cours_6

February 13, 2020

0.1 Recherche du k-ième élément

0.1.1 Recherche du 2ème plus petit élément

[1]:
```
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:
                first, sec = e, first
            else:
                sec = e
    return sec
```

Analyse:

- Au pire : 2*(n-2) + 1 comparaisons
- Au mieux: n-1 comparaisons
- C'est 2 fois plus que 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 ?

0.1.2 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.

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

$i\_pivot \leftarrow$ Partition(T, g, d);
si $i\_pivot = k$ **alors retourner** T[k];
si $k < i\_pivot$ **alors**
    retourner Quick-Select(T, g, i_pivot-1, k)
sinon
    retourner Quick-Select(T, i_pivot+1, d, k)
fin
return i;

Algorithme Quick-Select

[2]:
```python
def partition(T, g, d):
    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]     # le pivot va en T[i]
    return i
```

[3]:
```python
def selectionrapide(T, g=0, d=None, k=0):
    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)
```

[4]:
```python
k = 8
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
print(sorted(T)[k])
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
print(selectionrapide(T, k=k))
print(T)
```

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

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.):
    Comp(n) = *n* + *Comp(n-1)* -> env. n(n+1)/2 = n^2
  - Meilleur des cas: *Comp(n)* = *n* Explication: Dichotomie sur le rang *n* + *n*/2 + ... + *4* + 2 + *1* ∈ *O(n)*
  - Cas moyen:
$$E[Comp(n)] \in O(n)$$

- Trier par comparaison, c'est au minimum *n log n* comp.
- 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)

0.1.3 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*)

[5]:
```python
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
```

[6]:
```python
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,
```

1 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.

1.1 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.

[7]:
```
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]
```

[8]:
```
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']
```

2 Compléments de Python

- count
- sort, sorted
- bisect
- itertools

2.1 Comptage avec T.count(x)

[9]:
```
help(T.count)
```

```
Help on built-in function count:

count(value, /) method of builtins.list instance
    Return number of occurrences of value.
```

[10]:
```
import random
```

[11]:
```
%%timeit
x = random.randint(1, 100)
T.count(x)
```

```
1.18 ţs ś 10.7 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

2.2 Tri avec .sort() et sorted()

[12]:
```
help(T.sort)
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Stable sort *IN PLACE*.
```

[13]:
```
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.
```

[14]:
```
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']
```

[15]:
```
# 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']
```

[15]:
```
[('rouge', 10), ('bleu', 8), ('vert', 5)]
```

2.3 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

[16]:
```
from bisect import bisect_left, bisect_right
T = [1, 2, 3, 4, 4, 7, 7, 7, 7, 9]
print(bisect_left(T, 4))
print(bisect_right(T, 4))
```

```
3
5
```

[17]:
```
print(bisect_left(T, 0))
print(bisect_right(T, 0))
```

```
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))

Autres recettes basées sur bisect

[18]:
```
def occ_dicho2(T, x):
    g = bisect_left(T, x)
    d = bisect_right(T, x)
```

```
    return d-g
```

[19]:
```
import random
n = 10**6
T = [random.randint(1, 100) for _ in range(n)]
T.sort()
```

[20]:
```
%%timeit
x = random.randint(1, 100)
occ_dicho2(T, x)
```

```
2.52 ţs ś 214 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

2.4 Module itertools

- Itérateurs combinatoires
- Itérateurs infinis …

3 Correction TEA1