cours_6

March 10, 2021

Cours 6

Le cours du jour traite des points suivants:

- Recherche du k-ième élément
  - Cas particuliers du min, max
  - Cas du deuxième plus grand element
  - Algorithme de la selection rapide (Quick-Select)
- Génération de permutations
- Compléments de Python:
  - tri avec choix de la clé
  - module bisect: dichotomie
  - module itertools: product, arrangement, permutation, cycle

1 Recherche du k-ième élément

Définition du problème:

Entrée: T un tableau de n éléments (distincts) et k un nombre tel que 1 <= k <= n

Sortie: La valeur x appartenant à T tel que k-1 éléments de T sont inférieurs à x

Exemples d'applications

- Quel est l'âge médian en France ? (40,8 contre 41,7 en moy.)
- Combien gagne les 10% les plus riches ?
- Détection les événements rares (> 1%)

1.1 Exemples triviaux: Recherche min ou max

- k = 1 -> min
- k = n -> max

[1]:
```
# votre reponse
```

[2]:
```
def minimum(T):
    x = T[0]
    for e in T[1:]:
        if e < x:
            x = e
    return x
```

Analyse: * 1 comparaisons par valeur * 1 *(n-1) comparaisons au total => $\Theta(n)$, linéaire

1.1.1 Exemples triviaux: Recherche min *et* max

[3]:
```
def minmax(T):
    min, max = T[0], T[0]
    for i in range(1, len(T)):
        min = T[i] if T[i] < min else min
        max = T[i] if T[i] > max else max
    return min, max
```

Analyse: * 2 comparaisons par valeur * 2 *(n-1) comparaisons au total => $\Theta(n)$, linéaire

1.1.2 Recherche min *et* max : Faire mieux

Supposons T[i] et T[j] deux éléments de T.

Dans l'algorithme précédent, 4 comparaisons pour calculer

- minimum(minimum(T[i], min) , T[j]) : 2 comparaisons
- maximum(maximum(T[i], max), T[j]) : 2 comparaisons

Remarquons:

- minimum(T[i], T[j]) **et** maximum(T[i], T[j]) : 1 comparaison
- minimum(minimum(T[i], T[j]), min): 1 comparaison
- maximum(maximum(T[i], T[j]), max): 1 comparaison

Donc 3 comparaisons au lieu de 4 !

Stratégie

- Prendre deux éléments successifs T[i] et T[i+1]
- Mettre à jour le min et max en 3 comparaisons
- Gérer les cas limites:
  - Tableau à 1 élément
  - Nombre impair d'éléments

[4]:
```
def minmax3(T):
    if len(T) == 0: return None, None
    min = max = T[0]
    start = 1 if len(T) % 2 == 1 else 0
    for i in range(start, len(T), 2):
        if T[i] <= T[i+1]:   # T[i] <= T[i+1]
            min = T[i] if T[i] < min else min
            max = T[i+1] if T[i+1] > max else max
        else:                # T[i] > T[i+1]
            min = T[i+1] if T[i+1] < min else min
            max = T[i] if T[i] > max else max
    return min, max
```

[5]:
```
T = [2, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5]
print(minmax3(T))
```

```
(-4, 13)
```

Analyse:

- Pour tout couple: 3 comparaisons
- Soit env. 3*(n//2)=1.5*n comp. au lieu de 2*n, l'algorithme reste linéaire.
- Il n'est pas possible de faire mieux sans hypothèse sur le contenu du tableau.

Attention:

- les fonctions natives min et max de Python sont implementées en C
- la fonction minmax n'existe pas dans le langage

[6]:
```
from random import randint
n = 10000
T =[randint(-n, n) for _ in range(n)]
%timeit minmax(T)
%timeit minmax3(T)
```

```
1.27 ms ± 93.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.29 ms ± 13.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

On aurait sans doute dû compter également les accès mémoire:

- minmax : 8 pour deux valeurs
- minmax3 : 6 pour deux valeurs

Bref: pas logique … sans doute, effets de cache mémoire.

Parcours par indice versus par valeur

[7]:
```
def minmax_v2(T):
    min, max = T[0], T[0]
    for e in T[1:]:
        min = e if e < min else min
        max = e if e > max else max
    return min, max
```

[8]:
```
def minmax_v3(T):
    min, max = T[0], T[0]
    for i in range(1, len(T)):
        e = T[i]
        min = e if e < min else min
        max = e if e > max else max
    return min, max
```

[9]:
```
n = 10000
T =[randint(-n, n) for _ in range(n)]
%timeit minmax(T)
```

```
%timeit minmax_v2(T)
%timeit minmax_v3(T)
```

```
1.19 ms ± 8.09 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
677 µs ± 19.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1 ms ± 6.42 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

On a divisé le temps d'exécution par 2 ...

1.1.3 Recherche min et max : approche récursive

- Découpage en 2 parties égales
- Calcul des min **et** max à gauche et à droite
- Fusion des résultats
- Simple, clair ...
- Mais lent: cout de la récursion, pas de parallélisme

[10]:
```
def minmax_rec(T):
    if len(T) == 0:
        return None, None
    elif len(T) == 1:
        return T[0], T[0]
    else:
        mid = len(T) // 2
        min_g, max_g = minmax_rec(T[:mid])
        min_d, max_d = minmax_rec(T[mid:])
        min = min_g if min_g < min_d else min_d
        max = max_g if max_g > max_d else max_d
        return min, max
```

[11]:
```
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5]
print(minmax_rec(T))
```

```
(-4, 13)
```

[12]:
```
from random import randint
n = 10000
T =[randint(-n, n) for _ in range(n)]
%timeit minmax(T)
%timeit minmax3(T)
%timeit minmax_rec(T)
```

```
1.27 ms ± 113 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.33 ms ± 76 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
8.89 ms ± 605 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.2 Recherche du 2ème plus petit élément

[13]:
```
def deuxieme(T):
    if len(T) == 1: return None
    if len(T) >= 2:
        first, sec = (T[0], T[1]) if T[0] < T[1] else (T[1], T[0])
    for e in T[2:]:
        if e < sec:
            if e < first: # e < first <= sec
                first, sec = e, first
            else:         # first <= e < sec
                sec = e
    return sec
```

[14]:
```
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5]
print(deuxieme(T))
```

```
-2
```

Analyse:

- Au pire : 2*(n-2) + 1 comparaisons
- Au mieux: n-1 comparaisons
- En moyenne, c'est comme la recherche du min mais cela reste linéaire.
- Que se passe t'il si on cherche le troisième plus petit ?
- Que se passe t'il si on cherche le k-ième plus petit ?

=> Encore plus de mémoire ... pas viable

[15]:
```
n = 10000
T =[randint(-n, n) for _ in range(n)]
%timeit minimum(T)
%timeit deuxieme(T)
```

```
387 µs ± 43.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
333 µs ± 23.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

1.3 Recherche du k-ième plus petit élément

Pour T un tableau de taille n, approche naïve:

- Trier le tableau T
- Retourner l'élément en position k

Efficacité:

- Tri par comparaison: $\Omega(n \log n)$
- Tri par comptage: $\Theta(n + k)$ où k est le nombre de clés.

1.3.1 Retour sur le partitionnement du Quicksort

Que dire de la position du pivot après partitionnement ?

- i_pivot = k : le k-ième élément est trouvé
- k < i_pivot : le k-ième élément se trouve dans le sous-tableau T[g..i_pivot-1]
- k > i_pivot : le k-ième élément se trouve dans le sous-tableau T[i_pivot+1..d]

Algorithme 6: Quick-Select

Données : T un tableau, g, d deux indices du tableau, k
Résultat : La k-ième plus petite valeur de T

```
1 i_pivot ← Partition(T, g, d);
2 si i_pivot = k alors retourner T[k];
3 si k < i_pivot alors
4 |  retourner Quick-Select(T, g, i_pivot-1, k)
5 sinon
6 |  retourner Quick-Select(T, i_pivot+1, d, k)
7 fin
8 return i;
```

[16]:
```python
def pivot_median(T, g, d):
    i1, i2, i3 = [randint(g, d) for _ in range(3)]
    if T[i2] <= T[i1] <= T[i3]: return i1
    if T[i3] <= T[i1] <= T[i2]: return i1
    if T[i1] <= T[i2] <= T[i3]: return i2
    if T[i3] <= T[i2] <= T[i1]: return i2
    return i3


def partition(T, g, d, ipivot=None):
    if ipivot is None:
        ipivot = pivot_median(T, g, d)
    if ipivot < d:
        T[ipivot], T[d] = T[d], T[ipivot]

    pivot = T[d]
    i = g
    for j in range(g, d):
        if T[j] <= pivot:
            T[i], T[j] = T[j], T[i]
            i = i + 1
    T[i], T[d] = T[d], T[i]
    return i
```

[17]:
```python
def selectionrapide(T, g=0, d=None, k=0):
    """ Selection rapide du k-ieme element dans T[g..d]"""
```

```
    if d is None:
        d = len(T)-1
    ipivot = partition(T, g, d)
    if ipivot == k:  return T[k]
    if k < ipivot:
        return selectionrapide(T, g, ipivot-1, k)
    else:
        return selectionrapide(T, ipivot+1, d, k)
```

[18]:
```
k = 2
T = [1, 5, 2, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
print(sorted(T)[k])
T = [1, 5, 2, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
print(selectionrapide(T, k=k))
print(T)
```

```
-1
-1
[-4, -2, -1, 1, 2, 2, 2, 2, 2, 5, 5, 5, 7, 13, 9]
```

1.3.2 Analyse du Quick-Select

- Cela ressemble à l'exécution partielle du tri rapide
- La dichotomie est effectuée est la position (vs valeur)

Nombre de comparaisons:

- Pire des cas (mauvais part. [1, n-1] ou [n-1, 1]):

$$Comp(n) = n + Comp(n-1)$$

  Par substitution:

$$Comp(n) = n + (n-1) + ...1 = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$$

- Meilleur des cas ([n/2, n/2] + pivot au milieu):

$$Comp(n) = n \in \Omega(n)$$

Cas plus réaliste (bon partitionnement, quelque soit k):

$$\begin{aligned} Comp(n) &= Comp\left(\frac{}{2}\right) + n = Comp\left(\frac{}{4}\right) + \frac{}{2} + n \\ &= Comp\left(\frac{}{8}\right) + \frac{}{4} + \frac{}{2} + n \\ &= \sum_{k=0}^{r} \frac{}{2^k} = n\left(1 + \sum_{k=1}^{r} \frac{1}{2^k}\right) \end{aligned}$$

On se rappelle que $r = \log_2(n)$.

Une formule de math à admettre:

$$\sum_{k=1}^{r} \frac{1}{2^k} = 1 - 2^{-r}$$

Conclusion:

$$Comp(n) = 2n - 1$$

Cas moyen: $E[Comp(n)] \in O(n)$

- Trier par comparaison, c'est au minimum $n\ log\ n$ comp $\in \Omega(nlogn)$
- C'est donc trop pour le problème de sélection: env. n comp.
- Quick-Select est donc optimal pour le problème donné.
- Sa qualité dépend de la stratégie de choix du pivot (comme le tri rapide).

[19]:
```
def trirapide(T, g=0, d=None):
    if d is None:
        d = len(T) - 1
    if g < d:
        ipivot = partition(T, g, d)
        trirapide(T, g, ipivot - 1)
        trirapide(T, ipivot + 1, d)
    return T
```

[20]:
```
n = 1_000_000
T = [randint(-n, n) for _ in range(n)]
k = 2
```

[21]:
```
T_ = T.copy()
%timeit selectionrapide(T_, k=k)
T_ = T.copy()
%timeit trirapide(T_)[k]
```

```
398 ms ± 44.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
6.36 s ± 280 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

1.3.3 Recherche du k-ième plus petit élément (Comptage)

Nombre d'éléments parcourus:

- Calcul du min et max: *1.5 n*
- Comptage des clés: n
- Parcours des clés jusqu'à k: k (au pire k=n)

Conclusion: linéaire mais en pratique doit être inférieur à Quick-Select (env. *1.5 n*)

[22]:
```
def selectioncomptage(T, k):
    assert(k < len(T))
    m, M = min(T), max(T)
    Cpt = [0] * (M-m+1)
    for e in T: Cpt[e-m] += 1
    i, s = m, 0
    while s < k:
        s += Cpt[i]
        i += 1
    return i
```

[23]:
```
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
T = [1, 3, 3, 5, 5, 5, 7, 8]
for k in range(len(T)):
    print(selectioncomptage(T, k=k), selectionrapide(T, k=k), sep=' ', end= ',␣
 ↪')
print()
```

```
1 1, 3 3, 3 3, 5 5, 5 5, 5 5, 7 7, 8 8,
```

2 Generation des permutations

Problème: Génération de tous les arrangements possibles des éléments d'un tableau, d'une chaine

"ABC" -> "ABC", "ACB", "BAC", "BCA", "CBA", "CAB"

[1, 2, 3] -> [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1], [3, 1, 2]

Rappel : pour n éléments, il y a n! (=1*2* …*n) permutations.

2.1 principe de la récursion

1. Mettre à part un des éléments de T
2. Générer les permutations pour les éléments restants (taille n-1)
3. Répéter 1. pour un autre élément du tableau.

[24]:
```
def permutations(T):
    perms =[]
    if len(T) > 1:
        for i, Ti in enumerate(T):
            Ti = [Ti] if isinstance(T, list) else Ti
            perms_1 = permutations(T[:i] + T[i+1:])
            perms.extend([Ti + perm for perm in perms_1])
        return perms
    return [T]
```

[25]:
```
P = permutations([1, 2, 3])
print(f"{len(P)} permutations: {P}")
P = permutations("ABC")
```

```
print(f"{len(P)} permutations: {P}")
```

```
6 permutations: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2,
1]]
6 permutations: ['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

3 Compléments de Python

3.1 Tri avec .sort() et sorted()

[26]:
```
help(T.sort)
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

[27]:
```
help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

[28]:
```
T = ['rouge', 'vert', 'bleu']
T_trie = sorted(T)
print(T_trie)
print(T)
T.sort()
print(T)
```

```
['bleu', 'rouge', 'vert']
['rouge', 'vert', 'bleu']
['bleu', 'rouge', 'vert']
```

[29]:
```
# Tri sur la derniere lettre
T = ['rouge', 'vert', 'bleu']
print(sorted(T, key=lambda x: x[-1]))

# Tri décroissant sur le numero
T = [('rouge', 10) , ('vert', 5), ('bleu', 8)]
sorted(T, key=lambda x: x[1], reverse=True)
```

```
['rouge', 'vert', 'bleu']
```

[29]:
```
[('rouge', 10), ('bleu', 8), ('vert', 5)]
```

3.2 Dichotomie : module bisect

Documentation du module bisect

- bisect_left(T, x): le plus à gauche égal à x
- bisect_right(T, x): à droite du dernier x

[30]:
```
from bisect import bisect_left, bisect_right, bisect

T = [1, 2, 2, 2, 2, 4, 4, 7, 7, 7, 7, 9]
print(bisect_left(T, 7))
print(bisect_right(T, 7)) # ou bisect(T, 7)
```

```
7
11
```

[31]:
```
print(bisect_left(T, 5), bisect_right(T, 5))
print(bisect_left(T, 0), bisect_right(T, 0))
```

```
7 7
0 0
```

- bisect.insort_left(a, x, lo=0, hi=len(a)) > Insert x in a in sorted order. This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x) assuming that a is already sorted. Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.
- bisect.insort_right(a, x, lo=0, hi=len(a))

[32]:
```
def occ_dicho2(T, x):
    g = bisect_left(T, x)
    d = bisect_right(T, x)
    return d-g
```

[33]:
```
import random
n = 10**6
T = [random.randint(1, 100) for _ in range(n)]
T.sort()
```

```
x = random.randint(1, 100)
print(occ_dicho2(T, x), T.count(x))
```

```
9932 9932
```

[34]:
```
%%timeit
occ_dicho2(T, x)
```

```
1.09 µs ± 35.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

[35]:
```
%%timeit
T.count(x)
```

```
12.9 ms ± 247 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

3.3 Module itertools

- Itérateurs combinatoires
- Itérateurs infinis …