Algorithmique Programmation Objet

Python

Andrea G. B. Tettamanzi

Université de Nice Sophia Antipolis

Département Informatique

andrea.tettamanzi@unice.fr

CM - Séance 6

Tableaux et matrices, recherche dichotomique

Plan

- Tableaux
- Matrices
- Recherche dichotomique

Tableaux

- Un tableau (*array* en anglais) est une structure de données de base qui est un ensemble d'éléments, auquel on accède à travers un numéro d'index.
- Le temps d'accès à un élément par son indice est constant, quel que soit l'élément désiré
- Les éléments d'un tableau sont contigus dans l'espace mémoire. Avec l'index, on sait donc à combien de cases mémoire se trouve l'élément en partant du début du tableau.
- On désigne habituellement les tableaux par des lettres majuscules. SI T est un tableau alors T[i] représente l'élément à l'index i.

Tableaux

- Avantages : accès direct au ième élément
- Inconvénients : les opérations d’insertion et de suppression sont impossibles
- sauf si on crée un nouveau tableau, de taille plus grande ou plus petite (selon l'opération). Il est alors nécessaire de copier tous les éléments du tableau original dans le nouveau tableau. Cela fait donc beaucoup d'opérations.

Tableaux

- Un tableau peut avoir une dimension, on parle alors de vecteur
- Un tableau peut avoir plusieurs dimensions, on dit qu'il est multidimensionnel. On le note T[i][k]
- La taille d'un tableau doit être définie avant son utilisation et ne peut plus être changée.
- Les seules opérations possibles sont set et get (on affecte un élément à un index et on lit un élément à un index).
- On peut linéariser un tableau à plusieurs dimensions :
  - Étant donné un tableau bidimensionnel T, $n \times m$,
  - Un peut construire un tableau linéaire L de taille nm, tel que $L[im + j] = T[i][j]$.

Tableaux multidimensionnels

i\ j	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

T[i][j]=L[i *5 +j]

T[1][3]=L[1*5+3]=L[8]

T[i][j]=[i * #col + j]

Tableaux de tableaux

- On peut définir un tableau de tableaux (ou de n'importe quoi en fait)

T[0]	0	1	2	3	4	5	6
T[1]	0	1	2	3			
T[2]	0	1	2	3	4	5	
T[3]	0	1	2				

Élément ou index ?

- Il ne faut pas confondre un élément du tableau et l'index de cet élément

Algorithmes de Tri

- Une tâche assez fréquente étant donné un tableau
- Permuter ses éléments de façon qu'ils respectent un ordre
- En fait, on peut trier n'importe quelle structure linéaire
- Un algorithme de tri simple mais pas trop performant est le tri par insertion
- On verra des autres algorithmes plus performants dans une des prochaines séances

Tri par insertion

- D F A G B H E C
- On fait 2 paquets : 1 non trié - 1 trié
- Un paquet d'un élément est trié
- On prend un élément non trié et on le range à sa place dans le paquet trié
- On prend D : D - F A G B H E C
- On prend F : D F - A G B H E C
- On prend A : A D F - G B H E C
- On prend G : A D F G - B H E C
- On prend B : A B D F G - H E C

Tri par insertion

- Comment ranger dans le paquet trié ?
- On doit ranger B dans A D F G – B H E C
- On a A D F G B
- On parcourt de droite à gauche A D F G
- Tant que la valeur est > B, on l'échange avec B
- A D F **B G**
- A D **B F** G
- A **B D** F G
- Fin

Tri par insertion

- On pourrait dire :
  - On fait 2 tas : un trié, puis un non trié.
  - Au début le tas trié est vide, celui non trié contient tous les éléments
  - On prend un élément non trié et on le range à sa place dans le tas trié
- Cet algorithme est exact, mais
  - sa formulation est ambiguë, par exemple « prendre » veut dire le supprimer du tas non trié aussi.
  - Il n'est pas assez précis : comment range-t-on un élément, qu'est-ce que sa place ?
  - Pour supprimer l'ambiguïté on va utiliser un pseudo langage

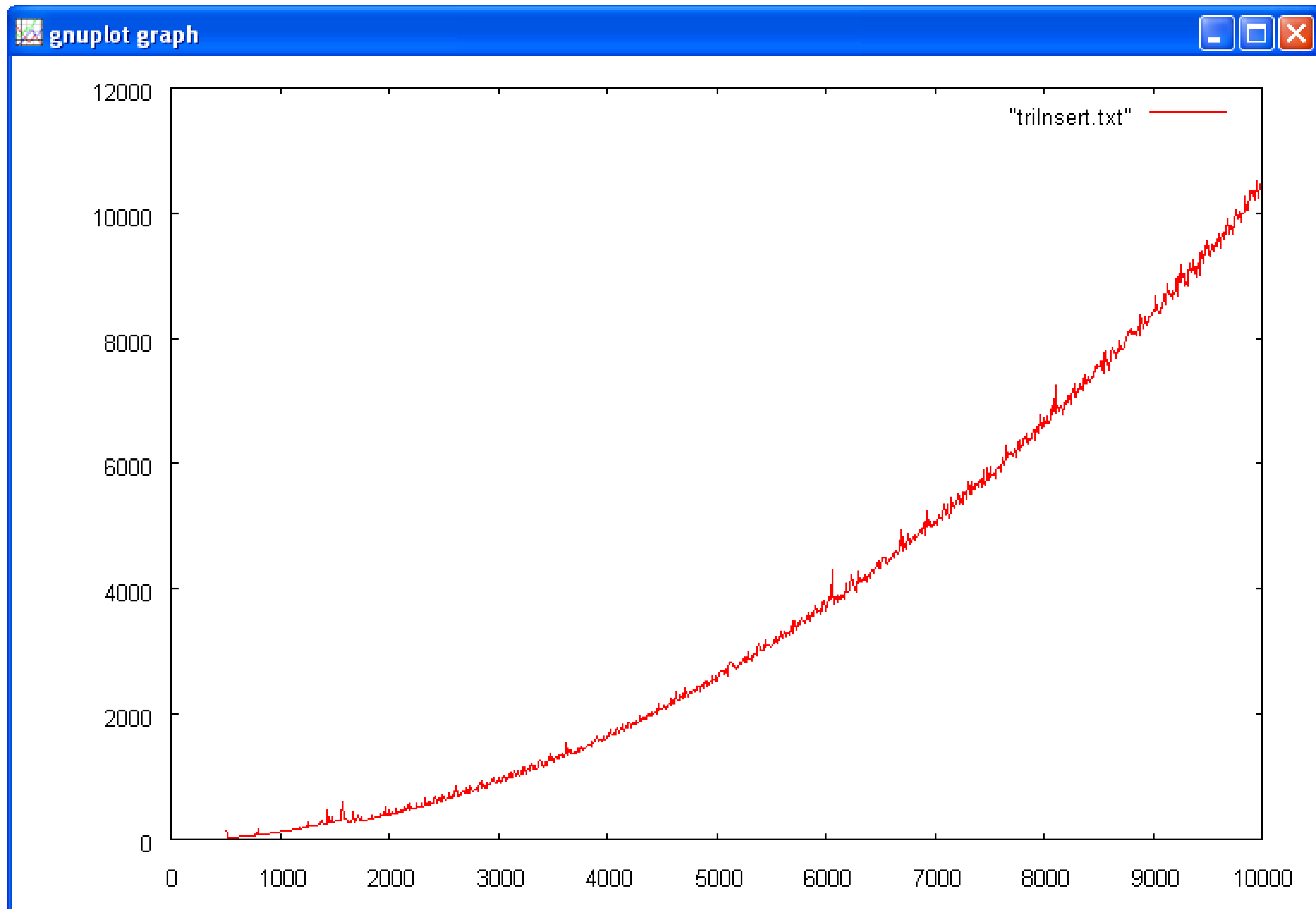
Tri par insertion : algorithme

```
triInsertion(entier[] t)
    pour i de 2 à n # n est la taille de t
      entier c ← t[i]
      entier k ← i-1
      # c doit être inséré dans le tableau ordonné t[1..i-1]
      # on cherche de droite à gauche la première valeur t[k]
      # plus petite que c
      tant que k≥1 et t[k]>c
        t[k+1] ← t[k] # on décale
        k ← k-1
      # on a trouvé la bonne place
      t[k+1] ← c
```

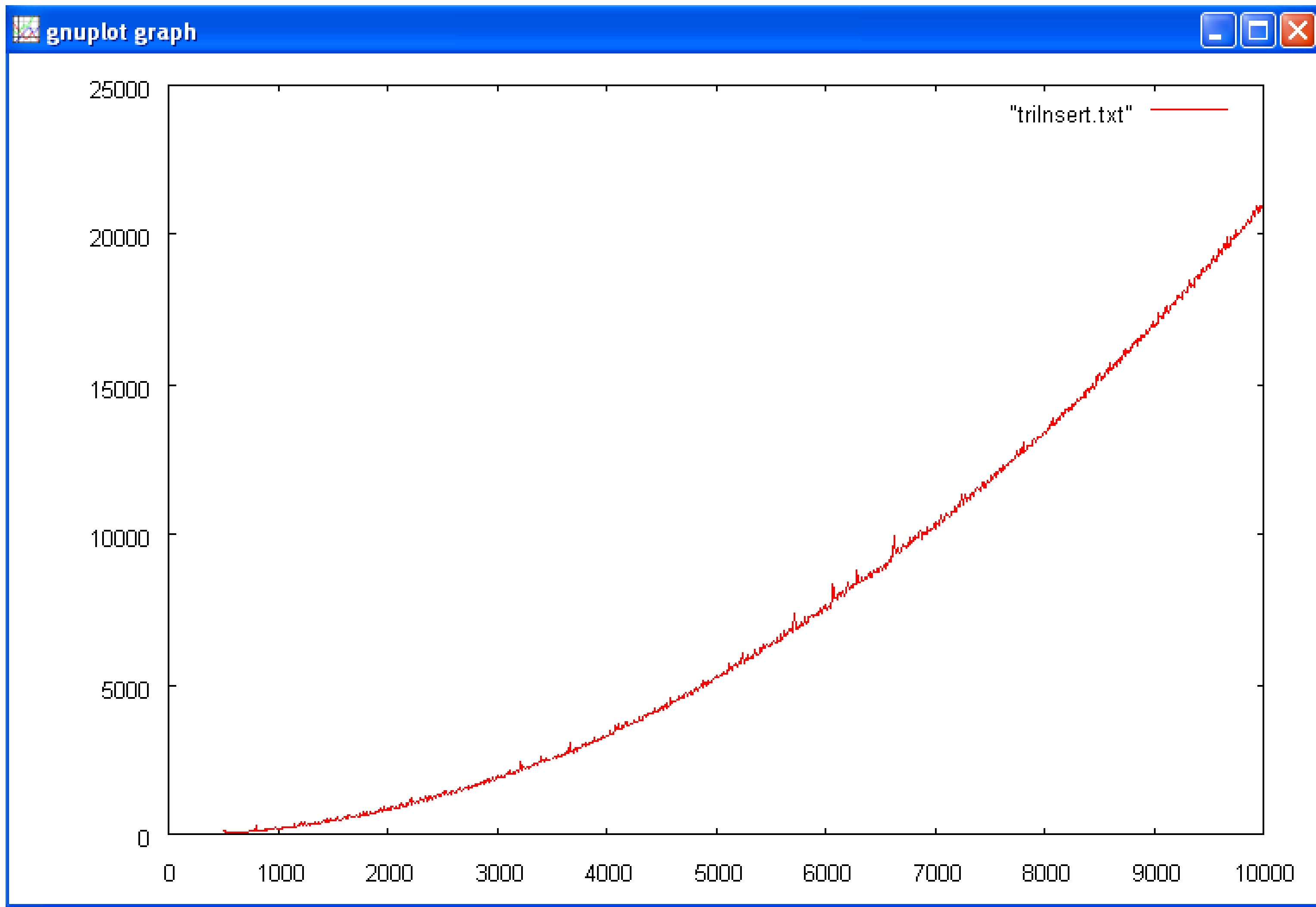
Tri par insertion : analyse empirique

- Francis Avnaim a fait l'étude suivante de la complexité expérimentale de cet algorithme, implémenté en Processing
- On trie des tableaux de taille croissante de 500 à 10000 par pas de 10
  - Aléatoires
  - Triés en ordre inverse
  - Déjà triés

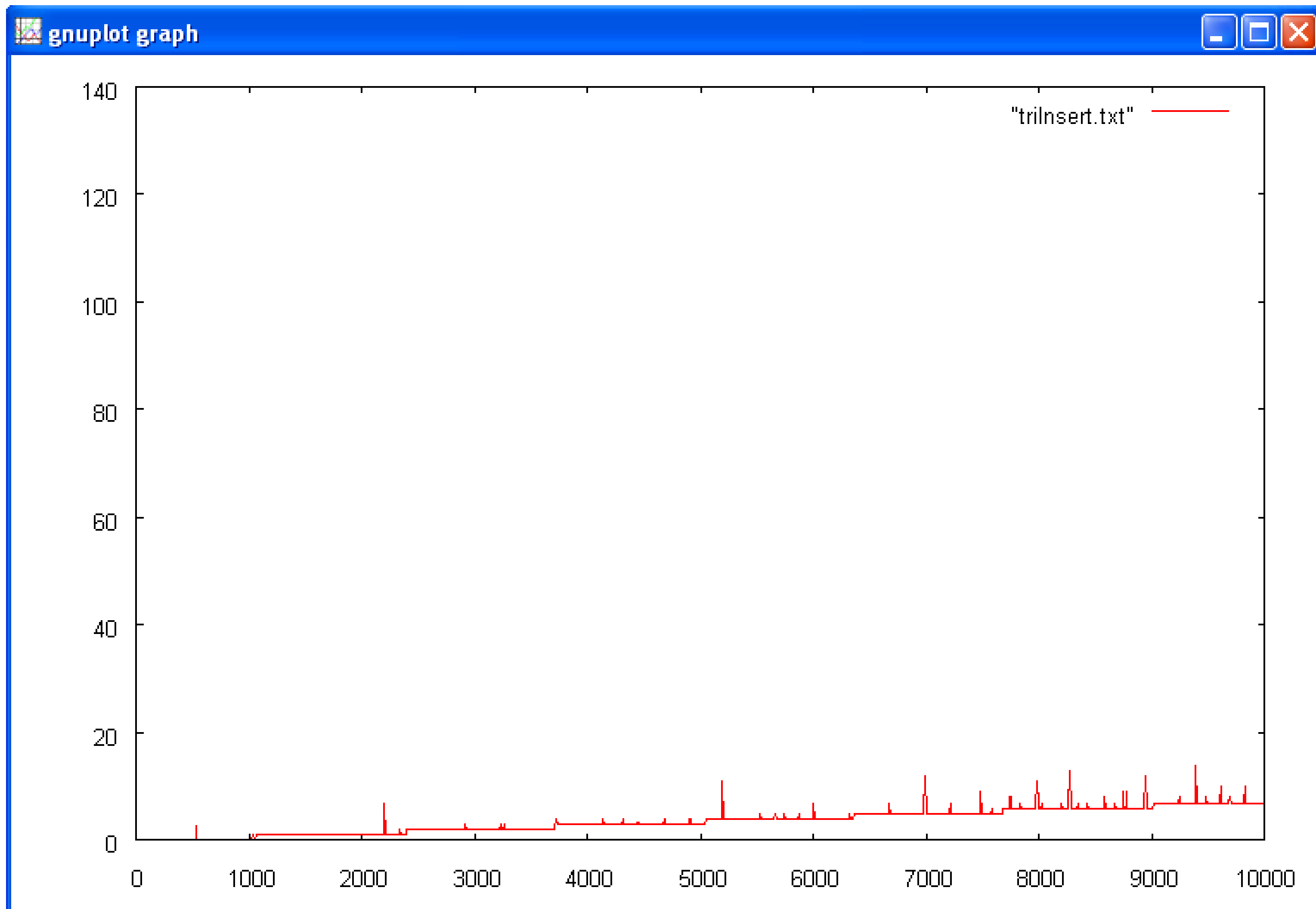
Aléatoire



En ordre inverse



Déjà trié



Tri par insertion : analyse empirique

- Conclusions de l'expérimentation
  - Tableaux aléatoires et en ordre inverse : $O(n^2)$?
  - Tableaux ordonnés : $O(n)$?

- Pour avoir une réponse fiable, il faut analyser l'algorithme...

Tri par insertion : analyse

- Parties bleues :
  - 7 op. él. n-1 fois soit 7(n-1)
- Partie jaune : 6 op. él.
- Partie mauve : 4 op. él.
- Combien de fois ?
- Nombre de fois ou les parties mauves et jaunes sont appelées ?
- Complexité mauve + jaune :
- Complexité totale inférieure à ?

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Tri par insertion : analyse

- Nombre de fois ou les parties mauves et jaunes sont appelées :
- au pire i fois (k de i-1 à 1), donc
  - Partie jaune : 6i à chaque fois
  - Partie mauve : 4i à chaque fois
- Complexité mauve + jaune:
  - 10i à chaque fois

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
        entier c ← t[i];
        entier k ← i-1;
        tant que (k≥1 et t[k]>c) {
            t[k+1] ← t[k];
            k ← k-1;
        }
        t[k+1] ← c;
    }
}
```

Tri par insertion : analyse

- Complexité mauve + jaune :
  - 10i à chaque fois
- Partie bleue : 7(n – 1)

$$7(n-1) + 10\sum_{i=2}^{n} i = \ldots$$

$$7(n-1) + 10 \cdot \frac{(n-1)n}{2} = O(n^2)$$

(pire cas)

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
        entier c ← t[i];
        entier k ← i-1;
        tant que (k≥1 et t[k]>c) {
            t[k+1] ← t[k];
            k ← k-1;
        }
        t[k+1] ← c;
    }
}
```

Si tableau déjà trié, partie jaune jamais exécutée : 11(n – 1)
(meilleur cas)

Diviser pour régner : calcul de x^n

- Méthode simple, par itérations :
  - Y ← 1
  - pour i de 1 à n : y ← y * x
- Complexité : O(n)
  - x^8 ?
  - Combien d'opérations ?
  - x^{16} ?
  - x^{12} ?
- Généralisation :
  - si n est pair
  - si n est impair

Diviser pour régner : calcul de x^n

- Algorithme plus performant, récursif :

```
PUISSANCE(↓x, entier ↓n, ↑resultat)
  si n = 1
    resultat ← x
  sinon
    PUISSANCE(x, n/2, p)
    si n pair
      resultat ← p*p
    sinon
      resultat ← p*p*x
```

Nombre de multiplications : log n + [nombre de bits à 1 dans n] – 1

Complexité : O(log n)

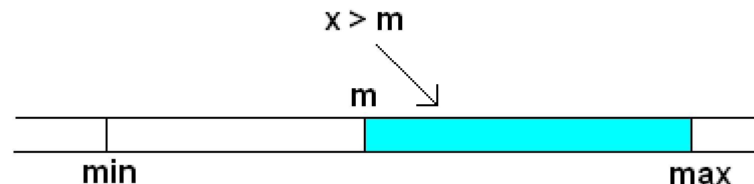
Recherche dans un tableau

- Considérons un tableau T de nombres entiers, de taille n
- On désire savoir si un nombre donné x est dans le tableau
- Pour le savoir, on parcourt le tableau en comparant les éléments de T à x (recherche séquentielle)
- Au plus (quand x n'est pas dans T), on fera n comparaisons
- L'algorithme est donc de complexité O(n)

Recherche dichotomique

- Peut-on faire mieux ?
- Oui, si le tableau est préalablement trié. C’est la recherche dichotomique

- Idée : on compare x à la valeur centrale m de T.
- Si x = m, on a trouvé x dans T.
- Sinon, si x < m, on cherche x dans la moitié inférieure du tableau, sinon on cherche x dans la moitié supérieure



Recherche dichotomique : algorithme

```
entier rechercheDicho(entier a[min..max], entier x) {
    si (min > max) {
        retourner -1;
    }
    sinon {
        entier i ← (min + max) / 2;
        entier m ← a[i];
        si (x = m)
            retourner i;
        sinon si (x < m)
            retourner rechercheDicho(a[min..i-1], x);
        sinon
            retourner rechercheDicho(a[i+1..max], x);
    }
}
```

Recherche dichotomique : analyse

- Appelons $C(n)$ le nombre d'opérations élémentaires exécutées pour effectuer la recherche dichotomique dans un tableau de taille n
- On a, avec k1 et k2 deux constantes majorant respectivement le nombre d'opérations des parties en bleu et en mauve :

$$
\begin{aligned}
C(n) &\leq (k1 + k2) + C(n/2) \\
&\leq 2\,(k1 + k2) + C(n/2^2) \\
&\leq 3\,(k1 + k2) + C(n/2^3) \\
&\quad \ldots \\
&\leq r\,(k1 + k2) + C(1) \\
&\leq r\,(k1 + k2) + k1 \leq 2r\,(k1+k2)
\end{aligned}
$$

Recherche dichotomique : analyse

On obtient ce qu'on appelle “équation de récurrence” :

$$t(n) = \begin{cases} k_1 + k_2 + t(n/2), & \text{si } n > 1 \\ k_1, & \text{sinon} \end{cases}$$

Il existe une méthode générale pour résoudre ce genre d'équation, basée sur les fonctions génératrices. Cependant, cette méthode est assez compliquée et nous n'avons pas le temps de la couvrir. Dans ce cas particulier, il est facile de deviner la solution :

$$t(n) = O(\log n)$$

Recherche dichotomique : version itérative

```
entier rechercheDichotomique(entier T[1..n], entier x)
min ← 1 ; max ← n
tant que min ≤ max
  m ← (min + max)/2
  si T[p] ≤ x
    min ← m + 1
  si T[p] ≥ x
    max ← m – 1
si T[m] = x
  renvoyer m
sinon
  renvoyer -1  # élément non trouvé
```

Merci de votre attention