

2.2 查找+排序lowB三人组

1. 递归实例： 汉诺塔问题

n个盘子时：

1. 把n-1个圆盘从A经过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经过A移动到C

```
def hanoi(n, A, B, C):  
    # A=from_pole B=through_pole C=to_pole  
    if n > 0:  
        hanoi(n-1, A, C, B)  
        print("%s->%s" % (A, C))  
        hanoi(n-1, B, A, C)  
  
hanoi(4, "A", "B", "C")
```

不太懂的小伙伴可以查看知乎：

[Fireman A的答案](#)

2. 列表查找

2.1列表查找： 从列表中查找指定元素

- 输入： 列表， 待查元素
- 输出： 元素下标或未查找到元素

2.1.1 顺序查找

- 从列表第一个元素开始，顺序进行搜索，直到找到为止。
 - 若n是列表长度，顺序查找的时间复杂度为 $O(n)$

2.1.2 二分查找

- 从有序列表的候选区data[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 $O(\log n)$

```
li = [1, 2, 5, 67, 2, 6, 7]  
  
def bin_search(li, val):  
    low = 0  
    high = len(li)-1  
    while low <= high:  
        mid = (low+high) // 2  
        if li[mid] == val:
```

```
        return mid
    elif li[mid] < val:
        low = mid + 1
    else:
        high = mid - 1
    return -1
```

递归版本的二分查找

```
def bin_search_rec(data_set, value, low, high):
    if low <= high:
        mid = (low+high) // 2
        if data_set[mid] == value:
            return mid
        elif data_set[mid] > value:
            return bin_search_rec(data_set, value, low, mid-1)
        else:
            return bin_search_rec(data_set, value, mid+1, high)
    else:
        return
```

尾递归：（递归在return的地方），因为不用管递归后跳出去的事，有些例如Java，C++会优化成循环，速度会快。但是Python没有优化递归，速度比循环慢。

为什么不先排序再查找

因为最快的排序是 $O(n\log n)$ + 二分查找 $O(\log n)$ 。顺序查找为 $O(n)$ 。所以直接用顺序查找。

3. 列表排序

- 列表排序
 - 将无序列表变为有序列表
- 应用场景：
 - 各种榜单
 - 各种表格
 - 给二分查找用
 - 给其他算法用
- 输入：无序列表
- 输出：有序列表
- 升序与降序

列表排序的稳定性：

例子：(2, A) (3, B) (1, C) (2, D)

排序完之后为 (1, C) (2, A) (2, D) (3, B)。 (2, A) (2, D) 的先后位置排序前后不变。

总结：相邻两个交换的是稳定的，非相邻两个交换的不稳定。

排序low B三人组：

- 冒泡排序
- 选择排序
- 插入排序

排序NB三人组:

- 快速排序
- 堆排序 (难点)
- 归并排序

没什么人用的排序:

- 基数排序
- 希尔排序
- 桶排序

3.1 冒泡排序

思路: 首先, 列表每两个相邻的数, 如果前边的比后边的大, 那么交换这两个数

cal_time.py在1.1章中

```
import random
from cal_time import*

@cal_time
def bubble_sort(li):
    for i in range(len(li)-1):
        for j in range(len(li) - i - 1):
            if li[j] > li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]

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

代码关键点:

- 趟数 (n-i)
- 无序区 (n-i-1)

冒泡排序-优化

- 如果冒泡排序中执行一趟而没有交换, 则列表已经是有序状态, 可以直接结束算法。

```
import random
from cal_time import*

@cal_time
def bubble_sort(li):
    for i in range(len(li)-1):
        exchange = False
        for j in range(len(li) - i - 1):
            if li[j] > li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
                exchange = True
        if not exchange:
            break
```

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

最好情况 $O(n)$

平均情况 $O(n^2)$

最坏情况 $O(n^2)$

稳定

3.2 选择排序

思路：

- 一趟遍历记录最小的数，放到第一个位置；
- 再一趟遍历记录剩余列表中最小的数，继续放置；

```
import random
from cal_time import*

def get_min_pos(li):
    min_pos = 0
    for i in range(1, len(li)):
        if li[i] < li[min_pos]:
            min_pos = i
    return min_pos

def select_sort(li):
    for i in range(len(li)-1):
        min_pos = i
        for j in range(i+1, len(li)):
            if li[j] < li[min_pos]:
                min_pos = j
        li[i], li[min_pos] = li[min_pos], li[i]

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

最好情况 $O(n)$

平均情况 $O(n^2)$

最坏情况 $O(n^2)$

选择排序不稳定

例子：2A 2B 1 3，排序完 1 2B 2A 3

3.3 插入排序

思路：

- 列表被分为有序区和无序区两个部分。最初有序区只有一个元素。

- 每次无序区选择一个元素，插入到有序区的位置，直到无序区变空。

代码关键点：

- 摸到的手牌（无序区选择的一个元素）
- 手里的牌（有序区）

```
import random
from cal_time import *

@cal_time
def insert_sort(li):
    for i in range(1, len(li)):
        tmp = li[i]
        j = i - 1
        while j >= 0 and li[j] > tmp:
            li[j+1] = li[j]
            j -= 1
        li[j+1] = tmp

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

最好情况 $O(n)$

平均情况 $O(n^2)$

最坏情况 $O(n^2)$