Tas_Binomiaux

July 6, 2019

1 Tas binomiaux

Marc Lorenzi
6 juillet 2019

```
In [1]: import matplotlib.pyplot as plt
        import random
        import math
```

```
In [2]: plt.rcParams['figure.figsize'] = (20, 8)
```

Un tas est un ensemble d'objets auxquels sont assignées des **priorités**. On doit pouvoir effectuer sur un tas les opérations suivantes :

- Tester si un tas est vide.
- Déterminer un objet de priorité minimale.
- Insérer un objet de priorité donnée.
- Supprimer un objet de priorité minimale.

Une dernière opération s'avérera ici cruciale. Nous voudrions aussi pouvoir

- Fusionner deux tas.

Il va de soi que l'on désire pouvoir effectuer ces opérations le plus rapidement possible ! Il existe différentes implémentations de la structure de tas. Les tas que nous étudierons ici s'appellent les **tas binomiaux**. Mis à part le test "le tas est-il vide ?", qui s'effectuera en $O(1)$, toutes les autres opérations seront réalisées en temps $O(\lg n)$ où n est le nombre d'objets dans le tas. On pourra également dire mieux pour l'opération d'insertion. Patience ...

Avant de parler de tas binomiaux, intéressons-nous aux **arbres binomiaux**.

1.1 1. Arbres binomiaux

1.1.1 1.1 Les arbres, leur représentation en Python

Nous allons considérer tout au long de ce notebook des arbres. Sans entrer dans une théorie sans fin, un arbre est soit "vide", soit non vide. S'il n'est pas vide, il est composé d'une **racine**, qui possède une valeur, et de zéro, un ou plusieurs **fils** qui sont eux mêmes des arbres.

Nous représenterons en Python

- L'arbre vide par `None`.
- Un arbre non vide de racine de valeur x et de fils $t_0, \ldots, t_{k-1}$ par le couple $(x, [t_0, \ldots, t_{k-1}])$ formé de la valeur de la racine et de la liste des fils (qui peut être vide).

Enfin nous confondrons dans le langage courant la racine avec sa valeur, les fils de l'arbre avec leurs racines, etc. Si peu de rigeur peut faire peur, mais normalement cela ne devrait pas poser de problème.

```
In [3]: def est_vide(t): return t == None
        def racine(t): return t[0]
        def fils(t): return t[1]
```

1.1.2 1.2 Qu'est-ce qu'un arbre binomial ?

Pour tout entier naturel n nous allons définir par récurrence sur n ce qu'est un arbre binomial d'ordre n.

- Un arbre binomial d'ordre 0 est constitué d'une simple racine et n'a pas de fils.
- Soit $n \in \mathbb{N}$. Soit $t = (x, [t_0, \ldots, t_{k-1}])$ un arbre non vide. On dit que t est un arbre binomial d'ordre $n+1$ lorsque
  - t_{k-1} est un arbre binomial d'ordre n.
  - $(x, [t_0, \ldots, t_{k-2}])$ est un arbre binomial d'ordre n.
  - La racine de t_{k-1} a une valeur supérieure ou égale à x.

Un arbre binomial d'ordre $n+1$ est donc fabriqué en "recollant" deux arbres binomiaux d'ordre n. Voici la fonction qui colle les arbres :

```
In [4]: def coller_arbres(t1, t2):
            if racine(t1) <= racine(t2):
                fils(t1).append(t2)
                return t1
            else:
                fils(t2).append(t1)
                return t2
```

```
In [5]: t1 = coller_arbres((4,[]),(7,[]))
        t2 = coller_arbres((5,[]),(3,[]))
        t3 = coller_arbres((1,[]),(2,[]))
        t4 = coller_arbres((6,[]),(8,[]))
        t = coller_arbres(t1, t2)
        u = coller_arbres(t3, t4)
        v = coller_arbres(t, u)
        print(v)
```

```
(1, [(2, []), (6, [(8, [])]), (3, [(5, []), (4, [(7, [])])])])
```

Remarque : La fonction ci-dessus modifie **physiquement** l'arbre t_1 (ou t_2, cela dépend). Pour ce que nous allons faire dans ce notebook cela ne pose pas de problème. En revanche, dans certaines applications cela peut être dramatique : dans l'opération de recollement, l'un des deux arbres est perdu irrémédiablement, et en plus on ne sait pas lequel.

Proposition : La complexité de la fonction `coller_arbres` est $O(1)$.

Démonstration : Évident, à condition d'admettre que la fonction Python `append` s'exécute en $O(1)$. En réalité ce n'est pas tout à fait exact, mais ceci est une autre histoire ...

Histoire de pouvoir manipuler de "gros" arbres binomiaux, voici une fonction qui renvoie un arbre binomial d'ordre n. La valeur des noeuds n'a pas d'importance, j'ai mis 0 parce qu'il fallait mettre quelque chose.

```
In [6]: def test_binomial(n):
            if n == 0: return (0, [])
            else:
                t1 = test_binomial(n - 1)
                t2 = test_binomial(n - 1)
                return coller_arbres(t1, t2)
```

```
In [7]: print(test_binomial(6))
```

```
(0, [(0, []), (0, [(0, [])]), (0, [(0, []), (0, [(0, [])])]), (0, [(0, []), (0, [(0, [])]), (0,
```

Beurk. Écrivons illico une fonction qui écrit un arbre t de façon un peu plus jolie.

- Si t est vide, la fonction affiche _.
- Sinon, elle affiche
  - La racine de t,
  - puis, entre parenthèses et séparés par des virgules, les fils de la racine.

```
In [8]: def arbre_vers_chaine(t):
            if t == None: s = '_'
            else:
                s = str(racine(t))
                fs = fils(t)
                if fs != []:
                    s += '('
                    n = len(fs)
                    for k in range(n):
                        s += arbre_vers_chaine(fs[k])
                        if k < n - 1: s += ','
                    s += ')'
                return s
```

```
In [9]: def print_arbre(t): print(arbre_vers_chaine(t))
```

```
In [10]: print_arbre(test_binomial(6))
```

```
0(0,0(0),0(0,0(0)),0(0,0(0),0(0,0(0))),0(0,0(0),0(0,0(0)),0(0,0(0),0(0,0(0)))),0(0,0(0),0(0,0(0)
```

Ah oui, c'est beaucoup mieux. Comme nous pouvons le voir sur l'exemple ci-dessus :-) nous avons le résultat suivant :

Proposition : Soit t un arbre binomial d'ordre n. (La racine de) t a exactement n fils.

Démonstration : Faisons une récurrence sur n.

- Pour $n = 0$, c'est évident. La racine d'un arbre binomial d'ordre 0 n'a pas de fils.
- Soit $n \in \mathbb{N}$. Supposons la propriété vraie pour n. Soit t un arbre binomial d'ordre $n+1$. t a été obtenu en recollant deux arbres binomiaux d'ordre n, t_1 et t_2. Pour fixer les idées, supposons que la racine de t_2 est inférieure à celle de t_1. Les fils de la racine de t sont donc :
  - Les fils de t_2 : il y en a n par l'hypothèse de récurrence.
  - t_1.

La racine de t a donc $n+1$ fils. On en déduit une fonction `ordre` qui renvoie en temps $O(1)$ l'ordre d'un tas binomial.

```
In [11]: def ordre(t): return len(fils(t))

In [12]: ordre(test_binomial(10))

Out[12]: 10
```

Pour s'entraîner aux récurrences et se familiariser avec les arbres binomiaux, voici deux exercices faciles.

Exercice Montrez que la racine d'un arbre binomial est le plus petit noeud de l'arbre.

Exercice : Soit t un arbre binomial d'ordre n. Montrez que les n fils de t sont, de gauche à droite, un arbre binomial d'ordre 0, un arbre binomial d'ordre 1, ..., un arbre binomial d'ordre $n-1$.

Si vous regardez le résultat affiché par `print_arbre(test_binomial(6))`, vous pouvez avoir un micro-doute : ceci est-il bien un arbre binomial ? Écrivons une fonction qui prend un arbre en paramètre et renvoie `True` si c'est un arbre binomial et `False` sinon.

```
In [13]: def est_arbre_binomial(t):
             if t[1] == []: return True
             else:
                 t1 = (racine(t), fils(t)[:-1])
                 t2 = fils(t)[-1]
                 return est_arbre_binomial(t1) and est_arbre_binomial(t2) \
                     and racine(t1) <= racine(t2) and ordre(t1) == ordre(t2)

In [14]: est_arbre_binomial(test_binomial(6))

Out[14]: True
```

1.1.3 1.3 Hauteur d'un arbre binomial

Définition : La hauteur $h(t)$ d'un arbre t est définie inductivement comme suit :

- L'arbre vide a pour hauteur 0.

- Si t n'a pas de fils, $h(t) = 1$.
- Si t est un arbre possédant k fils $t_0, \ldots, t_{k-1}$,

$$h(t) = 1 + \max(h(t_0), \ldots, h(t_{k-1}))$$

Proposition : La hauteur d'un arbre binomial d'ordre n est $n + 1$.
Démonstration : Une récurrence sur n, encore !

- Un arbre binomial d'ordre 0 est de hauteur 1.
- Soient t_1, t_2 deux arbres binomiaux d'ordre n. Soit t l'arbre obtenu en recollant t_1 et t_2. Les fils de t sont, par exemple, t_1 et les fils de t_2. Par l'hypothèse de récurrence, $h(t_1) = n + 1$ et les fils de t_2 sont de hauteur inférieure ou égale à n, puisque $h(t_2) = n + 1$. Donc, le max des hauteurs des fils de t est $n + 1$ (la hauteur de t_1) et ainsi, $h(t) = n + 2$.

```
In [15]: def hauteur(t):
             if t == None: return 0
             else:
                 fs = fils(t)
                 h = 0
                 for f in fs:
                     h = max(h, hauteur(f))
                 return h + 1

In [16]: print(hauteur(test_binomial(16)))

17
```

1.1.4 1.4 Représentation graphique

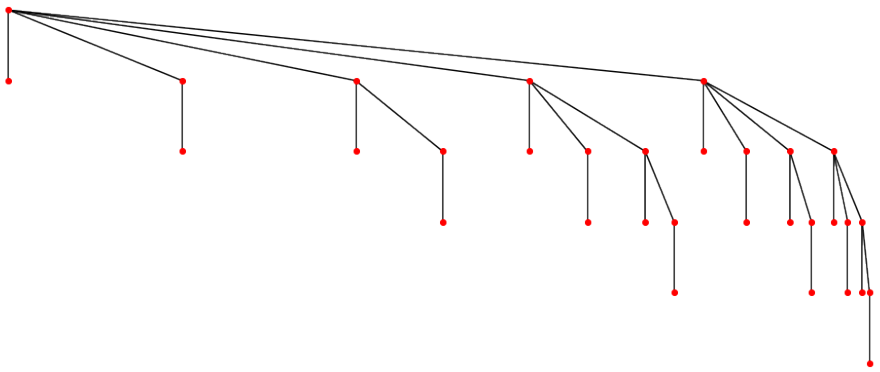
Il est très intéressant de visulaiser les arbres binomiausx, car ils ont des formes très remarquables. Je n'entre pas dans le détails des deux fonctions ci-dessous. La fonction `dessin_aux` est appelée par `dessin_arbre_binomial` pour dessiner l'allure d'un arbre binomial d'ordre n.

```
In [17]: def dessin_aux(t, xmin, xmax, ymin, ymax, dy):
             r = racine(t)
             fs = fils(t)
             n = len(fs)
             xm = (xmin + xmax) / 2
             for k in range(n):
                 x1 = xmin + k * (xmax - xmin) / n
                 x2 = xmin + (k + 1) * (xmax - xmin) / n
                 xmm = (x1 + x2) / 2
                 plt.plot([xmin, x1], [ymax, ymax - dy], 'k')
                 dessin_aux(fs[k], x1, x2, ymin, ymax - dy, dy)
             plt.plot([xmin], [ymax], 'or', markersize=6)
```

```
In [18]: def dessin_arbre_binomial(n):
             t = test_binomial(n)
             h = hauteur(t)
             dy = 10 / h
             plt.xlim(0, 10)
             plt.ylim(0, 10)
             plt.axis('off')
             dessin_aux(t, 1, 9, 1, 9, dy)
```

Voici par exemple un arbre binomial d'ordre 5.

```
In [19]: dessin_arbre_binomial(5)
```



Si vous avez fait les exercices proposés, vous savez que avez en prime, sous la racine, des arbres binomiaux d'ordres 0, 1, 2, 3 et 4. Et sous les racines de ces arbres, d'autres arbres binomiaux. Etc.

Combien de noeuds (c'est à dire de points rouges) ? Il y en a 32.

Combien de noeuds à chaque niveau ? 1, 5, 10, 10, 5, 1. Bigre, sont-ce des coefficients binomiaux ? La reste de cette section est là pour prouver que les tas binomiaux portent bien leur nom.

1.1.5 1.6 Nombre de noeuds d'un arbre binomial

Proposition : Un arbre binomial d'ordre n possède 2^n noeuds.

Démonstration : On fait une récurrence sur n.

- Un arbre binomial d'ordre 0 a $1 = 2^0$ noeud.
- Supposons la propriété vraie pour l'entier n. Soit t un arbre binomial d'ordre $n+1$. L'arbre t est obtenu en recollant deux arbres binomiaux d'ordre n. Par l'hypothèse de récurrence, chacun de ces deux arbres possède 2^n noeuds. Le nombre de noeuds de t est donc

$$2^n + 2^n = 2^{n+1}$$

```
In [20]: def nombre_noeuds(t):
             if t == None: return 0
             else:
                 n = 1
                 for f in fils(t):
                     n = n + nombre_noeuds(f)
                 return n

In [21]: nombre_noeuds(test_binomial(16))

Out[21]: 65536
```

1.1.6 1.6 Nombre de noeuds de profondeur donnée

Nous y voici. Nous allons enfin comprendre pourquoi un arbre binomial est appelé "binomial".

La profondeur d'un noeud dans l'arbre t est la distance de la racine de t à ce noeud. Pour faire concret, voyez le dessin de l'arbre. Comptez le nombre de traits entre la racine de l'arbre et le noeud qui vous intéresse : c'est cela, la profondeur. Ainsi, la racine est de profondeur 0, les (racines des) fils de la racine sont de profondeur 1, etc. Un arbre binomial d'ordre n est de hauteur $n+1$. Il possède donc des noeuds aux profondeurs $0, 1, \ldots, n$. Combien un arbre binomial a-t-il de noeuds à une profondeur donnée ?

Proposition : Soit t un arbre binomial d'ordre n. Soit $0 \leq k \leq n$. À la profondeur k, l'arbre t possède exactement $\binom{n}{k}$ noeuds.

On comprend enfin pourquoi un tel arbre est dit binomial.

Démonstration : Une récurrence s'impose.

- Pour $n = 0$ c'est évident.
- Soit $n \in \mathbb{N}$. Supposons la propriété vraie pour n. Soit t un arbre binomial d'ordre $n+1$, obtenu en recollant deux arbres binomiaux t_1 et t_2 d'ordre n. Par exemple, les fils de t sont les fils de t_2, suivis de t_1 (et la racine de t est celle de t_2). Soit $1 \leq k \leq n$. Les noeuds de t à la profondeur k sont :
  - Les noeuds de t_1 à la profondeur $k-1$. Il y en a $\binom{n}{k-1}$ par l'hypothèse de récurrence.
  - Les noeuds de t_2 à la profondeur k. Il y en a $\binom{n}{k}$ par l'hypothèse de récurrence.

Le nombre de noeuds de t à la profondeur k est donc

$$\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$$

Exercice : Il reste à voir ce qui se passe pour $k = 0$ et $k = n+1$.

Corollaire : $\sum_{k=0}^{n} \binom{n}{k} = 2^n$.

Démonstration : Additionnez les nombres de noeuds à chaque profondeur, vous obtiendrez le nombre total de noeuds de l'arbre.

1.2 2. Tas binomiaux

Rappelons-nous que les éléments d'un tas sont des **objets** munis d'une **priorité**. Pour simplifier la présentation qui va suivre, nous allons laisser de côté les objets. Les éléments d'un tas seront donc simplement des priorités, que nous supposerons être des **entiers**. Le lecteur qui veut à tout prix que son tas contienne aussi des objets remplacera tous les éléments du tas par des couples formés d'un objet et d'un entier.

1.2.1 2.1 C'est quoi ?

Définition : Soit $n \geq 0$. Soit $h = [t_0, \ldots, t_n]$ une liste d'arbres. On dit que h est un tas binomial lorsque, pour tout entier $0 \leq k \leq n$, t_k est soit l'arbre vide, soit un tas binomial d'ordre k. Nous ferons de plus l'hypothèse que t_n n'est pas vide.

On convient également que la liste vide [] est un tas binomial. Nous l'appellerons le tas vide.

L'entier $n+1$ est appelé la longueur du tas. Le tas vide est de longueur 0.

Notation : Pour un tas h ou un arbre t nous noterons $|h|$ (ou $|t|$) le nombre de noeuds du tas (ou de l'arbre).

Proposition : Soit $n \in \mathbb{N}$. Soit h un tas binomial de longueur $n+1$. On a

$$2^n \leq |h| < 2^{n+1}$$

et donc

$$n = \lfloor \lg |h| \rfloor$$

où lg désigne le logarithme en base 2 et les crochets désignent la partie entière.

Démonstration : Pour $k = 0, \ldots, n$, posons $a_k = 0$ si t_k est vide, et $a_k = 1$ sinon. Rappelons nous que $a_n = 1$ puisque t_n n'est pas vide. On a

$$|h| = \sum_{k=0}^{n} |t_k| = \sum_{k=0}^{n} a_k 2^k$$

Majorons :

$$|h| \leq \sum_{k=0}^{n} 2^k = 2^{n+1} - 1 < 2^{n+1}$$

Minorons :

$$|h| \geq a_n 2^n = 2^n$$

En passant au logarithme, on obtient

$$n \leq \lg |h| < n+1$$

et comme n est un entier on a bien le résultat souhaité.

1.2.2 2.2 Le plus petit noeud d'un tas binomial

Si vous êtes un lecteur consciencieux et que vous avez fait les exercices, alors vous savez que le plus petit noeud d'un **arbre** binomial est sa racine. Le plus petit noeud d'un tas binomial est donc l'une des racines des arbres dont il est constitué.

```
In [22]: def minimum(h):
             n = len(h)
             if n == 0:
                 raise Exception('Tas vide')
             else:
                 m = racine(h[n - 1])
                 for k in range(n - 1):
                     if h[k] != None: m = min(m, racine(h[k]))
                 return m
```

Remarque : Quelle est la complexité de la fonction `minimum` ? C'est clairement un $O(n)$ où n est la longueur du tas. Mais nous avons vu que n est la partie entière de $\lg |h|$. Ainsi, on peut obtenir le minimum du tas h en temps $O(\lg |h|)$.

1.2.3 2.3 Et ensuite ?

Le résultat du paragraphe précédent est assez prometteur. Nous allons voir maintenant comment effectuer les opérations suivantes :

- Insertion d'un objet dans un tas binomial.
- Fusion de deux tas binomiaux.
- Suppression d'un objet dans un tas binomial.

Et nous allons voir que toutes ces opérations peuvent se faire en temps logarithmique en le nombre de noeuds des tas.

1.3 3. Insertion

1.3.1 3.1 successeur d'un entier naturel

Soit $a \in \mathbb{N}^*$. Écrivons a en base 2 :

$$a = \sum_{k=0}^{n-1} a_k 2^k$$

où les a_k valent 0 ou 1 et $n \in \mathbb{N}$. Il est bien connu qu'une telle écriture est unique, à condition de supposer $a_{n-1} = 1$. Que vaut $a + 1$, le successeur de a ? Représentons a par la liste $[a_0, \ldots, a_{n-1}]$ et, histoire d'être complets, représentons l'entier 0 par la liste vide. Pour obtenir $a + 1$, on ajoute 1 à a_0. Deux cas se présentent :

- Si $a_0 = 0$, alors on a fini : $a + 1 = [a_0 + 1, a_1, \ldots, a_{n-1}]$.
- Si $a_0 = 1$, on a une "retenue" $r = 1$, et on calcule $a_1 + 1$. La discussion recommence ensuite sur a_1. Selon qu'il est égal à 1 ou 0, on a une retenue ou pas. S'il y a une retenue on continue, sinon on s'arrête.

Voici le code de la fonction `successeur`. On passe en paramètre la liste des chiffres binaires de l'entier a.

```
In [23]: def successeur(a):
             n = len(a)
             r = 1
             k = 0
             while k < n and r == 1:
                 if a[k] == 0:
                     a[k] = 1
                     r = 0
                 else:
                     a[k] = 0
                     r = 1
                 k = k + 1
             if r == 1: a.append(1)

In [24]: a = [1, 1, 0, 1]
         successeur(a)
         print(a)

[0, 0, 1, 1]


In [25]: a = [1, 1, 1, 1]
         successeur(a)
         print(a)

[0, 0, 0, 0, 1]
```

1.3.2 3.2 Insertion dans un tas binomial

Maintenant, insérons un objet x dans un tas binomial h. Le principe est le même, on désire d'une certaine façon "incrémenter" h.

On pose $r =$ l'abre binomial d'ordre 0 dont la racine est x.

- Si $h[0]$ est vide, on remplace $h[0]$ par r et on a fini.
- Sinon,
  - On recolle les arbres r et $h[0]$, et on appelle r l'arbre d'ordre 1 obtenu. On a d'une certaine façon une "retenue".
  - On pose $h[0] =$`None`.
  - On recommence pour $h[1]$.

Voici le code de la fonction `inserer`. C'est un copier-coller de la fonction d'incrémentation des entiers naturels, où on interprète 0 comme l'arbre vide et l'addition comme un recollement d'arbres.

```
In [26]: def inserer(x, h):
             r = (x, [])
             k = 0
             n = len(h)
             while k < n and r != None:
                 if h[k] == None:
                     h[k] = r
                     r = None
                 else:
                     r = coller_arbres(r, h[k])
                     h[k] = None
                 k = k + 1
             if r != None: h.append(r)
```

Un petit test ?

```
In [27]: h = []
         for k in range(100):
             inserer(k, h)
         print(h)
```

```
[None, None, (96, [(97, []), (98, [(99, [])])]), None, None, (64, [(65, []), (66, [(67, [])]), (
```

Ce n'est pas trop lisible, alors voici une fonction qui affiche les tas un peu plus joliment.

```
In [28]: def print_tas(h):
             s = ''
             n = len(h)
             for k in range(n):
                 if h[k] == None: s += str(k) + ': ' + '_'
                 else: s += str(k) + ': ' + arbre_vers_chaine(h[k])
                 s += '\n'
             if n == 0: print('_')
             else: print(s)
```

```
In [29]: print_tas(h)
```

```
0: _
1: _
2: 96(97,98(99))
3: _
4: _
5: 64(65,66(67),68(69,70(71)),72(73,74(75),76(77,78(79))),80(81,82(83),84(85,86(87)),88(89,90(91
6: 0(1,2(3),4(5,6(7)),8(9,10(11),12(13,14(15))),16(17,18(19),20(21,22(23)),24(25,26(27),28(29,30
```

Remarquez l'analogie avec les entiers. En base 2, 100 = 0010011. Regardez la position des arbres vides dans le tas h, elles sont aux indices où il y a des zéros dans la représentation binaire de 100.

Proposition : Soit h un tas binomial ayant $|h| = N$ éléments. Soit $N = \sum_{k=0}^{n-1} a_k 2^k$ la représentation de N en base 2. Pour tout k entre 0 et $n-1$, $h[k]$ est vide si et seulement si $a_k = 0$.

Démonstration : Le nombre d'éléments de h est précisément $\sum_{k=0}^{n-1} |h[k]|$, et $|h[k]|$ est égal à 0 ou à 2^k, puisqu'il est soit l'arbre vide soit un arbre binomial d'ordre k. On conclut par l'unicité de l'écriture des entiers en base 2.

1.3.3 3.3 Tas binomial "aléatoire"

Pour fabriquer un tas "aléatoire" contenant les éléments d'une liste s

- On mélange la liste
- On insère successivement dans un tas initialement vide les éléments de la liste mélangée.

```
In [30]: def random_tas(s):
             random.shuffle(s)
             h = []
             for x in s:
                 inserer(x, h)
             return h
```

```
In [31]: print_tas(random_tas(list(range(197))))
```

```
0: 162
1: _
2: 36(76,74(78))
3: _
4: _
5: _
6: 1(164,97(138),16(30,98(147)),5(139,27(101),46(189,71(87))),14(59,29(110),114(167,122(150)),23
7: 0(42,115(132),70(156,121(149)),95(99,123(169),130(176,141(142))),12(25,152(190),117(172,118(1
```

1.3.4 3.4 Est-ce bien un tas binomial ?

Il est temps de nous demander si le code que nous avons écrit fonctionne réellement. Voici tout d'abord une fonction `est_tas` qui renvoie `True` si son paramètre est un tas binomial, et `False` sinon.

```
In [32]: def est_tas_binomial(h):
             n = len(h)
             b = True
             for k in range(n):
                 b = b and (h[k] == None or (est_arbre_binomial(h[k]) and ordre(h[k]) == k))
                 if b == False: print(k)
             return b
```

```
In [33]: est_tas_binomial(random_tas(list(range(100000))))

Out[33]: True
```

Après 10^5 appels à `inserer` nous obtenons bien un tas binomial, avec 10^5 éléments. S'il y a une erreur dans notre code elle est subtile :-).

1.3.5 3.5 Complexité de la fonction d'insertion

Clairement, si h est un tas de longueur n, l'insertion d'un élément dans h se fait en temps $O(n)$ (rappelons-nous que `coller_arbres` se fait en temps constant). Comme n est la partie entière de $\lg |h|$, on a donc pour la fonction `inserer` une complexité en $O(\lg |h|)$.

Cependant on peut dire encore mieux.

Proposition : L'insertion de n objets dans un tas initialement vide se fait en temps $O(n)$.

Démonstration : Pour l'instant nous avons été assez vagues sur les calcul de complexité. Quelles ont les opérations élémentaires mises en jeu ? Ici, nous allons devoir être plus précis : je veux bien compter mais à condition que l'on me dise **quoi compter**. Rappelons-nous le code de la fonction `inserer` :

```
def inserer(x, h):
    r = (x, [])
    k = 0
    n = len(h)
    while k < n and r != None:
        if h[k] == None:
            h[k] = r
            r = None
        else:
            r = coller_arbres(r, h[k])
            h[k] = None
        k = k + 1
    if r != None: h.append(r)
```

Nous appellerons complexité de `inserer` le nombre d'appels à `coller_arbres`. Cette complexité est très précisément le nombre d'arbres non vides au début de la liste h. Ce choix de la complexité pourrait être discuté (et discutable), mais continuons ainsi.

En insérant n objets dans un tas initialement vide, on applique successivement la fonction `inserer` sur des tas ayant $0, 1, \ldots, n-1$ éléments. Rappelons nous que pour savoir quels sont les arbres dans un tas qui sont vides, il suffit de regarder les chiffres en base 2 du nombre d'éléments du tas : **la complexité de `inserer` sur un tas de taille k est le nombre de 1 au début du développement binaire de k.**

Notons C_k la complexité de l'insertion sur un tas de taille k. La complexité de n insertions successives à partir du tas vide est donc

$$\mathcal{C} = \sum_{k=0}^{n-1} C_k$$

Réorganisons cette somme par "C_k constants". La somme qui apparaît ci-dessous est en réalité une somme finie, comme nous le verrons plus loin, ou comme vous le voyez déjà.

$$\mathcal{C} = \sum_{j=0}^{\infty} \sum_{0 \leq k < n, C_k = j} j = \sum_{j=0}^{\infty} j\lambda_j$$

où λ_j est le nombre d'entiers k, $0 \leq k \leq n-1$, tels que $C_k = j$. Que vaut λ_j ? Combien y a-t-il d'entiers entre 0 et $n-1$ commençant en base 2 par exactement j uns ? En remarquant qu'en base 2, $11\ldots10 = 2^j - 1$ (j uns suivis d'un zéro), la question devient : combien y a-t-il d'entiers de la forme $2^j - 1 + 2^{j+1}p$ entre 0 et $n-1$?

Le lecteur est chargé de vérifier que

- Si $2^j > n$ il n'y en a pas.
- Si $2^j \leq n$, c'est à dire $j \leq \lfloor \lg n \rfloor$, il y en a $\left\lfloor \frac{n}{2^{j+1}} + \frac{1}{2} \right\rfloor$.

Ainsi,

$$\mathcal{C} = \sum_{j=0}^{\lfloor \lg n \rfloor} j \left\lfloor \frac{n}{2^{j+1}} + \frac{1}{2} \right\rfloor$$

```
In [34]: def complexite_theorique(n):
             s = 0
             p = math.floor(math.log(n, 2))
             for j in range(p + 1):
                 s = s + j * math.floor(n / 2 ** (j + 1) + 1/ 2)
             return s

In [35]: complexite_theorique(100)

Out[35]: 97
```

Majorons l'horrible somme $\mathcal{C}$ ci-dessus. On a

$$\mathcal{C} \leq \sum_{j=0}^{\lfloor \lg n \rfloor} j \left(\frac{n}{2^{j+1}} + \frac{1}{2} \right)$$

D'une part,

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{j}{2} \leq \frac{1}{2} \lfloor \lg n \rfloor^2 = o(n)$$

D'autre part,

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{nj}{2^{j+1}} \leq Sn$$

où

$$S = \sum_{j=0}^{\infty} \frac{j}{2^{j+1}}$$

est la somme d'une série convergente. Ainsi, on a

$$\mathcal{C} \leq Sn + o(n)$$

et donc, bien évidemment,

$$\mathcal{C} = O(n)$$

1.3.6 3.6 Tests

Les calculs que nous venons de faire, personne n'y croit n'est-ce pas ? Avec toutes ces parties entières de fractions de logarithmes de puissances de 2, on a bien dû se tromper quelque part ??? Alors faisons quelques tests.

Définissons tout d'abord un compteur global.

```
In [36]: compteur = 0
```

Modifions légèrement la fonction `inserer` pour qu'elle incrémente le compteur à chaque appel à `coller_arbres`.

```
In [37]: def inserer_test(x, h):
             global compteur
             r = (x, [])
             k = 0
             n = len(h)
             while k < n and r != None:
                 if h[k] == None:
                     h[k] = r
                     r = None
                 else:
                     r = coller_arbres(r, h[k])
                     compteur += 1
                     h[k] = None
                 k = k + 1
             if r != None: h.append(r)
```

Écrivons enfin une fonction qui renvoie la liste des valeurs du compteur lors de n insertions dans un tas initialement vide. Ou, si l'on préfère les valeurs du compteur divisées par le numéro de l'insertion, c'est à dire les complexités "moyennes" pour k insertions.

```
In [38]: def test_complexite(n, moy=False):
             global compteur
             compteur = 0
             h = []
             cs = []
             for k in range(1, n + 1):
                 inserer_test(0, h)
                 if moy: cs.append(compteur / k)
                 else: cs.append(compteur)
             return cs
```

```
In [39]: cs1 = test_complexite(200)
         print(cs1)
```

```
[0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, 11, 15, 15, 16, 16, 18, 18, 19, 19, 22, 22, 23, 23
```

```
In [40]: cs2 = [complexite_theorique(k) for k in range(1, 201)]
         print(cs2)
```

```
[0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, 11, 15, 15, 16, 16, 18, 18, 19, 19, 22, 22, 23, 23
```

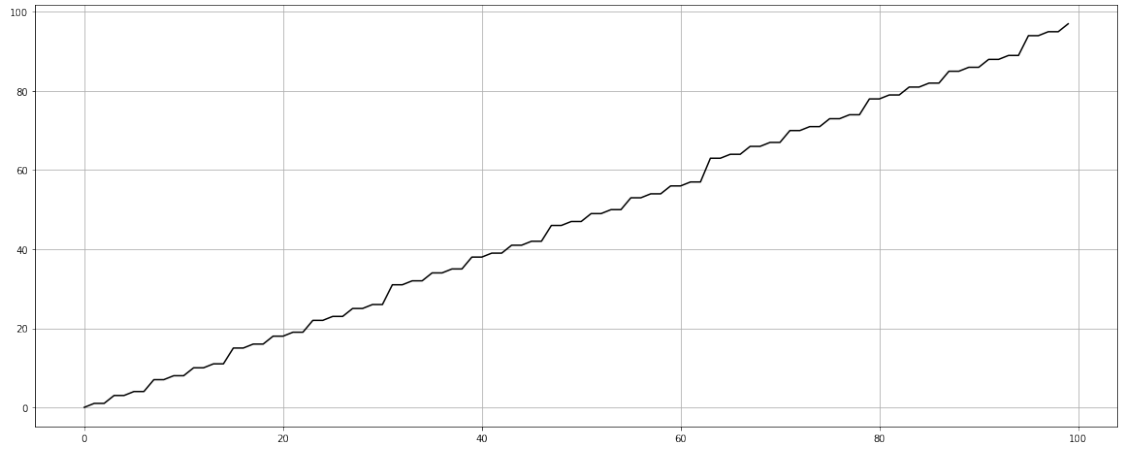
```
In [41]: cs1 == cs2
```

```
Out[41]: True
```

Youpi, théorie rigoureusement identique à la réalité. Les mauvaises langues s'en souviendront la prochaine fois :-).

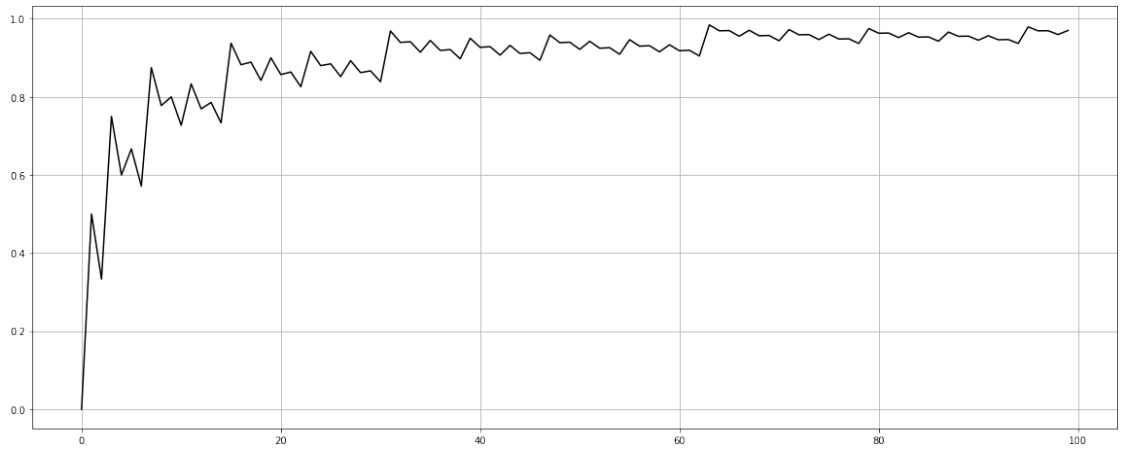
Dessinons ...

```
In [42]: cs = test_complexite(100, moy=False)
         plt.plot(cs, 'k')
         plt.grid()
```



Dessinons aussi la courbe de la "complexité moyenne".

```
In [43]: cs = test_complexite(100, moy=True)
         plt.plot(cs, 'k')
         plt.grid()
```



Exercice : Réévaluez les deux cellules ci-dessus pour 10000 insertions. Une idée de la valeur de S, la somme de la série vue plus haut ? Grosse flemme ? Lisez la suite :-).

Proposition :

$$\sum_{j=0}^{\infty} \frac{j}{2^{j+1}} = 1$$

Démonstration : Considérons la série entière

$$f(x) = \frac{1}{2} \sum_{j=0}^{\infty} \left(\frac{x}{2}\right)^j$$

Cette série a pour rayon de convergence 2. La fonction f est donc dérivable sur $]-2,2[$ et on a pour tout $x \in]-2,2[$

$$f'(x) = \frac{1}{2} \sum_{j=1}^{\infty} \frac{j}{2^j} x^{j-1}$$

et donc

$$\sum_{j=0}^{\infty} \frac{j}{2^{j+1}} = f'(1)$$

Par ailleurs,

$$f(x) = \frac{1}{2} \frac{1}{1-\frac{x}{2}} = \frac{1}{2-x}$$

donc

$$f'(x) = \frac{1}{(2-x)^2}$$

d'où $f'(1) = 1$.

1.4 4. Fusion de deux tas binomiaux

1.4.1 4.1 L'addition des entiers naturels

Avant de fusionner des tas, faisons quelques petits rappels sur ... l'addition des entiers naturels.

Tout entier naturel a s'écrit de façon unique

$$a = \sum_{k=0}^{\infty} a_k 2^k$$

où les $a_k \in \{0, 1\}$ sont tous nuls sauf un nombre fini.

Soient $a = \sum_{k=0}^{\infty} a_k 2^k$ et $b = \sum_{k=0}^{\infty} b_k 2^k$ deux entiers. Soit $c = a + b = \sum_{k=0}^{\infty} c_k 2^k$. Que valent les c_k en fonction des a_k et des b_k ? C'est facile, mais attention aux retenues.

Posons $r_0 = 0$ et, pour tout $k \geq 0$, $r_{k+1} = (a_k + b_k) \div 2$ où le symbole $\div$ désigne le quotient de la division euclidienne. On a alors

$$\forall k \in \mathbb{N}, c_k = (a_k + b_k + r_k) \bmod 2$$

Pour préparer le terrain pour la fusion des tas, il y a 8 cas, résumés dans le tableau ci-dessous.

	a_k	b_k	r_k	$\Rightarrow$	c_k	r_{k+1}
1	0	0	0		0	0
2	0	1	0		1	0
3	1	0	0		1	0
4	1	1	0		0	1
5	0	0	1		1	0
6	1	0	1		0	1
7	0	1	1		0	1
8	1	1	1		1	1

Représentons les entiers naturels en Python par la liste de leurs chiffres binaires, le chiffre des unités étant le premier élément de la liste. Bien évidemment nos listes sont finies et notre belle description théorique est un peu mise à mal. Alors commençons par définir une fonction `chiffre`. Elle prend en paramètres une liste a d'entiers et un entier k.

- Si k est strictement inférieur à la longueur de a, la fonction renvoie $a[k]$.
- Sinon, la fonction renvoie 0.

Avec cette petite astuce tout se passe donc comme si a était une liste infinie.

```
In [44]: def chiffre(a, k):
             if k < len(a): return a[k]
             else: return 0
```

Voici maintenant la fonction `somme_entiers`. Elle reprend ce qui a été dit plus haut. Remarquez que la longueur de $a + b$ est le maximum des longueurs de a et de b, plus, peut-être, 1, s' il y a une retenue sur la somme des derniers chiffres.

```
In [45]: def somme_entiers(a, b):
             n = max(len(a), len(b))
             c = n * [0]
```

```
            r = 0
            for k in range(n):
                ak = chiffre(a, k)
                bk = chiffre(b, k)
                w = (ak, bk, r)
                if   w == (0, 0, 0): c[k], r = 0, 0
                elif w == (0, 1, 0): c[k], r = 1, 0
                elif w == (1, 0, 0): c[k], r = 1, 0
                elif w == (1, 1, 0): c[k], r = 0, 1
                elif w == (0, 0, 1): c[k], r = 1, 0
                elif w == (1, 0, 1): c[k], r = 0, 1
                elif w == (0, 1, 1): c[k], r = 0, 1
                else:                c[k], r = 1, 1
            if r != 0: c.append(r)
            return c
```

```
In [46]: somme_entiers([0, 1, 1], [1, 1, 1])
```

```
Out[46]: [1, 0, 1, 1]
```

1.4.2 4.2 Fusion

Pourquoi cette digression sur la somme de deux entiers naturels, me direz vous ? Eh bien parce que un tas binomial "ressemble" beaucoup à un entier naturel. La fonction `fusionner` aura une forte ressemblance avec `somme_entiers`.

La fonction ci-dessous est à comparer à la fonction `chiffre` ...

```
In [47]: def element_tas(a, k):
             if k < len(a): return a[k]
             else: return None
```

Et voici la fonction de fusion. Soient $a = (a_k)_{k\geq 0}$ et $b = (b_k)_{k\geq 0}$ deux tas binomiaux où, pour tout entier k, a_k et b_k sont des arbres binomiaux d'ordre k, ou l'arbre vide. On suppose également, cela va de soi, que les a_k et les b_k sont tous vides sauf un nombre fini. Comment fusionner a et b ? Notons $c = (c_k)_{k\geq 0}$ le tas binomial que nous allons fabriquer.

Si t et t' sont deux arbres binomiaux de même ordre n, notons $t \oplus t'$ l'arbre binomial d'ordre $n+1$ obtenu en recollant t et t'. Notons également $\emptyset$ l'arbre vide.

On fabrique par récurrence sur k les c_k ainsi qu'une suite de retenues $(r_k)_{k\geq 0}$ où r_k est un arbre binomial d'ordre k, ou l'arbre vide.

Posons $r_0 = \emptyset$. Soit maintenant $k \in \mathbb{N}$. On dispose de a_k, b_k et r_k qui sont des arbres binomiaux d'ordre k, ou l'arbre vide.

- Si ces trois arbres sont vides, on pose $c_k = \emptyset$ et $r_{k+1} = \emptyset$.
- Si deux de ces arbres sont vides, on pose $c_k =$ celui des trois qui est non vide et $r_{k+1} = \emptyset$.
- Si un seul de ces arbres est vide, on pose $c_k = \emptyset$ et $r_{k+1} =$ le recollement des deux arbres non vides, qui est bien un arbre binomial d'ordre $k+1$.
- Si aucun de ces arbres n'est vide, on pose $c_k =$ l'un des trois arbres et $r_{k+1} =$ le recollement des deux autres arbres.

Résumons le tout dans un tableau. Il y a en tout 8 cas. Lorsqu'une lettre apparaît dans une case, l'arbre correspondant est supposé non vide.

Remarquons que dans le cas numéro 8 il y a une certaine dose d'arbitraire dans les choix de recollements.

	a_k	b_k	r_k	$\Rightarrow$	c_k	r_{k+1}
1	$\emptyset$	$\emptyset$	$\emptyset$		$\emptyset$	$\emptyset$
2	$\emptyset$	b_k	$\emptyset$		b_k	$\emptyset$
3	a_k	$\emptyset$	$\emptyset$		a_k	$\emptyset$
4	a_k	b_k	$\emptyset$		$\emptyset$	$a_k \oplus b_k$
5	$\emptyset$	$\emptyset$	r_k		r_k	$\emptyset$
6	a_k	$\emptyset$	r_k		$\emptyset$	$r_k \oplus a_k$
7	$\emptyset$	b_k	r_k		$\emptyset$	$r_k \oplus b_k$
8	a_k	b_k	r_k		r_k	$a_k \oplus b_k$

```
In [48]: def fusionner(a, b):
             n = max(len(a), len(b))
             c = n * [None]
             r = None
             for k in range(n):
                 ak = element_tas(a, k)
                 bk = element_tas(b, k)
                 w = (ak == None, bk == None, r == None)
                 if   w == (True,  True,  True):  c[k], r = None, None
                 elif w == (True,  False, True):  c[k], r = bk, None
                 elif w == (False, True,  True):  c[k], r = ak, None
                 elif w == (False, False, True):  c[k], r = None, coller_arbres(ak, bk)
                 elif w == (True,  True,  False): c[k], r = r, None
                 elif w == (False, True,  False): c[k], r = None, coller_arbres(r, ak)
                 elif w == (True,  False, False): c[k], r = None, coller_arbres(r, bk)
                 else: c[k], r = r, coller_arbres(ak, bk)
             if r != None: c.append(r)
             return c
```

```
In [49]: a = random_tas(list(range(100)))
         b = random_tas(list(range(100, 200)))
         c = fusionner(a, b)
         print_tas(c)
```

```
0: _
1: _
2: _
3: 8(39,19(21),105(145,157(188)))
4: _
5: _
6: 0(16,13(52),10(97,82(88)),2(3,29(68),31(56,95(99))),6(86,26(73),25(33,43(54)),27(61,83(96),28
7: 1(42,63(77),32(47,67(72)),5(22,7(75),30(48,34(44))),18(74,40(41),35(65,62(70)),23(53,66(87),3
```

```
In [50]: a = random_tas(list(range(10000)))
         b = random_tas(list(range(100000, 200000)))
         c = fusionner(a, b)
         print(est_tas_binomial(c))
```

```
True
```

Proposition : La complexité de `fusionner` est $O(\max(\lg|a|, \lg|b|)$.

Démonstration : La fonction effectue une simple boucle `for`. Chaque itération s'exécute en $O(1)$, et le nombre d'itérations est égal à la plus grande des longueurs des deux tas.

1.4.3 4.3 Insertion d'un élément (bis)

Remarquons que nous disposons, grâce à la fonction d efusion, d'une nouvelle façon d'insérer un élément dans un tas h : il suffit de fusionner le tas h et le tas qui contient x comme seul élément !

```
In [51]: def inserer_par_fusion(x, h):
             a = [(x, [])]
             return fusionner(a, h)
```

Remarque : De la façon dont nous avons écrit `fusionner`, cette fonction est moins efficace que la fonction `inserer` que nous avions écrite dans la section 3.

1.4.4 4.4 Suppression de l'élément minimal

Proposition : Soit t un arbre binomial. La liste des fils de t est un tas binomial.

Démonstration : Laissée au lecteur. Elle se fait par récurrence sur l'ordre de l'arbre binomial.

La fonction `supprimer_min` est alors évidente. Pour supprimer l'élément minimal du tas (non vide) h :

- On recherche l'indice j dans le tas de l'arbre dont la racine est le plus petit élément du tas h. Soit h' le tas formé par la liste des fils de $h[j]$.
- On met $h[j]$ à `None`.
- On fusionne h et h'.

Petit détail pour terminer : il se pourrait que le dernier arbre du tas obtenu soit vide. Dans ce cas on le supprime.

```
In [52]: def indice_min(h):
             n = len(h)
             if n == 0: raise Exception('Tas vide')
             j = 0
             while j < n and h[j] == None: j = j + 1
             for k in range(j, n):
                 if h[k] != None and racine(h[k]) < racine(h[j]): j = k
             return j
```

```
In [53]: def supprimer_min(h):
             j = indice_min(h)
             h1 = fils(h[j])
             h[j] = None
             h2 = fusionner(h1, h)
             if h2 != [] and h2[-1] == None: h2.pop()
             return h2

In [54]: h = random_tas(list(range(16)))
         print_tas(h)

0: _
1: _
2: _
3: _
4: 0(3,2(4),6(13,8(12)),1(9,5(11),7(15,10(14))))



In [55]: h = supprimer_min(h)
         print_tas(h)

0: 3
1: 2(4)
2: 6(13,8(12))
3: 1(9,5(11),7(15,10(14)))
```

Proposition : La complexité de `supprimer_min` est $O(\lg |h|)$.

Démonstration : Soit $n = \lg |h|$. La recherche de j s'effectue en temps $O(n)$. La fusion de h_1 et h également.

1.4.5 4.5 Pour résumer

Pour résumer ... voici un tableau résumant les complexités des fonctions que nous avons étudiées.

Fonction	*Paramtre(s)*	*Complexit*	*Complexit amortie*
est_vide	h	$O(1)$	
minimum	h	$O(\lg \|h\|)$	
inserer	h	$O(\lg \|h\|)$	$O(1)$
supprimer_min	h	$O(\lg \|h\|)$	
fusionner	h_1, h_2	$O(\max(\lg \|h_1\|, \lg \|h_2\|))$	

```
In [ ]:
```