Cours Algorithmique III

Saïd Jabbour

CRIL - CNRS UMR 8188
Université d'Artois
Lens

27 septembre 2016

Programme

- Tris
- Récursivité
- Complexité
- Arbres binaires, Arbres de recherche de recherche, Arbres AVL
- Tas, Tri par tas
- Compression : Algorithme d'Huffman
- Graphes
- Algorithmes de Backtrack

Programme

Problèmes

- Tri de tableaux
- Problème des n reines
- Problème du cavalier
- Chercher une solution à un problème de programmation linéaire (simplexe)
- Chercher des motifs dans une chaîne
- Chercher une partition
- Coloriage de graphes

Objectifs :

- Algorithmes pour répondre ces problèmes (combinatoires)
- Utilisation les structures de données adéquates

Tri : Objectif

- Au départ :
  - Eléments dans un ordre quelconque
- A la fin :
  - Les **mêmes** éléments
  - Chaque élément est inférieur à celui qui le suit

Idée du tri

- Réaliser des permutations (échanges) d'éléments pour passer de la situation de départ à la situation d'arrivée
- Objectif secondaire :
  - Le moins de permutations possible

Tris simples

Sélection

Insertion

Bulles

Le tri par sélection (dit naïf)

- Principe :
  - Pour chaque position successive dans le tableau, on cherche l'élément qui occupera cette position dans le tableau trié, et on l'y place en permutant cet élément avec l'élément courant
- En version simple :
  - On cherche le plus petit élément et on l'échange avec le premier
  - Puis on cherche le plus petit élément à partir du deuxième et on l'échange avec le deuxième
  - Etc.

Pourquoi ça marche ?

- Après le premier échange, le plus petit élément est en position 1
- Après le deuxième échange, les deux premiers éléments sont les deux premiers éléments du tableau trié
- Après le ième échange, les i premiers éléments sont les i premiers éléments du tableau trié
- Arrivé à i = n, on a forcément fini (en fait même à i = n-1)

Rapide ou pas ?

- A l'étape i :
  - On cherche le plus petit élément entre les indices i et n : n - i lectures et comparaisons
  - Selon la position, on fait ou pas un échange
- Si on somme, on a au total :
  - Environ n^2 comparaisons (ne dépend pas du tableau)
  - Au pire n échanges, au mieux 0

Le tri par insertion

- Principe :
  - On va trier de façon itérative les i premiers éléments du tableau en mettant systématiquement le ième élément à sa place parmi les i-1 premiers
- En version simple :
  - Evidemment le premier élément est trié
  - On trie les deux premiers éléments (simple)
  - On cherche la place que devrait avoir le troisième pour que les trois premiers éléments soient triés et on le déplace si besoin
  - Etc...

Pourquoi ça marche ?

- Après la première étape, les deux premiers éléments sont triés
- Après la deuxième étape, les trois premiers éléments sont triés
- ...
- Après la i^{eme} étape, les i+1 premiers éléments sont triés
- Arrivés à i = n-1, le tableau entier est trié

Rapide ou pas ?

- A l'étape i :
  - On cherche la place du (i+1)-ème élément parmi les i premiers : au pire i comparaisons
  - On le met à sa place : au pire i décalages
- Au total :
  - Au pire ($\Theta(n^2)$) comparaisons (au mieux n)
  - Au pire ($\Theta(n^2)$) décalages (au mieux 0)

Le tri à bulles

- Principe :
  - On parcourt le tableau en arrangeant les éléments consécutifs, jusqu'à ce que plus aucun réarrangement ne soit nécessaire
- En version simple :
  - On parcourt tout le tableau en échangeant systématiquement les contenus des cases d'indice k et k+1 s'ils ne sont pas dans l'ordre
  - Tant qu'on a eu des changements à faire, on recommence

Pourquoi ça marche ?

- Après le premier passage, la plus grande valeur est à l'indice n
- Après le deuxième, les deux dernières cases sont les deux plus grandes valeurs du tableau, dans l'ordre
- ...
- Après la i^{eme} étape, les i dernières valeurs du tableau sont les i dernières valeurs du tableau trié
- A i = n-1 au pire, on a fini
- Si aucun changement n'a été nécessaire, toutes les cases ont un contenu inférieur à celui de la case suivante
- C'est exactement la définition d'un tableau trié
- Donc le critère est bon aussi

Rapide ou pas

- A l'étape i :
  - (n-1) comparaisons (indépendant du tableau)
  - Au pire (n - i) échanges (au mieux 0)
- Au total :
  - ($\Theta(n^2)$) comparaisons
  - Au pire ($\Theta(n^2)$) échanges (au mieux 0)

Variantes

- Optimisation facile :
  - A l'étape i, on ne parcourt que de l'indice 1 à l'indice n-i+1 puisque le reste est déjà à sa place (le gain de temps est non négligeable)
- Tri bulles descendant :
  - On prend les indices dans l'ordre inverse (équivalent)

Variantes (2)

- **Tri ”Shaker” ou ”Shuttle” :**
  - On reprend au début dès qu'un échange a été détecté
- **Tri bidirectionnel ou ”Boustrophedon” :**
  - Une fois en montant, une fois en descendant, etc..
- Seule la première variante fait gagner du temps

Tri simples : implémentations

```
def triSelection(liste):
    for i in range(0, len(liste)-1):
        for j in range(i+1, len(liste)):
            if liste[i] > liste[j]:
                liste[i],liste[j] = liste[j],liste[i]
```

```
def triInsertion(liste):
    for i in range(1, len(liste)):
        m = liste[i]
        j = i-1
        while j >= 0 and liste[j] > m:
            liste[j+1] = liste[j]
            j -= 1
        liste[j+1] = m
```

Tri simples : implémentations (suite)

```
def triBulles(liste):
    permute = True
    while permute:
        permute = False
        for i in range(len(liste)-1):
            if liste[i+1] < liste[i]:
                tmp = liste[i]
                liste[i],liste[i+1] = liste[i+1],liste[i]
                permute = True
```

La récursivité

Le principe de récursivité

Tout objet est dit récursif s'il se définit à partir de lui-même

Ainsi, une fonction est dite récursive si elle comporte, dans son corps, au moins un appel à elle-même

De même, une structure est récursive si un de ses attributs en est une autre instance

Correspondance mathématique

Principe de récurrence Exemple : définition des entiers (Peano)

- 0 est un entier
- Si n est un entier, alors n+1 est un entier

Exemples de fonctions récursives

Calcul de la somme des entiers de 1 à n

- On calcule la somme jusqu'à n-1
- Puis on ajoute n

Idem avec le produit (fonction factorielle)

Un peu de vocabulaire

Pour une fonction récursive, on parlera :

- De récursivité **terminale** si aucune instruction n'est exécutée après l'appel de la fonction à elle-même ou encore
- Une fonction récursive est dite `terminale` si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).
- De récursivité **non terminale** dans l'autre cas

Terminale / Non Terminale

- Une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.

- En récursivité terminale, les appels récursifs n'ont pas besoin d'êtres empilés dans la pile d'exécution car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

- Certains langages utilisent cette propriété pour exécuter les récursions terminales aussi efficacement que les itérations.

Récursivité : Exemple

Terminal

```
def f(n):
    if n==0 :
       print ("Hello")
    else :
      f(n-1)
```

Non terminale

```
def f(n) :
    if n>0:
       f(n-1)
     else :
        print ("Hello")
```

Récursivité directe

Lorsque f s'appelle elle-même, on parle de `récursivité directe`

Lorsque f appelle g qui appelle f, il s'agit aussi de récursivité

- On l'appelle alors `indirecte`

Exemples de structures récursives

Liste récursive

- Le premier élément
- Et le reste de la liste (qui est aussi une liste)

Une expression arithmétique est :

- Soit une valeur
- Soit une expression, un opérateur et une autre expression

Implémentation

Comment programmer une fonction récursive ?

Quels sont les pièges à éviter ?

Programmer une fonction récursive

Il suffit de la faire s'appeler elle-même

```
def f(n):
    return f(n-1)
```

La fonction f est récursive : elle s'appelle elle-même

Mais f n'a-t-elle pas un problème ?

Une obligation : s'arrêter

La fonction f telle qu'elle est écrite ne s'arrête pas :

- Appel : f(2)
- Appel : f(1)
- Appel : f(0)
- Appel : f(-1)
- Appel : f(-2)
- Etc...

Comment y parvenir

Première étape : **la condition terminale**

- Obligatoirement au début de toute fonction récursive
- Une condition : le cas particulier
- Pour ce cas, pas d'autre appel à la fonction : la chaîne d'appels s'arrête

Exemple

```
def f(int n):
  if n==0 :
    print ("Hello")
  else :
    f(n-1)
```

Ici quand n vaut 0, on s'arrête

Problème : arrive-t-on à n = 0 ?

Terminaison de la fonction

Il faut que la fonction s'arrête
La condition terminale ne sert à rien si elle ne devient jamais vraie

Exemple avec la fonction précédente :

- f(-2) provoque une pile d'appels infinie
- Probablement d'autres tests à faire (si $n < 0$, envoyer une exception par exemple)

Une bonne solution

```
def f(int n) :
  if n < 0 :
    sys.exit(0)
  if n==0 :
    print("Hello")
  else :
    f(n-1)
```

Pourquoi ça marche ?

Si n est négatif : on s'arrête sur une exception

Si n est nul : c'est le cas d'arrêt ("Hello")

Si n est positif : on appelle f avec la valeur n-1

- Chaine d'appels avec des valeurs entières strictement décroissantes de 1 en 1
- On arrive forcément à 0
- On affiche "Hello"
- On remonte la pile des appels (sans rien faire, ici la récursivité est terminale)

Théorème de Gödel

Il n'existe pas de moyen automatique pour savoir si un programme termine ou pas

Conclusion

Il faut regarder cas par cas, et à la main

Même si aucune méthode n'est générale, le principe de récurrence aide souvent

En résumé

Une fonction récursive doit comporter :

- Un cas d'arrêt dans lequel aucun autre appel n'est effectué
- Un cas général dans lequel un ou plusieurs autres appels sont effectués

La chaîne d'appel doit conduire au critère d'arrêt

- Optionnellement, des cas impossibles ou incorrects à traiter par des exceptions

Récursification facile ; récursivité obligatoire ?

Les boucles for

Très bonne candidate

Toute boucle for peut se transformer en une fonction récursive

Principe :

- Pour faire des choses pour un indice allant de 1 à n
  - On les fait de 1 à n-1 (même traitement avec une donnée différente)
  - Puis on les fait pour l'indice n (cas particulier)

Traduction

```
def f(n) :
  for i in range(n) :
    traiter(i)

def f(n) :
  if n==0 :
    traiter(0);
  else :
     f(n-1)
     traiter(n)
```

Exemple : fonction factorielle

Appel de fact(5) récursif

```
def fact(n) :
  res = 1
  for   i in range(2, n+1) :
    res = res*i
  return res

def fact(int n) :
  if n==0 :
    return 1
   else :
     return fact(n-1)*n
```

Appel de fact(5) récursif
Phase de descente récursive

```
Appel à fact(5)
    Appel à fact(4)
        Appel à fact(3)
            Appel à fact(2)
                Appel à fact(1)
                    Appel à fact(0)
```

Condition terminale

- Retour de la valeur 1

Suite

Phase de remontée (après l'appel à fact(n-1) on multiplie par n : la récursivité n'est pas terminale)

```
Retour de la valeur 1
      Retour de la valeur 2
           Retour de la valeur 6
                Retour de la valeur 24
                     Retour de la valeur 120
```

Quelques conséquences

La plupart des traitement sur les tableaux peuvent se mettre sous forme récursive :

- Tris (sélection, insertion)
- Recherche séquentielle (attention : pas dichotomique)
- Inversion
- Problème des huit reines
- Etc...

Une constatation

L'écriture sous forme récursive est toujours plus simple que l'écriture sous forme itérative

Une question

Même fonction est-elle plus efficace sous forme récursive ou sous forme itérative ? (Ou, sous une autre forme, y a-t-il un choix optimal généralisable ?)

La réponse est non. La réponse à la question inverse est non. Il n'y a pas de généralité

En revanche

La plupart des traitements itératifs simples sont facilement traduisibles sous forme récursive (exemple du for)

L'inverse est faux

Il arrive même qu'un problème ait une solution récursive triviale alors qu'il est très difficile d'en trouver une solution itérative

La fonction d'ackermann

Ack(m, n) vaut :

$n+1$	si m=0
$Ack(m-1,1)$	sinon et si n=0
$Ack(m-1,Ack(m,n-1))$	autrement

Remarque : on finit bien car max(m,n) est strictement décroissant sur les appels (à l'exception de $Ack(1,0)$ qui finit trivialement)

Exercice

Donner une version itérative, récursive non terminale et récursive terminale de la suite de fibonacci définit comme suit :

- $f(0) = 1$
- $f(1) = 1$
- $f(n) = f(n-1) + f(n-2),\ n \geq 2$

Complexité

Temps de calcul

Ordres de grandeur Comparaisons de complexités

Temps de calcul

Complexité en temps

$C(A, D)$ = temps d'exécution de l'algorithme A appliqué aux données D

Eléments de calcul de la complexité en temps

- C(opération élémentaire) = constant
- C(si F alors I sinon J) $C(F) + max(C(I), C(J))$
- C(pourtout i de e_1 à e_2 faire E_i) $\leq C(e_1) + C(e_2) + \sum C(E_i)$
- Temps de calcul de procédures récursives : solution d'équations de récurrence

Temps de calcul (2)

$C(A, D)$ dépend en général de D

Complexité au pire

$C_{MAX}(A, n) = max\{C(A, D)\ ;\ D \text{ de taille } n\}$

Complexité au mieux

$C_{MIN}(A, n) = min\{C(A, D); D \text{ de taille } n\}$

Complexité en moyenne

$C_{MOY}(A, n) = \sum_{D \text{ de taille } n} p(D) \times C(A, D)$

où $p(D)$ est la probabilité d'avoir la donnée D

$C_{MIN}(A, n) \leq C_{MOY}(A, n) \leq C_{MAX}(A, n)$

Ordre de grandeur

	$log_2(n)$	$n^{1/2}$	$nlog_2 n$	n^2	n^3	2^n
n=5	2.32	2.24	11.6	25	125	32
n=10	3.32	3.16	33.2	100	1000	1024
n=100	6.64	10	664	10000	10^6	10^{30}
n=1000	9.97	31.62	9970	10^6	10^9	10^{300}
n=10000	13.29	100	132900	10^8	10^{12}	10^{3000}

Ordre de grandeur (2)

Coût $C(n)$	Evolution quand la taille est 10 fois plus grande : $C(n \times 10)$
$log_2 n$	$C(n) + 3,32$
$n^{1/2}$	$C(n) \times 3,16$
n	$C(n) \times 10$
$nlog2n$	$C(n) \times (10 + e)$
n^2	$C(n) \times 100$
n^3	$C(n) \times 1000$
2^n	$C(n)^{10}$

Comparaisons de complexités

Ordres de grandeur asymptotique
$C(n) = O(f(n))$ s'il existe une constante $k > 0$ et un entier positif N tels que pour tout $n \geq N$ on a $C(n) \leq k\ f(n)$

On dit alors que $C(n)$ est "grand O" de $f(n)$.

Exemples

- $n = O(n^2)$ En effet si $n \geq 1$ on a que $n \leq n^2$ et donc la condition est vérifiée avec $N = 1$ et $k = 1$.
- $log_2 n = O(n^\alpha)$ pour tout $\alpha > 0$
- $n^\alpha = O(a^n)$ pour tout $\alpha > 0$ et $a > 1$

Ordres de grandeur asymptotique

$C(n) = \Omega(f(n))$ s'il existe une constante $k > 0$ et un entier positif N tels que pour tout $n \geq N$ on a

$C(n) \geq kf(n)$

On dit alors que $C(n)$ est "grand oméga" de $f(n)$.

Exemples :

- Si $C(n) = O(f(n))$ alors $f(n) = \Omega(C(n))$

  En effet si $n \geq N$ on a que $C(n) \leq kf(n)$ avec

  $k > 0$. Donc à partir de N on a que $f(n) \geq 1/k\ C(n)$.

Ordres de grandeur asymptotique

$C(n) = \Theta(f(n))$ si $C(n) = O(f(n)) et C(n) = \Omega(f(n))$

On dit alors que $C(n)$ est "grand thêta" de $f(n)$.

Exemples :

- $2n^2 + n = \Theta(n^2)$
  En effet pour $n \geq 1$ on a
  $2n^2 + n \leq 2n^2 + n^2 = 3n^2$
  et donc $2n^2 + n = O(n^2)$
  De plus, pout tout n 0 on a $2n^2 + n \geq 2n^2$ et donc
  $2n^2 + n = \Omega(n^2)$

Tris avancés

Tri fusion

Tri rapide

Le tri fusion ou Mergesort

Principe

Algorithme

Exemple

Le principe du tri fusion

Idée de base : ”diviser pour régner”
Pour trier un tableau :

- On trie deux moitiés de tableau (de 1 à n/2 et de n/2+1 à n)
- On les fusionne

On reconnaît un fonctionnement récursif

Cas d’arrêt évident : tableau à une case

La fusion

On dispose de deux tableaux triés qu'on veut fusionner

On passe par un tableau annexe qu'on va remplir alternativement avec les éléments des deux tableaux

- Si le premier des éléments restants du premier tableau est inférieur au premier des éléments restants du second, c'est lui qu'on ajoute
- Sinon c'est l'autre

La fusion (suite)

Une fois que tous les éléments ont été traités, on a un tableau trié regroupant les éléments des deux tableaux

- (en fait deux sous-tableaux contigus de notre tableau de départ)

On recopie ensuite ce tableau annexe dans le tableau de départ à la bonne place

La fusion (suite)

```
def fusion(a,b) :
  c = []
  n = len(a)
  m = len(b)
  i = 0
  j = 0
  # on continue tant que les deux listes ne sont pas vides
  while i < len(a) and j < len(b) :
     if a[i] < b[j] :
      c.append(a[i])
      i = i + 1
     else :
      c.append(b[j])
      j = j + 1
   if i = len(a) :
     for j in range(j,len(b)) :
       c.append(b[j])
   else :
     for j in range(i,len(a)) :
       c.append(a[i])
return c
```

La fusion (suite)

```
def triFusion(a) :
# renvoi  la liste des éléments de a triés
  n = len(a)
  if n <= 1 return a
  m = n // 2
  return fusion( triFusion(a[0:m]) , triFusion(a[m:n]) )
```

Exemple

Une fois le tableau annexe rempli, il faut le recopier dans le tableau initial

- À faire à chaque étape
- Avec les bons indices
- Attention aux paramètres des appels récursifs

Algorithme efficace ?

Si on note $C(n)$ le nombre d'opérations qu'il faut pour trier un tableau de taille n, on a :

- $C(n) = 2 \times C(n/2) + n$

  (n opérations pour la fusion)

- Si on résout la récurrence, on obtient que :

  $C(n) = \Theta(n \log_2 n)$

Algorithme efficace (2)

- **Pour un tri, $n \log_2 n$ est la complexité temporelle optimale**
- Pourtant, ce n'est pas le tri le plus utilisé
  - Pas la meilleure constante ($k\ n \log_2 n$)
    - Surtout : on a besoin d'un deuxième tableau aussi grand que le premier (problème de complexité spatiale)

Le tri rapide ou Quicksort

Principe

Algorithme

Exemple

Un peu de culture générale

Sir Charles Antony Richard Hoare : 1962

Encore un tri dichotomique

L'algorithme le plus utilisé au monde

Encore un tri dichotomique

L'algorithme le plus utilisé au monde

Le principe du QuickSort

- On va séparer le tableau en deux parties, puis les trier
- Contrairement au tri fusion, la séparation ne sera pas arbitraire
- On n'aura plus besoin de fusionner

Principe de la séparation

- On choisit une valeur du tableau, qui servira de pivot
- On réorganise le tableau de façon à ce que :
  - Toutes les valeurs inférieures au pivot soient placées avant lui dans le tableau
  - Et toutes les valeurs supérieures après

Implementation

```
def triRapide(liste, debut, fin):
    if debut < fin:
        # partitionnement de la  liste
        pivot = partition(liste, debut, fin)
        # sort both halves
        triRapide(liste, debut, pivot-1)
        triRapide(liste, pivot+1, fin)
    return liste
```

```
def partition(liste, debut, fin):
pivot = liste[debut]
left = debut+1
right = fin
fait = False
while not fait:
    while left <= right and liste[left] <= pivot:
        left = left + 1
    while liste[right] >= pivot and right >=left:
        right = right -1
    if right < left:
        fait= True
    else:
        # permuter places
        temp=liste[left]
        liste[left]=liste[right]
        liste[right]=temp
# permute debut avec liste[right]
temp=liste[debut]
liste[debut]=liste[right]
liste[right]=temp
return right
```

Quelques remarques

- Ici on considère que le premier élément est le pivot
  - Si ce n'est pas lui, un premier échange suffit
- On avance dans le tableau en remettant systématiquement les valeurs inférieures au pivot dans les premières cases
- Le dernier échange met le pivot à sa place

Autres Méthodes

Deux itérateurs, partant l'un du début+1 l'autre de la fin (le pivot étant encore au début)

- Tant qu'on est plus petit que le pivot en partant du début, on avance
- Tant qu'on est plus grand en partant de la fin, on recule
- Si les deux positions ne sont pas confondues, on échange
- Puis on recommence jusqu'à ce que les positions soient confondues

Ensuite

- **Une fois le tableau partitionné** :
  - on trie les deux ”moitiés” (encore de la récursivité)
  - autour du pivot (exclus, il est déjà à sa place)
- Une fois qu’elles sont triées, pas besoin de fusionner
  - Les éléments du sous-tableau de gauche sont forcément inférieurs à ceux du sous-tableau de droite
- Cas d’arrêt encore une fois évident :
  - moins de 2 cases

Pourquoi ça marche

On prend un pivot

Les plus petits devant, les plus grands derrière

Et on trie les deux parties autour du pivot

Algorithme efficace

- Fortement dépendant du choix du pivot
  - Choix optimal = clé médiane
  - Pire des choix : le plus petit ou le plus grand
    - Laisse une case et un tableau de n-1
    - On se retrouve grosso modo au tri par sélection
- Mais un choix du pivot optimal passe par un autre algorithme
  - D'où perte de temps

En pratique

On tape au hasard

Pire des cas : on prend toujours le plus grand ou le plus petit

Dans ce cas, on est en $\Theta(n^2)$

En moyenne $\Theta(n \log_2 n)$

Avec une constante inférieure à celle du tri fusion

Ne nécessite pas de tableau annexe

Efficace quand ?

Pour les structures de grande taille

Pour les petits tableaux, on lui préfère un tri simple, comme la sélection ou l'insertion

Optimisation : en dessous d'une certaine taille limite, on ne fait plus l'appel récursif mais un tri par insertion