Module EA4 – Éléments d'Algorithmique II

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Complexité et ordres de grandeur

Si un algorithme (ou plutôt un programme) met 1 centième de seconde à traiter les entrées de taille 10, alors pour une entrée de taille 1000, en fonction de sa complexité, il mettra...

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$0,03s$	$1s$	$3s$	$100s$	$10\,000s$	$10^{288}ans$

Complexité et ordres de grandeur

Si un algorithme (ou plutôt un programme) met 1 centième de seconde à traiter les entrées de taille 10, alors pour une entrée de taille 1000, en fonction de sa complexité, il mettra...

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$0,03s$	$1s$	$3s$	$100s$	$10\,000s$	$10^{288}ans$

Autre manière de voir les choses : en une heure, ce programme peut traiter des entrées de taille...

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
10^{360000}	$3\,600\,000$	$600\,000$	6000	710	28

`recherche(x, L)`

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

recherche(x, L)

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

```
def recherche_lineaire(x, L) : # ou recherche séquentielle
  for elt in L :
    if elt == x : return True
  return False
```

Recherches dans une liste

recherche(x, L)

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

```
def recherche_lineaire(x, L) : # ou recherche séquentielle
  for elt in L :
    if elt == x : return True
  return False
```

(remarque : c'est très exactement ce que fait le test `(x in L)`)

Recherches dans une liste

recherche(x, L)

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

```
def recherche_lineaire(x, L) : # ou recherche séquentielle
  for elt in L :
    if elt == x : return True
  return False
```

(remarque : c'est très exactement ce que fait le test `(x in L)`)

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

variante : retourner une position où x apparaît

```
def recherche_lineaire(x, L) :
  for (i, elt) in enumerate(L) :
    if elt == x : return i
  return -1
```

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

variante : retourner une position où x apparaît

```
def recherche_lineaire(x, L) :
  for (i, elt) in enumerate(L) :
    if elt == x : return i
  return -1
```

(remarque : c'est très exactement ce que fait L.index(x))

`occurrences(x, L)`

Étant donné une liste `L` et un élément `x`, compter les occurrences de `x` dans `L`

occurrences(x, L)

Étant donné une liste `L` et un élément `x`, compter les occurrences de `x` dans `L`

```
def occurrences(x, L) :
  res = 0
  for (i, elt) in enumerate(L) :
    if elt == x : res += 1
  return res
```

occurrences(x, L)

Étant donné une liste `L` et un élément `x`, compter les occurrences de `x` dans `L`

```
def occurrences(x, L) :
  res = 0
  for (i, elt) in enumerate(L) :
    if elt == x : res += 1
  return res
```

(remarque : c'est très exactement ce que fait `L.count(x)`)

max(L)

Étant donné une liste L contenant des éléments comparables, déterminer le plus grand élément qui apparaît dans L

max(L)

Étant donné une liste L contenant des éléments comparables, déterminer le plus grand élément qui apparaît dans L

```
def max(L) :
  tmp = L[0]
  for elt in L :
    if elt > tmp : tmp = elt
  return tmp
```

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- affectations, incrémentations de compteurs

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- affectations, incrémentations de compteurs

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- affectations, incrémentations de compteurs

`max(L)` $\Longrightarrow$ $n - 1 = \Theta(n)$ comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- affectations, incrémentations de compteurs

`max(L)` $\Longrightarrow n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow n=\Theta(n)$ comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- affectations, incrémentations de compteurs

`max(L)` $\Longrightarrow n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow n=\Theta(n)$ comparaisons

`recherche_lineaire(x, L)` $\Longrightarrow$ entre 1 et n comparaisons

$\Longrightarrow \Theta(n)$ comparaisons au pire

$\Longrightarrow \dfrac{n+1}{2}=\Theta(n)$ en moyenne dans le cas *favorable*

$\Longrightarrow \Theta(n)$ comparaisons en moyenne

Complexité de ces recherches

Peut-on faire mieux que $\Theta(n)$?

Recherche dans un tableau *trié*

`recherche(x, T)`

Étant donné un tableau `T` *trié* et un élément `x`, déterminer si `x` apparaît dans `T`

on peut alors faire beaucoup plus efficace :

Recherche dans un tableau *trié*

recherche(x, T)

Étant donné un tableau `T` *trié* et un élément `x`, déterminer si `x` apparaît dans `T`

on peut alors faire beaucoup plus efficace :

```
def recherche_dichotomique(x, T) :
  if len(T) == 0 : return False
  n = len(T)//2
  if x == T[n] : return True
  elif x < T[n] : return recherche_dichotomique(x, T[:n])
  else : return recherche_dichotomique(x, T[n+1:])
```