

# 1 基本数据结构和算法

这些算法全部自己敲一遍：

## 1.1 链表

链表

双向链表

## 1.2 二叉树

二叉树

二叉查找树

伸展树(splay tree 分裂树)

平衡二叉树 AVL

红黑树

B 树,B+,B\*

R 树

Trie 树(前缀树)

后缀树

最优二叉树(赫夫曼树)

二叉堆 (大根堆, 小根堆)

二项树

二项堆

斐波那契堆(Fibonacci Heap)

## 1.3 哈希表/散列表 (Hash Table)

散列函数

碰撞解决

## 1.4 字符串算法

排序

查找

BF 算法

KMP 算法

BM 算法

正则表达式  
数据压缩

## 1.5 图的算法

图的存储结构和基本操作（建立，遍历，删除节点，添加节点）  
最小生成树  
拓扑排序  
关键路径  
最短路径: Floyd,Dijkstra,bellman-ford,spfa

## 1.6 排序算法

### 1.6.1 交换排序算法

冒泡排序  
插入排序  
选择排序  
希尔排序  
快速排序  
归并排序  
堆排序

### 1.6.2 线性排序算法

桶排序

## 1.7 查找算法

顺序表查找：顺序查找  
有序表查找：二分查找  
分块查找：块内无序，块之间有序；可以先二分查找定位到块，然后再到块中顺序查找  
动态查找：二叉排序树，AVL 树，B-，B+（这里之所以叫 动态查找表，是因为表结构是查找的过程中动态生成的）  
哈希表： $O(1)$

## 1.8 15 个经典基础算法

Hash

快速排序

快速选择 SELECT

BFS/DFS （广度/深度优先遍历）

红黑树 （一种自平衡的二叉查找树）

KMP 字符串匹配算法

DP (动态规划 dynamic programming)

A\*寻路算法： 求解最短路径

Dijkstra: 最短路径算法 （八卦下：Dijkstra 是荷兰的计算机科学家,提出”信号量和 PV 原语“,”解决哲学家就餐问题“,”死锁“也是它提出来的）

遗传算法

启发式搜索

图像特征提取之 SIFT 算法

傅立叶变换

SPFA(shortest path faster algorithm) 单元最短路径算法

## 1.9 海量数据处理

Hash 映射/分而治之

Bitmap

Bloom filter(布隆过滤器)

Trie 树

数据库索引

倒排索引(Inverted Index)

双层桶划分

外排序

simhash 算法

分布处理之 Mapreduce

## 1.10 算法设计思想

迭代法

穷举搜索法

递推法

动态规划

贪心算法

回溯

分治算法

### 1.11 算法问题选编

这是一个算法题目合集,题目是我从网络和书籍之中整理而来,部分题目已经做了思路整理。  
问题分类包括:

字符串  
堆和栈  
链表  
数值问题  
数组和数列问题  
矩阵问题  
二叉树  
图  
海量数据处理  
智力思维训练  
系统设计  
还有部分来自算法网站和书籍:

九度 OJ  
leetcode  
剑指 offer  
开源项目中的算法

YYCache  
cocos2d-objc  
...

## 2 程序员必须知道的 n 大基础实用算法

来源: <http://kb.cnblogs.com/page/210687/>

排序方法	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n \log_2 n)$	$O(n \log_2 n)$		$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定	较复杂

blog.csdn.com/whuslei

## 2.1 快速排序算法

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序  $n$  个项目要  $O(n \log n)$  次比较。在最坏状况下则需要  $O(n^2)$  次比较，但这种状况并不常见。事实上，快速排序通常明显比其他  $O(n \log n)$  算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

快速排序使用分治法（Divideandconquer）策略来把一个串行（list）分为两个子串行（sub-lists）。

算法步骤：

1 从数列中挑出一个元素，称为“基准”（pivot），

2 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

3 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会退出，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

//算法的精髓就是 i 的位置后面的所有数都比 pivot 大，前面的数都比 pivot 小。

也就是当 j 指向的数比 pivot 大时，不交换，比 pivot 小时，就需要把这个数交换到 i 指向的位置，并移动 i

```
#include <stdio.h>

int partition(int *arr,int low,int high)
{
    int pivot=arr[high];
    int i=low-1;
    int j,tmp;
    for(j=low;j<high;++j)
        if(arr[j]<pivot){
            tmp=arr[++i];
            arr[i]=arr[j];
            arr[j]=tmp;
        }
    tmp=arr[i+1];
    arr[i+1]=arr[high];
    arr[high]=tmp;
    return i+1;
}

void quick_sort(int *arr,int low,int high)
{
    if(low<high){
        int mid=partition(arr,low,high);
        quick_sort(arr,low,mid-1);
        quick_sort(arr,mid+1,high);
    }
}

//test
int main()
{
    int arr[10]={ 1,4,6,2,5,8,7,6,9,12};
    quick_sort(arr,0,9);
    int i;
    for(i=0;i<10;++i)
        printf("%d ",arr[i]);
}
```

## 算法复杂度

最坏情况下的快排时间复杂度：

最坏情况发生在划分过程产生的两个区域分别包含 n-1 个元素和一个 0 元素的时候，

即假设算法每一次递归调用过程中都出现了，这种划分不对称。那么划分的代价为  $O(n)$ ，

因为对一个大小为  $0$  的数组递归调用后，返回  $T(0) = O(1)$ 。

估算法的运行时间可以递归的表示为：

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n).$$

可以证明为  $T(n) = O(n^2)$ 。

因此，如果在算法的每一层递归上，划分都是最大程度不对称的，那么算法的运行时间就是  $O(n^2)$ 。

**最快情况下快排时间复杂度：**

最快情况下，即 `PARTITION` 可能做的最平衡的划分中，得到的每个子问题都不能大于  $n/2$ 。

因为其中一个子问题的大小为  $\lfloor n/2 \rfloor$ 。另一个子问题的大小为  $\lfloor n/2 \rfloor - 1$ 。

在这种情况下，快速排序的速度要快得多：

$T(n) \leq 2T(n/2) + O(n)$ 。可以证得， $T(n) = O(n \lg n)$ 。

## 2.2 堆排序算法

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆排序的平均时间复杂度为  $O(n \lg n)$ 。

算法步骤：

创建一个堆  $H[0..n-1]$

把堆首（最大值）和堆尾互换

3.把堆的尺寸缩小 1，并调用 `shift_down(0)`，目的是把新的数组顶端数据调整到相应位置

4.重复步骤 2，直到堆的尺寸为 1

```
=====
#include <stdlib.h>
#define PARENT(i) (i)/2
#define LEFT(i) 2*(i)+1
#define RIGHT(i) 2*(i)+1

void swap(int *a, int *b)
{
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}

void max_heapify(int *arr, int index, int len)
```

```

{
    int l = LEFT(index);
    int r = RIGHT(index);
    int largest;
    if (l < len && arr[l] > arr[index])
        largest = l;
    else
        largest = index;
    if (r < len && arr[r] > arr[largest])
        largest = r;
    if (largest != index) { //将最大元素提升，并递归
        swap(&arr[largest], &arr[index]);
        max_heapify(arr, largest, len);
    }
}

void build_maxheap(int *arr, int len)
{
    int i;
    if (arr == NULL || len <= 1)
        return;
    for (i = len / 2 + 1; i >= 0; --i) // (len/2+1) means the last point who has
children.
        max_heapify(arr, i, len);
}

void heap_sort(int *arr, int len)
{
    int i;
    if (arr == NULL || len <= 1)
        return;
    build_maxheap(arr, len);

    for (i = len - 1; i >= 1; --i) {
        swap(&arr[0], &arr[i]);
        max_heapify(arr, 0, --len);
    }
}

int main()
{
    int arr[10] = { 1, 4, 6, 2, 5, 8, 7, 6, 9, 12 };
    int i;
    heap_sort(arr, 10);
    for (i = 0; i < 10; ++i)

```



```
        printf("%d ", arr[i]);  
    system("pause");  
}
```

=====

## 2.3 归并排序

归并排序（**Mergesort**，台湾译作：合并排序）是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（**DivideandConquer**）的一个非常典型的应用。

算法步骤：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤 3 直到某一指针达到序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

## 2.4 选择排序

选择排序与冒泡排序非常的相似，冒泡是将最大的放到最后面，而选择是将最小的放在最前面。不过理念还是有些不同。选择是一次就会选出最小的。而冒泡只是比较相邻的元素。会存在多次交换。性能没那么好。

假定存在数组 `array[0..n-1]`，选择排序的核心思想是：

第  $i$  趟排序是从后面的  $n - i + 1$  ( $i = 1, 2, 3, 4, \dots, n - 1$ ) 个元素中选择一个值最小的元素与该  $n - i + 1$  个元素的最前门的那个元素交换位置，即与整个序列的第  $i$  个元素交换位置。如此下去，直到  $i = n - 1$ ，排序结束。

也可描述为：

每一趟排序从序列中未排序的那些元素中选择一个值最小的元素，然后将其与这些未排序的元素的第一个元素交换位置。

特点：

1. **算法**完成需要  $n - 1$  趟排序，按照算法的描述， $n - 1$  趟排序之后数组中的前  $n - 1$  个元素已经处于相应的位置，第  $n$  个元素也处于相应的位置上。
2. 第  $i$  趟排序，实际上就是需要将数组中第  $i$  个元素放置到数组的合适位置，这里需要一个临时变量  $j$  来遍历序列中未排序的那些元素，另一临时变量  $d$  来记录未排序的那些元素中值最小的元素的下标值，
3. 一趟遍历开始时，令  $d = i$ ，假定未排序序列的第一个元素就是最小的元素，遍历完成后，变量  $d$  所对应的值就是值最小的元素，判断  $d$  是否是未排序序列的第一个元素，如果是，则不需要交换元素，如果不是，则需要交换 `array[d]` 和 `array[i]`。

4. 此方法是不稳定排序算法，可对数组{a1 = 49, a2 = 38, a3 = 65, a4 = 49, a5 = 12, a6 = 42} 排序就可以看出，排序完成后 a1 和 a4 的相对位置改变了。
5. 此方法移动元素的次数比较少，但是不管序列中元素初始排列状态如何，第 i 趟排序都需要进行 n - i 次元素之间的比较，因此总的比较次数为  $1 + 2 + 3 + 4 + 5 + \dots + n - 1 = n(n-1)/2$ ，时间复杂度是  $O(n^2)$ 。

```
=====
void selectSort(int array[], int n)
{
    int i, j, d;
    int temp;
    for (i = 0; i < n - 1; ++i)
    {
        d = i;
        for (j = i + 1; j < n; ++j)
            if (array[j] < array[d])
                d = j;

        if (d != i)
        {
            temp = array[d];
            array[d] = array[i];
            array[i] = temp;
        }
    }
}
```

```
=====
```

## 2.5 冒泡排序

原理是临近的数字两两进行比较,按照从小到大或者从大到小的顺序进行交换,

这样一趟过去后,最大或最小的数字被交换到了最后一位,

然后再从头开始进行两两比较交换,直到倒数第二位时结束,其余类似看例子

例子为从小到大排序,

原始待排序数组 | 6 | 2 | 4 | 1 | 5 | 9 |

第一趟排序(外循环)

第一次两两比较  $6 > 2$  交换(内循环)

交换前状态| 6 | 2 | 4 | 1 | 5 | 9 |

交换后状态| 2 | 6 | 4 | 1 | 5 | 9 |

第二次两两比较,  $6 > 4$  交换

交换前状态| 2 | 6 | 4 | 1 | 5 | 9 |

交换后状态| 2 | 4 | 6 | 1 | 5 | 9 |

第三次两两比较,  $6 > 1$  交换

交换前状态| 2 | 4 | 6 | 1 | 5 | 9 |

交换后状态| 2 | 4 | 1 | 6 | 5 | 9 |

第四次两两比较,  $6 > 5$  交换

交换前状态| 2 | 4 | 1 | 6 | 5 | 9 |

交换后状态| 2 | 4 | 1 | 5 | 6 | 9 |

第五次两两比较,  $6 < 9$  不交换

交换前状态| 2 | 4 | 1 | 5 | 6 | 9 |

交换后状态| 2 | 4 | 1 | 5 | 6 | 9 |

第二趟排序(外循环)

第一次两两比较  $2 < 4$  不交换

交换前状态 | 2 | 4 | 1 | 5 | 6 | 9 |

交换后状态 | 2 | 4 | 1 | 5 | 6 | 9 |

第二次两两比较,  $4 > 1$  交换

交换前状态 | 2 | 4 | 1 | 5 | 6 | 9 |

交换后状态 | 2 | 1 | 4 | 5 | 6 | 9 |

第三次两两比较,  $4 < 5$  不交换

交换前状态 | 2 | 1 | 4 | 5 | 6 | 9 |

交换后状态 | 2 | 1 | 4 | 5 | 6 | 9 |

第四次两两比较,  $5 < 6$  不交换

交换前状态 | 2 | 1 | 4 | 5 | 6 | 9 |

交换后状态 | 2 | 1 | 4 | 5 | 6 | 9 |

第三趟排序(外循环)

第一次两两比较  $2 > 1$  交换

交换后状态 | 2 | 1 | 4 | 5 | 6 | 9 |

交换后状态 | 1 | 2 | 4 | 5 | 6 | 9 |

第二次两两比较,  $2 < 4$  不交换

交换后状态 | 1 | 2 | 4 | 5 | 6 | 9 |

交换后状态 | 1 | 2 | 4 | 5 | 6 | 9 |

第三次两两比较,  $4 < 5$  不交换

交换后状态 | 1 | 2 | 4 | 5 | 6 | 9 |

交换后状态 | 1 | 2 | 4 | 5 | 6 | 9 |

第四趟排序(外循环)无交换

第五趟排序(外循环)无交换

排序完毕,输出最终结果 1 2 4 5 6 9

=====

```
void bubble_sort_v2(int unsorted[], int len)
{
    int i, j;
    for(i = 0; i < len - 1; ++i) {
        for(j = 0; j < (len - 1 - i); ++j) {
            if(unsorted[j] > unsorted[j + 1]) {
                unsorted[j] = unsorted[j] ^ unsorted[j + 1]; // use XOR to exchange
                unsorted[j + 1] = unsorted[j + 1] ^ unsorted[j];
                unsorted[j] = unsorted[j] ^ unsorted[j + 1];
            }
        }
    }
}
```

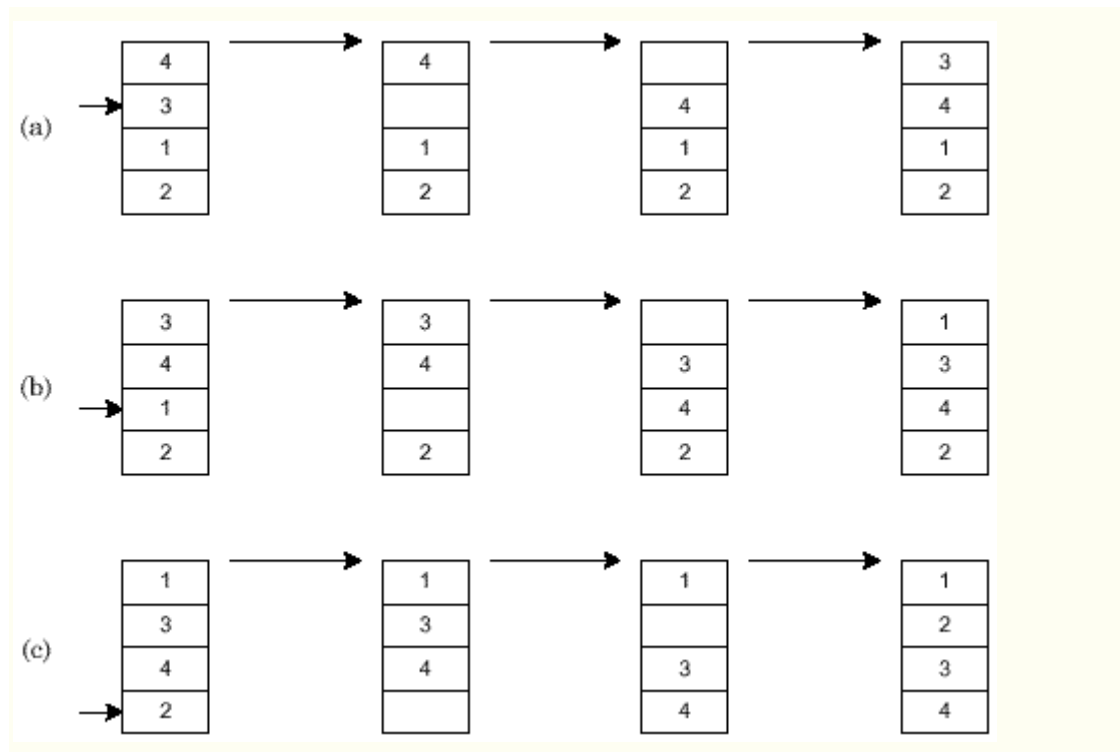
=====

## 2.6 插入排序

插入排序就是每一步都将一个待排数据按其大小插入到已经排序的数据中的适当位置,直到全部插入完毕。

插入排序方法分直接插入排序和折半插入排序两种,这里只介绍直接插入排序,折半插入排序留到“查找”内容中进行。

图 1 演示了对 4 个元素进行直接插入排序的过程,共需要(a),(b),(c)三次插入。



=====

```
static void insertion_sort(int unsorted[], int len)
{
    for (int i = 1; i < len; i++)
    {
        if (unsorted[i - 1] > unsorted[i])
        {
            int temp = unsorted[i];
            int j = i;
            while (j > 0 && unsorted[j - 1] > temp)
            {
                unsorted[j] = unsorted[j - 1];
                j--;
            }
            unsorted[j] = temp;
        }
    }
}
```

=====

## 2.7 希尔排序

希尔(Shell)排序又称为**缩小增量排序**，它是一种**插入排序**。它是**直接插入排序算法的一种威力加强版**。

该方法因 DL. Shell 于 1959 年提出而得名。

希尔排序的**基本思想**是：

把记录按**步长 gap** 分组，对每组记录采用**直接插入排序**方法进行排序。

随着**步长逐渐减小**，所分成的组包含的记录越来越多，当步长的值减小到 **1** 时，整个数据合成一组，构成一组有序记录，则完成排序。

我们来通过演示图，更深入的理解一下这个过程。

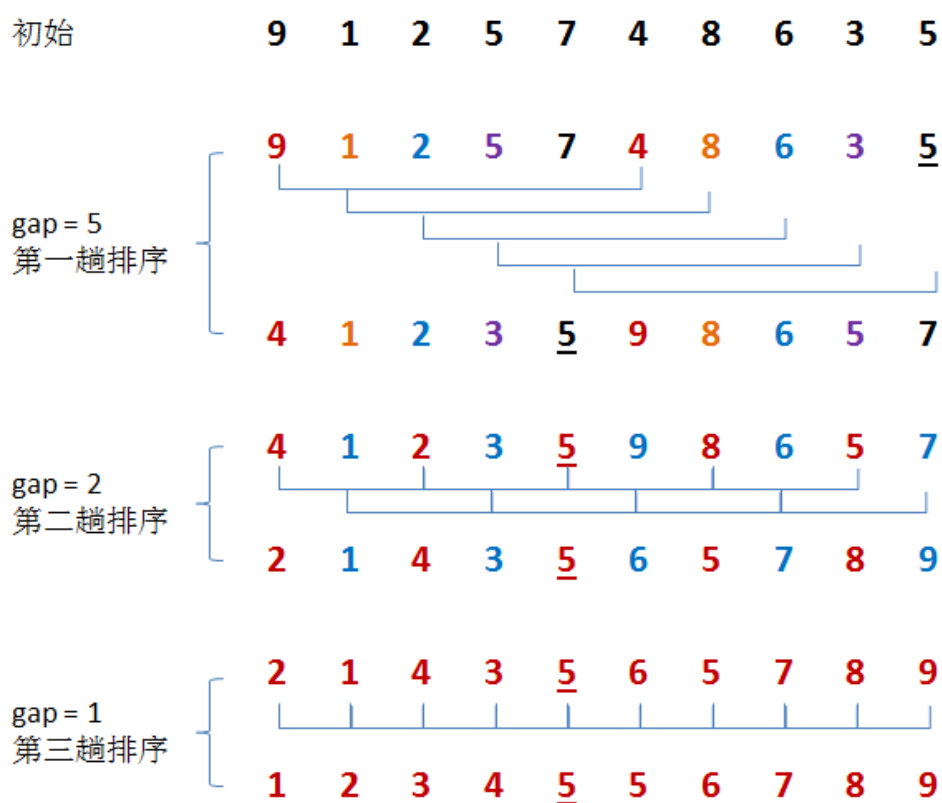


图-希尔排序示例图

Victor Zhang

在上面这幅图中：

初始时，有一个大小为 10 的无序序列。

在**第一趟排序**中，我们不妨设  $gap1 = N / 2 = 5$ ，即相隔距离为 5 的元素组成一组，可以分为 5 组。

接下来，按照直接插入排序的方法对每个组进行排序。

在**第二趟排序**中，我们把上次的  $gap$  缩小一半，即  $gap2 = gap1 / 2 = 2$  (取整数)。这样每相隔距离为 2 的元素组成一组，可以分为 2 组。

按照直接插入排序的方法对每个组进行排序。

在**第三趟排序**中，再次把  $gap$  缩小一半，即  $gap3 = gap2 / 2 = 1$ 。这样相隔距离为 1 的元素组成一组，即只有一组。

按照直接插入排序的方法对每个组进行排序。此时，**排序已经结束**。

需要注意一下的是，图中有两个相等数值的元素 **5** 和 **5**。我们可以清楚的看到，在排序过程中，**两个元素位置交换了**。

所以，希尔排序是不稳定的算法。

## 2.8 顺序查找算法

**说明：顺序查找适合于存储结构为顺序存储或链接存储的线性表。**

基本思想：顺序查找也称为线形查找，**属于无序查找算法**。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值  $k$  相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于  $k$  的结点，表示查找失败。


复杂度分析：

查找成功时的平均查找长度为：（假设每个数据元素的概率相等）  $ASL = 1/n(1+2+3+\cdots+n) = (n+1)/2$ ；

当查找不成功时，需要  $n+1$  次比较，时间复杂度为  $O(n)$ ；


所以，顺序查找的时间复杂度为  $O(n)$ 。

**C++实现源码：**

```

//顺序查找
int SequenceSearch(int a[], int value, int n)
{
    int i;
```



```
for(i=0; i<n; i++)
    if(a[i]==value)
        return i;
return -1;
}
```



## 2.9 二分查找算法


说明：元素必须是有序的，如果无序的则要先进行排序操作。

基本思想：也称为折半查找，属于有序查找算法。用给定值  $k$  先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据  $k$  与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

**复杂度分析：最坏情况下，关键词比较次数为  $\log_2(n+1)$ ，且期望时间复杂度为  $O(\log_2 n)$ ；**

注：折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。——《大话数据结构》

### C++实现源码：



```
//二分查找（折半查找），版本1
int BinarySearch1(int a[], int value, int n)
{
    int low, high, mid;
    low = 0;
    high = n-1;
    while(low<=high)
    {
        mid = (low+high)/2;
        if(a[mid]==value)
            return mid;
        if(a[mid]>value)
            high = mid-1;
        if(a[mid]<value)
            low = mid+1;
    }
    return -1;
}
```

```
//二分查找，递归版本
int BinarySearch2(int a[], int value, int low, int high)
{
    int mid = low+(high-low)/2;
    if(a[mid]==value)
        return mid;
    if(a[mid]>value)
        return BinarySearch2(a, value, low, mid-1);
    if(a[mid]<value)
        return BinarySearch2(a, value, mid+1, high);
}
```

## 2.10 插值查找算法（二分查找的变种）

在介绍插值查找之前，首先考虑一个新问题，为什么上述算法一定要是折半，而不是折四分之一或者折更多呢？

打个比方，在英文字典里面查“apple”，你下意识翻开字典是翻前面的书页还是后面的书页呢？如果再让你查“zoo”，你又怎么查？很显然，这里你绝对不会是从中间开始查起，而是有一定目的的往前或往后翻。

同样的，比如要在取值范围 1 ~ 10000 之间 100 个元素从小到大均匀分布的数组中查找 5，我们自然会考虑从数组下标较小的开始查找。

经过以上分析，折半查找这种查找方式，不是自适应的（也就是说是傻瓜式的）。二分查找中查找点计算如下：

$mid = (low + high) / 2$ ，即  $mid = low + 1/2 * (high - low)$ ；

通过类比，我们可以将查找的点改进为如下：

$mid = low + (key - a[low]) / (a[high] - a[low]) * (high - low)$ ，

也就是将上述的比例参数  $1/2$  改进为自适应的，根据关键字在整个有序表中所处的位置，让  $mid$  值的变化更靠近关键字  $key$ ，这样也就间接地减少了比较次数。

基本思想：基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，差值查找也属于有序查找。

注：对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好的多。反之，数组中如果分布非常不均匀，那么插值查找未必是很合适的选择。

复杂度分析：查找成功或者失败的时间复杂度均为  $O(\log_2(\log_2 n))$ 。

### C++实现源码：

```
//插值查找
int InsertionSearch(int a[], int value, int low, int high)
{
    int mid = low + (value - a[low]) / (a[high] - a[low]) * (high - low);
```

```

if(a[mid]==value)
    return mid;
if(a[mid]>value)
    return InsertionSearch(a, value, low, mid-1);
if(a[mid]<value)
    return InsertionSearch(a, value, mid+1, high);
}

```



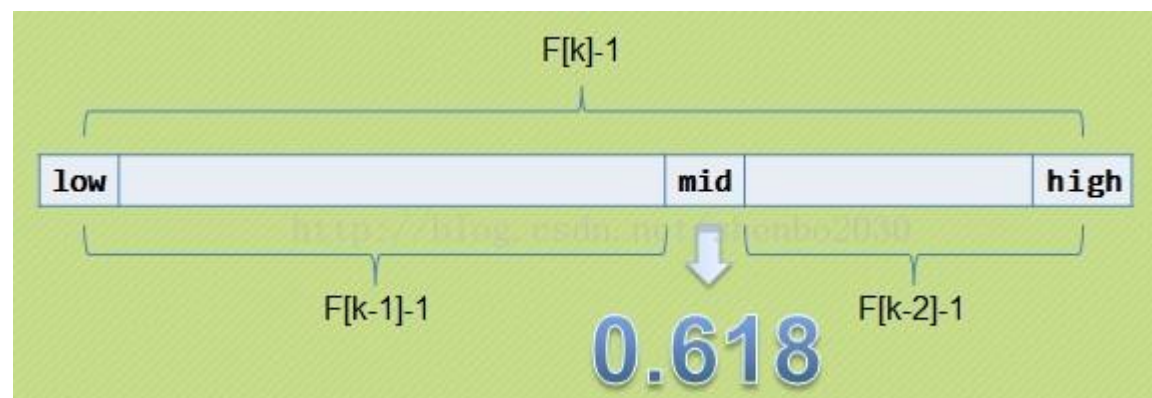
## 2.11 斐波拉契查找算法（二分查找的变种）

在介绍斐波那契查找算法之前，我们先介绍一下很它紧密相连并且大家都熟知的一个概念——黄金分割。

黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为 1:0.618 或 1.618:1。

0.618 被公认为最具有审美意义的比例数字，这个数值的作用不仅仅体现在诸如绘画、雕塑、音乐、建筑等艺术领域，而且在管理、工程设计等方面也有着不可忽视的作用。因此被称为黄金分割。

大家记不记得斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89……（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近 0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。



**基本思想：**也是二分查找的一种提升算法，通过运用黄金比例的概念在数列中选择查找点进行查找，提高查找效率。同样地，斐波那契查找也属于一种有序查找算法。

相对于折半查找，一般将待比较的 key 值与第  $mid = (low + high) / 2$  位置的元素比较，比较结果分三种情况：

- 1) 相等，mid 位置的元素即为所求
- 2)  $>$ ， $low = mid + 1$ ;
- 3)  $<$ ， $high = mid - 1$ 。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小 1，及  $n = F(k) - 1$ ;

开始将 k 值与第  $F(k-1)$  位置的记录进行比较(及  $mid = low + F(k-1) - 1$ ), 比较结果也分为三种

- 1) 相等，mid 位置的元素即为所求

2)  $>$ ,  $low=mid+1, k-=2$ ;

说明:  $low=mid+1$  说明待查找的元素在  $[mid+1, high]$  范围内,  $k-=2$  说明范围  $[mid+1, high]$  内的元素个数为  $n-(F(k-1))=Fk-1-F(k-1)=Fk-F(k-1)-1=F(k-2)-1$  个, 所以可以递归的应用斐波那契查找。

3)  $<$ ,  $high=mid-1, k-=1$ 。

说明:  $low=mid+1$  说明待查找的元素在  $[low, mid-1]$  范围内,  $k-=1$  说明范围  $[low, mid-1]$  内的元素个数为  $F(k-1)-1$  个, 所以可以递归 的应用斐波那契查找。

**复杂度分析: 最坏情况下, 时间复杂度为  $O(\log 2n)$ , 且其期望复杂度也为  $O(\log 2n)$ 。**

**C++实现源码:**

```
// 斐波那契查找.cpp

#include "stdafx.h"
#include <memory>
#include <iostream>
using namespace std;

const int max_size=20;//斐波那契数组的长度

/*构造一个斐波那契数组*/
void Fibonacci(int * F)
{
    F[0]=0;
    F[1]=1;
    for(int i=2;i<max_size;++i)
        F[i]=F[i-1]+F[i-2];
}

/*定义斐波那契查找法*/
int FibonacciSearch(int *a, int n, int key) //a 为要查找的数组, n 为要查找的数组长度, key 为要查找的关键字
{
    int low=0;
    int high=n-1;

    int F[max_size];
    Fibonacci(F); //构造一个斐波那契数组 F

    int k=0;
    while(n>F[k]-1) //计算 n 位于斐波那契数列的位置
        ++k;
```

```

int * temp;//将数组 a 扩展到 F[k]-1 的长度
temp=new int [F[k]-1];
memcpy(temp, a, n*sizeof(int));

for(int i=n;i<F[k]-1;++i)
    temp[i]=a[n-1];

while(low<=high)
{
    int mid=low+F[k-1]-1;
    if(key<temp[mid])
    {
        high=mid-1;
        k-=1;
    }
    else if(key>temp[mid])
    {
        low=mid+1;
        k-=2;
    }
    else
    {
        if(mid<n)
            return mid; //若相等则说明 mid 即为查找到的位置
        else
            return n-1; //若 mid>=n 则说明是扩展的数值, 返回 n-1
    }
}
delete [] temp;
return -1;
}

int main()
{
    int a[] = {0, 16, 24, 35, 47, 59, 62, 73, 88, 99};
    int key=100;
    int index=FibonacciSearch(a, sizeof(a)/sizeof(int), key);
    cout<<key<<" is located at:"<<index;
    return 0;
}

```



## 2.12 树表查找算法

<http://www.cnblogs.com/maybe2030/p/4715035.html#top>

有点复杂。慢慢研究。

## 2.13 分块查找算法

## 2.14 哈希查找算法

## 2.15 BFPRT(线性查找算法)

BFPRT 算法解决的问题十分经典，即从某  $n$  个元素的序列中选出第  $k$  大（第  $k$  小）的元素，通过巧妙的分析，BFPRT 可以保证在最坏情况下仍为线性时间复杂度。该算法的思想与快速排序思想相似，当然，为使得算法在最坏情况下，依然能达到  $O(n)$  的时间复杂度，五位算法作者做了精妙的处理。

算法步骤：

1. 将  $n$  个元素每 5 个一组，分成  $n/5$  (上界) 组。
2. 取出每一组的中位数，任意排序方法，比如插入排序。
3. 递归的调用 selection 算法查找上一步中所有中位数的中位数，设为  $x$ ，偶数个中位数的情况下设定为选取中间小的一个。
4. 用  $x$  来分割数组，设小于等于  $x$  的个数为  $k$ ，大于  $x$  的个数即为  $n-k$ 。
5. 若  $i=k$ ，返回  $x$ ；若  $i < k$ ，在小于  $x$  的元素中递归查找第  $i$  小的元素；若  $i > k$ ，在大于  $x$  的元素中递归查找第  $i-k$  小的元素。

终止条件： $n=1$  时，返回的即是  $i$  小元素

## 2.16 DFS（深度优先搜索）

深度优先搜索算法（Depth-First-Search），是搜索算法的一种。它沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点  $v$  的所有边都被探寻过，搜索将回溯到发现节点  $v$  的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。DFS 属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。一般用堆数据结构来辅助实现 DFS 算法。

深度优先遍历图算法步骤：

1. 访问顶点  $v$ ；
2. 依次从  $v$  的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和  $v$  有路径相通的顶点都被访问；
3. 若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

上述描述可能比较抽象，举个实例：

DFS 在访问图中某一起始顶点  $v$  后，由  $v$  出发，访问它的任一邻接顶点  $w_1$ ；再从  $w_1$  出发，访问与  $w_1$  邻接但还没有访问过的顶点  $w_2$ ；然后再从  $w_2$  出发，进行类似的访问，…如此进行下去，直至到达所有的邻接顶点都被访问过的顶点  $u$  为止。

接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

## 2.17 BFS(广度优先搜索)

广度优先搜索算法 (Breadth-First-Search)，是一种图形搜索算法。简单的说，BFS 是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止。BFS 同样属于盲目搜索。一般用队列数据结构来辅助实现 BFS 算法。

算法步骤：

1. 首先将根节点放入队列中。
2. 从队列中取出第一个节点，并检验它是否为目标。

如果找到目标，则结束搜寻并回传结果。

否则将它所有尚未检验过的直接子节点加入队列中。

3. 若队列为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

4. 重复步骤 2。

## 2.18 Dijkstra 算法

戴克斯特拉算法 (Dijkstra's algorithm) 是由荷兰计算机科学家艾兹赫尔·戴克斯特拉提出。迪科斯彻算法使用了广度优先搜索解决非负权有向图的单源最短路径问题，算法最终得到一个最短路径树。该算法常用于路由算法或者作为其他图算法的一个子模块。

该算法的输入包含了一个有权重的有向图  $G$ ，以及  $G$  中的一个来源顶点  $S$ 。我们以  $V$  表示  $G$  中所有顶点的集合。每一个图中的边，都是两个顶点所形成的有序元素对。 $(u,v)$  表示从顶点  $u$  到  $v$  有路径相连。我们以  $E$  表示  $G$  中所有边的集合，而边的权重则由权重函数  $w:E \rightarrow [0, \infty]$  定义。因此， $w(u,v)$  就是从顶点  $u$  到顶点  $v$  的非负权重 (weight)。边的权重可以想像成两个顶点之间的距离。任两点间路径的权重，就是该路径上所有边的权重总和。已知有  $V$  中有顶点  $s$  及  $t$ ，Dijkstra 算法可以找到  $s$  到  $t$  的最低权重路径(例如，最短路径)。这个算法也可以在一个图中，找到从一个顶点  $s$  到任何其他顶点的最短路径。对于不含负权的有向图，Dijkstra 算法是目前已知的最快的单源最短路径算法。

算法步骤：

1. 初始时令  $S=\{V_0\}$ ,  $T=\{\text{其余顶点}\}$ ， $T$  中顶点对应的距离值

若存在  $\langle V_0, V_i \rangle$ ， $d(V_0, V_i)$  为  $\langle V_0, V_i \rangle$  弧上的权值

若不存在  $\langle V_0, V_i \rangle$ ， $d(V_0, V_i)$  为  $\infty$

2. 从  $T$  中选取一个其距离值为最小的顶点  $W$  且不在  $S$  中，加入  $S$



3.对其余  $T$  中顶点的距离值进行修改：若加进  $W$  作中间顶点，从  $v_0$  到  $v_i$  的距离值缩短，则修改此距离值  
重复上述步骤 2、3，直到  $S$  中包含所有顶点，即  $W=v_i$  为止

## 2.19 动态规划算法

动态规划（Dynamic programming）是一种在数学、计算机科学和经济学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再合并子问题的解以得出原问题的解。通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

关于动态规划最经典的问题当属背包问题。

算法步骤：

1.最优子结构性质。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。

2.子问题重叠性质。子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率。

## 2.20 朴素贝叶斯分类算法

朴素贝叶斯分类算法是一种基于贝叶斯定理的简单概率分类算法。贝叶斯分类的基础是概率推理，就是在各种条件的存在不确定，仅知其出现概率的情况下，如何完成推理和决策任务。概率推理是与确定性推理相对应的。而朴素贝叶斯分类器是基于独立假设的，即假设样本每个特征与其他特征都不相关。

朴素贝叶斯分类器依靠精确的自然概率模型，在有监督学习的样本集中能获得非常好的分类效果。在许多实际应用中，朴素贝叶斯模型参数估计使用最大似然估计方法，换言之朴素贝叶斯模型能工作并没有用到贝叶斯概率或者任何贝叶斯模型。

尽管是带着这些朴素思想和过于简单化的假设，但朴素贝叶斯分类器在很多复杂的现实情形中仍能够取得相当好的效果。

# 3 推荐阅读



## 刷题必备

《剑指 offer》  
《编程之美》  
《结构之法:面试和算法心得》  
《算法谜题》 都是思维题

## 基础

《编程珠玑》 **Programming Pearls**  
《编程珠玑(续)》  
《数据结构与算法分析》  
《Algorithms》 这本近千页的书只有 6 章,其中四章分别是排序,查找,图,字符串,足见介绍细致

## 算法设计

《算法设计与分析基础》  
《算法引论》 告诉你如何创造算法 断货  
《Algorithm Design Manual》 算法设计手册 红皮书

《算法导论》 是一本对算法介绍比较全面的经典书籍

《Algorithms on Strings,Trees and Sequences》  
《Advanced Data Structures》 各种诡异高级的数据结构和算法 如元胞自动机、斐波纳契堆、线段树 600 块

## 延伸阅读

《深入理解计算机系统》  
《TCP/IP 详解三卷》  
《UNIX 网络编程二卷》  
《UNIX 环境高级编程: 第 2 版》  
  
《The practice of programming》 Brian Kernighan 和 Rob Pike  
《writing efficient programs》 优化  
《The science of programming》 证明代码段的正确性 800 块一本