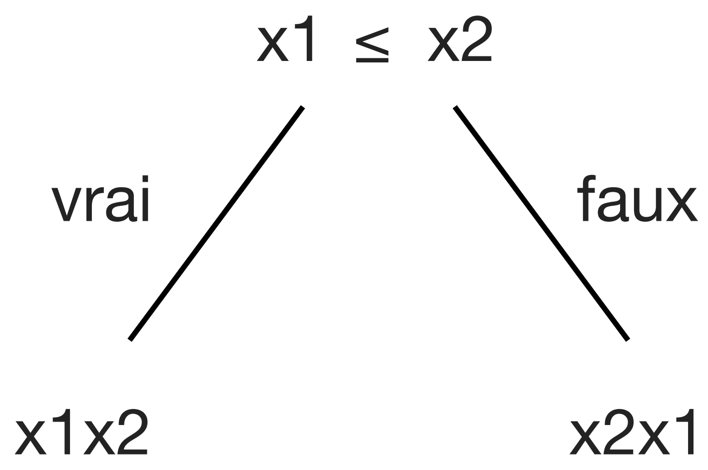
- **La complexité optimale est la borne inférieure des complexités dans le pire cas**

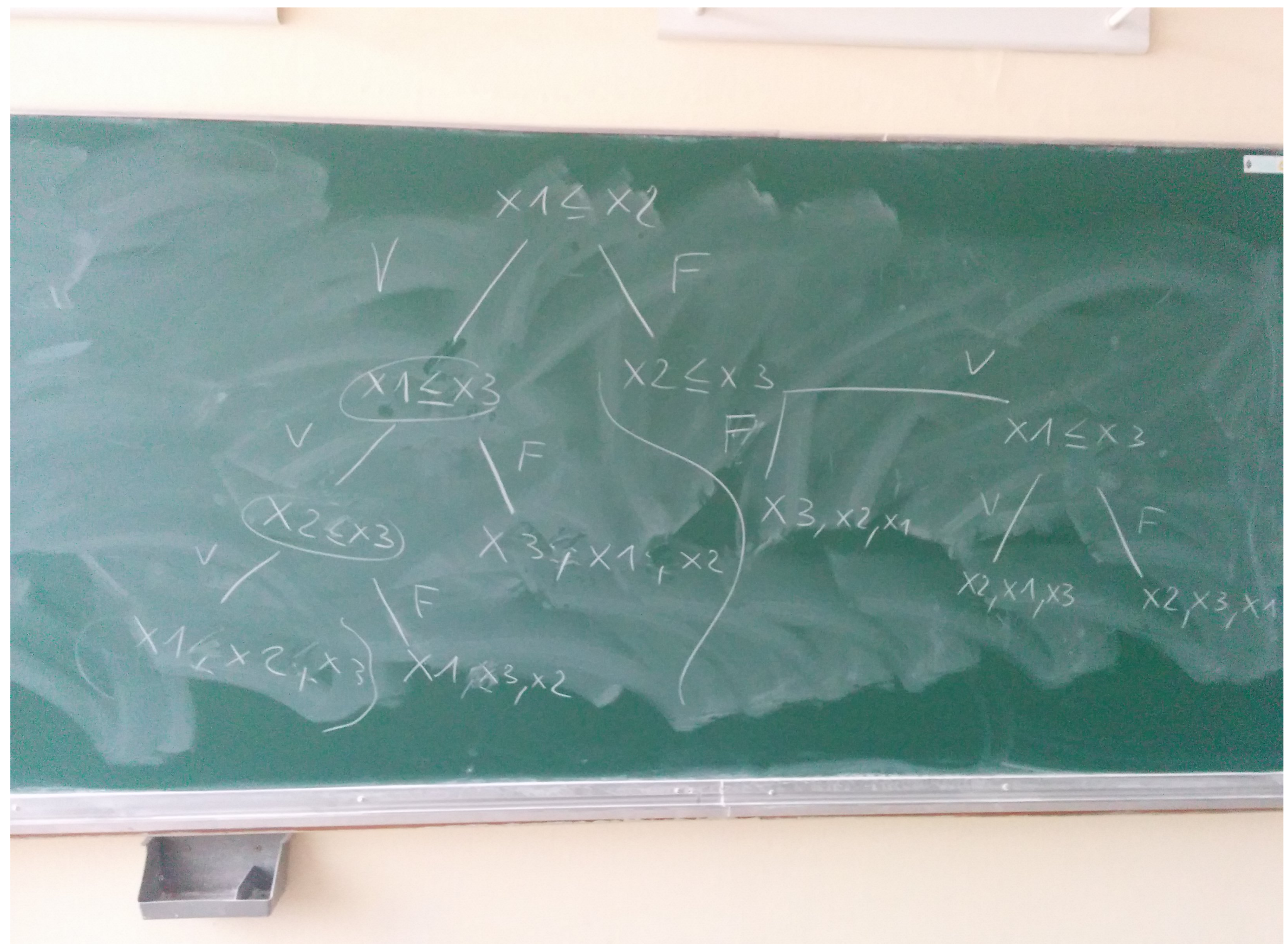
- **Hypothèse forte : On ne connait rien de plus que la spécification PREDORDT sur les n éléments à trier. La seule opération pour le classement est la comparaison _ ≤_ et on utilise l'échange pour le classement (ou le transfert en mémoire).**
  - unité d'évaluation : la comparaison _ ≤_ entre deux éléments
  - n éléments distincts

- **Un algorithme de tri détermine alors une permutation parmi les n! possibles**
- **Arbre de décision binaire (**localement complet**) :**
  - étiquette de comparaison en chaque noeud interne (2n! - 1) // feuilles -> noeud interne
  - permutation en chaque feuille (au minimum n! feuilles pour tous les cas possibles)

Arbre de décision

- **Exemple**
    - Soit un ensemble d'entiers {x1,x2,x3} à trier, n=3 -> 6 possibilités





UNIVERSITÉ DE STRASBOURG

Hauteur de l'arbre binaire de décision

- **Hauteur h minimale -> borne inférieure de complexité**
  - nombre maximum de tests le long d'une branche
- **h** $\geq \lfloor \log_2(2n! - 1) \rfloor = \lceil \log_2(2n!) - 1 \rceil = \lceil \log_2(n!) \rceil$
- **Avec la formule de Stirling, on obtient h ≥ nlogn**
  - $Max_{tri,opt}(n) \in \Omega(nlogn)$
  - cet ordre de grandeur est effectivement atteint pour certains tris (e.g., heapsort, mergsort), on a donc :
    - $Max_{tri,opt}(n) \in \Theta(nlogn)$
- $\mathbf{Moy_{tri,opt}(n) \in \Theta(nlogn)}$
- **On peut faire mieux (linéaire) si on lève l'hypothèse de comparaison en la remplacant par des hypothèses sur la structure/taille des éléments à trier.**

Tri par sélection : algorithme

- **A chaque étape, on extrait le minimum et on le range puis on recommence sans lui...**

- **Sélection ordinaire**
  - avec un simple tableau

- **`tri : Table -> Table`**
- **`tri(t) = t'`**
- **`avec (t',i) = init (t,0) tant que i <= n-2`**
  - **`répéter (ech(t',i,k),i+1)`**
  - **`avec (k,j) = init (i,i+1) tant que j <= n-1`**
    - **`répéter (si t'[j] < t'[k] alors j sinon k,j+1)`**

Tri par sélection : complexité

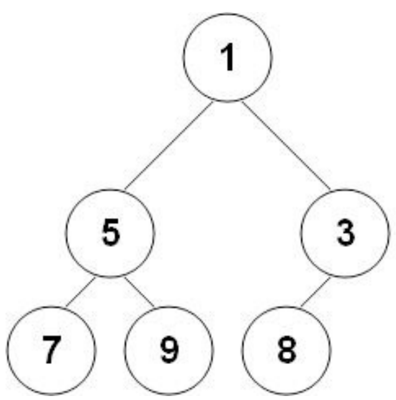
- **Nombre de comparaisons**
  - n(n-1)/2
- **Nombre d'échanges**
  - n-1
  - 3(n-1) transferts en mémoire

- **Complexité en temps maximum, minimum et moyen**

$$\Theta(n^2)$$

Tri par sélection : structures de données avancées

- **Sélection arborescente ou tri par la méthode tu tas**
  - J.W.J Williams puis améliorée par R.W. Floyd
- **Définition : un arbre binaire étiqueté est partiellement ordonné si tout noeud a une étiquette supérieure ou égale à celle de son père (s'il existe). Il s'agit d'un tournoi binaire parfait.**
- **En pratique, on peut utiliser une représentation contigue, un tableau, tel que :**
  - les deux fils d'un noeud i sont en position 2i + 1 et 2i + 2
  - le père d'un noeud i est en position $\lfloor (i - 1) / 2 \rfloor$



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

- $\lfloor n / 2 \rfloor$ **noeuds internes**
- $\lceil n / 2 \rceil$ **feuilles**

Tri par sélection (heapsort) : algorithmique

- **Deux phases :**
  - construction du tas (ct)
  - tri par sélection sur le tas (ts)

- `tri : Table -> Table`
- `tri(t) = ts(ct(t)) = ts(t')= t''`
  - `avec (i,t') = init(`$\lfloor n/2 \rfloor$`-1,t) tant que i >= 0`
    - `répéter (i-1, ins(t',i,n-1)) /* ct */`
  - `avec (i,t'') = init (n-1,t')`
    - `répéter (i-1, ins(ech(t',0,i),0,i-1)) /* ts */`

Sélection par échange - insertion tas binaire

- `ins(t,k,p) = // p : profondeur; k := indice`
  - `si p < 2k + 1 alors t // FIN (plus loin que prof)`
  - `sinon si p == 2k+1 // est-ce que p est mon fils ?`
    - `alors si t[k] > t[p] // père plus petit que fils ?`
      - `alors ech(t,k,p) // simple échange (c'est fini)`
      - `sinon t // rien à faire`
    - `sinon si t[k] > t[j] // p>2k+1 && test plus petit frère`
      - `alors ins(ech(t,k,j),j,p) //percolate down`
      - `sinon t // rien à faire`

    - `avec j = si t[2k+1] < t[2k+2] alors 2k+1 sinon 2k+2`

Tri par sélection : complexité

- **Nombre de comparaisons et d'échange pour l'insertion et la suppression**
  - $\Theta(\log_2 n)$
- **Construction du tas**
  - O(n) comparaisons ou échanges
- **Complexité en temps maximum, minimum et moyen**
  - $\Theta(n \log n)$
  - dominé par le tri à proprement dit...
  - on atteint la complexité optimale !

Tri par insertion

- **La $i^{ème}$ étape consiste à insérer le $i^{ème}$ éléments de la table parmi les i-1 précédents (déjà triés entre eux).**
- **Deux modes :**
  - Insertion séquentielle
    - on cherche à insérer t[i] à sa place dans i-1…0 avec des décalages à droite
    - complexité : $\Theta(n^2)$ (en transferts et en comparaisons)
  - Insertion dichotomique
    - l'insertion de t[i] se fait selon un procédé dichotomique
    - complexité : $\Theta(n^2)$ (lié aux transferts -décalages- et pas aux comparaisons)

  - Efficaces pour les petits ensembles déjà presque triés !

Tri par insertion

- **Insertion séquentielle**

- `tri : Table -> Table`
- `tri(t) = t'`
- `avec (i,t') = init (1,t) tant que i <= n-1`
    - `répéter (i+1, modt(t'',j+1,t'[i]))`
        - `avec (j,t'')= init (i-1,t') ttq j>=0 && t'[i] < t'[j]`
            - `répéter (j-1, modt(t'',j+1,t''[j]))`

Insertion séquentielle : deux versions en C

- **Avec tableau auxiliaire**

```
for(i=1; i < n; i++){
    tmp = t;
    for(j=i-1; j>=0 && t[i]<t[j]; j–)
        tmp[j+1] = tmp[j];
    tmp[j+1] = t[i];
    t = tmp;
                        }
```

- **Sans tableau temporaire**

```
for(i=1; i < n; i++){
    tmp = t[i];
    for(j=i-1; j>=0 && tmp<t[j]; j–)
        t[j+1] = t[j];
    t[j+1] = tmp;
                        }
```

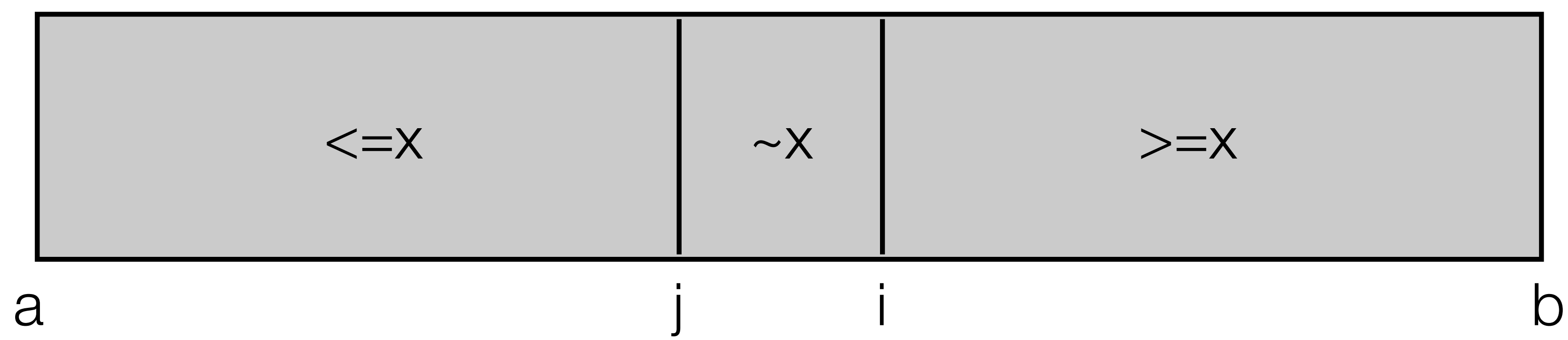
Tri par insertion

- **Insertion dichotomique**
- `tri: Table -> Table`
- `tri(t) = t'`
- `avec (i,t') = init(1,t) tant que i < n`
  - `répéter (i+1, modt(t'',j,t'[i]))`
    - `avec j = dich(t',0,i,t'[i])`
    - `avec (k,t'') = init (i,t') tant que j < k`
      - `répéter (k-1, modt(t'',k,t''[k-1]))`

- `dich(t,a,b,x) = si a == b alors a`
  - `sinon si t[(a+b)/2] < x alors dich (t,m+1,b,x)`
    - `sinon dich(t,a,m,x)`

Tri rapide (quicksort)

- **Principes (C.A.R. Hoare)**
  - A chaque étape, on sélectionne dans un intervalle [a,b] un élément pivot x par rapport auquel on place les autres éléments de l'intervalle



  - on recommence récursivement dans les tranches [a,j] et [i,b].
  - le pivot peut être choisi n'importe où...par exemple au milieu : x = t[(a+b)/2]
- **Complexité**
- Dans le pire cas : $\Theta(n^2)$
- En moyenne (là où il est considéré comme le meilleur) : $\Theta(n \log n)$

Tri rapide (quicksort)

- `qs: Table Ent Ent -> Table`
- `tri(t) = qs(t,0,n-1)`
- `qs(t,a,b) = si a < b`
    - `alors qs(qs(t',a,j),i,b)`
    - `avec (t',i,j) = placement(t,a,b)`
    - `sinon t`
- `placement(t,a,b) = (t',i,j)`
- `avec x = t[(a+b)/2] /* attention !..? */`
    - `(t',i,j) = init (t,a,b) tant que i <= j`
        - `répéter (si i' <= j' alors (ech(t',i',j'),i'+1,j'-1))`
            - `sinon (t',i',j')`
            - `avec i' = init i tant que x>t[i'] repéter i'+1`
            - `avec j' = init j tant que t[j']>x répéter j'-1`

UNIVERSITÉ DE STRASBOURG

Exercices de TD

- **De la spécification vers l'implémentation en C**
  - Sélection + insertion simple (+ QuickSort en projet)
- **Spécification**
  - BubbleSort + ShakerSort (+ MergeSort et ShellSort à la maison)
- **Ré-inventer CoutingSort (Tri par compteur)**
  - De la spécification à l'implémentation en C
    - clés à trier entre 0 et k-1 -> tableau de compteurs à k éléments
    - complexité O(n+k)

Bubble Sort : en itératif

- `tri_BS : Table -> Table`
- `tri_BS(t) = t' // n taille table`
- `avec (t',k) = init (t,1) tant que k > 0`
    - `répéter (t'',k') // «fausse instruction» !`
    - `avec (i,(t'',k')) = init (0,(t',0)) tant que i < n-1`
        - `répéter si t''[i] > t''[i+1] alors`
            - `(i+1, (ech(t'',t''[i],t''[i+1]), k'+1))`
            - `// changement`
            - `sinon (i + 1, (t'', k'))`
            - `// pas de changement -> arrêt ?`

Cocktail Sort : en récursif / itératif («optimisé»)

- `tri_CS(t) = CS(t,0,n-1,0,1) // a, b := extrémités`
- `CS(t,a,b,d,k) = si k == 0 ou a >= b alors t // d := direction`
    - `sinon si d == 0 alors CS(t',a,b-1,1,k') // mode droite`
    - `avec (i,t',k') = init (a,t,0) tant que i < b`
        - `répeter si t'[i] > t'[i+1]`
        - `(i+1, ech(t',t'[i],t'[i+1]), k'+1)`
        - `sinon (i+1,t',k')`
    - `sinon CS(t', a+1, b, 0, k') // mode gauche (d == 1)`
    - `avec (i,t',k') = init (b,t,0) tant que i > a`
        - `répeter si t'[i] < t'[i-1]`
        - `(i-1, ech(t',t'[i],t'[i-1]), k'+1)`
        - `sinon (i-1,t',k')`

Shaker Sort en C (modèle itératif)

- `Table tri_CS(Table t, n) {`
- `int k=1;`
- `int i,tmp=0; int a=0; int b=n-1;`
- `while (k > 0 ou a < b){`
  - `k = 0;`
  - `for (i=a;i<b;i++) // mode droite`
  - `{if(t[i] > t[i+1]){tmp=t[i]; t[i]=t[i+1]; t[i+1]=tmp; k++;}} b--;`
  - `if(k>0){k = 0;`
  - `for (i=b;i>a;i--) // mode gauche`
  - `{if(t[i] < t[i-1]){tmp=t[i]; t[i]=t[i-1]; t[i-1]=tmp; k++;}} a++;`
    - `}`
      - `}`
      - `}`

Tri par compteur

- `tri : Table -> Table`
- `tri(t) = t' // k := taille de l'intervalle de valeurs`
- `avec u = init tabnouv tant que i < n répéter modt(u,t[i],u[t[i]]+1)`
    - `(t',j,i) = init (t,0,0) tant que j < k // j index valeur`
        - `répéter (t'', j + 1, i + u[j])`
        - `avec (m, t'') = init (0, t') tant que m < u[j]`
            - `répéter (m+1, modt(t'', i + m, j))`

- Note : t, t' et t'' peuvent être confondus en pratique, **mais pas u** !

Tri par compteur en C

- `#define k taille_int_vals`
- `int u[k]; // + initialisation tabnouv`
- `int i,j,m;`

- `for (i=0;i<n;i++) // tableau de compteurs`
  - `u[t[i]]++;`

- `for (i=0,j=0; j<k; i+= u[j], j++)`
  - `for (m=0; m < u[j]; m++)`
    - `t[i+m] = j;`

UNIVERSITÉ DE STRASBOURG

Tri par compteur en pseudo code (complexité plus lisible)

- `# variables:`

```
#     input -- the array of items to be sorted; key(x) returns the key for item x
#     n -- the length of the input
#     k -- a number such that all keys are in the range 0..k-1
#     count -- an array of numbers, with indexes 0..k-1, initially all zero
#     output -- an array of items, with indexes 0..n-1
#     x -- an individual input item, used within the algorithm
#     total, oldCount, i -- numbers used within the algorithm

# calculate the histogram of key frequencies: // taille n
for x in input:
    count[key(x)] += 1

# calculate the starting index for each key: // taille k
total = 0
for i in range(k):    # i = 0, 1, ... k-1
    oldCount = count[i]
    count[i] = total
    total += oldCount

# copy to output array, preserving order of inputs with equal keys (stability): // taille n
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1

return output
```

Projet SDA 2 2015 : vers un tri efficace

- **Spécifier et implémenter plusieurs algorithmes de tris**
  - un tri efficace par sélection, i.e. Heapsort (B-Heap)
  - un tri efficace par insertion, i.e. B-Tree (ou AVL)
  - un tri du type « diviser pour mieux régner », e.g. MergeSort ou QuickSort
  - un tri de votre choix, e.g. hybride ou sans comparaison
- **Comparer les temps de calcul des différents algorithmes**
  - générer de grands tableaux de valeurs aléatoires (plusieurs par taille)
  - étude moyenne/médiane, min, max par taille
- **Améliorations générales :**
  - 1- étudier/justifier la complexité au pire et en moyenne de vos implémentations
  - 2- prouver vos algorithmes
  - 3- utiliser ces algorithmes dans un algorithme de parcours de graphe