nbheap

November 26, 2021

1 Heap

La structure *heap* ou tas en français est utilisée pour trier. Elle peut également servir à obtenir les k premiers éléments d'une liste.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

Un tas est peut être considéré comme un tableau T qui vérifie une condition assez simple, pour tout indice i, alors $T[i] \geqslant \max(T[2i+1], T[2i+2])$. On en déduit nécessairement que le premier élément du tableau est le plus grand. Maintenant comment transformer un tableau en un autre qui respecte cette contrainte ?

[2]:
```
%matplotlib inline
```

1.1 Transformer en tas

[3]:
```
def swap(tab, i, j):
    "Echange deux éléments."
    tab[i], tab[j] = tab[j], tab[i]


def entas(heap):
    "Organise un ensemble selon un tas."
    modif = 1
    while modif > 0:
        modif = 0
        i = len(heap) - 1
        while i > 0:
            root = (i-1) // 2
            if heap[root] < heap[i]:
                swap(heap, root, i)
                modif += 1
            i -= 1
    return heap

ens = [1,2,3,4,7,10,5,6,11,12,3]
entas(ens)
```

[3]:
```
[12, 11, 5, 10, 7, 3, 1, 6, 4, 3, 2]
```

Comme ce n'est pas facile de vérifer que c'est un tas, on le dessine.

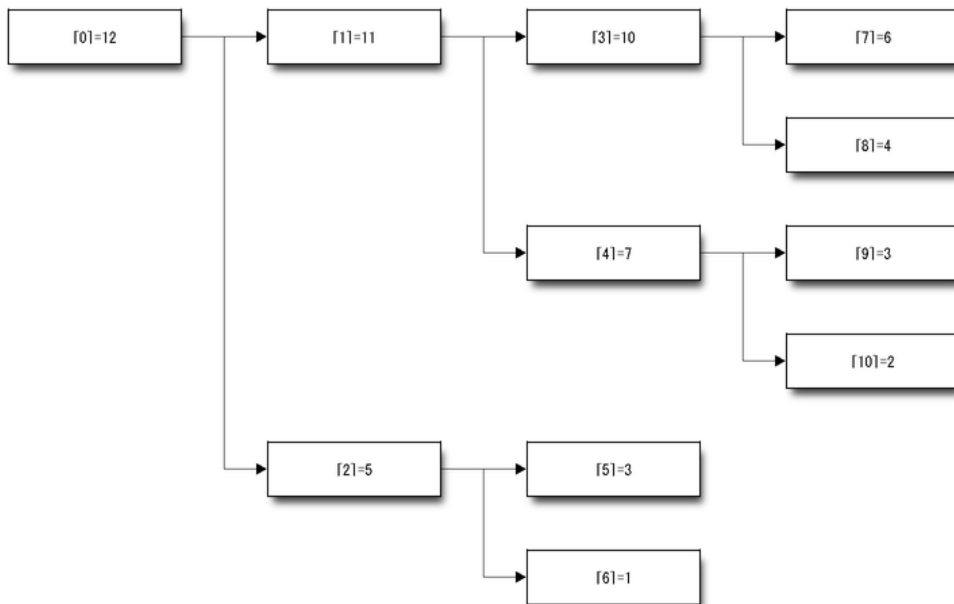
1.2 Dessiner un tas

[4]:
```
from pyensae.graphhelper import draw_diagram

def dessine_tas(heap):
    rows = ["blockdiag {"]
    for i, v in enumerate(heap):
        if i*2+1 < len(heap):
            rows.append('"[{}]={}" -> "[{}]={}";'.format(
                i, heap[i], i * 2 + 1, heap[i*2+1]))
            if i*2+2 < len(heap):
                rows.append('"[{}]={}" -> "[{}]={}";'.format(
                    i, heap[i], i * 2 + 2, heap[i*2+2]))
    rows.append("}")
    return draw_diagram("\n".join(rows))

ens = [1,2,3,4,7,10,5,6,11,12,3]
dessine_tas(entas(ens))
```

[4]:



Le nombre entre crochets est la position, l'autre nombre est la valeur à cette position. Cette représentation fait apparaître une structure d'arbre binaire.

1.3 Première version

[5]:
```
def swap(tab, i, j):
    "Echange deux éléments."
    tab[i], tab[j] = tab[j], tab[i]
```

```python
def _heapify_max_bottom(heap):
    "Organise un ensemble selon un tas."
    modif = 1
    while modif > 0:
        modif = 0
        i = len(heap) - 1
        while i > 0:
            root = (i-1) // 2
            if heap[root] < heap[i]:
                swap(heap, root, i)
                modif += 1
            i -= 1


def _heapify_max_up(heap):
    "Organise un ensemble selon un tas."
    i = 0
    while True:
        left = 2*i + 1
        right = left+1
        if right < len(heap):
            if heap[left] > heap[i] >= heap[right]:
                swap(heap, i, left)
                i = left
            elif heap[right] > heap[i]:
                swap(heap, i, right)
                i = right
            else:
                break
        elif left < len(heap) and heap[left] > heap[i]:
            swap(heap, i, left)
            i = left
        else:
            break


def topk_min(ens, k):
    "Retourne les k plus petits éléments d'un ensemble."

    heap = ens[:k]
    _heapify_max_bottom(heap)

    for el in ens[k:]:
        if el < heap[0]:
            heap[0] = el
            _heapify_max_up(heap)
    return heap


ens = [1,2,3,4,7,10,5,6,11,12,3]
for k in range(1, len(ens)-1):
    print(k, topk_min(ens, k))
```

```
1 [1]
```

```
2 [2, 1]
3 [3, 2, 1]
4 [3, 3, 1, 2]
5 [4, 3, 1, 3, 2]
6 [5, 4, 3, 3, 2, 1]
7 [5, 6, 3, 4, 2, 3, 1]
8 [5, 7, 3, 6, 2, 3, 1, 4]
9 [5, 10, 3, 7, 2, 3, 1, 6, 4]
```

1.4 Même chose avec les indices au lieu des valeurs

[6]:

```
def _heapify_max_bottom_position(ens, pos):
    "Organise un ensemble selon un tas."
    modif = 1
    while modif > 0:
        modif = 0
        i = len(pos) - 1
        while i > 0:
            root = (i-1) // 2
            if ens[pos[root]] < ens[pos[i]]:
                swap(pos, root, i)
                modif += 1
            i -= 1


def _heapify_max_up_position(ens, pos):
    "Organise un ensemble selon un tas."
    i = 0
    while True:
        left = 2*i + 1
        right = left+1
        if right < len(pos):
            if ens[pos[left]] > ens[pos[i]] >= ens[pos[right]]:
                swap(pos, i, left)
                i = left
            elif ens[pos[right]] > ens[pos[i]]:
                swap(pos, i, right)
                i = right
            else:
                break
        elif left < len(pos) and ens[pos[left]] > ens[pos[i]]:
            swap(pos, i, left)
            i = left
        else:
            break


def topk_min_position(ens, k):
    "Retourne les positions des k plus petits éléments d'un ensemble."

    pos = list(range(k))
    _heapify_max_bottom_position(ens, pos)

    for i, el in enumerate(ens[k:]):
```

```
        if el < ens[pos[0]]:
            pos[0] = k + i
            _heapify_max_up_position(ens, pos)
    return pos


ens = [1,2,3,7,10,4,5,6,11,12,3]
for k in range(1, len(ens)-1):
    pos = topk_min_position(ens, k)
    print(k, pos, [ens[i] for i in pos])
```

```
1 [0] [1]
2 [1, 0] [2, 1]
3 [2, 1, 0] [3, 2, 1]
4 [10, 2, 0, 1] [3, 3, 1, 2]
5 [5, 10, 0, 2, 1] [4, 3, 1, 3, 2]
6 [6, 5, 2, 10, 1, 0] [5, 4, 3, 3, 2, 1]
7 [5, 7, 10, 6, 1, 2, 0] [4, 6, 3, 5, 2, 3, 1]
8 [5, 3, 10, 7, 1, 2, 0, 6] [4, 7, 3, 6, 2, 3, 1, 5]
9 [5, 4, 10, 3, 1, 2, 0, 7, 6] [4, 10, 3, 7, 2, 3, 1, 6, 5]
```

[7]:

```
import numpy.random as rnd

X = rnd.randn(10000)

%timeit topk_min(X, 20)
```

```
5.59 ms ± 728 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[8]:

```
%timeit topk_min_position(X, 20)
```

```
7.85 ms ± 544 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.5 Coût de l'algorithme

[9]:

```
from cpyquickhelper.numbers import measure_time
from tqdm import tqdm
from pandas import DataFrame

rows = []
for n in tqdm(list(range(1000, 20001, 1000))):
    X = rnd.randn(n)
    res = measure_time('topk_min_position(X, 100)',
                       {'X': X, 'topk_min_position': topk_min_position},
                       div_by_number=True,
                       number=10)
    res["size"] = n
    rows.append(res)

df = DataFrame(rows)
df.head()
```

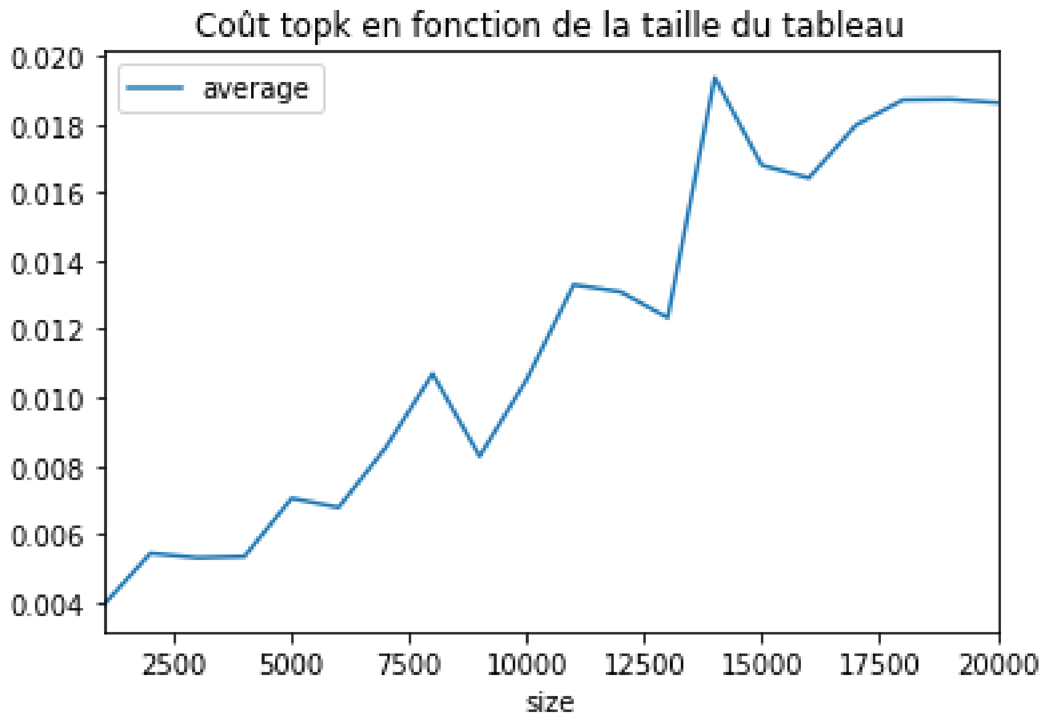
```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 20/20 [00:23<00:00,  1.78s/it]
```

[9]:

	average	deviation	min_exec	max_exec	repeat	number	context_size	size
0	0.003916	0.000363	0.003336	0.004570	10	10	240	1000
1	0.005436	0.001039	0.004235	0.007412	10	10	240	2000
2	0.005306	0.001051	0.004090	0.007401	10	10	240	3000
3	0.005341	0.000830	0.004376	0.007003	10	10	240	4000
4	0.007047	0.001786	0.005223	0.012082	10	10	240	5000

[10]:

```
import matplotlib.pyplot as plt
df[['size', 'average']].set_index('size').plot()
plt.title("Coût topk en fonction de la taille du tableau");
```



A peu près linéaire comme attendu.

[11]:

```
rows = []
X = rnd.randn(10000)
for k in tqdm(list(range(500, 2001, 150))):
    res = measure_time('topk_min_position(X, k)',
                       {'X': X, 'topk_min_position': topk_min_position, 'k': k},
                       div_by_number=True,
                       number=5)
    res["k"] = k
    rows.append(res)

df = DataFrame(rows)
df.head()
```

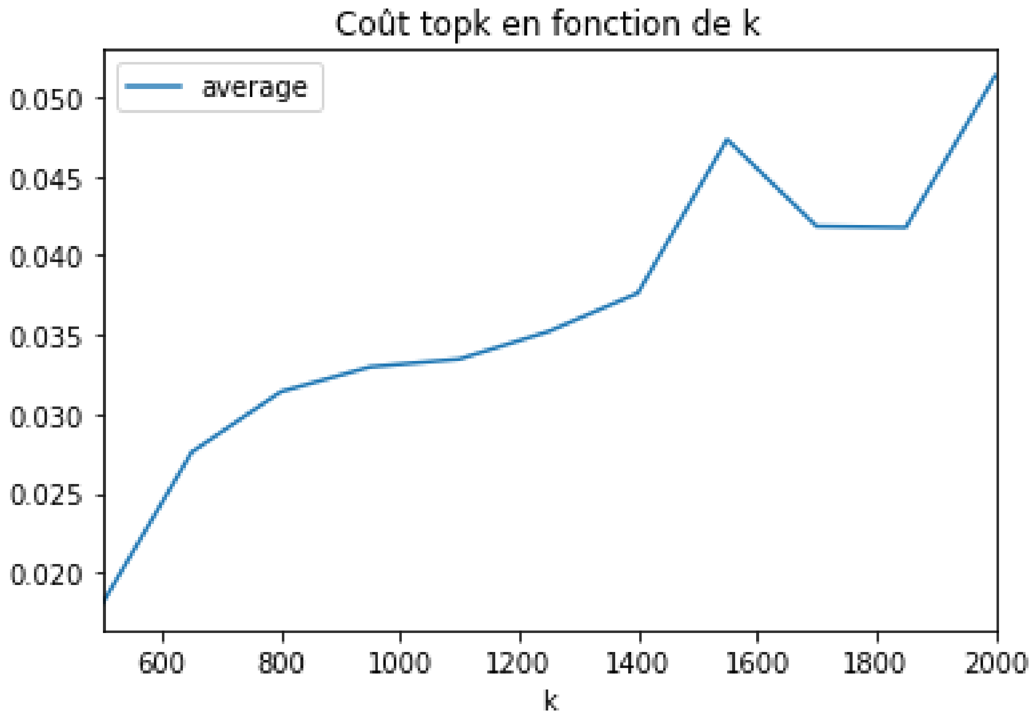
```
  0%|          | 0/11 [00:00<?, ?it/s]
  9%|¿          | 1/11 [00:00<00:09,  1.11it/s]
 18%|¿ ¿         | 2/11 [00:02<00:09,  1.05s/it]
 27%|¿ ¿ ¿        | 3/11 [00:03<00:09,  1.20s/it]
 36%|¿ ¿ ¿ ¿       | 4/11 [00:05<00:09,  1.34s/it]
 45%|¿ ¿ ¿ ¿ ¿      | 5/11 [00:07<00:08,  1.44s/it]
 55%|¿ ¿ ¿ ¿ ¿ ¿     | 6/11 [00:08<00:07,  1.54s/it]
 64%|¿ ¿ ¿ ¿ ¿ ¿ ¿    | 7/11 [00:10<00:06,  1.64s/it]
 73%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿   | 8/11 [00:13<00:05,  1.86s/it]
 82%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿  | 9/11 [00:15<00:03,  1.93s/it]
 91%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿  | 10/11 [00:17<00:01,  1.98s/it]
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 11/11 [00:19<00:00,  2.16s/it]
```

[11]:

	average	deviation	min_exec	max_exec	repeat	number	context_size	k
0	0.018026	0.002823	0.015226	0.025344	10	5	240	500
1	0.027577	0.008949	0.018939	0.043367	10	5	240	650
2	0.031409	0.011282	0.020159	0.056507	10	5	240	800
3	0.032973	0.007518	0.025192	0.047946	10	5	240	950
4	0.033467	0.007725	0.025187	0.051844	10	5	240	1100

[12]:

```
df[['k', 'average']].set_index('k').plot()
plt.title("Coût topk en fonction de k");
```



Pas évident, au pire en $O(n \ln n)$, au mieux en $O(n)$.

1.6 Version simplifiée

A-t-on vraiment besoin de `_heapify_max_bottom_position` ?

[13]:
```
def _heapify_max_up_position_simple(ens, pos, first):
    "Organise un ensemble selon un tas."
    i = first
    while True:
        left = 2*i + 1
        right = left+1
        if right < len(pos):
            if ens[pos[left]] > ens[pos[i]] >= ens[pos[right]]:
                swap(pos, i, left)
                i = left
            elif ens[pos[right]] > ens[pos[i]]:
                swap(pos, i, right)
                i = right
            else:
                break
        elif left < len(pos) and ens[pos[left]] > ens[pos[i]]:
            swap(pos, i, left)
            i = left
        else:
            break


def topk_min_position_simple(ens, k):
    "Retourne les positions des k plus petits éléments d'un ensemble."

    pos = list(range(k))
    pos[k-1] = 0

    for i in range(1, k):
        pos[k-i-1] = i
        _heapify_max_up_position_simple(ens, pos, k-i-1)

    for i, el in enumerate(ens[k:]):
        if el < ens[pos[0]]:
            pos[0] = k + i
            _heapify_max_up_position_simple(ens, pos, 0)
    return pos


ens = [1,2,3,7,10,4,5,6,11,12,3]
for k in range(1, len(ens)-1):
    pos = topk_min_position_simple(ens, k)
    print(k, pos, [ens[i] for i in pos])
```

```
1 [0] [1]
2 [1, 0] [2, 1]
3 [2, 1, 0] [3, 2, 1]
4 [10, 2, 1, 0] [3, 3, 2, 1]
5 [5, 10, 2, 1, 0] [4, 3, 3, 2, 1]
6 [5, 6, 10, 2, 1, 0] [4, 5, 3, 3, 2, 1]
7 [6, 7, 10, 5, 2, 1, 0] [5, 6, 3, 4, 3, 2, 1]
```

```
8 [5, 4, 10, 7, 6, 2, 1, 0] [4, 10, 3, 6, 5, 3, 2, 1]
9 [3, 4, 6, 5, 7, 10, 2, 1, 0] [7, 10, 5, 4, 6, 3, 3, 2, 1]
```

[14]:
```
X = rnd.randn(10000)

%timeit topk_min_position_simple(X, 20)
```

```
7.5 ms ± 810 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[15]:
```

```