INF4100 Conception et analyse d'algorithmes

Introduction

Guy Tremblay

Hiver 2008

Aperçu

Problème de tri

- Problème = trier, en ordre croissant, une série d'items
- Solution bien connue = tri par fusion avec procédure récursive

Procédure de tri par fusion (*mergesort*)

```
procedure trier( ref int A[*], int n )
{
  trierRec( A, 1, n );
}

procedure trierRec( ref int A[*], int inf, int sup )
{
  if (inf != sup) {
    int m = (inf+sup)/2;

    trierRec ( A, inf, m   );
    trierRec ( A, m+1, sup );
    fusionner( A, inf, m, sup );
  }
}
```

```
procedure fusionner( ref int A[*],
                     int inf, int mid, int sup )
{
  int C[inf:sup];
  int i1 = inf,
      i2 = mid+1,
      i  = inf;

  while( i1 <= mid & i2 <= sup ) {
    if (A[i1] < A[i2]) {
      C[i++] = A[i1++];
    } else {
      C[i++] = A[i2++];
    }
  }

  for [k = i1 to mid] { C[i++] = A[k]; }
  for [k = i2 to sup] { C[i++] = A[k]; }

  A[inf:sup] = C[inf:sup];
}
```

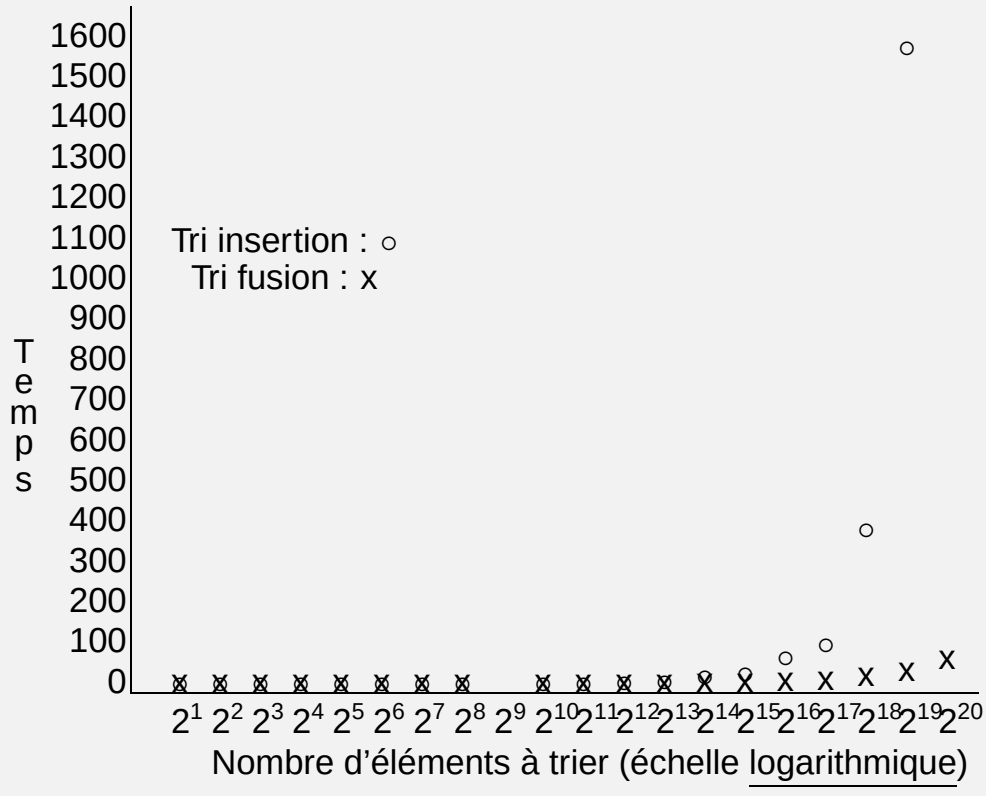
Procédure de tri par fusion (bis)

```
procedure trier( ref int A[*], int n )
{
  trierRec( A, 1, n );
}

procedure trierRec( ref int A[*], int inf, int sup )
{
  if (inf != sup) {
    int m = inf;

    trierRec ( A, inf, m   );
    trierRec ( A, m+1, sup );
    fusionner( A, inf, m, sup );
  }
}
```

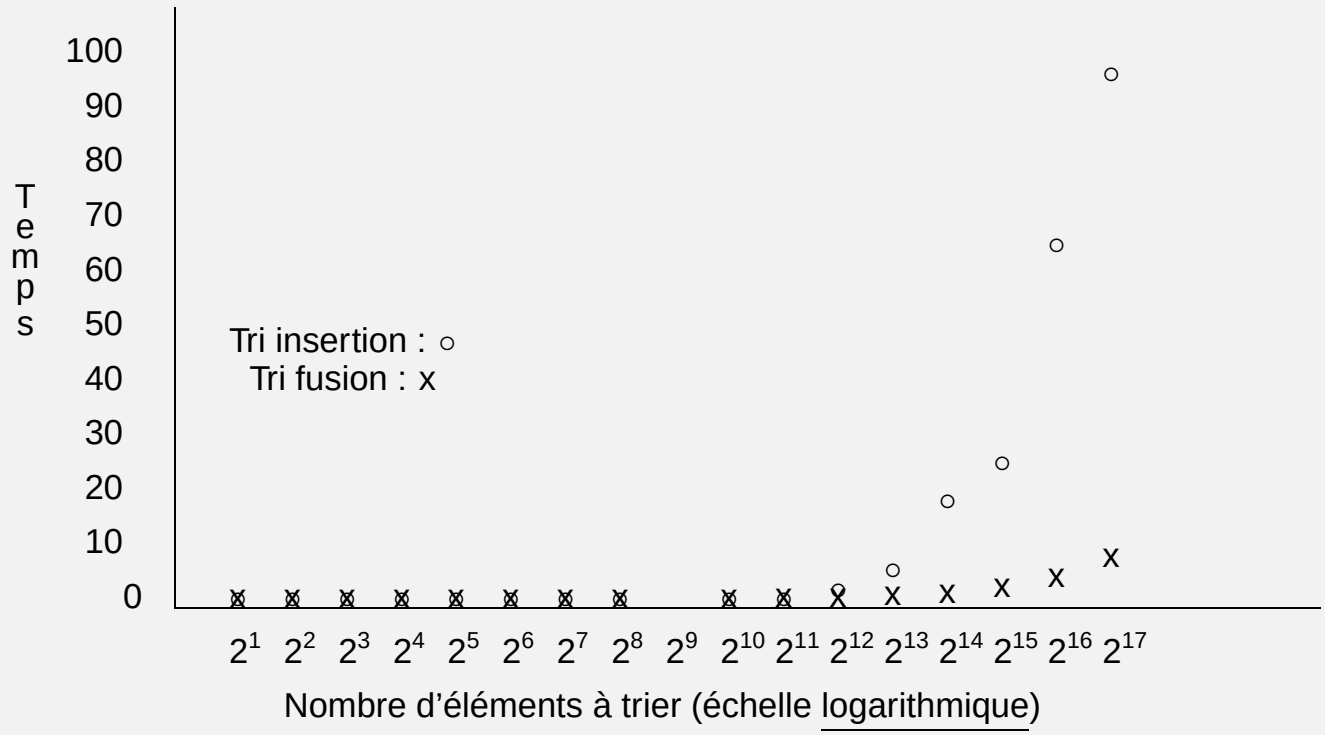
Procédure de tri par insertion

```
procedure trierInsertion( ref int A[*], int n )
{
  for [i = 2 to n] {
    int v = A[i];
    int j = i;
    while ( j > 1 & A[j-1] > v ) {
      A[j] = A[j-1];
      j = j-1;
    }
    A[j] = v;
  }
}
```

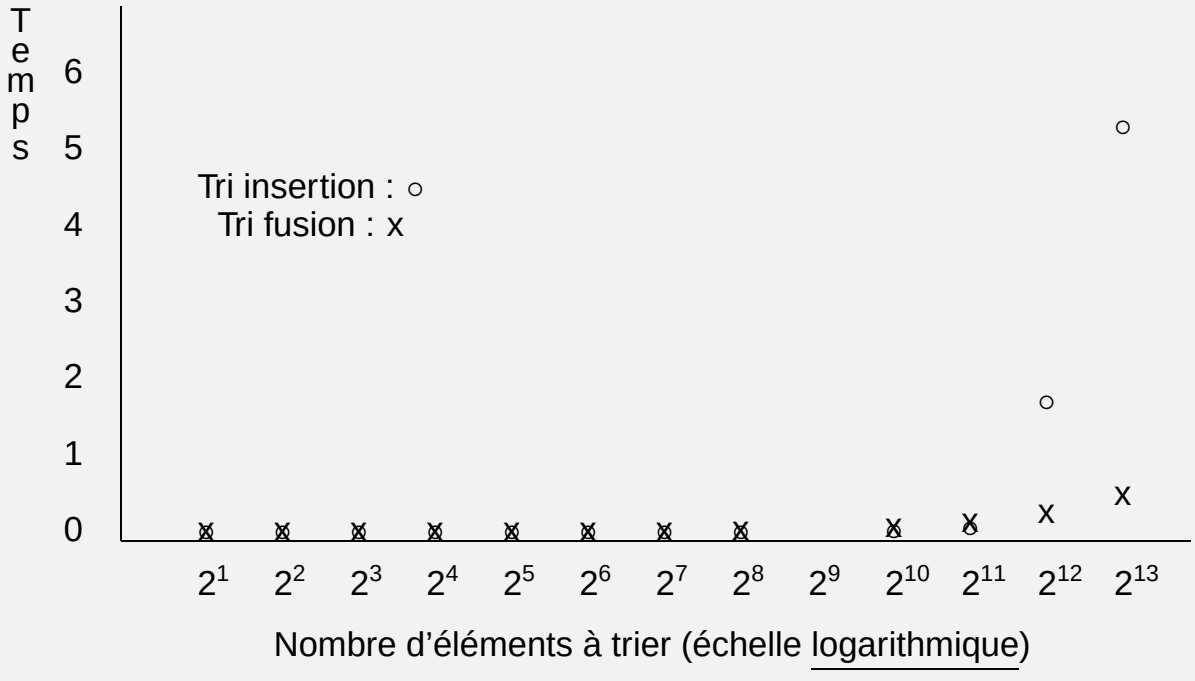
Temps d'exécution (en secondes) pour les deux procédures de tri



Temps d'exécution (en secondes) pour les deux procédures de tri



Temps d'exécution (en secondes) pour les deux procédures de tri



Analyse asymptotique

- $T_{tri\_fusion}(n) \in O(n \lg n)$
- $T_{tri\_fusion_{bis}}(n) \in O(n^2)$
- $T_{tri\_insertion}(n) \in O(n^2)$

Problème = nombres de Fibonacci

- Problème = Calculer le n-ième nombre de Fibonacci
- Suite de Fibonacci : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Solution naturellement récursive

Fibonacci récursif ordinaire

```
procedure fib( int n ) returns int r
{
   if (n <= 1) {
     r = n;
   } else {
     r = fib(n-1) + fib(n-2);
   }
}
```

Temps d'exécution (en minutes) pour les deux fonctions `fibonacci`

Valeur de n	**Fibo récursif**
1	0:00.00
5	0:00.00
10	0:00.00
15	0:00.00
20	0:00.00
25	0:00.00
30	0:00.08
35	0:00.98
40	0:11.01
45	2:01.62
50	22:31.95
55	_
60	_
65	_
70	_

Fibonacci récursif avec mémorisation

```
const int INDEFINI = low(int)

procedure fib_memo( int n, ref int A[0:*] )
                    returns int r
{
   if ( A[n] != INDEFINI) { r = A[n]; return; }

   if (n <= 1) {
     r = n;
   } else {
     r  = fib_memo(n-1, A) + fib_memo(n-2, A);
   }
   A[n] = r;
}

procedure fib( int n ) returns int r
{
  int A[0:n] = ([n+1] INDEFINI);
  r = fib_memo( n, A );
}
```

Temps d'exécution (en minutes) pour les deux fonctions `fibonacci`

Valeur de n	**Fibo récursif**	**Fibo avec mémorisation**
1	0:00.00	0:00.00
5	0:00.00	0:00.00
10	0:00.00	0:00.00
15	0:00.00	0:00.00
20	0:00.00	0:00.00
25	0:00.00	0:00.00
30	0:00.08	0:00.00
35	0:00.98	0:00.00
40	0:11.01	0:00.00
45	2:01.62	0:00.00
50	22:31.95	0:00.00
55	_	0:00.00
60	_	0:00.00
65	_	0:00.00
70	_	0:00.00

Analyse asymptotique

- $T_{fib}(n) \in \Omega(2^n)$
- $T_{fib\_memo}(n) \in \Theta(n)$

Problème = sac à dos

- Problème = Remplir un sac à dos avec une série d'items
- Contrainte = il ne faut pas dépasser la capacité du sac (poids maximum)
- Objectif = maximiser la valeur des items mis dans le sac

- Solution possible = on choisit en priorité l'item qui a le meilleur rapport bénéfice par unité de poids

Algorithme vorace pour le sac à dos

```
PROCEDURE BeneficeSac( capacite: nat;
                       pois, benefs: ARRAY[n] OF nat )
            RETOURNE ( set{nat}, nat )
DEBUT
   items ← Liste des items en ordre décroissant
                 de rapport bénéfice/unité de poids
   benef ← 0
   S ← {}
   w ← 0
   numItem ← 0
   TANTQUE w < capacite FAIRE
     numItem ← numItem + 1
     i ← items[numItem]
     S ← S U {i}
     w' ← min( poids[i], capacite-w )
     w ← w + w'
     benef ← benef + w' * benefs[i] / poids[i]
   FIN
   // On retourne l'ensemble des items choisis et le bénéfice total associé.
   RETOURNER ( S, benef )
FIN
```

Gloutonnerie payante ou non?

1. Sac à dos fractionnaire : La gloutonnerie produit toujours la solution optimale :)
2. Sac à dos 0–1 : La gloutonnerie ne produit pas toujours la solution optimale : (

Quelques éléments de conclusion

- La récursion est une stratégie fondamentale pour résoudre divers problèmes — *diviser-pour-régner*
- Pour qu'une solution récursive soit efficace, certaines contraintes doivent parfois être satisfaites :
  - Les sous-problèmes doivent être de même taille — décomposition équilibrée
  - La solution à un sous-problème ne doit être calculée qu'une seule fois
- Certains problèmes, mais pas tous, peuvent être résolus à l'aide d'algorithmes voraces — on prend ce qui semble le mieux à court terme
- Le temps d'exécution d'un algorithme peut être décrit de façon synthétique et succincte à l'aide de la notation asymptotique