

# 算法

## 算法概念

我们通过计算机进行编程，计算机多才多艺，但不太善于独立思考，我们必须提供详尽的细节，使用它们能够明白的语言将算法提供给它们。

如果将最终写好运行的程序比作战场，我们码农便是指挥作战的将军，而我们所写的代码便是士兵和武器。数据结构和算法则是兵法。我们可以不看兵法在战场上肉搏，如此，可能会胜利，可能会失败。即使胜利，可能也会付出巨大的代价。我们写程序亦然：如果不懂算法，有时面对问题可能会没有任何思路，不知如何下手去解决；大部分时间可能解决了问题，可是对程序运行的效率和开销没有意识，性能低下；有时会借助别人开发的利器暂时解决了问题，可是遇到性能瓶颈的时候，又不知该如何进行针对性的优化。

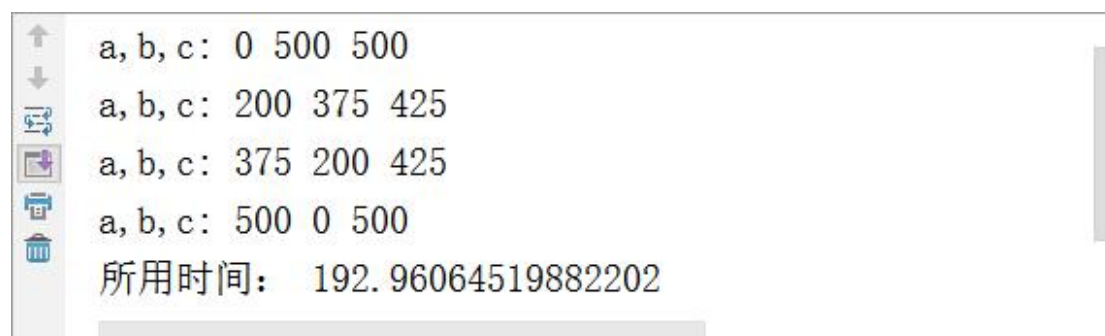
如果我们常看兵法，便可做到胸有成竹，有时会事半功倍！同样，如果我们常看算法，我们写程序时也能游刃有余、明察秋毫，遇到问题时亦能入木三分、迎刃而解。

## 算法的提出

**【示例】**如果  $a+b+c=1000$ ，且  $a^2+b^2=c^2$  ( $a,b,c$  为自然数)，如何求出所有  $a$ 、 $b$ 、 $c$  可能的组合？

```
import time
start_time=time.time()
for a in range(1001):
    for b in range(1001):
        for c in range(1001):
            if a+b+c==1000 and a**2+b**2==c**2:
                print('a,b,c:',a,b,c)
end_time=time.time()
print('所用时间: ',(end_time-start_time))
```

执行结果如图所示：



```
a, b, c: 0 500 500
a, b, c: 200 375 425
a, b, c: 375 200 425
a, b, c: 500 0 500
所用时间: 192.96064519882202
```

算法是独立存在的一种解决问题的方法和思想。

对于算法而言，实现的语言并不重要，重要的是思想。

算法可以有不同的语言描述实现版本（如 C 描述、C++描述、Python 描述等），我们现在是在用 Python 语言进行描述实现。

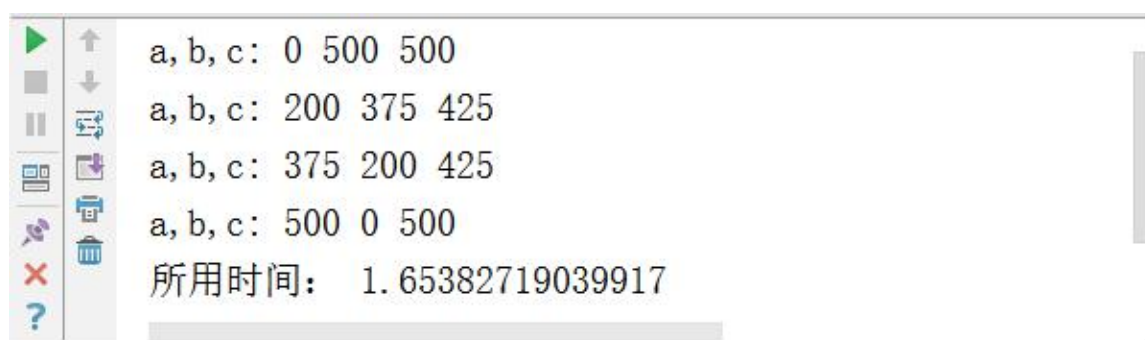
## 算法的五大特征

- (1) 输入性：有零个或多个外部量作为算法的输入
- (2) 输出性：算法至少有一个量作为输出
- (3) 确定性：算法中每条指令清晰，无歧义
- (4) 有穷性：算法中每条指令的执行次数有限，执行每条指令时间也有限
- (5) 可行性：算法原则上能够精确的运行，而且人们用纸和笔做有限次运算后即可完成

**【示例】**如果  $a+b+c=1000$ ，且  $a^2+b^2=c^2$ （ $a,b,c$  为自然数），如何求出所有  $a$ 、 $b$ 、 $c$  可能的组合？

```
import time
start_time=time.time()
for a in range(1001):
    for b in range(1001):
        c=1000-a-b
        if a**2+b**2==c**2:
            print('a,b,c:',a,b,c)
end_time=time.time()
print('所用时间：',(end_time-start_time))
```

执行结果如图所示：



```
a, b, c: 0 500 500
a, b, c: 200 375 425
a, b, c: 375 200 425
a, b, c: 500 0 500
所用时间: 1.65382719039917
```

## 算法效率衡量

### 执行时间反应算法效率

对于同一问题，我们给出了两种解决算法，在两种算法的实现中，我们对程序执行的时间进行了测算，发现两段程序执行的时间相差悬殊（210.9348847 秒相比于 1.653827 秒），

由此我们可以得出结论：实现算法程序的执行时间可以反应出算法的效率，即算法的优劣。

单靠时间值绝对可信吗？假设我们将第二次尝试的算法程序运行在一台配置古老性能低下的计算机中，情况会如何？很可能运行的时间并不会比在我们的电脑中运行算法一的 214.583347 秒快多少。

单纯依靠运行的时间来比较算法的优劣并不一定是客观准确的！程序的运行离不开计算机环境（包括硬件和操作系统），这些客观原因会影响程序运行的速度并反应在程序的执行时间上。那么如何才能客观的评判一个算法的优劣呢？

## 时间复杂度

一般来说，一个算法执行所消耗的时间从理论上是算不出来的，只有通过上机运行才能测试出来。当然，我们也没必要知道一个算法它具体执行的时间是多少，而我们又知道，一个算法花费的时间与算法中语句的执行次数是成正比的。哪个算法语句执行的次数多，它花费的时间就多。

### 【示例】执行次数

```
def test(n):
    count = 0;
    for i in range(count,n):
        for j in range(count,n):
            count+=1
        for k in range(0,2*n):
            count+=1
    icount=10
    while icount>0:
        count+=1
        icount-=1
```

从上面的示例我们可以得到执行次数为： $f(n)=n^2+2*n+10$ 。

对于算法进行特别具体的细致分析虽然很好，但是实践中的实际价值有限。对于算法最重要的是数量级和趋势，这些是分析算法主要的部分。而计量算法基本操作数量的规模函数中哪些常量因子可以忽略不计。

时间复杂度实际上就是一个函数，该函数计算的是执行基本操作的次数。一个算法语句总的执行次数是关于问题规模  $N$  的某个函数，记为  $f(N)$ ,  $N$  称为问题的规模。语句总的执行次数。记为  $T[N]$ , 当  $N$  不断变化时， $T[N]$  也在变化，算法的执行次数的增长速率和  $f(N)$  的增长速率相同。则  $T[N]=O(f(N))$ , 称  $O(f(N))$  为时间复杂度的  $O$  渐进表示法。

分析算法时，存在几种可能的考虑：

算法完成工作最少需要多少基本操作，即最优时间复杂度

算法完成工作最多需要多少基本操作，即最坏时间复杂度

算法完成工作平均需要多少基本操作，即平均时间复杂度

对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值。

对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作。

对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种衡量并没有保证，不是每个计算都能在这个基本操作内完成。而且，对于平均情况的计算，也会因为应用算法的实例分布可能并不均匀而难以计算。

时间复杂度的几条基本计算规则：

- (1) 基本操作，即只有常数项，认为其时间复杂度为  $O(1)$
- (2) 顺序结构，时间复杂度按加法进行计算
- (3) 循环结构，时间复杂度按乘法进行计算
- (4) 分支结构，时间复杂度取最大值
- (5) 判断一个算法的效率时，往往只需要关注操作数量的最高次项，其它次要项和常数项可以忽略
- (6) 在没有特殊说明时，我们所分析的算法的时间复杂度都是指最坏时间复杂度

## 算法分析

### 【示例】第一种解决方式

```
import time
start_time=time.time()
for a in range(1001):
    for b in range(1001):
        for c in range(1001):
            if a+b+c==1000 and a**2+b**2==c**2:
                print('a,b,c:',a,b,c)
end_time=time.time()
print('所用时间: ',(end_time-start_time))
```

时间复杂度：  $T(n) = O(n*n*n) = O(n^3)$

### 【示例】第二种解决方式

```
import time
start_time=time.time()
for a in range(1001):
    for b in range(1001):
        c=1000-a-b
        if a**2+b**2==c**2:
```

```

print('a,b,c:',a,b,c)
end_time=time.time()
print('所用时间: ',(end_time-start_time))

```

时间复杂度:  $T(n) = O(n*n*(1+1)) = O(n*n) = O(n^2)$

由此可见, 我们尝试的第二种算法要比第一种算法的时间复杂度好多的。

## 常见时间复杂度

执行次数函数举例	阶	非正式术语
12	$O(1)$	常数阶
$2n+3$	$O(n)$	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	$O(\log n)$	对数阶
$2n+3n\log_2 n+19$	$O(n\log n)$	$n\log n$ 阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
$2^n$	$O(2^n)$	指数阶

注意, 经常将  $\log_2 n$  (以 2 为底的对数) 简写成  $\log n$

## 常见时间复杂度之间的关系

所消耗的时间从小到大:

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

### 【示例】时间复杂度示例

```

O(5)
O(2n + 1)
O(n^2 + n + 1)
O(3n^3 + 1)

```

## 空间复杂度

一个程序的空间复杂度是指运行完一个程序所需内存的大小。利用程序的空间复杂度, 可以对程序的运行所需要的内存多少有个预先估计。一个程序执行时除了需要存储空间和存储本身所使用的指令、常数、变量和输入数据外, 还需要一些对数据进行操作的工作单元和存储一些为现实计算所需信息的辅助空间。程序执行时所需存储空间包括以下两部分。

(1) 固定部分。这部分空间的大小与输入/输出的数据的个数多少、数值无关。主要包括指令空间（即代码空间）、数据空间（常量、简单变量）等所占的空间。这部分属于静态空间。

(2) 可变空间，这部分空间的主要包括动态分配的空间，以及递归栈所需的空间等。这部分的空间大小与算法有关。

例如：要判断某年是不是闰年，你可能会花一点心思来写一个算法，每给一个年份，就可以通过这个算法计算得到是否闰年的结果。

另外一种方法是，事先建立一个有 2050 个元素的数组，然后把所有的年份按下标的数字对应，如果是闰年，则此数组元素的值是 1，如果不是元素的值则为 0。这样，所谓的判断某一年是否为闰年就变成了查找这个数组某一个元素的值的问题。

第一种方法相比起第二种来说很明显非常节省空间，但每一次查询都需要经过一系列的运算才能知道是否为闰年。第二种方法虽然需要在内存里存储 2050 个元素的数组，但是每次查询只需要一次索引判断即可。这就是通过一笔空间上的开销来换取计算时间开销的小技巧。到底哪一种方法好？其实还是要看你用在什么地方。

一个算法所需的存储空间用  $f(n)$  表示。 $S(n)=O(f(n))$  其中  $n$  为问题的规模， $S(n)$  表示空间复杂度。

### 【示例】空间复杂度

```
def reserse(a,b):
    n=len(a)
    for i in range(n):
        b[i]=a[n-1-i]
```

上方的代码中，当程序调用 `reserse()` 方法时，要分配的内存空间包括：引用 `a`、引用 `b`、局部变量 `n`、局部变量 `i`。因此  $f(n)=4$ ，4 为常量。所以该算法的空间复杂度  $S(n)=O(1)$

通常，我们都是用“时间复杂度”来指运行时间的需求，是用“空间复杂度”指空间需求。当直接要让我们求“复杂度”时，通常指的是时间复杂度。显然对时间复杂度的追求更是属于算法的潮流！

## 排序算法

排序算法（英语：Sorting algorithm）是一种能将一串数据依照特定顺序进行排列的一种算法。

### 排序算法的稳定性

**稳定性：**稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录  $R$  和  $S$ ，且在原本的列表中  $R$  出现在  $S$  之前，在排序过的列表中  $R$  也将会是在  $S$  之前。

当相等的元素是无法分辨的，比如像是整数，稳定性并不是一个问题。然而，假设以下的数对将要以其第一个数字来排序。

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

在这个状况下,有可能产生两种不同的结果,一个是让相等键值的纪录维持相对的次序,而另外一个则没有:

(3, 1) (3, 7) (4, 1) (5, 6) (维持次序)

(3, 7) (3, 1) (4, 1) (5, 6) (次序被改变)

不稳定排序算法可能会在相等的键值中改变纪录的相对次序,但是稳定排序算法从来不会如此。不稳定排序算法可以被特别地实现为稳定。作这件事情的一个方式是人工扩充键值的比较,如此在其他方面相同键值的两个对象间之比较,(比如上面的比较中加入第二个标准:第二个键值的大小)就会被决定使用在原先数据次序中的条目,当作一个同分决赛。然而,要记住这种次序通常牵涉到额外的空间负担。

## 冒泡排序

冒泡排序(英语: Bubble Sort)是一种简单的排序算法。它重复地遍历要排序的数列,一次比较两个元素,如果他们的顺序错误就把他们交换过来。遍历数列的工作是重复地进行直到没有再需要交换,也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

冒泡排序算法的运作如下:

比较相邻的元素。如果第一个比第二个大(升序),就交换他们两个。

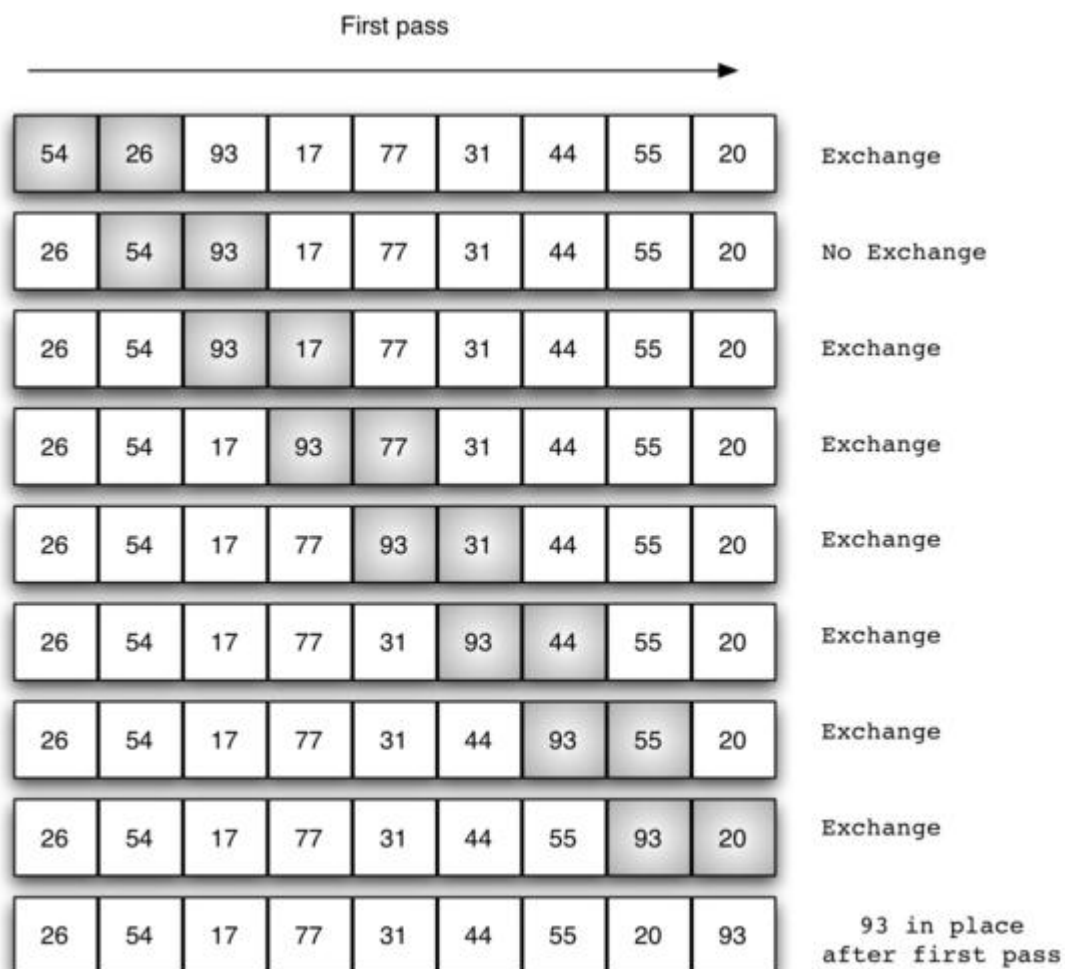
对每一对相邻元素作同样的工作,从开始第一对到结尾的最后一对。这步做完后,最后的元素会是最大的数。

针对所有的元素重复以上的步骤,除了最后一个。

持续每次对越来越少的元素重复上面的步骤,直到没有任何一对数字需要比较。

### 冒泡排序的分析





那么我们需要进行  $n-1$  次冒泡过程，每次对应的比较次数如下图所示：

Pass	Comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
...	...
$n - 1$	1

### 【示例】冒泡排序

```
def bubble_sort(alist):
    for j in range(len(alist) - 1, 0, -1):
        # j 表示每次遍历需要比较的次数，是逐渐减小的
        for i in range(j):
            if alist[i] > alist[i+1]:
```



```
alist[i], alist[i+1] = alist[i+1], alist[i]
```

```
li = [54,26,93,17,77,31,44,55,20]
```

```
bubble_sort(li)
```

```
print(li)
```

执行结果

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

时间复杂度：

最优时间复杂度： $O(n)$ （表示遍历一次发现没有任何可以交换的元素，排序结束。）

最坏时间复杂度： $O(n^2)$

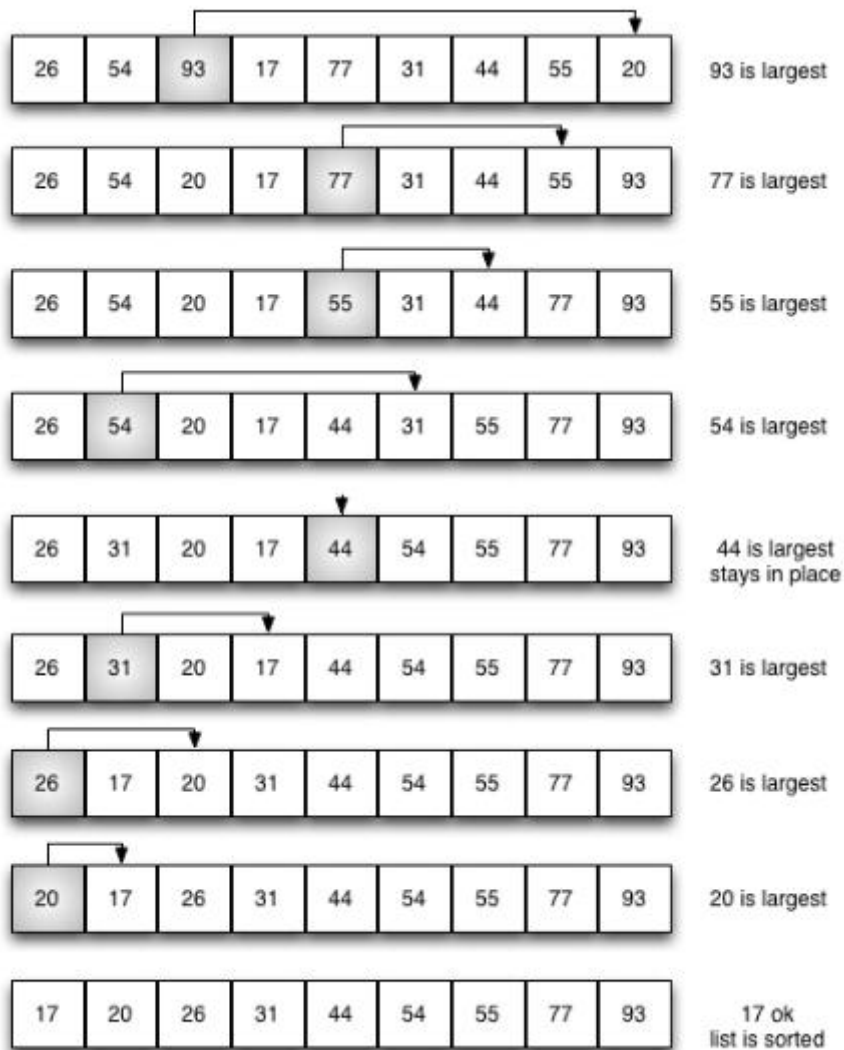
稳定性：稳定

## 选择排序

选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对  $n$  个元素的表进行排序总共进行至多  $n-1$  次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

排序过程：



### 【示例】选择排序

```
def selection_sort(alist):
    n = len(alist)
    # 需要进行 n-1 次选择操作
    for i in range(n-1):
        # 记录最小位置
        min_index = i
        # 从 i+1 位置到末尾选择出最小数据
        for j in range(i+1, n):
            if alist[j] < alist[min_index]:
                min_index = j
        # 如果选择出的数据不在正确位置，进行交换
        if min_index != i:
            alist[i], alist[min_index] = alist[min_index], alist[i]
```

```
alist = [54,226,93,17,77,31,44,55,20]
selection_sort(alist)
print(alist)
```

执行结果

```
[17, 20, 31, 44, 54, 55, 77, 93, 226]
```

时间复杂度:

最优时间复杂度:  $O(n^2)$

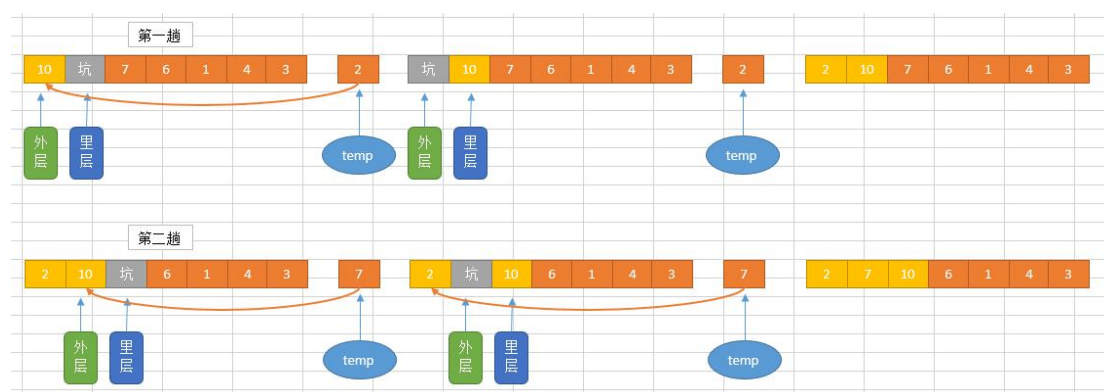
最坏时间复杂度:  $O(n^2)$

稳定性: 不稳定 (考虑升序每次选择最大的情况)

## 插入排序

插入排序 (英语: Insertion Sort) 是一种简单直观的排序算法。它的工作原理是通过构建有序序列, 对于未排序数据, 在已排序序列中从后向前扫描, 找到相应位置并插入。插入排序在实现上, 在从后向前扫描过程中, 需要反复把已排序元素逐步向后挪位, 为最新元素提供插入空间。

插入排序分析:



【示例】插入排序

```
def insert_sort(alist):
    n=len(alist)
    for j in range(1,n):
        i=j
        while i>0:
            if alist[i]<alist[i-1]:
                alist[i],alist[i-1]=alist[i-1],alist[i]
            else:
```

```

        break

    i-=1

if __name__ == '__main__':
    alist=[54, 226, 93, 17, 77, 31, 44, 55, 20]
    print('原数组: ')
    print(alist)
    print('排序后: ')
    insert_sort(alist)
    print(alist)

```

执行结果

```

↑
↓
原数组:
[54, 226, 93, 17, 77, 31, 44, 55, 20]
排序后:
[17, 20, 31, 44, 54, 55, 77, 93, 226]

```

时间复杂度:

最优时间复杂度:  $O(n)$  (升序排列, 序列已经处于升序状态)

最坏时间复杂度:  $O(n^2)$

稳定性: 稳定

## 快速排序

快速排序(英语: Quicksort), 又称为交换排序, 通过一趟排序将要排序的数据分割为独立的两部分。假设要排序的列表是  $A[0] \cdots A[N-1]$ , 首先任意选取一个数据(通常选用列表的第一个数)作为基准数据, 然后将所有比它小的数都放到它左边, 所有比它大的数都放到它右边, 这个过程称为一趟快速排序。值得注意的是, 快速排序不是一种稳定的排序算法, 也就是说, 多个相同的值的相对位置也许会在算法结束时产生变动。

步骤为:

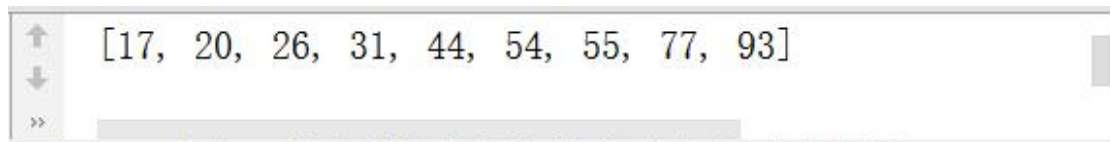
- (1) 设置两个变量 low、high, 排序开始的时候: low=0, high=N-1;
- (2) 以第一个列表元素作为基准数据, 赋值给 mid, 即 mid=A[0];
- (3) 从 high 开始向前搜索, 即由后开始向前搜索(high--), 找到第一个小于 mid 的值 A[high], 将 A[high]和 A[low]的值交换;
- (4) 从 low 开始向后搜索, 即由前开始向后搜索(low++), 找到第一个大于 mid 的 A[low], 将 A[low]和 A[high]的值交换;

(5) 重复第 3、4 步，直到 low=high;

**【示例】快速排序**

```
def quick_sort(alist, start, end):
    """快速排序"""
    # 递归的退出条件
    if start >= end:
        return
    # 设定起始元素为要寻找位置的基准元素
    mid = alist[start]
    # low 为序列左边的由左向右移动的游标
    low = start
    # high 为序列右边的由右向左移动的游标
    high = end
    while low < high:
        # 如果 low 与 high 未重合，high 指向的元素不比基准元素小，则 high 向左移动
        while low < high and alist[high] >= mid:
            high -= 1
        # 将 high 指向的元素放到 low 的位置上
        alist[low] = alist[high]
        # 如果 low 与 high 未重合，low 指向的元素比基准元素小，则 low 向右移动
        while low < high and alist[low] < mid:
            low += 1
        # 将 low 指向的元素放到 high 的位置上
        alist[high] = alist[low]
    # 退出循环后，low 与 high 重合，此时所指位置为基准元素的正确位置
    # 将基准元素放到该位置
    alist[low] = mid
    # 对基准元素左边的子序列进行快速排序
    quick_sort(alist, start, low-1)
    # 对基准元素右边的子序列进行快速排序
    quick_sort(alist, low+1, end)
alist = [54,26,93,17,77,31,44,55,20]
quick_sort(alist,0,len(alist)-1)
print(alist)
```

执行结果



时间复杂度:

最优时间复杂度:  $O(n\log n)$

最坏时间复杂度:  $O(n^2)$

稳定性: 不稳定

## 归并排序

归并排序是采用分治法的一个非常典型的应用。归并排序的思想就是先递归分解数组，再合并数组。

将数组分解最小之后，然后合并两个有序数组，基本思路是比较两个数组的最前面的数，谁小就先取谁，取了后相应的指针就往后移一位。然后再比较，直至一个数组为空，最后把另一个数组的剩余部分复制过来即可。

### 【示例】归并排序

```
def merge_sort(alist):
    if len(alist) <= 1:
        return alist
    # 二分分解
    num = len(alist)//2
    left = merge_sort(alist[:num])
    right = merge_sort(alist[num:])
    # 合并
    return merge(left, right)

def merge(left, right):
    """合并操作，将两个有序数组 left[] 和 right[] 合并成一个大的有序数组"""
    #left 与 right 的下标指针
    l, r = 0, 0
    result = []
    while l < len(left) and r < len(right):
        if left[l] < right[r]:
            result.append(left[l])
            l += 1
        else:
            result.append(right[r])
            r += 1
    result += left[l:]
    result += right[r:]
    return result
```

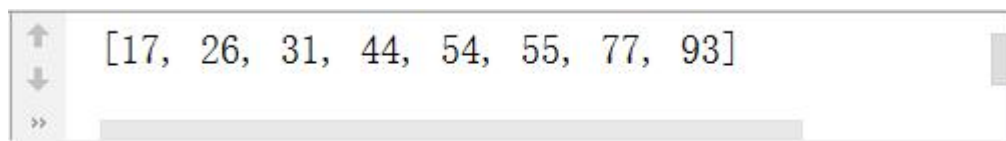
```

        r += 1
    result += left[l:]
    result += right[r:]
    return result

alist = [54,26,93,17,77,31,44,55]
sorted_alist = merge_sort(alist)
print(sorted_alist)

```

执行结果



时间复杂度

最优时间复杂度:  $O(n \log n)$

最坏时间复杂度:  $O(n \log n)$

稳定性: 稳定

← 不同

## 查找算法

### 顺序查找法

最基本的查找技术，过程：从表中的第一个（或最后一个）记录开始，逐个进行记录的关键字和给定值比较，若某个记录的关键字和给定值相等，则查找成功，找到所查的记录；如果直到最后一个（或第一个）记录，其关键字和给定值比较都不等时，则表示没有查到记录，查找不成功。

#### 【示例】顺序查找法

```

# 从 a 列表中查找值 v, 如果找到则返回第一次出现的下标，否则返回 -1
def sequenceSearch(a, v):
    for i in range(len(a)):
        if a[i] == v:
            return i
    return -1

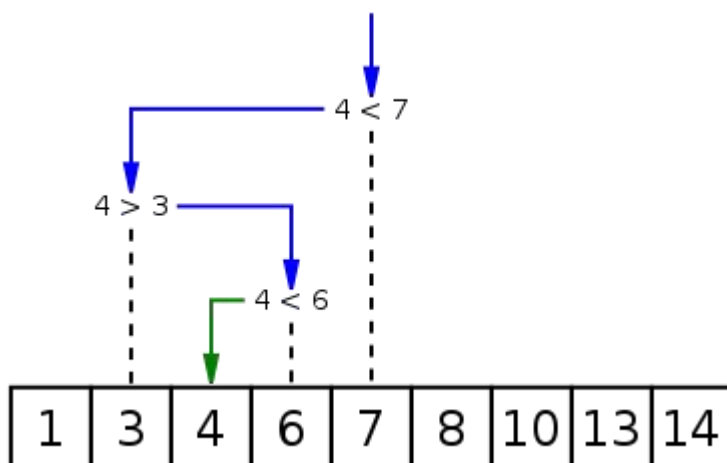
if __name__ == '__main__':
    a = [11, 22, 33, 44, 55, 11]
    v = 22
    index = sequenceSearch(a, v)
    print('查找到的索引为: ', index)

```



## 二分查找法

二分查找又称折半查找，优点是比較次数少，查找速度快，平均性能好；其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。



### 【示例】二分查找法（非递归实现）

```
def binary_search(alist, item):
    first = 0
    last = len(alist) - 1
    while first <= last:
        midpoint = (first + last) // 2
        if alist[midpoint] == item:
            return True
        elif item < alist[midpoint]:
            last = midpoint - 1
        else:
            first = midpoint + 1
    return False

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(binary_search(testlist, 12))
print(binary_search(testlist, 13))
```

执行结果



A screenshot of a Python interactive shell. On the left, there are navigation icons: an up arrow, a down arrow, a search icon, and a prompt icon. The main area displays the text 'False' on the first line and 'True' on the second line.

**【示例】二分查找法（递归实现）**

```
def binary_search(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
            return True
        else:
            if item<alist[midpoint]:
                return binary_search(alist[:midpoint],item)
            else:
                return binary_search(alist[midpoint+1:],item)

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(testlist, 3))
print(binary_search(testlist, 13))
```

执行结果



A screenshot of a Python interactive shell, identical to the one above. It shows the text 'False' on the first line and 'True' on the second line.