

# ALGOA

## 一些基本操作

### 移位操作：

左移位  $\ll$ : 相当于乘  $2^n$ ,  $n$  为移动位数, 对2进制操作  $\gg$ : 相当于除以  $2^n$

左移位可以用于快速计算2的幂次方, 右移位可用于提取二进制数中的特定位置或执行除法等操作。需要注意的是, 在右移位运算中, 如果原数为负数, 可能会使用符号扩展来填充高位, 这意味着最高位会继续保持原来的值 (0或1), 以保持负数的符号。

$|$ : 按位或运算符

将两个数的二进制进行按位运算, 用来  $**==$  设置某一位上的值  $==**$

```
x = 5 # 二进制表示为 101
x = x | 2 # 将第二位设置为1, 结果为 7 (111)
```

$\&$ : 按位与运算符

进行与运算, 用来  $==$  检查某一位的值  $==$

```
y = 6 # 二进制表示为 110
result = y & 2 # 检查第二位, 结果为 2 (010)
```

移位运算与位运算的结合：

你可以使用左移位操作符  $\ll$  来设置特定位置上的值, 然后再使用按位或运算符  $|$  将它们合并。

```
value = 0 # 初始化为0
value = value | (1 << 2) # 将第二位设置为1, 结果为 4 (100)
```

$(1 \ll n)$  可以将第  $n$  位设置为1, 然后再进行检查或者设置对方的值

### 位掩码 (Bitmask)

在计算机编程中, 是一个用于操作位 (二进制位) 的二进制数字, 它通常用于执行位级别的操作, 如设置、清除或测试特定位置。

掩码通常是一个二进制数, 其中某些位被设置为1, 而其他位设置为0。这些位掩码用于与其他二进制数执行按位操作, 以达到特定的目的。掩码的作用是控制哪些位会受到操作的影响, 哪些位会被保留不变。

设置位: 通过与一个掩码执行按位或操作, 可以将特定位置设置为1, 而其他位不受影响。

```
value = value | mask
```

清除位：通过与一个取反掩码（所有位为1的掩码取反，即0的位为1，1的位为0）执行按位与操作，可以将特定位设置为0，而其他位不受影响。

```
value = value & ~mask
```

测试位：通过与一个掩码执行按位与操作，可以检查特定位是否为1。

```
is_set = (value & mask) != 0
```

补码：

补码（Two's complement）是一种用于在计算机中表示有符号整数的二进制数编码方式。在补码表示中，正数和负数都可以使用相同的位操作来进行加减运算，因此它在计算机中的应用非常广泛。

\$\$补码 = 反码 + 1\$\$

以下是补码的基本特点：

- 正数表示：**正数的补码表示与二进制表示相同。例如，十进制的数字5在补码中表示为 `00000101`，与其二进制表示相同。
- 负数表示：**负数的补码表示是通过将其绝对值的二进制表示取反（按位取反），然后加1来获得的。例如，十进制的-5在补码中表示为 `11111011`。这是因为5的二进制表示是 `00000101`，取反后为 `11111010`，再加1得到 `11111011`。
- 加法运算：**在补码中，正数和负数的加法运算可以直接使用位操作（加法）来进行。这使得计算机可以使用相同的硬件来执行加法操作，而不需要区分正数和负数。
- 表示范围：**补码表示可以表示的整数范围是由位数决定的。例如，8位的补码可以表示从-128到127的整数范围。
- 负数的表示和检测：**在补码中，最高位表示符号，1表示负数，0表示正数。这使得检测一个数是正数还是负数变得很容易。

补码表示在计算机硬件中非常常见，因为它简化了整数运算的实现。补码的优势在于它允许将加法和减法都表示为相同的位操作，这对于处理数学运算非常有用，同时也避免了溢出的问题。

```
~ (1 << 2)
# -5
```

二进制位数：

在计算机编程中，常见的二进制表示位数是8位、16位、32位和64位。以下是它们的一些常见应用：

### 1. 8位二进制：

- 通常称为字节 ( byte ) 。
- 常用于字符编码 ( 如ASCII ) 、小整数、颜色表示 ( 每个分量8位 ) 、以及I/O端口的控制等。
- 范围为0到255。

### 2. 16位二进制：

- 常用于表示大整数、Unicode字符编码 ( UTF-16 ) 、音频采样值等。
- 范围为0到65535。

### 3. 32位二进制：

- 常用于整数、浮点数表示、IPv4地址、指针、图形处理等。
- 范围为0到约42亿 ( 正负数 ) 。

### 4. 64位二进制：

- 常用于大整数、浮点数表示、64位操作系统、内存地址、高精度计算等。
- 范围为0到约 $10^{19}$ 。

一个十六进制位 ( 0-F ) 等于4个二进制位，即4比特。一个十六进制数字可以精确地表示4个二进制位的值，例如，十六进制数字A表示二进制的1010，十六进制数字C表示二进制的1100。十六进制在编程中经常用于表示二进制数据，如内存地址、颜色值、ASCII字符等。这是因为它比二进制更易读，同时也更紧凑。在Python和其他编程语言中，你可以使用前缀"0x"来表示十六进制数，例如0x1A表示十进制的26。要在Python中转换二进制和十六进制，你可以使用内置函数bin()和hex()。

bin() 函数：

bin() 函数用于将整数转换为二进制表示。语法：bin(x)，其中 x 是要转换的整数。返回一个字符串，表示 x 的二进制表示，以 "0b" 作为前缀。例如，bin(10) 返回字符串 '0b1010'。

```
binary_representation = bin(42)
print(binary_representation) # 输出 '0b101010'
```

hex() 函数：

hex() 函数用于将整数转换为十六进制表示。语法：hex(x)，其中 x 是要转换的整数。返回一个字符串，表示 x 的十六进制表示，以 "0x" 作为前缀。例如，hex(255) 返回字符串 '0xff'。

```
hexadecimal_representation = hex(255)
print(hexadecimal_representation) # 输出 '0xff'
```

相互转换

```
binary_string = '0b1010'
decimal_number = int(binary_string, 2) # 将二进制字符串转换为整数
print(decimal_number) # 输出 10
```

```
hexadecimal_string = '0xff'
decimal_number = int(hexadecimal_string, 16) # 将十六进制字符串转换为整数
print(decimal_number) # 输出 255
```

十六进制 (Hexadecimal，通常简称为Hex) 是一种基数为16的数制系统，用于表示数字和数据。它包含十个数字 (0-9) 和六个字母 (A-F)，总共16个字符。在十六进制中，每个字符代表四个二进制位 (4比特)，因此它是二进制的紧凑表示。

以下是十六进制数字的表示：

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (与十进制相同)
- A, B, C, D, E, F (分别对应十进制的10, 11, 12, 13, 14, 15)

在十六进制中，每个字符的权重 (位值) 是16的幂次方，从右到左依次为1, 16, 256, 4096，依此类推。十六进制通常用于计算机领域，因为它提供了更紧凑的表示，更容易转换为二进制。

以下是一些十六进制数字的示例：

- 0x0：表示十进制的0，二进制的0
- 0x1：表示十进制的1，二进制的1
- 0xA：表示十进制的10，二进制的0b1010
- 0xF：表示十进制的15，二进制的0b1111
- 0x10：表示十进制的16，二进制的0b10000
- 0xFF：表示十进制的255，二进制的0b11111111
- 0x100：表示十进制的256。

### 逻辑运算

^ 是异或运算符，也被称为按位异或运算符。它用于执行位级别的逻辑运算，针对两个二进制数的每个对应位进行运算。异或运算的规则如下：

- 如果两个对应位的值相同，结果为 0。
- 如果两个对应位的值不同，结果为 1。

异或运算的真值表如下：

输入 A	输入 B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

异或运算常用于位操作和编程中的多种情况，例如：

1. 交换两个变量的值：通过异或运算，你可以在不使用临时变量的情况下交换两个变量的值。

```

a = a ^ b
b = a ^ b # 先把不同的变成相同的，相同的为不同
a = a ^ b # 再把不同的还回相同，相同的变为不同，就回
          # 到原先的a了。

```

## 2. 数据加密：异或运算用于加密和解密数据。

在位操作中，异或运算是一个非常有用的工具，它允许你在位级别执行复杂的操作，同时也经常用于编程中的一些巧妙技巧。

## Invariant de boucle:

Invariants de boucles (Loop Invariants) :

initial: 循环开始前该变量为真 itération : 循环过程不断迭代，变量值更改，但是性质保持不变 fini : 循环终止，该变量能够判断结果是否为真

Il s'agit de déterminer quelle expression est utilisée pour définir le début, la progression et la fin de la boucle.

循环不变式：在循环过程中，迭代前这个式子保持为真，下一次迭代前仍然为真，终止时便可以说明结果为真。

比如：插入排序中，子数列 $A[1, \dots, n-1]$ 已经排序好，对于初始 $A[0]$ 为真，排序好；迭代中，下一次前为真；终止时 $j=n+1$ ，然后 $A[1, \dots, n]$ 为 $(A[j-1])$ ，已经排序好，证明算法正确

这些循环不变式帮助确保循环的正确性，因为它们描述了每次迭代后程序应该满足的条件。在循环结束时，这些条件应该保持为真。这对于验证算法的正确性非常有帮助。不过，请注意，为了确保这些循环不变式的正确性，还需要进行适当的数学证明。

precondition: assumer au début quelle condition est vraie

postcondition: garantir à la fin la correction du résultat

def sort (A, n): for i = 1 to n - 1: insert ( i , A) # **assumer x dans [1, n-1]** def insert (x, A): key = A[x] j = x - 1 while j ≥ 0 and A[j] > key: [j + 1] = A[j] j = j - 1 A[j+1] = key

## indécidable

PCP问题：不可判定(indécidable)

停机问题(**Problème de l'arrêt**)

```

bool halts(char *f_code, char *t);

void modified_halts(char *f_code) {
    if (halts(f_code, f_code)) { // 当halts(f_code, f_code)返回true

```

```

    while (true) { /*empty*/ } // 死循环
}
else { // 当halts(f_code, f_code)返回false
    return; // 立即停止运行
}
}

```

其中f\_code是我们要进行测试的函数f的ASCII源代码, 我们可以认为对f\_code进行编译得到了函数f.

当f对t停机时, halts(f\_code, t)返回true; 当f对t不停机, halts(f\_code, t)返回false. 反过来我们可以说, halts(f\_code, t)返回true, f对t停机; halts(f\_code, t)返回false, f对t不停机。

假设 modified\_halts 这个函数的ASCII源代码是 modified\_halts\_code``, 如果我们把modified\_halts\_code 作为modified\_halts`的输入会是什么情况?

如果 modified\_halts 对 modified\_halts\_code 停机, 说明 halts(modified\_halts\_code, modified\_halts\_code) 返回false(针对 modified\_halts 函数而言), 然而 halts(f\_code, f\_code) 为false 说明 modified\_halts 对 modified\_halts\_code 不停机;

如果 modified\_halts 对 modified\_halts\_code 不停机, 说明 halts(modified\_halts\_code, modified\_halts\_code) 返回true, 然而 halts(f\_code, f\_code) 为true 说明 modified\_halts 对 modified\_halts\_code 停机.

综合以上两种情况, "modified\_halts 对 modified\_halts\_code 停机"当且仅当"modified\_halts 对 modified\_halts\_code 不停机", 这是一个矛盾, 说明不存在这样一个halts函数可用于判断任意函数的可停机性.

Le Théorème de Rice dit (informellement) que toute propriété non-triviale(非平凡) sur les programmes est indécidable : "任何非平凡 ( non-trivial ) 的关于程序的性质都是不可判定的 ( undecidable ) 。”

"性质" 指的是一个关于程序行为的特征或特点, 如程序是否会停止 ( 停机问题 )、程序是否会输出某个特定值、程序是否满足某种规范等。

"非平凡" 表示性质不是对所有程序都成立或都不成立的情况。也就是说, 这个性质对于某些程序是真实的, 而对于其他程序是假的。非平凡性质是有趣的性质, 因为它们涉及到程序的不同行为。

"不可判定" 意味着无法编写一个通用算法来判断任何给定程序是否满足这个性质。换句话说, 对于非平凡性质, 没有一种方法可以对所有情况进行判定, 存在一些情况下无法得出结论。

这个定理的要点是, 对于许多有趣的、非平凡的程序性质, 我们不能期望编写一个计算机程序来确定所有情况下是否成立。这些问题被称为不可判定的问题, 它们在计算理论和计算机科学中起着重要的作用, 强调了计算机的局限性和复杂性。

Par exemple, le 10e problème de Hilbert: "Décider si une équation diophantienne (quelconque) a une solution"; Problème de correspondance de Post (PCP)

## La machine de Turing

Une machine déterministe à deux compteurs (M2C):

L'état initial est 1.

Tous les valeurs des compteurs sont initialement 0.

- qui contient des symboles issus d'un alphabet fini  $A$  ;
- $A$  contient un symbole spécial  $\square$  indiquant qu'une case est vide ;
- une tête de lecture  $cur \in \mathbb{Z}$  désigne la case courante  $M[cur] \in A$  de la mémoire ;
- $M$  contient des données d'entrée : un mot sur  $A \setminus \{\square\}$  commençant à la case 0.

- un état correspond a une instruction spéciale accept qui arrête la machine et répond « oui » ;
- un état correspond a une instruction spéciale reject qui arrête la machine et répond « non » ;
- la machine peut bloquer (deadlock) si la case courante ne correspond pas à l'instruction courante.




La Complexité d'une machine de Turing:

1. Complexité temporelle : nombre d'instructions réalisées avant arrêt ;
2. Complexité spatiale : nombre maximal de cases non vides simultanément(同时地).

## Complexité

排序算法复杂度分析：

排序算法	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
希尔排序	$O(n \log n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
基数排序	$O(N \cdot M)$	$O(N \cdot M)$	$O(N \cdot M)$	$O(M)$	稳定

冒泡、选择、直接排序需要两个for循环，每次只关注一个元素，平均时间复杂度为 $O(n^2)$ （一遍找元素 $O(n)$ ，一遍找位置 $O(n)$ ）

快速、归并、希尔、堆基于二分思想，log以2为底，平均时间复杂度为 $O(n \log n)$ （一遍找元素 $O(n)$ ，一遍找位置 $O(\log n)$ ）

稳定性记忆-"快希选堆"（快牺牲稳定性）

排序算法的稳定性：排序前后相同元素的相对位置不变，则称排序算法是稳定的；否则排序算法是不稳定的。

## Quelques classes de complexité des problèmes de décision

**PTIME (P)** :可以在多项式时间内解决的问题。这意味着问题的解决方案可以在输入规模 $n$ 的多项式时间内找到，通常表示为 $O(n^k)$ 。例如，线性规划中判断是否存在一个解就属于P类问题。


**PSPACE** :可以在多项式空间内解决的问题。这指的是问题的解决方案可以在多项式空间（相对于输入规模）内找到。例如，在类似Rush Hour或Sokoban的问题中，确定是否可以从一个状态过渡到另一个状态就属于PSPACE问题。

**EXPTIME** :可以在指数时间内解决的问题。这意味着问题的解决方案的运行时间以 $2^n$ 的指数形式增长，其中 $n$ 是输入规模。例如，在围棋中确定在给定的棋盘大小上第一个玩家是否能赢，就属于EXPTIME问题。

**EXPSPACE** :可以在指数空间内解决的问题。这指的是问题的解决方案需要指数级的存储空间。例如，在比较两个简单正则表达式（如 $(a \cdot b^* + c)^*$ 和 $(a + b + c)^*$ ）是否具有相同的语言时，这就是一个EXPSPACE问题。



## problème de NP, SAT, ILP

**SAT**是指给定一个逻辑表达式，我们可以让其值为true。合取范式：由交连接的子句。

**k-SAT**:  $k$ 指的是每个子句中包含 $k$ 个文字 SAT属于NP-complete问题

## Complexité moyenne : variables aléatoires indicatrices



$$X = \sum_{i=1}^n X_i$$

## La Complexité de Amorti

## Classes de complexité des algorithmes probabilistes



## Algorithmes Probalistes

Ce qui utilise une source de hasard. Plus précisément le déroulement de l'algorithme fait appel à des données tirées au hasard.

Un algorithme est dit probabiliste si son comportement dépend à la fois des données du problème et de valeurs produites par un générateur de nombres aléatoires.

Par exemple, Monte Carlo, Las Vegas et Atlantic City

- Monte Carlo

```
import random

def monte_carlo_pi(num_samples):
    inside_circle = 0
```



```

for _ in range(num_samples):
    x = random.uniform(0, 1)
    y = random.uniform(0, 1)
    if x**2 + y**2 <= 1:
        inside_circle += 1

return 4 * inside_circle / num_samples
# 为了求pi，这里采用的是单位圆面积与正方形比值方式

num_samples = 1000000
estimated_pi = monte_carlo_pi(num_samples)
print(f"Estimated Pi: {estimated_pi}")

```

- Las Vegas Donne toujours un résultat exact, mais le temps de calcul est petit avec une très forte probabilité.

```

# 随机排序一个列表
import random

def las_vegas_shuffle(arr):
    while True:
        shuffled = random.sample(arr, len(arr))
        if shuffled != arr:
            return shuffled

my_list = [1, 2, 3, 4, 5]
shuffled_list = las_vegas_shuffle(my_list)
print("Shuffled List:", shuffled_list)

```

- Atlantic City

Donne une réponse avec une très forte probabilité sur l'exactitude de la réponse pour un temps de calcul faible en moyenne probabiliste.

## Structure de Données

### 存储

大端序存储方式 ( Big-Endian ) 是一种将多字节数据的最高有效字节 ( MSB ) 存储在内存地址最低的存储方式，也就是字节序的高位字节在前，低位字节在后。这与我们的阅读习惯相似，比如一个十进制数1234，我们从左到右阅读，高位在前，低位在后。

例如，一个32位的整数0x12345678，在大端序存储方式下，会被存储在内存中如下：

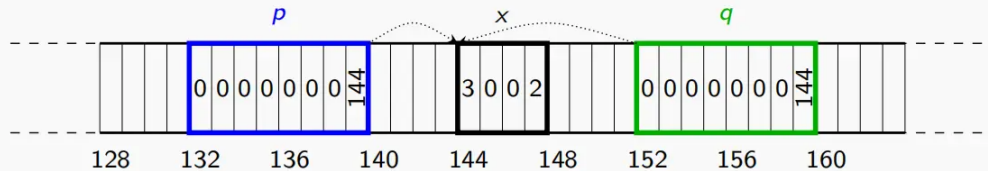
内存地址：0x00 0x01 0x02 0x03 存储内容：0x12 0x34 0x56 0x78

其中，0x12是最高有效字节，被存储在内存地址最低的位置 ( 0x00 )，而0x78是最低有效字节，被存储在内存地址最高的位置 ( 0x03 )。

```

x = 2;
&x = adresse;
p -> x;
p = adresse;
*p est une variable dont l'adresse est dans p

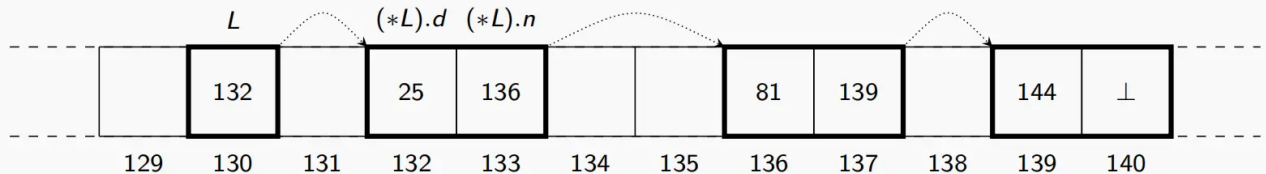
```



- En réalité, les variables (et notamment les pointeurs) ont souvent une taille  $> 1$ .
- *x* est un entier sur 32 bits qui contient la valeur  $2 + 0 \times 256 + 0 \times 256^2 + 3 \times 256^3 = 16\,777\,218$  (*big endian*);
- *p* est un pointeur sur un entier sur 32 bits :
  - Sa taille est 64 bits
  - Il contient l'adresse (du début) de *x* : 144
  - \**p* vaut 16 777 218
- *q* est un pointeur sur un entier sur 8 bits :
  - Sa taille est 64 bits
  - Il contient l'adresse (du début) de *x* : 144
  - \**q* vaut 3

## Liste Chaînée

- Un alternative est la **liste chaînée** :



Chaque cellule *C* contient la donnée *C.d* et un pointeur vers la cellule suivante *C.n*

accès:  $O(n)$  insertion/suppression:  $O(n)$  insertion/suppression avec pointeur:  $O(1)$

Le début de la liste est un pointeur vers la première cellule

```

cur = self._head
while cur.n is not None:
    cur = cur.n
cur.n = new_node

```

*cur* est un pointeur vers la tête de la liste, et *self.\_head* est aussi un pointeur vers la tête de la liste. Lorsque vous exécutez *cur = self.\_head*, vous placez *cur* à la même adresse que *self.\_head*, c'est-à-dire la tête de la liste. **Les adresses sont les mêmes !**

Ensuite, vous utilisez une boucle *while* pour parcourir la liste. À chaque itération, vous déplacez *cur* vers la cellule suivante (*cur = cur.n*). Lorsque la boucle se termine, *cur* pointe vers la dernière cellule de la liste, tandis que *self.\_head* continue de pointer vers la tête de la liste. Ici, le pointeur *cur* se déplace, mais les adresses des cellules de la liste restent les mêmes ; aucune nouvelle cellule n'est créée.

当你执行 `cur.n = new_node` 时，你实际上是将链表的最后一个节点的 `n` 属性（指向下一个节点的指针）设置为新节点 `new_node`。这会使链表的最后一个节点指向新节点，因此新节点成为了链表的最后一个节点。

但是，`self.head` 仍然指向链表的头节点，它没有发生变化。所以，`self.head` 仍然指向链表的头部，而 `cur` 指向链表的最后一个节点。

在 Python 中，链表通常由头指针（`self.head`）来引领整个链表的访问，而 `cur` 只是一个临时指针，用于遍历链表，不会影响头指针的位置。

```
class Node:
    def __init__(self, value):
        self.d = value
        self.n = None

class SimpleList:
    def __init__(self):
        self._head = None
    # creer une liste self._head
    def insert_front(self, value):
        NewNode = Node(value)
        NewNode.n = self._head
        self._head = NewNode

    def append(self, value):
        new_node = Node(value)
        if self._head is None:
            self._head = new_node          # pointer vers une nouvelle cellule
            return
        cur = self._head
        while cur.n is not None:
            cur = cur.n
        cur.n = new_node
        return

    def travel(self) -> list:
        liste = []
        cur = self._head
        while cur is not None:
            liste.append(cur.d)
            cur = cur.n
        return liste

    # ajouter au debut ce qui est l'insertion de l'element x apr`es la cellule
    # point`ee par L
    # consigner un emplacement
    # L est une liste ce qui est head._head non head
    def insert(self, pos, x):
        cur = self._head
        # deplacer jusqu'a pos
        for _ in range(pos-1):
            cur = cur.n
        # créer une nouvelle cellule
```

```

L_new = Node(x)
L_new.n = cur.n
cur.n = L_new

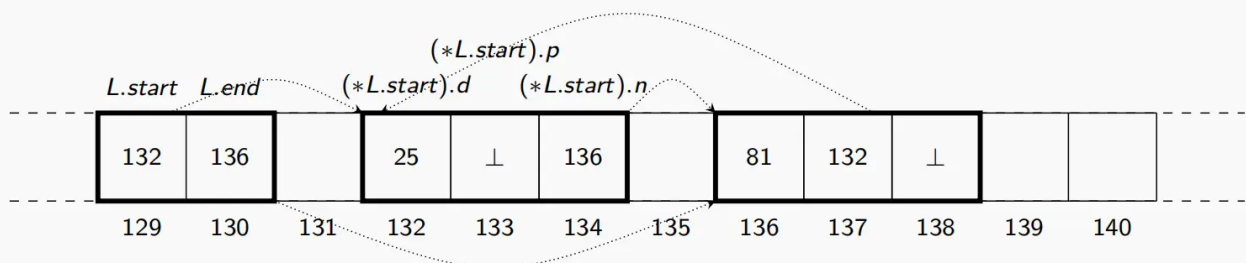
def delete(self, x):
    cur = self._head
    # deplacer
    while cur.n.d != x:
        cur = cur.n
    # delete
    cur.n = cur.n.n

def search(self, item) -> bool:
    cur = self._head
    while cur is not None:
        if cur.d == item:
            return True
        cur = cur.n
    return False

def print_(L):
    if L is not None:
        print(L.d)
        print_(L.n)

```

## Listes doublement chaînée



accès	$O(n)$
insertion/suppression	$O(n)$
insertion/suppression avec pointeur (dont début et fin)	$O(1)$

```

# Créer une liste doublement chaînée
class Double_List:
    def __init__(self, item):
        self.d = item
        self.prev = None
        self.n = None

    def is_empty():
        return self.d == None or self.n == None

    def insert_after(x, L):

```

```

    L_x = Double_List(x)
    if L.n == None:
        L.n = L_x.prev
        L_x.n = None
        return
    insert_after(x, L.n)

def insert_before(x, L):
    L_x = Double_List(x)
    L_x.n = L.n
    L = L_x

def delete_end(fin, L):

```

### =="Accéder à la fin d'une liste simplement chaînée en $O(1)$ : ajouter juste un pointeur"==

在普通的单链表中，要访问链表的末尾，你需要从链表的头部开始遍历链表，直到达到最后一个节点。这样的操作的时间复杂度是线性的，即 $O(n)$ ，其中 $n$ 是链表的长度。

然而，如果你在链表中添加一个额外的指针，指向链表的末尾节点，那么你可以直接通过这个指针访问链表的末尾，而不需要遍历整个链表。这样，你可以在常数时间复杂度内访问链表的末尾，因为你可以直接跳到末尾节点，而不需要遍历。

这个方法的实现非常简单：每当你在链表中添加一个新节点时，同时更新这个指向链表末尾的指针，以保持它指向新的末尾节点。这样，你就可以随时以 $O(1)$ 的时间复杂度访问链表的末尾。

这种技巧对于需要频繁访问链表末尾的应用程序非常有用，因为它可以显著提高访问效率。

## Files et piles

File: First in First Out(FIFO)

Pile: Last in First Out(LIFO)

Opération:

• insérer / empiler (enqueue / push) • supprimer / dépiler (dequeue / pop) • prochain élément / dessus de pile (next / top) • vide

不理解

- On peut utiliser une **pile** pour simuler celle des appels récursifs ;

### Exemple

```

1 def fact2(n):
2     if n == 0:
3         r = 1
4     else:
5         r = n * fact2(n - 1)
6
7     return r

1 def fact3(n):
2     # frame est un type enregistrement avec les champs
3     # n : entier, r : entier, p : pointeur sur un entier
4     # a : entier, grow : booléen
5     r = 0
6     W.push(frame(n, 0, &r, 1, True)) # W est une pile
7     while not W.empty():
8         f = W.top()
9
10        if f.grow: # la pile croît
11            if f.n == 0:
12                *f.p = 1
13                W.pop()
14            else:
15                W.top().grow = False
16                W.push(frame(f.n - 1, 0, &f.r, f.n, True))
17        else: # la pile décroît
18            *f.p = f.r * f.a
19            W.pop()
20
21    return r

```

## Files de priorité

- Toujours sortir le plus petit élément de la file ex. plus courts chemins, arbres couvrants, codage de Huffman, qualité de service, etc.
- On peut trier la file (typiquement)  $O(n \log_2 n)$  mais il faut recommencer à chaque insertion ( $O(n)$ ) ;
- Pour faire mieux : passer de la liste/tableau à l'arbre.

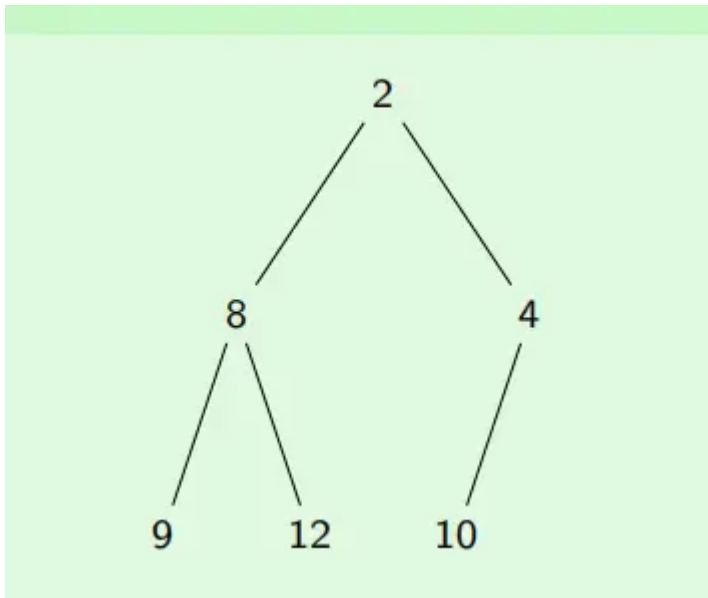
## L'arbre

- La hauteur de l'arbre est la longueur du plus long chemin de la racine à une feuille (ici 4).
- Un arbre est équilibré en hauteur si les hauteurs de ses sous-arbre diffèrent d'au plus 1 ;
- Un arbre n-aire équilibré avec m nœuds a une hauteur d'au plus  $\log_n m$  ;
- On construit une structure avec des opérations en  $O(\text{hauteur})$ .

## Tas binaire

Un tas binaire (binary heap) est un arbre binaire tel que :

1. seul le dernier niveau n'est pas rempli donc il est équilibré
2. dans chaque sous-arbre == la racine est plus petite que les racines de ses sous-arbres == donc plus petite que tous les éléments de ces sous-arbres



Pour une file de priorités, il nous faut trois opérations :

1. test du vide (trivial)
2. insertion d'un élément
3. extraction du minimum

#### Tas binaire : insertion

remonter à sa place.

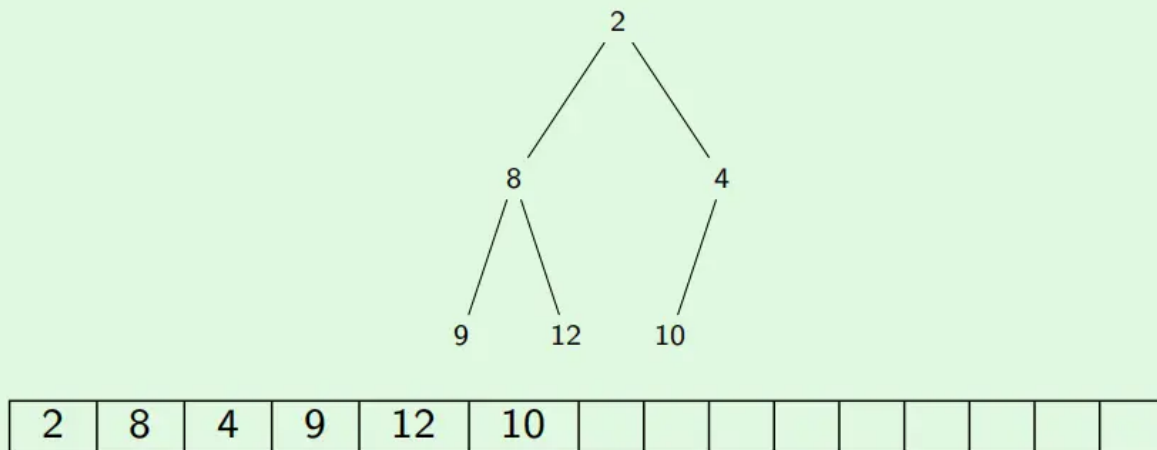
#### Tas binaire : extraction du minimum

1 Le minimum est la racine ; 2 On le remplace par le dernier élément de l'arbre le plus **a droite dans le niveau le plus profond** 3 On fait descendre cet élément à sa place le cas échéant on échange avec le plus petit successeur

#### Tas implicites

Si on a un accès en  $O(1)$  au parent et aux successeurs, insertion et extraction sont en  $O(\log_2 n)$  ;

En fait, le tas implicite est ==une liste==. Mais il **enregistre sa valeur de façon tas binaire**.



Pour un indice  $i$  donnée :

1. indice du nœud parent :  $\frac{i-1}{2}$
2. indice du nœud successeur (fils) gauche :  $2i + 1$  et droit :  $2i + 2$

个数上来看是，第 $j$ 个的沿着方向过去的子节点是第 $2j$ 个

#### Exercice

Écrire l'algorithme d'insertion pour le tas binaire implicite, avec la fonction `heap_fix_up(A, i)` qui fait monter l'élément d'indice  $i$  à sa place dans le tas binaire  $A$ .

```

def heap_insert(A, x, n):
    if n == len(A) - 1:
        raise ValueError('heap is full')
    else:
        A[n] = x    # n est l'indice
        n = n + 1   # nouveau n est le longueur
        heap_fix_up(A, n-1)

# i est l'indice de element
def heap_fix_up(A, i):
    p = (i - 1) // 2
    while p >= 0 and A[p] > A[i]:
        swap(A[p], A[i])
        i = p
        p = (i - 1) // 2
  
```

#### ==Delete==

```

def heap_del(A, x):
    n = len(A)
    index = A.index(x)
    A[index] = A[n-1]
    heap_fix_down(A, index, n)
  
```



```
def heap_fix_down(A, i, n):
    p = 2 * i + 1
    while p <= n:
        # seulement s'il y a un fils
        if A[p] and not A[p+1]:
            if A[i] > A[p]:
                swap(A[i], A[p])
            break
        elif not (A[p] and A[p+1]):
            break
        else:
            if A[i] > A[p] or A[i] > A[p+1]:
                swap(A[i], min(A[p], A[p+1]))
            i = p
            p = 2 * i + 1
```

### Tri par tas (heapsort)

- Par extraction successive du minimum, on peut trier efficacement un tableau de  $n$  éléments :
  - 1 insertion des  $n$  éléments :  $n \times O(\log_2 n)$ ;
  - 2 extraction dans l'ordre des  $n$  éléments :  $n \times O(\log_2 n)$
  - 3 au total  $2n \times O(\log_2 n) = O(n \log_2 n)$
  - 4 mais il faut un tableau supplémentaire pour le tas.

==Lemme== Si  $A$  est un tas binaire implicite à  $n$  éléments alors les feuilles sont au indices  $\lfloor n/2 \rfloor \dots (n-1)$ .

- on construit un tas **pour le maximum** au lieu du minimum • chaque maximum extrait est **remis à la place du dernier élément** du tas qui l'a remplacé. • **le coût est toujours**  $O(n \log_2 n)$ , mieux que merge sort qui nécessite un peu de mémoire supplémentaire et quick sort qui est en  $O(n^2)$ .

```
def heatsort(A):
    for i in range(n // 2 - 1, -1, -1):
        # remettre la max a la place de debut
        heat_fix_down_max(A, i, n)

    for i in range(n-1, -1, -1):
        swap(A, 0, i)
        heat_fix_down_max(A, 0, i+1)
```

- Le tas implicite est très simple à implémenter et possède de bonnes complexités ;

vide	$O(1)$
min	$O(1)$
extract_min	$O(\log_2 n)$
insertion	$O(\log_2 n)$

## Ensembles et tableaux associatifs

### Arbres binaires de recherche (ABR)

Definition:

La racine du fils gauche plus petite que la racine

La racine du fils droite plus grande que la racine

```
class BinaryTreeNode:
    def __init__(self, value=None):
        self.parent = None
        self.left = None
        self.right = None
        self.value = value

def abr_search(A, x):
    if A == None:
        return False
    elif x == A.value:
        return True
    elif x < A.value:
        abr_search(A.left, x)
    elif x > A.value:
        abr_search(A.right, x)

def abr_insert(A, x):
    if A == None:
        new_node = BinaryTreeNode(x)
        return new_node
    else:
        if x < A.value:
            abr_insert(A.left, x)
        elif x > A.value:
            abr_insert(A.right, x)
        return A
```

Pour supprimer le noeud N, il existe deux possibilités:

- Si le noeud N à supprimer a zéro ou un successeur, il juste faut remplacer \*N par \*Y(successeur sole) et supprimer Y.
- Sinon, il faut remplacer par Y plus grand que lui ou le plus grand le plus petit

```
def abr_del(A, x):
    if A is None:
        return A
```

```

if A.valeur == x:
    if A.left is None:
        return A.right
    elif A.right is None:
        return A.left

    A.value = find_min(A.right).value
    A.right = abr_del(A.right, A.value)
elif x < A.value:
    abr_del(A.left, x)
elif x > A.value:
    abr_del(A.right, x)
return A

def find_min(A):
    if A is None:
        return A

    if not A.left:
        return A
    else:
        return find_min(A.left)

# trouver le parent du nœud réalisant le minimum
def find_parent_min(A):
    if A is None:
        return None

    parent = None

    while A.left:
        parent = A
        A = A.left
    return parent

```

## Arbre auto-équilibrant

自平衡二叉搜索树，它是一个二叉树，故满足二叉树的性质，并且它能够在插入或删除时自动保持平衡。

根据满足平衡条件的严格程度可以划分种类：

红黑树(Arbres rouges et noirs), AVL

### AVL

Un arbre AVL(Adelson-Velsky & Landis):

Pour tout noeud x, les **sous-arbre** enraciné en x est **équilibrant(en haut)**.

它是平衡二叉树，同一个节点的两个子树的高度差不超过1。并且它的两个子树也是平衡二叉树。

高度为 $O(\log_2 n)$ .

### AVL rotation

Pour équilibrer l'arbre il faut faire des rotations qui préserve l'invariant de l'ABR.

```
# initial: (A, x, None, 0)
def avl_insert(A, x, p, bfu):
    retrace = (p != None)

    if A is None:
        new_node = AVL_node()
        new_node.value = x
        new_node.b = 0
    else:
        if x < A.value:
            A.left = avl_insert(A.left, x, A, -1)
        else:
            A.right = avl_insert(A.right, x, A, 1)

    B = avl_balance(A)
    if B.b == 0:
        retrace = False

    if retrace:
        p.b = p.b + bfu

    return B

def balance(A):
    R = A
    if A is not None:
        if A.b == 2:
            if A.right.b < 0:
                A.right = avl_rotate_right(A.right)

            R = avl_rotate_left(A)

        elif A.b == -2:
            if A.left.b > 0:
                A.left = avl_rotate_left(A.left)

            R = avl_rotate_right(A)

    return R
```

On peut aussi définir des opérations efficaces  $O(n_1 \log(n_1/n_2 + 1))$  et parallélisables d'union intersection, différence d'ensemble.

\$\$\$recherche:  $O(\log_2 n)$  \ insertion:  $O(\log_2 n)$  \ suppression:  $O(\log_2 n)$  \\$\$\$

### Exercice

En supposant qu'on n'ait besoin que de l'union, l'intersection, la différence d'ensembles et la recherche d'un élément, proposer une structure de données et des algorithmes pour réaliser les opérations ensemblistes en  $O(n_1 + n_2)$  et la recherche en  $O(\log_2 n)$ .

Reponse:

### Tables de hachage

Pour **indiquer l'absence** d'un élément: Utiliser une **valeur spéciale**(Comme **None**)

Il équivaut à ==une liste==.

Insertion, suppression et recherche sont en  $O(1)$

Chaque élément pointe vers l'indice.

### Exercice

**Chemins.** Soit  $M$  une matrice de taille  $m \times n$  contenant des 0 et des 1. On s'intéresse aux chemins qui vont de la case  $(0, 0)$  à la case  $(m - 1, n - 1)$  en ne passant que par des 1 et en ne se déplaçant que sur une case immédiatement à droite ou en bas.

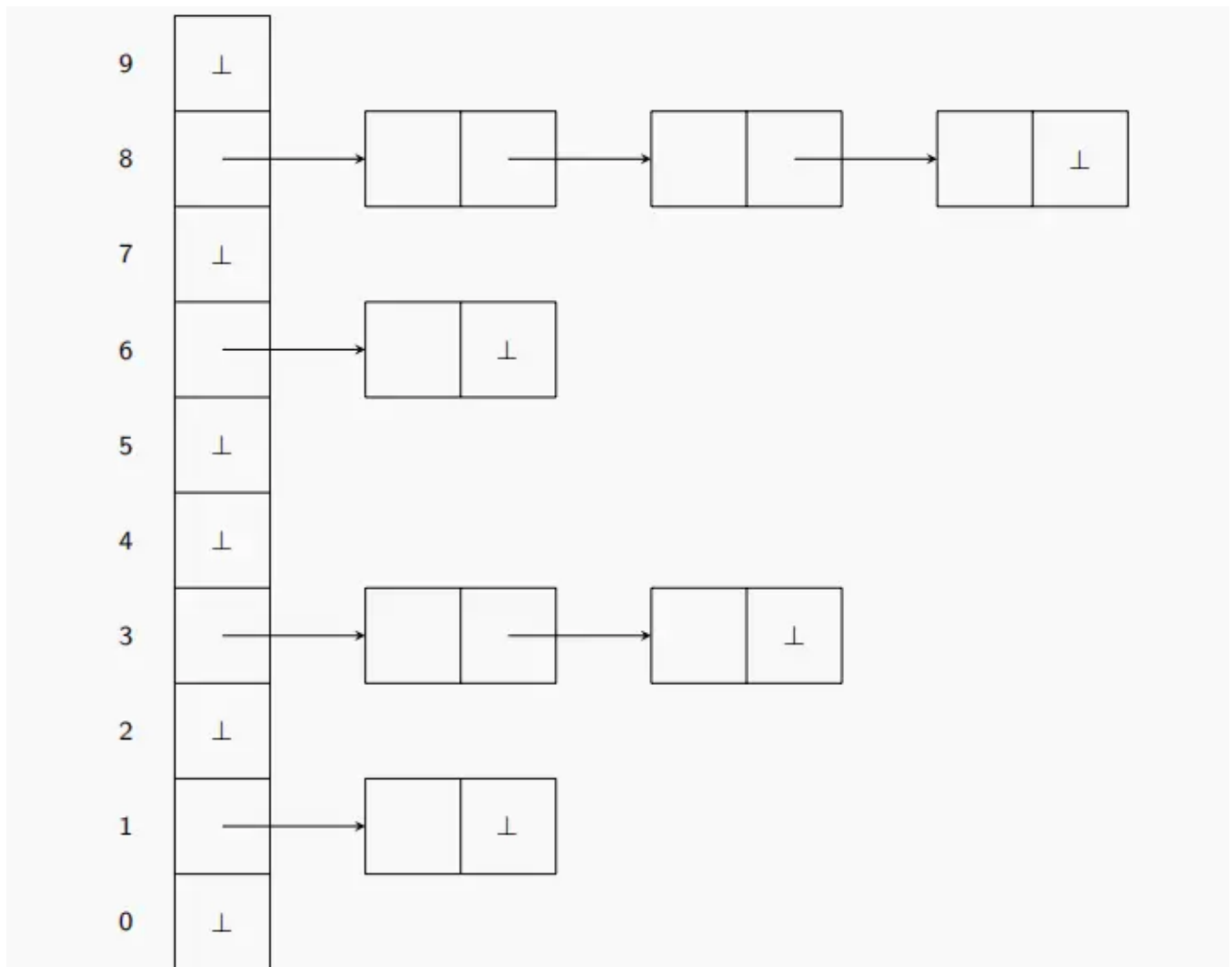
En s'inspirant de la programmation dynamique, donner un algorithme non-récuratif qui calcule le nombre de tels chemins.

Reponse:

### 链地址

哈希表加上链表：

T 表示一个哈希表，这是一个数组，每个元素是一个链表，访问某一个元素所在的位置需要我们用哈希函数获取索引。，这只能找到链表对应头指针的位置。要想找到元素需要在链表中寻找。



L'opération de table de hachage

```
def h(x, T):
    # hash(x), x = key
    return x % len(T)

def ht_search(T, x):
    return list_search(T[h(x, T)], x)
    # h(x): 哈希函数找到x所在的索引i, 然后通过list_search()函数找到T[i]这个链表和x所在的地方

def ht_insert(T, x):
    T[h(x, T)] = list_insert_front(T[h(x, T)], x)

def ht_del(T, x):
    list_del(T[h(x, T)], x)

def list_search(l, x):
    if l is None:
        return False

    if x == l.v:
        return True
```

```

    return list_search(l.n, x)

def list_insert_front(l, x):
    new_node = Node(x)
    if l is None:
        return new_node
    new_node.n = l
    return new_node

```

## Exercice

Quelle est la complexité pire cas de ces opérations ?

$O(n)$ ,  $O(1)$ ,  $O(n)$

Adresse ouverte

La complexité de Hachage

Pire cas:

complexité moyenne:

### Exercice

Une fonction de hachage à valeurs dans  $[0..m - 1]$  est **universelle** si  $P(h(x) = h(y)) = \frac{1}{m}$ .

- ❶ Montrer que si  $h$  est indépendante et uniforme alors elle est universelle.
- ❷ En supposant  $h$  universelle et qu'on a  $n$  éléments dans notre ensemble, quelle est l'espérance du nombre de collisions ?
- ❸ Combien de personnes ont leur anniversaire le même jour dans la classe ?

## Trier

Bubble

Selection

Quick sort

Merge sort

Counting sort