PC 2
Algorithmes de tri et de recherche

- Complexité des algorithmes
- Algorithmes de tri
- Algorithmes de recherche
  - Recherche séquentielle
  - Recherche dichotomique
  - Hachage

Complexité des algorithmes

Donnée de taille n

- Temps de calcul $T(n)$
- Espace mémoire utilisé $E(n)$

Evaluation:

- En moyenne
- Au pire des cas

Calcul à une constante multiplicative près: pas d'unité de temps ou d'espace.

Pour deux fonctions réelles positives f et g, on écrit

$$f = O(g)$$

si f ne croit pas plus vite que g :

$$\exists C, N \geq 0,\ \forall n \geq N,\ f(n) \leq Cg(n)$$

On écrit aussi

$$f = \Omega(g) \text{ pour } g = O(f)$$

et

$$f = \Theta(g) \text{ pour } f = O(g) \text{ et } g = O(f).$$

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:

$$\begin{array}{rcl} a & = & \begin{array}{|c|c|c|c|c|}\hline 24 & 5 & 81 & 2 & 45 \\ \hline\end{array} \\ b & = & \begin{array}{|c|c|c|c|c|}\hline 2 & 5 & 24 & 45 & 81 \\ \hline\end{array} \\ \pi & = & \begin{array}{|c|c|c|c|c|}\hline 4 & 2 & 1 & 5 & 3 \\ \hline\end{array} \end{array}$$

Complexité : $O(n^2)$ pour les schémas simples et $O(n \log n)$ pour les plus élaborés.

Tri par insertion

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

```
static void triInsertion() {
  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.

Programme complet

```
class TriInsertion {
  final static int N = 10;
  static int[] a = new int[N];
  static void initialisation() {
    for (int i= 0; i< N; i++)
      a[i] = (int) (Math.random() * 128);
  }
  static void impression() {
    for (int i= 0; i< N; i++)
      System.out.print(a[i]+ " ");
      System.out.println();
  }
  static void triInsertion() {...}
  public static void main(String[] args) {
    initialisation();
    impression();
    triInsertion();
    impression();
  }
}
```

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*).

5	1	1	1	1
2	5	2	2	2
1	2	5	3	3
3	3	3	5	4
4	4	4	4	5

```
static void triBulle() {
  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 raisonne sur la permutation π donnant la place du i-ème élément. 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 $N(N-1)/2$

Algorithmes de recherche

But : consultation et mise à jour d'un ensemble de données avec les fonctions de

- Recherche
- Insertion
- Suppression

On s'intéresse à des ensembles de N éléments avec N éventuellement très grand ($N \approx$ 1Giga-bit)

Recherche séquentielle

Recherche d'un entier v dans la table

$$a[0], a[1], \ldots, a[N-1]$$

```
static int Recherche(int v) {
  for (int i=0; i < N; i++)
    if (v == a[i]) return i;
  return -1;
}
```

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*:

$$a[0] < a[1] < \ldots < a[N-1]$$

```
static 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.

Dichotomies

Si, pour $n \geq 1$

$$T(n) \leq T(n/2) + a$$

alors

$$\boxed{T(n) = O(\log n)}$$

Démonstration : il suffit de vérifier par récurrence sur n que

$$T(n) \leq a \log n + b$$

avec $b = T(0)$.

Hachage

Principe: on utilise une fonction

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

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

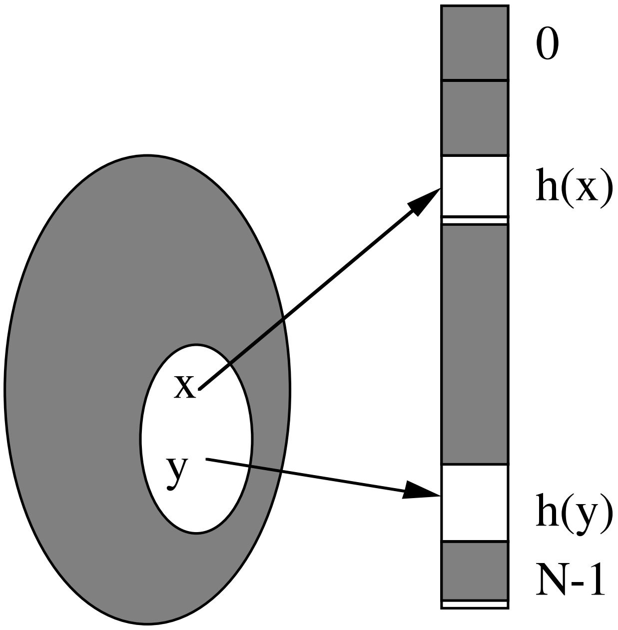

Figure 1:

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 :

```
static 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 :

```
static 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: $\alpha = \mathrm{card}(E)/N$.

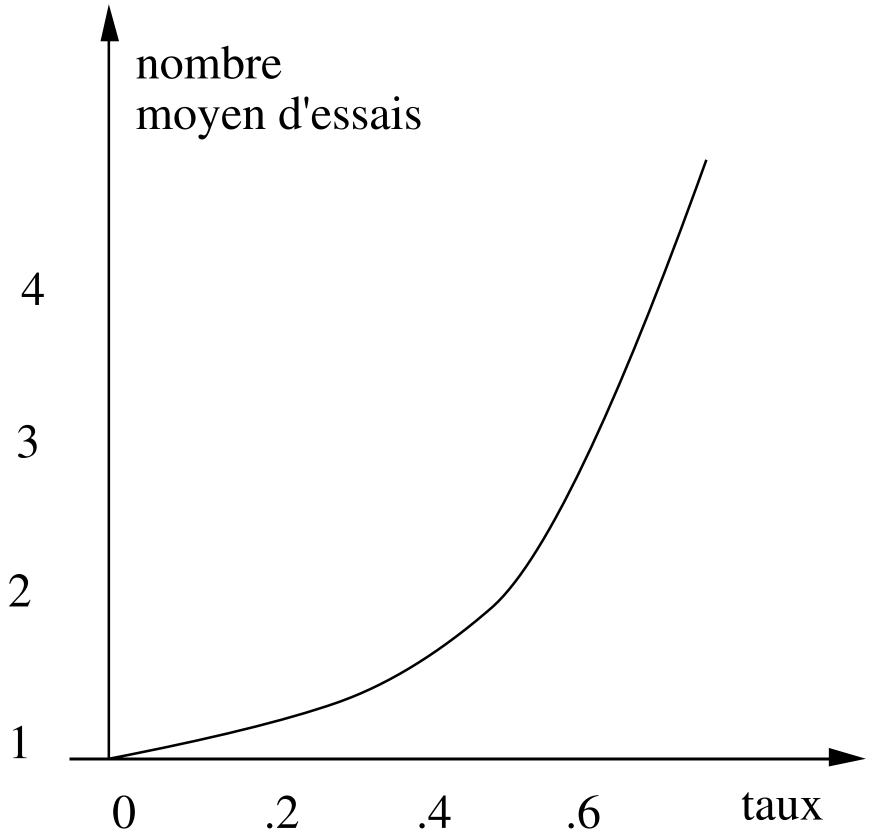

Figure 2:

Nombre moyen d'essais pour insérer un élément:

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