PC 2
Algorithmes de tri et de recherche

- Algorithmes de tri
  - Schémas simples
  - Heapsort
- Algorithmes de recherche
  - Recherche séquentielle
  - Recherche dichotomique
  - Hachage

Algorithmes de tri

Donnée : N nombres entiers $a[0], a[1], \ldots, a[N-1]$.

Résultat : Réarrangement croissant $b[0], b[1], \ldots, b[N-1]$ des nombres $a[i]$.

Autre formulation : Recherche de la permutation π de $\{0, 1, \ldots, N-1\}$ telle que

$$a[\pi[i]] = b[i]$$

L'entier $\pi(i)$ est la place dans a du i-ème élément de b.
Exemple:

a	=	24	5	81	2	45
b	=	2	5	24	45	81
π	=	4	2	1	5	3

Tri par insertion

Principe: A la i-ème étape, insérer $a[i]$ parmi $a[0], a[1], \ldots, a[i-1]$.

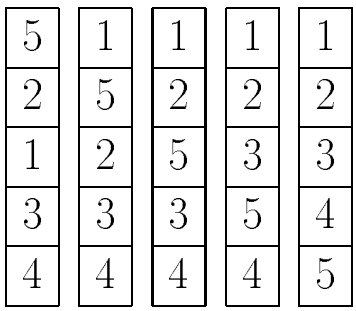
```
void TriInsertion(void) {
  int i,j,v;

  for (i=1; i< N; ++i) {
    v = a[i]; j = i;
    while (j>0 && a[j-1] > v) {
      a[j] = a[j-1];
      --j;
    }
    a[j] = v;
  }
}
```

Temps de calcul: $\Theta(n^2)$ au pire des cas et en moyenne.

Tri de la bulle

Principe: Faire venir en $a[i]$ l'élément minimum de $a[i], a[i+1], \ldots, a[N-1]$ par échange d'éléments adjacents (*les éléments légers montent*).




```
void TriBulle(void) {
  int i, j, t;
  for (i= 0; i< N; ++i)
    for (j= N-1; j> i; --j)
      if (a[j-1] > a[j]) {
        t= a[j-1]; a[j-1]= a[j]; a[j]= t;
      }
}
```

Complexité: $\Theta(n^2)$ au pire des cas et en moyenne.

Analyse en moyenne

On suppose que le tableau est une permutation π. Une *inversion* de π est une paire (i, j) d'indices telle que $i < j$ et $\pi[i] > \pi[j]$. Chaque échange de deux éléments de a par l'algorithme supprime exactement une inversion de π. Le nombre d'échanges est donc égal au nombre d'inversions de π.

Or le nombre moyen d'inversions d'une permutation[1] est

$$\frac{N(N-1)}{4}$$

d'où une complexité moyenne en $\Theta(N^2)$

[1] le nombre total d'inversions de π et de la permutation $\tilde{\pi}$ image miroir de π est $N(N-1)/2$

Tri par sélection

Principe: A la i-ème étape, on cherche le minimum de $a[i], a[i+1], \ldots, a[N-1]$ et on l'échange avec $a[i]$.

```
void Triselection(void) {
  int i, j, min, t;

  for (i= 0; i< N-1; ++i) {
    min= i;
    for (j= i+1; j< N; ++j)
      if (a[j]< a[min])
        min= j;
    t= a[min]; a[min]= a[i]; a[i]= t;
  }
}
```

Complexité: $\Theta(N^2)$ mais seulement $O(N)$ échanges.

Heapsort

On va maintenant montrer une méthode de tri plus efficace : le *tri sur un arbre*. C'est une méthode de tri sur place de complexité $O(n \log n)$.

On opère en deux étapes :

1. On fabrique un tas (heap), i.e. un tableau tel que $a[i/2] \geq a[i]$ pour $0 \leq i < N$. Pour cela, on tamise successivement $a[N-1], \ldots, a[0]$.
2. On extrait successivement le maximum $a[0]$ en l'échangeant avec $a[M]$ pour $M = N-1, N-2, \ldots, 0$ et en tamisant le nouveau $a[0]$.

Le tamisage de $a[k]$ consiste à échanger itérativement $a[k]$ avec le maximum de $a[2k]$ et $a[2k+1]$ jusqu'à ce que la condition des tas soit satisfaite.

Le tamisage

La fonction `Tamiser(k)` insère $a[k]$ parmi $a[k] \ldots a[M]$:

```
void Tamiser(int k) {

 int j, v;

  v = a[k];
  while (k< M/ 2) {
    j = 2*k+1;
    if (j<M && a[j+1] > a[j])
      ++j;
    if (v >= a[j]) break;
    a[k] = a[j]; k = j;
  }
  a[k] = v;
}
```

```
void Heapsort(void) {
  int k, t;

  M= N-1;
  for (k = N/ 2-1; k>=0; --k)
    Tamiser(k);
  do {
    t = a[0]; a[0]= a[M]; a[M]= t;
    M= M-1; Tamiser(1);
  }
  while ( M>= 1);
}
```

Exemple :

24	5	81	2	45
24	**45**	81	2	**5**
81	45	**24**	2	5
5	45	24	2	**81**
45	**5**	24	2	81
2	5	24	**45**	81
24	5	**2**	45	81
2	5	**24**	45	81
5	**2**	24	45	81
2	**5**	24	45	81

Recherche séquentielle

Recherche d'un entier v dans la table $a[0], a[1], \ldots, a[N-1]$. Il est commode d'utiliser une *sentinelle*:

```
int Recherche(int v) {
  int i=0;
  a[N]= v;
  while (v != a[i])
    ++i;
  return i;
}
```

Complexité: $\Theta(N)$ (avec $N/2$ opérations en moyenne).

En pratique: environ 1mn pour 1 Giga éléments (si on fait 10^7 comparaisons/s).

Recherche dichotomique

Hypothèse: La table est *ordonnée*: $a[1] < a[2] < \ldots < a[N]$

```
int Dicho (int v) {
  int i, g, d;
  g= 0; d= N-1;
  while (g <= d) {
    i=(g+d)/2;
    if (v = a[i]) return i;
    if (v < a[i]) d= i-1;
    else g= i+1;
  }
  return -1;
}
```

Complexité: $\Theta(\log N)$. Pour $N = 10^4$, on a 14 comparaisons environ au lieu de 5000 pour la recherche séquentielle.

Hachage

Principe: on utilise une fonction

$$h : E \to [0..N-1]$$

telle que $a[h(x)] = x$. La recherche devient immédiate.

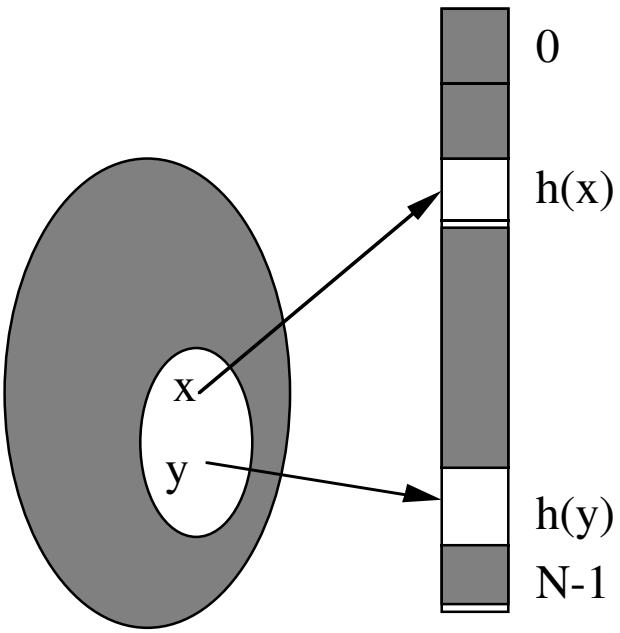


Figure 1:

Problèmes:

- Choix de la fonction de hachage. Par exemple,

$$h(x) = x \bmod N$$

- Gestion des *collisions*: $x \neq x'$ avec $h(x) = h(x')$. Deux solutions:

1. hachage ouvert: on gère des listes d'éléments de même clé.
2. hachage fermé: on utilise des valeurs de *remplacement*.

Hachage fermé

Principe: On utilise une suite de fonctions de hachage $h_1(x), h_2(x), \ldots$ pour ranger x. Par exemple:

$$h_i(x) = h(x) + i \bmod N$$

(hachage linéaire) ou encore

$$h_i(k) = h(k) + ih'(k) \bmod N$$

(hachage double).

On choisit h' de façon que N et $h'(k)$ soient toujours premiers entre eux (par exemple N est une puissance de 2 et $h'(k)$ est impair).

Programmes pour la recherche et l'insertion avec un hachage linéaire :

Recherche d'un élément :

```
int Recherche (int x) {
  int i;
  i= h(x);
  while (a[i] != VIDE) {
    if (a[i]==x) return i;
    i= (i+1) % N;
  }
  return -1;
}
```

Insertion d'un élément :

```
int Inserer(int x) {
  int i;
  i= h(x);
  while (a[i] != VIDE && x != a[i])
     i= (i+1) % N;
  a[i] = x;
  return i;
}
```

Analyse en moyenne du hachage fermé

Hypothèse: Le rehachage est *uniforme*: Les valeurs $h_1(x), h_2(x), \ldots$ sont successivement indépendantes et ont chacune une distribution uniforme.

Taux de remplissage: $\alpha = \text{card}(E)/N$.

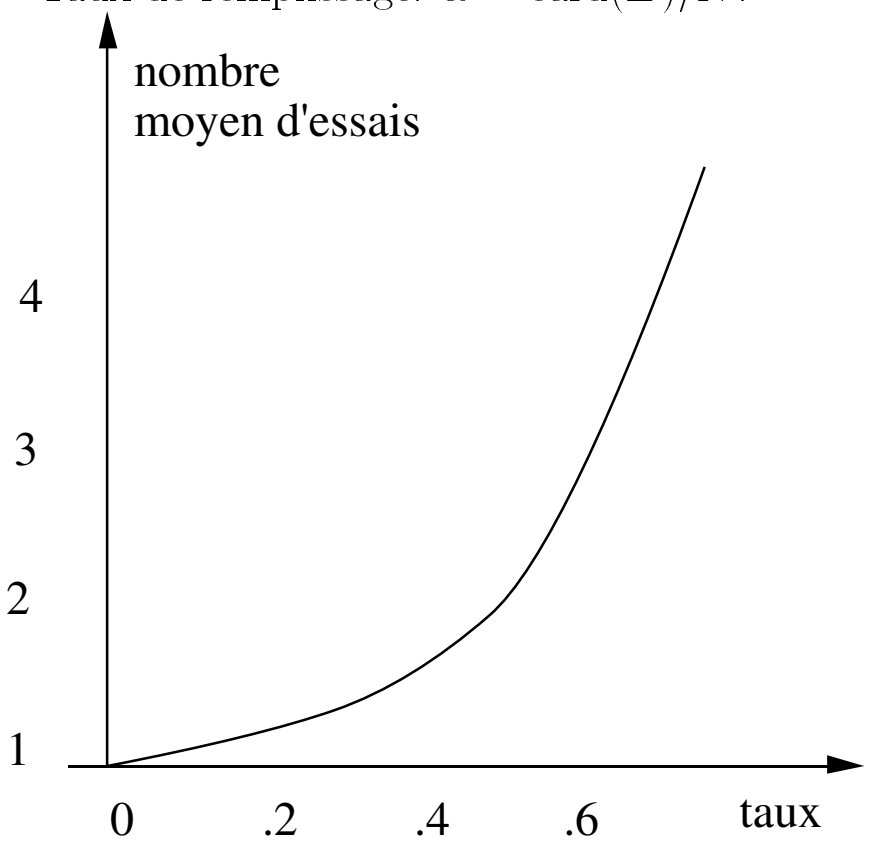


Figure 2:

Nombre moyen d'essais pour insérer un élément:

$$1 + \alpha + \alpha^2 + \ldots = \frac{1}{1-\alpha}$$