`<<`: 相当于乘 2^n, n 为移动位数，对 2 进制操作 `>>`: 相当于除以 2^n, `|`: 按位或运算符, `&`: 按位与运算符

Variant de boucle: Propriétés inductives –while: après la k e itération, j vaut x −1 −k et 0 ≤j < x, j = j – 1, donc j ≤n −2 –for : après la k e itération, i vaut k + 1 et i < n

terminaison : Dans la boucle **while** d'insert, la valeur de j est dans N (par l'invariant qu'on a prouvé) et décroit strictement à chaque itération. **Récursif :** Par induction/récurrence sur len(A) - i : la fonction termine et renvoit le maximum du tableau à partir de l'indice.

On trouve un invariant qui doit être satisfait par les itérations de tout hypothétique algorithme ;mais qui n'est pas satisfait par la configuration finale recherchée. Complexité asymptotique: $g(n) = \Theta(n)$

meilleur cas pire cas **fib:**

- $T(n+2) = T(n+1) + T(n) + 1 \geq T(n+1) + T(n)$ et $T(0) = 1, T(1) = 2$
- Polynôme caractéristique : $x^2 - x - 1$. Racines $\varphi_1 = \frac{1+\sqrt{5}}{2}$ et $\varphi_2 = \frac{1-\sqrt{5}}{2}$
- $T(n) \geq A\varphi_1^n + B\varphi_2^n$ et avec les conditions initiales $A + B = 1$ et $B - A = \frac{3}{\sqrt{5}}$.
- $T(n) \geq (\frac{1}{2} + \frac{3\sqrt{5}}{10})\varphi_1^n + (\frac{1}{2} - \frac{3\sqrt{5}}{10})\varphi_2^n$ et $T(n) = \Omega((\frac{3}{2})^n)$

Quick: O(n2), O(nlogn), O(nlogn)

Heap : O(nlogn), O(1)

Merge: 同 heap,O(n)

cas moyen (espérance) :

Sans information supplémentaire, on suppose ici que chaque permutation des éléments de A est **équiprobable**. $E(X) = \sum_{i=2}^{n-1} X * P(Ei)$

Nombre d'inversions: $\sum_{i=0}^{i=n-1}\sum_{j=i+1}^{j=n} E(X_i) = \sum_{i=0}^{i=n-1}\sum_{j=i+1}^{j=n} P(X_i) = \sum_{i=0}^{i=n-1}\sum_{j=i+1}^{j=n} \frac{1}{2} = \frac{n(n-1)}{4}$

Démontrer l'invariant suivant de la boucle pour : Après l'itération 0, … , Supposons que l'invariant soit vrai après l'itération k-1, … , lors de l'itération k, …分治算法考虑合起来的时候是否跟回溯相似

Grill : $C[k \,//\, n][k \,\%\, n]$, or 存取点的坐标，比如(i, j) 二分查找不一定是递归，可以循环

```python
def carre(C, T, k):
    # k est le numéro de la case que
    if k == n*n:
        if test_carre(C):
            print_carre(C)
    else:
        # i est le nombre que j'essai
        for i in range(1, n*n+1):
            if not T[i]:
                T[i] = True
                C[k // n][k % n] = i
                carre(C, T, k + 1)
                T[i] = False
```

Backtracking: 如果有 k 就不用弹出，因为是用 k 判断终止而不是 len

```python
def backtracking(state, res, selected, choices, n):
    if len(state) == n and is_solution(state): # 第一个是终止
        record(state, res)
        return
    else:
        for i, choice in enumerate(choices):
            if is_valid(i, choice, selected):
                selected = True
                makechoice(state, choice)
                backtracking(state, res, selected, choices, n)
                selected = False
                undo_choice(state) # pop if 有k就不用，用k判定终止
```

```python
def echiquier_cavalier(n: int):
    # state est le parcours possible
    def is_solution(state):
        return len(state) == n * n

    def record_solution(state, res):
        res.append(state.copy())

    def is_valid(x, y):
        return 0 <= x < n and 0 <= y < n and echiquier[x][y] == -1

    def make_choice(state, x, y):
        state.append((x, y))
        echiquier[x][y] = len(state)
```

Diviser er régner 先分到最小，然后合起来的时候一定要注意从整体考虑，比如两部分，该如何选择其中一部分。可能选择的这一步要多写一个函数，如 merge_sort

Master theorem:

$$T(n) = aT(\tfrac{n}{b}) + O(n^d)$$

$$Total\ work = \textstyle\sum_0^{log_b n} O(n^d)(\frac{a}{b^d})^i$$

$$T(n) = a'T(\lceil n/b \rceil) + a''T(\lfloor n/b \rfloor) + \Theta(n^k)$$

$+\Theta(n^k)$

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log_b n) & si\ a = b^k \\ \Theta(n^{\log_b a}) & si\ a > b^k \end{cases}$$

等比数列公比：$\frac{a}{b^d}$ 与 1作比较：

若大于1，则最后一项占比最大：$a > b^d, T(n) = O(a^{log_b n})$

若小于1，最第一项占比最大：$a < b^d, T(n) = O(n^d)$

若等于1，则要将整个求和：$a = b^d, T(n) = O(n^d log_b n)$

Dp:类似分治，找到合适的递归方程，然后从小往大开始用循环并用矩阵存储值，每一个 i 都是这个 i 的解。sous-structure optimale, sous-problèmes qui se chevauchent

Algo glouton : à chaque étape on fait le choix qui optimise localement l'objectif

```python
# 最短作业优先调度算法：最短作业优先进行
def sjf(C, n):
    return sorted(list(range(0,n)), key=lambda x: C[x])
# trouve le prochain instant auquel il y a des nouvelle
def next_srtf(D, t, n):
    m = math.inf
    for x in D:
        if x < m and x > t:
            m = x
    return m
```

```python
# 最短剩余时间优先调度算法
def srtf(C, D, t, n):
    if t < math.inf:
        m = math.inf
        k = -1
        for x in range(0, n):
            if D[x] <= t and C[x] < m:
                k = x
                m = C[x]
        if k == -1:
            # on a pas trouvé de tâche à or
            return []
        else:
            t2 = next_srtf(D, t, n)
            # la tâche en cours sera finie ava
            if t2 >= t + C[k]:
                t2 = t + C[k]
                D[k] = math.inf
            C[k] = C[k] - (t2 - t)
            return [k] + srtf(C, D, t2, n)
    else:
        return []
```

```python
def partition(A, left, right):
    i = left   # 选择哨兵
    j = right
    while i < j:
        while A[j] >= A[left] and i < j:
            j -= 1
        while A[j] <= A[left] and i < j:
            i += 1
        A[i], A[j] = A[j], A[i]
    A[i], A[left] = A[left], A[i]    # 
    return i
```

```python
for k in range(len(A)):
    if i > left_end:
        A[k] = tmp[j]
        j += 1
    elif j > right_end or tmp[i] < tmp[j]:
        A[k] = tmp[i]
        i += 1
    else:
        A[k] = tmp[j]
        j += 1
```

File : enqueue, dequeue—FIFO

Pile : empiler, depiler(pop)—LIFO

Dérécursification : ?

Tas de binaire == Files de priorité: Toujours sortir le plus petit élément de la file $O(n\,log2\,n)$)

重点问题是分清楚是 max heap 还是 min heap，反正就是要注意性质$n \,//\, 2 - 1$和$2 * i + 1, 2 * i + 2$的运用。插入删除都涉及往前回溯还是往后回溯，堆排序涉及到往后回溯。$\lfloor n/2 \rfloor$是叶子，$\lfloor n/2 \rfloor - 1$是最后一个叶子上面的父节点的索引。

```python
def heapsort(A):
    n = len(A)
    for i in range(n//2-1, -1, -1):
        heap_fix_down_max(A, i, n)
    for i in range(n-1, 0, -1):
        A[0], A[i] = A[i], A[0]
        heap_fix_down_max(A, 0, i)
```

```python
# 确保永远是最大堆
def heap_fix_down_max(A, i, n):
    largest = i
    left  =2*i + 1
    right = 2*i + 2
    if left < n and A[left] > A[largest]:
        largest = left
    if right < n and A[right] > A[largest]:
        largest = right
    if largest != i:
        A[largest], A[i] = A[i], A[largest]
        heap_fix_down_max(A, largest, n)
```

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

```python
def sift_down(self, i: int):
    """ 从节点 i 开始，从顶至底堆化"""
    while True:
        # 判断节点 i, l, r 中值最大的节点，记为 ma
        l, r, ma = self.left(i), self.right(i), i
        if l < self.size() and self.max_heap[l] > self.max_heap[ma]:
            ma = l
        if r < self.size() and self.max_heap[r] > self.max_heap[ma]:
            ma = r
        # 若节点 i 最大或索引 l, r 越界，则无须继续堆化，跳出
        if ma == i:
            break
        # 交换两节点
        self.swap(i, ma)
        # 循环向下堆化
        i = ma
```

Arbre binaire

Un arbre n-aire a au plus n sous-arbres qui sont eux-aussi n-aires. La hauteur de l'arbre est la longueur du plus long chemin de la racine à une feuille

Un arbre est équilibré en hauteur si les hauteurs de ses sous-arbre diffèrent d'au plus 1. Un arbre n-aire équilibré = une hauteur d'au plus $log_n m$ $O(hauteur)$

二叉搜索树先注意定义，左边的永远小于根节点，右边的永远大于根节点。并且区分 prefix, inorder, postfix，这三种遍历方式。递归时要注意终止条件，和哪些时候到左边或者右边，分清楚多种情况，什么时候该添加或者删除

删除：三种情况—某一边没有，两边都有—让右子树最小的替换到 x

AVL：est un ABR dans lequel pour tout nœud x , le sous-arbre enraciné en x est équilibré (en hauteur)

le facteur d'équilibre : A.b = hauteur(A.d) - hauteur(A.g) [-2, 2] $O(log2\ n)$ 操作结合 ABR

```python
def avl_rotate_left(A):
    R = A.d
    A.d = R.g
    R.g = A
    A.b = A.b - 1
    if R.b > 0:
        A.b = A.b - R.b
    R.b = R.b - 1
    if A.b < 0:
        R.b = R.b + A.b
    return R
```

```python
def avl_rotate_right(A):
    R = A.g
    A.g = R.d
    R.d = A
    A.b = A.b + 1
    if R.b < 0:
        A.b = A.b - R.b
    R.b = R.b + 1
    if A.b > 0:
        R.b = R.b + A.b
    return R
```

```python
def avl_balance(A):
    R = A
    if A != None:
        if A.b == 2:
            if A.d.b < 0:
                A.d = avl_rotate_right(A.d)
            R = avl_rotate_left(A)
        elif A.b == -2:
            if A.g.b > 0:
                A.g = avl_rotate_left(A.g)
            R = avl_rotate_right(A)
    return R
```

```python
def avl_insert(A, x, p, bfu):
    retrace = (p != None)
    if A == None:
        B = node(x, p)
    else:
        if x < A.r:
            A.g = avl_insert(A.g, x, A, -1)
        else:
            A.d = avl_insert(A.d, x, A, +1)
        B = avl_balance(A)
        if B.b == 0:
            retrace = False
    if retrace:
        p.b = p.b + bfu
    return B
```

Tables de hachage: collisions T 是一个 list, h 映射—x 在 T 的位置

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

```python
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
```

La complexité pire cas ne dépend pas de la taille m de la table，去链表搜索

adressage ouvert :

$hash(x)\ +\ i$

```python
def htoa_search(T, x):
    i = 0
    m = len(T)
    while i < m and T[h(x, i)] is not None and T[h(x, i)] != x:
        i = i + 1
    return i < m and T[h(x, i)] == x
```

```python
def htoa_insert(T, x):
    i = 0
    m = len(T)
    while i < m and T[h(x, i)] is not None:
        i = i + 1
    k = h(x, i)
    if T[k] is None:
        T[k] = x
    else:
        raise Exception("Table pleine")
```

如果遇到符号判定，直接 « if 是不是 »就行

Soit A_i le tableau après l'instruction i et s_i sa taille ;

On définit la **fonction de potentiel** $\phi(i) = 2i - s_i$;

Coût potentiel des insertions :

- Si $i < s_i$: $1 + (2 * (i+1) - s_{i+1}) - (2 * i - s_i) = 3$ car $s_{i+1} = s_i$;
- Si $i = s_i$: $(1 + s_i) + (2 * (i+1) - s_{i+1}) - (2 * i - s_i) = 3 + s_i - k$ car $s_{i+1} = s_i + k$;

Avec $k = s_i$, on a donc une complexité amortie $\tilde{O}(1)$.

Le **coût potentiel** de l'instruction i_k est $p_k = c_k + \phi(\vec{v}_k) - \phi(\vec{v}_{k-1})$;

- On calcule le coût potentiel de chaque opération :
  - si $i_k =$ 'c', $p_k = 1 + \phi(\vec{v}_k) - \phi(\vec{v}_{k-1}) = 1 + ((s+1)+1) - (s+1) = 2$
  - si $i_k =$ del, $p_k = 1 + (s+1) - ((s+1)+1) = 0$
  - si $i_k =$ clear, $p_k = (1+s) + 1 - (s+1) = 1$
- Chaque opération a bien un coût potentiel $O(1)$ donc le coût potentiel total est $O(n)$, donc le coût amorti total est aussi $O(n)$ et le coût amorti de clear est $\tilde{O}(1)$.

Le **coût amorti** de chaque instruction est la moyenne $\frac{1}{n}\sum_{k=1}^{n} c_k$

失衡节点的平衡因子	子节点的平衡因子	应采用的旋转方法
>1（即左偏树）	≥ 0	右旋
>1（即左偏树）	<0	先左旋后右旋
<-1（即右偏树）	≤ 0	左旋
<-1（即右偏树）	>0	先右旋后左旋

```python
# 遍历所有列
for col in range(n):
    # 计算该格子对应的主对角线和副对角线
    diag1 = row - col + n - 1
    diag2 = row + col
    # 剪枝：不允许该格子所在列、主对角线、副对角线存在皇后
    if not cols[col] and not diags1[diag1] and not diags2[diag2]:
        # 尝试：将皇后放置在该格子
```

```python
# 放置下一行
backtrack(row + 1, n, state, res, cols, diags1, diags2)
# 回退：将该格子恢复为空位
state[row][col] = "#"
cols[col] = diags1[diag1] = diags2[diag2] = False
```

```python
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if s[i - 1] == t[j - 1]:
            # 若两字符相等，则直接跳过此两字符
            dp[i][j] = dp[i - 1][j - 1]
        else:
            # 最少编辑步数 = 插入、删除、替换这三种操作的最少编辑步数 + 1
            dp[i][j] = min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1
```