Cours d'algorithmique et programmation 2

Anne Parrain

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Section 1

La dichotomie

Chercher une valeur dans une liste

Comment faire ?

- parcourir toute la liste
- si la liste est triée, effectuer une recherche dichotomique

Idée : Prendre la valeur au milieu de la liste et comparer avec la valeur cherchée pour savoir

C'est la même idée que pour jouer au nombre mystérieux.

À vous !

L'implémentation de la recherche dichotomique

```
1  def recherche(l,val) :
2      '''liste(int), int -> int
3      retourne l'indice de val si val est presente dans l.
4      retourne -1 sinon.
5      '''
6      deb = 0
7      fin = len(l)
8      while deb < fin :
9          milieu = (deb+fin)//2
10         if l[milieu] == val :
11             return milieu
12         elif l[milieu] < val :
13             deb = milieu+1
14         else :
15             fin = milieu
16     return -1
```

L'implémentation de la recherche dichotomique (2)

et en récursif ? À vous !

```
1     def recherche_rec(l,val,deb,fin) :
2     '''liste(int), int, int, int -> int
3     retourne l'indice de val si val est presente dans l
4     entre l'indice deb inclus et fin exclu.
5     retourne -1 sinon.
6     '''
7     if deb < fin :
8         milieu = (deb+fin)//2
9         if l[milieu] == val :
10            return milieu
11        elif l[milieu] < val :
12            return recherche_rec(l, val, milieu+1, fin)
13        else :
14            return recherche_rec(l, val, deb, milieu)
15    else :
16        return -1
```

avec un appel initial : `recherche_rec(l,val,0,len(l))`

Quelle efficacité ? Combien d'opérations ?

Pour étudier la complexité d'un algorithme, on va s'attacher

- à compter le nombre d'opérations à effectuer,
- souvent en fonction de la taille de la donnée.

On s'intéressera surtout aux ordres de grandeur.

On étudie essentiellement :

- la complexité dans le pire des cas (c'est le plus facile à faire !)
- la complexité en moyenne

Complexité de la recherche d'une valeur dans une liste

- Dans le cas d'une recherche linéaire, i.e. on parcourt toute la liste élément après élément pour essayer de trouver la valeur : au pire n accès et n comparaisons.

  La complexité de la recherche par parcours simple est en $O(n)$ ou de complexité linéaire.

Nombre d'éléments	Nombre de comparaisons
4	4
10	10
50	50
10 000	10 000
100 000	100 000

Complexité de la recherche d'une valeur dans une liste

- Dans le cas d'une recherche dichotomique : à chaque étape, on divise l'espace de recherche par deux $\Rightarrow$ le nombre d'étapes (accès, comparaison) est l'entier immédiatement supérieur à $log_2(n)$.

  La complexité de la recherche dichotomique est en $O(log(n))$ ou de complexité logarithmique.

Nombre d'éléments	Nombre de comparaisons
4	2
10	4
50	6
10 000	14
100 000	17

Section 2

Les tris

Un premier tri : le tri par sélection

Idée :

- on cherche le plus petit élément de la liste
- on l'insère en début de la liste en permutant sa valeur avec l'élément en tête
- et on recommence

À vous :

- la recherche du plus petit élément
- le tri par sélection

Le tri par sélection - implémentation

```
1  def tri_selection(l) :
2      '''(inout) liste(int) -> rien
3      algo de tri par selection du plus petit element
4      '''
5      for i in range(len(l)) :
6          # recherche du plus petit element
7          ind_mini = recherche_mini(l,i)
8          # permutation a la position i
9          l[i], l[ind_mini] = l[ind_mini], l[i]
10     # fin for
11 # fin tri_selection
```

Le tri par sélection - implémentation (2)

```
1  def recherche_mini(l,ind) :
2      '''liste(int), int -> int
3      recherche l'indice du plus petit element de l
4      qui se trouve entre l'indice ind et la fin de l.
5      '''
6      assert ind < len(l)
7      mini = ind
8      for i in range(ind,len(l)) :
9          if l[i] < l[mini] :
10             mini = i
11     return mini
```

Complexité du tri par sélection

On note n le nombre d'éléments de la liste.
Pour chaque élément d'indice i de la liste, on va faire :

- parcourir n-i éléments restants de la liste
- faire n-i comparaison pour trouver le plus petit élément
- et au pire faire n-i affectations
- plus deux affectations pour permuter l'élément i et le plus petit élément

Soit n(n+1)/2 comparaisons et au pire n(n+1)/2 + 2n affectations.

On simplifie en gardant l'ordre de grandeur : ici, le degré du polynôme.

On dira que le tri par sélection est en $O(n^2)$,
ou qu'il est de complexité quadratique.

Peut-on améliorer le tri par sélection ?

Oui ! Un peu. . .

Idée :

- en même temps qu'on cherche le plus petit élément, on cherche le plus grand
- et à chaque tour on range le plus petit et le plus grand à leur place

Pas de changement pour l'ordre de grandeur de la complexité : toujours quadratique.

Mais la fonction des comparaisons croît moins vite que dans le tri précédent. À vous !

Le tri par sélection - amélioration (1)

```
1  def recherche_min_max(l,deb,fin) :
2      '''liste(int), int, int -> int, int
3      recherche l'indice du plus petit element
4      et du plus grand element de l
5      qui se trouvent entre les indices deb et fin de l.
6      '''
7      assert deb < fin
8      mini = deb
9      maxi = fin - 1
10     for i in range(deb,fin) :
11         if l[i] < l[mini] :
12             mini = i
13         if l[i] > l[maxi] :
14             maxi = i
15     return mini, maxi
```

Le tri par sélection - amélioration (2)

```
1  def tri_selection_min_max(l) :
2      '''(inout) liste(int) -> rien
3      algo de tri par selection du plus petit element
4      et du plus grand element
5      '''
6      prem = 0
7      dern = len(l)
8      while prem < dern :
9          # recherche du plus petit element
10         ind_mini,ind_maxi = recherche_min_max(l,prem,dern
11         # positionnement du plus petit a sa place
12         l[prem], l[ind_mini] = l[ind_mini], l[prem]
13         prem += 1
14         # positionnement du plus grand a sa place
15         dern -= 1
16         l[dern], l[ind_maxi] = l[ind_maxi], l[dern]
17     # fin for
18 # fin tri_selection
```

Le tri à bulles

Le tri à bulles remonte le plus grand élément vers sa place à chaque tour de liste.

L'idée du tri à bulles est la suivante :

On peut profiter de cette recherche du maximum
pour remonter un peu vers le haut les plus grands éléments.

Implémentation du tri à bulles

```
1 def tri_a_bulles(l) :
2     '''(inout) liste(int) -> rien
3     trie l par l'algorithme du tri a bulles
4     '''
5     for i in range(len(l)) :
6         ## on remonte une bulle tout en haut
7         for ind in range(len(l)-i-1) :
8             if l[ind] > l[ind+1] :
9                 l[ind], l[ind+1] = l[ind+1], l[ind]
```

Quelle est la complexité du tri à bulles ?

Implémentation du tri à bulles - amélioration

On peut s'arrêter dès que plus aucune *bulle* n'est remontée !

```
1  def tri_a_bulles(l) :
2      '''(inout) liste(int) -> rien
3      trie l par l'algorithme du tri a bulles
4      '''
5      bulle = True
6      i = 0
7      while i <len(l) and bulle :
8          bulle = False
9          ## on remonte une bulle tout en haut
10         for ind in range(len(l)-i-1) :
11             if l[ind] > l[ind+1] :
12                 l[ind], l[ind+1] = l[ind+1], l[ind]
13                 # des qu'une permutation a eu lieu
14                 # on le marque dans bulle
15                 bulle = True
16         i += 1
```

Le tri par insertion

L'idée du tri par insertion est la suivante :

- on prend chaque élément l'un après l'autre
- et on l'insère à sa place dans la partie du tableau qui est déjà triée

À vous !

Implémentation du tri par insertion

```
1  def tri_insertion(l) :
2      '''(inout) liste(int) -> rien
3      trie l par l'algorithme du tri par insertion
4      '''
5      for i in range(1,len(l)):
6          ref = l[i]
7          # on insere ref dans la liste l[0:i-1]
8          # en decalant les plus grands elements
9          # jusqu'a la position i
10         j = i-1
11         while j >= 0 and l[j] > ref :
12             l[j+1] = l[j]
13             l[j] = ref
14             j = j-1
```

Le tri fusion

Idée :

- fusionner deux listes listes triées en une seule est facile... Comment faire ? Quelle est la complexité ?
- trier une liste, c'est trier les deux moitiés de la liste, et fusionner le résultat !

À vous !

- écrivez la fonction `fusionner(l,deb,milieu,fin)`
- cette fonction recopie les tronçons à fusionner avant de mettre à jour `l`

Implémentation de la fusion

```
1  def fusionner(l,deb,milieu,fin):
2      '''(inout) list(int),int,int,int
3      [def:milieu[ et [milieu:fin[ sont tries au depart.
4      l'intervalle [deb:fin[ de l sera trie en sortie.'''
5      part1, part2 = l[deb:milieu], l[milieu:fin]
6      i, j, k = 0, 0, deb
7      while i < len(part1) and j < len(part2) :
8          if part1[i] < part2[j] :
9              l[k] = part1[i]
10             i += 1
11         else :
12             l[k] = part2[j]
13             j += 1
14         k += 1
15     while i < len(part1) :
16         l[k] = part1[i]
17         i, k = i+1, k+1
18     while j < len(part2) :
19         l[k] = part2[j]
20         j, k = j+1, k+1
```

Implémentation du tri fusion

```
1  def tri_fusion(l,deb,fin) :
2      '''(inout) list(int) -> rien
3      trie l par l'algorithme du tri fusion.
4      '''
5      if fin-deb < 2 :
6          return
7      else :
8          milieu = (deb+fin)//2
9          tri_fusion(l,deb,milieu)
10         tri_fusion(l,milieu,fin)
11         fusionner(l,deb,milieu,fin)
```

La complexité du tri fusion est en $O(nlog(n))$!

Conclusion

Pour visualiser la rapidité des différents tris que nous avons vus :
http://www.sorting-algorithms.com/