

0. 汉诺塔问题

1. 查找算法

- 1.1 顺序查找 (Linear Search)
- 1.2 二分查找 (Binary Search)
- 1.3 内置idx()函数

2. 排序算法

- 2.1 冒泡排序 (Bubble Sort)
- 2.2 选择排序 (Select Sort)
- 2.3 插入排序 (Insert Sort)
- 2.4 快速排序
- 2.5 堆排序 (Heap Sort)
 - 2.5.1 堆排序过程
 - 2.5.2 topk问题
- 2.6 归并排序
- NB三人组总结
- 2.7 希尔排序
- 2.8 计数排序
- 2.9 桶排序 (Bucket Sort)
- 2.10 基数排序

3. 数据结构

- 3.1 数据结构介绍
- 3.2 列表
- 3.3 栈
- 3.4 队列
- 3.5 链表
 - 3.5.1 链表的创建和遍历
 - 3.5.2 链表的插入和删除
 - 3.5.3 链表总结
- 3.6 哈希表 (散列表)
- 3.7 树
 - 3.7.1 二叉树
 - 3.7.2 二叉树的遍历
 - 3.7.3 二叉搜索树
 - 3.7.4 AVL树
 - 3.7.5 二叉搜索树扩展应用——B树

4. 贪心算法

- 4.1 找零问题
- 4.2 背包问题
- 4.3 数字拼接问题
- 4.4 活动选择问题

5. 动态规划

- 5.1 钢条切割问题
- 5.2 最长公共子序列LCS(Longest Common Subsequence)

6. 欧几里得算法

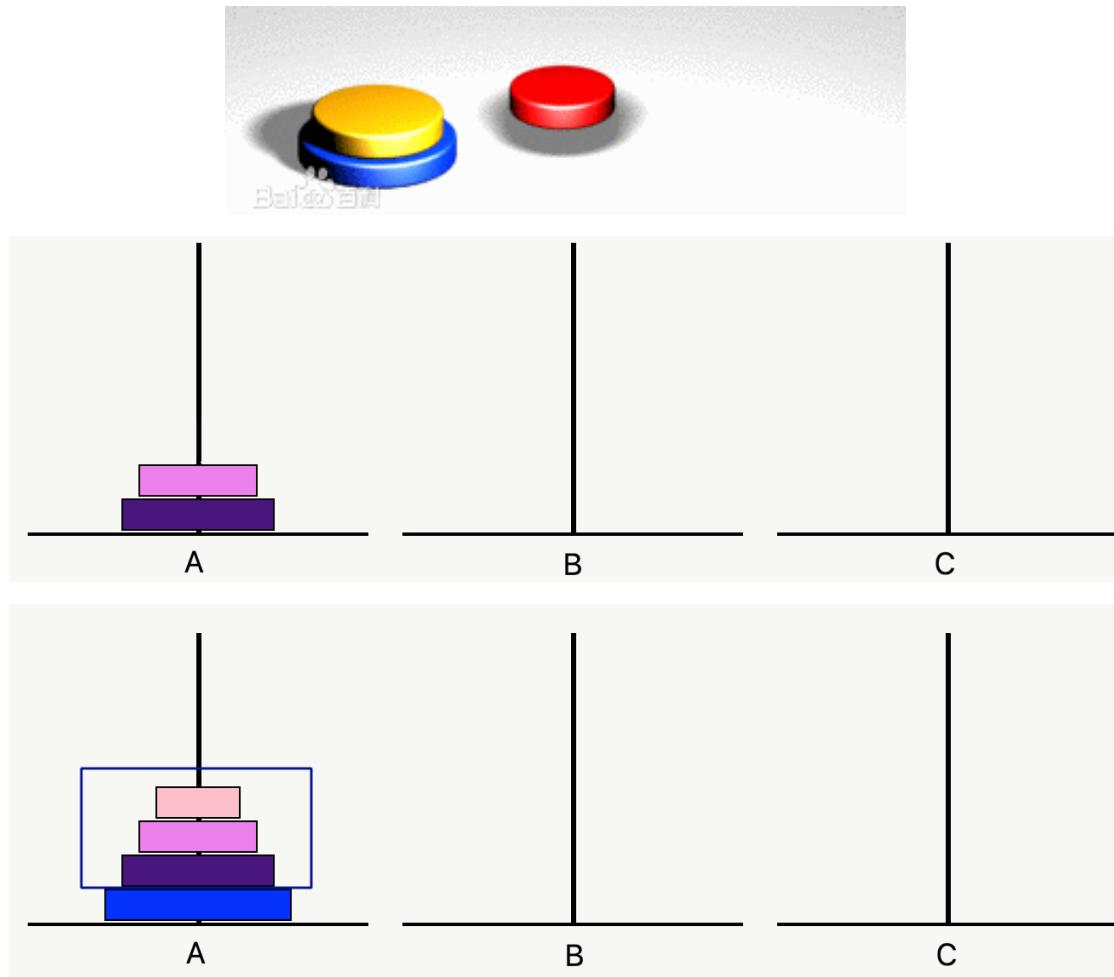
7. RSA算法

8. 字符串算法

- 8.1 哈希法
- 8.2 KMP

- 8.3: 字典树 (trie树)
- 8.4 AC自动机
- 8.5 Manacher算法 (回文串问题)
 - 暴力解法
 - 中心扩展法
- 9. Dijkstra (迪杰斯特拉算法)
- 10. Floyd算法
 - 10.1 Dijkstra和Floyd算法比较

0. 汉诺塔问题



当 $n = 2$ 时:

1. 把小圆盘从A移动到B;
2. 把大圆盘从A移动到C;
3. 把小圆盘从B移动到C;

当 $n > 2$ 时:

1. 把上面 $n - 1$ 个盘子从A经C移动到B;
2. 把第 n 个圆盘从A移动到C;
3. 把 $n - 1$ 个小圆盘从B经A移动到C;

```
def hanoi(n, a, b, c):
    ...
    n: 剩下盘子数
    a: 第一个柱子名
    b: 第二个柱子名
    c: 第三个柱子名
    ...
    if n > 0:
        hanoi(n-1, a, c, b)
        print('moving from %s to %s' % (a, c))
        hanoi(n-1, b, a, c)
```

```
hanoi(3, 'A', 'B', 'C')
```

```
moving from A to C
moving from A to B
moving from C to B
moving from A to C
moving from B to A
moving from B to C
moving from A to C
```

```
hanoi(2, 'A', 'B', 'C')
```

```
moving from A to B
moving from A to C
moving from B to C
```

1. 查找算法

1.1 顺序查找

1.2 二分查找

1.1 顺序查找 (Linear Search)

顺序查找: 也叫线性查找, 从列表第一个元素开始, 顺序进行搜索, 直到找到元素或搜索到列表最后一个元素为止
时间复杂度: $O(n)$

```
def linear_search(li, target):
    for idx, val in enumerate(li):
        if val == target:
            return idx
    return None
```

```
idx = linear_search([1,4,2,5], 5)
print(idx)
```

3

1.2 二分查找 (Binary Search)

二分查找：又叫折半查找，从有序列表的初始候选区 $li[0:n]$ 开始，通过对待查找的值与候选区中间值的比较，可以使候选区减少一半。

时间复杂度： $O(\log n)$

```
def bianry_search(li, target):
    begin, end = 0, len(li)-1
    while end >= begin:
        if target == li[(begin+end)//2]:
            return (begin+end)//2
        elif target < li[(begin+end)//2]:
            end = (begin+end)//2 - 1
        else:
            begin = (begin+end)//2 + 1
    return None
```

```
idx = bianry_search([1,3,5,6,8,10], 4)
print(idx)
```

```
idx = bianry_search([1,3,5,6,8,10], 6)
print(idx)
```

None

3

```
def bin_search(li, target):
    left, right = 0, len(li)-1
    while left <= right:
        mid = (left + right) >> 1
        if target == li[mid]:
            return mid
        elif target < li[mid]:
            right = mid - 1
        else:
            left = mid + 1
    return None
```

```
import random

li = [random.randint(0, 9) for _ in range(10)]
li.sort()
print(li)
idx = bin_search(li, 3)
print(idx)
```

```
[0, 0, 1, 3, 5, 5, 5, 7, 9, 9]
3
```

1.3 内置idx()函数

内部用的是线性查找

```
li = ['a', 'v', 'd', 'g', 'b']
idx = list.index(li, 'g')
print(idx)
```

```
3
```

```
li = list(range(15))
idx = list.index(li, 6)
print(idx)
```

```
6
```

2. 排序算法

- 2.1 冒泡排序
- 2.2 选择排序
- 2.3 插入排序
- 2.4 快速排序
- 2.5 堆排序
- 2.6 归并排序
- 2.7 希尔排序
- 2.8 计数排序
- 2.9 基数排序
- 2.10 内置sort()

2.1 冒泡排序 (Bubble Sort)

- 列表每两个相邻的数，如果前面比后面大(默认升序)，则交换这两个数；
- 一趟排序完成后，则无序区减少一个数，有序区增加一个数。
- 时间复杂度： $O(n^2)$ ，有两层循环

```
def bubble_sort(li):  
    for i in range(len(li)-1): # 第i趟比较  
        for j in range(len(li)-i-1):  
            if li[j] > li[j+1]:  
                li[j], li[j+1] = li[j+1], li[j] # 交换两个元素  
    return li
```

```
li = bubble_sort([4,2,7,0,6,8,1,10,0])  
li
```

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

改进1：

若一趟比较没有发生元素交换，则认为无序区元素已经有序，可以停止排序

```
def bubble_sort(li):  
    for i in range(len(li)-1): # 第i趟比较  
        exchange = False # 标志位  
        for j in range(len(li)-i-1):  
            if li[j] > li[j+1]:  
                li[j], li[j+1] = li[j+1], li[j] # 交换两个元素  
                exchange = True  
        if not exchange:  
            return li  
    return li
```

```
import random  
  
li = list(range(50))  
random.shuffle(li)  
print(li)  
li = bubble_sort(li)  
print(li)
```

```
[12, 32, 45, 21, 43, 30, 38, 24, 17, 3, 8, 39, 31, 44, 46, 34, 4, 35, 28, 6, 26, 48, 36, 47, 18, 19, 37, 13, 1, 49, 42, 20, 41, 7, 29, 23, 0, 14, 15, 22, 10, 16, 9, 27, 5, 40, 2, 11, 25, 33]
[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, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

2.2 选择排序 (Select Sort)

```
def select_sort_simple(li):
    sorted_li = []
    for i in range(len(li)): # 一共n趟查找
        min_val = min(li) # O(n)
        sorted_li.append(min_val)
        li.remove(min_val) # O(n)
    return sorted_li
```

```
import random

li = [random.randint(0,100) for i in range(10)]
print(li)
print(select_sort_simple(li))
```

```
[80, 85, 35, 35, 3, 17, 14, 64, 85, 60]
[3, 14, 17, 35, 35, 60, 64, 80, 85, 85]
```

上面的选择排序不推荐：

1. 会新生成一个列表，占用内存空间；
2. 虽然只有一层循环，但是`min()`和`remove()`操作都是 $O(n)$ ，因此总的时间复杂度是 $O(n^2)$

```
def select_sort(li):
    for i in range(len(li)-1): # 第i趟选择，共需n-1趟
        min_idx = i
        for j in range(i+1, len(li)):
            if li[j] < li[min_idx]:
                min_idx = j
        li[i], li[min_idx] = li[min_idx], li[i]
    return li
```

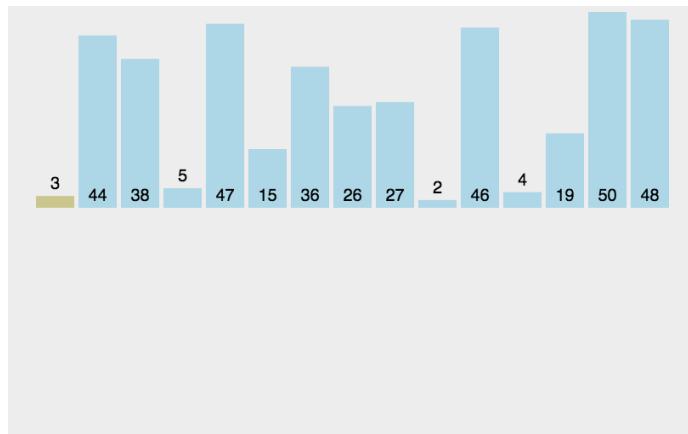
```
import random

li = [random.randint(0,100) for i in range(10)]
print(li)
print(select_sort(li))
```

```
[31, 63, 80, 87, 68, 17, 38, 24, 22, 8]
[8, 17, 22, 24, 31, 38, 63, 68, 80, 87]
```

2.3 插入排序 (Insert Sort)

1. 初始时手里（有序区）只有一张牌；
2. 每次（从无序区）摸一张牌，插入到手里已有牌的正确位置
3. 时间复杂度： $O(n^2)$



```
def insert_sort(li):
    for i in range(1, len(li)): # 第i次插入, 共需n-1次插入, 从第二个元素开始, 表示摸到的牌
        tmp = li[i] # 摸到的牌
        j = i - 1 # 从摸到的牌的前一张牌开始, 进行比较和挪动
        while j >= 0 and li[j] > tmp:
            li[j+1] = li[j]
            j -= 1
        li[j+1] = tmp
    return li
```

```
import random

li = [random.randint(0,20) for i in range(10)]
print(li)
print(insert_sort(li))
```

```
[20, 16, 18, 9, 8, 1, 5, 3, 18, 12]
[1, 3, 5, 8, 9, 12, 16, 18, 18, 20]
```

```

import time

li = list(range(10000))
random.shuffle(li)
begin_time = time.time()
print(bubble_sort(li))
end_time = time.time()
print(end_time - begin_time)

```

2.4 快速排序

思想：

1. 取一个元素p（第一个元素），使元素p归位；
2. 列表被p分成两部分，左边都比p小，右边都比p大；
3. 递归完成排序。

时间复杂度： $n \log n$

```

def partition1(li, left, right):
    ...
    归位函数
    ...
    tmp = li[left] # 缓存第一个元素，也就是
    将第一个元素归位
    flag = True
    while left < right:
        if li[right] < tmp and flag:
            li[left] = li[right]
            left += 1
            flag = False
        elif li[right] >= tmp and flag:
            right -= 1
        elif li[left] < tmp and not
            flag:
            left += 1
        elif li[left] >= tmp and not
            flag:
            li[right] = li[left]
            right -= 1
            flag = True
    li[left] = tmp
    return left

```

```
def partition(li, left, right):
    tmp = li[left]
    while left < right:
        while right > left and li[right] >= tmp:
            right -= 1
        li[left] = li[right]
        while left < right and li[left] <= tmp:
            left += 1
        li[right] = li[left]
    li[left] = tmp
    return left
```

```
li = [random.randint(0,20) for i in range(15)]
li2 = li
print(li)
partition1(li,0,len(li)-1)
partition1(li2,0,len(li2)-1)
print(li)
print(li2)
```

```
[1, 10, 17, 20, 17, 12, 3, 16, 2, 6, 4, 7, 1, 5, 14]
[1, 10, 17, 20, 17, 12, 3, 16, 2, 6, 4, 7, 1, 5, 14]
[1, 10, 17, 20, 17, 12, 3, 16, 2, 6, 4, 7, 1, 5, 14]
```

```
def _quick_sort(li, left, right):
    ...
    利用递归实现快速排序
    ...
    if left < right:
        mid = partition(li, left, right)
        _quick_sort(li, left, mid-1)
        _quick_sort(li, mid+1, right)

def quick_sort(li):
    _quick_sort(li, 0, len(li)-1)
```

```
import random

li = list(range(10))
random.shuffle(li)
print(li)
quick_sort(li)
print(li)
```

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

```
import time
from copy import deepcopy

li = list(range(10000))
random.shuffle(li)

l1 = deepcopy(li)
l2 = deepcopy(li)

begin_time = time.time()
bubble_sort(l1)
end_time = time.time()
print('冒泡排序时间: ', end_time - begin_time)

begin_time = time.time()
quick_sort(l2)
end_time = time.time()
print('快速排序时间: ', end_time - begin_time)
```

```
冒泡排序时间: 8.77525782585144
快速排序时间: 0.024220943450927734
```

快速排序的两个问题：

1. 最坏情况：

当待排序列表是逆序列表时，不能很好的partition，此时时间复杂度为 $O(n^2)$ ；

解决办法：随机选择一个位置的元素和起始位置元素交换，开始快排

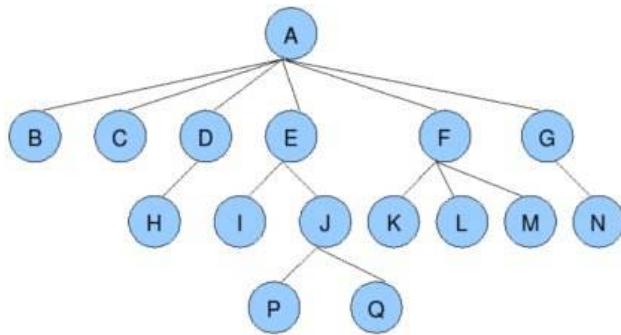
2. 递归

python解释器为了避免内存溢出和性能影响，设置了最大递归深度为998，当调用栈超过998层就会报错；

解决办法：手动修改最大递归深度，`sys.setrecursionlimit(3000)`

2.5 堆排序（Heap Sort）

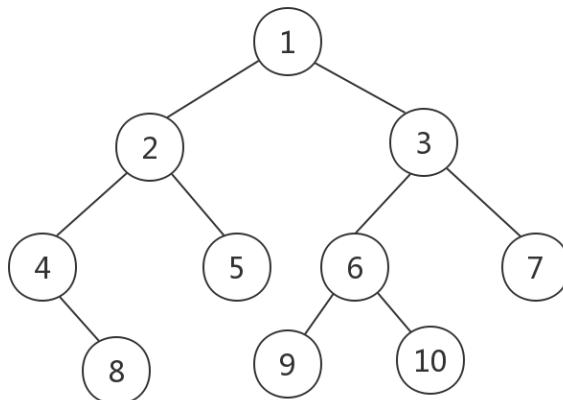
树



1. 根节点: A
2. 树的深度 (高度) : 4
3. **树的度:** 6 树中最多子节点的个数
4. 孩子节点/父节点
5. 子树

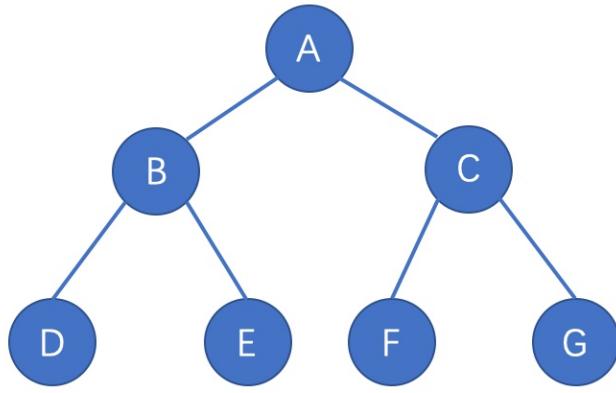
二叉树

1. 度不超过2的树;
2. 每个节点最多有2个孩子节点;
3. 两个孩子节点被区分为 左孩子节点 和 右孩子节点



满二叉树

一个二叉树，如果每一层的结点数都达到最大值，则这个二叉树就是满二叉树

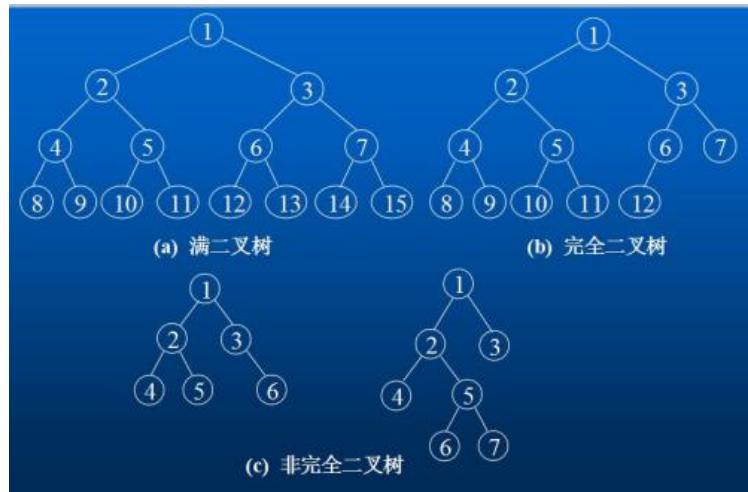


满二叉树

知乎 @GhostClock

完全二叉树

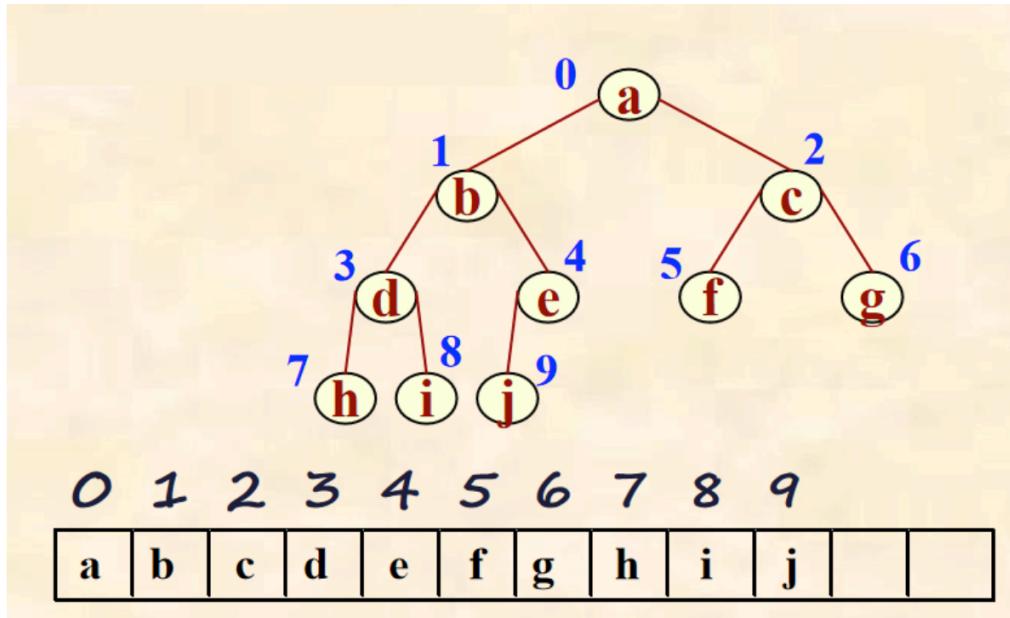
叶节点只能出现在最下层和次下层，并且最下面一层的节点都集中在该层最左边的若干位置的二叉树



二叉树的存储方式

1. 链式存储方式
2. 顺序存储方式

二叉树的顺序存储方式：

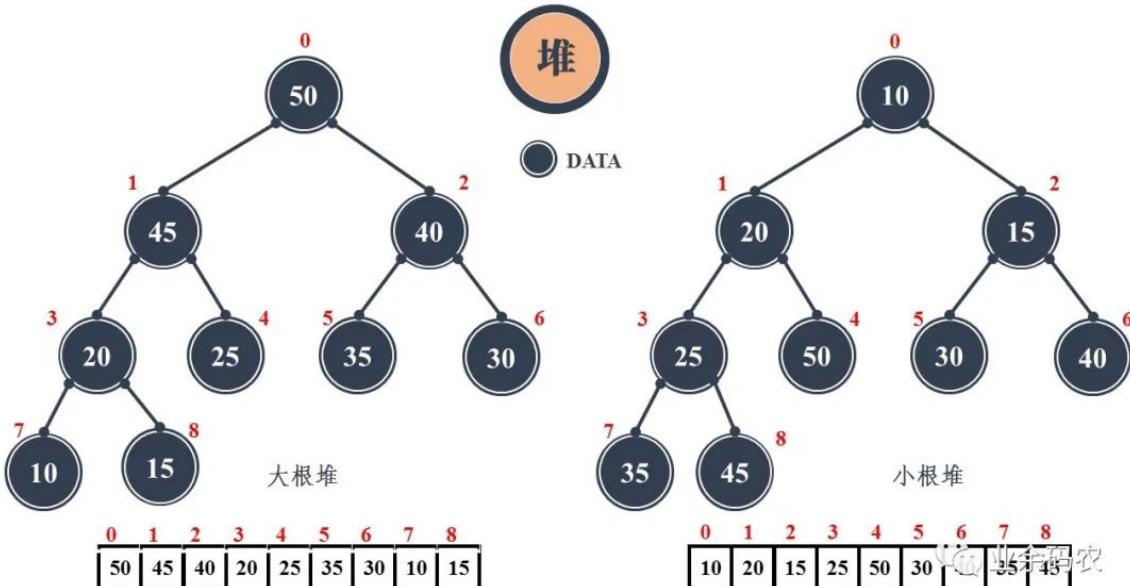


1. 父节点与左孩子节点的编号下标关系: $i \rightarrow 2i+1$

2. 父节点与右孩子节点的编号下标关系: $i \rightarrow 2i+2$

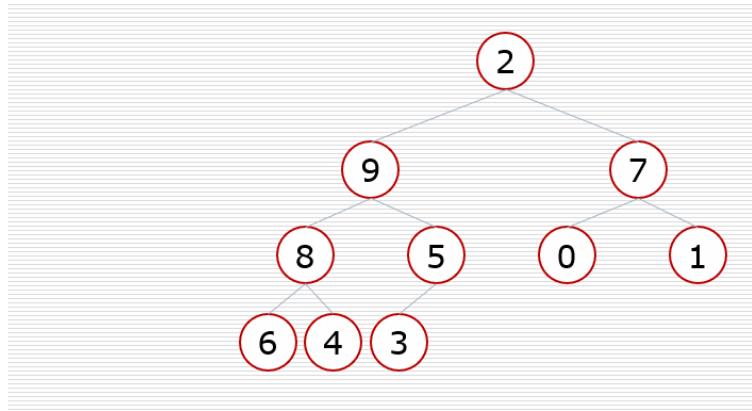
大根堆、小根堆

1. 一棵 完全二叉树， 满足任一节点都比其孩子节点大；
2. 一棵 完全二叉树， 满足任一节点都比其孩子节点小；



堆的向下调整：

假设根节点的左右子树都是堆，但根节点不满足堆的性质，可以通过一次向下的调整来将其变成一个堆。



2.5.1 堆排序过程

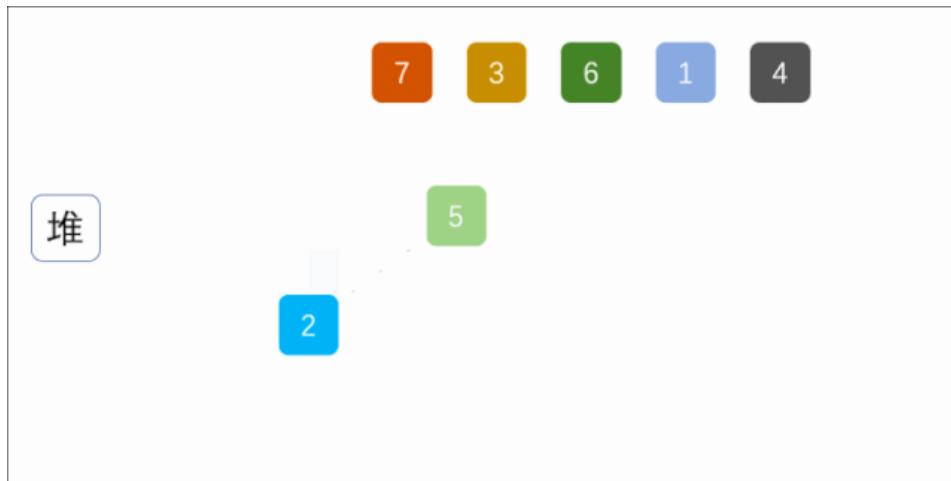
1. 构造堆

从下往上构建大根堆

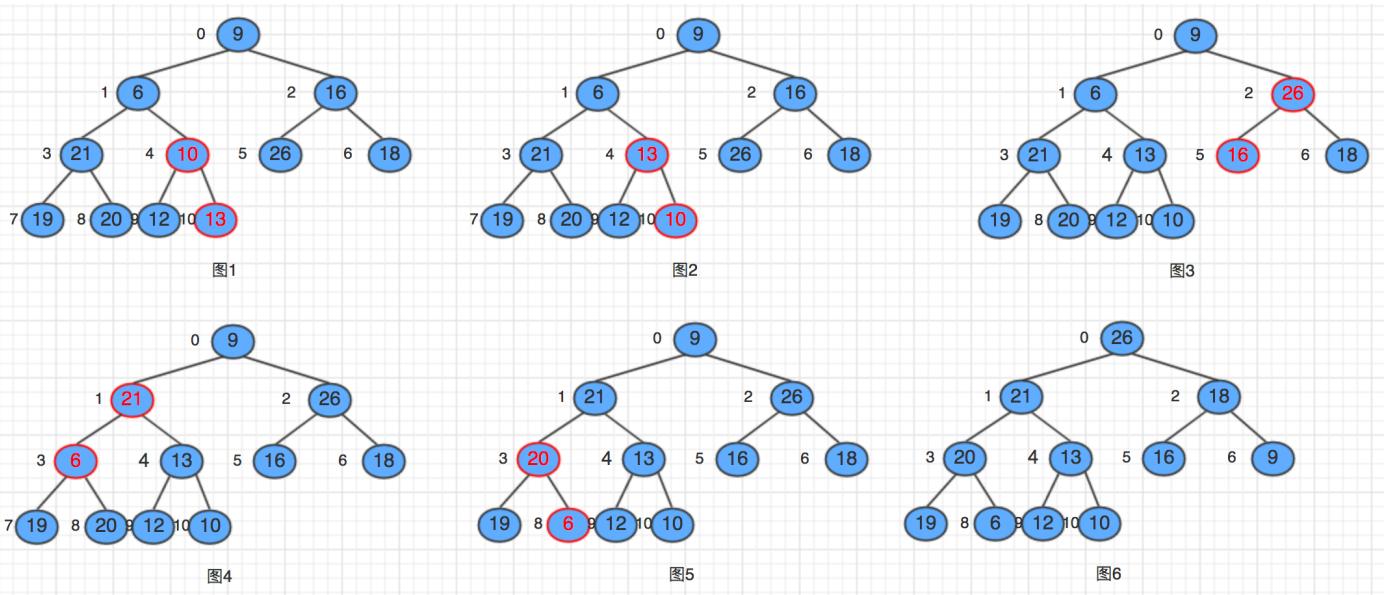
2. 挨个出数

- 将根节点元素拿出；
- 将最后一个叶节点挪到根节点位置；
- 运用堆的向下调整，实现大根堆；
- 重复上面的操作，直至所有元素被排好序

堆排序过程：



构建大根堆

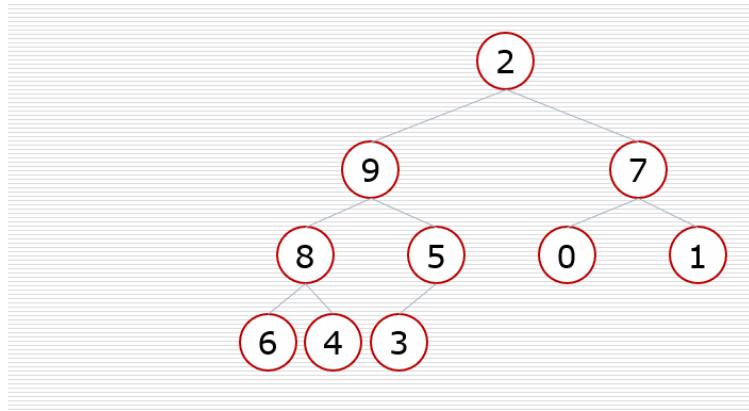


```

def sift(li, low, high):
    ...
    堆的向下调整
    low: 堆的根节点位置
    high: 堆最后一个元素位置
    ...
    i = low          # 根节点
    j = 2 * i + 1 # 左子节点
    tmp = li[i]

    while j <= high:
        # 判断右子节点
        if j+1 <= high and li[j+1] > li[j]: # 右子节点存在, 且大于左子节点
            j += 1

        if li[j] > tmp: # 子节点 比 根节点大
            li[i] = li[j]
            i = j
            j = 2 * i + 1
        else:
            break
    else:
        li[i] = tmp
    
```



```
li = [2,9,7,8,5,0,1,6,4,3]
sift(li,0,len(li)-1)
print(li)
```

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

```
def heap_sort(li):
    ...
    1. 构造大根堆
    2. 将根节点元素取出
    3. 将最后一个叶子节点挪到根节点位置
    4. 堆的向下调整，实现大根堆
    5. 重复上面2~4步，直到所有元素已排好序
    ...
# 1. 构造大根堆
n = len(li)
idx = (n-1)//2 # 获取最后一个非叶子节点编号
for i in range(idx, -1, -1):
    sift(li, i, n-1) # 始终将n-1作为high
for i in range(n-1, 0, -1):
    # 2. 将根节点元素取出, 和第i个节点交换元素
    li[0], li[i] = li[i], li[0]
    # 3. 对剩下的堆进行向下调整
    sift(li, 0, i-1)
```

```

import random

li = [random.randint(0, 15) for _ in range(10)]
print(li)
heap_sort(li)
print(li)

print()
li = [random.randint(0, 15) for _ in range(9)]
print(li)
heap_sort(li)
print(li)

```

```
[15, 12, 1, 2, 3, 7, 2, 12, 15, 6]
[1, 2, 3, 6, 7, 7, 12, 15, 15]
```

```
[12, 3, 11, 7, 15, 12, 2, 13, 4]
[2, 3, 4, 7, 11, 12, 13, 13, 15]
```

Heap Sort时间复杂度：

sift()的时间复杂度为 $O(\log n)$

堆排序时间复杂度： $O(n \log n)$

堆的内置模块heapq

```

import heapq
import random

li = list(range(24))
random.shuffle(li)
print(li)

# 建堆，默认是小根堆
heapq.heapify(li)
print(li)

# 依次弹出元素
for i in range(len(li)):
    print(heapq.heappop(li), end=', ')

```

```
[11, 21, 23, 19, 8, 2, 3, 1, 7, 10, 20, 13, 5, 18, 14, 15, 0, 9, 4, 6, 12, 17, 16, 22]
[0, 1, 2, 4, 6, 5, 3, 15, 7, 8, 16, 13, 23, 18, 14, 21, 19, 9, 11, 10, 12, 17, 20, 22]
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
```

2.5.2 topk问题

现有n个数，设计算法得到前k大的数。 (k<n)

解决思路：

1. 排序后切片： $O(n \log n)$
2. 排序LowB三人组（冒泡、插入、选择）： $O(kn)$
3. 堆排序思路： $O(n \log k)$

堆排序实现topk解决思路：

1. 取列表前k个元素建立一个小根堆，则堆顶便是目前第k大的数；
2. 依次向后遍历原列表，对于列表中的每个元素，如果小于堆顶，则忽略该元素；如果大于堆顶，则将该元素更换为堆顶元素，并对堆进行一次调整；
3. 遍历列表所有元素，倒序弹出堆顶。

```
...
堆排序实现topk
...
def sift(li, low, high):
    ...
    向下调整为小根堆
    ...
    i = low # i代表子树的根
    j = 2 * i + 1 # 左子节点
    tmp = li[low]
    while j <= high:
        if j+1 <= high and li[j+1] < li[j]:
            j += 1 # 切到右子节点
        if li[j] < li[i]:
            li[i], li[j] = li[j], li[i]
            i = j
            j = 2 * i + 1
        else:
            break

def topk(li, k):
    ...
    获取li列表中前k大的数
    ...
    heap = li[:k]

    # 1. 取前k个元素，构建小根堆
    i = (k - 2) // 2 # 最后一个非叶子节点
    for j in range(i, -1, -1):
        sift(heap, j, k-1)

    # 2. 遍历剩余len(li)-k个元素
    for i in range(k, len(li)):
```

```

if li[i] > heap[0]: # 若第k个元素比堆顶元素大，则替换调
    heap[0] = li[i]
    sift(heap, 0, k-1)

# 3. 倒序取出前k个数
for i in range(k-1, -1, -1):
    heap[0], heap[i] = heap[i], heap[0]
    sift(heap, 0, i - 1)

return heap

```

```

import random

li = [random.randint(0,50) for i in range(10)]
# li = [25, 25, 2, 14, 1, 15, 1, 18]
print('原列表: ', li, sep='\n')

top = topk(li, 4)
print(top)

```

原列表:

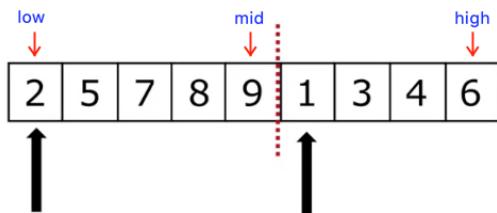
```
[47, 4, 0, 42, 34, 28, 6, 34, 11, 24]
[47, 42, 34, 34]
```

2.6 归并排序

6	5	3	1	8	7	2	4
---	---	---	---	---	---	---	---

归并

假设现在的列表分两段有序，如何将其合成为一个有序列表？



这种操作称为一次归并

```

def merge(li, low, mid, high):
    ...
    一次归并操作
    ...

    i = low
    j = mid + 1

    new_li = []
    while i <= mid:
        if j <= high:
            if li[i] <= li[j]:
                new_li.append(li[i])
                i += 1
            else:
                new_li.append(li[j])
                j += 1
        else:
            new_li.extend(li[i:mid+1])
            break
    if j <= high:
        new_li.extend(li[j:high+1])
    # 把new_li重新写会li, 后面递归会用到
    li[low:high+1] = new_li

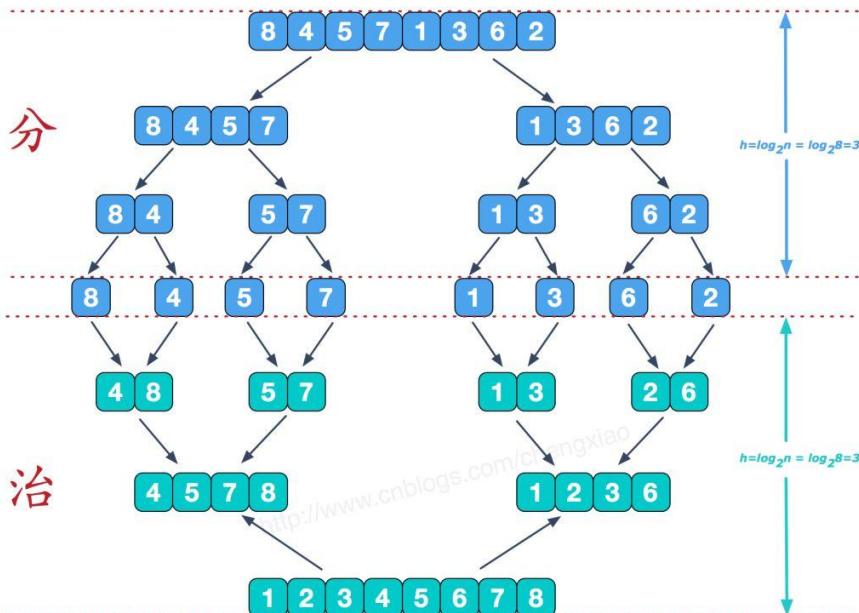
```

```

li = [1,3,4,6,2,5,7,8,9]
merge(li,0,3,len(li)-1)
print(li)

```

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



时间复杂度: $O(n \log n)$

| 有 $\log n$ 层, 每一层都需遍历, 所以是 $n \log n$

空间复杂度: $O(n)$

| 在 merge 过程中有新开辟一个列表, 所以归并排序不是原地排序, 空间复杂度为 $O(n)$

```
def merge_sort(li, low, high):
    ...
    归并排序: 递归实现
    ...
    if low < high:
        mid = (low + high) // 2
        merge_sort(li, low, mid) # 左边部分
        merge_sort(li, mid+1, high) # 右边部分
        merge(li, low, mid, high) # 归并左右两边
```

```
li = [8, 4, 5, 7, 1, 3, 6, 2]
merge_sort(li, 0, len(li)-1)
print(li)
```

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

NB三人组总结

1. 三种排序算法（快速排序、堆排序、归并排序）的时间复杂度都是 $n \log n$;
2. 一般情况下, 就运行时间而言: 快速排序 < 归并排序 < 堆排序
3. 三种排序算法的缺点:
 - 快速排序: 极端情况下排序效率低 (比如原始列表为倒序)
 - 归并排序: 需要额外的内存开销;
 - 堆排序: 在快的排序算法中相对较慢

排序方法	时间复杂度			空间复杂度	稳定性	代码复杂度
	最坏情况	平均情况	最好情况			
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	简单
快速排序	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	平均情况 $O(n \log n)$; 最坏情况 $O(n^2)$	不稳定	较复杂
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定	复杂
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定	较复杂

快速排序空间复杂度说明:

| 快排中涉及到递归调用, 递归调用会给函数开辟空间。快排中平均递归深度为 $\log n$, 最坏情况递归深度为 n

稳定性说明：

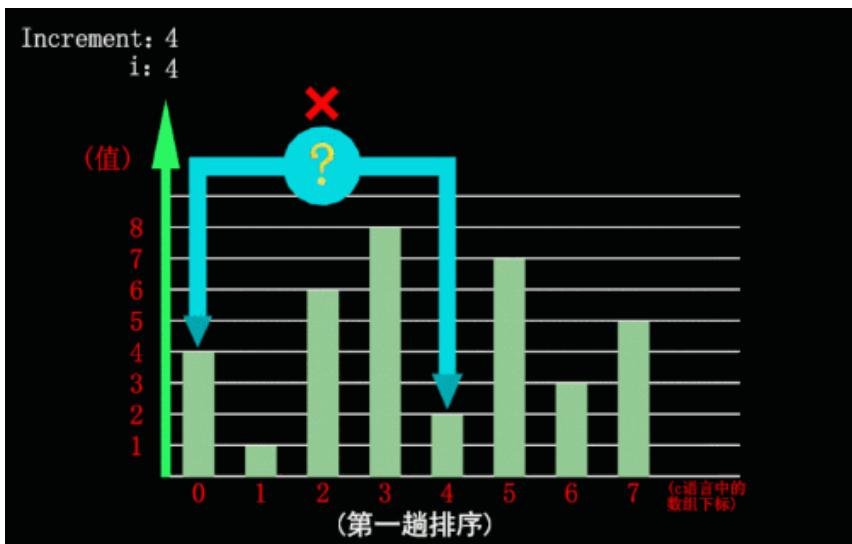
如果是相邻位置元素比较交换，则是稳定的，比如冒泡排序；

如果是隔着位置对元素进行比较交换，则是不稳定的，比如选择排序。0, 2, 4, 2, 1, 4

python内置排序是改进的归并排序，因为它是稳定的

2.7 希尔排序

1. 希尔排序 (Shell Sort) 是一种分组插入排序 算法；
2. 首先取一个整数 $d_1 = n/2$, 将元素分为 d_1 个组，每组相邻元素之间距离为 d_1 ，在各组内进行直接插入排序；
3. 取第二个整数 $d_2 = d_1/2$, 重复上述分组排序过程，直到 $d_i = 1$ ，即所有元素在同一组内进行直接插入排序；
4. 希尔排序每趟并不使某些元素有序，而是使整体数据越来越接近有序；最后一趟排序使得所有数据有序。



```
def insert_sort_gap(li, gap):
    ...
    分组插入排序
    gap: 组间距
    ...

    for i in range(gap, len(li)):
        tmp = li[i]
        j = i - gap
        while j >= 0 and li[j] >= tmp:
            li[j+gap] = li[j] # 因为第j位置元素比tmp大，所以将该位置元素往后挪gap位
            j -= gap
        li[j+gap] = tmp # while终止条件1) j<0; 2)li[j]<tmp 所以需将tmp写回到li[j+gap]位置上

def shell_sort(li):
    ...
    希尔排序
    ...
    gap = len(li) // 2
    while gap >= 1:
        insert_sort_gap(li, gap)
        gap //= 2
```

```
li = [3,5,2,0,7,6,4,1]
shell_sort(li)
print(li)
```

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

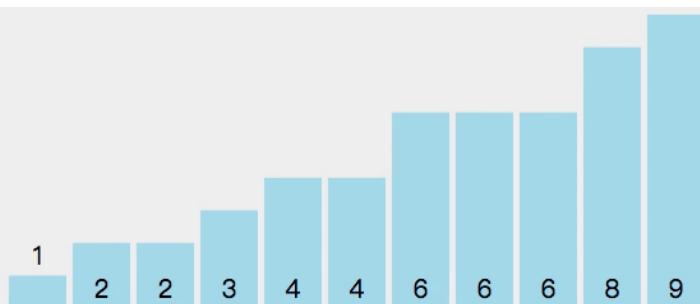
希尔排序时间复杂度：

希尔排序的时间复杂度与gap序列的选取有关。

2.8 计数排序

排序前提：

已知列表中数据的范围，比如都在0到100之间，设计时间复杂度为 $O(n)$ 的算法。



```
def count_sort(li, max_count=100):
    ...
    计数排序
    max_count: 元素的最大值
    数值范围: [0,max_count]
    ...
    count = [0 for _ in range(max_count+1)]

    for val in li:
        # 计数
        count[val] += 1

    li.clear()
```

```

for idx, val in enumerate(count):
    for i in range(val):
        li.append(idx)

```

```

import random

li = [random.randint(0,20) for i in range(15)]
print(li)
count_sort(li)
print(li)

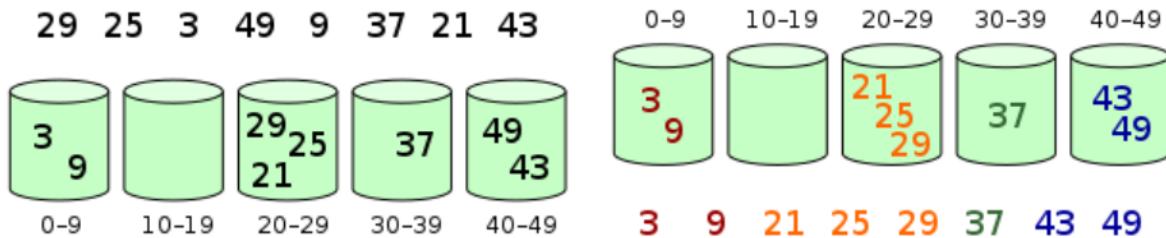
```

```
[3, 6, 7, 9, 20, 18, 13, 6, 2, 4, 13, 8, 13, 12, 12]
[2, 3, 4, 6, 6, 7, 8, 9, 12, 12, 13, 13, 13, 18, 20]
```

2.9 桶排序 (Bucket Sort)

- 在计数排序中，如果元素的范围比较大（比如1到1亿之间），我们需要开辟一个很大的列表，如何改造算法？

桶排序 (Bucket Sort)：首先将元素分在不同的桶中，再对桶中的元素进行排序，最后按桶的顺序，依次输出所有元素，即得到有序数列。



```

def bucket_sort(li, n=100, max_num=10000):
    ...
    桶排序
    n: 桶的个数
    max_num: 最大的元素
    ...

    # 1. 创建n个桶
    buckets = [[] for _ in range(n)]

    for val in li:
        # 2. 将元素依次放入对应的桶中
        idx = min(val // (max_num // 100), n-1) # min()的作用，将max_num元素放入最后一个桶,
        因为索引越界了
        buckets[idx].append(val)

        # 3. 放入的同时，对桶内元素进行插入排序
        for i in range(len(buckets[idx])-1, 0, -1):

```

```

# 1,3,4,5,2
if buckets[idx][i-1] > buckets[idx][i]:
    buckets[idx][i-1], buckets[idx][i] = buckets[idx][i], buckets[idx][i-1]
else:
    break
# 4. 将所有桶中元素依次取出
sorted_li = []
for buc in buckets:
    sorted_li.extend(buc)
return sorted_li

```

```

import random

li = [random.randint(0,100000) for _ in range(10000)]
sorted_li = bucket_sort(li, max_num=100000)
print(sorted_li)

```

桶排序性能总结：

1. 桶排序的表现取决于数据的分布。也就是需要对不同的数据排序时采用不同的分桶策略。（比方说，数据是均匀分布的，桶的大小可以一样；如果数据是正态分布的，那均值附近应分得更精细）；
2. 平均时间复杂度： $O(n + k)$;
3. 最坏情况时间复杂度： $O(n^2 k)$
4. 空间复杂度： $O(nk)$

2.10 基数排序

多关键字排序：

比如现有一员工表，要求按照年龄排序，年龄相同的按照薪资进行排序。

基数排序：

对数据的排序也可以看做是多关键字排序，先按个位排，然后按十位排，依次进行



@五分钟学算法之基数排序

```
def radix_sort(li):
    max_num = max(li) # 获取最大元素，决定进行几次桶排序
    length = len(str(max_num))

    buckets = [ [] for _ in range(10)] # 创建10个桶

    for i in range(length):
        for val in li:
            # 将元素放入对应的桶中
            i_val = (val % (10***(i+1)))//(10***i)
            buckets[i_val].append(val)

        # 将桶中元素拿出
        li.clear()
        for buc in buckets:
            li.extend(buc)
            buc.clear() # 拿出后记得清空桶
```

```
import random

li = [random.randint(0,200) for _ in range(15)]
print(li)

radix_sort(li)
print(li)
```

```
[113, 16, 180, 89, 76, 117, 86, 135, 63, 87, 174, 110, 181, 16, 73]
[16, 16, 63, 73, 76, 86, 87, 89, 110, 113, 117, 135, 174, 180, 181]
```

基数排序性能

1. 时间复杂度: $O(kn)$, 其中k表示最大元素的位数;
2. 空间复杂度: $O(k + n)$

与NB三人组比较:

$k = \text{math.floor}(\text{math.log}(10, \text{length}))$ 以10为底, length为数字的位数;
 $n \log n$, 其中 $\log n = \text{math.log}(2, n)$ 以2为底, n为元素个数

所以, 如果元素位数小, 个数多, 基数排序要比NB三人组快。

3. 数据结构

- 3.1 列表
- 3.2 栈
- 3.3 队列
- 3.4 链表
- 3.5 哈希表
- 3.6 树

3.1 数据结构介绍

数据结构按照其逻辑结构可分为线性结构、树结构、图结构

- 线性结构: 数据结构中的元素存在一对一的相互关系;
- 树结构: 数据结构中的元素存在一对多的关系;
- 图结构: 数据结构中的元素存在多对多的关系。

3.2 列表

列表 (其它语言称数组) 是一种基本数据类型。

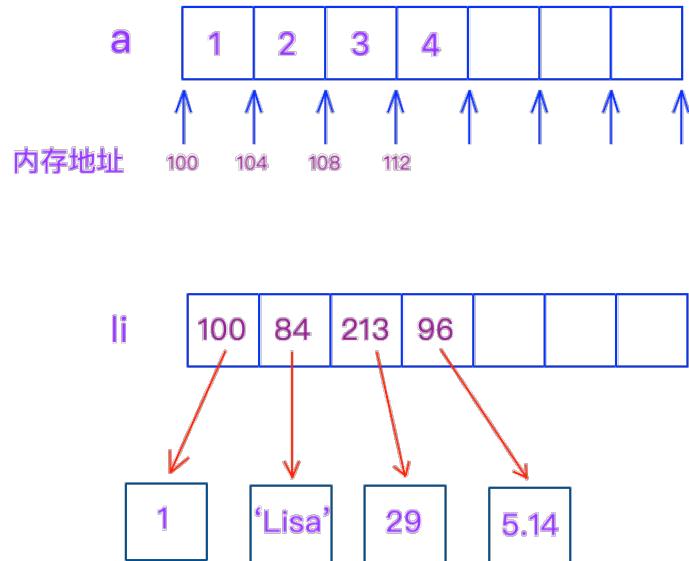
关于列表的问题:

- 列表中的元素是如何存储的?
- 列表的基本操作: 按下标查找、插入元素、删除元素...
- 这些操作的时间复杂度是多少?
- Python的列表是如何实现的?

其它语言中的数组与列表的两点不同:

1. 数组中的元素类型必须相同, 而Python中列表元素类型无限制;
2. 创建数组时需指定长度, 而Python列表无需指定长度。

32位机器，一个整数占4个字节，一个地址占6个字节



1. 数组连续内存空间中存的是元素，查找元素时通过计算地址值直接得到元素，因此元素类型需相同（即所占内存空间长度相同）；
2. 列表连续内存空间中存的是元素的地址值，因而元素类型可以不相同；
3. 按下标查找元素操作是 $O(1)$;
4. 插入和删除元素操作是 $O(n)$ ，因为涉及挪动元素。

3.3 栈

栈 (Stack) 是一个数据集合，可以理解为只能在一端进行插入或删除操作的列表。

栈的特点：后进先出LIFO (Last In, First Out)

栈的概念：栈顶、栈底

栈的基本操作：

- 进栈 (压栈) : push
- 出栈: pop
- 取栈顶: gettop (只看不取)

```
class Stack():
    def __init__(self):
        self.stack = []

    def push(self, element):
        self.stack.append(element)

    def pop(self):
        return self.stack.pop()

    def get_top(self):
        if len(self.stack) > 0:
```

```

        return self.stack[-1]
    else:
        return None

def is_empty(self):
    return len(self.stack) == 0

```

```

stack = Stack()
stack.push(1)
stack.push(1)
stack.push('good')
stack.push(9)

print(stack.get_top())
print(stack.pop())
print(stack.get_top())

```

```

9
9
good

```

括号匹配问题：

1. (){}[] 匹配
2. [()] 不匹配
3. {}() 不匹配

```

def brace_match(string):
    ...

用栈来解决括号匹配问题
    ...

braces = {'}': '{', ']': '[', ')': '(', '>': '<'}
stack = Stack()
for s in string:
    top = stack.get_top()
    if s in braces.values(): # 如果是左括号，直接放入栈顶
        stack.push(s)
        continue
    elif stack.is_empty(): # 如果不是左括号，且栈为空，返回False
        return False
    elif top == braces[s]: # 如果匹配，则弹出
        stack.pop()
    else:
        return False
if stack.is_empty():
    return True
else:
    return False

```

```

s1 = '{}()'
s2 = '[]]'
s3 = '{()}'
s4 = '{{{{}}}}'
print(brace_match(s1))
print(brace_match(s2))
print(brace_match(s3))
print(brace_match(s4))

```

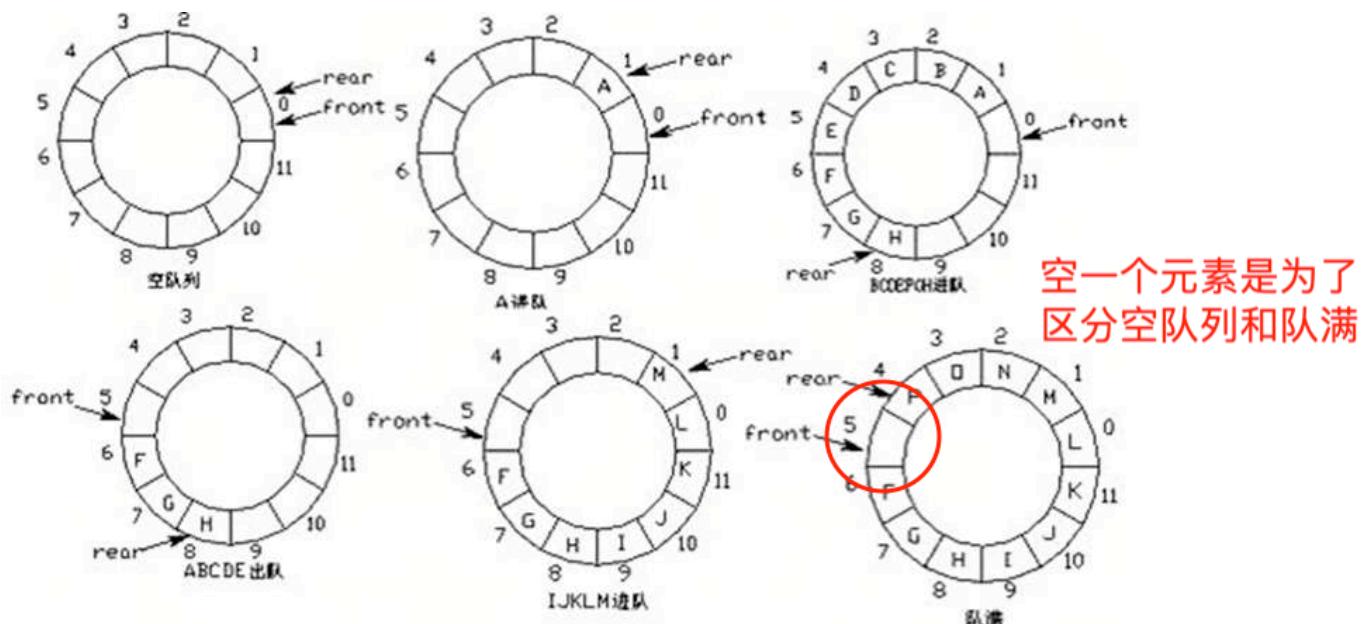
```

True
False
False
True

```

3.4 队列

- 队列 (Queue) 是一个数据集合，仅允许在列表的一端进行插入，另一端进行删除；
- 进行插入的一端称为队尾 (rear)，插入动作称为进队或入队；
- 进行删除的一端称为对头 (front)，删除操作称为出队；
- 队列的性质：先进先出FIFO (First-in, First-out)



环形队列：当队尾指针 $rear == \text{Maxsize} - 1$ 时，再前进一个位置就自动到0.

- 队首指针前进1: $\text{front} = (\text{front} + 1) \% \text{Maxsize}$
- 队尾指针前进1: $\text{rear} = (\text{rear} + 1) \% \text{Maxsize}$
- 队空条件: $\text{rear} == \text{front}$
- 队满条件: $(\text{rear} + 1) \% \text{Maxsize} == \text{front}$

```

class Queue():
    def __init__(self, size=100):
        self.queue = [0 for _ in range(size)]

```

```

    self.size = size
    self.rear = 0 # 队尾
    self.front = 0 # 队首

def push(self, element):
    if (self.rear + 1) % self.size == self.front:
        raise Exception('队已满')
    self.rear = (self.rear + 1) % self.size
    self.queue[self.rear] = element

def pop(self):
    if self.rear == self.front:
        return None
    self.front = (self.front + 1) % self.size
    return self.queue[self.front]

def is_empty(self):
    return self.rear == self.front

def is_full(self):
    return (self.rear + 1) % self.size == self.front

```

```

queue = Queue(12)
queue.push(0)
queue.push(1)
queue.push(2)
queue.push(3)
queue.push(4)
queue.push(5)
queue.push(6)
queue.push(7)
queue.push(8)
queue.push(9)
queue.push(10)
# queue.push(11)
# queue.push(12)

print(queue.pop())
print(queue.pop())
print(queue.is_empty())

```

```

0
1
False

```

双向队列：两端都支持进队和出队操作

Python内置队列模块

```
from collections import deque
```

- 创建队列: queue = deque()
- 进队: append()
- 出队: popleft()
- 双向队列首进队: appendleft()
- 双向队列尾出队: pop()

```
from collections import deque

# 同时支持单向和双向队列
queue = deque()

queue.append(12) # 队尾进队
queue.append(1024) # 队尾进队
print(queue.popleft()) # 队首出队

# 双向队列操作
queue.appendleft(99) # 队首进队
print(queue.pop()) # 队尾出队

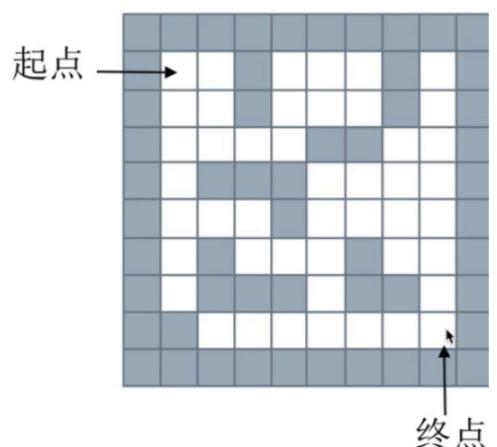
# 当队列满了后, deque会自动将队首元素出队
queue2 = deque([1,2,3,4,5,6,7,8,9], 5)
print(queue2.popleft())
```

```
12
1024
5
```

迷宫问题:

给一个二维列表，表示迷宫（0表示通道，1表示围墙）。给出算法，求一条走出迷宫的路径。

```
maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,1,1,0,0,1],
    [1,0,1,1,1,0,0,0,0,1],
    [1,0,0,0,1,0,0,0,0,1],
    [1,0,1,0,0,1,0,0,1],
    [1,0,1,1,1,0,1,1,0,1],
    [1,1,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1]
]
```



栈——深度优先搜索算法

1. 又叫回溯法
2. 思路：从一个节点开始，任意找下一个能走的点，当找不到能走的点时，退回上一个点寻找是否有其它方向的点。
3. 使用栈存储当前路径。

```

maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,1,1,0,0,1],
    [1,0,1,1,1,0,0,0,0,1],
    [1,0,0,0,1,0,0,0,0,1],
    [1,0,1,0,0,0,1,0,0,1],
    [1,0,1,1,1,0,1,1,0,1],
    [1,1,0,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1],
]

def path_search(x1,y1,x2,y2):
    ...
    x1,y1: 起始点坐标
    x2,y2: 终点坐标
    ...

    stack = []
    stack.append((x1,y1))

    px = x1
    py = y1

    while px != x2 or py != y2:
        # 按 上—>右—>下—>左 的步骤搜索
        if maze[px-1][py] != 1 and (px-1, py) not in stack:
            px -= 1
            stack.append((px, py))
        elif maze[px][py+1] != 1 and (px, py+1) not in stack:
            py += 1
            stack.append((px, py))
        elif maze[px+1][py] != 1 and (px+1, py) not in stack:
            px += 1
            stack.append((px, py))
        elif maze[px][py-1] != 1 and (px, py-1) not in stack:
            py -= 1
            stack.append((px, py))
        else:
            maze[px][py] = 1
            stack.pop()
            px, py = stack[-1]
    if px == x1 and py == y1:
        raise Exception('No path available! ')

```

```
return stack
```

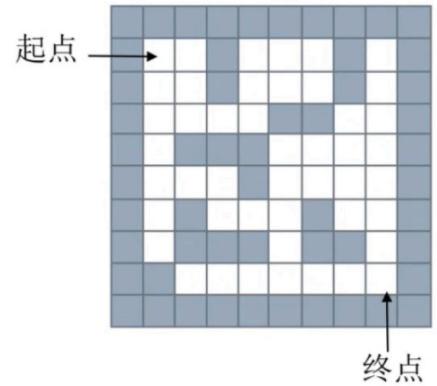
```
maze = [  
    [1,1,1,1,1,1,1,1,1,1],  
    [1,0,0,1,0,0,0,1,0,1],  
    [1,0,0,1,0,0,0,1,0,1],  
    [1,0,0,0,0,1,1,0,0,1],  
    [1,0,1,1,1,0,0,0,0,1],  
    [1,0,0,0,1,0,0,0,0,1],  
    [1,0,1,0,0,0,1,0,0,1],  
    [1,0,1,1,1,0,1,1,0,1],  
    [1,1,0,0,0,0,0,0,0,1],  
    [1,1,1,1,1,1,1,1,1,1],  
]  
  
print(path_search(1,1,8,8))
```

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

队列——广度优先搜索

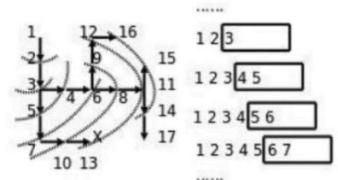
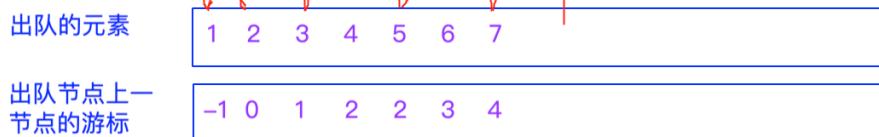
广度优先搜索的路径是最短的。

队列——广度优先搜索



▶ 思路：从一个节点开始，寻找**所有接**下来能继续走的点，继续不断寻找，直到找到出口。

▶ 使用队列存储当前正在考虑的节点



```
from collections import deque
```

```
maze = [  
    [1,1,1,1,1,1,1,1,1,1],  
]
```

```

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

dirs = [
    lambda x,y:(x-1, y), # 上
    lambda x,y:(x, y+1), # 右
    lambda x,y:(x+1, y), # 下
    lambda x,y:(x, y-1) # 左
]

def output_path(path):
    ...
    输出路径
    ...
    node = path[-1] # 取出终点
    out_path = [ ]

    while node[2] != -1:
        out_path.append(node[0:2])
        node = path[node[2]]

    out_path.append(path[0][0:2])

    out_path.reverse()

    return out_path

def maze_path(x1,y1,x2,y2):
    ...
    x1,y1: 起始点坐标
    x2,y2: 终点坐标
    ...

    queue = deque() # 记录搜索到的最新结点
    queue.append((x1, y1, -1)) # 把起点加入队列，第3个元素表示该节点的上一节点在path中的索引
    path = [] # 记录历史路径的列表

    while len(queue) != 0: # 只要队列里还有值，就代表还有路可走
        cur_node = queue.popleft() # 取出当前节点

```

```

path.append(cur_node) # 将当前节点放入历史路径列表中
if cur_node[0] == x2 and cur_node[1] == y2: # 表示已走到终点
    output = output_path(path)
    return output
for next_node in dirs:
    next_x, next_y = next_node(cur_node[0], cur_node[1])
    if maze[next_x][next_y] == 0: # 路是通的
        # 将next_node节点入队
        queue.append((next_x, next_y, len(path)-1))
        # 将next_node标记为已走过
        maze[next_x][next_y] = 2
else:
    raise Exception('无路可走了!')

```

```

path = maze_path(1,1,8,8)
for p in path:
    print(p)

```

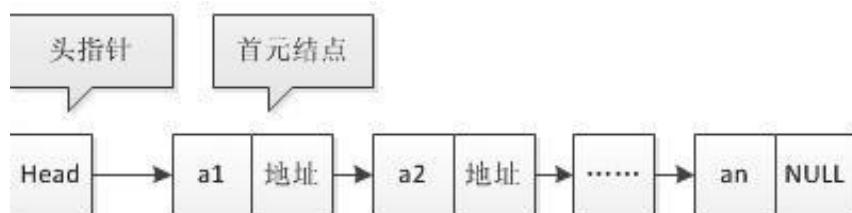
```

(1, 1)
(2, 1)
(3, 1)
(4, 1)
(5, 1)
(5, 2)
(5, 3)
(6, 3)
(6, 4)
(6, 5)
(7, 5)
(8, 5)
(8, 6)
(8, 7)
(8, 8)

```

3.5 链表

链表是由一系列节点组成的元素的集合。每个节点包含两部分，数据域item和指向下一节点的指针next。通过节点之间的相互连接，最终串成一个链表。



```
class Node():
    def __init__(self, item):
        self.item = item
        self.next = None
```

```
node1 = Node(11)
node2 = Node(22)
node3 = Node(33)

node1.next = node2
node2.next = node3

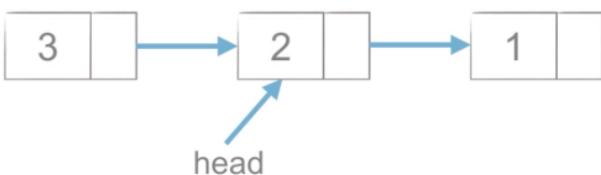
print(node1.item)
print(node1.next.item)
print(node1.next.next.item)
```

```
11
22
33
```

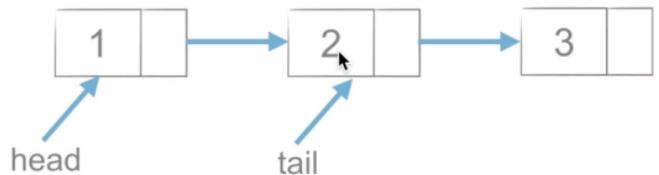
3.5.1 链表的创建和遍历

头插法和尾插法：

头插法



尾插法



```
def creat_linklist_head(li):
    ...
    头插法创建链表,
    遍历链表得到的是倒序的
    ...
    head = Node(li[0])

    for ele in li[1:]:
        node = Node(ele)
        node.next = head
        head = node

    return head

def creat_linklist_tail(li):
    ...
```

```

尾插法创建链表
遍历链表得到的是倒序的
...
head = Node(li[0])
tail = head

for ele in li[1:]:
    node = Node(ele)
    tail.next = node
    tail = node

return head

```

```

def print_linklist(head):
    ...
遍历打印链表
...
while head and head.next:
    print(head.item, end = ', ')
    head = head.next
else:
    print(head.item)

```

```

lk = creat_linklist_head([1,2,3,4,5,6])
print_linklist(lk)

lk = creat_linklist_tail([1,2,3,4,5,6])
print_linklist(lk)

```

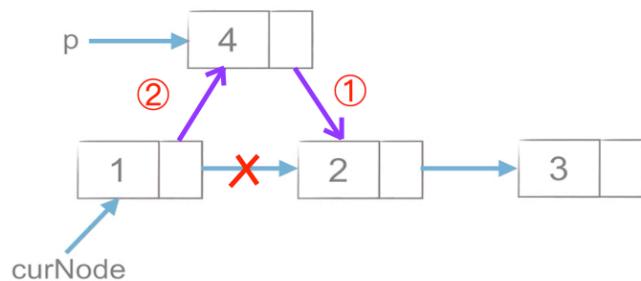
```

6, 5, 4, 3, 2, 1
1, 2, 3, 4, 5, 6

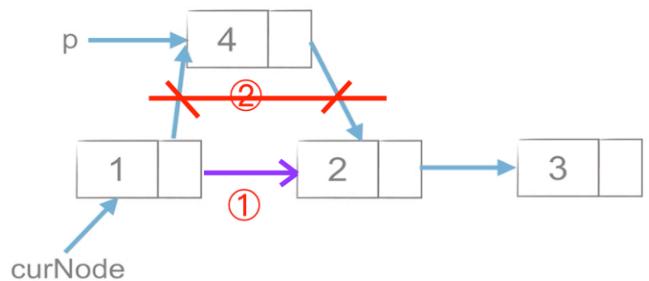
```

3.5.2 链表的插入和删除

插入



删除



```

def insert_node(cur_node, new_node):
    ...
在cur_node之后插入新节点new_node

```

```
    ...
    new_node.next = cur_node.next
    cur_node.next = new_node
```

```
def del_node(cur_node):
    ...
    将cur_node的后一节点从链表中删除
    ...
    p = cur_node.next
```

```
    cur_node.next = p.next
    del p
```

```
lk = creat_linklist_tail([1,2,3,4,5,6,7])
print_linklist(lk)
```

```
cur_node = lk.next
```

```
new_node = Node(88)
insert_node(cur_node, new_node)
print_linklist(lk)
```

```
del_node(cur_node)
print_linklist(lk)
```

```
1, 2, 3, 4, 5, 6, 7
1, 2, 88, 3, 4, 5, 6, 7
1, 2, 3, 4, 5, 6, 7
```

3.5.3 链表总结

操作	顺序表	链表
按元素查找	$O(n)$	$O(n)$
按下标查找	$O(1)$	$O(n)$
在某元素后插入	$O(n)$	$O(1)$
删除某元素	$O(n)$	$O(1)$

1. 链表的插入和删除操作明显优于顺序表；
2. 链表的内存空间可以更灵活的分配。

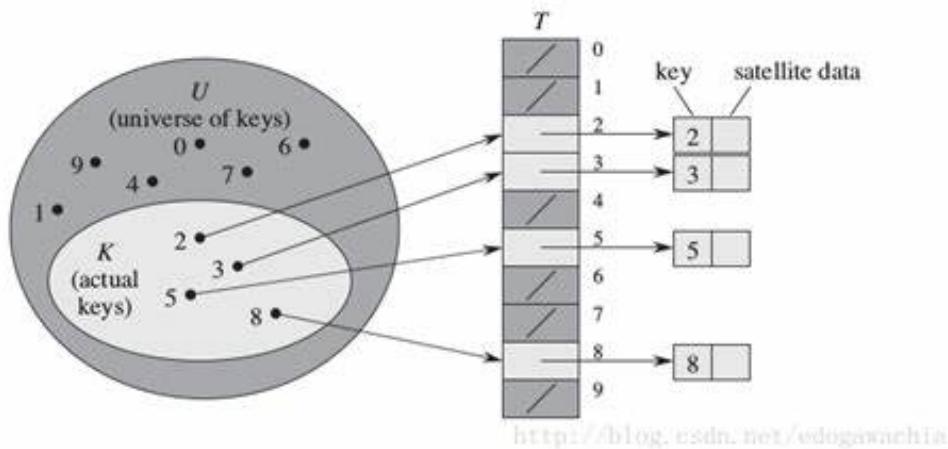
3.6 哈希表（散列表）

哈希表是一个通过哈希函数来计算数据存储位置的数据结构，通常支持如下操作：

- insert(key, value)
- get(key)
- delete(key)

直接寻址表：

U是所有可能key的集合



直接寻址技术缺点：

1. 当域 U 很大时，创建 T 列表需要大量内存，很不实际；
2. 如果 U 很大，而实际 K 很少，则大量空间被浪费；
3. 无法处理关键字不是数字的情况。

改进直接寻址表：哈希 (Hashing)

- 构建大小为 m 的寻址表 T ；
- key为 k 的元素放到 $h(k)$ 位置上；
- $h(k)$ 是一个函数，其将域 U 映射到表 $T[0, 1, \dots, m-1]$

哈希表

1. 哈希表（Hash Table, 又称散列表），是一种线性表的存储结构。哈希表由一个直接寻址表和一个哈希函数组成。哈希函数 $h(k)$ 将关键字 k 作为自变量，返回元素的存储下标。
2. 假如有一个长度为7的哈希表，哈希函数 $h(k) = k \% 7$ 。元素集合{14, 22, 3, 5}的存储方式如下图：



哈希冲突

1. 哈希表的大小是有限的，而要存的值的总数量是无限的，因此对于任何哈希函数，都会出现两个不同元素映射到同一位置的情况，称为哈希冲突；
2. 比如 $h(k)=k \% 7$, $h(0)=h(7)=h(14)=\dots$

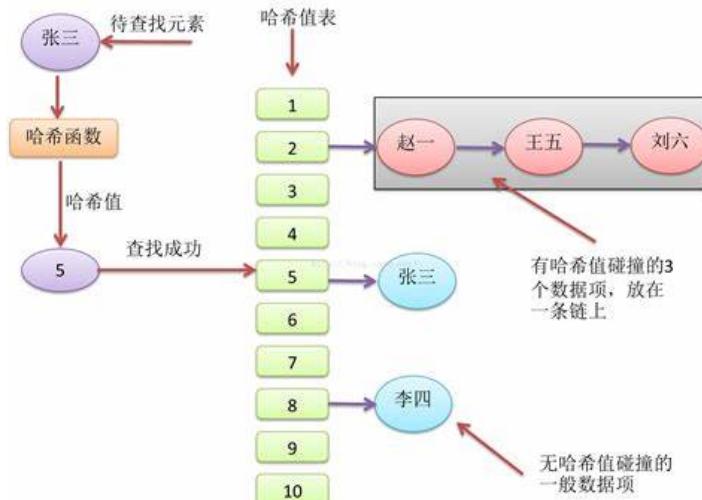
解决哈希冲突——开放寻址法

如果哈希函数返回的位置已经有值，则可以向后探查新的位置来存储这个值。

- 线性探查：如果位置 i 被占用，则依次向后探查 $i+1, i+2, \dots$ 直到找到空位，进行存储；
- 二次探查：如果位置 i 被占用，则探查 $i+1^2, i-1^2, i+2^2, i-2^2, \dots$
- 二度哈希：有 n 个哈希函数，当使用第1个哈希函数 h_1 发生冲突时，则尝试使用 h_2, h_3, \dots

解决哈希冲突——拉链法

哈希表每个位置都连接一个链表，当冲突发生时，冲突元素将被加到该位置链表的最后。



```
class Linklist():
    class Node():
        def __init__(self, item):
            self.item = item
            self.next = None

    class LinklistIterator():
        def __init__(self, node):
            self.node = node

        def __next__(self):
            if self.node:
                cur_node = self.node
                self.node = cur_node.next
                return cur_node.item
            else:
                raise StopIteration
        def __iter__(self):
            return self

    def __init__(self, iterable=None):
        self.head = None
        self.tail = None
        if iterable:
            self.extend(iterable)
```

```

def extend(self, iterable):
    for obj in iterable:
        self.append(obj)

def append(self, obj):
    node = Linklist.Node(obj)
    if not self.head: # 链表为空
        self.head = node
        self.tail = node
    else: # 追加节点
        self.tail.next = node
        self.tail = node

def find(self, obj):
    for n in self:
        if n == obj:
            return True
    else:
        return False

def __iter__(self):
    return self.LinklistIterator(self.head)

def __repr__(self):
    return "<" + ", ".join(map(str, self)) + ">"

```

```

lk = Linklist([1, 2, 3, 4, 5])
print(lk)

```

```

<1, 2, 3, 4, 5>

```

```

class HashTable():

    def __init__(self, size=101):
        self.size = size
        self.T = [Linklist() for _ in range(size)]

    def h(self, k):
        return k % self.size

    def insert(self, k):
        h = self.h(k)
        # 不能重复插入
        if self.find(k):

```

```
        raise Exception('不能重复插入')

    else:
        self.T[h].append(k)

def find(self, k):
    h = self.h(k)
    return self.T[h].find(k)
```

```
table = HashTable()
```

```
table.insert(0)
table.insert(101)
table.insert(4)
# table.insert(4)
```

```
print(table.T)
```

```
print(table.find(0))  
print(table.find(1))
```

哈希表的应用——md5算法

MD5(Message-Digest Algorithm 5)曾经是密码学中常用的哈希函数，可以把任意长度的数据映射为128位的哈希值，其曾经包含如下特征：

- 同样的消息，其MD5值必定相同；
 - 可以快速计算出任意给定消息的MD5值；
 - 除非暴力枚举，否则不能根据哈希值反推出消息本身；
 - 两条消息之间即使只有微小的差别，其对应的MD5值也应是完全不同，完全不相关的；
 - 不能在有意义的时间内人工构造两个不同的消息，使得其具有相同的MD5值。

应用举例: 文件的哈希值

算出两个文件的哈希值，若两个文件的哈希值相同，则可认为这两个文件是相同的（哈希值相同，文件不同的概率太太太低），因此：

- 用户可以利用它来验证下载的文件是否完整；
 - 云存储服务商可以利用它来判断用户要上传的文件是否已存在服务器上，从而实现秒传的功能，同时避免存储过多相同文件的副本。

哈希表的应用——SHA2算法

1. 历史上MD5和SHA-1曾经是使用最为广泛的cryptographic hash function, 但是随着密码学的发展, 这两个

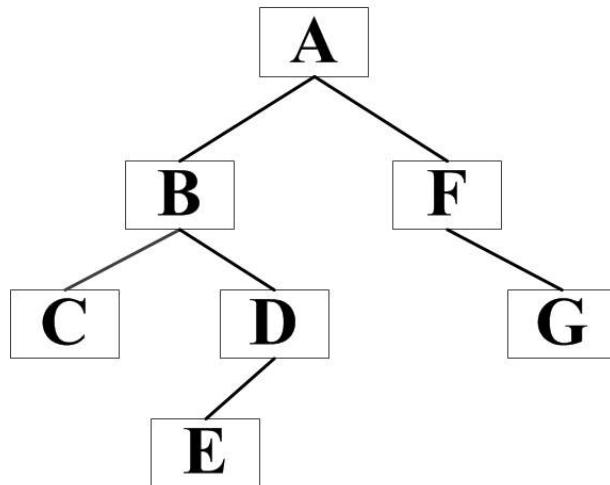
- 哈希函数的安全性相继受到了各种挑战。
2. 因此现在安全性较重要的场合推荐使用SHA-2等新的更安全的哈希函数；
 3. SHA-2包含了一系列的哈希函数：SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, 其对应的哈希值长度分别为224, 256, 384或512位。
 4. SHA-2具有和MD5类似的性质。

3.7 树

3.7.1 二叉树

二叉树的链式存储：将二叉树的节点定义为一个对象，节点之间通过类似链表的连接方式来连接。

因为二叉树可能不是完全二叉树，因此用列表存储不方便



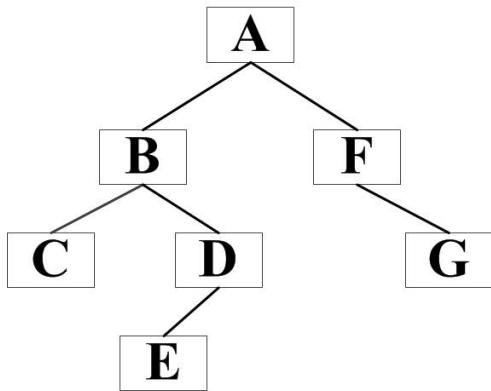
```
class BiTreeNode():
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None

a = BiTreeNode('A')
b = BiTreeNode('B')
c = BiTreeNode('C')
d = BiTreeNode('D')
e = BiTreeNode('E')
f = BiTreeNode('F')
g = BiTreeNode('G')

a.lchild = b
a.rchild = f
b.lchild = c
b.rchild = d
f.rchild = g
d.lchild = e

print(a.lchild.rchild.data)
```

3.7.2 二叉树的遍历



二叉树的遍历方式：

- 前序遍历：ABCDEF根-左-右
- 中序遍历：CBEDAFG 左-根-右
- 后续遍历：CEDBGFA 左-右-根
- 层次遍历：ABFCDEG

```

def pre_order(root):
    ...
    前序遍历
    ...
    if root:
        print(root.data, end=', ')
        pre_order(root.lchild)
        pre_order(root.rchild)

def in_order(root):
    ...
    中序遍历
    ...
    if root:
        in_order(root.lchild)
        print(root.data, end=', ')
        in_order(root.rchild)

def post_order(root):
    ...
    后序遍历
    ...
    if root:
        post_order(root.lchild)
        post_order(root.rchild)
        print(root.data, end=', ')
  
```

```

from collections import deque

def level_order(root):
    ...
    层次遍历
    ...
    queue = deque()
    queue.append(root)

    while len(queue) > 0:
        cur_node = queue.popleft()
        print(cur_node.data, end=', ')
        if cur_node.lchild:
            queue.append(cur_node.lchild)
        if cur_node.rchild:
            queue.append(cur_node.rchild)

```

```

pre_order(a)
print()
in_order(a)
print()
post_order(a)
print()
level_order(a)

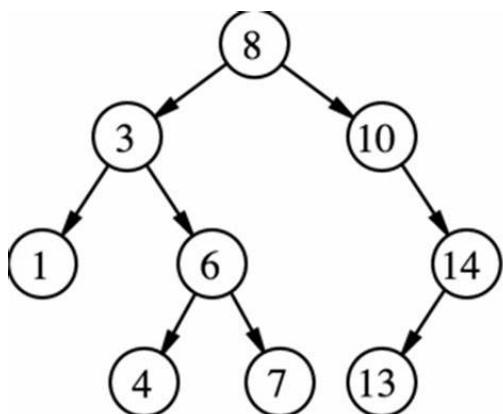
```

A, B, C, D, E, F, G,
C, B, E, D, A, F, G,
C, E, D, B, G, F, A,
A, B, F, C, D, G, E,

3.7.3 二叉搜索树

二叉搜索树时一个二叉树且满足性质：

设X是二叉树的一个节点。
如果Y是X的左子树上的一个节点，那么 $Y.key \leq X.key$ ；
如果Y是X的右子树上的一个节点，那么 $Y.key \geq X.key$ ；



```

class BiTreeNode():

    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None
        self.parent = None

class BST():

    ...

    二叉搜索树
    ...

    def __init__(self, iterable):
        self.root = None
        if iterable:
            for val in iterable:
                self.insert_no_cur(val)

    # 递归方式插入(不太好理解)
    def insert(self, node, val):
        if not node: # 比如对于上图, 要插入0, 1的左节点为空, 需创建一个node
            node = BiTreeNode(val)
        elif val < node.data:
            node.lchild = insert(node.lchild, val)
            node.lchild.parent = node
        elif val > node.data:
            node.rchild = insert(node.rchild, val)
            node.rchild.parent = node
        return node

    # 不用递归实现插入
    def insert_no_cur(self, val):
        if not self.root:
            # 空二叉搜索树, 则初始化根节点
            self.root = BiTreeNode(val)
            return
        node = self.root
        while True:
            if val < node.data:
                if not node.lchild: # 左子节点为空
                    child_node = BiTreeNode(val)
                    node.lchild = child_node
                    child_node.parent = node
                    return
                else:
                    node = node.lchild
                    continue
            elif val > node.data:

```

```

        if not node.rchild:
            child_node = BiTreeNode(val)
            node.rchild = child_node
            child_node.parent = node
            return
        else:
            node = node.rchild
            continue
    else:
        return

def __query(self, node, val):
    ...
    递归查询
    ...
    if not node:
        return None
    elif val == node.data:
        return node
    elif val < node.data:
        return self.__query(node.lchild, val)
    elif val > node.data:
        return self.__query(node.rchild, val)

def query(self, val):
    ...
    递归查询
    ...
    return self.__query(self.root, val)

def __remove_leaf(self, node):
    ...
    情况一：删除叶子节点
    ...
    if not node.parent: # 根节点
        self.root = None
    elif node == node.parent.lchild: #
        node.parent.lchild = None
    elif node == node.parent.rchild:
        node.parent.rchild = None

def __remove_node_with_single_child(self, node):
    ...
    情况二：要删除的节点只有一个孩子节点
    ...
    if not node.parent: # 要删除的节点是根节点
        if node.lchild:
            self.root = node.lchild
        elif node.rchild:

```

```

        self.root = node.rchild
        self.root.parent = None
    elif node == node.parent.lchild:
        if node.lchild:
            node.parent.lchild = node.lchild
            node.lchild.parent = node.parent
        else:
            node.parent.lchild = node.rchild
            node.rchild.parent = node.parent
    elif node == node.parent.rchild:
        if node.lchild:
            node.parent.rchild = node.lchild
            node.lchild.parent = node.parent
        else:
            node.parent.rchild = node.rchild
            node.rchild.parent = node.parent

def remove(self, val):
    """
    删除节点
    """

    if not self.root: # 代表空树
        return

    node = self.query(val) # 查询要删除的节点
    if not node: # 要删除的节点不存在
        raise Exception('Error key!')

    # 情况1: node是叶子节点
    if not node.lchild and not node.rchild:
        self.__remove_leaf(node)
    # 情况2: node节点只有一个孩子
    elif (node.lchild and not node.rchild) or (not node.lchild and node.rchild):
        self.__remove_node_with_single_child(node)

    # 情况3: node有两个子树
    else:
        # 找该节点右子树最小节点（一直往左找）
        node_tmp = node.rchild
        while node_tmp.lchild:
            node_tmp = node_tmp.lchild

        # 数据替换到要删除的节点
        node.data = node_tmp.data

        # 将右子树最小节点删除
        if node_tmp.rchild:
            self.__remove_node_with_single_child(node_tmp)

```

```

        else:
            self.__remove_leaf(node_tmp)

    def in_order(self, root):
        ...
        中序遍历
        ...
        if root:
            self.in_order(root.lchild)
            print(root.data, end=' ')
            self.in_order(root.rchild)

```

```

import random

bst = BST([8, 3, 10, 1, 6, 4, 7, 13, 14, ])
print('root =', bst.root.data)

# 中序遍历
bst.in_order(bst.root)
print()

# 查询
print(bst.query(11), bst.query(5), bst.query(13).data)
print()

# 删除节点
# bst.remove(1)
bst.remove(6)
bst.in_order(bst.root)

```

```

root = 8
1 3 4 6 7 8 10 13 14
None None 13

1 3 4 7 8 10 13 14

```

二叉搜索树节点删除：

1. 若是叶子节点，直接删除；
2. 若要删除的节点只有一个孩子，则将它的parent和child直接相连；
3. 若要删除的节点是根节点，则将右子树最小节点替换到该位置上。

二叉搜索树的效率：

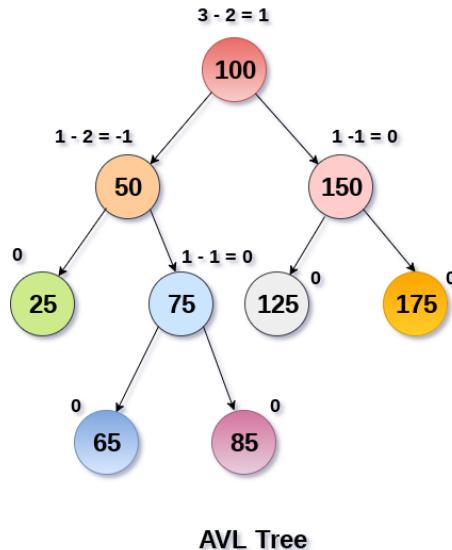
1. 平均情况下，二叉搜索树进行搜索的时间复杂度为 $O(\log n)$ ；
2. **最坏情况下，二叉搜索树可能非常偏斜，这样时间复杂度接近线性；**
3. 接近方案：

- 随机化插入
- AVL树

3.7.4 AVL树

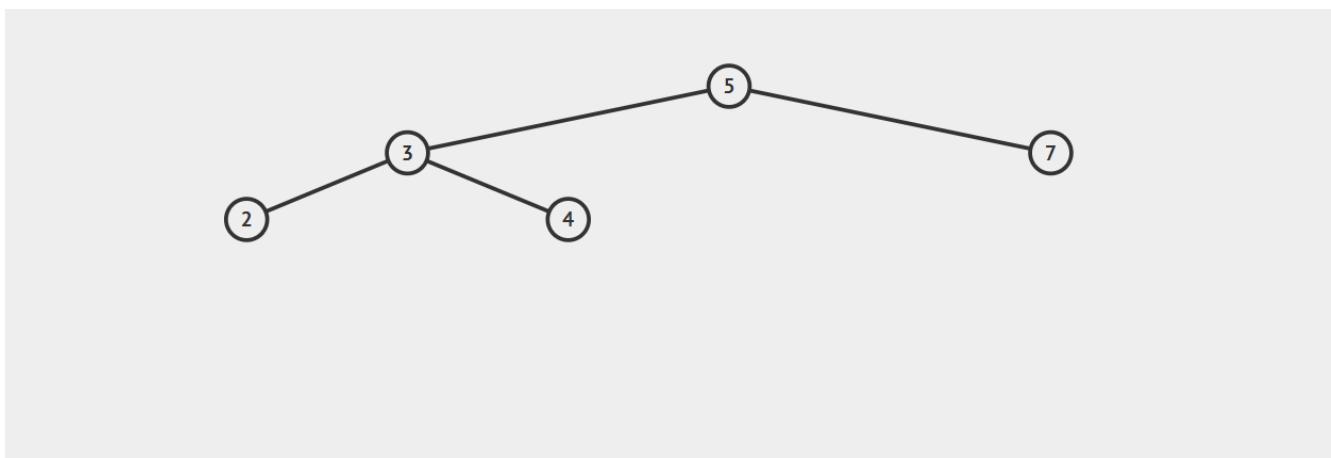
AVL树：是一棵自平衡的二叉搜索树。

1. 根的左右子树的高度差的绝对值不能超过1；
2. 根的左右子树都是平衡二叉树。

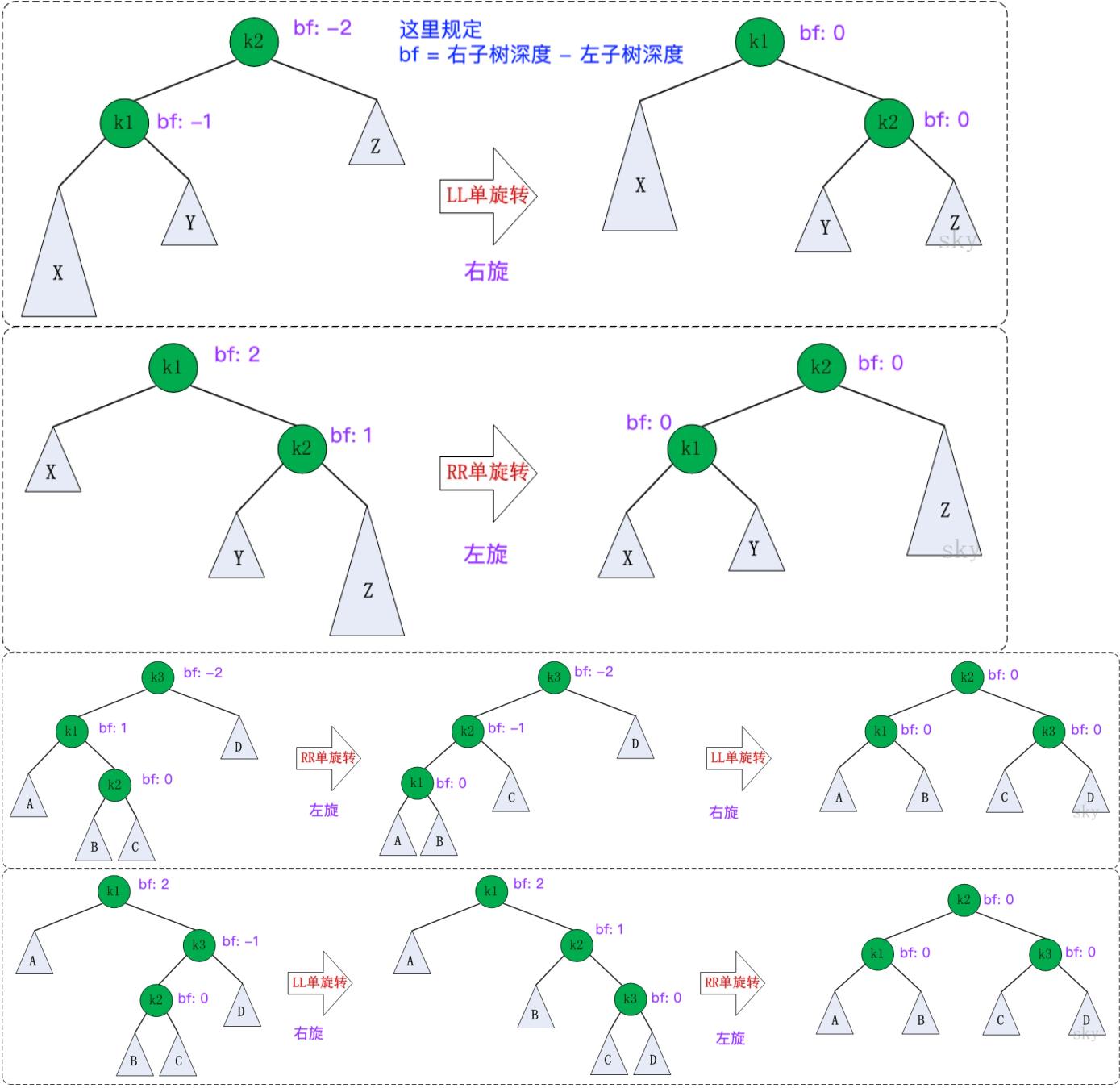


AVL树插入：

1. 插入一个节点可能会破坏AVL树的平衡，可以通过**旋转**操作来进行修正。
2. 插入一个节点后，只有从插入节点到根节点的路径上的节点的平衡可能被改变。我们需要找出第一个平衡条件被破坏的节点，称之为K。K的两棵子树高度差为2.



3. 不平衡的出现可能有4中情况：



```
class AVLNode(BiTreeNode):
    def __init__(self, data):
        BiTreeNode.__init__(self, data)
        self.bf = 0 # 平衡因子

class AVLTree(BST):
    def __init__(self, li=None):
        BST.__init__(self, li)

    def rotate_left(self, k1, k2):
        ...
        k1
        k2
        n1
        n3
```

```

        n2      n3          n1      n2      new
        new

    ...
    k1.lchild = k2.lchild
    if k2.lchild:
        k2.lchild.parent = k1

    k2.lchild = k1
    k1.parent = k2

    # 更新bf (balance factor)
    k1.bf = 0
    k2.bf = 0
    return k2

def rotate_right(self, k2, k1):
    ...
        k2
        k1      n1      n2      k1
        n2      n3      new      n3      n1
        new
    ...
    k2.lchild = k1.rchild
    if k1.rchild:
        k1.rchild.parent = k2

    k2.parent = k1
    k1.rchild = k2

    # 更新bf
    k1.bf = 0
    k2.bf = 0
    return k1

def rotate_left_right(self, k3, k1):
    ...
        k3      |      k3      |      k2
        k1      D      k2      D      k1      k3
        A       k2      |      k1      C      |      A       B       C       D
        B       C      |      A       B      |      |      |
    ...
    k2 = k1.rchild

    # 先对k1子树进行左旋转
    k2 = self.rotate_left(k1, k2)

    k3.lchild = k2
    k2.parent = k3

```

```

# 再对k3树进行右旋转
k3 = self.rotate_right(k3, k2)

# 更新bf
# case1: k2节点后插入B
if k2.lchild:
    k1.bf = 0
    k3.bf = 1
elif k2.rchild:
    k1.bf = -1
    k3.bf = 0
else: # 此时没有A、B、C、D节点，只是插入k2导致k3的平衡被破坏
    k1.bf = 0
    k3.bf = 0

return k3

```

```
def rotate_right_left(self, k1, k3):
    '''

    k1           |           k1           |           k2
    A             k3           |           A             k2           |           k1           k3
    |             |           |             |             |             |
    k2           D           |           B           k3           |           A           B           C
    |             |           |             |             |             |
    B             C           |           C           D           |           |
    |             |           |             |
    ...           ...         |           ...         |           ...         |

    k2 = k3.lchild

```

```

# 先对k3子树进行右旋转
k2 = self.rotate_right(k3, k2)

k1.rchild = k2
k2.parent = k1

# 再对k1树进行左旋转
k3 = self.rotate_left(k1, k2)

# 更新bf
if k2.lchild: # case1: k2节点后插入的是B
    k1.bf = 0
    k3.bf = 1
elif k2.rchild: # case2: k2节点后插入的是C
    k1.bf = -1
    k3.bf = 0
else: # case3: 此时没有A、B、C、D节点，只是插入k2导致k3的平衡被破坏
    k1.bf = 0

```

```

k3.bf = 0

return k2

def insert_no_cur(self, val):
    #=====1.插入val=====
    if not self.root:
        # 空二叉搜索树，则初始化根节点
        self.root = AVLNode(val)
        return
    node = self.root
    while True:
        if val < node.data:
            if not node.lchild: # 左子节点为空，则创建一个左子节点
                child_node = AVLNode(val)
                node.lchild = child_node
                child_node.parent = node
            else: # 若左子节点不为None，则将左子节点作为根，与val进行比较
                node = node.lchild
                continue
        elif val > node.data:
            if not node.rchild:
                child_node = AVLNode(val)
                node.rchild = child_node
                child_node.parent = node
            else:
                node = node.rchild
                continue
        else: # 若插入的值与已有节点相同，不做任何处理
            return

    #=====2.更新bf=====
    # node 为插入节点的父节点
    while child_node.parent: # 直到更新到根节点位置，也就是节点的parent为None为止
        if child_node == node.lchild: # 从node的左边插入child_node
            node.bf -= 1 # 更新后的node.bf要么等于0，要么等于-1
            if node.bf == 0: # 也就是新插入节点不会导致树的失衡
                break
            else: # 继续向上更新bf
                parent_node = node.parent
                if node == parent_node.lchild: # 说明新插入是left-left形式，要进行右旋
                    parent_node.bf -= 1 # 更新后的bf只可能是-2, -1, 0三种情况
                    if parent_node.bf == 0: # 平衡了，不用再往上更新
                        return
                elif parent_node.bf == -1: # 继续往上更新
                    child_node = node
                    node = node.parent
                    continue

```

```

        elif parent_node.bf == -2: # 进行右旋
            root_node = self.rotate_left(node, child_node)
        else: # 说明新插入是right-left, 要进行左-右旋转
            parent_node.bf += 1
            if parent_node.bf == 0:
                return
            elif parent_node.bf == 1:
                child_node = node
                node = node.parent
                continue
            elif parent_node.bf == 2:
                root_node = self.rotate_left_right(parent_node, node)
        else: # 从node的右边插入child_node
            node.bf += 1
            if node.bf == 0:
                break
            else: # 继续向上更新
                parent_node = node.parent
                if node == parent_node.node.lchild: # 说明新插入是right-left形式
                    parent_node.bf -= 1
                    if parent_node.bf == 0:
                        return
                    elif parent_node.bf == -1:
                        child_node = node
                        node = parent_node
                        continue
                    else: # 进行右-左旋转
                        root_node = self.rotate_right_left(parent, node)

#=====3. 连接旋转后的子树=====
up_node = parent_node.parent
if up_node:
    if parent_node == up_node.lchild:
        up_node.lchild = root_node
        root_node.parent = up_node
    else:
        up_node.rchild = root_node
        root_node.parent = up_node
else:
    self.root = root_node
    return

def in_order(self, root):
    ...
    中序遍历
    ...
    if root:
        self.in_order(root.lchild)
        print(root.data, end=' ')

```

```
self.in_order(root.rchild)
```

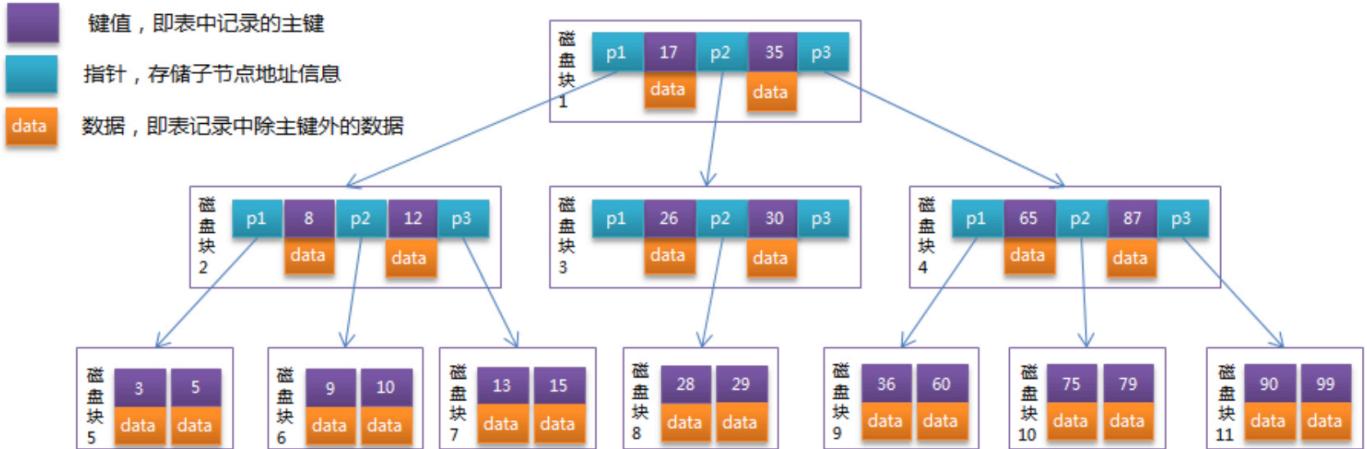
```
import random

tree = AVLTree(random.shuffle([i for i in range(1,11)]))
tree.in_order(tree.root)
print()
tree.insert_no_cur(11)
tree.in_order(tree.root)
```

11

3.7.5 二叉搜索树扩展应用——B树

B树（B-Tree）：B树是一棵自平衡的多路搜索树。通常用于数据库的索引。



因为数据库数据时分块存在硬盘上的，AVL树虽然查询效率也很高，但它是二路平衡树，树的高度为 $\log_2(n)$ ，而B树是多路平衡树，可以大大降低树的高度，加快搜索效率。

4. 贪心算法

1. 贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。
2. 贪心算法并不保证会得到最优解，但是在某些问题上贪心算法的解就是最优解。要会判断一个问题能否用贪心算法来计算。

4.1 找零问题

假设商店老板需要找零n元钱，钱币的面额有：100元、50元、20元、5元、1元，如何找零使得所需钱币的数量最少？

```

money = [100, 50, 20, 5, 1]

def change(li, n):
    num = [0 for _ in range(len(li))]

    for idx, m in enumerate(money):
        num[idx] = n // m
        n %= m
    return num

```

```
change(money, 321)
```

```
[3, 0, 1, 0, 1]
```

4.2 背包问题

一个小偷在某个商店发现有n个商品，第*i*个商品价值 v_i 元，重 w_i 千克。他希望拿走的价值尽量高，但他的背包最多只能容纳W千克的东西。他应该拿走哪些商品？

- **0-1背包：**对于一个商品，小偷要么把它完整拿走，那么留下。不能只拿走一部分，或把一个商品拿走多次。
(商品为金条)
- **分数背包：**对于一个商品，小偷可以拿走其中任意一部分。
(商品为金砂)

明显，贪心算法求解分数背包得到的是最优解。而0-1背包问题，则不一定。

```

...
商品1: v1=60, w1=10
商品2: v2=100, w2=20
商品3: v3=120, w3=30
背包容量: W=50
...

def fractional_backpack(goods, W):
    num = [0 for _ in range(len(goods))]
    total_value = 0
    for idx, (price, weight) in enumerate(goods):
        if W >= weight:
            num[idx] = 1
            total_value += price
            W -= weight
        else:
            num[idx] = W / weight
            total_value += price * W / weight
            break

```

```

    return num, total_value

goods = [(60, 10), (120, 30), (100, 20)]
goods.sort(key=lambda x:x[0]/x[1], reverse=True)

num, total_value = fractional_backpack(goods, 50)
print(num)
print(total_value)

num, total_value = fractional_backpack(goods, 100)
print(num)
print(total_value)

```

```

[1, 1, 0.6666666666666666]
240.0
[1, 1, 1]
280

```

4.3 数字拼接问题

有n个非负整数，将其按照字符串拼接的方式拼接为一个整数。如何拼接可以使得到的整数最大？例：

32, 94, 128, 1286, 6, 71可以拼接的最大整数为：94716321286128

```

def num_join(li):

    li = list(map(str, li))

    # 同冒泡排序
    for i in range(len(li)-1):
        for j in range(i+1, len(li)):
            if li[i] + li[j] < li[j] + li[i]:
                li[i], li[j] = li[j], li[i]

    return ''.join(li)

```

```

li = [32, 94, 128, 1286, 6, 71]
num = num_join(li)
print(num)

```

```
94716321286128
```

4.4 活动选择问题

假设有n个活动，这些活动要占用同一块场地，而场地在某时刻只能供一个活动使用。

每个活动都有一个开始时间 s_i 和结束时间 f_i （题目中时间以整数表示），表示活动在 $[s_i, f_i]$ 区间占用场地。

问：安排哪些活动能够使该场地举办的活动个数最多？

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

贪心结论：最先结束的活动一定是最优解的一部分

证明：假设a是所有活动中最先结束的活动，b是最优解中最先结束的活动。

- 如果 $a = b$, 结论成立；
- 如果 $a \neq b$, 则b的结束时间一定晚于a, 则此时用a替换掉b, a一定不与最优解中的其它活动时间重叠, 因此替换掉b后也是最优解。

```
activations = [(1,4),(3,5),(0,6),(5,7),(3,9),(5,9),(6,10),(8,11),(8,12),(2,14),(12,16)]
# 首先按活动结束时间进行排序
activations.sort(key=lambda x:x[1])
```

```
def act_selection(activations):
    act = [activations[0]]
    for i in range(1, len(activations)):
        # 若时间不重叠, 则加入
        if activations[i][0] >= act[-1][1]:
            act.append(activations[i])
    return act
```

```
print(act_selection(activations))
```

```
[(1, 4), (5, 7), (8, 11), (12, 16)]
```

5. 动态规划

```
...
递归实现斐波那契数列, 会出现子问题重复计算问题
...
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)

def fibonacci_no_cur(n):
```

```

val = [0, 1, 1]
if n > 2:
    for i in range(n-2):
        num = val[-1] + val[-2]
        val.append(num)
    return val[-1]
else:
    return val[n]

```

```

import time

start = time.time()
print(fibonacci(35))
print('递归实现: ', time.time() - start)

start = time.time()
print(fibonacci_no_cur(35))
print('非递归实现: ', time.time() - start)

```

9227465
 递归实现: 2.176862955093384
 9227465
 非递归实现: 6.580352783203125e-05

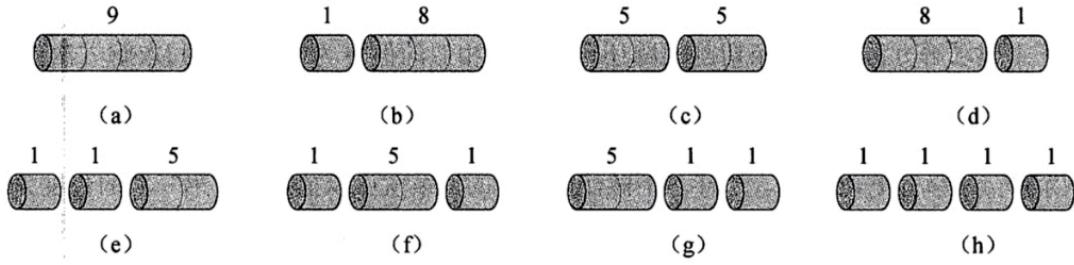
动态规划(DP) = 递推式 + 重复子问题

5.1 钢条切割问题

► 某公司出售钢条，出售价格与钢条长度之间的关系如下表：

长度 <i>i</i>	1	2	3	4	5	6	7	8	9	10
价格 <i>p_i</i>	1	5	8	9	10	17	17	20	24	30

► 问题：现有一段长度为n的钢条和上面的价格表，求切割钢条方案，使得总收益最大。



长度 <i>i</i>	1	2	3	4	5	6	7	8	9	10
价格 <i>p_i</i>	1	5	8	9	10	17	17	20	24	30

<i>i</i>	0	1	2	3	4	5	6	7	8	9	10
<i>r[i]</i>	0	1	5	8	10	13	17	18	22	25	30

钢条切割问题的递推式1：

- ▶ 设长度为n的钢条切割后最优收益值为 r_n , 可以得出递推式:
 - ▶ $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$
 - ▶ 第一个参数 p_n 表示不切割
 - ▶ 其他 $n-1$ 个参数分别表示另外 $n-1$ 种不同切割方案, 对方案 $i=1,2,\dots,n-1$
 - ▶ 将钢条切割为长度为*i*和 $n-i$ 两段
 - ▶ 方案*i*的收益为切割两段的最优收益之和
 - ▶ 考察所有的*i*, 选择其中收益最大的方案

最优子结构

1. 可以将求解规模为n的原问题, 划分为规模更小的子问题: 完成一次切割后, 可以将产生的两段钢条看成两个独立的钢条切割问题。
2. 组合两个子问题的最优解, 并在所有可能的两段切割方案中选取组合收益最大的, 构成原问题的最优解;
3. 钢条切割满足**最优子结构**: 问题的最优解由相关子问题的最优解组合而成, 这些子问题可以独立求解。

钢条切割问题的递推式2:

- ▶ 钢条切割问题还存在更简单的递归求解方法
 - ▶ 从钢条的左边切割下长度为*i*的一段, 只对右边剩下的一段继续进行切割, 左边的不再切割
 - ▶ 递推式简化为 $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$
 - ▶ 不做切割的方案就可以描述为: 左边一段长度为*n*, 收益为 p_n , 剩余一段长度为0, 收益为 $r_0=0$ 。

```

...
递归实现递推式1
...

def cut_rod_cur(p, n):
    if n == 0: # 长度为0
        return 0
    else:
        pn = p[n-1]
        for i in range(1, n):
            pn = max(pn, cut_rod_cur(p, i) + cut_rod_cur(p, n-i))
    return pn
...

```

递归实现递推式2

```

...
def cut_rod_cur2(p, n):
    if n == 0: # 长度为0
        return 0
    else:
        max_val = 0
        for i in range(0, n):
            max_val = max(max_val, p[i] + cut_rod_cur2(p, n-i-1))
    return max_val

```

```
p = [1,5,8,9,10,17,17,20,24,30,32,36,40,42,44,48,50]
```

```

import time

start = time.time()
print(cut_rod_cur(p,15))
print('递推公式1: ', time.time() - start)

start = time.time()
print(cut_rod_cur2(p,15))
print('递推公式2: ', time.time() - start)

```

```

45
递推公式1:  1.8961639404296875
45
递推公式2:  0.013830184936523438

```

自顶向下递归实现：

上面两个实现都是自顶向下的实现方式，效率很差，时间复杂度是 $O(2^n)$

动态规划的思想：

- 每个子问题只求解一遍，保存求解结果；

- 之后需要此问题，只需查找保存的结果。

```

'''  

动态规划思想实现递推式2，也就是自底向上  

'''  

def cut_rod_dp(p, n):  

    ri = [0] # ri的临时存放表  

    for i in range(1, n+1): # 求ri, i从1到n  

        tmp = 0  

        for j in range(0, i): # r从1到n-i  

            tmp = max(tmp, p[j] + ri[i-j-1])  

        ri.append(tmp)  

    return ri[n]

```

时间复杂度: $O(n^2)$

```

p = [1,5,8,9,10,17,17,20,24,30,32,36,40,42,44,48,50]  

import time  

start = time.time()  

print(cut_rod_cur(p,15))  

print('递推公式1: ', time.time() - start)  

start = time.time()  

print(cut_rod_cur2(p,15))  

print('递推公式2: ', time.time() - start)  

start = time.time()  

print(cut_rod_dp(p, 15))  

print('DP实现: ', time.time()-start)

```

```

45
递推公式1:  1.9533970355987549
45
递推公式2:  0.014536857604980469
45
DP实现:  9.179115295410156e-05

```

钢条切割问题——重构解：

- ▶ 如何修改动态规划算法，使其不仅输出最优解，还输出最优切割方案？

- ▶ 对每个子问题，**保存切割一次时左边切下的长度**

长度 <i>i</i>	1	2	3	4	5	6	7	8	9	10	
价格 <i>p_i</i>	1	5	8	9	10	17	17	20	24	30	
<i>i</i>	0	1	2	3	4	5	6	7	8	9	10
<i>r[i]</i>	0	1	5	8	10	13	17	18	22	25	30
<i>s[i]</i>	0	1	2	3	2	2	6	1	2	3	10

...
重构解
...

```
def cut_rod_extend(p, n):
    ri = [0] # ri的临时存放表
    si = [0] # ri对应最优值对应切割方案中左边一段的长度

    for i in range(1, n+1): # 求ri, i从1到n
        tmp_r = 0
        tmp_s = 0
        for j in range(0, i): # r从1到n-i
            if p[j] + ri[i-j-1] > tmp_r:
                tmp_r = p[j] + ri[i-j-1]
                tmp_s = j + 1
        ri.append(tmp_r)
        si.append(tmp_s)
    return ri[n], si
```

```
def cut_rod_solution(p, n):
    val, s = cut_rod_extend(p,n)
    print(s)
    ones = []
    while n > 0:
        ones.append(s[n])
        n -= ones[-1]
    return ones
```

```
p = [1,5,8,9,10,17,17,20,24,30]

cut_rod_solution(p, 9)
```

```
[0, 1, 2, 3, 2, 2, 6, 1, 2, 3]
```

5.2 最长公共子序列LCS(Longest Common Subsequence)

- ▶ 一个序列的子序列是在该序列中删去若干元素后得到的序列。
- ▶ 例：“ABCD”和“BDF”都是“ABCDEFG”的子序列
- ▶ 最长公共子序列 (LCS) 问题：给定两个序列X和Y，求X和Y长度最大的公共子序列。
- ▶ 例：X="ABBCBDE" Y="DBBCDDB" LCS(X,Y)="BBCD"
- ▶ 应用场景：字符串相似度比对

定理：令 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 为两个序列， $Z = \langle z_1, z_2, \dots, z_k \rangle$ 为 X 和 Y 的任意LCS，则：

1. 如果 $x_m = y_n$ ，则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS;
2. 如果 $x_m \neq y_n$ ，那么 $z_k \neq x_m$ 意味着 Z 是 X_{m-1} 和 Y 的一个LCS;
3. 如果 $x_m \neq y_n$ ，那么 $z_k \neq y_n$ 意味着 Z 是 X 和 Y_{n-1} 的一个LCS;

最优解的递推式：

$$C[i, j] = \begin{cases} 0, & \text{当 } i = 0 \text{ 或 } j = 0 \\ C[i - 1, j - 1] + 1, & \text{当 } i, j > 0 \text{ 且 } x_i = y_j \\ MAX(C[i, j - 1], C[i - 1, j]) & \text{当 } i, j > 0 \text{ 且 } x_i \neq y_j \end{cases}$$

上面的定理和递推公式可用如下表格表示：

第一行和第一列表示空串

j	0	1	2	3	4	5	6
i	y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	-1	-1
2	B	0	1	-1	-1	1	-2
3	C	0	1	1	2	-2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

```
def LCS_length(x,y):
    ...
    求两个字符串的最长公共子序列的长度
    ...

    x_len = len(x)
    y_len = len(y)

    lcs_table = [[0 for _ in range(y_len+1)] for _ in range(x_len+1)]

    for row in range(1, x_len+1):
        for col in range(1, y_len+1):
            if x[row-1] == y[col-1]: #两个字母相同
                lcs_table[row][col] = lcs_table[row-1][col-1] + 1 # 等于左上方值 + 1
            else: # 两个字母不想等, 则取max(左边, 右边)
                lcs_table[row][col] = max(lcs_table[row-1][col], lcs_table[row][col-1])
    return lcs_table[x_len][y_len]

    ...
    在计算lcs_table的同时, 记录子串长度值的路径
    规定: 1-来自左上方, 2-来自上方, 3-来自左方
    ...

def LCS(x, y):
```

```

x_len = len(x)
y_len = len(y)

lcs_table = [[0 for _ in range(y_len+1)] for _ in range(x_len+1)]
trace = [[0 for _ in range(y_len+1)] for _ in range(x_len+1)]

for row in range(1, x_len+1):
    for col in range(1, y_len+1):
        if x[row-1] == y[col-1]: #两个字母相同
            lcs_table[row][col] = lcs_table[row-1][col-1] + 1 # 等于左上方值 + 1
            trace[row][col] = 1
        elif lcs_table[row-1][col] > lcs_table[row][col-1]: # 说明来自上方
            lcs_table[row][col] = lcs_table[row-1][col]
            trace[row][col] = 2
        else: # 说明来自左方
            lcs_table[row][col] = lcs_table[row][col-1]
            trace[row][col] = 3
return lcs_table[x_len][y_len], trace

```

'''

根据路径求出最长子串

'''

```

def LCS_substr(x,y):
    m = len(x)
    n = len(y)

    length, trace = LCS(x, y)
    substr = []

    while m > 0 and n > 0:
        if trace[m][n] == 1: # 来自左上方, 说明该位置字符是相同的
            substr.append(x[m-1])
            m -= 1
            n -= 1
            continue
        elif trace[m][n] == 2: # 来自上方, 该位置字符是不同的
            m -= 1
            continue
        else: # 来自左方, 该位置字符是不同的
            n -= 1
            continue
    return ''.join(reversed(substr))

```

```

LCS_length('ABCBDAB', 'BDCABA')
length, trace = LCS('ABCBDAB', 'BDCABA')
for _ in trace:
    print(_)
LCS_substr('ABCBDAB', 'BDCABA')

```

```

[0, 0, 0, 0, 0, 0, 0]
[0, 3, 3, 3, 1, 3, 1]
[0, 1, 3, 3, 3, 1, 3]
[0, 2, 3, 1, 3, 3, 3]
[0, 1, 3, 2, 3, 1, 3]
[0, 2, 1, 3, 3, 2, 3]
[0, 2, 2, 3, 1, 3, 1]
[0, 1, 2, 3, 2, 1, 3]

```

```
'BDAB'
```

6. 欧几里得算法

最大公约数 (Greatest Common Divisor, GCD)

欧几里得算法: $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$, 例:

```
gcd(60, 21) = gcd(21, 18) = gcd(18, 3) = gcd(3, 0) = 3
```

```

...
递归实现gcd
...
def gcd_rec(a, b):
    if a % b == 0:
        return b
    else:
        return gcd_rec(b, a % b)
...
```

```

非递归实现gcd
...
def gcd_no_rec(a, b):
    while b != 0:
        tmp = b
        b = a % b
        a = tmp

```

```

    else:
        return a

    ...

分数的运算
    ...

class Fraction():

    def __init__(self, molecular, denominator):
        self.molecular = molecular # 分子
        self.denominator = denominator
        gcd = self.gcd(molecular, denominator)
        self.molecular = molecular / gcd
        self.denominator = denominator / gcd

    def gcd(self, molecular, denominator):
        while denominator != 0:
            tmp = denominator
            denominator = molecular % denominator
            molecular = tmp
        else:
            return molecular

    ...

分数加法
    ...

    def __add__(self, other):
        m1 = self.molecular
        d1 = self.denominator
        m2 = other.molecular
        d2 = other.denominator

        # 先统一分母，且最小公倍数
        # 6, 8 --> gcd:2 --> 6/2=3, 8/2=4 -->最小公倍数: 2*3*4
        gcd = self.gcd(d1, d2)
        denominator = gcd * (d1 / gcd) * (d2 / gcd) # 分母的最小公倍数
        molecular = m1 * (d1 / gcd) + m2 * (d2 / gcd)
        return Fraction(molecular, denominator)

    def __str__(self):
        return '%d/%d'%(self.molecular, self.denominator)

```

```
f1 = Fraction(1,2)
f2 = Fraction(1,3)
total = f1 + f2
print(total)
```

5 / 6

7. RSA算法

传统密码：加密算法是秘密的，比如凯撒码；

现代密码系统：加密算法是公开的，但秘钥是秘密的：

- 对称加密（DES、3DES、Blowfish、IDEA、RC4、RC5、RC6 和 AES）
- 非对称加密（RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用））

公钥：用于加密

私钥：用于解密

RSA加解密过程：

1. 随机选取两个质数 p 和 q；
2. 计算 $n = pq$ ；
3. 选取一个与 $\phi(n)$ 互质的小奇数 e，其中 $\phi(n) = (p-1)(q-1)$
4. 对 $\phi(n)$ ，计算 e 的乘法逆元 d，即满足 $(e \cdot d) \bmod \phi(n) = 1$
5. 公钥 (e, n) 私钥 (d, n)
6. 加密过程： $c = (m^e) \bmod n$
7. 解密过程： $m = (c^d) \bmod n$

8. 字符串算法

1. 哈希法（最直观的方法）
2. KMP算法（最基础的方法）
3. 扩展KMP算法
4. Manacher算法（解决回文串问题（aba））
5. AC自动机（Trie+KMP）

- 哈希：最简单直观，易实现，便于拓展；
- KMP：本质是利用模板串自身的信息，去减少匹配时的冗余比较，达到优秀的时间复杂度； $O(n)$
- 扩展KMP、Manacher算法：利用对称性，降低时间复杂度， $O(n)$
- AC自动机：Trie树与KMP的结合，可以解决一个串和多个串的匹配问题。

8.1 哈希法

问题：

有一个字符串集合，假设为 `['abc', 'bcd', 'adf', 'bce', 'edaf', 'adfc','ad']`，问字符串'adfc'在该集合中吗？

该问题可以用字符串哈希实现，效率比暴力计算高。

字符串哈希就是将一个字符串映射为一个整数。

哈希公式：

$$\text{hash}[i] = \text{hash}[i - 1] * p + \text{id}(s[i])$$

其中， p 为质数， $\text{id}(x)$ 为 $x - 'a' + 1$ 或者 x 的ASCII码。

```
class Node():

    def __init__(self, value=None):
        self.next = None
        self.pre = None
        self.value = value

    '''计算hash值'''
    def get_hash(string:str, p:int=31):
        h = 0
        for idx, char in enumerate(string):
            if idx == 0:
                h += ord(string[idx])
            else:
                h = h * p + ord(string[idx])
        return h

class StrHash():

    def __init__(self, mod=31, maxn=1e6+7):
        self.mod = mod
        self.li = [None for _ in range(int(maxn))]

    def insert(self, string):
        shash = get_hash(string, self.mod)

        if self.li[shash] is None:
            node = Node(string)
            self.li[shash] = node
        else:
            # 若有hash冲突, 采用头插法插入
            root_node = self.li[shash]
            if root_node.next is None:
                node = Node(string)
                root_node.next = node
                node.pre = root_node
            else:
                node = Node(string)
                root_node.next.pre = node
                node.next = root_node.next
                root_node.next = node
```

```
node.pre = root_node

def __contains__(self, item):
    shash = get_hash(item, self.mod)
    root_node = self.li[shash]
    if root_node is None:
        return False
    else:
        while root_node is not None:
            if item == root_node.value:
                return True
            else:
                root_node = root_node.next
    else:
        return False
```

```
get_hash('ab')
```

```
3105
```

```
shash = StrHash()
shash.insert('ab')
shash.insert('a')
shash.insert('abc')
shash.insert('def')

print('ab' in shash)
print('abd' in shash)
```

```
True
False
```

8.2 KMP

待匹配字符串T: 'ABACDEFBEBDFACCD'
模式字符串P: 'ACCD'

字符串匹配问题就是看T中是否有P。

概念：

1. 子串：

'ABCDABC'的前缀子串有：['A', 'AB', 'ABC', 'ABCD', 'ABCD', 'ABCDA', 'ABCDAB', 'ABCDABC']，其中，除它本身外，其它称为真前缀子串；'ABCDABC'的后缀子串有：['C', 'BC', 'ABC', 'DABC', 'CDABC', 'BCDABC', 'ABCDABC']，其中，除它本身外，其余称为真后缀子串。

KMP算法是一种改进的字符串匹配算法，由D.E.Knuth, J.H.Morris和V.R.Pratt提出的，因此人们称它为克努特—莫里斯—普拉特操作（简称KMP算法）。KMP算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。

暴力解法

```
...
T: ABCACEBDACEABC
P: ACEAB
...

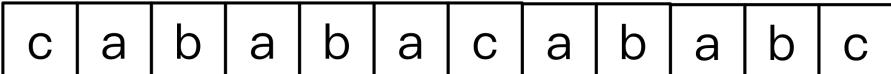
def str_match(text, pattern):
    t_idx = 0 # 待匹配字符串指针
    while t_idx <= len(text)-len(pattern):
        for i in range(t_idx, t_idx+len(pattern)):
            if text[i] == pattern[i-t_idx]:
                if i-t_idx == len(pattern)-1:
                    return True
                else:
                    continue
            else:
                t_idx += 1
                break
    return False
```

```
t = 'ABCACEBDACEABCBEABCBDFSRKJNMHBVFGRTEWXCVBNMSDFGH'
p1 = 'TEWXCVBNMSD'
p2 = 'TEWXCVBNMSG'

print(str_match(t, p1))
print(str_match(t, p2))
```

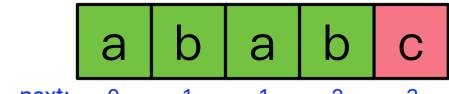
```
True
False
```

KMP算法

待匹配串: 

模版串: 

next: 0 1 1 2 3



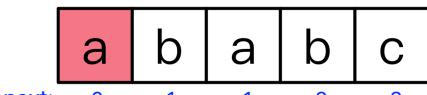
next: 0 1 1 2 3



next: 0 1 1 2 3



next: 0 1 1 2 3



next: 0 1 1 2 3



next: 0 1 1 2 3

	A	B	A	B	A	A	A	B	A	B	A	B	A
值	0	1	1	2	3	4	2	2	3	4	5	6	
下标	0	1	2	3	4	5	6	7	8	9	10	11	12

当下标i处不匹配时，将指针指向最大前缀的后一位置处。

```
...
0 1 2 3 4 5 6 7 8 9 10 11
A B A B A A A B A B A   A
0 1 1 2 3 4 2 2 3 4 5 6
...

```

```
def next_idx(pattern):
    # 前两个元素始终为0,1
    idx = [1 for _ in range(len(pattern))]
    idx[0] = 0
    # 从第3个元素开始计算
    for i in range(2, len(pattern)):
        for j in range(i - 1, 0, -1):
            if pattern[:j] == pattern[i-j:i]:
                idx[i] = j + 1
                break
    return idx
```

```
print(next_idx('ABABAAABABAA'))
print(next_idx('ABCDEFG'))
```

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

```
def str_match_kmp(text, pattern):
    p_next = next_idx(pattern)

    t_idx = 0 # text指针当前位置
    p_idx = 0 # pattern指针当前位置

    while t_idx < len(text) and p_idx < len(pattern):
        if text[t_idx] == pattern[p_idx]:
            if p_idx == len(pattern) - 1:
                return True
            else:
                t_idx += 1
                p_idx += 1
                continue
        else:
            next_id = p_next[p_idx]
            if next_id == 0:
                t_idx += 1
                continue
            else:
                p_idx = next_id - 1
                continue
    else:
        return False
```

```
t = 'ABCDESDSFGCDGABCDEF'
p = 'ABCDEF'
print(str_match(t, p))
```

```
True
```

```

t = 'ABCACEBDACEABCBEABCBDFAFSRKJNMHBVFGRTIEWXCVBNMSDFGH'
p1 = 'TEWXCVBNMSD'
p2 = 'TEWXCVBNMSG'

print(str_match(t, p1))
print(str_match(t, p2))

```

```

True
False

```

```

text = ''
with open('kmp_text.txt') as f:
    lines = f.readlines()
    for line in lines:
        text = text + line
print(text)

pattern = '所以复杂度'
pattern2 = '所以复杂度啊'

```

说明KMP算法看懂了觉得特别简单，思路很简单，看不懂之前，查各种资料，看的稀里糊涂，即使网上最简单的解释，依然看的稀里糊涂。我花了半天时间，争取用最短的篇幅大致搞明白这玩意到底是啥。这里不扯概念，只讲算法过程和代码理解：KMP算法求解什么类型问题字符串匹配。给你两个字符串，寻找其中一个字符串是否包含另一个字符串，如果包含，返回包含的起始位置。如下面两个字符串：比如我们已经知道ababab， $q=4$ 时， $\text{next}[4]=2$ ($k=2$ ，表示该字符串的前5个字母组成的子串ababa存在相同的最长前缀和最长后缀的长度是3，所以 $k=2, \text{next}[4]=2$)。这个结果可以理解成我们自己观察算的，也可以理解成程序自己算的，这不是重点，重点是程序根据目前的结果怎么算 $\text{next}[5]$ 的。那么对于字符串ababab，我们计算 $\text{next}[5]$ 的时候，此时 $q=5, k=2$ （上一步循环结束后的结果）。那么我们需要比较的是 $\text{str}[k+1]$ 和 $\text{str}[q]$ 是否相等，其实就是 $\text{str}[1]$ 和 $\text{str}[5]$ 是否相等！，为啥从 $k+1$ 比较呢，因为上一次循环中，我们已经保证了 $\text{str}[k]$ 和 $\text{str}[q]$ （注意这个 q 是上次循环的 q ）是相等的（这句话自己想想，很容易理解），所以到本次循环，我们直接比较 $\text{str}[k+1]$ 和 $\text{str}[q]$ 是否相等（这个 q 是本次循环的 q ）。如果相等，那么跳出`while()`，进入`if()`， $k=k+1$ ，接着 $\text{next}[q]=k$ 。即对于ababab，我们会得出 $\text{next}[5]=3$ 。这是程序自己算的，和我们观察的是一样的。如果不等，我们可以用“ababac”描述这种情况。不等，进入`while()`里面，进行 $k=\text{next}[k]$ ，这句话是说，在 $\text{str}[k + 1] != \text{str}[q]$ 的情况下，我们往前找一个 k ，使 $\text{str}[k + 1] == \text{str}[q]$ ，是往前一个一个找呢，还是有更快的找法呢？（一个一个找必然可以，即你把 $k = \text{next}[k]$ 换成 $k--$ 也是完全能运行的（更正：这句话不对啊，把 $k=\text{next}[k]$ 换成 $k-$ 是不行的，评论25楼举了个反例）。但是程序给出了一种更快的找法，那就是 $k = \text{next}[k]$ 。程序的意思是说，一旦 $\text{str}[k + 1] != \text{str}[q]$ ，即在后缀里面找不到时，我是可以直接跳过中间一段，跑到前缀里面找， $\text{next}[k]$ 就是相同的最长前缀和最长后缀的长度。所以， $k=\text{next}[k]$ 就变成， $k=\text{next}[2]$ ，即 $k=0$ 。此时再比较 $\text{str}[0+1]$ 和 $\text{str}[5]$ 是否相等，不等，则 $k=\text{next}[0]=-1$ 。跳出循环。（这个解释能懂不？）我们可以看到，匹配串每次往前移动，都是一大段一大段移动，假设匹配串里不存在重复的前缀和后缀，即 next 的值都是 -1 ，那么每次移动其实就是一个匹配串往前移动 m 个距离。然后重新一一比较，这样就比较 m 次，概括为，移动 m 距离，比较 m 次，移到末尾，就是比较 n 次， $O(n)$ 复杂度。假设匹配串里存在重复的前缀和后缀，我们移动的距离相对小了点，但是比较的次数也小了，整体代价也是 $O(n)$ 。所以复杂度是一个线性的复杂度。

```
import time

begin = time.time()
print(str_match(text, pattern))
print(str_match(text, pattern2))
print('暴力法总共耗时: ', time.time() - begin)

begin = time.time()
print(str_match_kmp(text, pattern))
print(str_match_kmp(text, pattern2))
print('KMP总共耗时: ', time.time() - begin)
```

```
True
False
暴力法总共耗时:  0.002076864242553711
True
False
KMP总共耗时:  0.0011548995971679688
```

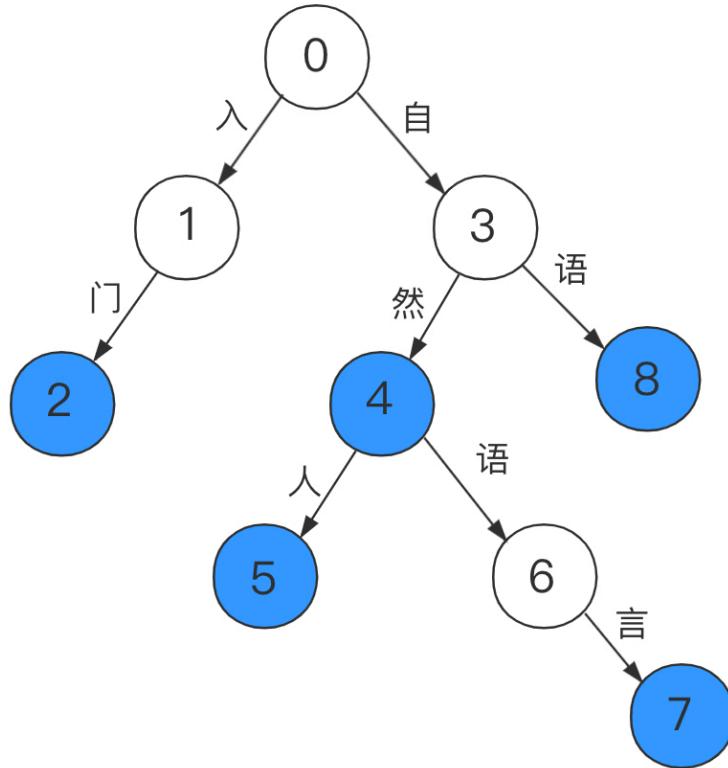
KMP的核心：主要是通过next()，避免了待匹配字符串指针的回溯，从而节省匹配时间。

8.3：字典树（trie树）

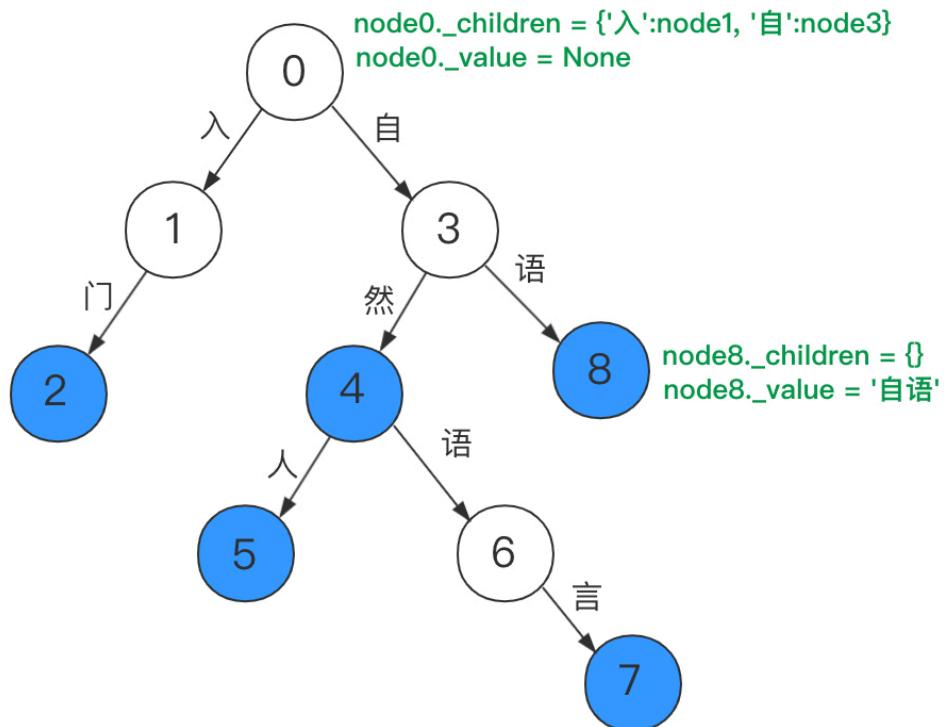
在字典分词算法中，我们需要判断当前字符串是否在字典中。如果用有序集合（TreeMap）的话，复杂度是 $O(\log n)$ ；如果用散列表（HashMap）的话，时间复杂度虽然下降了，但内存复杂度却上去了。

字典树，又叫trie树、前缀树，有如下性质：

1. 字典树每条边都对应一个字；
2. 从根节点往下的路径构成一个个字符串；
3. 字典树并不直接在节点上存储字符串，而是将词语视作根节点到某节点之间的一条路径，并在终点节点（蓝色）上做标记“该节点对应词语的结尾”；
4. 字符串就是一条路径，要查询一个单词，只需顺着这条路径从根节点往下走。



当词典大小为 n 时，虽然最坏情况下字典树的复杂度依然是 $O(\log n)$ （假设子节点用对数复杂度的数据结构存储，所有词语都是单字），但他的实际速度要比二分查找快。这是因为随着路径的深入，前缀匹配是递进的过程，算法不必比较字符串的前缀。



```
class Node:

    def __init__(self, value):
        self._children = {}
```

```

        self._value = value

    def _add_child(self, char, value, overwrite=False):
        child = self._children.get(char)
        if child is None: # 对应子节点为None
            child = Node(value)
            self._children[char] = child
        elif overwrite:
            child._value = value
        return child

class Trie(Node):

    def __init__(self):
        super().__init__(None) # 初始化一个根节点

    # 覆写__contains__魔法方法
    def __contains__(self, key):
        return self[key] is not None # 等价于 self.__getitem__(key)

    # 可以像对待dict一样操作字典树
    def __getitem__(self, key):
        root = self
        for char in key:
            root = root._children.get(char)
            if root is None:
                return None
        return root._value

    def __setitem__(self, key, value):

        root = self

        for idx, char in enumerate(key):
            if idx < len(key) - 1:
                root = root._add_child(char, None, False)
            else: # 蓝色节点
                root = root._add_child(char, value, True)

```

```

trie = Trie()
trie['入门'] = 'introduction'
trie['自然人'] = 'human'
trie['自语'] = 'speak to oneself'
trie['自然'] = 'nature'

print('自然' in trie)
print('自然猪' in trie)

# 删

```

```

trie['自然'] = None
print('自然' in trie)

# 改
trie['自然人'] = 'nature human'
print(trie['自然人'])

# 查
print(trie['入门'])

```

```

True
False
False
nature human
introduction

```

8.4 AC自动机

Aho-Corasick automaton, 该算法在1975年产生于贝尔实验室，是著名的多模匹配算法。

多模式匹配 (multi-pattern matching) : 给定多个词语（也称模式串， pattern），从母文本中匹配它们的问题称多模式匹配，比如：

母文本: ushers

模式串集合: {he, she, his, hers}

前置知识: trie树、KMP、BFS

KMP: 单对单文本匹配，比如母文本: ABACDEFADBCA, pattern: ADBCA

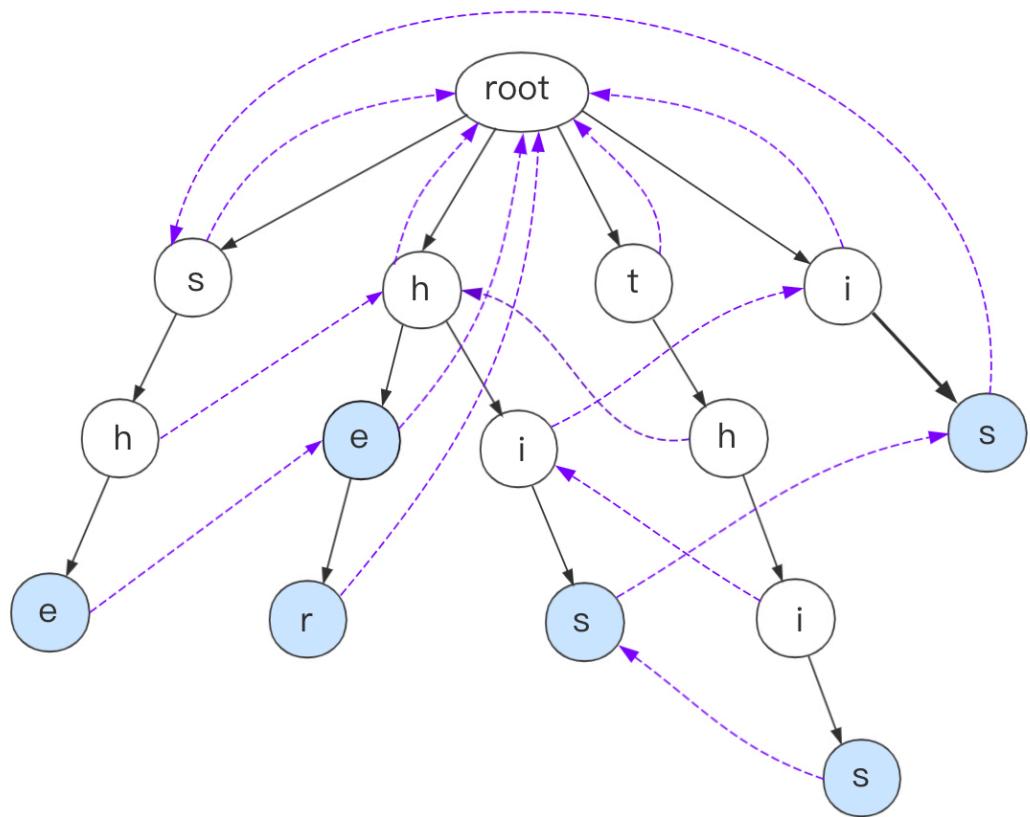
trie树: 多对单文本匹配，比如母文本集合: {自然、自然人、自然语言、入门} pattern: 自然语言

AC自动机: 单对多文本匹配。

AC自动机的实现原理:

1. 第模式串构建trie树；
2. 构建fail指针: (fail本质上当前单词的最长后缀，比如: this的最长后缀是his, 所以this的s指向his的s。)
 - 它是BFS来构建；
 - 它的第一层全部指向root
 - fail指向: 它的父亲的fail节点的同字符儿子，若没有找到，就继续跳fail，直到跳到root还没有，指向root
3. 匹配 (本质上是在fail链上找单词)
 - 从根出发，从第一个文本首字母出发；
 - 进行trie匹配
 - 如果有，进入它
 - 如果没有跳fail，直到root
 - 如果跳到root，下一个文本串
 - 找到单词，不断跳fail，并进行回溯

s: sherthis
 p: she he her this his is



匹配过程:

1.

```

class ACNode():

  def __init__(self, character=None):
    self.children = {}
    self.pre = None
    self.character = character
    self.pattern = []
    self.fail = None

  '''根据character获取子节点'''
  def child_node(self, character):
    if character in self.children:
      return self.children[character]
    else:
      return None

  def extend_pattern(self, pattern):
    self.pattern.extend(pattern)

  def append_pattern(self, pattern):
    self.pattern.append(pattern)
  
```

```

from collections import deque

class ACAutomaton():

    def __init__(self, pattern_list=[ ]):
        self.root = ACNode()
        self.root.fail = self.root # root的fail指向它本身
        self.fail_finished = False
        for pattern in pattern_list:
            self.add_pattern(pattern)

    '''添加模式串'''
    def add_pattern(self, pattern):
        cur_node = self.root
        for character in pattern: # 遍历模式串
            child_node = cur_node.child_node(character)
            if child_node:
                cur_node = child_node
            else:
                child_node = ACNode(character)
                cur_node.children[character] = child_node
                child_node.pre = cur_node
                cur_node = child_node
        cur_node.append_pattern(pattern)

    '''遍历所有pattern'''
    def get_all(self, cur_node):
        if len(cur_node.pattern) > 0:
            print(cur_node.pattern)
        for key in cur_node.children:
            self.get_all(cur_node.children[key])

    '''构建fail链'''
    def __construct_fail(self):
        if self.fail_finished:
            return
        # BFS遍历每个节点，构建fail链
        queue = deque()

        # 第一层的节点的fail指向root
        for key in self.root.children:
            self.root.children[key].fail = self.root
            queue.append(self.root.children[key])

        while len(queue) > 0:
            cur_node = queue.popleft()
            cur_char = cur_node.character

```

```
# 将当前节点的子节点添加到queue中
children = cur_node.children
for key, node in children.items():
    queue.append(node)

# 设置当前节点的fail节点
if cur_node.fail is None:
    parent_node = cur_node.pre
    fail_node = parent_node.fail
    while cur_char not in fail_node.children:
        if fail_node == self.root:
            cur_node.fail = fail_node
            break
        else:
            fail_node = fail_node.fail
            continue
    else:
        cur_node.fail = fail_node.children[cur_char]
        cur_node.extend_pattern(fail_node.children[cur_char].pattern)
self.fail_finished = True

'''模式匹配'''
def match(self, text):

    # 先检查并构建fail链
    self.__construct_fail()

    # 根据fail链进行匹配
    root = self.root
    out_pattern = []

    for character in text:
        if character in root.children: # 如果匹配成功，则沿着trie树继续向下匹配
            root = root.children[character]
            if len(root.pattern) > 0:
                out_pattern.extend(root.pattern)
        else: # 如果匹配不成功，则根据fail链进行匹配
            fail_node = root.fail

            while character not in fail_node.children:
                fail_node = fail_node.fail
                if fail_node == self.root:
                    break
            else:
                root = fail_node.children[character]
                if len(root.pattern) > 0:
                    out_pattern.extend(root.pattern)

    return out_pattern
```

```

patterns = ['she', 'he', 'her', 'his', 'this', 'is']
ac = ACAutomaton(patterns)
ac.get_all(ac.root)
print(ac.match('shерthis'))

```

```

['she']
['he']
['her']
['his']
['this']
['is']
['she', 'he', 'her', 'this', 'his', 'is']

```

8.5 Manacher算法（回文串问题）

a

```

import time

def display_run_time(func):
    def wrapper(*args):
        t1 = time.time()
        result = func(*args)
        t2 = time.time()
        print('Total Time: %.6fs' % (t2 - t1))
        return result
    return wrapper

```

暴力解法

遍历每一个子串，校验它们是否是回文串，时间复杂度为 $O(n^3)$

```

@display_run_time
def brute_force(string):
    size = len(string)

    max_len = 0
    start = 0
    for i in range(size):
        for j in range(i+1, size+1): # 两层循环，遍历所有的子串
            sub_string = string[i:j]
            sub_len = len(sub_string)
            for k in range(sub_len):
                if sub_string[k] != sub_string[sub_len-k-1]:
                    break
                elif sub_len-k-1 < k and sub_len > max_len:
                    max_len = sub_len

```

```
        start = i
        break
return start, max_len
```

```
start, max_len = brute_fore('dababac')
print(start, max_len)
```

```
Total Time:0.000027s
1 5
```

中心扩展法

以被考虑词为中心，向两边扩展，分两种情况讨论：

1. 长度为奇数，比如：aba
2. 长度为偶数，比如：abba

时间复杂度 $O(n^2)$

```
@display_run_time
def center_extend(string):
    ...
    abadc
    abba
    ...
    match_string = ''
    size = len(string)

    # 按奇数考虑
    for i in range(size):
        j = i-1
        k = i+1
        while j >= 0 and k < size and string[j] == string[k]:
            j -= 1
            k += 1
        else:
            if k-j-1 > len(match_string):
                match_string = string[j+1:k]

    # 按偶数考虑
    for i in range(size):
        j = i
        k = i+1
        while j >= 0 and k < size and string[j] == string[k]:
            j -= 1
            k += 1
        else:
            if k-j-1 > len(match_string):
```

```
        match_string = string[j+1:k]

    return match_string
```

```
print(center_extend('a'))
print(center_extend('aba'))
print(center_extend('abba'))
print(center_extend('eabba'))
print(center_extend('acbdadbe'))
```

```
Total Time:0.000004s
a
Total Time:0.000006s
aba
Total Time:0.000006s
abba
Total Time:0.000006s
abba
Total Time:0.000007s
bdadb
```

```
@display_run_time
def manacher(string):
    max_len = 0

    # 1. 字符串预处理
    pre_str = '#'
    for s in string:
        pre_str = pre_str + s + '#'

    length = len(pre_str)

    # 2. 初始化半径列表和最右边解
    rad = [0 for _ in range(len(pre_str))]
    max_right = 0 # max_right对应的字符包含在回文串中
    mid_point = 0 # max_right对应的回文中点

    # 3. 遍历每个字符, 找最长回文串
    for i in range(length):
        # 根据已有回文半径, 得到当前i位置扩充前的回文半径
        if i < max_right:
            # 如果 i 在max_right的左边, 则rad[i]要么等于i 关于 mid_pos 对称点 i' 的rad[i'] ,
            # 要么等于max_right - i
            rad[i] = min(rad[2 * mid_point - i], max_right - i)
        else:
```

```
    rad[i] = 1 # 点i在max_right的右边

    # 拓展rad[i]
    while i - rad[i] >=0 and i + rad[i] < length and pre_str[i - rad[i]] ==
pre_str[i + rad[i]]:
        rad[i] += 1

    # 更新max_right和mid_pos
    if rad[i] + i -1 > max_right:
        max_right = rad[i] + i -1
        mid_point = i

    # 更新最长回文串的长度
    max_len = max(rad[i], max_len)
return max_len - 1
```

```
print(manacher('abbcbb'))
print(manacher('ddcdabbcbbfafgag'))
print(manacher('aaaa'))
print(manacher('a'))
```

```
Total Time:0.000023s
```

```
5
```

```
Total Time:0.000043s
```

```
5
```

```
Total Time:0.000015s
```

```
4
```

```
Total Time:0.000005s
```

```
1
```

```
chr(98)
```

```
'b'
```

```
import random
from copy import deepcopy

mid_str = [chr(random.randint(97, 123)) for i in range(100)]
pre_str = [chr(random.randint(97, 123)) for i in range(100)]
post_str = [chr(random.randint(97, 123)) for i in range(200)]
```

```
str1 = ''.join(mid_str)
str2 = str1[::-1]

mid_str = str1 + 'a' + str2

string = ''.join(pre_str) + mid_str + ''.join(post_str)

print(mid_str)
print('-'*10)
print(string)
```

xhjmvhndkdpdxaygeaayjscmcfcekw{bfmdobkarrdtzlootyxkpa{towzbo{xycy{nnaqnrldlxjexiqybeds
opbzpuinernbeabenreniupzbposdebyqixejaxldrqnqann{ycyx{obzwot{apkxytoolztdrrakbodmfb{wdkec
fcmcsjyaegeraxdpxdkdnhhvmjhx

aabshucickixhikvhabblnbmqrssrhfqzz{mtnquspsrebvyanddyhmo{qlurbiflwpedmwzgihqdpzvddqwvyf
mwgbjwruvpyqhxhjmvhndkdpdxaygeaayjscmcfcekw{bfmdobkarrdtzlootyxkpa{towzbo{xycy{nnaqnr
dlxjexiqybedsopbzpuinernbeabenreniupzbposdebyqixejaxldrqnqann{ycyx{obzwot{apkxytoolztdrra
kbodmfb{wdkecfcmcsjyaegeraxdpxdkdnhhvmjhxbnsnijrhnxjbkjqdvhnwdlzpmqfdmpyeuvfjedc{kdcibq
mlfkrugebjackxbtwww{wmfxszbzkgedgsxozkmqsettigjduhtqfvsil{ebdtjth{gppeubfvaxemfjwxndr
lrbvevoifbsy{tqccmsjbulnlcziopvkfsdhhzjkxxfdws1{gvlpdolhosyncjghvp

```
brute_fore(string)
```

Total Time:0.069387s

(100, 201)

```
center_extend(string)
```

Total Time:0.000334s

```
'xhjmvhndkdpdxaygeaayjscmcfcckdw{bfmdobkarrdtzlootyxkpa{towzbo{xycy{nnaqnrdlxjexiqybed  
sopbzpuinernbeaebnreniupzbposdebyqixejaxldrnqann{ycyx{obzwot{apkxytoolztdrrakbodmfb{wdke  
cfcmcscopyaaegyaxdpdkdnhhvmjhx'
```

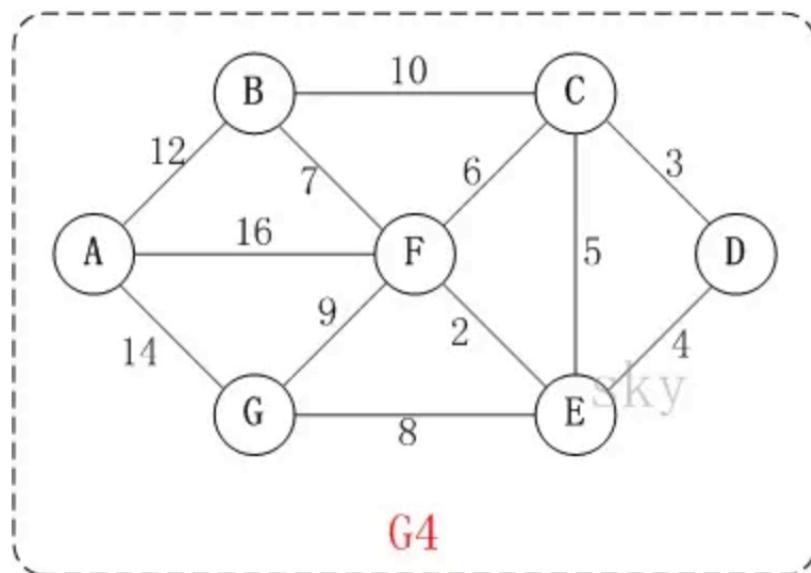
```
manacher(string)
```

```
Total Time:0.001298s
```

201

9. Dijkstra (迪杰斯特拉算法)

对于如下一个五向带权连通图，求一个顶点到剩余其它顶点的最短路径。



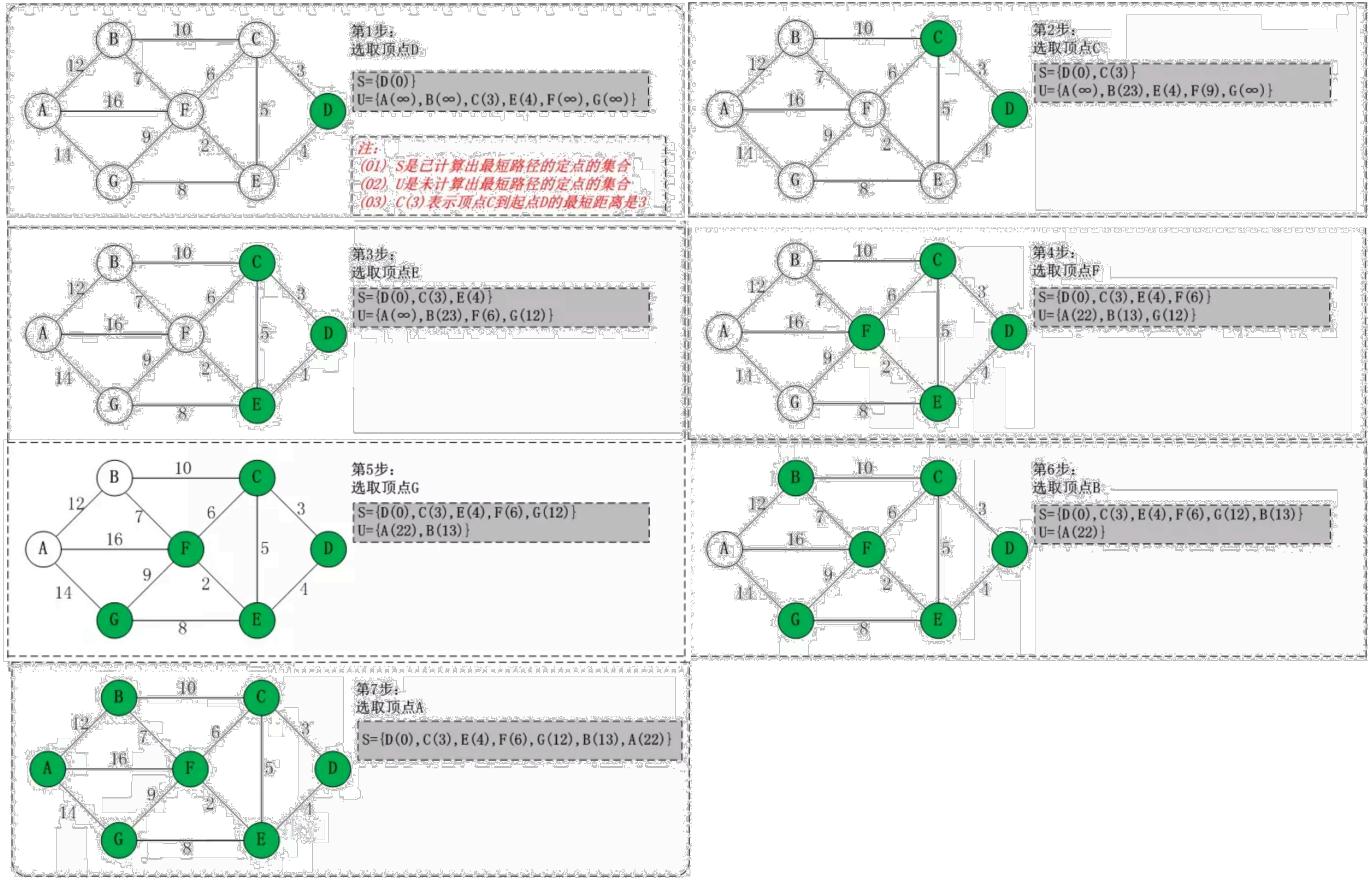
Dijkstra算法的核心是维护三个数组：

- **visited**数组：可以将顶点标记为已被访问的和未被访问的；
- **path**数组：维护起点到任意顶点的路径；
- **distance**数组：维护任意时刻各顶点到起点的距离。

实现步骤：

1. 将图转化为一个图邻接矩阵；
2. 初始化visited、path、distance三数组；
3. 对剩余n-1个顶点进行探寻：

- 从distance中找出当前未被探寻(visit=false)的，距起点最近的顶点；
- 更新visited、distance、path数组
- 以当前顶点为中转点，更新剩余未被访问顶点到源点的距离，更新原则：
 - 未被访问的顶点；
 - 中转点 + 中转点到目标点的距离 < 源点到目标点的距离



注意事项:

- 不能出现负权值边；
- 时间复杂度为 $O(n^2)$

```
from copy import deepcopy
inf = float('inf')

def dijkstra(weight, start):
    count = len(weight)

    # 定义visited、distance、path三个数组
    visited = [False for _ in range(count)]
    path = [[start] for _ in range(count)]
    distance = [inf for _ in range(count)]

    # 根据起点信息初始化visited、distance
    visited[start] = True
    for i in range(count):
```

```

distance[i] = weight[start][i]

# 循环更新三个数组
for _ in range(1, count): # 循环n-1次
    # 寻找最近的未标记节点
    mdis = inf
    midx = 0
    for idx, dis in enumerate(distance):
        if dis != 0 and dis < mdis and visited[idx] == False:
            mdis = dis
            midx = idx

    # 更新visited和path和当前顶点的distance
    visited[midx] = True
    distance[midx] = mdis
    path[midx].append(midx)

    # 循环更新起始顶点到剩余顶点的distance
    for idx, val in enumerate(visited):
        if val == True:
            continue
        elif mdis + weight[midx][idx] < distance[idx] and weight[midx][idx] != inf:
            distance[idx] = mdis + weight[midx][idx]
            path[idx] = deepcopy(path[midx])

return distance, path

```

```

weight = [
    [0, 12, inf, inf, inf, 16, 14],
    [12, 0, 10, inf, inf, 7, inf],
    [inf, 10, 0, 3, 5, 6, inf],
    [inf, inf, 3, 0, 4, inf, inf],
    [inf, inf, 5, 4, 0, 2, 8],
    [16, 7, 6, inf, 2, 0, 9],
    [14, inf, inf, inf, 8, 9, 0]
]
distance, path = dijkstra(weight, start=2)
print(distance)
print(path)

```

```

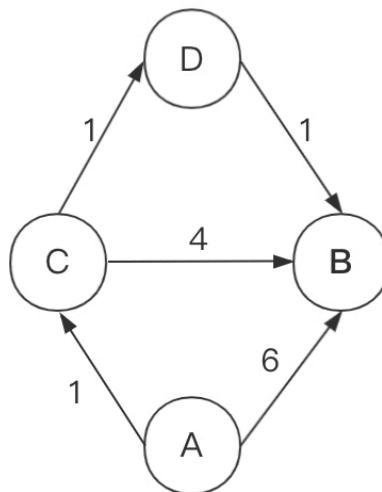
[22, 10, 0, 3, 5, 6, 13]
[[2, 5, 0], [2, 1], [2], [2, 3], [2, 4], [2, 5], [2, 4, 6]]

```

10. Floyd算法

解决的问题：

求一个带权有向图 (Weighted Directed Graph) 的任意两点的最短距离的计算，运用了动态规划的思想，**算法的时间复杂度为** $O(V^3)$ ，**空间复杂度为** $O(V^2)$ 。



算法思想：

从任意节点*i*到任意节点*j*的最短路径不外乎有两种可能：

1. 直接从*i*到*j*；
2. 从*i*经过若干中间节点*k*到达*j*。

所以我们假设 $Dis(i, j)$ 为节点*i*到节点*j*的最短路径的距离，对于每一个节点*k*，我们检查

$Dis(i, k) + Dis(k, j) < Dis(i, j)$ 是否成立，如果成立，证明从 $i \rightarrow k \rightarrow j$ 比 $i \rightarrow j$ 路径更短，我们便设置 $Dis(i, j) = Dis(i, k) + Dis(k, j)$ ，这样一来，当我们遍历往所有节点*k*， $Dis(i, j)$ 中记录的便是 $i \rightarrow j$ 最短路径的距离。

算法关键：

```

for(k=0;k<n;k++) //中转站0~k
    for(i=0;i<n;i++) //i为起点
        for(j=0;j<n;j++) //j为终点
            if(d[i][j]>d[i][k]+d[k][j]) //松弛操作
                d[i][j]=d[i][k]+d[k][j];
  
```

```

inf = 99999999 # 表示无穷远距离

def floyd(weight):
    vex_num = len(weight)

    # 定义路径矩阵、距离矩阵
    path = [[-1 for _ in range(vex_num)] for _ in range(vex_num)]

    # 将顶点i到它自身的路径标记为它本身
    for i in range(vex_num):
        path[i][i] = i

    # 以任意顶点为中转点，更新所有两点之间的距离。
  
```

```

for k in range(vex_num): # 中间点
    for i in range(vex_num): # 起点
        for j in range(vex_num): # 终点
            if weight[i][k] + weight[k][j] < weight[i][j]:
                weight[i][j] = weight[i][k] + weight[k][j]
                path[i][j] = [i,k,j]
return weight, path

```

```

weight = [
    [0, 6, 1, inf],
    [inf, 0, inf, inf],
    [inf, 4, 0, 1],
    [inf, 1, inf, 0]
]

distance, path = floyd(weight)
for dis in distance:
    print(dis)
print()
for p in path:
    print(p)

```

```

[0, 3, 1, 2]
[99999999, 0, 99999999, 99999999]
[99999999, 2, 0, 1]
[99999999, 1, 99999999, 0]

[0, [0, 3, 1], -1, [0, 2, 3]]
[-1, 1, -1, -1]
[-1, [2, 3, 1], 2, -1]
[-1, -1, -1, 3]

```

10.1 Dijkstra和Floyd算法比较

Dijkstra	Floyd
不能处理负权图	能处理负权图
处理单源最短路径	处理多源最短路径
时间复杂度 $O(n^2)$	时间复杂度 $O(n^3)$

其实，也可以对每一个顶点执行一遍Dijkstra算法得到任意两点之间的距离，时间复杂度也是 $O(n^3)$ ；对于稀疏的图，n次Dijkstra更出色；而对于稠密的图，可以使用Floyd算法。