

十五个经典算法研究与总结

作者: *July*

时间: 2010 年 12 月末-2011 年 12 月。

微博: <http://weibo.com/julyweibo>

出处: http://blog.csdn.net/v_JULY_v

声明: 版权所有, 侵权必究。

文档制作者: 花明月暗 & 有鱼网 <http://www.youyur.com/> C E O 吴超

前言:

本人的原创作品[经典算法研究系列](#),自从 10 年 12 月末至 11 年 12 月,写了近一年。可以这么说,开博头俩个月一直在整理微软等公司的面试题,而后的四个月至今,则断断续续,除了继续微软面试 100 题系列,和程序员编程艺术系列之外,便在写这经典算法研究系列和相关算法文章。

本经典算法研究系列,涵盖 A*.Dijkstra.DP.BFS/DFS.红黑树.KMP.遗传.启发式搜索.图像特征提取 SIFT.傅立叶变换.Hash.快速排序.SPFA.快递选择 SELECT 等 15 个经典基础算法,共计 31 篇文章,包括算法理论的研究与阐述,及其编程的具体实现。很多个算法都后续写了续集,如第二个算法: Dijkstra 算法,便写了 4 篇文章; sift 算法包括其编译及实现,写了 5 篇文章;而红黑树系列,则更是最后写了 6 篇文章,成为了国内最为经典的红黑树教程。

OK,任何人有任何问题,欢迎随时在 blog 上留言评论,或来信: zhoulei0907@yahoo.cn 批评指正。谢谢。以下已经写了的 15 个经典算法集锦,算是一个目录+索引,共计 31 篇文章:

十五个经典算法研究集锦+目录

一、A*搜索算法

一(续)、A*, Dijkstra, BFS 算法性能比较及 A*算法的应用

二、Dijkstra 算法初探

二(续)、彻底理解 Dijkstra 算法

二(再续)、Dijkstra 算法+fibonacci 堆的逐步 c 实现

二(三续)、Dijkstra 算法+Heap 堆的完整 c 实现源码

三、动态规划算法

四、BFS 和 DFS 优先搜索算法

五、教你透彻了解红黑树 (红黑数系列六篇文章之其中两篇)

五(续)、红黑树算法的实现与剖析

六、教你初步了解 KMP 算法、updated (KMP 算法系列三篇文章)

六(续)、从 KMP 算法一步一步谈到 BM 算法

六(三续)、KMP 算法之总结篇 (必懂 KMP)

七、遗传算法 透析 GA 本质

八、再谈启发式搜索算法

九、图像特征提取与匹配之 SIFT 算法 (SIFT 算法系列五篇文章)

九(续)、sift 算法的编译与实现

九（再续）、教你一步一步用 c 语言实现 sift 算法、上
九（再续）、教你一步一步用 c 语言实现 sift 算法、下
九（三续）：SIFT 算法的应用--目标识别之 Bag-of-words 模型
十、从头到尾彻底理解傅里叶变换算法、上
十、从头到尾彻底理解傅里叶变换算法、下
十一、从头到尾彻底解析 Hash 表算法
十一（续）、倒排索引关键词 Hash 不重复编码实践
十二、快速排序算法 （快速排序算法 3 篇文章）
十二（续）、快速排序算法的深入分析
十二（再续）：快速排序算法之所有版本的 c/c++ 实现
十三、通过浙大上机复试试题学 SPFA 算法
十四、快速选择 SELECT 算法的深入分析与实现
十五、多项式乘法与快速傅里叶变换

注解

自从本人写这个算法系列以来，总有不少的朋友问我如何学算法，问我怎么会有那么多的时间来学算法，在此，我愿回复各位俩句话：1、兴趣。2、没有兴趣的东西一般不会占用我的时间。

非常感谢，各位对我的支持与关注，谢谢大家。完。

版权所有，侵权必究。严禁用于任何商业用途，违者定究法律责任。

一、A*搜索算法

作者：July、二零一一年一月

博主说明：

- 1、本经典算法研究系列，此系列文章写的不够好之处，还望见谅。
 - 2、本经典算法研究系列，系我参考资料，一篇一篇原创所作，转载必须注明作者本人 July 及出处。
 - 3、本经典算法研究系列，精益求精，不断优化，永久更新，永久勘误。
- 欢迎，各位，与我一同学习探讨，交流研究。
有误之处，不吝指正。
-
-

引言

1968 年，的一篇论文，“P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths in graphs. IEEE Trans. Syst. Sci. and Cybernetics, SSC-4(2):100-107, 1968”。从此，一种精巧、高效的算法-----A*算法横空出世了，并在相关领域得到了广泛的应用。

启发式搜索算法

要理解 A*搜寻算法，还得从启发式搜索算法开始谈起。

所谓启发式搜索，就在于当前搜索结点往下选择下一步结点时，可以通过一个启发函数来进行选择，选择代价最少的结点作为下一步搜索结点而跳转其上（遇到有一个以上代价最少的结点，不妨选距离当前搜索点最近一次展开的搜索点进行下一步搜索）。

DFS 和 BFS 在展开子结点时均属于盲目型搜索，也就是说，它不会选择哪个结点在下一次搜索中更优而去跳转到该结点进行下一步的搜索。在运气不好的情形中，均需要试探完整个解集空间，显然，只能适用于问题规模不大的搜索问题中。

而与 DFS,BFS 不同的是，一个经过仔细设计的启发函数，往往在很快的时间内就可得到一个搜索问题的最优解，对于 NP 问题，亦可在多项式时间内得到一个较优解。是的，关键就是如何设计这个启发函数。

A*搜寻算法

A*搜寻算法，俗称 A 星算法，作为启发式搜索算法中的一种，这是一种在图形平面上，有多个节点的路径，求出最低通过成本的算法。常用于游戏中的 NPC 的移动计算，或线上游戏的 BOT 的移动计算上。该算法像 Dijkstra 算法一样，可以找到一条最短路径；也像 BFS 一样，进行启发式的搜索。

A*算法最为核心的部分，就在于它的一个估值函数的设计上：

$$f(n)=g(n)+h(n)$$

其中 $f(n)$ 是每个可能试探点的估值，它有两部分组成：

一部分，为 $g(n)$ ，它表示从起始搜索点到当前点的代价（通常用某结点在搜索树中的深度来表示）。

另一部分，即 $h(n)$ ，它表示启发式搜索中最为重要的一部分，即当前结点到目标结点的估值， $h(n)$ 设计的好坏，直接影响着具有此种启发式函数的启发式算法的是否能称为 A*算法。

一种具有 $f(n)=g(n)+h(n)$ 策略的启发式算法能成为 A*算法的充分条件是：

- 1、搜索树上存在着从起始点到终了点的最优路径。
- 2、问题域是有限的。
- 3、所有结点的子结点的搜索代价值 > 0。
- 4、 $h(n) \leq h^*(n)$ ($h^*(n)$ 为实际问题的代价值)。

当此四个条件都满足时，一个具有 $f(n)=g(n)+h(n)$ 策略的启发式算法能成为 A*算法，并一定能找到最优解。

对于一个搜索问题，显然，条件 1,2,3 都是很容易满足的，而条件 4： $h(n) \leq h^*(n)$ 是需要精心设计的，由于 $h^*(n)$ 显然是无法知道的，所以，一个满足条件 4 的启发策略 $h(n)$ 就来的难能可贵了。

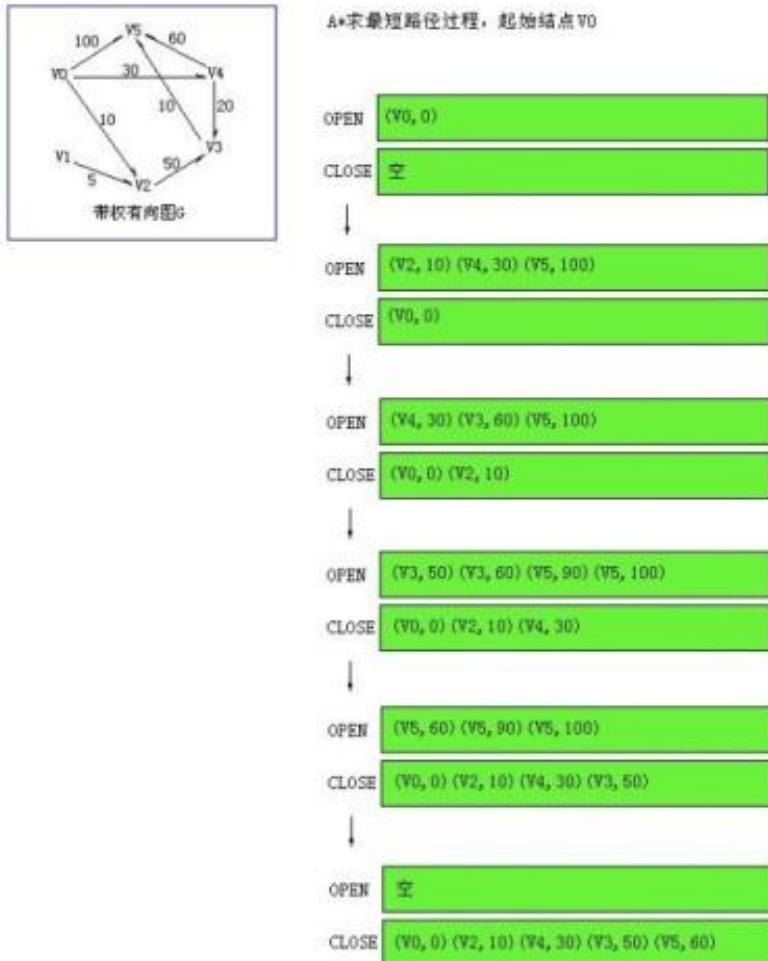
不过，对于图的最优路径搜索和八数码问题，有些相关策略 $h(n)$ 不仅很好理解，而且已经在理论上证明是满足条件 4 的，从而为这个算法的推广起到了决定性的作用。

且 $h(n)$ 距离 $h^*(n)$ 的量度不能过大，否则 $h(n)$ 就没有过强的区分能力，算法效率并不会

很高。对一个好的 $h(n)$ 的评价是： $h(n)$ 在 $h^*(n)$ 的下界之下，并且尽量接近 $h^*(n)$ 。

A*搜寻算法的实现

先举一个小小例子：即求 $V_0 \rightarrow V_5$ 的路径， $V_0 \rightarrow V_5$ 的过程中，可以经由 V_1, V_2, V_3, V_4 各点达到目的点 V_5 。下面的问题，即是求此起始顶点 $V_0 \rightarrow$ 途径任意顶点 $V_1, V_2, V_3, V_4 \rightarrow$ 目标顶点 V_5 的最短路径。



rickone 2006/09/11

//是的，图片是引用 rickone 的。

通过上图，我们可以看出：A*算法最核心的过程，就在每次选择下一个当前搜索点时，是从所有已探知的但未搜索过点中(可能是不同层，亦可不在同一条支路上)，选取 f 值最小的结点进行展开。

而所有“已探知的但未搜索过点”可以通过一个按 f 值升序的队列(即优先队列)进行排列。

这样，在整体的搜索过程中，只要按照类似广度优先的算法框架，从优先队列中弹出队首元素(f 值)，对其可能子结点计算 g 、 h 和 f 值，直到优先队列为空(无解)或找到终止点为止。

A*算法与广度、深度优先和 Dijkstra 算法的联系就在于：当 $g(n)=0$ 时，该算法类似于 DFS，当 $h(n)=0$ 时，该算法类似于 BFS。且同时，如果 $h(n)$ 为 0，只需求出 $g(n)$ ，即求出起点到任意顶点 n 的最短路径，则转化为单源最短路径问题，即 Dijkstra 算法。这一点，可以

通过上面的 A*搜索树的具体过程中将 $h(n)$ 设为 0 或将 $g(n)$ 设为 0 而得到。

A*算法流程：

首先将起始结点 S 放入 OPEN 表，CLOSE 表置空，算法开始时：

- 1、如果 OPEN 表不为空，从表头取一个结点 n，如果为空算法失败。
- 2、n 是目标解吗？是，找到一个解（继续寻找，或终止算法）。
- 3、将 n 的所有后继结点展开，就是从 n 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表，并把 S 放入 CLOSE 表，同时计算每一个后继结点的估价值 $f(n)$ ，将 OPEN 表按 $f(x)$ 排序，最小的放在表头，重复算法，回到 1。

```
//OPEN-->CLOSE, 起点-->任意顶点 g(n)-->目标顶点 h(n)
closedset := the empty set                                //已经被估算的节点集合
openset := set containing the initial node                //将要被估算的节点集合
g_score[start] := 0                                         //g(n)
h_score[start] := heuristic_estimate_of_distance(start, goal) //h(n)
f_score[start] := h_score[start]

while openset is not empty      //若 OPEN 表不为空
    x := the node in openset having the lowest f_score[] value //x 为 OPEN 表中最小的
    if x = goal                                              //如果 x 是一个解
        return reconstruct_path(came_from,goal)                  //
    remove x from openset
    add x to closedset                                         //x 放入
```

CLSOE 表

```
for each y in neighbor_nodes(x)
    if y in closedset
        continue
    tentative_g_score := g_score[x] + dist_between(x,y)

    if y not in openset
        add y to openset
        tentative_is_better := true
    else if tentative_g_score < g_score[y]
        tentative_is_better := true
    else
        tentative_is_better := false
    if tentative_is_better = true
        came_from[y] := x
        g_score[y] := tentative_g_score
        h_score[y] := heuristic_estimate_of_distance(y, goal) //x-->y-->goal
        f_score[y] := g_score[y] + h_score[y]

return failure

function reconstruct_path(came_from,current_node)
```

```
if came_from[current_node] is set
    p = reconstruct_path(came_from,came_from[current_node])
    return (p + current_node)
else
    return the empty path
```

与结点写在一起的数值表示那个结点的价值 $f(n)$, 当 OPEN 表为空时 CLOSE 表中将求得从 V_0 到其它所有结点的最短路径。

考虑到算法性能, 外循环中每次从 OPEN 表取一个元素, 共取了 n 次 (共 n 个结点), 每次展开一个结点的后续结点时, 需 $O(n)$ 次, 同时再对 OPEN 表做一次排序, OPEN 表大小是 $O(n)$ 量级的, 若用快排就是 $O(n \log n)$, 乘以外循环总的复杂度是 $O(n^2 * \log n)$,

如果每次不是对 OPEN 表进行排序, 因为总是不断地有新的结点添加进来, 所以不用进行排序, 而是每次从 OPEN 表中求一个最小的, 那只需要 $O(n)$ 的复杂度, 所以总的复杂度为 $O(n^2)$, 这相当于 Dijkstra 算法。

本文完。

July、二零一一年二月十日更新。

后续: July、二零一一年三月一日更新。

简述 A*最短路径算法的方法:

目标: 从当前位置 A 到目标位置 B 找到一条最短的行走路径。

方法:

从 A 点开始, 遍历所有的可走路径, 记录到一个结构中, 记录内容为 (位置点, 最小步数)

当任何第二次走到一个点的时候, 判断最小步数是否小于记录的内容, 如果是, 则更新掉原最小步数, 一直到所有的路径点都不能继续都了为止, 最终那个点被标注的最小步数既是最短路径,

而反向找跟它相连的步数相继少一个值的点连起来就形成了最短路径, 当多个点相同, 则任意取一条即可。

总结:

A*算法实际是个穷举算法, 也与课本上教的最短路径算法类似。课本上教的是两头往中间走, 也是所有路径都走一次, 每一个点标注最短值。(i am sorry。文章, 写的差, 恳请各位读者参考此 A*搜寻算法的后续一篇文章: 一 (续)、A*, Dijkstra, BFS 算法性能比较及 A*算法的应用。谢谢大家。) 本文完。

一 (续) 、A*, Dijkstra, BFS 算法性能比较及 A*算法的应用

-----作者:July 二零一一年三月十日。

出处: http://blog.csdn.net/v_JULY_v

引言：

最短路径的各路算法 A*算法、Dijkstra 算法、BFS 算法，都已在本 BLOG 内有所阐述了。其中，Dijkstra 算法，后又写了一篇文章继续阐述：二（续）、理解 Dijkstra 算法。但，想必，还是有部分读者对此类最短路径算法，并非已了然于胸，或者，无一个总体大概的印象。

本文，即以演示图的形式，比较它们各自的寻路过程，让各位对它们有一个清晰而直观的印象。

我们比较，以下五种算法：

1. A* (使用曼哈顿距离)
2. A* (采用欧氏距离)
3. A* (利用切比雪夫距离)
4. Dijkstra
5. Bi-Directional Breadth-First-Search(双向广度优先搜索)

咱们以下图为例，图上绿色方块代表起始点，红色方块代表目标点，紫色的方块代表障碍物，白色的方块代表可以通行的路径。

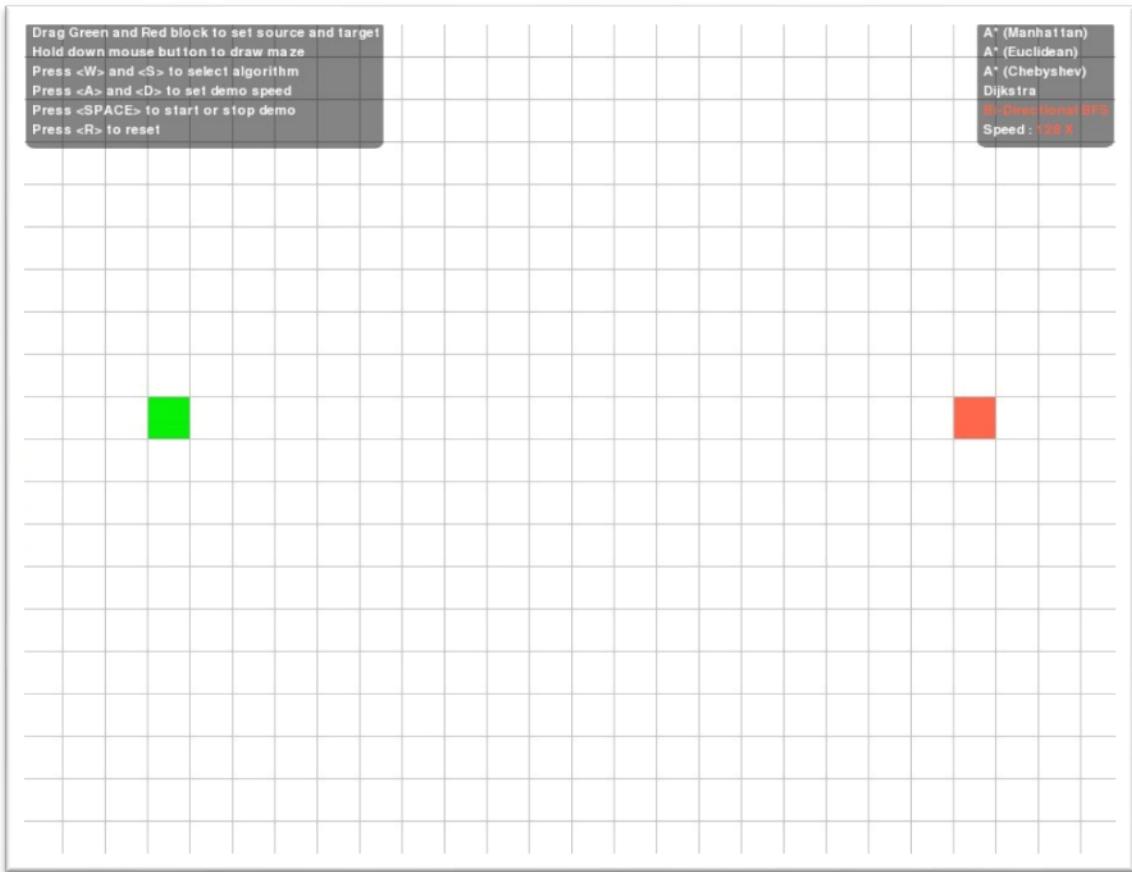
下面，咱们随意摆放起始点绿块，目标点红块的位置，然后，在它们中间随便画一些障碍物，

最后，运行程序，比较使用上述五种算法，得到各自不同的路径，各自找寻过程中所覆盖的范围，各自的工作流程，并从中可以窥见它们的效率高低。

A*、Dijkstra、BFS 算法性能比较演示：

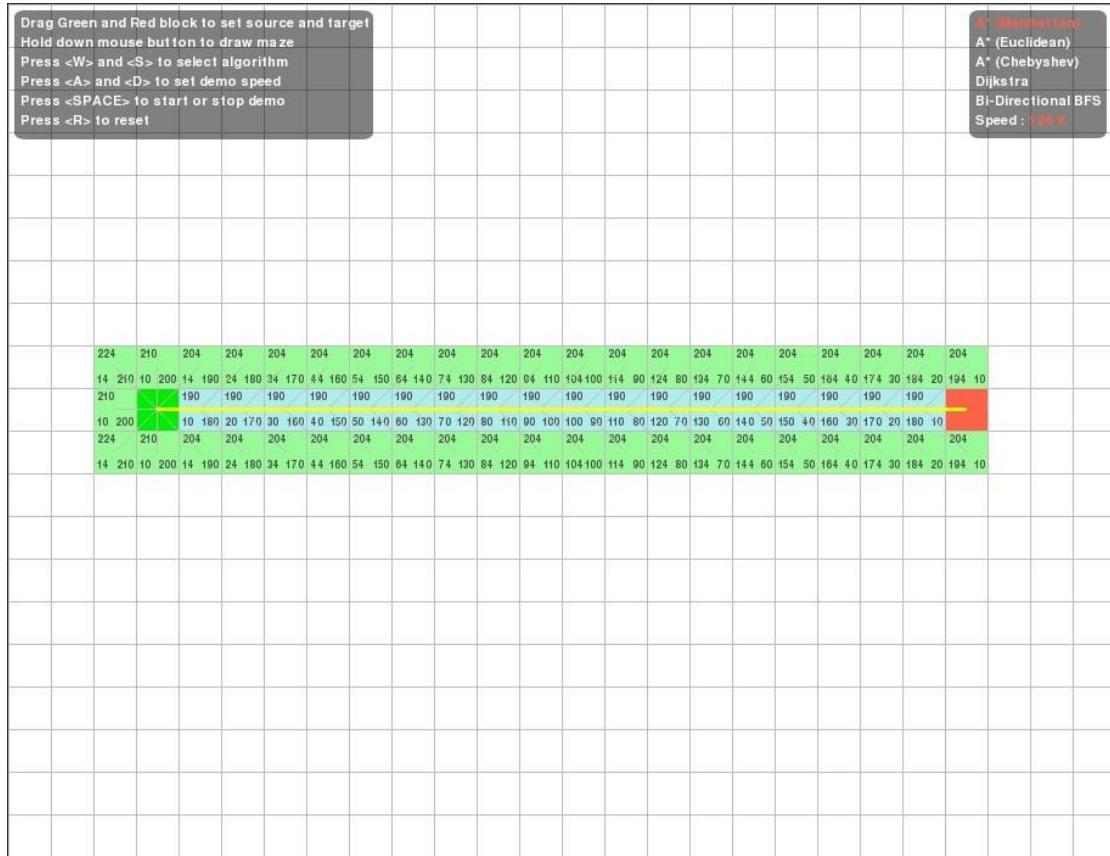
ok，任意摆放绿块与红块的三种状态：

一、起始点绿块，与目标点红块在同一条水平线上：

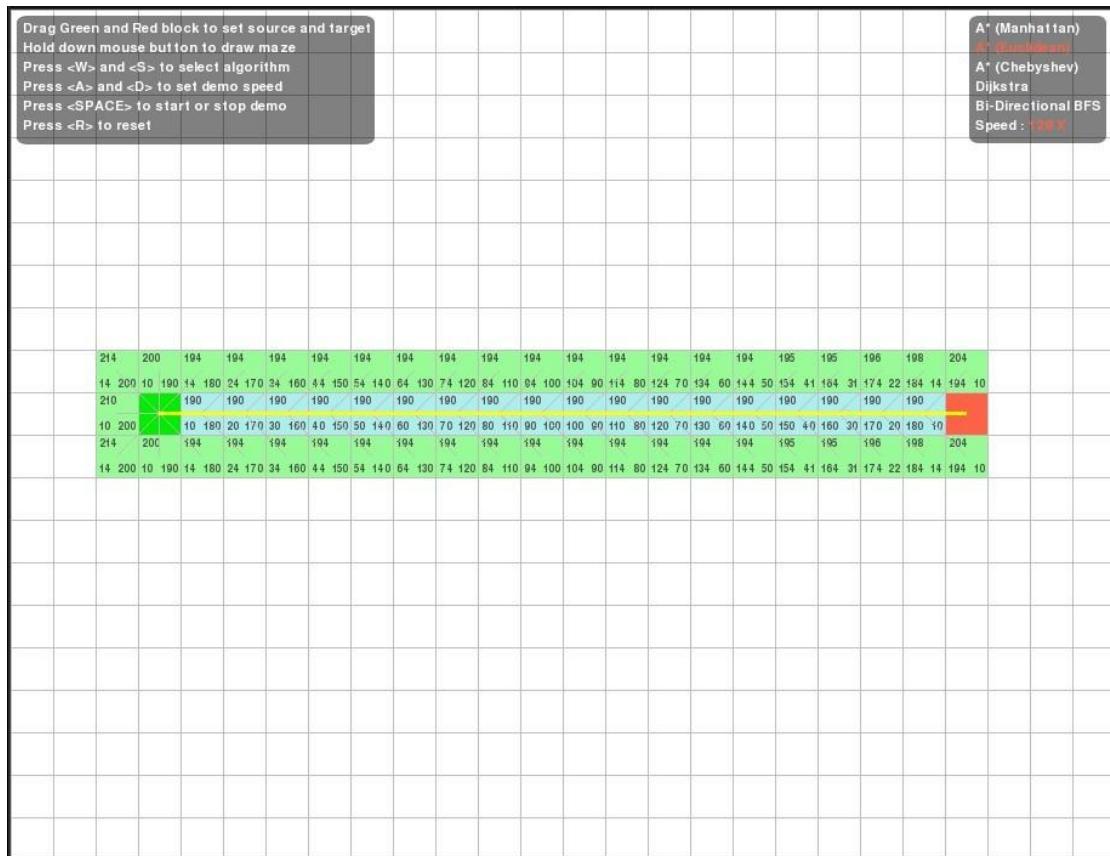


各自的搜寻路径为：

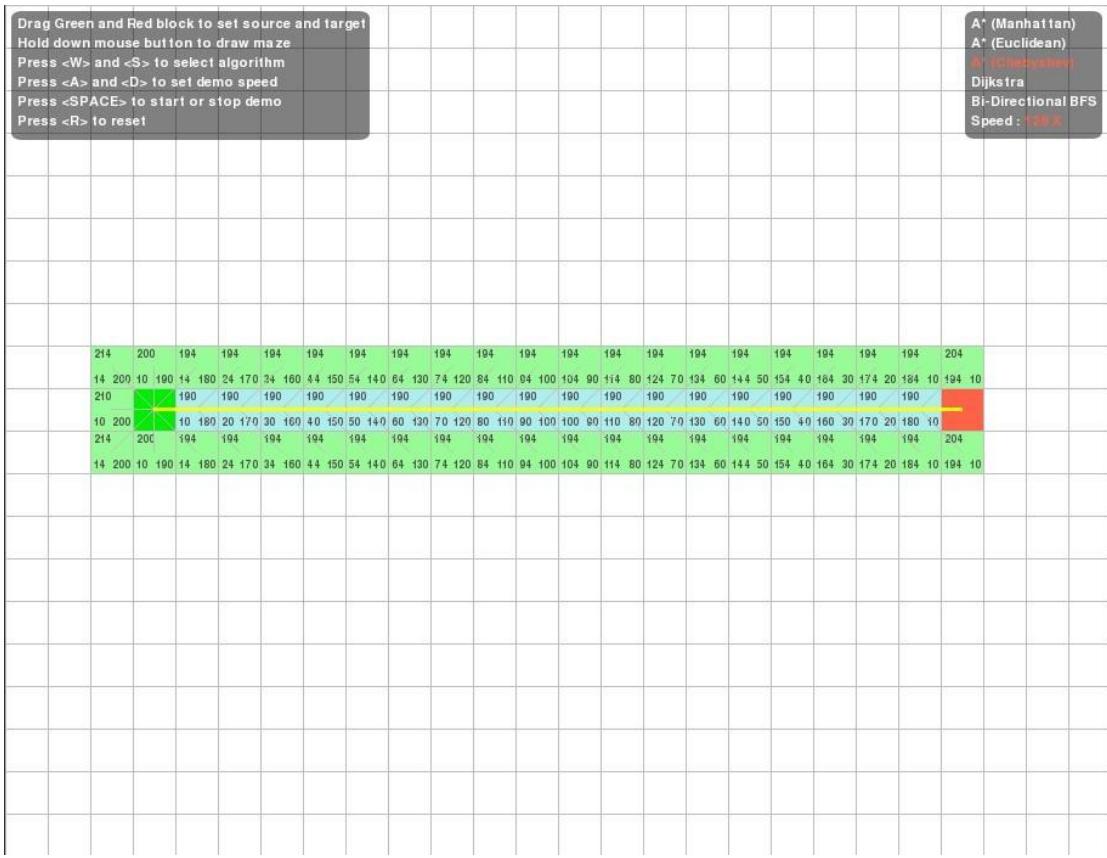
1. A*(使用曼哈顿距离)



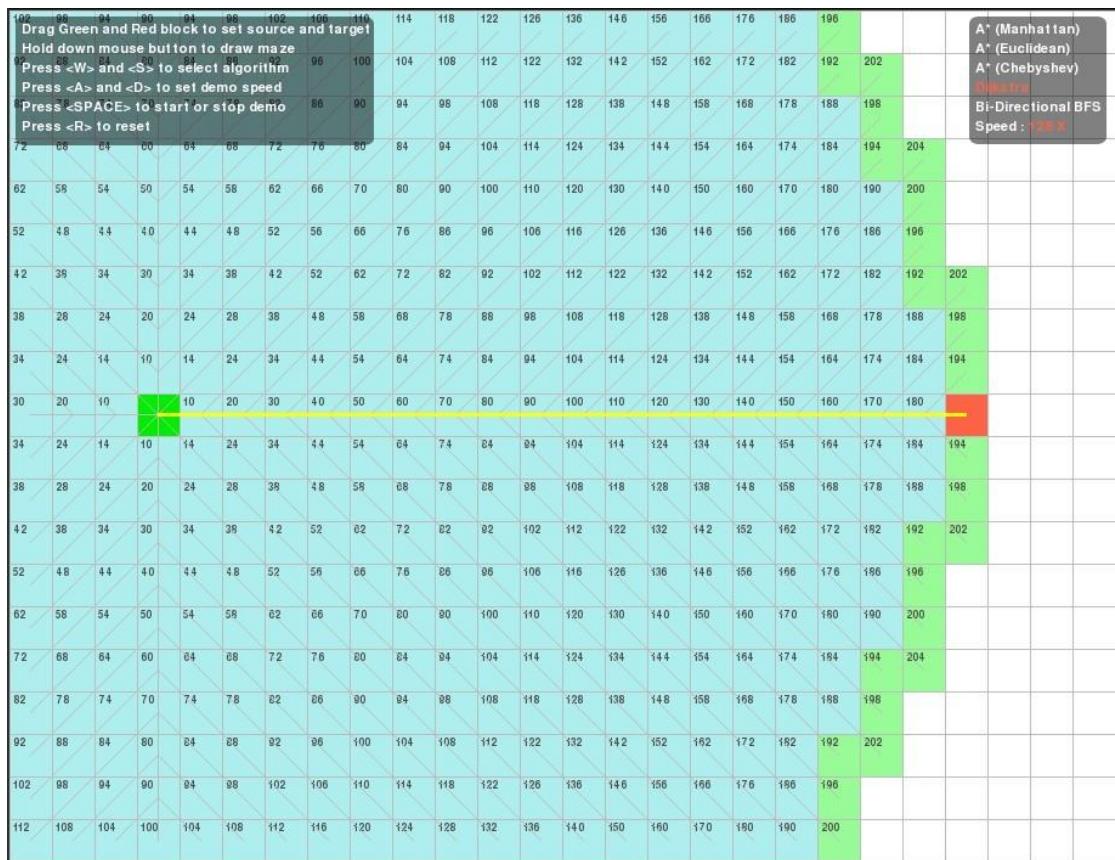
2. A*(采用欧氏距离)



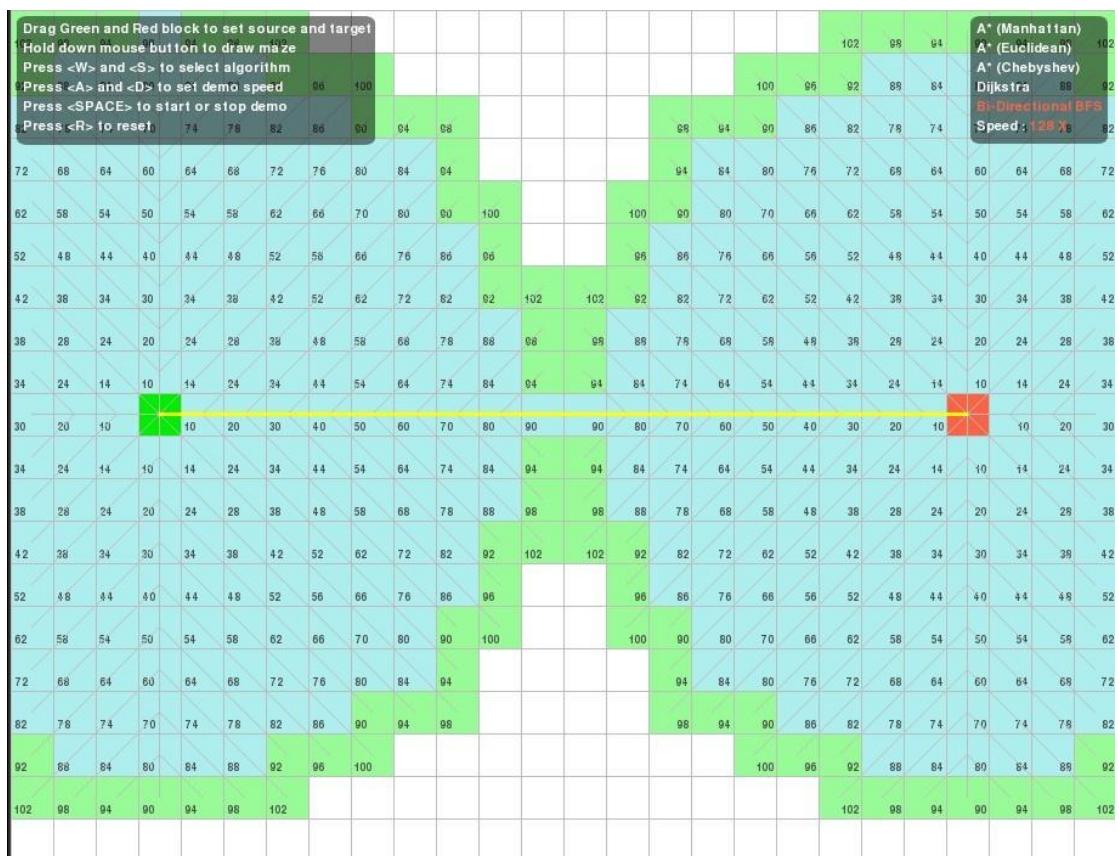
3. A*(利用切比雪夫距离)



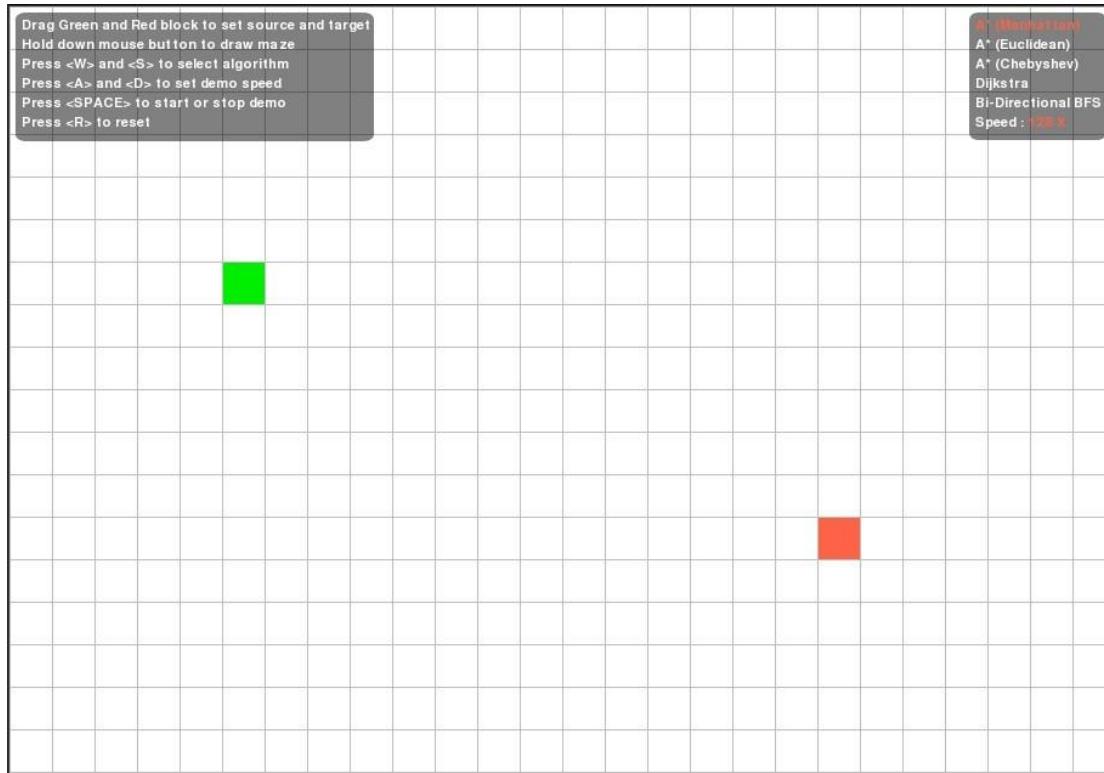
4. **Dijkstra 算法.** //很明显，Dijkstra 搜索效率明显差于上述 A* 算法。(看它最后找到目标点红块所走过的路径，和覆盖的范围，即能轻易看出来，下面的比较，也是基于同一个道理。看路径，看覆盖的范围，评价一个算法的效率)。



5. Bi-Directional Breadth-First-Search(双向广度优先搜索)

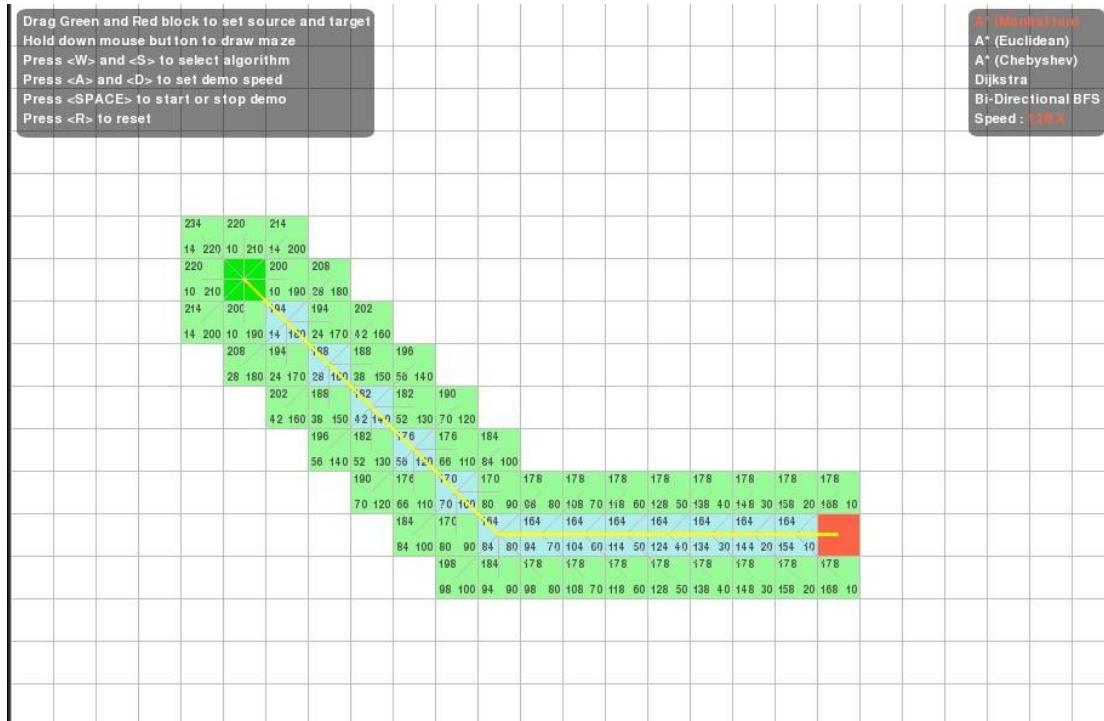


二、起始点绿块，目标点红块在一斜线上：



各自的搜寻路径为：

1. A*(使用曼哈顿距离)



2. A*(采用欧氏距离)



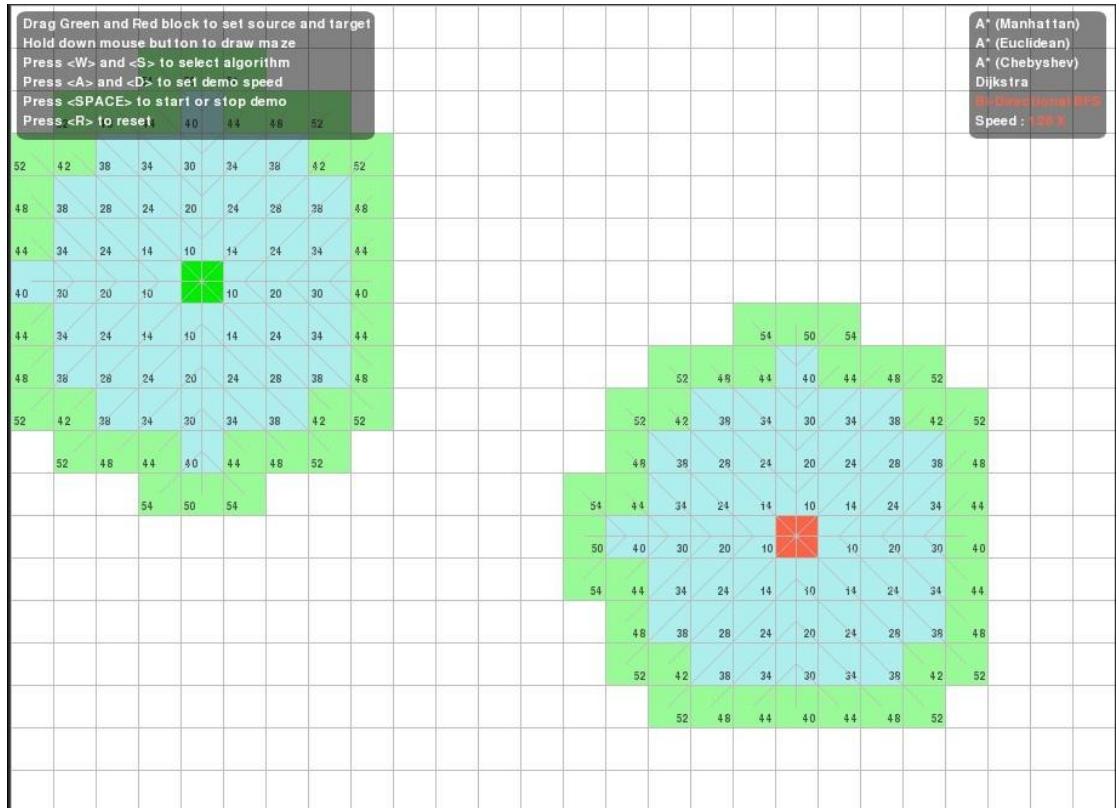
3. A*(利用切比雪夫距离)

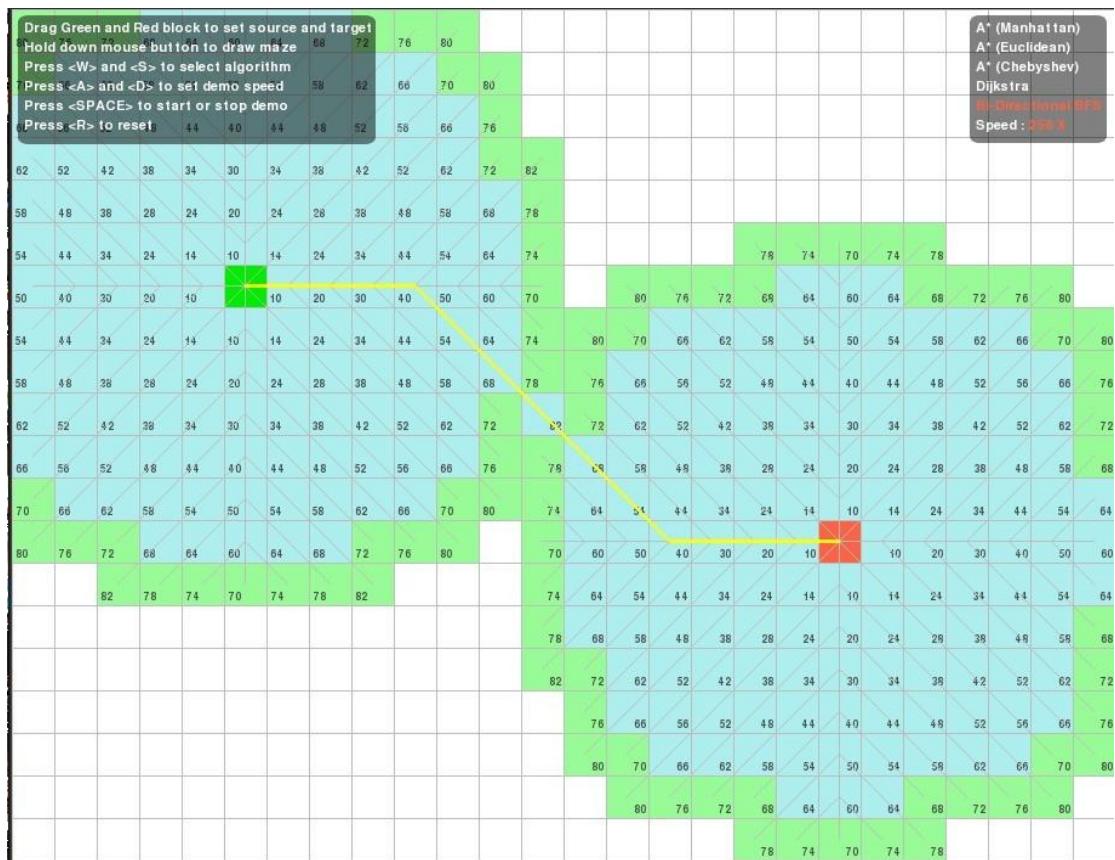


4. Dijkstra 算法。 //与上述 A* 算法比较，覆盖范围大，搜寻效率较低



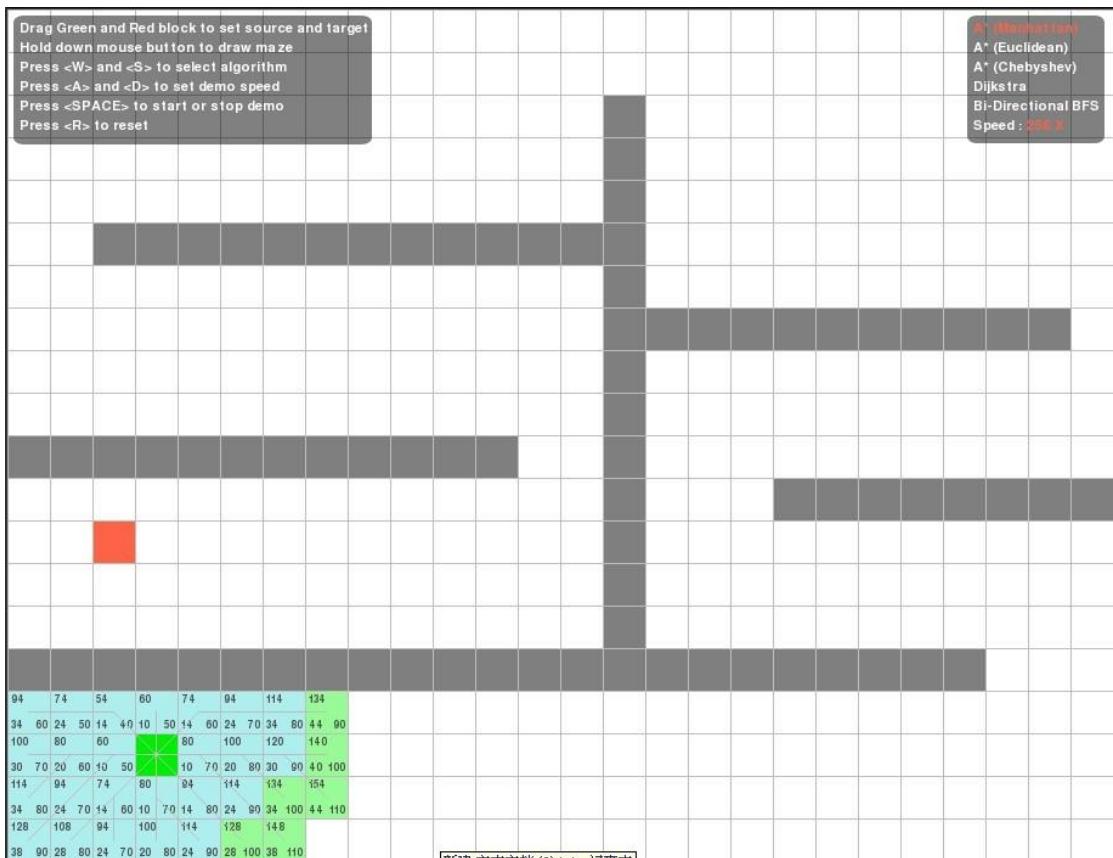
5. Bi-Directional Breadth-First-Search(双向广度优先搜索)



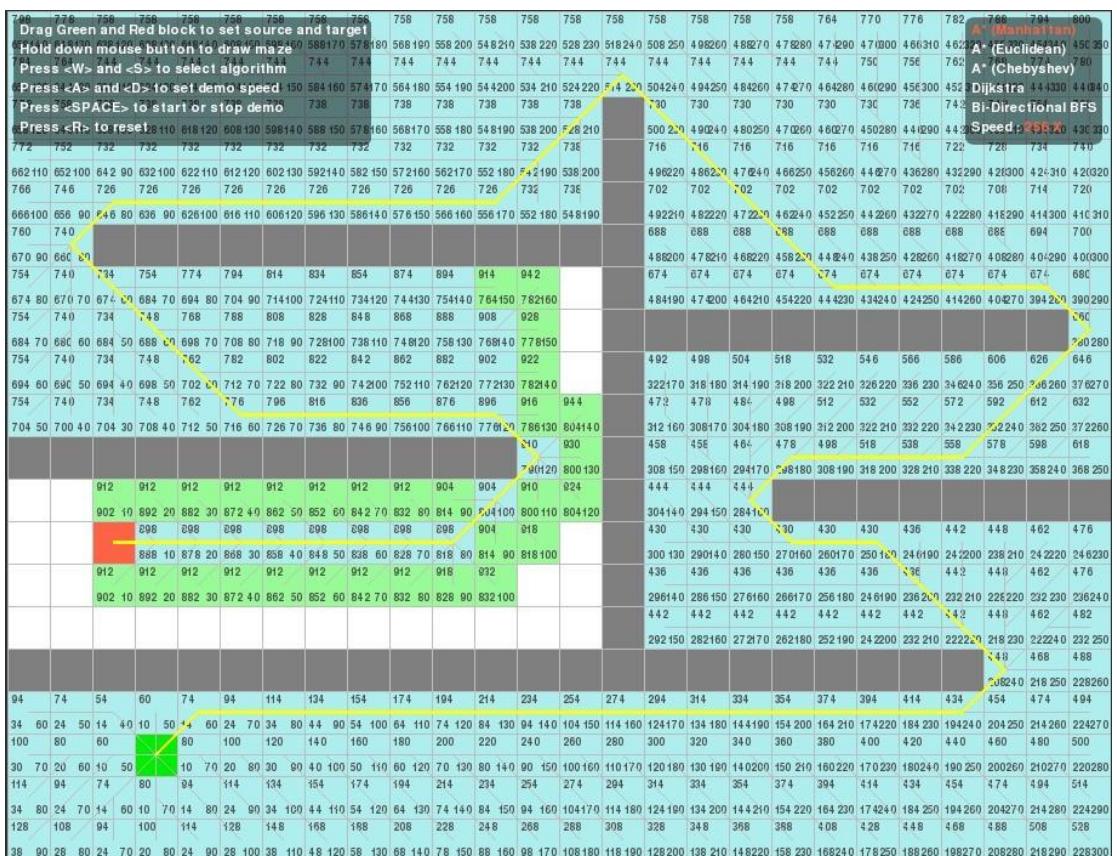


三、起始点绿块，目标点红块被多重障碍物阻挡：

各自的搜寻路径为（同样，还是从绿块到红块）：



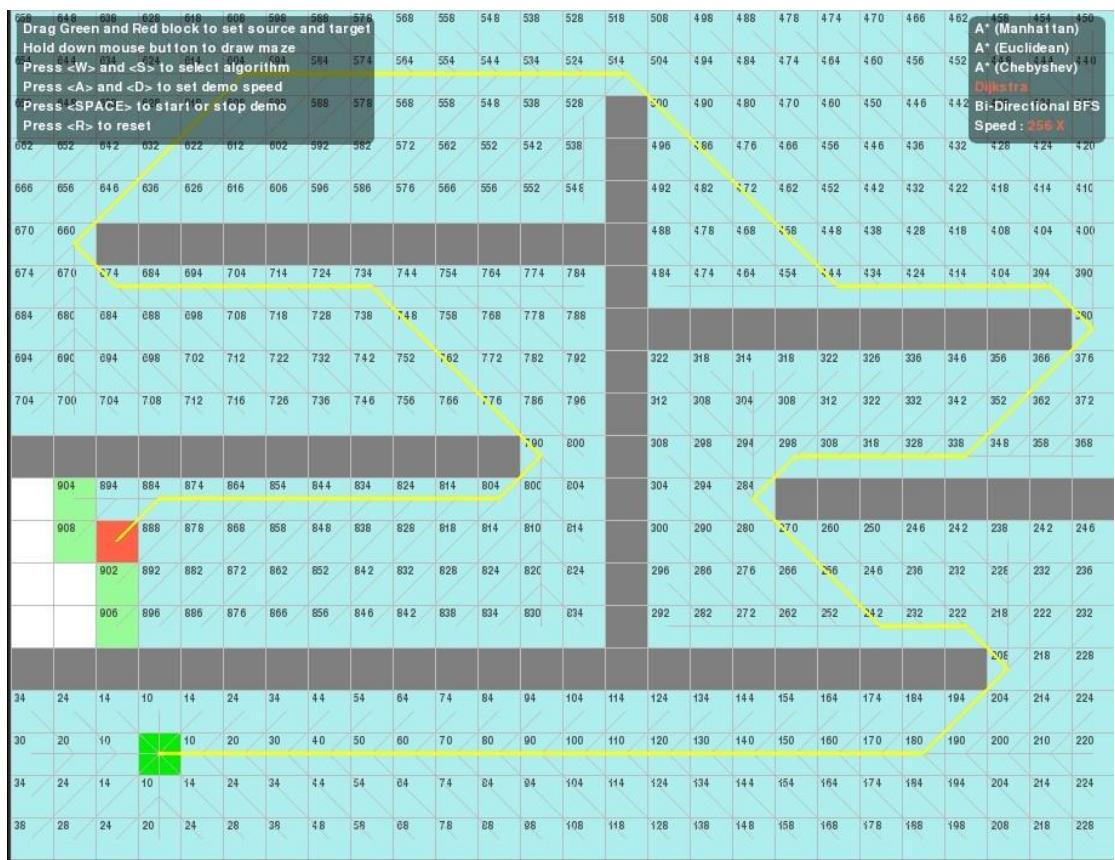
1. A* (使用曼哈顿距离)



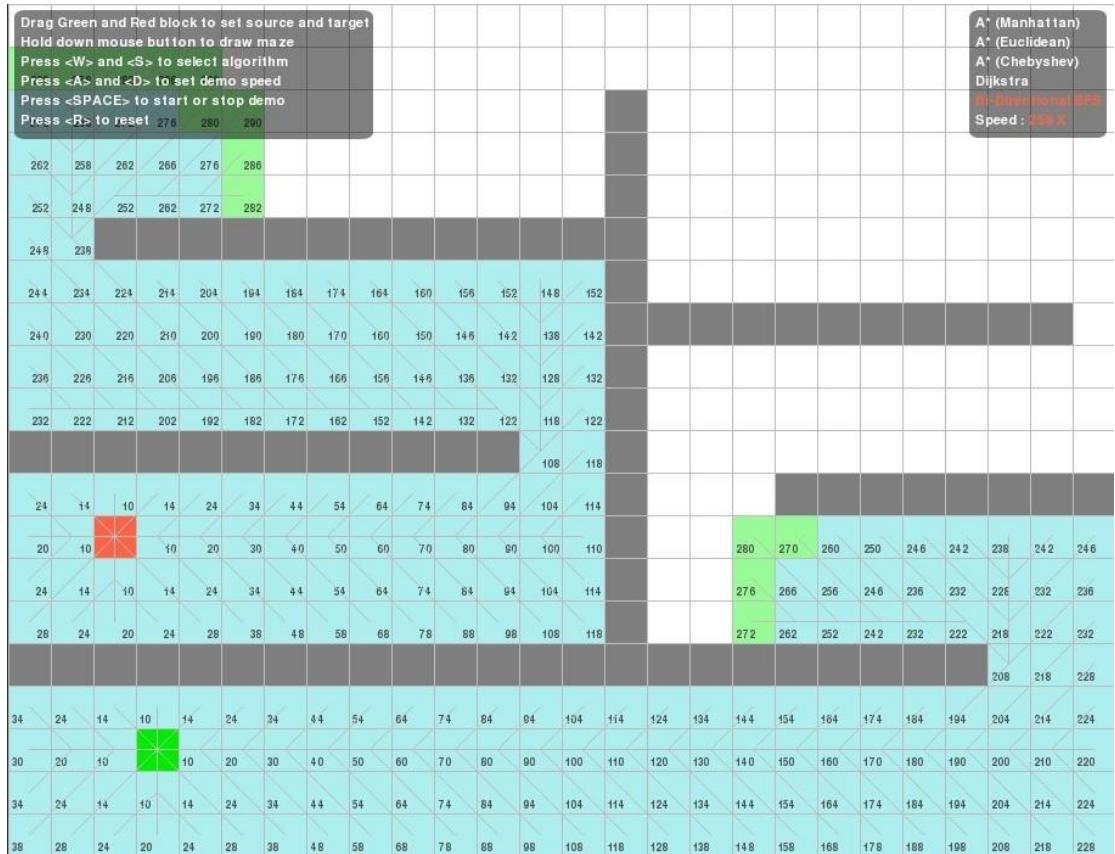
2. A*(采用欧氏距离)

3. A*(利用切比雪夫距离)

4. Dijkstra



5. Bi-Directional Breadth-First-Search(双向广度优先搜索) //覆盖范围同上述
Dijkstra 算法一样很大，效率低下。



A*搜索算法的高效之处

如上，是不是对 A*、Dijkstra、双向 BFS 算法各自的性能有了个总体大概的印象了？由上述演示，我们可以看出，在最短路径搜寻效率上，一般有 $A^*>Dijkstra$ 、双向 BFS，其中 Dijkstra、双向 BFS 到底哪个算法更优，还得看具体情况。

由上，我们也可以看出，A*搜索算法的确是一种比较高效的寻路算法。

A*算法最为核心的过程，就在每次选择下一个当前搜索点时，是从所有已探知的但未搜索过点中（可能是不同层，亦可不在同一条支路上），选取 f 值最小的结点进行展开。

而所有“已探知的但未搜索过点”可以通过一个按 f 值升序的队列（即优先队列）进行排列。

这样，在整体的搜索过程中，只要按照类似广度优先的算法框架，从优先队列中弹出队首元素（f 值），对其可能子结点计算 g、h 和 f 值，直到优先队列为空（无解）或找到终止点为止。

A*算法与广度、深度优先和 Dijkstra 算法的联系就在于：当 $g(n)=0$ 时，该算法类似于 DFS，当 $h(n)=0$ 时，该算法类似于 BFS。且同时，如果 $h(n)$ 为 0，只需求出 $g(n)$ ，即求出起点到任意顶点 n 的最短路径，则转化为单源最短路径问题，即 Dijkstra 算法。这一点，可以通过上面的 A*搜索树的具体过程中将 $h(n)$ 设为 0 或将 $g(n)$ 设为 0 而得到。

BFS、DFS 与 A*搜索算法的比较

参考了算法驿站上的部分内容：

不管以下论述哪一种搜索，都统一用这样的形式表示：搜索的对象是一个图，它面向一个问题，不一定有明确的存储形式，但它里面的一个结点都有可能是一个解（可行解），搜索的目的有两个方面，或者求可行解，或者从可行解集中求最优解。

我们用两张表来进行搜索，一个叫 OPEN 表，表示那些已经展开但还没有访问的结点集，另一个叫 CLOSE 表，表示那些已经访问的结点集。

蛮力搜索（BFS，DFS）

BFS（Breadth-First-Search 宽度优先搜索）

首先将起始结点放入 OPEN 表，CLOSE 表置空，算法开始时：

- 1、如果 OPEN 表不为空，从表中开始取一个结点 S，如果为空算法失败
- 2、S 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3

3、将 S 的所有后继结点展开，就是从 S 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表末尾，而把 S 放入 CLOSE 表，重复算法到 1。

DFS（Depth-First-Search 深度优先搜索）

首先将起始结点放入 OPEN 表，CLOSE 表置空，算法开始时：

- 1、如果 OPEN 表不为空，从表中开始取一个结点 S，如果为空算法失败
- 2、S 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3

3、将 S 的所有后继结点展开，就是从 S 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表开始，而把 S 放入 CLOSE 表，重复算法到 1。

是否有看出：上述的 BFS 和 DFS 有什么不同？

仔细观察 OPEN 表中待访问的结点的组织形式，BFS 是从表头取结点，从表尾添加结点，也就是说 OPEN 表是一个队列，是的，BFS 首先让你想到‘队列’；而 DFS，它是从 OPEN 表头取结点，也从表头添加结点，也就是说 OPEN 表是一个栈！

DFS 用到了栈，所以有一个很好的实现方法，那就是递归，系统栈是计算机程序中极重要的部分之一。用递归也有个好处就是，在系统栈中只需要存结点最大深度那么大的空间，

也就是在展开一个结点的后续结点时可以不用一次全部展开，用一些环境变量记录当前的状态，在递归调用结束后继续展开。

利用系统栈实现的 DFS

```
函数 dfs(结点 s)
{
    s 超过最大深度了吗？是：相应处理，返回；
    s 是目标结点吗？是：相应处理；否则：
    {
        s 放入 CLOSE 表；
        for(c=s.第一个子结点；c 不为空；c=c.下一个子结点())
            if(c 不在 CLOSE 表中)
                dfs(c); 递归
    }
}
```

如果指定最大搜索深度为 n ，那系统栈最多使用 n 个单位，它相当于有状态指示的 OPEN 表，状态就是 c ，在栈里存了前面搜索时的中间变量 c ，在后面的递归结束后， c 继续后移。在象棋等棋类程序中，就是用这样的 DFS 的基本模式搜索棋局局面树的，因为如果用 OPEN 表，有可能还没完成搜索 OPEN 表就暴满了，这是难于控制的情况。

我们说 DFS 和 BFS 都是蛮力搜索，因为它们在搜索到一个结点时，在展开它的后续结点时，是对它们没有任何‘认识’的，它认为它的孩子们都是一样的‘优秀’，但事实并非如此，后续结点是有好有坏的。好，就是说它离目标结点‘近’，如果优先处理它，就会更快的找到目标结点，从而整体上提高搜索性能。

启发式搜索

为了改善上面的算法，我们需要对展开后续结点时对子结点有所了解，这里需要一个估值函数，估值函数就是评价函数，它用来评价子结点的好坏，因为准确评价是不可能的，所以称为估值。打个比方，估值函数就像一台显微镜，一双‘慧眼’，它能分辨出看上去一样的孩子们的手，哪个很脏，有细菌，哪个没有，很干净，然后对那些干净的孩子进行奖励。这里相当于是需要‘排序’，排序是要有代价的，而花时间做这样的工作会不会对整体搜索效率有所帮助呢，这完全取决于估值函数。

排序，怎么排？用哪一个？快排吧，`qsort!` 不一定，要看要排多少结点，如果很少，简单排序法就很 OK 了。看具体情况了。

排序可能是对 OPEN 表整体进行排序，也可以是对后续展开的子结点排序，排序的目的就是要使程序有启发性，更快的搜出目标解。

如果估值函数只考虑结点的某种性能上的价值，而不考虑深度，比较有名的就是有序搜索（Ordered-Search），它着重看好能否找出解，而不看解离起始结点的距离（深度）。

如果估值函数考虑了深度，或者是带权距离（从起始结点到目标结点的距离加权和），那就是 A*，举个问题例子，八数码问题，如果不考虑深度，就是说不要求最少步数，移动一步就相当于向后多展开一层结点，深度多算一层，如果要求最少步数，那就需要用 A*。

简单的来说 A* 就是将估值函数分成两个部分，一个部分是路径价值，另一个部分是一般性启发价值，合在一起算整个结点的价值。

从 A* 的角度看前面的搜索方法，如果路径价值为 0 就是有序搜索，如果路径价值就用所在结点到起始结点的距离（深度）表示，而启发值为 0，那就是 BFS 或者 DFS，它们两

刚好是个反的，BFS 是从 OPEN 表中选一个深度最小的进行展开，

而 DFS 是从 OPEN 表中选一个深度最大的进行展开。当然只有 BFS 才算是特殊的 A*，所以 BFS 可以求要求路径最短的问题，只是没有任何启发性。下文稍后，会具体谈 A* 搜索算法思想。

BFS、DFS、Kruskal、Prim、Dijkstra 算法时间复杂度

上面，既然提到了 A* 算法与广度、深度优先搜索算法的联系，那么，下面，也顺便再比较下 BFS、DFS、Kruskal、Prim、Dijkstra 算法时间复杂度吧：

一般说来，我们知道，BFS，DFS 算法的时间复杂度为 $O(V+E)$ ，

最小生成树算法 Kruskal、Prim 算法的时间复杂度为 $O(E \cdot \lg V)$ 。

而 Prim 算法若采用斐波那契堆实现的话，算法时间复杂度为 $O(E + V \cdot \lg V)$ ，当 $|V| \ll |E|$ 时， $E + V \cdot \lg V$ 是一个较大的改进。

// $|V| \ll |E|$, => $O(E + V \cdot \lg V) \ll O(E \cdot \lg V)$, 对吧。:D

Dijkstra 算法，斐波那契堆用作优先队列时，算法时间复杂度为 $O(V \cdot \lg V + E)$ 。

//看到了吧，与 Prim 算法采用斐波那契堆实现时，的算法时间复杂度是一样的。

所以我们，说，BFS、Prime、Dijkstra 算法是有相似之处的，单从各算法的时间复杂度比较看，就可窥之一二。

A* 搜索算法的思想

ok，既然，A* 搜索算法作为一种好的、高效的寻路算法，咱们就来想办法实现它吧。

实现一个算法，首先得明确它的算法思想，以及算法的步骤与流程，从我之前的一篇文章中，可以了解到：

A* 算法，作为启发式算法中很重要的一种，被广泛应用在最优路径求解和一些策略设计的问题中。

而 A* 算法最为核心的部分，就在于它的一个估值函数的设计上：

$$f(n) = g(n) + h(n)$$

其中 $f(n)$ 是每个可能试探点的估值，它有两部分组成：

一部分，为 $g(n)$ ，它表示从起始搜索点到当前点的代价（通常用某结点在搜索树中的深度来表示）。

另一部分，即 $h(n)$ ，它表示启发式搜索中最为重要的一部分，即当前结点到目标结点的估值，

$h(n)$ 设计的好坏，直接影响着具有此种启发式函数的启发式算法的是否能称为 A* 算法。

一种具有 $f(n) = g(n) + h(n)$ 策略的启发式算法能成为 A* 算法的充分条件是：

1、搜索树上存在着从起始点到终了点的最优路径。

2、问题域是有限的。

3、所有结点的子结点的搜索代价值 > 0。

4、 $h(n) \leq h^*(n)$ ($h^*(n)$ 为实际问题的代价值)。

当此四个条件都满足时，一个具有 $f(n) = g(n) + h(n)$ 策略的启发式算法能成为 A* 算法，并一定能找到最优解。

对于一个搜索问题，显然，条件 1,2,3 都是很容易满足的，而条件 4： $h(n) \leq h^*(n)$ 是需要精心设计的，由于 $h^*(n)$ 显然是无法知道的，所以，一个满足条件 4 的启发策略 $h(n)$ 就来的难能可贵了。

不过，对于图的最优路径搜索和八数码问题，有些相关策略 $h(n)$ 不仅很好理解，而且已经在理论上证明是满足条件 4 的，从而为这个算法的推广起到了决定性的作用。

A* 搜索算法的应用

ok，咱们就来应用 A*搜寻算法实现八数码问题，下面，就是其主体代码，由于给的注释很详尽，就不再啰嗦了，有任何问题，请不吝指正：

```
//节点结构体
typedef struct Node
{
    int data[9];
    double f,g;
    struct Node * parent;
}Node,*Lnode;

//OPEN CLOSED 表结构体
typedef struct Stack
{
    Node * npoint;
    struct Stack * next;
}Stack,* Lstack;

//选取 OPEN 表上 f 值最小的节点，返回该节点地址
Node * Minf(Lstack * Open)
{
    Lstack temp = (*Open)->next,min = (*Open)->next,minp = (*Open);
    Node * minx;
    while(temp->next != NULL)
    {
        if((temp->next ->npoint->f) < (min->npoint->f))
        {
            min = temp->next;
            minp = temp;
        }
        temp = temp->next;
    }
    minx = min->npoint;
    temp = minp->next;
    minp->next = minp->next->next;
    free(temp);
    return minx;
}

//判断是否可解
int Canslove(Node * suc, Node * goal)
{
    int a = 0,b = 0,i,j;
    for(i = 1; i< 9;i++)
        for(j = 0;j < i;j++)
```

```

{
    if((suc->data[i] > suc->data[j]) && suc->data[j] != 0)
        a++;
    if((goal->data[i] > goal->data[j]) && goal->data[j] != 0)
        b++;
}
if(a%2 == b%2)
    return 1;
else
    return 0;
}

//判断节点是否相等 , 1 相等, 0 不相等
int Equal(Node * suc,Node * goal)
{
for(int i = 0; i < 9; i ++ )
    if(suc->data[i] != goal->data[i])return 0;
    return 1;
}

//判断节点是否属于 OPEN 表 或 CLOSED 表, 是则返回节点地址, 否则返回空地址
Node * Belong(Node * suc,Lstack * list)
{
Lstack temp = (*list) -> next ;
if(temp == NULL) return NULL;
while(temp != NULL)
{
    if(Equal(suc,temp->npoint))return temp -> npoint;
    temp = temp->next;
}
return NULL;
}

//把节点放入 OPEN 或 CLOSED 表中
void Putinto(Node * suc,Lstack * list)
{
    Stack * temp;
    temp =(Stack *) malloc(sizeof(Stack));
    temp->npoint = suc;
    temp->next = (*list)->next;
    (*list)->next = temp;
}

```

///////////计算 f 值部分-开始///////////

```

double Fvalue(Node suc, Node goal, float speed)
{//计算 f 值
    double Distance(Node,Node,int);
    double h = 0;
    for(int i = 1; i <= 8; i++)
        h = h + Distance(suc, goal, i);
    return h*speed + suc.g; //f = h + g (speed 值增加时搜索过程以找到目标为优先因此可能
    不会返

```

```

回最优解)
}

double Distance(Node suc, Node goal, int i)
{//计算方格的错位距离
    int k,h1,h2;
    for(k = 0; k < 9; k++)
    {
        if(suc.data[k] == i)h1 = k;
        if(goal.data[k] == i)h2 = k;
    }
    return double(fabs(h1/3 - h2/3) + fabs(h1%3 - h2%3));
}
//////////////////计算 f 值部分-结束///////////////////

```

```

//////////////////扩展后继节点部分的函数-开始/////////////////
int BelongProgram(Lnode * suc ,Lstack * Open ,Lstack * Closed ,Node goal ,float speed)
{//判断子节点是否属于 OPEN 或 CLOSED 表 并作出相应的处理
    Node * temp = NULL;
    int flag = 0;
    if((Belong(*suc,Open) != NULL) || (Belong(*suc,Closed) != NULL))
    {
        if(Belong(*suc,Open) != NULL) temp = Belong(*suc,Open);
        else temp = Belong(*suc,Closed);
        if(((*suc)->g) < (temp->g))
        {
            temp->parent = (*suc)->parent;
            temp->g = (*suc)->g;
            temp->f = (*suc)->f;
            flag = 1;
        }
    }
    else
{

```

```

Putinto(* suc, Open);
(*suc)->f = Fvalue(**suc, goal, speed);
}
return flag;
}

void Spread(Lnode * suc, Lstack * Open, Lstack * Closed, Node goal, float speed)
{//扩展后继节点总函数
int i;
Node * child;
for(i = 0; i < 4; i++)
{
if(Canspread(**suc, i+1)) //判断某个方向上的子节点可否扩展
{
child = (Node *) malloc(sizeof(Node)); //扩展子节点
child->g = (*suc)->g + 1; //算子节点的 g 值
child->parent = (*suc); //子节点父指针指向父节点
Spreadchild(child, i); //向该方向移动空格生成子节点
if(BelongProgram(&child, Open, Closed, goal, speed)) // 判断子节点是否属

```

于 OPEN 或 CLOSED 表 并作出相应的处理

```

free(child);
}
}
}

//////////////////扩展后继节点部分的函数-结束/////////////////

```

```

Node * Process(Lnode * org, Lnode * goal, Lstack * Open, Lstack * Closed, float speed)
{//总执行函数
while(1)
{
if((*Open)->next == NULL) return NULL; //判断 OPEN 表是否为空, 为空则失败退出
Node * minf = Minf(Open); //从 OPEN 表中取出 f 值最小的节点
Putinto(minf, Closed); //将节点放入 CLOSED 表中
if(Equal(minf, *goal)) return minf; //如果当前节点是目标节点, 则成功退出
Spread(&minf, Open, Closed, **goal, speed); //当前节点不是目标节点时扩展
当前节点的后继

```

```

节点
}
}
}
```

```

int Shownum(Node * result)
{//递归显示从初始状态到达目标状态的移动方法

```

```
if(result == NULL) return 0;
else
{
    int n = Shownum(result->parent);
    for(int i = 0; i < 3; i++)
    {
        printf("\n");
        for(int j = 0; j < 3; j++)
        {
            if(result->data[i*3+j] != 0)
                printf(" %d ",result->data[i*3+j]);
            else printf("   ");
        }
        printf("\n");
    }
    return n+1;
}
```

后记：

日后，本 BLOG 将陆续实现所有经典的算法。是为记。完。

版权声明：

1、本人对本 BLOG 内所有任何文章和资料享有版权，转载，请注明作者本人，并以链接形式注明出处。

2、侵犯本人版权相关利益者，个人会在新浪微博、CSDN 迷你博客中永久追踪，给予谴责。

同时，保留追究法律责任的权利。向您的厚道致敬，谢谢。

July、二零一一年三月十日。

二、Dijkstra 算法初探

July 二零一一年一月

本文主要参考：算法导论 第二版、维基百科。

写的不好之处，还望见谅。

本经典算法研究系列文章，永久勘误，永久更新、永久维护。

July、二零一一年二月十日更新。

一、Dijkstra 算法的介绍

Dijkstra 算法，又叫迪科斯彻算法（Dijkstra），
算法解决的是有向图中单个源点到其他顶点的最短路径问题。

举例来说，
如果图中的顶点表示城市，而边上的权重表示著城市间开车行经的距离，
Dijkstra 算法可以用来找到两个城市之间的最短路径。

二、Dijkstra 的算法实现

Dijkstra 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，以 E 表示 G 中所有边的集合。

(u, v) 表示从顶点 u 到 v 有路径相连，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 的非负花费值（cost），边的花费可以想像成两个顶点之间的距离。

任两点间路径的花费值，就是该路径上所有边的花费值总和。

已知有 V 中有顶点 s 及 t ，Dijkstra 算法可以找到 s 到 t 的最低花费路径(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。

好，咱们来看下此算法的具体实现：

Dijkstra 算法的实现一（维基百科）：

$u := \text{Extract_Min}(Q)$ 在顶点集合 Q 中搜索有最小的 $d[u]$ 值的顶点 u 。这个顶点被从集合 Q 中删除并返回给用户。

```
1 function Dijkstra(G, w, s)
2     for each vertex v in V[G]                                // 初始化
3         d[v] := infinity
4         previous[v] := undefined
5     d[s] := 0
6     S := empty set
7     Q := set of all vertices
8     while Q is not an empty set                               // Dijkstra 演算法主體
9         u := Extract_Min(Q)
10        S := S union {u}
11        for each edge (u,v) outgoing from u
12            if d[v] > d[u] + w(u,v)                          // 拓展边 (u,v)
13                d[v] := d[u] + w(u,v)
14                previous[v] := u
```

如果我们只对在 s 和 t 之间寻找一条最短路径的话，我们可以在第 9 行添加条件如果满足 $u = t$ 的话终止程序。

现在我们可以通过迭代来回溯出 s 到 t 的最短路径

```
1 s := empty sequence  
2 u := t  
3 while defined u  
4     insert u to the beginning of S  
5     u := previous[u]
```

现在序列 S 就是从 s 到 t 的最短路径的顶点集.

Dijkstra 算法的实现二（算法导论）：

```
DIJKSTRA( $G, w, s$ )  
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2  $S \leftarrow \emptyset$   
3  $Q \leftarrow V[G]$  // $V^*O(1)$   
4 while  $Q \neq \emptyset$   
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$  // $\text{EXTRACT-MIN}, V^*O(V), V^*O(\lg V)$   
6      $S \leftarrow S \cup \{u\}$   
7     for each vertex  $v \in \text{Adj}[u]$   
8       do  $\text{RELAX}(u, v, w)$  //松弛技术,  $E^*O(1), E^*O(\lg V)$ 。
```

因为 Dijkstra 算法总是在 $V-S$ 中选择“最轻”或“最近”的顶点插入到集合 S 中，所以我们说它使用了贪心策略。

（贪心算法会在日后的博文中详细阐述）。

=====

二零一一年二月九日更新：

此 Dijkstra 算法最初的时间复杂度为 $O(V^*V+E)$, 源点可达的话, $O(V^*\lg V+E^*\lg V) \Rightarrow O(E^*\lg V)$

当是稀疏图的情况时, $E=V^*V/\lg V$, 算法的时间复杂度可为 $O(V^2)$ 。

但我们知道, 若是斐波那契堆实现优先队列的话, 算法时间复杂度, 则为 $O(V^*\lg V + E)$ 。

三、Dijkstra 算法的执行速度

我们可以用大 O 符号将 Dijkstra 算法的运行时间表示为边数 m 和顶点数 n 的函数。
Dijkstra 算法最简单的实现方法是用一个链表或者数组来存储所有顶点的集合 Q ,

所以搜索 Q 中最小元素的运算 ($\text{Extract-Min}(Q)$) 只需要线性搜索 Q 中的所有元素。
这样的话算法的运行时间是 $O(E^2)$ 。

对于边数少于 E^2 的稀疏图来说, 我们可以用邻接表来更有效的实现迪科斯彻算法。
同时需要将一个二叉堆或者斐波纳契堆用作优先队列来寻找最小的顶点 (Extract-Min)。

当用到二叉堆时候，算法所需的时间为 $O((V+E)\log E)$ ，
斐波纳契堆能稍微提高一些性能，让算法运行时间达到 $O(V+E\log E)$ 。(此处一月十六日
修正。)

开放最短路径优先 (OSPF, Open Shortest Path First) 算法是迪科斯彻算法在网络路由中的一个具体实现。

与 Dijkstra 算法不同，Bellman-Ford 算法可用于具有负数权值边的图，只要图中不存在总花费为负值且从源点 s 可达的环路即可用此算法（如果有这样的环路，则最短路径不存在，因为沿环路循环多次即可无限制的降低总花费）。

与最短路径问题相关最有名的一个问题是旅行商问题 (Traveling salesman problem)，此类问题要求找出恰好通过所有标点一次且最终回到原点的最短路径。

然而该问题为 NP-完全的；换言之，与最短路径问题不同，旅行商问题不太可能具有多项式时间解法。

如果有已知信息可用来估计某一点到目标点的距离，则可改用 A*搜寻算法，以减小最短路径的搜索范围。

二零一一年二月九日更新：

BFS、DFS、Kruskal、Prim、Dijkstra 算法时间复杂度的比较：

一般说来，我们知道，BFS、DFS 算法的时间复杂度为 $O(V+E)$ ，
最小生成树算法 Kruskal、Prim 算法的时间复杂度为 $O(E \cdot \lg V)$ 。

而 Prim 算法若采用斐波那契堆实现的话，算法时间复杂度为 $O(E+V \cdot \lg V)$ ，当 $|V| \ll |E|$ 时， $E+V \cdot \lg V$ 是一个较大的改进。

// $|V| \ll |E|$, $\Rightarrow O(E+V \cdot \lg V) \ll O(E \cdot \lg V)$, 对吧。:D

Dijkstra 算法，斐波纳契堆用作优先队列时，算法时间复杂度为 $O(V \cdot \lg V + E)$ 。

//看到了吧，与 Prim 算法采用斐波那契堆实现时，的算法时间复杂度是一样的。

所以我们，说，BFS、Prime、Dijkstra 算法是有相似之处的，单从各算法的时间复杂度比较看，就可窥之一二。

四、图文解析 Dijkstra 算法

ok，经过上文有点繁杂的信息，你还并不对此算法了如指掌，清晰透彻。

没关系，咱们来幅图，就好了。请允许我再对此算法的概念阐述下，

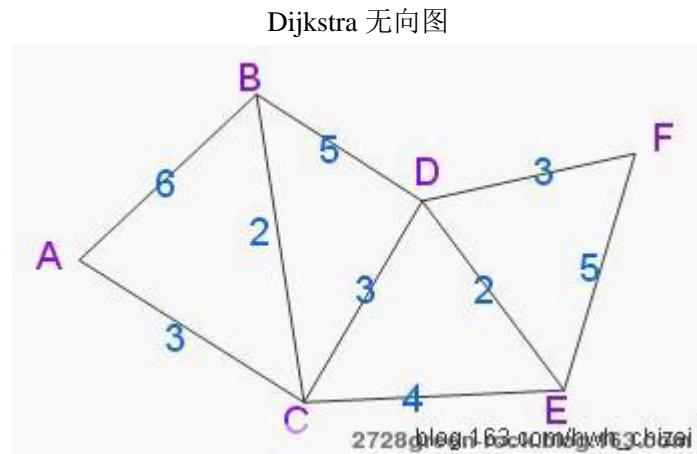
Dijkstra 算法是典型最短路径算法，用于计算一个节点到其他所有节点的最短路径。
不过，针对的是非负值权边。

主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

[Dijkstra 算法能得出最短路径的最优解，但由于它遍历计算的节点很多，所以效率低。]

ok，请看下图：

如下图，设 A 为源点，求 A 到其他各所有一一顶点（B、C、D、E、F）的最短路径。
线上所标注为相邻线段之间的距离，即权值。
(注：此图为随意所画，其相邻顶点间的距离与图中的目视长度不能一一对应)



算法执行步骤如下表：

步骤	S集合中	U集合中
1	选入 A, 此时 $S = \{A\}$ 此时最短路径 $A \rightarrow A=0$ 以 A 为中间点, 从 A 开始找	$U = \{B, C, D, E, F\}$ $A \rightarrow B=6$ $A \rightarrow C=3$ $A \rightarrow$ 其他 U 中的顶点 = ∞ 发现 $A \rightarrow C=3$ 权值为最短
2	选入 C, 此时 $S = \{A, C\}$ 此时最短路径 $A \rightarrow A=0, A \rightarrow C=3$ 以 C 为中间点, 从 $A \rightarrow C=3$ 这条最短路径开始找	$U = \{B, D, E, F\}$ $A \rightarrow C \rightarrow B=5$ (比上面第一步的 $A \rightarrow B=6$ 要短) 此时到 B 权值为 $A \rightarrow C \rightarrow B=5$ $A \rightarrow C \rightarrow D=6$ $A \rightarrow C \rightarrow E=7$ $A \rightarrow C \rightarrow$ 其他 U 中的顶点 = ∞ 发现 $A \rightarrow C \rightarrow B=5$ 权值为最短
3	选入 B, 此时 $S = \{A, C, B\}$ 此时最短路径 $A \rightarrow A=0, A \rightarrow C=3, A \rightarrow C \rightarrow B=5$ 以 B 为中间点, 从 $A \rightarrow C \rightarrow B=5$ 这条最短路径开始找	$U = \{D, E, F\}$ $A \rightarrow C \rightarrow B \rightarrow D=10$ (比上面第二步的 $A \rightarrow C \rightarrow D=6$ 要长) 此时到 D 权值更改为 $A \rightarrow C \rightarrow D=6$ $A \rightarrow C \rightarrow B \rightarrow$ 其他 U 中的顶点 = ∞ 发现 $A \rightarrow C \rightarrow D=6$ 权值为最短
4	选入 D, 此时 $S = \{A, C, B, D\}$ 此时最短路径 $A \rightarrow A=0, A \rightarrow C=3, A \rightarrow C \rightarrow B=5, A \rightarrow C \rightarrow D=6$ 以 D 为中间点, 从 $A \rightarrow C \rightarrow D$ 这条最短路径开始找	$U = \{E, F\}$ $A \rightarrow C \rightarrow D \rightarrow E=9$ (比上面第二步的 $A \rightarrow C \rightarrow E=7$ 要长) 此时到 E 权值更改为 $A \rightarrow C \rightarrow E=7$ $A \rightarrow C \rightarrow D \rightarrow F=9$ 发现 $A \rightarrow C \rightarrow E=7$ 权值为最短
5	选入 E, 此时 $S = \{A, C, B, D, E\}$ 此时最短路径 $A \rightarrow A=0, A \rightarrow C=3, A \rightarrow C \rightarrow B=5, A \rightarrow C \rightarrow D=6, A \rightarrow C \rightarrow E=7$ 以 E 为中间点, 从 $A \rightarrow C \rightarrow E=7$ 这条最短路径开始找	$U = \{F\}$ $A \rightarrow C \rightarrow E \rightarrow F=12$ (比上面第四步的 $A \rightarrow C \rightarrow D \rightarrow F=9$ 要长) 此时到 F 权值更改为 $A \rightarrow C \rightarrow D \rightarrow F=9$ 发现 $A \rightarrow C \rightarrow D \rightarrow F=9$ 权值为最短
6	选入 F, 此时 $S = \{A, C, B, D, E, F\}$ 此时最短路径 $A \rightarrow A=0, A \rightarrow C=3, A \rightarrow C \rightarrow B=5, A \rightarrow C \rightarrow D=6, A \rightarrow C \rightarrow E=7, A \rightarrow C \rightarrow D \rightarrow F=9$	U 集合已空, 查找完毕。 blog.163.com/bwh_zhizai

是不是对此 Dijkstra 算法有不错的了解了。那么，此文也完了。:D。

---July、2010 年 12 月 24 日。平安夜。

=====

此文，写的实在不怎么样。不过，承蒙大家厚爱，此经典算法研究系列的后续文章，个人觉得写的还行。

所以，还请，各位可关注此算法系列的后续文章。谢谢。

二之续、彻底理解 Dijkstra 算法

作者: July 二零一一年二月十三日。

参考代码: introduction to algorithms, Second Edition.

了解什么是 Dijkstra 算法, 请参考:

经典算法研究系列: 二、Dijkstra 算法初探

http://blog.csdn.net/v_JULY_v/archive/2010/12/24/6096981.aspx

本文由单源最短路径问题开始, 而后描述 Bellman-Ford 算法, 到具体阐述 Dijkstra 算法, 阐述详细剖析 Dijkstra 算法的每一个步骤, 教你彻底理解此 Dijkstra 算法。

一、单源最短路径问题

我们知道, 单源最短路径问题: 已知图 $G=(V, E)$, 要求找出从某个定源顶点 $s \in V$, 到每个 $v \in V$ 的最短路径。

简单来说, 就是一个图 G 中, 找到一个定点 s , 然后以 s 为起点, 要求找出 s 到图 G 中其余各个点的最短距离或路径。

此单源最短路径问题有以下几个变形:

I、单终点最短路径问题:

每个顶点 v 到指定终点 t 的最短路径问题。即单源最短路径问题的相对问题。

II、单对顶点最短路径问题:

给定顶点 u 和 v , 找出从 u 到 v 的一条最短路径。

III、每对顶点间最短路径问题:

针对任意每两个顶点 u 和 v , 找出从 u 到 v 的最短路径。

最简单的想法是, 将每个顶点作为源点, 运行一次单源算法即可以解决这个问题。

当然, 还有更好的办法, 日后在本 BOIG 内阐述。

二、Bellman-Ford 算法

1、回路问题

一条最短路径不能包含负权回路, 也不能包含正权回路。

一些最短路径的算法, 如 Dijkstra 算法, 要求图中所有的边的权值都是非负的, 如在公路地图上, 找一条从定点 s 到目的顶点 v 的最短路径问题。

2、Bellman-Ford 算法

而 Bellman-Ford 算法, 则允许输入图中存在负权边, 只要不存在从源点可达的负权回路, 即可。

简单的说, 图中可能存在负权边, 但此条负权边, 构不成负权回路, 不影响回路的形成。且, Bellman-Ford 算法本身, 便是可判断图中是否存在从源点可达的负权回路, 若存在负权回路, 算法返回 FALSE, 若不存在, 返回 TRUE。

Bellman-Ford 算法的具体描述

BELLMAN-FORD(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE(G, s) //对每个顶点初始化，O(V)
2 for i ← 1 to |V[G]| - 1
3   do for each edge (u, v) ∈ E[G]
4     do RELAX(u, v, w) //针对每个顶点(V-1个)，都运用松弛技术 O(E)，计
为 O ((v-1)*E)
5   for each edge (u, v) ∈ E[G]
6     do if d[v] > d[u] + w(u, v)
7       then return FALSE //检测图中每条边，判断是否包含负权回路，
//若 d[v]>d[u]+w(u, v)，则表示包含，返回 FALSE,
8   return TRUE           //不包含负权回路，返回 TRUE
```

Bellman-Ford 算法的时间复杂度，由上可得为 $O (V * E)$ 。

3、关于判断图中是否包含负权回路的问题：

根据定理，我们假定， u 是 v 的父辈，或父母，那么

当 $G(V,E)$ 是一个有向图或无向图(且不包含任何负权回路)， $s \in V$ ， s 为 G 的任意一个顶点，则对任意边 $(u, v) \in V$ ，有

$$d[s, v] \leq d[s, u] + 1$$

此定理的详细证明，可参考算法导论一书上，第 22 章中引理 22.1 的证明。

或者根据第 24 章中通过三角不等式论证 Bellman-Ford 算法的正确性，也可得出上述定理的变形。

即假设图 G 中不包含负权回路，可证得

$$\begin{aligned} d[v] &= d(s, u) \\ &\leq d(s, u) + w(u, v) //根据三角不等式 \\ &= d[u] + w[u, v] \end{aligned}$$

所以，在不包含负权回路的图中，是可以得出 $d[v] \leq d[u] + w(u, v)$ 。

于是，就不难理解，在上述 Bellman-Ford 算法中，

```
if d[v] > d[u] + w(u, v), => 包含负权回路，返回 FALSE
else if => 不包含负权回路，返回 TRUE。
```

ok，咱们，接下来，立马切入 Dijkstra 算法。

三、深入浅出，彻底解剖 Dijkstra 算法

I、松弛技术 RELAX 的介绍

Dijkstra 算法使用了松弛技术，对每个顶点 $v \in V$ ，都设置一个属性 $d[v]$ ，用来描述从源点 s 到 v 的最短路径上权值的上界，称为最短路径的估计。

首先，得用 $O(V)$ 的时间，来对最短路径的估计，和对前驱进行初始化工作。

INITIALIZE-SINGLE-SOURCE(G, s)

```
1 for each vertex  $v \in V[G]$ 
2   do  $d[v] \leftarrow \infty$ 
3    $\pi[v] \leftarrow \text{NIL}$     // $O(V)$ 
4  $d[s] 0$ 
```

RELAX(u, v, w)

```
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$       // $O(E)$ 
```

图。

II、Dijkstra 算法

此 Dijkstra 算法分三个步骤，

INSERT (第 3 行), EXTRACT-MIN (第 5 行), 和 DECREASE-KEY(第 8 行的 RELAX, 调用此减小关键字的操作)。

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ ) //对每个顶点初始化， $O(V)$ 
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$           //INSERT,  $O(1)$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$     //简单的  $O(V^2)$ ; 二叉/项堆，和 FIB-HEAP
        的话，则都为  $O(V \cdot \lg V)$ .
6    $S \leftarrow S \cup \{u\}$ 
7   for each vertex  $v \in \text{Adj}[u]$ 
8     do RELAX( $u, v, w$ )    //简单方式： $O(E)$ ，二叉/项堆， $E \cdot O(\lg V)$ ，
        FIB-HEAP， $E \cdot O(1)$ .
```

四、Dijkstra 算法的运行时间

在继续阐述之前，得先声明一个问题，DIJKSTRA (G, w, s) 算法中的第 5 行， $\text{EXTRACT-MIN}(Q)$ ，最小优先队列的具体实现。

而 Dijkstra 算法的运行时间，则与此最小优先队列的采取何种具体实现，有关。

最小优先队列三种实现方法：

1、利用从 1 至 $|V|$ 编好号的顶点，简单地将每一个 $d[v]$ 存入一个数组中对应的第 v 项，如上述 DIJKSTRA (G, w, s) 所示，Dijkstra 算法的运行时间为 $O(V^2+E)$ 。

2、如果是二叉/项堆实现最小优先队列的话， $\text{EXTRACT-MIN}(Q)$ 的运行时间为 $O(V \cdot \lg V)$ ，所以，Dijkstra 算法的运行时间为 $O(V \cdot \lg V + E \cdot \lg V)$ ，

若所有顶点都是从源点可达的话， $O((V+E)*\lg V) = O(E*\lg V)$ 。

当是稀疏图时，则 $E=O(V^2/\lg V)$ ，此 Dijkstra 算法的运行时间为 **$O(V^2)$** 。

3、采用斐波那契堆实现最小优先队列的话，EXTRACT-MIN(Q)的运行时间为 $O(V*\lg V)$ ，所以，此 Dijkstra 算法的运行时间即为 **$O(V*\lg V+E)$** 。

综上所述，此最小优先队列的三种实现方法比较如下：

- EXTRACT-MIN + RELAX**
- I、简单方式： $O(V^2 + E)$
 - II、二叉/项堆： $O(V*\lg V + |E|*\lg V)$
 - 源点可达： $O(E*\lg V)$
 - 稀疏图时，有 $E=o(V^2/\lg V)$ ，
 $\Rightarrow O(V^2)$
 - III、斐波那契堆： $O(V*\lg V + E)$

当 $|V| << |E|$ 时，采用 DIJKSTRA(G, w, s) + FIB-HEAP-EXTRACT-MIN(Q)，即斐波那契堆实现最小优先队列的话，优势就体现出来了。

五、Dijkstra 算法 + FIB-HEAP-EXTRACT-MIN(H)，斐波那契堆实现最小优先队列

由以上内容，我们已经知道，用斐波那契堆来实现最小优先队列，可以将运行时间提升到 $O(V\lg V+E)$ 。

$|V|$ 个 EXTRACT-MIN 操作，每个平摊代价为 $O(\lg V)$ ， $|E|$ 个 DECREASE-KEY 操作的每个平摊时间为 $O(1)$ 。

下面，重点阐述 DIJKSTRA(G, w, s) 中，斐波那契堆实现最小优先队列的操作。

由上，我们已经知道，DIJKSTRA 算法包含以下的三个步骤：

INSERT(第 3 行), **EXTRACT-MIN**(第 5 行), 和 **DECREASE-KEY**(第 8 行的 **RELAX**)。

先直接给出 Dijkstra 算法 + FIB-HEAP-EXTRACT-MIN(H) 的算法：

DIJKSTRA(G, w, s)

- 1 **INITIALIZE-SINGLE-SOURCE(G, s)**
- 2 $S \leftarrow \emptyset$
- 3 $Q \leftarrow V[G]$ // 第 3 行, **INSERT** 操作, $O(1)$
- 4 **while** $Q \neq \emptyset$
- 5 **do** $u \leftarrow \text{EXTRACT-MIN}(Q)$ // 第 5 行, **EXTRACT-MIN** 操作, $V*\lg V$
- 6 $S \leftarrow S \cup \{u\}$
- 7 **for each vertex** $v \in \text{Adj}[u]$
- 8 **do** **RELAX**(u, v, w) // 第 8 行, **RELAX** 操作, $E*O(1)$

FIB-HEAP-EXTRACT-MIN(H) // 平摊代价为 $O(\lg V)$

- 1 $z \leftarrow \min[H]$
- 2 **if** $z \neq \text{NIL}$

```

3   then for each child x of z
4       do add x to the root list of H
5       p[x] ← NIL
6       remove z from the root list of H
7       if z = right[z]
8           then min[H] ← NIL
9       else min[H] ← right[z]
10      CONSOLIDATE(H)
11      n[H] ← n[H] - 1
12  return z

```

六、Dijkstra 算法 +fibonacci 堆各项步骤的具体分析

ok, 接下来, 具体分步骤阐述以上各个操作:

第 3 行的 **INSERT** 操作:

FIB-HEAP-INSERT(H, x) //平摊代价, O (1) .

```

1 degree[x] ← 0
2 p[x] ← NIL
3 child[x] ← NIL
4 left[x] ← x
5 right[x] ← x
6 mark[x] ← FALSE
7 concatenate the root list containing x with root list H
8 if min[H] = NIL or key[x] < key[min[H]]
9     then min[H] ← x
10    n[H] ← n[H] + 1

```

第 5 行的 **EXTRACT-MIN** 操作:

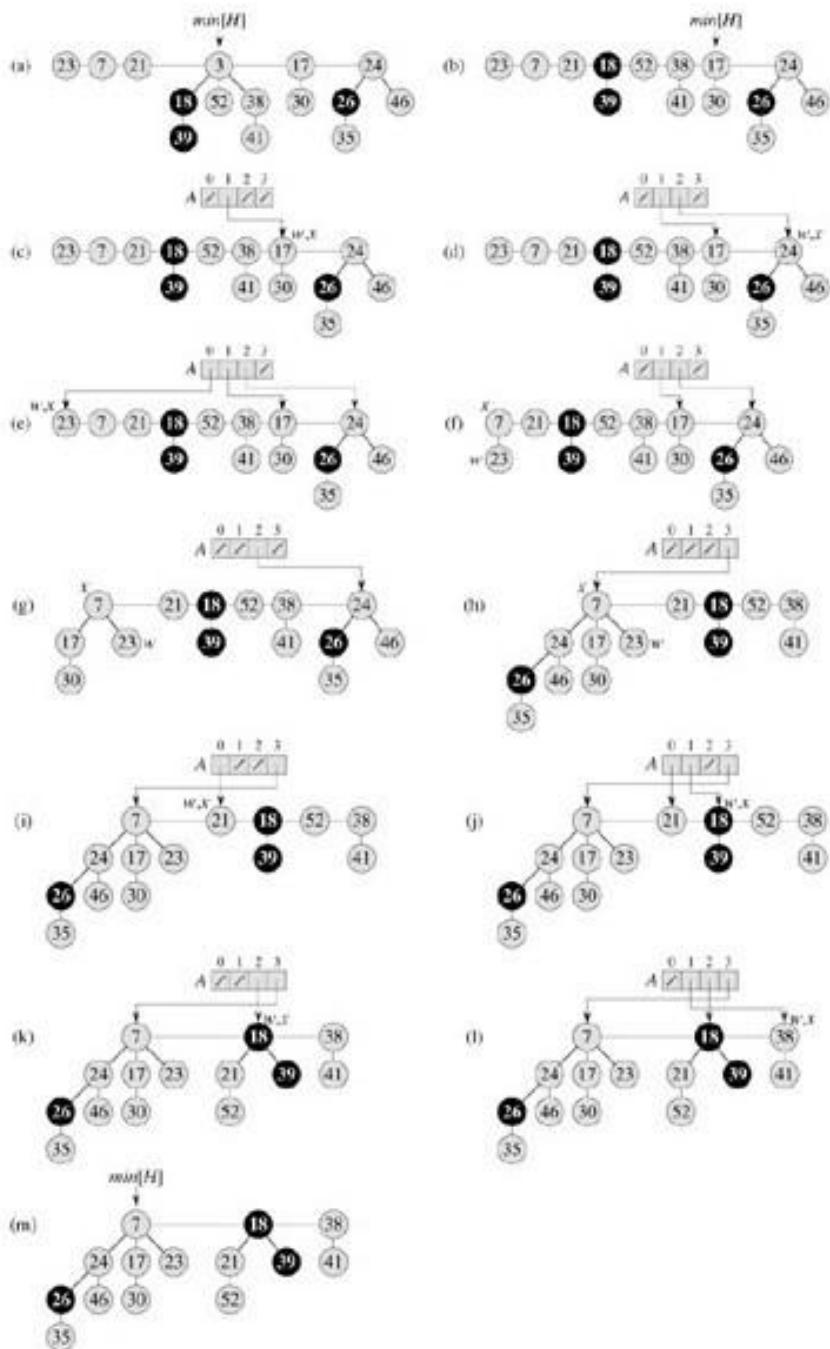
FIB-HEAP-EXTRACT-MIN(H) //平摊代价为 O (lgV)

```

1 z ← min[H]
2 if z ≠ NIL
3     then for each child x of z
4         do add x to the root list of H
5         p[x] ← NIL
6     remove z from the root list of H
7     if z = right[z]
8         then min[H] ← NIL
9     else min[H] ← right[z]
10    CONSOLIDATE(H) //CONSOLIDATE 算法在下面, 给出。
11    n[H] ← n[H] - 1
12  return z

```

下图是 FIB-HEAP-EXTRACT-MIN 的过程示意图:



CONSOLIDATE(H)

```

1 for i ← 0 to D(n[H])
2   do A[i] ← NIL
3 for each node w in the root list of H
4   do x ← w
5     d ← degree[x]      //子数
6     while A[d] ≠ NIL
7       do y ← A[d]

```

```

8      if key[x] > key[y]
9          then exchange x <-> y
10         FIB-HEAP-LINK(H, y, x) //下面给出。
11         A[d] ← NIL
12         d ← d + 1
13         A[d] ← x
14 min[H] ← NIL
15 for i ← 0 to D(n[H])
16   do if A[i] ≠ NIL
17     then add A[i] to the root list of H
18     if min[H] = NIL or key[A[i]] < key[min[H]]
19       then min[H] ← A[i]

```

FIB-HEAP-LINK(H, y, x) //y 链接至 x。

- 1 remove y from the root list of H
- 2 make y a child of x, incrementing degree[x]
- 3 mark[y] ← FALSE

第 8 行的 **RELAX** 的操作，已上已经给出：

RELAX(u, v, w)

- 1 if $d[v] > d[u] + w(u, v)$
- 2 then $d[v] \leftarrow d[u] + w(u, v)$
- 3 $n[v] \leftarrow u$ //O (E)

一般来说，在 Dijkstra 算法中，DECREASE-KEY 的调用次数远多于 EXTRACT-MIN 的调用，所以在不增加 EXTRACT-MIN 操作的平摊时间前提下，尽量减小 DECREASE-KEY 操作的平摊时间，都能获得对比二叉堆更快的实现。

以下，是二叉堆，二项堆，斐波那契堆的各项操作的时间复杂度的比较：

操作	二叉堆(最坏)	二项堆(最坏)	斐波那契堆(平摊)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

斐波那契堆，日后会在本 BLOG 内，更进一步的深入与具体阐述。

且同时，此文，会不断的加深与扩展。

完。

本人 **July** 对本博客所有任何文章、内容和资料享有版权。
转载务必注明作者本人及出处，并通知本人。谢谢。

July、二零一一年二月十三日。

二之再续、**Dijkstra** 算法+fibonacci 堆的逐步 c 实现

作者:JULY、二零一一年三月十八日

出处: http://blog.csdn.net/v_JULY_v

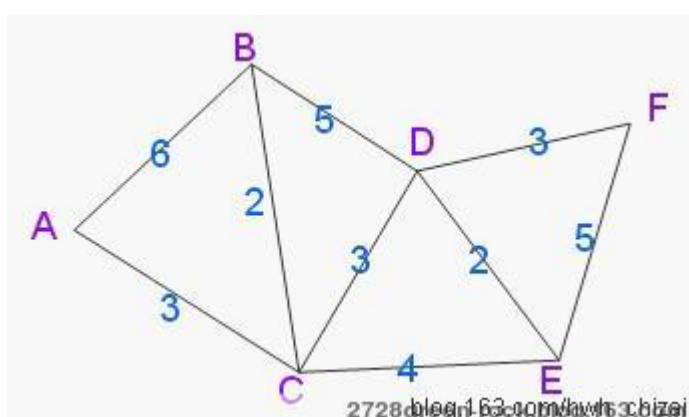
引言:

来考虑一个问题，

平面上 6 个点, A,B,C,D,E,F, 假定已知其中一些点之间的距离,

现在, 要求 A 到其它 5 个点, B,C,D,E,F 各点的最短距离。

如下图所示:



经过上图, 我们可以轻而易举的得到 A->B,C,D,E,F 各点的最短距离:

目的	路径	最短距离
A=>A,	A->A	0
A=>B,	A->C->B	3+2=5
A=>C,	A->C	3
A=>D,	A->C->D	3+3=6
A=>E,	A->C->E	3+4=7
A=>F,	A->C->D->F	3+3+3=9

我想，如果是单单出上述一道填空题，要你答出 A->B,C,D,E,F 各点的最短距离，一个小学生，掰掰手指，也能在几分钟之内，填写出来。

我们的问题，当然不是这么简单，上述只是一个具体化的例子而已。实际上，很多的问题，如求图的最短路径问题，就要用到上述方法，不断比较、不断寻找，以期找到最短距离的路径，此类问题，便是 Dijkstra 算法的应用了。当然，还有 BFS 算法，以及更高效的 A* 搜索算法。

A* 搜索算法已在本 BLOG 内有所详细的介绍，本文咱们结合 fibonacci 堆实现 Dijkstra 算法。

即，Dijkstra + fibonacci 堆 c 实现。

我想了下，把一个算法研究够透彻之后，还要编写代码去实现它，才叫真正掌握了一个算法。本 BLOG 内经典算法研究系列，已经写了 **18 篇** 文章，十一个算法，所以，还有 **10** 多个算法，待我去实现。

代码风格

实现一个算法，首先要了解此算法的原理，了解此算法的原理之后，便是写代码实现。在打开编译器之前，我先到网上搜索了一下“Dijkstra 算法+fibonacci 堆实现”。

发现：网上竟没有过 Dijkstra + fibonacci 堆实现的 c 代码，而且如果是以下几类的代码，我是直接跳过不看的：

- 1、没有注释(看不懂)。
- 2、没有排版(不舒服)。
- 3、冗余繁杂(看着烦躁)。

fibonacci 堆实现 Dijkstra 算法

ok，闲话少说，咱们切入正题。下面，咱们来一步一步利用 fibonacci 堆实现 Dijkstra 算法吧。

前面说了，要实现一个算法，首先得明确其算法原理及思想，而要理解一个算法的原理，又得知道发明此算法的目的是什么，即，此算法是用来干什么的？

由前面的例子，我们可以总结出：Dijkstra 算法是为了解决一个点到其它点最短距离的问题。

我们总是要找源点到各个目标点的最短距离，在寻路过程中，如果新发现了一个新的点，发现当源点到达前一个目的点路径通过新发现的点时，路径可以缩短，那么我们就必须及时更新此最短距离。

ok, 举个例子：如我们最初找到一条路径，A->B，这条路径的最短距离为 6，后来找到了 C 点，发现若 A->C->B 点路径时，A->B 的最短距离为 5，小于之前找到的最短距离 6，所以，便得此更新 A 到 B 的最短距离：为 5，最短路径为 A->C->B.

好的，明白了此算法是干什么的，那么咱们先用伪代码尝试写一下吧(有的人可能会说，不是吧，我现在，什么都还没搞懂，就要我写代码了。额，你手头不是有资料么，如果全部所有的工作，都要自己来做的话，那就是一个浩大的工程了。:D。)。

咱们先从算法导论上，找来 Dijkstra 算法的伪代码如下：

```
DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s) //1、初始化结点工作
2 S ← Ø
3 Q ← V[G] //2、插入结点操作
4 while Q ≠ Ø
5   do u ← EXTRACT-MIN(Q) //3、从最小队列中，抽取最小点工作
6     S ← S ∪ {u}
7     for each vertex v ∈ Adj[u]
8       do RELAX(u, v, w) //4、松弛操作。
```

伪代码毕竟与能在机子上编译运行的代码，还有很多工作要做。

首先，咱们看一下上述伪代码，可以看出，基本上，此 Dijkstra 算法主要分为以下四个步骤：

- 1、初始化结点工作
- 2、插入结点操作
- 3、从最小队列中，抽取最小点工作
- 4、松弛操作。

ok，由于第 2 个操作涉及到斐波那契堆，比较复杂一点，咱们先来具体分析第 1、2、4 个操作：

1、得用 O (V) 的时间，来对最短路径的估计，和对前驱进行初始化工作。

```
INITIALIZE-SINGLE-SOURCE(G, s)
1 for each vertex v ∈ V[G]
2   do d[v] ← ∞
3   n[v] ← NIL //O (V)
4 d[s] 0
```

我们根据上述伪代码，不难写出以下的代码：

```
void init_single_source(Graph *G,int s)
{
    for (int i=0;i<G->n;i++) {
        d[i]=INF;
```

```

    pre[i]=-1;
}
d[s]=0;
}

```

2、插入结点到队列的操作

2 $S \leftarrow \emptyset$
 3 $Q \leftarrow V[G]$ //2、插入结点操作

代码:

```

for (i=0;i<G->n;i++)
S[i]=0;

```

4、松弛操作。

首先得理解什么是松弛操作:

Dijkstra 算法使用了松弛技术, 对每个顶点 $v \in V$, 都设置一个属性 $d[v]$, 用来描述从源点 s 到 v 的最短路径上权值的上界, 称为最短路径的估计。

```

RELAX(u, v, w)
1 if d[v] > d[u] + w(u, v)
2   then d[v] ← d[u] + w(u, v)
3   n[v] ← u      //O (E)

```

同样, 我们不难写出下述代码:

```

void relax(int u,int v,Graph *G)
{
    if (d[v]>d[u]+G->w[u][v])
    {
        d[v] = d[u]+G->w[u][v]; //更新此最短距离
        pre[v]=u;    //u 为 v 的父结点
    }
}

```

再解释一下上述 **relax** 的代码, 其中 u 为 v 的父结点, 当发现其父结点 $d[u]$

加上经过路径的距离 $G->w[u][v]$, 小于子结点到源点的距离 $d[v]$, 便得更新此最短距离。

请注意, 说的明白点: 就是本来最初 A 到 B 的路径为 A->B, 现在发现, 当 A 经过 C 到达 B 时, 此路径距离比 A->B 更短, 当然, 便得更新此 A 到 B 的最短路径了, 即是: A->C->B, C 即成为了 B 的父结点(如此解释, 我相信您已经明朗。:D。)。

即 A=>B <==> A->C->B, 执行赋值操作。

ok, 第 1、2、4 个操作步骤, 咱们都已经写代码实现了, 那么, 接下来, 咱们来编写第 3 个操作的代码: 3、从最小队列中, 抽取最小点工作。

相信，你已经看出来了，我们需要构造一个最小优先队列，那用什么来构造最小优先队列呢？对了，堆。什么堆最好，效率最高，呵呵，就是本文要实现的 fibonacci 堆。

为什么？OK，请看最小优先队列的三种实现方法比较：

- EXTRACT-MIN + RELAX
- I、简单方式： $O(V^*V + E^*1)$
 - II、二叉/项堆： $O(V^*\lg V + |E|*\lg V)$
源点可达： $O(E^*\lg V)$
稀疏图时，有 $E=o(V^2/\lg V)$ ，
 $\Rightarrow O(V^2)$
 - III、斐波那契堆： $O(V^*\lg V + E)$

其中， V 为顶点， E 为边。好的，这样我们就知道了：Dijkstra 算法中，当用斐波那契堆作优先队列时，算法时间复杂度为 $O(V^*\lg V + E)$ 。

额，那么接下来，咱们要做的是什么列？当然是要实现一个 fibonacci 堆了。可要怎么实现它，才能用到我们 Dijkstra 算法中呢？对了，写成一个库的形式。库？呵呵，是一个类。

OK，以下就是这个 fibonacci 堆的实现：

[view plain](#) [copy to clipboard](#) [print](#)

```
1. //FibonacciHeap.h
2. #ifndef _FIBONACCI_HEAP_H_INCLUDED_
3. #define _FIBONACCI_HEAP_H_INCLUDED_
4.
5. #include <functional>
6. #include <algorithm>
7.
8. template<typename T>
9. struct Fib_node
10. {
11.     Fib_node* ns_; //后驱结点
12.     Fib_node *pt_; //父母结点
13.     Fib_node* ps_; //前驱结点
14.     Fib_node* fc_; //头结点
15.     int rank_; //孩子结点
16.     bool marked_; //孩子结点是否删除的标记
17.     T* pv_;
18.     Fib_node(T* pv = 0) : pv_(pv) { }
19.     T& value(void) { return *pv_; }
20.     void set_src(T* pv) { pv_ = pv; }
21. }; //Fib_node 的数据结构
22.
```

```

23.
24. template<class Node, class OD>
25. Node* merge_tree(Node*a, Node* b, OD small) //合并结点
26. {
27.     if(small(b->value(), a->value()))
28.         swap(a, b);
29.     Node* fc = a->fc_;
30.     a->fc_ = b;
31.     a->ns_ = a->ps_ = a->pt_ = 0;
32.     ++a->rank_;
33.
34.     b->pt_ = a; //a 为 b 的父母
35.     b->ns_ = fc; //第一个结点赋给 b 的前驱结点
36.     b->ps_ = 0;
37.     if(fc != 0)
38.         fc->ps_ = b;
39.     return a;
40. }
41.
42. template<typename Node>
43. void erase_node(Node* me) //删除结点
44. {
45.     Node* const p = me->pt_;
46.     --p->rank_;
47.     if(p->fc_ == me) //如果 me 是头结点
48.     {
49.         if((p->fc_ = me->ns_) != 0)
50.             me->ns_->ps_ = 0;
51.     }
52.     else
53.     {
54.         Node *prev = me->ps_;
55.         Node *next = me->ns_; //可能为 0
56.         prev->ns_ = next;
57.         if(next != 0)
58.             next->ps_ = prev;
59.     }
60. }
61.
62.
63. template<class Node, class OD>
64. Node* merge_fib_heap(Node* a, Node* b, OD small) //调用上述的 merge_tree 合并
   fib_heap.
65. {

```

```

66. enum {SIZE = 64};    //
67. Node* v[SIZE] = {0};
68. int k;
69. while(a != 0)
70. {
71.     Node* carry = a;
72.     a = a->ns_;
73.     for(k = carry->rank_; v[k] != 0; ++k)
74.     {
75.         carry = merge_tree(carry, v[k], small);
76.         v[k] = 0;
77.     }
78.     v[k] = carry;
79. }
80. while(b != 0)
81. {
82.     Node* carry = b;
83.     b = b->ns_;
84.     for(k = carry->rank_; v[k] != 0; ++k)
85.     {
86.         carry = merge_tree(carry, v[k], small);
87.         v[k] = 0;
88.     }
89.     v[k] = carry;
90. }
91. Node** t = std::remove(v, v+SIZE, (Node*)0);
92. int const n = t - v;
93. if(n > 0)
94. {
95.     for(k = 0; k < n - 1; ++k)
96.         v[k]->ns_ = v[k+1];
97.     for(k = 1; k < n; ++k)
98.         v[k]->ps_ = v[k-1];
99.     v[n-1]->ns_ = v[0]->ps_ = 0;
100. }
101. return v[0];
102. }
103.
104. template<typename T, class OD = std::less<T> >
105. struct Min_fib_heap //抽取最小结点
106. {
107.     typedef Fib_node<T> Node;
108.     typedef Node Node_type;
109.

```

```

110. Node* roots_;
111. Node* min_; //pointer to the minimum node
112. OD less_;
113.
114. Min_fib_heap(void): roots_(0), min_(0), less_() { }
115. bool empty(void) const { return roots_ == 0; }
116. T& top(void) const { return min_->value(); }
117.
118. void decrease_key(Node* me) //删除
119. { //precondition: root_ not zero
120.   if(less_(me->value(), min_->value()))
121.     min_ = me;
122.   cascading_cut(me);
123. }
124. void push(Node* me) //压入
125. {
126.   me->pt_ = me->fc_ = 0;
127.   me->rank_ = 0;
128.   if(roots_ == 0)
129.   {
130.     me->ns_ = me->ps_ = 0;
131.     me->marked_ = false;
132.     roots_ = min_ = me;
133.   }
134.   else
135.   {
136.     if(less_(me->value(), min_->value()))
137.       min_ = me;
138.     insert2roots(me);
139.   }
140. }
141. Node* pop(void) //弹出
142. {
143.   Node* const om = min_;
144.   erase_tree(min_);
145.   min_ = roots_ = merge_fib_heap(roots_, min_->fc_, less_);
146.   if(roots_ != 0) //find new min_
147.   {
148.     for(Node* t = roots_->ns_; t != 0; t = t->ns_)
149.       if(less_(t->value(), min_->value()))
150.         min_ = t;
151.   }
152.   return om;
153. }

```

```

154. void merge(void) //合并
155. {
156.     if(empty()) return;
157.     min_ = roots_ = merge_fib_heap(roots_, (Node*)0, less_);
158.     for(Node* a = roots_->ns_; a != 0; a = a->ns_)
159.         if(less_(a->value(), min_->value() ))
160.             min_ = a;
161. }
162. private:
163. void insert2roots(Node* me) //插入
164. { //precondition: 1) root_ != 0; 2) me->value() >= min_->value()
165.     me->pt_ = me->ps_ = 0;
166.     me->ns_ = roots_;
167.     me->marked_ = false;
168.     roots_->ps_ = me;
169.     roots_ = me;
170. }
171. void cascading_cut(Node* me) //断开
172. { //precondition: me is not a root. that is me->pt_ != 0
173.     for(Node* p = me->pt_; p != 0; me = p, p = p->pt_)
174.     {
175.         erase_node(me);
176.         insert2roots(me);
177.         if(p->marked_ == false)
178.         {
179.             p->marked_ = true;
180.             break;
181.         }
182.     }
183. }
184. void erase_tree(Node* me) //删除
185. {
186.     if(roots_ == me)
187.     {
188.         roots_ = me->ns_;
189.         if(roots_ != 0)
190.             roots_->ps_ = 0;
191.     }
192.     else
193.     {
194.         Node* const prev = me->ps_;
195.         Node* const next = me->ns_;
196.         prev->ns_ = next;
197.         if(next != 0)

```

```

198.     next->ps_ = prev;
199. }
200. }
201. }; //Min_fib_heap 的类
202.
203.
204. template<typename Fitr>
205. bool is_sorted(Fitr first, Fitr last)
206. {
207.     if(first != last)
208.         for(Fitr prev = first++; first != last; prev = first++)
209.             if(*first < *prev) return false;
210.     return true;
211. }
212. template<typename Fitr, class OD>
213. bool is_sorted(Fitr first, Fitr last, OD cmp)
214. {
215.     if(first != last)
216.         for(Fitr prev = first++; first != last; prev = first++)
217.             if(cmp(*first, *prev)) return false;
218.     return true;
219. }

```

由于本 BLOG 日后会具体阐述这个斐波那契堆的各项操作，限于篇幅，在此，就不再啰嗦解释上述程序了。

ok，实现了 fibonacci 堆，接下来，咱们可以写 Dijkstra 算法的代码了。为了版述清晰，再一次贴一下此算法的伪代码：

```

DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← ∅
3 Q ← V[G] //第 3 行, INSERT 操作, O (1)
4 while Q ≠ ∅
5     do u ← EXTRACT-MIN(Q) //第 5 行, EXTRACT-MIN 操作, V*IgV
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v, w) //第 8 行, RELAX 操作, E*O(1)

```

编写的 Dijkstra 算法的 c 代码如下：

[view plain](#)[copy to clipboard](#)[print](#)

```

1. void Dijkstra(int s, T d[], int p[])
2. {
3.     //寻找从顶点 s 出发的最短路径,在 d 中存储的是 s->i 的最短距离

```

```

4.      //p 中存储的是 i 的父节点
5.      if (s < 1 || s > n)
6.          throw OutOfBounds();
7.
8.      //路径可到达的顶点列表,这里可以用上述实现的 fibonacci 堆代码。
9.      Chain<int> L;
10.
11.     ChainIterator<int> I;
12.     //初始化 d, p, and L
13.     for (int i = 1; i <= n; i++)
14.     {
15.         d[i] = a[s][i];
16.
17.         if (d[i] == NoEdge)
18.         {
19.             p[i] = 0;
20.         }
21.         else
22.         {
23.             p[i] = s;
24.             L.Insert(0,i);
25.         }
26.     }
27.
28.     //更新 d, p
29.     while (!L.IsEmpty())
30.     {
31.         //寻找最小 d 的点 v
32.         int *v = I.Initialize(L);
33.         int *w = I.Next();
34.         while (w)
35.         {
36.             if (d[*w] < d[*v])
37.                 v = w;
38.
39.             w = I.Next();
40.         }
41.
42.         int i = *v;
43.         L.Delete(*v);
44.         for (int j = 1; j <= n; j++)
45.         {
46.             if (a[i][j] != NoEdge
47.                 && (!p[j] || d[j] > d[i] + a[i][j])) //d[i]是父节点

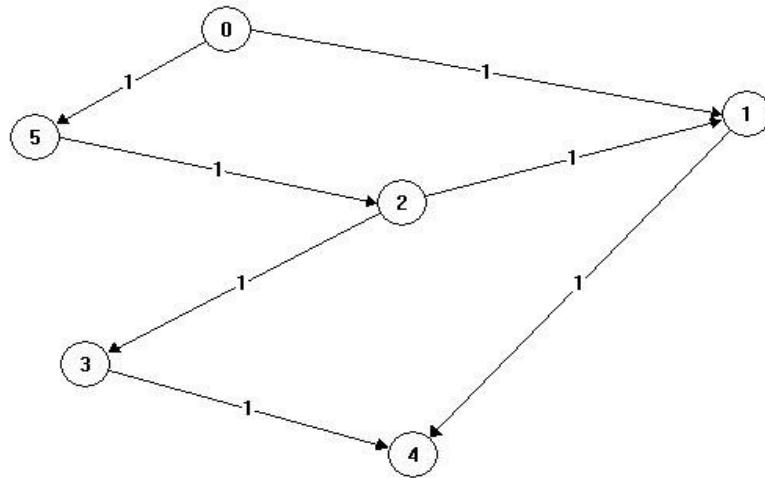
```

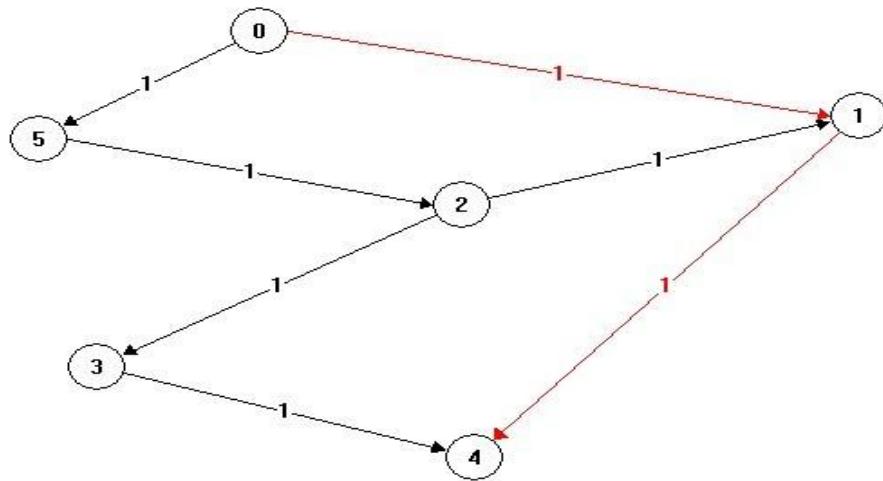
```

48.         {
49.             // 刷新更小的 d[j]
50.             d[j] = d[i] + a[i][j];
51.
52.             // 如果 j 没有父节点,则添加到 L
53.             if (!p[j])
54.                 L.Insert(0,j);
55.
56.             // 更新父节点
57.             p[j] = i;
58.         }
59.     }
60. }
61. }
```

更好的代码，还在进一步修正中。日后，等完善好后，再发布整个工程出来。

下面是演示此 Dijkstra 算法的工程的俩张图（0 为源点，4 为目标点，第二幅图中的红色路径即为所求的 0->4 的最短距离的路径）：





完。

版权所有。转载本 BLOG 内任何文章，请以超链接形式注明出处。

谢谢，各位。

二之三续、Dijkstra 算法+Heap 堆的完整 c 实现源码

作者:JULY、二零一一年三月十八日

出处: http://blog.csdn.net/v_JULY_v。

引言:

此文的写作目的很简单，就一个理由，个人认为：上一篇文章，[二之再续、Dijkstra 算法+fibonacci 堆的逐步 c 实现](#)，写的不够好，特此再写 Dijkstra 算法的一个续集，谓之二之三续。

鉴于读者理解斐波那契堆的难度，本文，以简单的最小堆为示例。同时，本程序也有参考网友的实现。有任何问题，欢迎指正。

Dijkstra 算法+Heap 堆完整算法思想

在前一篇文章中，我们已经了解到，Dijkstra 算法如下：

```

DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s) //1、初始化结点工作
2 S ← ∅
3 Q ← V[G] //2、初始化队列
4 while Q ≠ ∅
5   do u ← EXTRACT-MIN(Q) //3、从最小队列中，抽取最小结点(在此之前，先建立最小堆)
6     S ← S ∪ {u}
7     for each vertex v ∈ Adj[u]
8       do RELAX(u, v, w) //4、松弛操作。

```

如此，咱们不再赘述，直接即可轻松编写如下 c/c++源码：

```

void dijkstra(ALGraph G,int s,int d[],int pi[],int Q[])
{ //Q[]是最小优先队列，Q[1..n]中存放的是图顶点标号,Q[0]中存放堆的大小
  //优先队列中有 key 的概念，这里 key 可以从 d[] 中取得。比如说，Q[2]的大小(key)为
  d[ Q[2] ]

  initSingleSource(G,s,d,pi); //1、初始化结点工作

  //2、初始化队列
  Q[0] = G.vexnum;
  for(int i=1;i<=Q[0];i++)
  {

    Q[i] = i-1;
  }
  Q[1] = s;
  Q[s+1] = 0;

  int u;
  int v;
  while(Q[0]!=0)

  {
    buildMinHeap(Q,d); //3.1、建立最小堆
    u = extractMin(Q,d); //3.2、从最小队列中，抽取最小结点
    ArcNode* arcNodePt = G.vertices[u].firstarc;
    while(arcNodePt!=NULL)
    {
      v = arcNodePt->adjvex;
      relax(u,v,G,d,pi); //4、松弛操作。
      arcNodePt = arcNodePt->nextarc;
    }
  }
}

```

}

ok, 接下来, 咱们一步一步编写代码来实现此 Dijkstra 算法, 先给出第 1、初始化结点工作, 和 4、松弛操作两个操作的源码:

```
void initSingleSource(ALGraph G,int s,int d[],int pi[])
{
    //1、初始化结点工作
    for(int i=0;i<G.vexnum;i++)
    {
        d[i] = INFINITY;
        pi[i] = NIL;
    }
    d[s] = 0;
}

void relax(int u,int v,ALGraph G,int d[],int pi[])
{
    //4、松弛操作。
    //u 是新加入集合 S 的顶点的标号
    if(d[v]>d[u]+getEdgeWeight(G,u,v))
    {
        d[v] = d[u] + getEdgeWeight(G,u,v);
        pi[v] = u;
    }
}
```

ok, 接下来, 咱们具体阐述第 3 个操作, 3、从最小队列中, 抽取最小结点(在此之前, 先建立最小堆)。

Heap 最小堆的建立与抽取最小结点

在我的这篇文章[二、堆排序算法](#)里头, 对最大堆的建立有所阐述:

2.3.1、建堆(**O (N)**)

BUILD-MAX-HEAP(A)

```
1 heap-size[A] ← length[A]
2 for i ← ⌊length[A]/2⌋ downto 1
3     do MAX-HEAPIFY(A, i)
//建堆, 怎么建列?原来就是不断的调用 MAX-HEAPIFY(A, i)来建立最大堆。
```

建最小堆, 也是一回事, 把上述代码改俩处即可, 一, MAX->MIN, 二, MAX-HEAPIFY(A, i)->MIN-HEAPIFY(A, i)。如此说来, 是不是很简单列, 是的, 本身就很简单。

先是建立最小堆的工作：

```
void buildMinHeap(int Q[],int d[]) //建立最小堆
{
    for(int i=Q[0]/2;i>=1;i--)
    {
        minHeapify(Q,d,i); //调用 minHeapify，以保持堆的性质。
    }
}
```

然后，得编写 **minHeapify** 代码，来保持最小堆的性质：

```
void minHeapify(int Q[],int d[],int i)
{ //smallest,l,r,i 都是优先队列元素的下标，范围是从 1 ~ heap-size[Q]
    int l = 2*i;
    int r = 2*i+1;
    int smallest;
    if(l<=Q[0] && d[ Q[l] ] < d[ Q[i] ])

    {
        smallest = l;
    }
    else
    {
        smallest = i;
    }
    if(r<=Q[0] && d[ Q[r] ] < d[ Q[smallest] ])

    {
        smallest = r;
    }
    if(smallest!=i)
    {
        int temp = Q[i];
        Q[i] = Q[smallest];
        Q[smallest] = temp;

        minHeapify(Q,d,smallest);
    }
}
```

你自个比较一下建立最小堆，与建立最大堆的代码，立马看见，如出一辙，不过是改几个字母而已：

```
MAX-HEAPIFY(A, i) //建立最大堆的代码
1 | ← LEFT(i)
```

```

2 r ← RIGHT(i)
3 if l ≤ heap-size[A] and A[l] > A[i]
4   then largest ← l
5   else largest ← i
6 if r ≤ heap-size[A] and A[r] > A[largest]
7   then largest ← r
8 if largest ≠ i
9   then exchange A[i] <-> A[largest]
10      MAX-HEAPIFY(A, largest)

```

ok, 最后, 便是 **3、从最小队列中, 抽取最小结点的工作了, 如下:**

```

int extractMin(int Q[],int d[]) //3、从最小队列中, 抽取最小结点
{ //摘取优先队列中最小元素的内容, 这里返回图中顶点的标号(0 ~ G.vexnum-1),
//这些标号是保存在 Q[1..n]中的
if(Q[0]<1)
{
  cout<<"heap underflow!"<<endl;
  return -10000;
}
int min = Q[1];
Q[1] = Q[Q[0]];
Q[0] = Q[0] - 1;
minHeapify(Q,d,1);
return min;
}

```

ALGraph 图的建立

先定义几个宏,

```

#define MAX_VERTEX_NUM 20 //图中最大的节点数目
#define INFINITY 10000
#define NIL -1

```

再建立几个数据结构:

```

typedef struct ArcNode //弧节点, 就是邻接链表的表节点
{
  int adjvex; //该弧所指向尾节点的位置, 其实保存的就是数组的下标
  ArcNode *nextarc; //指向第一条弧的指针
  int weight; //权重。
}ArcNode;

```

```
typedef struct VNode
{
    ArcNode* firstarc;
}VNode,AdjList[MAX_VERTEX_NUM];
```

```
typedef struct
{
    AdjList vertices;
    int vexnum,arcnum;
}ALGraph;
```

编写几个功能函数：

```
void initALGraph(ALGraph* GPt,int vn) //初始化结点
{
    GPt->arcnum = 0;
    GPt->vexnum = vn;
    for(int i=0;i<vn;i++)
    {
        GPt->vertices[i].firstarc = NULL;
    }
}

void insertArc(ALGraph* GPt,int vhead,int vtail,int w) //增加结点操作
{
    ArcNode* arcNodePt = new ArcNode;
    arcNodePt->nextarc = NULL;
    arcNodePt->adjvex = vtail;
    arcNodePt->weight = w;

    ArcNode* tailPt = GPt->vertices[vhead].firstarc;
    if(tailPt==NULL)
    {
        GPt->vertices[vhead].firstarc = arcNodePt;
    }
    else
    {
        while(tailPt->nextarc!=NULL)
        {
            tailPt = tailPt->nextarc;
        }
        tailPt->nextarc = arcNodePt;
    }
}
```

```

GPt->arcnum++;
}

void displayGraph(ALGraph G) //打印结点
{
    ArcNode* arcNodePt;
    for(int i=0;i<G.vexnum;i++)
    {
        arcNodePt = G.vertices[i].firstarc;
        cout<<"vertex"<<i<<": ";
        while(arcNodePt!=NULL)
        {
            cout<<arcNodePt->adjvex<<"("<<"weight"<<arcNodePt->weight<<")"<<
            " ";
            arcNodePt = arcNodePt->nextarc;
        }
        cout<<endl;
    }
}

int getEdgeWeight(ALGraph G,int vhead,int vtail) //求边的权重
{
    ArcNode* arcNodePt = G.vertices[vhead].firstarc;
    while(arcNodePt!=NULL)
    {
        if(arcNodePt->adjvex==vtail)
        {
            return arcNodePt->weight;
        }
        arcNodePt = arcNodePt->nextarc;
    }
    return INFINITY;
}

```

主函数测试用例

最后，便是编写主函数测试本程序：

```

int main(){

    ALGraph G;
    ALGraph* GPt = &G;
    initALGraph(GPt,5);
}

```

```

insertArc(GPt,0,1,10);
insertArc(GPt,0,3,5);
insertArc(GPt,1,2,1);
insertArc(GPt,1,3,2);
insertArc(GPt,2,4,4);
insertArc(GPt,3,1,3);
insertArc(GPt,3,2,9);
insertArc(GPt,3,4,2);
insertArc(GPt,4,2,4);
insertArc(GPt,4,0,7);

cout<<"显示出此构造的图:"<<endl;
displayGraph(G);
cout<<endl;

int d[MAX_VERTEX_NUM];
int pi[MAX_VERTEX_NUM];
int Q[MAX_VERTEX_NUM+1];
//Q[]的第一个元素只保存堆的大小，不保存元素。所以定义长度时+1

dijkstra(G,0,d,pi,Q);

for(int i=0;i<G.vexnum;i++)
{
    cout<<"从源点 0 到点"<<i<<"的最短路径信息:"<<endl;
    cout<<"长度为"<<d[i]<<endl;
    cout<<"路径为";
    printRoute(i,pi);
    cout<<endl;
    if(i==G.vexnum-1)
    {
        cout<<endl;
    }
}
return 0;
}

```

最后的运行结果，如下所示：

```
显示于此构造的图：  
vertex0: 1<weight10> 3<weight5>  
vertex1: 2<weight1> 3<weight2>  
vertex2: 4<weight4>  
vertex3: 1<weight3> 2<weight9> 4<weight2>  
vertex4: 2<weight4> 0<weight7>  
  
从源点0到点0的最短路径信息：  
长度为0  
路径为0  
从源点0到点1的最短路径信息：  
长度为8  
路径为0 3 1  
从源点0到点2的最短路径信息：  
长度为9  
路径为0 3 1 2  
从源点0到点3的最短路径信息：  
长度为5  
路径为0 3  
从源点0到点4的最短路径信息：  
长度为7  
路径为0 3 4  
  
Press any key to continue...  
搜狗拼音 半：
```

全文，到此完。

版权所有。转载本 **BLOG** 内任何文章，请以超链接形式注明出处。

否则，一经发现，必定永久谴责+追究法律责任。谢谢，各位。

三、动态规划算法

作者 July 二零一零年十二月三十一日

本文参考：微软面试 100 题系列 V0.1 版第 19、56 题、算法导论、维基百科。

第一部分、什么是动态规划算法

ok，咱们先来了解下什么是动态规划算法。

动态规划一般也只能应用于有最优子结构的问题。最优子结构的意思是局部最优解能决定全局最优解(对有些问题这个要求并不能完全满足，故有时需要引入一定的近似)。简单地说，问题能够分解成子问题来解决。

动态规划算法分以下 4 个步骤：

1. 描述最优解的结构
2. 递归定义最优解的值
3. 按自底向上的方式计算最优解的值 //此 3 步构成动态规划解的基础。
4. 由计算出的结果构造一个最优解。 //此步如果只要求计算最优解的值时，可省略。

好，接下来，咱们讨论适合采用动态规划方法的最优化问题的两个要素：最优子结构性质，和子问题重叠性质。

- 最优子结构

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）。意思就是，总问题包含很多个子问题，而这些子问题的解也是最优的。

- 重叠子问题

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。

第二部分、动态规划算法解 LCS 问题

下面，咱们运用此动态规划算法解此 LCS 问题。有一点必须声明的是，LCS 问题即最长公共子序列问题，它不要求所求得的字符在所给的字符串中是连续的（例如：输入两个字符串 BDCABA 和 ABCBDAB，字符串 BCBA 和 BDAB 都是它们的最长公共子序列，则输出它们的长度 4，并打印任意一个子序列）。

ok，咱们马上进入面试题第 56 题的求解，即运用经典的动态规划算法：

2.0、LCS 问题描述

56.最长公共子序列。

题目：如果字符串一的所有字符按其在字符串中的顺序出现在另外一个字符串二中，则字符串一称之为字符串二的子串。

注意，并不要求子串（字符串一）的字符必须连续出现在字符串二中。

请编写一个函数，输入两个字符串，求它们的最长公共子串，并打印出最长公共子串。

例如：输入两个字符串 **BDCABA** 和 **ABCBDA**，字符串 **BCBA** 和 **BDAB** 都是它们的最长公共子序列，则输出它们的长度 **4**，并打印任意一个子序列。

分析：求最长公共子序列（Longest Common Subsequence, LCS）是一道非常经典的动态规划题，因此一些重视算法的公司像 MicroStrategy 都把它当作面试题。

事实上，最长公共子序列问题也有最优子结构性质。

记：

$X_i = \langle x_1, \dots, x_i \rangle$ 即 X 序列的前 i 个字符 ($1 \leq i \leq m$) (前缀)

$Y_j = \langle y_1, \dots, y_j \rangle$ 即 Y 序列的前 j 个字符 ($1 \leq j \leq n$) (前缀)

假定 $Z = \langle z_1, \dots, z_k \rangle \in \text{LCS}(X, Y)$ 。

- 若 $x_m = y_n$ (最后一个字符相同)，则不难用反证法证明：该字符必是 X 与 Y 的任一最长公共子序列 Z (设长度为 k) 的最后一个字符，即有 $z_k = x_m = y_n$ 且显然有 $Z_{k-1} \in \text{LCS}(X_{m-1}, Y_{n-1})$ 即 Z 的前缀 **Zk-1 是 Xm-1 与 Yn-1 的最长公共子序列**。此时，问题化归成求 X_{m-1} 与 Y_{n-1} 的 $\text{LCS}(X, Y)$ 的长度等于 $\text{LCS}(X_{m-1}, Y_{n-1})$ 的长度加 1)。
- 若 $x_m \neq y_n$ ，则亦不难用反证法证明：要么 $Z \in \text{LCS}(X_{m-1}, Y)$ ，要么 $Z \in \text{LCS}(X, Y_{n-1})$ 。由于 $z_k \neq x_m$ 与 $z_k \neq y_n$ 其中至少有一个必成立，若 $z_k \neq x_m$ 则有 $Z \in \text{LCS}(X_{m-1}, Y)$ ，类似的，若 $z_k \neq y_n$ 则有 $Z \in \text{LCS}(X, Y_{n-1})$ 。此时，问题化归成求 X_{m-1} 与 Y 的 LCS 及 X 与 Y_{n-1} 的 LCS 。 $\text{LCS}(X, Y)$ 的长度为： $\max\{\text{LCS}(X_{m-1}, Y)\text{的长度}, \text{LCS}(X, Y_{n-1})\text{的长度}\}$ 。

由于上述当 $x_m \neq y_n$ 的情况下，求 $\text{LCS}(X_{m-1}, Y)$ 的长度与 $\text{LCS}(X, Y_{n-1})$ 的长度，这两个问题不是相互独立的：两者都要求 $\text{LCS}(X_{m-1}, Y_{n-1})$ 的长度。另外两个序列的 LCS 中包含了两个序列的前缀的 LCS ，故问题具有最优子结构性质考虑用动态规划法。

也就是说，解决这个LCS问题，你要求三个方面的东西：1、 $LCS(X_{m-1}, Y_{n-1}) + 1$ ；
2、 $LCS(X_{m-1}, Y)$ ， $LCS(X, Y_{n-1})$ ； 3、 $\max\{LCS(X_{m-1}, Y), LCS(X, Y_{n-1})\}$ 。

2.1、最长公共子序列的结构

最长公共子序列的结构有如下表示：

设序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 的一个最长公共子序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ ，
 则：

1. 若 $x_m = y_n$ ，则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列；
2. 若 $x_m \neq y_n$ 且 $z_k \neq x_m$ ，则 Z 是 X_{m-1} 和 Y 的最长公共子序列；
3. 若 $x_m \neq y_n$ 且 $z_k \neq y_n$ ，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

其中 $X_{m-1} = \langle x_1, x_2, \dots, x_{m-1} \rangle$, $Y_{n-1} = \langle y_1, y_2, \dots, y_{n-1} \rangle$, $Z_{k-1} = \langle z_1, z_2, \dots, z_{k-1} \rangle$ 。

2.2、子问题的递归结构

由最长公共子序列问题的最优子结构性质可知，要找出 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 的最长公共子序列，可按以下方式递归地进行：当 $x_m = y_n$ 时，找出 X_{m-1} 和 Y_{n-1} 的最长公共子序列，然后在其尾部加上 $x_m (=y_n)$ 即可得 X 和 Y 的一个最长公共子序列。当 $x_m \neq y_n$ 时，必须解两个子问题，即找出 X_{m-1} 和 Y 的一个最长公共子序列及 X 和 Y_{n-1} 的一个最长公共子序列。这两个公共子序列中较长者即为 X 和 Y 的一个最长公共子序列。

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如，在计算 X 和 Y 的最长公共子序列时，可能要计算出 X 和 Y_{n-1} 及 X_{m-1} 和 Y 的最长公共子序列。而这两个子问题都包含一个公共子问题，即计算 X_{m-1} 和 Y_{n-1} 的最长公共子序列。

与矩阵连乘积最优计算次序问题类似，我们来建立子问题的最优值的递归关系。用 $c[i, j]$ 记录序列 X_i 和 Y_j 的最长公共子序列的长度。其中 $X_i = \langle x_1, x_2, \dots, x_i \rangle$, $Y_j = \langle y_1, y_2, \dots, y_j \rangle$ 。当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列，故 $c[i, j] = 0$ 。其他情况下，由定理可建立递归关系如下：

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

2.3、计算最优值

直接利用上节节末的递归式，我们将很容易就能写出一个计算 $c[i,j]$ 的递归算法，但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中，总共只有 $\Theta(m^*n)$ 个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法 $LCS_LENGTH(X,Y)$ 以序列 $X=<x_1, x_2, \dots, x_m>$ 和 $Y=<y_1, y_2, \dots, y_n>$ 作为输入。输出两个数组 $c[0..m, 0..n]$ 和 $b[1..m, 1..n]$ 。其中 $c[i,j]$ 存储 X_i 与 Y_j 的最长公共子序列的长度， $b[i,j]$ 记录指示 $c[i,j]$ 的值是由哪一个子问题的解达到的，这在构造最长公共子序列时要用到。最后， X 和 Y 的最长公共子序列的长度记录于 $c[m,n]$ 中。

```
1.      Procedure LCS_LENGTH(X,Y);
2.      begin
3.          m:=length[X];
4.          n:=length[Y];
5.          for i:=1 to m do c[i,0]:=0;
6.          for j:=1 to n do c[0,j]:=0;
7.          for i:=1 to m do
8.              for j:=1 to n do
9.                  if x[i]=y[j] then
10.                      begin
11.                          c[i,j]:=c[i-1,j-1]+1;
12.                          b[i,j]:="\uparrow";
18.                      end
19.                  else
20.                      begin
21.                          c[i,j]:=c[i,j-1];
22.                          b[i,j]:="<span style='color:green; font-family:monospace">\leftarrow";
23.                      end;
24.          return(c,b);
25.      end;
```

由算法 LCS_LENGTH 计算得到的数组 b 可用于快速构造序列 $X=<x_1, x_2, \dots, x_m>$ 和 $Y=<y_1, y_2, \dots, y_n>$ 的最长公共子序列。首先从 $b[m,n]$ 开始，沿着其中的箭头所指的方向在数组 b 中搜索。

- 当 $b[i,j]$ 中遇到 " \nwarrow " 时 (意味着 $x_i=y_j$ 是 LCS 的一个元素), 表示 X_i 与 Y_j 的最长公共子序列是由 X_{i-1} 与 Y_{j-1} 的最长公共子序列在尾部加上 x_i 得到的子序列;
- 当 $b[i,j]$ 中遇到 " \uparrow " 时, 表示 X_i 与 Y_j 的最长公共子序列和 X_{i-1} 与 Y_j 的最长公共子序列相同;
- 当 $b[i,j]$ 中遇到 " \leftarrow " 时, 表示 X_i 与 Y_j 的最长公共子序列和 X_i 与 Y_{j-1} 的最长公共子序列相同。

这种方法是按照反序来找 LCS 的每一个元素的。由于每个数组单元的计算耗费 $O(1)$ 时间, 算法 LCS_LENGTH 耗时 $O(mn)$ 。

2.4、构造最长公共子序列

下面的算法 $LCS(b, X, i, j)$ 实现根据 b 的内容打印出 X_i 与 Y_j 的最长公共子序列。通过算法的调用 $LCS(b, X, \text{length}[X], \text{length}[Y])$, 便可打印出序列 X 和 Y 的最长公共子序列。

```

1.      Procedure LCS(b,X,i,j);
2.      begin
3.          if i=0 or j=0 then return;
4.          if b[i,j] = " $\nwarrow$ " then
5.              begin
6.                  LCS(b,X,i-1,j-1);
7.                  print(x[i]); {打印 x[i]}
8.              end
9.          else if b[i,j] = " $\uparrow$ " then LCS(b,X,i-1,j)
10.                 else LCS(b,X,i,j-1);
11.      end;

```

在算法 LCS 中, 每一次的递归调用使 i 或 j 减 1, 因此算法的计算时间为 $O(m+n)$ 。

例如, 设所给的两个序列为 $X=<A, B, C, B, D, A, B>$ 和 $Y=<B, D, C, A, B, A>$ 。
由算法 LCS_LENGTH 和 LCS 计算出的结果如下图所示:

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

我来说明下此图（参考算法导论）。在序列 $X=\{A, B, C, B, D, A, B\}$ 和 $Y=\{B, D, C, A, B, A\}$ 上，由 LCS_LENGTH 计算出的表 c 和 b 。第 i 行和第 j 列中的方块包含了 $c[i, j]$ 的值以及指向 $b[i, j]$ 的箭头。在 $c[7,6]$ 的项 4，表的右下角为 X 和 Y 的一个 $LCS< B, C, B, A >$ 的长度。对于 $i, j > 0$ ，项 $c[i, j]$ 仅依赖于是否有 $x_i = y_j$ ，及项 $c[i-1, j]$ 和 $c[i, j-1]$ 的值，这几个项都在 $c[i, j]$ 之前计算。为了重构一个 LCS 的元素，从右下角开始跟踪 $b[i, j]$ 的箭头即可，这条路径标示为阴影，这条路径上的每一个“↖”对应于一个使 $x_i = y_j$ 为一个 LCS 的成员的项（高亮标示）。

所以根据上述图所示的结果，程序将最终输出：“B C B A”，或“B D A B”。

可能还是有读者对上面的图看的不是很清楚，下面，我再通过对最大子序列，最长公共子串与最长公共子序列的比较来阐述相关问题@Orisun:

- **最大子序列：**最大子序列是要找出由数组成的一维数组中和最大的连续子序列。比如 $\{5,-3,4,2\}$ 的最大子序列就是 $\{5,-3,4,2\}$ ，它的和是 8, 达到最大；而 $\{5,-6,4,2\}$ 的最大子序列是 $\{4,2\}$ ，它的和是 6。你已经看出来了，找最大子序列的方法很简单，只要前 i 项的和还没有小于 0 那么子序列就一直向后扩展，否则丢弃之前的子序列开始新的子序列，同时我们要记下各个子序列的和，最后找到和最大的子序列。更多请参看：程序员编程艺术第七章、求连续子数组的最大和。

- **最长公共子串:** 找两个字符串的最长公共子串，这个子串要求在原字符串中是连续的。其实这又是一个序贯决策问题，可以用动态规划来求解。我们采用一个二维矩阵来记录中间的结果。这个二维矩阵怎么构造呢？直接举个例子吧：“bab”和“caba”（当然我们现在一眼就可以看出来最长公共子串是“ba”或“ab”）

	b	a	b
c	0	0	0
a	0	1	0
b	1	0	1
a	0	1	0

我们看矩阵的斜对角线最长的那个就能找出最长公共子串。

不过在二维矩阵上找最长的由 1 组成的斜对角线也是件麻烦费时的事，下面改进：
当要在矩阵是填 1 时让它等于其左上角元素加 1。

	b	a	b
c	0	0	0
a	0	1	0
b	1	0	2
a	0	2	0

这样矩阵中的最大元素就是最长公共子串的长度。

在构造这个二维矩阵的过程中由于得出矩阵的某一行后其上一行就没用了，所以实际上在程序中可以用一维数组来代替这个矩阵。

- **最长公共子序列 LCS 问题:** 最长公共子序列与最长公共子串的区别在于最长公共子序列不要求在原字符串中是连续的，比如 ADE 和 ABCDE 的最长公共子序列是 ADE。

我们用动态规划的方法来思考这个问题如是求解。首先要找到状态转移方程：

等号约定， C_1 是 S_1 的最右侧字符， C_2 是 S_2 的最右侧字符， S_1' 是从 S_1 中去除 C_1 的部分， S_2' 是从 S_2 中去除 C_2 的部分。

$LCS(S_1, S_2)$ 等于：

- (1) $LCS(S_1, S_2')$
- (2) $LCS(S_1', S_2)$
- (3) 如果 C_1 不等于 C_2 : $LCS(S_1', S_2')$ ；如果 C_1 等于 C_2 : $LCS(S_1', S_2') + C_1$ ；

边界终止条件：如果 S_1 和 S_2 都是空串，则结果也是空串。

下面我们同样要构建一个矩阵来存储动态规划过程中子问题的解。这个矩阵中的每个数字代表了该行和该列之前的 LCS 的长度。与上面刚刚分析出的状态转移议程相对应，矩阵中每个格子里的数字应该这么填，它等于以下 3 项的最大值：

- (1) 上面一个格子里的数字
- (2) 左边一个格子里的数字
- (3) 左上角那个格子里的数字（如果 C_1 不等于 C_2 ）；左上角那个格子里的数字+1（如果 C_1 等于 C_2 ）

举个例子：

	G	C	T	A	
	0	0	0	0	0
G	0	1	1	1	1
B	0	1	1	1	1
T	0	1	1	2	2
A	0	1	1	2	3

填写最后一个数字时，它应该是下面三个的最大者：

- (1) 上边的数字 2
- (2) 左边的数字 2
- (3) 左上角的数字 $2+1=3$, 因为此时 $C1==C2$

所以最终结果是 3。

在填写过程中我们还是记录下当前单元格的数字来自于哪个单元格，以方便最后我们回溯找出最长公共子串。有时候左上、左、上三者中有多个同时达到最大，那么任取其中之一，但是在整个过程中你必须遵循固定的优先标准。在我的代码中优先级别是左上>左>上。

下图给出了回溯法找出 LCS 的过程：

		G	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2
G	0	1	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4
A	0	1	2	3	3	3	4	4	4
T	0	1	2	3	3	4	4	4	4
G	0	1	2	3	3	4	4	5	5

2.5、算法的改进

对于一个具体问题，按照一般的算法设计策略设计出的算法，往往在算法的时间和空间需求上还可以改进。这种改进，通常是利用具体问题的一些特殊性。

例如，在算法 `LCS_LENGTH` 和 `LCS` 中，可进一步将数组 `b` 省去。事实上，数组元素 `c[i,j]` 的值仅由 `c[i-1,j-1]`, `c[i-1,j]` 和 `c[i,j-1]` 三个值之一确定，而数组元素 `b[i,j]` 也只是用来指示 `c[i,j]` 究竟由哪个值确定。因此，在算法 `LCS` 中，我们可以不借助于数组 `b` 而借助于数组 `c` 本身临时判断 `c[i,j]` 的值是由 `c[i-1,j-1]`, `c[i-1,j]` 和 `c[i,j-1]` 中哪一个数值元素所确定，代价是 $O(1)$

时间。既然 b 对于算法 LCS 不是必要的，那么算法 LCS_LENGTH 便不必保存它。这一来，可节省 $\theta(mn)$ 的空间，而 LCS_LENGTH 和 LCS 所需要的时间分别仍然是 $O(mn)$ 和 $O(m+n)$ 。不过，由于数组 c 仍需要 $O(mn)$ 的空间，因此这里所作的改进，只是在空间复杂性的常数因子上的改进。

另外，如果只需要计算最长公共子序列的长度，则算法的空间需求还可大大减少。事实上，在计算 $c[i,j]$ 时，只用到数组 c 的第 i 行和第 $i-1$ 行。因此，只要用 2 行的数组空间就可以计算出最长公共子序列的长度。更进一步的分析还可将空间需求减至 $\min(m, n)$ 。

第三部分、最长公共子序列问题代码

ok，最后给出此面试第 56 题的代码，参考代码如下，请君自看：

```
1.      // LCS.cpp : 定义控制台应用程序的入口点。
2.      //
3.
4.      //copyright@zhedahht
5.      //updated@2011.12.13 July
6.      #include "stdafx.h"
7.      #include "string.h"
8.      #include <iostream>
9.      using namespace std;
10.
11.     // directions of LCS generation
12.     enum decreaseDir {kInit = 0, kLeft, kUp, kLeftUp};
13.
14.     void LCS_Print(int **LCS_direction,
15.                  char* pStr1, char* pStr2,
16.                  size_t row, size_t col);
17.
18.     // Get the length of two strings' LCSs, and print one of the LCSs
19.     // Input: pStr1          - the first string
20.     //         pStr2          - the second string
21.     // Output: the length of two strings' LCSs
22.     int LCS(char* pStr1, char* pStr2)
23.     {
24.         if(!pStr1 || !pStr2)
25.             return 0;
26.
27.         size_t length1 = strlen(pStr1);
28.         size_t length2 = strlen(pStr2);
29.         if(!length1 || !length2)
```

```

30.         return 0;
31.
32.         size_t i, j;
33.
34.         // initiate the length matrix
35.         int **LCS_length;
36.         LCS_length = (int**)(new int[length1]);
37.         for(i = 0; i < length1; ++ i)
38.             LCS_length[i] = (int*)new int[length2];
39.
40.         for(i = 0; i < length1; ++ i)
41.             for(j = 0; j < length2; ++ j)
42.                 LCS_length[i][j] = 0;
43.
44.         // initiate the direction matrix
45.         int **LCS_direction;
46.         LCS_direction = (int**)(new int[length1]);
47.         for( i = 0; i < length1; ++ i)
48.             LCS_direction[i] = (int*)new int[length2];
49.
50.         for(i = 0; i < length1; ++ i)
51.             for(j = 0; j < length2; ++ j)
52.                 LCS_direction[i][j] = kInit;
53.
54.         for(i = 0; i < length1; ++ i)
55.         {
56.             for(j = 0; j < length2; ++ j)
57.             {
58.                 //之前此处的代码有问题，现在订正如下:
59.                 if(i == 0 || j == 0)
60.                 {
61.                     if(pStr1[i] == pStr2[j])
62.                     {
63.                         LCS_length[i][j] = 1;
64.                         LCS_direction[i][j] = kLeftUp;
65.                     }
66.                 else
67.                 {
68.                     if(i > 0)
69.                     {
70.                         LCS_length[i][j] = LCS_length[i - 1][j];
71.                         LCS_direction[i][j] = kUp;
72.                     }
73.                 if(j > 0)

```

```

74.             {
75.                 LCS_length[i][j] = LCS_length[i][j - 1];
76.                 LCS_direction[i][j] = kLeft;
77.             }
78.         }
79.     }
80.     // a char of LCS is found,
81.     // it comes from the left up entry in the direction matrix
82.     else if(pStr1[i] == pStr2[j])
83.     {
84.         LCS_length[i][j] = LCS_length[i - 1][j - 1] + 1;
85.         LCS_direction[i][j] = kLeftUp;
86.     }
87.     // it comes from the up entry in the direction matrix
88.     else if(LCS_length[i - 1][j] > LCS_length[i][j - 1])
89.     {
90.         LCS_length[i][j] = LCS_length[i - 1][j];
91.         LCS_direction[i][j] = kUp;
92.     }
93.     // it comes from the left entry in the direction matrix
94.     else
95.     {
96.         LCS_length[i][j] = LCS_length[i][j - 1];
97.         LCS_direction[i][j] = kLeft;
98.     }
99. }
100. }
101. LCS_Print(LCS_direction, pStr1, pStr2, length1 - 1, length2 - 1); //调用
    下面的 LCS_Print 打印出所求子串。
102. return LCS_length[length1 - 1][length2 - 1]; //返回
    长度。
103. }
104.
105. // Print a LCS for two strings
106. // Input: LCS_direction - a 2d matrix which records the direction of
107. //          LCS generation
108. //          pStr1      - the first string
109. //          pStr2      - the second string
110. //          row        - the row index in the matrix LCS_direction
111. //          col        - the column index in the matrix LCS_direction
112. void LCS_Print(int **LCS_direction,
113.                 char* pStr1, char* pStr2,
114.                 size_t row, size_t col)
115. {

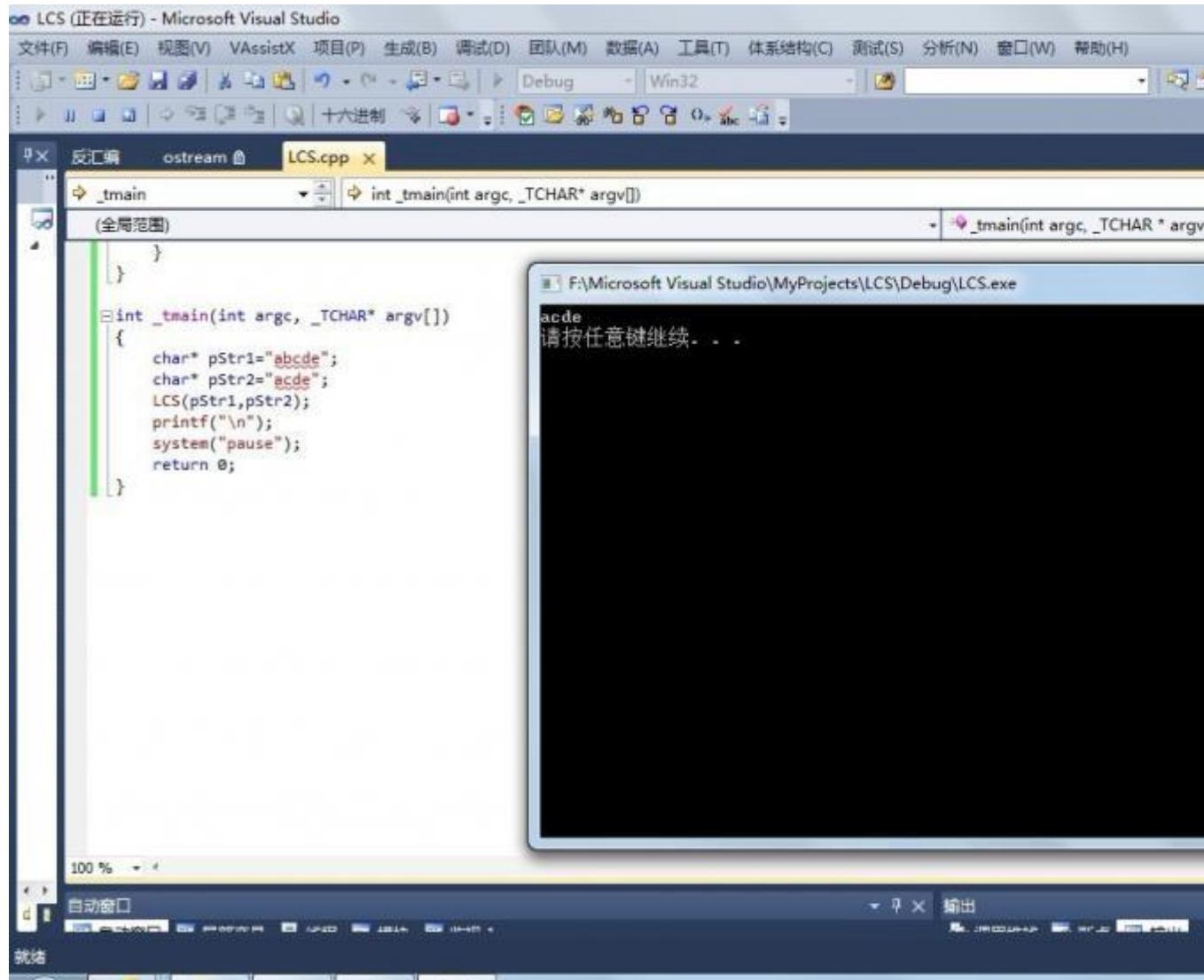
```

```

116.         if(pStr1 == NULL || pStr2 == NULL)
117.             return;
118.
119.         size_t length1 = strlen(pStr1);
120.         size_t length2 = strlen(pStr2);
121.
122.         if(length1 == 0 || length2 == 0 || !(row < length1 && col < length2))
123.             return;
124.
125.         // kLeftUp implies a char in the LCS is found
126.         if(LCS_direction[row][col] == kLeftUp)
127.         {
128.             if(row > 0 && col > 0)
129.                 LCS_Print(LCS_direction, pStr1, pStr2, row - 1, col - 1);
130.
131.             // print the char
132.             printf("%c", pStr1[row]);
133.         }
134.         else if(LCS_direction[row][col] == kLeft)
135.         {
136.             // move to the left entry in the direction matrix
137.             if(col > 0)
138.                 LCS_Print(LCS_direction, pStr1, pStr2, row, col - 1);
139.         }
140.         else if(LCS_direction[row][col] == kUp)
141.         {
142.             // move to the up entry in the direction matrix
143.             if(row > 0)
144.                 LCS_Print(LCS_direction, pStr1, pStr2, row - 1, col);
145.         }
146.     }
147.
148.     int _tmain(int argc, _TCHAR* argv[])
149.     {
150.         char* pStr1="abcde";
151.         char* pStr2="acde";
152.         LCS(pStr1,pStr2);
153.         printf("\n");
154.         system("pause");
155.         return 0;
156.     }

```

程序运行结果如下所示：



扩展：如果题目改成求两个字符串的最长公共子字符串，应该怎么求？子字符串的定义和子串的定义类似，但要求是连续分布在其他字符串中。

比如输入两个字符串 BDCABA 和 ABCBDAB 的最长公共字符串有 BD 和 AB，它们的长度都是 2。

第四部分、LCS 问题的时间复杂度

算法导论上指出，

1. 最长公共子序列问题的一个一般的算法、时间复杂度为 $O(mn)$ 。然后，Masek 和 Paterson 给出了一个 $O(mn/\lg n)$ 时间内执行的算法，其中 $n \leq m$ ，而且此序列是从

一个有限集合中而来。在输入序列中没有出现超过一次的特殊情况下，**Szymanski** 说明这个问题可在 $O(n+m) \lg(n+m)$ 内解决。

2. 一篇由 **Gilbert** 和 **Moore** 撰写的关于可变长度二元编码的早期论文中有这样的应用：在所有的概率 p_i 都是 0 的情况下构造最优二叉查找树，这篇论文给出一个 $O(n^3)$ 时间的算法。**Hu** 和 **Tucker** 设计了一个算法，它在所有的概率 p_i 都是 0 的情况下，使用 $O(n)$ 的时间和 $O(n)$ 的空间，最后，**Knuth** 把时间降到了 $O(n \lg n)$ 。

关于此动态规划算法更多可参考 算法导论一书第 15 章 动态规划问题，至于关于此面试第 56 题的更多，可参考我即将整理上传的答案 V04 版第 41-60 题的答案。

补充：一网友提供的关于此最长公共子序列问题的 java 算法源码，我自行测试了下，正确：

```
import java.util.Random;

public class LCS{
    public static void main(String[] args){

        //设置字符串长度
        int substringLength1 = 20;
        int substringLength2 = 20; //具体大小可自行设置

        // 随机生成字符串
        String x = GetRandomStrings(substringLength1);
        String y = GetRandomStrings(substringLength2);

        Long startTime = System.nanoTime();
        // 构造二维数组记录子问题 x[i] 和 y[i] 的 LCS 的长度
        int[][] opt = new int[substringLength1 + 1][substringLength2 + 1];

        // 动态规划计算所有子问题
        for (int i = substringLength1 - 1; i >= 0; i--){
            for (int j = substringLength2 - 1; j >= 0; j--){
                if (x.charAt(i) == y.charAt(j))
                    opt[i][j] = opt[i + 1][j + 1] + 1; //参考上文我给的公式。
                else
                    opt[i][j] = Math.max(opt[i + 1][j], opt[i][j + 1]); //参考上文我给的公式。
            }
        }
    }
}
```

理解上段，参考上文我给的公式：

根据上述结论，可得到以下公式，

如果我们记字符串 X_i 和 Y_j 的 LCS 的长度为 $c[i,j]$ ，我们可以递归地求 $c[i,j]$ ：

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

```
System.out.println("substring1:"+x);
System.out.println("substring2:"+y);
System.out.print("LCS:");

int i = 0, j = 0;
while (i < substringLength1 && j < substringLength2){
    if (x.charAt(i) == y.charAt(j)){
        System.out.print(x.charAt(i));
        i++;
        j++;
    } else if (opt[i + 1][j] >= opt[i][j + 1])
        i++;
    else
        j++;
}
Long endTime = System.nanoTime();
System.out.println(" Totle time is " + (endTime - startTime) + " ns");
}

//取得定长随机字符串
public static String GetRandomStrings(int length){
    StringBuffer buffer = new StringBuffer("abcdefghijklmnopqrstuvwxyz");
    StringBuffer sb = new StringBuffer();
    Random r = new Random();
```

```

int range = buffer.length();
for (int i = 0; i < length; i++){
    sb.append(buffer.charAt(r.nextInt(range)));
}
return sb.toString();
}
}

```

eclipse 运行结果为:

```

substring1:akqrshrengxqiyxuloqk
substring2:tdzbujtlqhecaqgwfzbc
LCS:qheq Totle time is 818058 ns

```

OK, 更多, 请参考: 程序员编程艺术第十一章、最长公共子序列 (LCS) 问题。完。

四、BFS 和 DFS 优先搜索算法

作者: July 二零一一年一月一日

本人参考: 算法导论

本人声明: 个人原创, 转载请注明出处。

ok, 开始。

翻遍网上, 关于此类 BFS 和 DFS 算法的文章, 很多。但, 都说不出个所以然来。

读完此文, 我想,

你对图的广度优先搜索和深度优先搜索定会有个通通透透, 彻彻底底的认识。

咱们由 **BFS** 开始:

首先, 看下算法导论一书关于 此 **BFS** 广度优先搜索算法的概述。

算法导论第二版, 中译本, 第 324 页。

广度优先搜索 (BFS)

在 **Prime** 最小生成树算法, 和 **Dijkstra** 单源最短路径算法中, 都采用了与 **BFS** 算法类似的思想。

// u 为 v 的先辈或父母。

BFS(G, s)

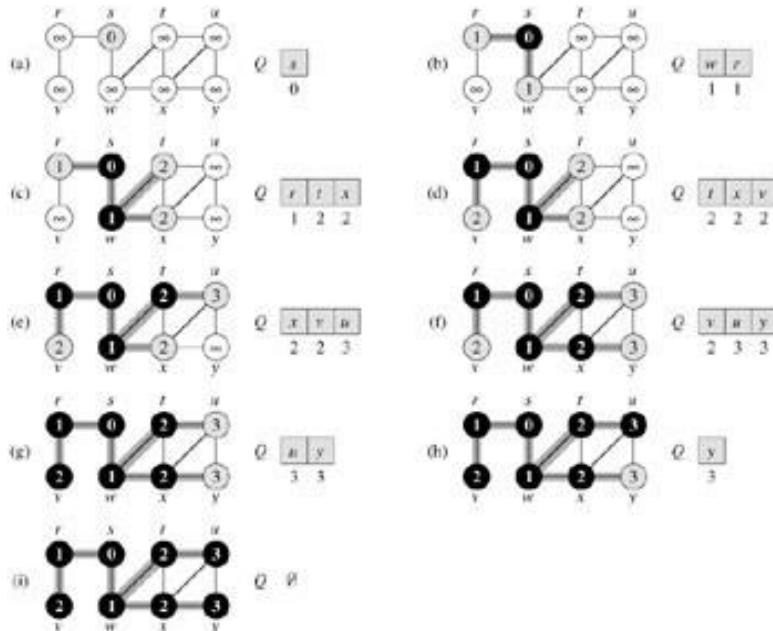
1 for each vertex $u \in V[G] - \{s\}$

```

2   do color[u] ← WHITE
3   d[u] ← ∞
4   n[u] ← NIL
//除了源顶点 s 之外，第 1-4 行置每个顶点为白色，置每个顶点 u 的 d[u] 为无穷大，  

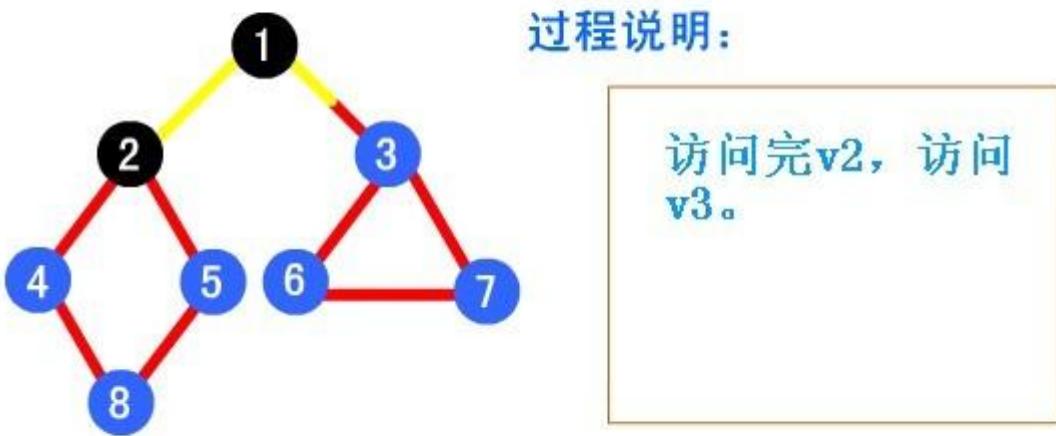
//置每个顶点的父母为 NIL。
5 color[s] ← GRAY
//第 5 行，将源顶点 s 置为灰色，这是因为在过程开始时，源顶点已被发现。
6 d[s] ← 0 //将 d[s] 初始化为 0。
7 n[s] ← NIL //将源顶点的父顶点置为 NIL。
8 Q ← ∅
9 ENQUEUE(Q, s) //入队
//第 8、9 行，初始化队列 Q，使其仅含源顶点 s。
10 while Q ≠ ∅
11   do u ← DEQUEUE(Q) //出队
    //第 11 行，确定队列头部 Q 头部的灰色顶点 u，并将其从 Q 中去掉。
12   for each v ∈ Adj[u] //for 循环考察 u 的邻接表中的每个顶点 v
13     do if color[v] = WHITE
        then color[v] ← GRAY //置为灰色
14     d[v] ← d[u] + 1 //距离被置为 d[u]+1
15     n[v] ← u //u 记为该顶点的父母
16     ENQUEUE(Q, v) //插入队列中
17   color[u] ← BLACK //u 置为黑色

```



由下图及链接的演示过程，清晰在目，也就不用多说了：

图的广度优先遍历



先遍历序列：v1、v2

广度优先遍历演示地址：

<http://sjjq.js.zwu.edu.cn/SFXX/sf1/qdyxbl.html>

ok，不再赘述。接下来，具体讲解深度优先搜索算法。

深度优先探索算法 DFS

//u 为 v 的先辈或父母。

DFS(G)

1 for each vertex u ∈ V [G]

2 do color[u] ← WHITE

3 n[u] ← NIL

//第 1-3 行，把所有顶点置为白色，所有 n 域被初始化为 NIL。

4 time ← 0 //复位时间计数器

5 for each vertex u ∈ V [G]

6 do if color[u] = WHITE

7 then DFS-VISIT(u) //调用 DFS-VISIT 访问 u，u 成为深度优先森林中一棵新的树

//第 5-7 行，依次检索 V 中的顶点，发现白色顶点时，调用 DFS-VISIT 访问该顶点。

//每个顶点 u 都对应于一个发现时刻 d[u]和一个完成时刻 f[u]。

DFS-VISIT(u)

1 color[u] ← GRAY //u 开始时被发现，置为白色

```

2 time ← time +1           //time 递增
3 d[u] <-time             //记录 u 被发现的时间
4 for each v ∈ Adj[u]    //检查并访问 u 的每一个邻接点 v
5   do if color[v] = WHITE      //如果 v 为白色，则递归访问 v。
6     then n[v] ← u            //置 u 为 v 的先辈
7       DFS-VISIT(v)          //递归深度，访问邻结点 v
8 color[u] <-BLACK         //u 置为黑色，表示 u 及其邻接点都已访问完成
9 f[u] > time ← time +1  //访问完成时间记录在 f[u] 中。
//完

```

第 1-3 行，5-7 行循环占用时间为 $O(V)$ ，此不包括调用 DFS-VISIT 的时间。

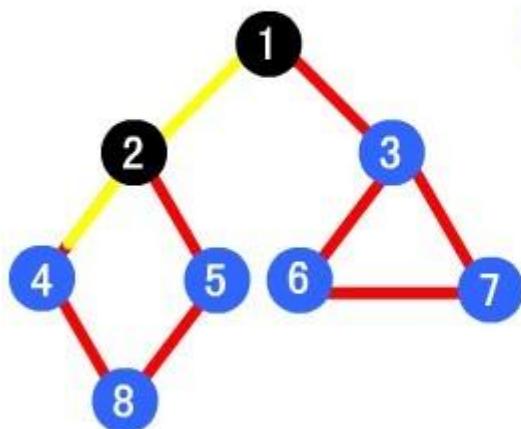
对于每个顶点 $v(-V)$ ，过程 DFS-VISIT 仅被调用依次，因为只有对白色顶点才会调用此过程。

第 4-7 行，执行时间为 $O(E)$ 。

因此，总的执行时间为 $O(V+E)$ 。

下面的链接，给出了深度优先搜索的演示系统：

图的深度优先遍历

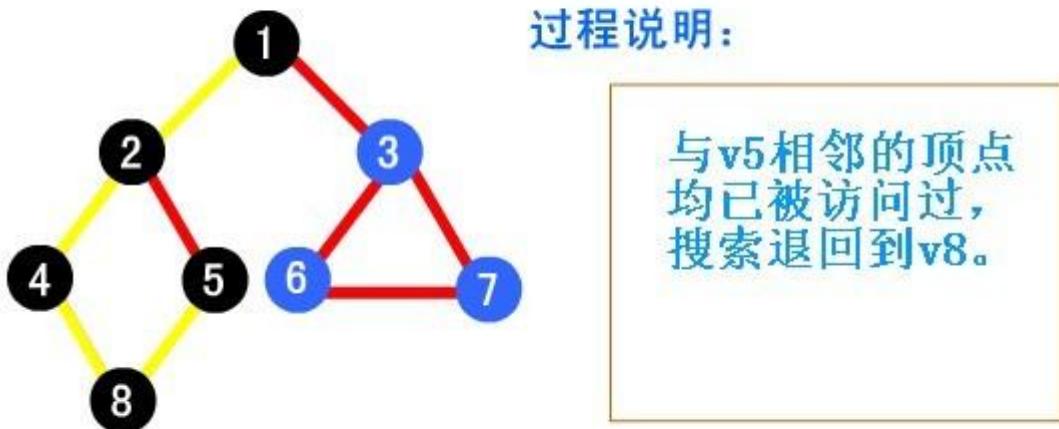


过程说明：

同 v_2 邻接的有
 v_1, v_4, v_5 ，
其中 v_4, v_5 未被
访问过，我们先
访问 v_4 。

先遍历序列： v_1, v_2

图的深度优先遍历



先遍历序列：v1、v2、v4、v8、v5

图的深度优先遍历演示系统：

<http://sjjq.js.zwu.edu.cn/SFXX/sf1/sdyxbl.html>

=====

最后，咱们再来看深度优先搜索的递归实现与非递归实现

1、**DFS** 递归实现：

```
void dftR(PGraphMatrix inGraph)
{
    PVexType v;
    assertF(inGraph!=NULL,"in dftR, pass in inGraph is null\n");
    printf("\n====start of dft recursive version====\n");
    for(v=firstVertex(inGraph);v!=NULL;v=nextVertex(inGraph,v))
        if(v->marked==0)
            dfsR(inGraph,v);
    printf("\n====end of dft recursive version====\n");
}

void dfsR(PGraphMatrix inGraph,PVexType inV)
{
    PVexType v1;
    assertF(inGraph!=NULL,"in dfsR,inGraph is null\n");
    // Your code here
}
```

```

assertF(inV!=NULL,"in dfsR,inV is null\n");
inV->marked=1;
visit(inV);
for(v1=firstAdjacent(inGraph,inV);v1!=NULL;v1=nextAdjacent(inGraph,inV,v1))
//v1 当为 v 的邻接点。
if(v1->marked==0)
dfsR(inGraph,v1);
}

```

2、DFS 非递归实现

非递归版本---借助结点类型为队列的栈实现

联系树的前序遍历的非递归实现：

可知，其中无非是分成“探左”和“访右”两大块访右需借助栈中弹出的结点进行。

在图的深度优先搜索中，同样可分成“深度探索”和“回访上层未访结点”两块：

1、图的深度探索这样一个过程和树的“探左”完全一致，

只要对已访问过的结点作一个判定即可。

2、而图的回访上层未访结点和树的前序遍历中的“访右”也是一致的。

但是，对于树而言，是提供 `rightSibling` 这样的操作的，因而访右相当好实现。

在这里，若要实现相应功能，考虑将每一个当前结点的下层结点中，如果有 m 个未访问结点，

则最左的一个需要访问，而将剩余的 $m-1$ 个结点按从左到右的顺序推入一个队列中。

并将这个队列压入一个堆栈中。

这样，当当前的结点的邻接点均已访问或无邻接点需要回访时，
 则从栈顶的队列结点中弹出队列元素，将队列中的结点元素依次出队，
 若已访问，则继续出队(当当前队列结点已空时，则继续出栈，弹出下一个栈顶的队列)，
 直至遇到有未访问结点(访问并置当前点为该点)或直到栈为空(则当前的深度优先搜索树停止搜索)。

将算法通过精简过的 C 源程序的方式描述如下：

```

//dfsUR:功能从一个树的某个结点 inV 发，以深度优先的原则访问所有与它相邻的结点
void dfsUR(PGraphMatrix inGraph,PVexType inV)
{
    PSingleRearSeqQueue tmpQ; //定义临时队列，用以接受栈顶队列及压栈时使用
    PSeqStack testStack; //存放当前层中的 m-1 个未访问结点构成队列的堆栈。
    //一些变量声明,初始化动作
    //访问当前结点
}

```

```

inV->marked=1; //当 marked 值为 1 时将不必再访问。
visit(inV);

do
{
    flag2=0;
    //flag2 是一个重要的标志变量, 用以、说明当前结点的所有未访问结点的个数,两个以上的用 2 代表
    //flag2:0:current node has no adjacent which has not been visited.
    //1:current node has only one adjacent node which has not been visited.
    //2:current node has more than one adjacent node which have not been
visited.

v1=firstAdjacent(inGraph,inV); //邻接点 v1
while(v1!=NULL) //访问当前结点的所有邻接点
{
    if(v1->marked==0) //..
    {
        if(flag2==0) //当前结点的邻接点有 0 个未访问
        {
            //首先, 访问最左结点
            visit(v1);
            v1->marked=1; //访问完成
            flag2=1; //
            //记录最左儿子
            lChildV=v1;
            //save the current node's first unvisited(has been visited at this
time)adjacent node
        }
        else if(flag2==1) //当前结点的邻接点有 1 个未访问
        {
            //新建一个队列, 申请空间, 并加入第一个结点
            flag2=2;
        }
        else if(flag2==2)//当前结点的邻接点有 2 个未被访问
        {
            enQueue(tmpQ,v1);
        }
    }
}

```

```

v1=nextAdjacent(inGraph,inV,v1);
}

if(flag2==2)//push adjacent nodes which are not visited.
{
    //将存有当前结点的 m-1 个未访问邻接点的队列压栈
    seqPush(testStack,tmpQ);
    inV=lChildV;
}
else if(flag2==1)//only has one adjacent which has been visited.
{
    //只有一个最左儿子，则置当前点为最左儿子
    inV=lChildV;
}
else if(flag2==0)
    //has no adjacent nodes or all adjacent nodes has been visited
{
    //当当前的结点的邻接点均已访问或无邻接点需要回访时，则从栈顶的队列结点中弹出队
    //列元素，
    //将队列中的结点元素依次出队，若已访问，则继续出队(当当前队列结点已空时，
    //则继续出栈，弹出下一个栈顶的队列)，直至遇到有未访问结点(访问并置当前点为该点)
    //或直到栈为空。
    flag=0;
    while(!isNullSeqStack(testStack)&&!flag)
    {
        v1=frontQueueInSt(testStack); //返回栈顶结点的队列中的队首元素
        deQueueInSt(testStack); //将栈顶结点的队列中的队首元素弹出
        if(v1->marked==0)
        {
            visit(v1);
            v1->marked=1;
            inV=v1;
            flag=1;
        }
    }
}
}while(!isNullSeqStack(testStack));//the algorithm ends when the stack is null
}
-----
```

上述程序的几点说明：

所以，这里应使用的数据结构的构成方式应该采用下面这种形式：

1)队列的实现中，每个队列结点均为图中的结点指针类型.

定义一个以队列尾部下标加队列长度的环形队列如下：

```
struct SingleRearSeqQueue;
typedef PVexType QElemType;
typedef struct SingleRearSeqQueue* PSingleRearSeqQueue;
struct SingleRearSeqQueue
{
    int rear;
    int quelen;
    QElemType dataPool[MAXNUM];
};
```

其余基本操作不再赘述。

2)堆栈的实现中，每个堆栈中的结点元素均为一个指向队列的指针，定义如下：

```
#define SEQ_STACK_LEN 1000
#define StackElemType PSingleRearSeqQueue
struct SeqStack;
typedef struct SeqStack* PSeqStack;
struct SeqStack
{
    StackElemType dataArea[SEQ_STACK_LEN];
    int slot;
};
```

为了提供更好的封装性，对这个堆栈实现两种特殊的操作

2.1) deQueueInSt 操作用于将栈顶结点的队列中的队首元素弹出.

```
void deQueueInSt(PSeqStack inStack)
{
    if(isEmptyQueue(seqTop(inStack))||isNullSeqStack(inStack))
    {
        printf("in deQueueInSt,under flow!\n");
        return;
    }
    deQueue(seqTop(inStack));
    if(isEmptyQueue(seqTop(inStack)))
        inStack->slot--;
}
```

2.2) frontQueueInSt 操作用以返回栈顶结点的队列中的队首元素.

```
QElemType frontQueueInSt(PSeqStack inStack)
{
    if(isEmptyQueue(seqTop(inStack))||isNullSeqStack(inStack))
```

```
{  
    printf("in frontQueueInSt,under flow!\n");  
    return    '\r';  
}  
  
return getHeadData(seqTop(inStack));  
}
```

=====

ok，本文完。

July、二零一一年一月一日。Happy 2011 new year!

五、教你透彻了解红黑树

作者：July、saturnman 2010 年 12 月 29 日

本文参考：Google、算法导论、STL 源码剖析、计算机程序设计艺术。

本人声明：个人原创，转载请注明出处。

推荐阅读：[**Left-Leaning Red-Black Trees**](#), Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008.

直接下载：<http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>

红黑树系列，六篇文章于今日已经完成：

- [**1、教你透彻了解红黑树**](#)
 - [**2、红黑树算法的实现与剖析**](#)
 - [**3、红黑树的 c 源码实现与剖析**](#)
 - [**4、一步一图一代码，R-B Tree**](#)
 - [**5、红黑树插入和删除结点的全程演示**](#)
 - [**6、红黑树的 c++完整实现源码**](#)
-

一、红黑树的介绍

先来看下算法导论对 R-B Tree 的介绍：

红黑树，一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是 Red 或 Black。

通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的。

前面说了，红黑树，是一种二叉查找树，既然是二叉查找树，那么它必满足二叉查找树的一般性质。

下面，在具体介绍红黑树之前，咱们先来了解下 二叉查找树的一般性质：

1.在一棵二叉查找树上，执行查找、插入、删除等操作，的时间复杂度为 $O(\lg n)$ 。

因为，一棵由 n 个结点，随机构造的二叉查找树的高度为 $\lg n$ ，所以顺理成章，一般操作的执行时间为 $O(\lg n)$ 。

//至于 n 个结点的二叉树高度为 $\lg n$ 的证明，可参考算法导论 第 12 章 二叉查找树 第 12.4 节。

2.但若是一棵具有 n 个结点的线性链，则此些操作最坏情况运行时间为 $O(n)$ 。

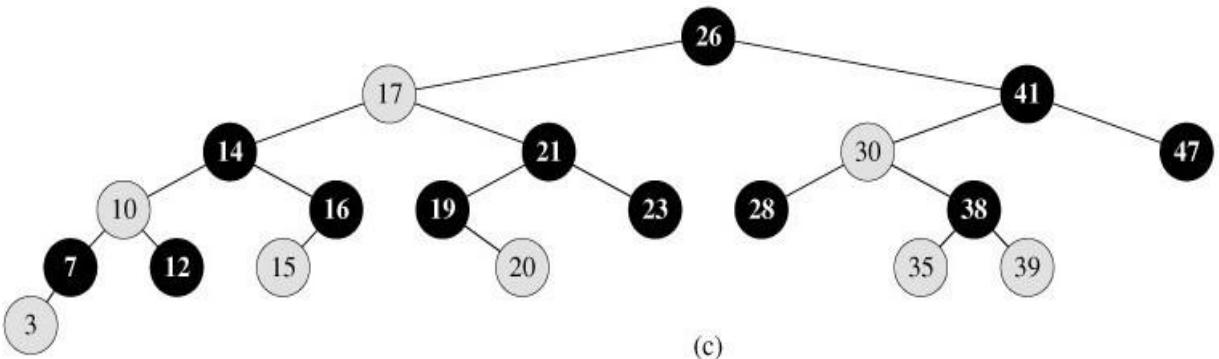
而红黑树，能保证在最坏情况下，基本的动态几何操作的时间均为 $O(\lg n)$ 。

ok，我们知道，红黑树上每个结点内含五个域，color, key, left, right, p。如果相应的指针域没有，则设为 NIL。

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：

- 1) 每个结点要么是红的，要么是黑的。
- 2) 根结点是黑的。
- 3) 每个叶结点，即空结点（NIL）是黑的。
- 4) 如果一个结点是红的，那么它的俩个儿子都是黑的。
- 5) 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

下图所示，即是一颗红黑树：



此图忽略了叶子和根部的父结点。

二、树的旋转知识

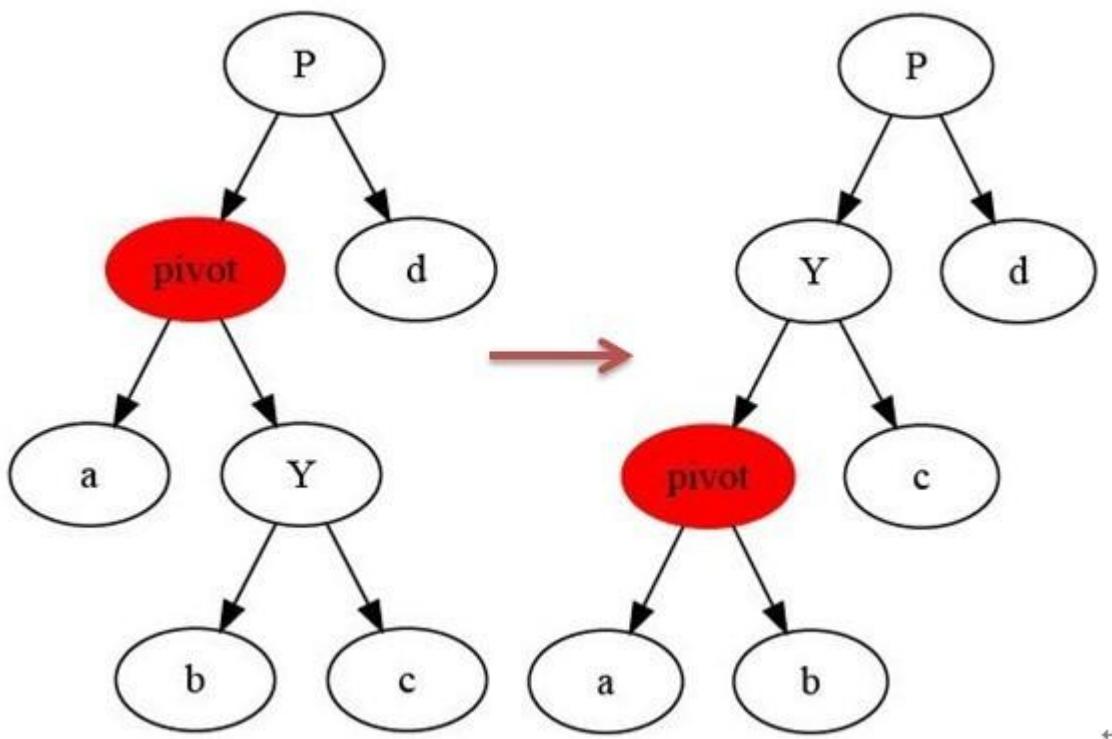
当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了保持红黑树的性质，我们可以通过对树进行旋转，即修改树中某些结点的颜色及指针结构，以达到对红黑树进行

插入、删除结点等操作时，红黑树依然能保持它特有的性质（如上文所述的，五点性质）。

树的旋转，分为左旋和右旋，以下借助图来做形象的解释和介绍：

1. 左旋



如上图所示：

当在某个结点 pivot 上，做左旋操作时，我们假设它的右孩子 y 不是 NIL[T]，pivot 可以为树内任意右孩子而不是 NIL[T] 的结点。

左旋以 pivot 到 y 之间的链为“支轴”进行，它使 y 成为该子树新的根，而 y 的左孩子 b 则成为 pivot 的右孩子。

来看算法导论对此操作的算法实现（以 x 代替上述的 pivot）：

```

LEFT-ROTATE( $T, x$ )
1  $y \leftarrow \text{right}[x]$  ▷ Set  $y$ .
2  $\text{right}[x] \leftarrow \text{left}[y]$       ▷ Turn  $y$ 's left subtree into  $x$ 's right subtree.
3  $p[\text{left}[y]] \leftarrow x$ 
4  $p[y] \leftarrow p[x]$             ▷ Link  $x$ 's parent to  $y$ .
5 if  $p[x] = \text{nil}[T]$ 
6   then  $\text{root}[T] \leftarrow y$ 
```

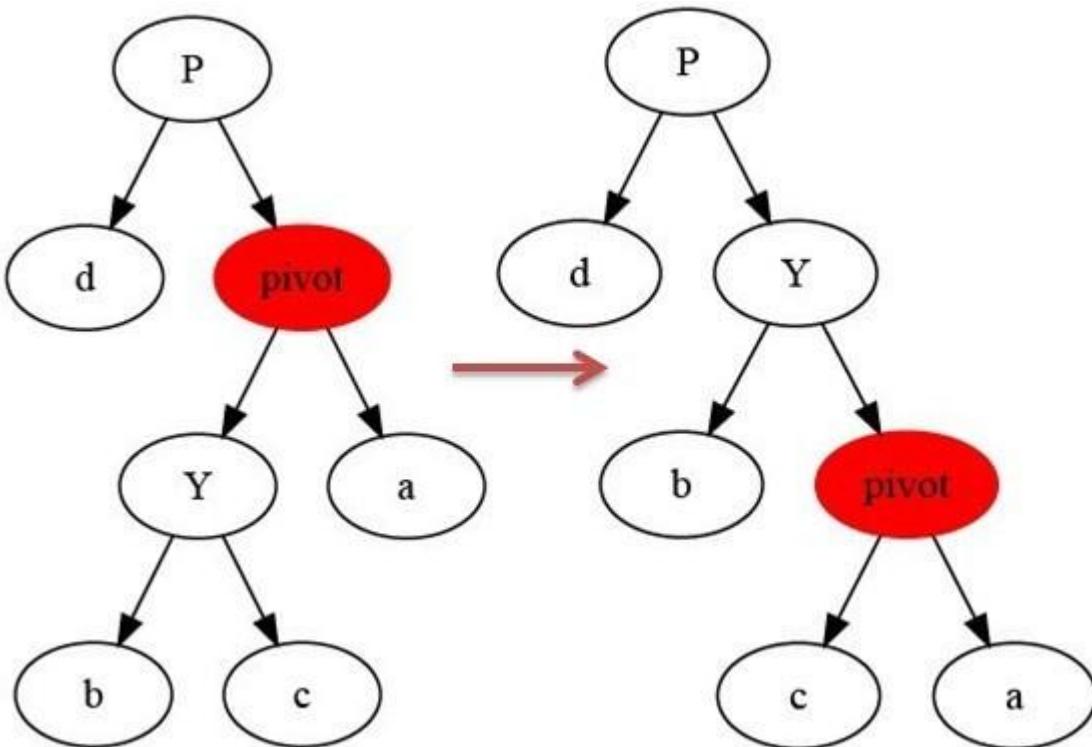
```

7   else if  $x = \text{left}[p[x]]$ 
8       then  $\text{left}[p[x]] \leftarrow y$ 
9       else  $\text{right}[p[x]] \leftarrow y$ 
10   $\text{left}[y] \leftarrow x$             $\triangleright$  Put  $x$  on  $y$ 's left.
11   $p[x] \leftarrow y$ 

```

2.右旋

右旋与左旋差不多，再此不做详细介绍。



对于树的旋转，能保持不变的只有原树的搜索性质，而原树的红黑性质则不能保持，在红黑树的数据插入和删除后可利用旋转和颜色重涂来恢复树的红黑性质。

至于有些书如 STL 源码剖析有对双旋的描述，其实双旋只是单旋的两次应用，并无新的内容，因此这里就不再介绍了，而且左右旋也是相互对称的，只要理解其中一种旋转就可以了。

三、红黑树插入、删除操作的具体实现

三、I、ok，接下来，咱们来具体了解红黑树的**插入**操作。
向一棵含有 n 个结点的红黑树插入一个新结点的操作可以在 $O(\lg n)$ 时间内完成。

算法导论：

```
RB-INSERT(T, z)
1 y ← nil[T]
2 x ← root[T]
3 while x ≠ nil[T]
4   do y ← x
5     if key[z] < key[x]
6       then x ← left[x]
7       else x ← right[x]
8 p[z] ← y
9 if y = nil[T]
10  then root[T] ← z
11  else if key[z] < key[y]
12    then left[y] ← z
13    else right[y] ← z
14 left[z] ← nil[T]
15 right[z] ← nil[T]
16 color[z] ← RED
17 RB-INSERT-FIXUP(T, z)
```

咱们来具体分析下，此段代码：

RB-INSERT(T, z)，将 z 插入红黑树 T 之内。

为保证红黑性质在插入操作后依然保持，上述代码调用了一个辅助程序

RB-INSERT-FIXUP

来对结点进行重新着色，并旋转。

```
14 left[z] ← nil[T]
15 right[z] ← nil[T] //保持正确的树结构
```

第 16 行，将 z 着为红色，由于将 z 着为红色可能会违背某一条红黑树的性质，所以，在第 17 行，调用 RB-INSERT-FIXUP (T,z) 来保持红黑树的性质。

RB-INSERT-FIXUP(T, z)，如下所示：

```
1 while color[p[z]] = RED
2   do if p[z] = left[p[p[z]]]
3     then y ← right[p[p[z]]]
4       if color[y] = RED
5         then color[p[z]] ← BLACK           ▷ Case 1
6           color[y] ← BLACK               ▷ Case 1
7           color[p[p[z]]] ← RED          ▷ Case 1
8           z ← p[p[z]]                  ▷ Case 1
9       else if z = right[p[z]]
10      then z ← p[z]                   ▷ Case 2
11      LEFT-ROTATE(T, z)             ▷ Case 2
12      color[p[z]] ← BLACK          ▷ Case 3
```

```

13           color[p[p[z]]] ← RED      ▷ Case 3
14           RIGHT-ROTATE(T, p[p[z]])    ▷ Case 3
15       else (same as then clause
           with "right" and "left" exchanged)
16 color[root[T]] ← BLACK

```

ok, 参考一网友的言论, 用自己的语言, 再来具体解剖下上述俩段代码。
为了保证阐述清晰, 我再写下红黑树的 5 个性质:

- 1) 每个结点要么是红的, 要么是黑的。**
- 2) 根结点是黑的。**
- 3) 每个叶结点, 即空结点 (**NIL**) 是黑的。**
- 4) 如果一个结点是红的, 那么它的俩个儿子都是黑的。**
- 5) 对每个结点, 从该结点到其子孙结点的所有路径上包含相同数目的黑结点。**

在对红黑树进行插入操作时, 我们一般总是插入红色的结点, 因为这样可以在插入过程中尽量避免对树的调整。

那么, 我们插入一个结点后, 可能会使原树的哪些性质改变列?

由于, 我们是按照二叉树的方式进行插入, 因此元素的搜索性质不会改变。

如果插入的结点是根结点, 性质 2 会被破坏, 如果插入结点的父结点是红色, 则会破坏性质 4。

因此, 总而言之, 插入一个红色结点只会破坏性质 2 或性质 4。

我们的回复策略很简单,

其一、把出现违背红黑树性质的结点向上移, 如果能移到根结点, 那么很容易就能通过直接修改根结点来恢复红黑树的性质。直接通过修改根结点来恢复红黑树应满足的性质。

其二、穷举所有的可能性, 之后把能归于同一类方法处理的归为同一类, 不能直接处理的化归到下面的几种情况,

//注: 以下情况 3、4、5 与上述算法导论上的代码 RB-INSERT-FIXUP(T, z), 相对应:

情况 1: 插入的是根结点。

原树是空树, 此情况只会违反性质 2。

对策: 直接把此结点涂为黑色。

情况 2: 插入的结点的父结点是黑色。

此不会违反性质 2 和性质 4, 红黑树没有被破坏。

对策: 什么也不做。

情况 3: 当前结点的父结点是红色且祖父结点的另一个子结点 (叔叔结点) 是红色。

此时父结点的父结点一定存在, 否则插入前就已不是红黑树。

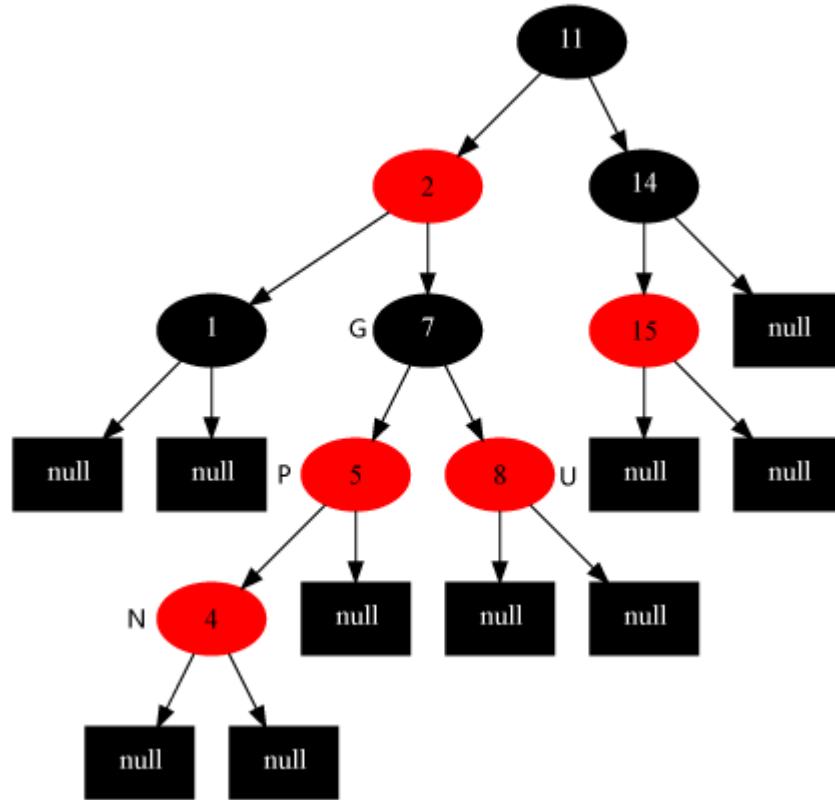
与此同时, 又分为父结点是祖父结点的左子还是右子, 对于对称性, 我们只要解开一个方向就可以了。

在此, 我们只考虑父结点为祖父左子的情况。

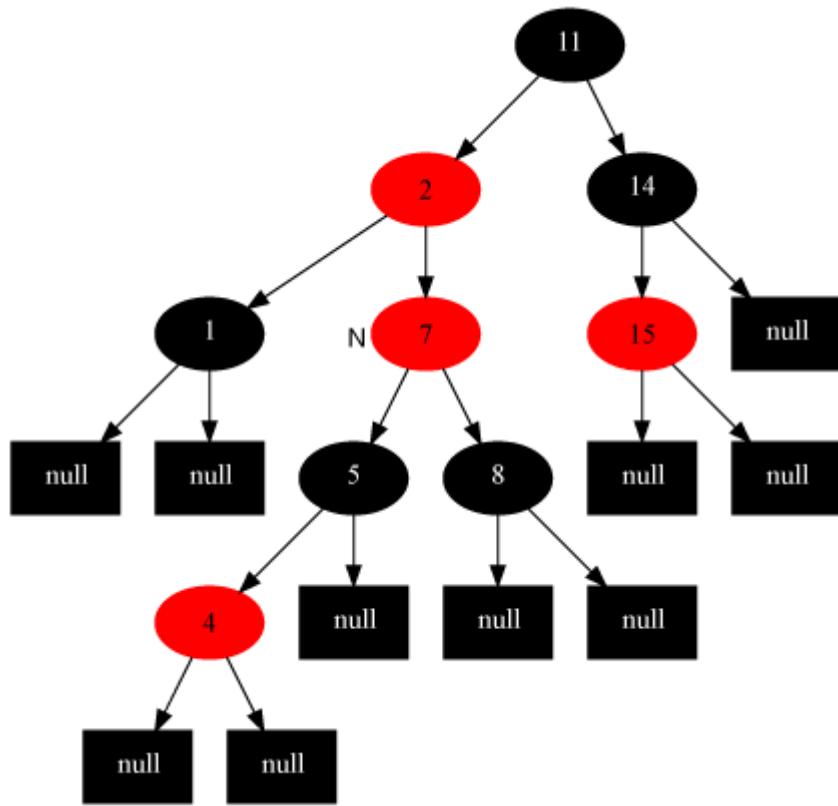
同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。

对策：将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法。

针对情况 3，变化前（图片来源：[saturnman](#)）[[插入 4 节点](#)]：



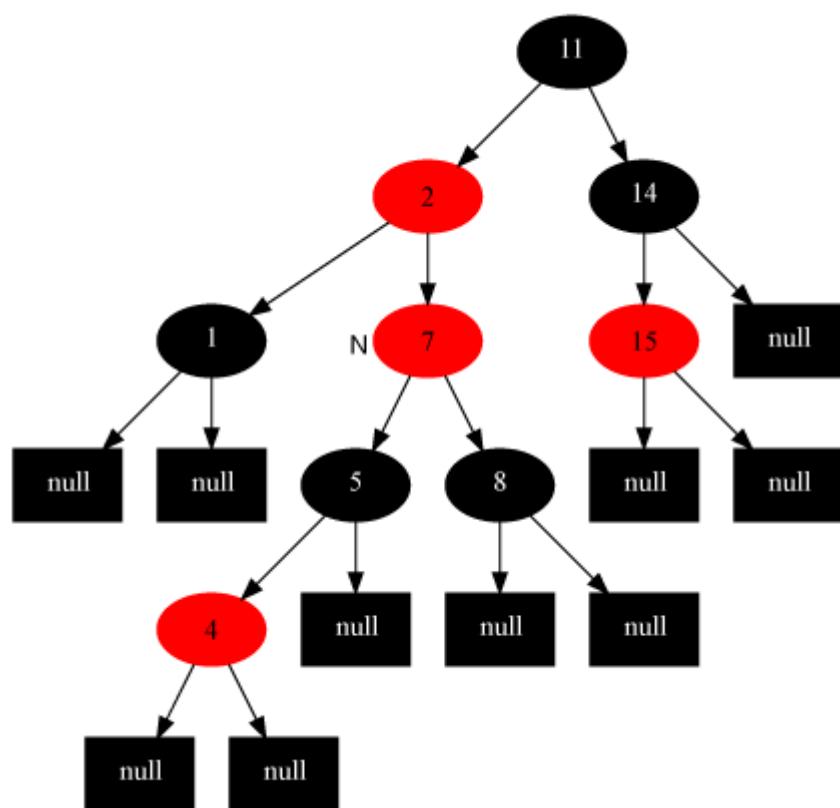
变化后：



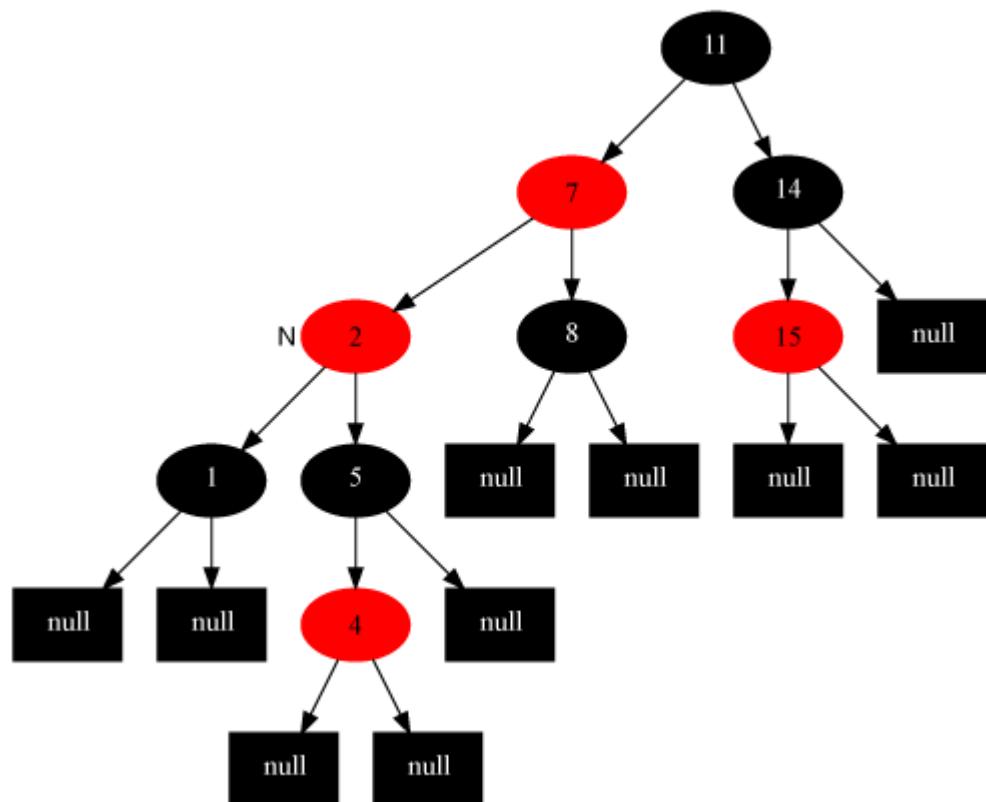
情况 4：当前节点的父节点是红色，叔叔节点是黑色，当前节点是其父节点的右子

对策：当前节点的父节点做为新的当前节点，以新当前节点为支点左旋。

如下图所示，变化前[[插入 7 节点](#)]：

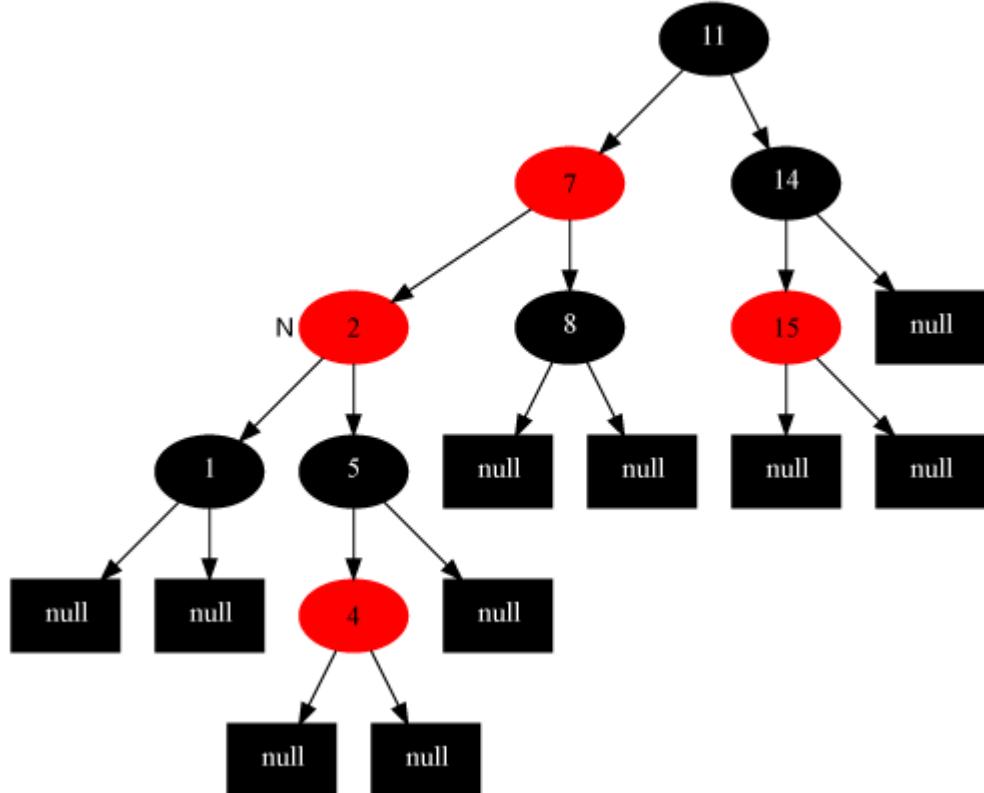


变化后：

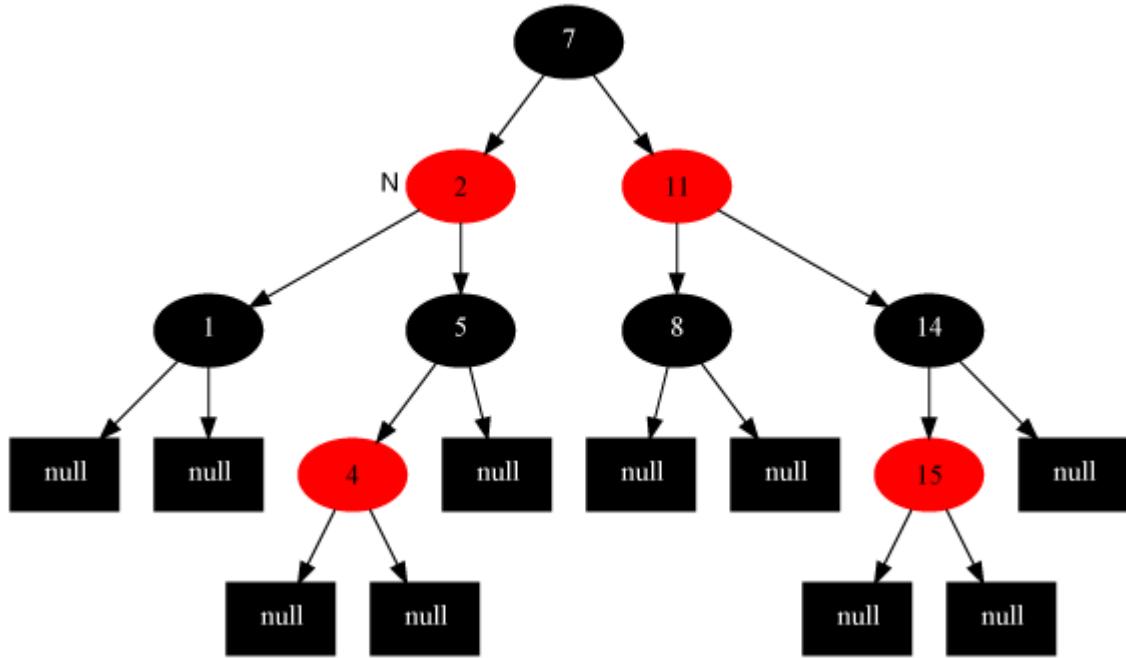


情况 5: 当前节点的父节点是红色, 叔叔节点是黑色, 当前节点是其父节点的左子

解法: 父节点变为黑色, 祖父节点变为红色, 在祖父节点为支点右旋
如下图所示[[插入 2 节点](#)]



变化后:



=====

三、II、ok，接下来，咱们最后来了解，红黑树的删除操作：

算法导论一书，给的算法实现：

RB-DELETE(T, z) 单纯删除结点的总操作

```

1 if left[z] = nil[T] or right[z] = nil[T]
2   then y ← z
3   else y ← TREE-SUCCESSOR(z)
4 if left[y] ≠ nil[T]
5   then x ← left[y]
6   else x ← right[y]
7 p[x] ← p[y]
8 if p[y] = nil[T]
9   then root[T] ← x
10  else if y = left[p[y]]
11    then left[p[y]] ← x
12    else right[p[y]] ← x
13 if y ≠ z
14  then key[z] ← key[y]
15  copy y's satellite data into z
16 if color[y] = BLACK
17  then RB-DELETE-FIXUP( $T, x$ )
18 return y

```

RB-DELETE-FIXUP(T, x) 恢复与保持红黑性质的工作

```
1 while  $x \neq \text{root}[T]$  and  $\text{color}[x] = \text{BLACK}$ 
2   do if  $x = \text{left}[p[x]]$ 
3     then  $w \leftarrow \text{right}[p[x]]$ 
4       if  $\text{color}[w] = \text{RED}$ 
5         then  $\text{color}[w] \leftarrow \text{BLACK}$            ▷ Case 1
6            $\text{color}[p[x]] \leftarrow \text{RED}$            ▷ Case 1
7            $\text{LEFT-ROTATE}(T, p[x])$            ▷ Case 1
8            $w \leftarrow \text{right}[p[x]]$            ▷ Case 1
9       if  $\text{color}[\text{left}[w]] = \text{BLACK}$  and  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
10      then  $\text{color}[w] \leftarrow \text{RED}$            ▷ Case 2
11         $x \leftarrow p[x]$            ▷ Case 2
12      else if  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
13        then  $\text{color}[\text{left}[w]] \leftarrow \text{BLACK}$            ▷ Case 3
14           $\text{color}[w] \leftarrow \text{RED}$            ▷ Case 3
15           $\text{RIGHT-ROTATE}(T, w)$            ▷ Case 3
16           $w \leftarrow \text{right}[p[x]]$            ▷ Case 3
17         $\text{color}[w] \leftarrow \text{color}[p[x]]$            ▷ Case 4
18         $\text{color}[p[x]] \leftarrow \text{BLACK}$            ▷ Case 4
19         $\text{color}[\text{right}[w]] \leftarrow \text{BLACK}$            ▷ Case 4
20         $\text{LEFT-ROTATE}(T, p[x])$            ▷ Case 4
21         $x \leftarrow \text{root}[T]$            ▷ Case 4
22    else (same as then clause with "right" and "left" exchanged)
23  $\text{color}[x] \leftarrow \text{BLACK}$ 
```

为了保证以下的介绍与阐述清晰，我第三次重写下红黑树的 5 个性质

- 1) 每个结点要么是红的，要么是黑的。
- 2) 根结点是黑的。
- 3) 每个叶结点，即空结点 (**NIL**) 是黑的。
- 4) 如果一个结点是红的，那么它的两个儿子都是黑的。
- 5) 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

(相信，重述了 3 次，你应该有深刻记忆了。:D)

saturnman:

红黑树删除的几种情况：

博主提醒：

以下所有的操作，是针对红黑树已经删除结点之后，
为了恢复和保持红黑树原有的 5 点性质，所做的恢复工作。

前面，我已经说了，因为插入、或删除结点后，可能会违背、或破坏红黑树的原有的性质，所以为了使插入、或删除结点后的树依然维持为一棵新的红黑树，那就要做俩方面的工作：

- 1、部分结点颜色，重新着色
- 2、调整部分指针的指向，即左旋、右旋。

而下面所有的文字，则是针对红黑树删除结点后，所做的修复红黑树性质的工作。

二零一一年一月七日更新。

(注：以下的情况 3、4、5、6，与上述算法导论之代码 RB-DELETE-FIXUP(T, x) 恢复与保持

中 case1, case2, case3, case4 相对应。)

情况 1：当前节点是红色

解法，直接把当前节点染成黑色，结束。

此时红黑树性质全部恢复。

情况 2：当前节点是黑色且是根节点

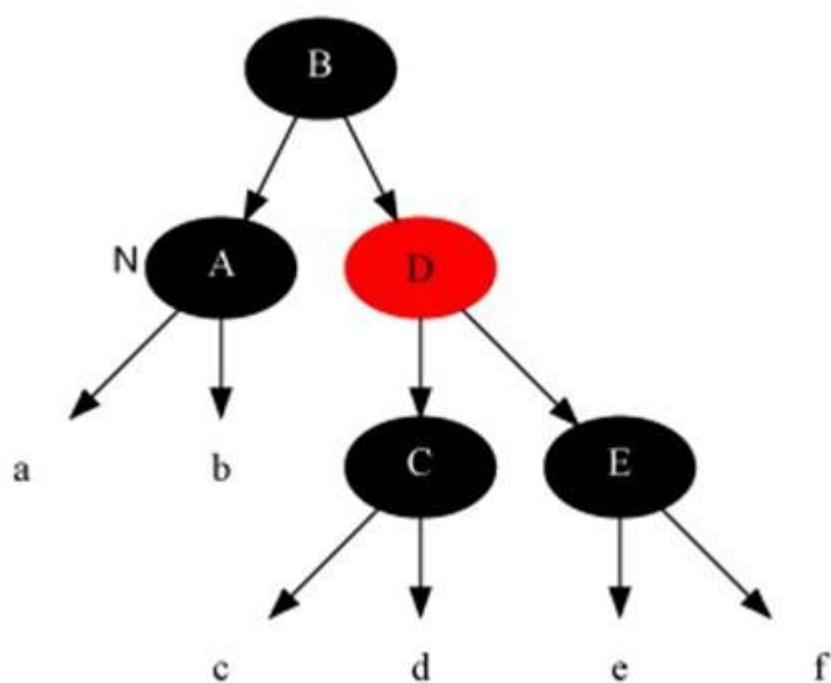
解法：什么都不做，结束

情况 3：当前节点是黑色，且兄弟节点为红色(此时父节点和兄弟节点的子节点分为黑)。

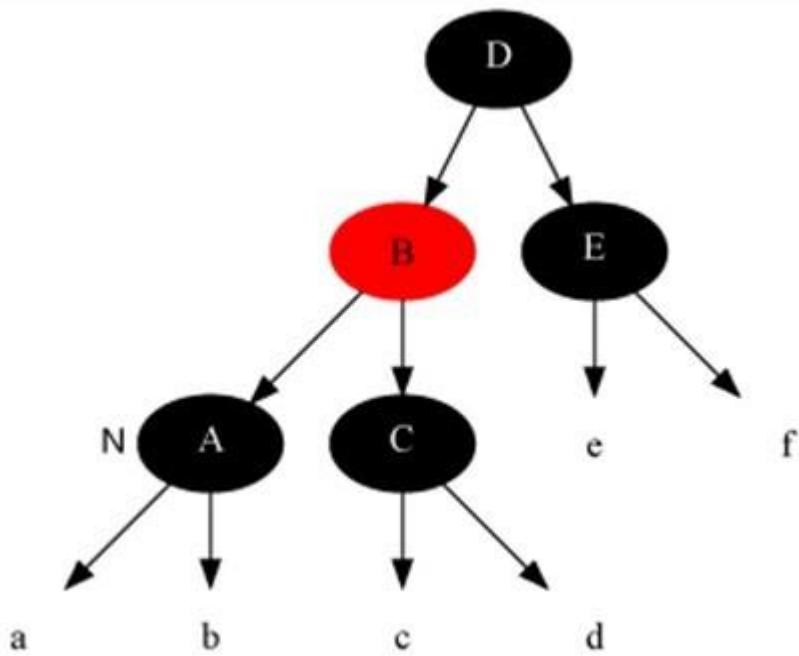
解法：把父节点染成红色，把兄弟结点染成黑色，之后重新进入算法（我们只讨论当前节点是其父节点左孩子时的情况）。

然后，针对父节点做一次左旋。此变换后原红黑树性质 5 不变，而把问题转化为兄弟节点为黑色的情况。

3. 变化前：



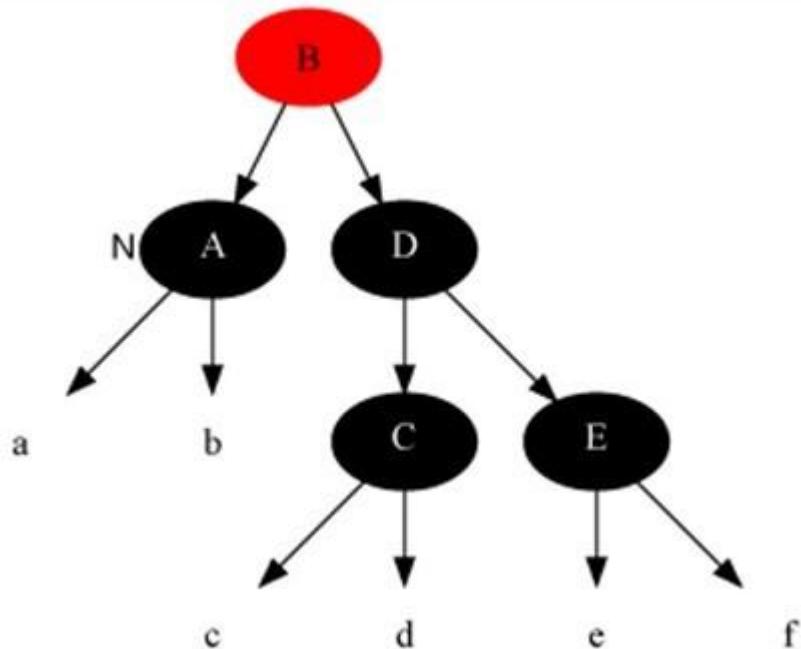
3.变化后:



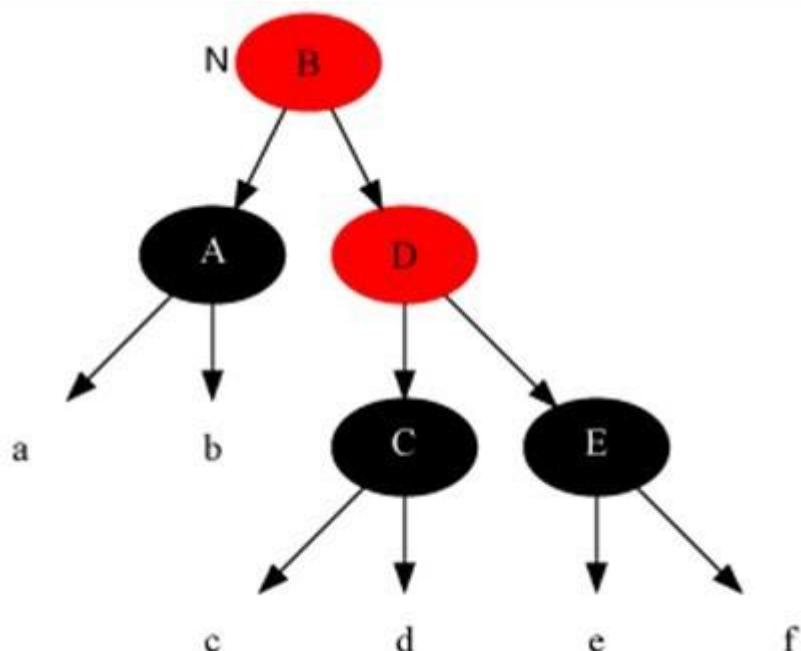
情况 4: 当前节点是黑色, 且兄弟是黑色, 且兄弟节点的两个子节点全为黑色。

解法: 把当前节点和兄弟节点中抽取一重黑色追加到父节点上, 把父节点当成新的当前节点, 重新进入算法。(此变换后性质 5 不变)

4.变化前



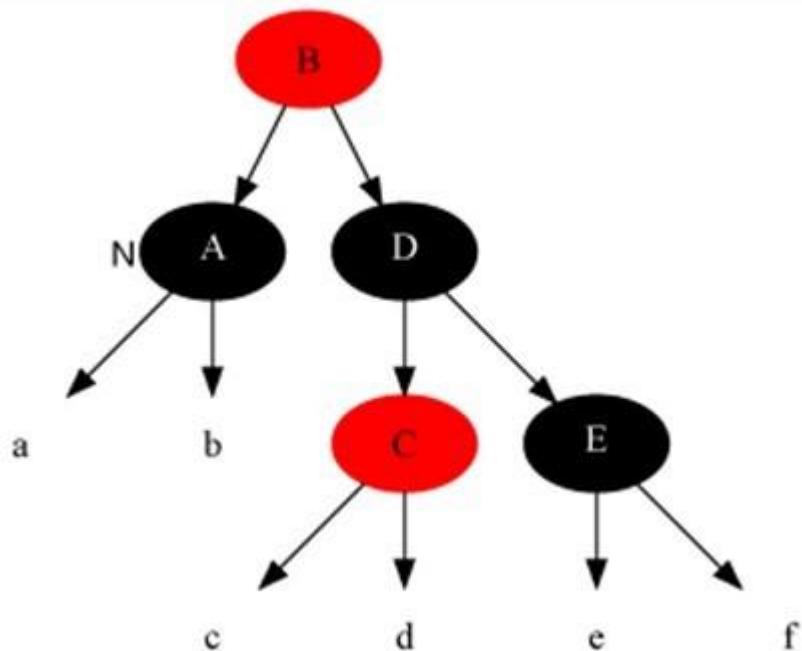
4.变化后



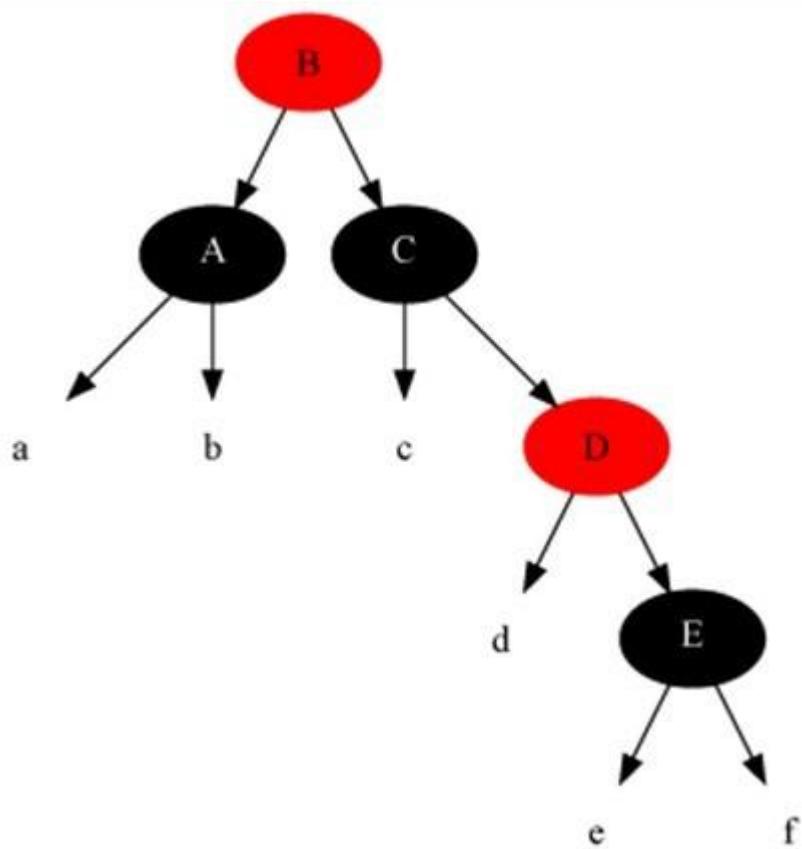
情况 5: 当前节点颜色是黑色, 兄弟节点是黑色, 兄弟的左子是红色, 右子是黑色。

解法: 把兄弟结点染红, 兄弟左子节点染黑, 之后再在兄弟节点为支点解右旋, 之后重新进入算法。此是把当前的情况转化为情况 6, 而性质 5 得以保持。

5. 变化前:



5. 变化后:

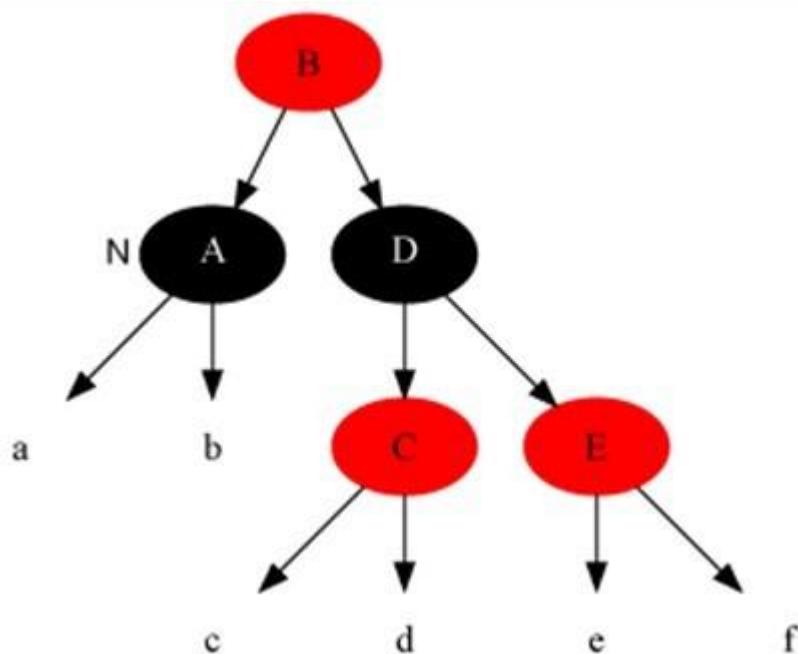


情况 6: 当前节点颜色是黑色, 它的兄弟节点是黑色, 但是兄弟节点的右子是红色, 兄弟节点左子的颜色任意。

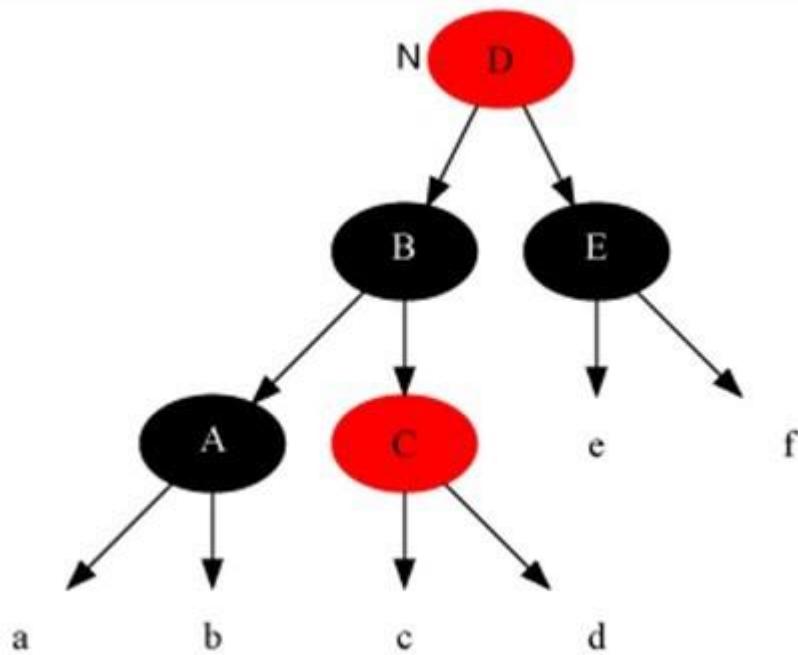
解法: 把兄弟节点染成当前节点父节点的颜色, 把当前节点父节点染成黑色, 兄弟节点右子染成黑色,

之后以当前节点的父节点为支点进行左旋, 此时算法结束, 红黑树所有性质调整正确。

6. 变化前:



6. 变化后:



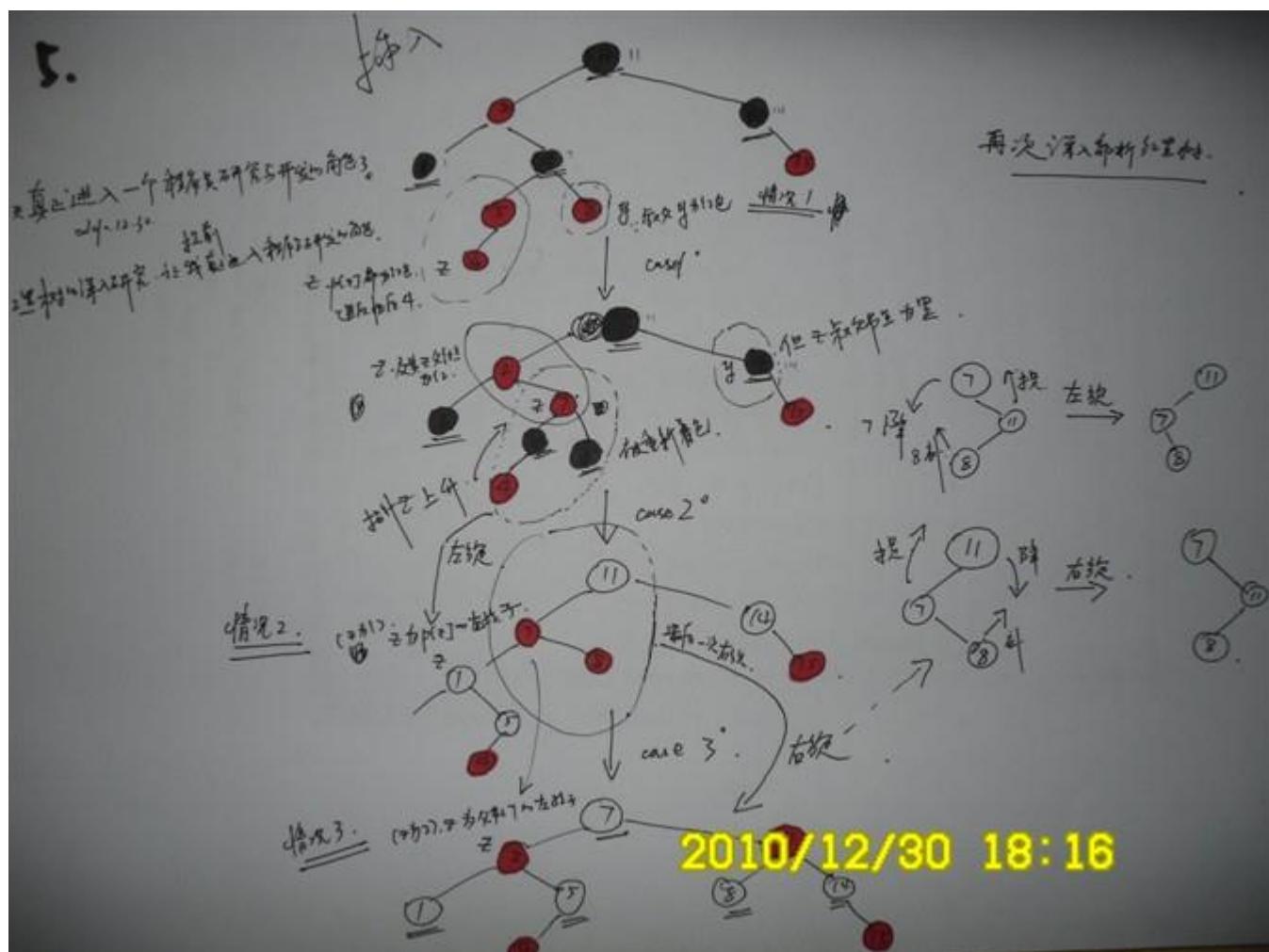
限于篇幅, 不再过多赘述。更多, 可参考算法导论或下文我写的第二篇文章。

完。

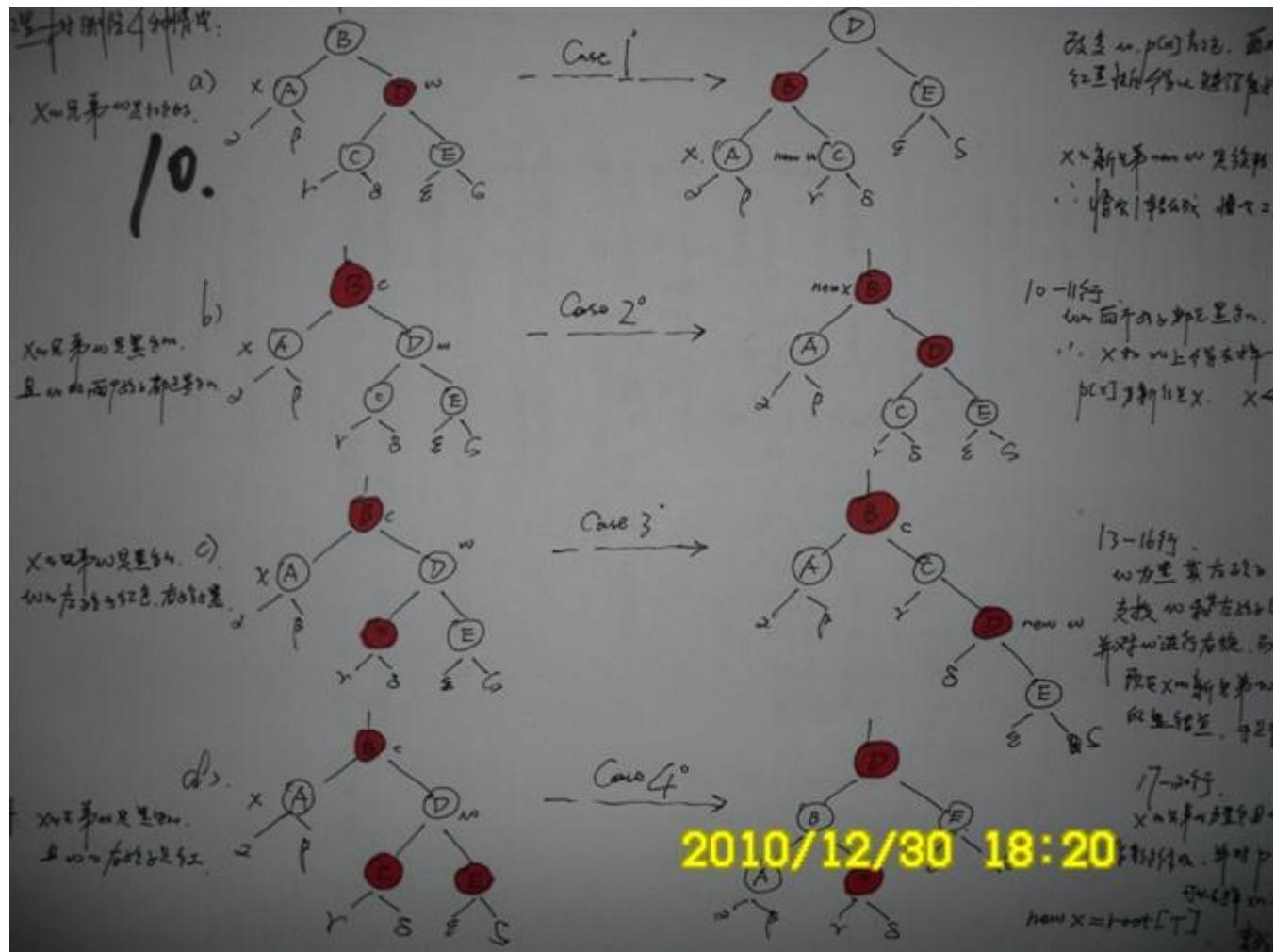
July、二零一零年十二月二十九日初稿。三十日凌晨修订。行文 3 个小时以上。

今下午画红黑树画了好几个钟头，贴俩张图：

红黑树插入的 3 种情况：



红黑树删除的 4 种情况：



ok, 只贴俩张, 更多, 参考我写的关于红黑树的第二篇文章:

红黑树算法的层层剖析与逐步实现 [推荐]

http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx

这篇文章针对算法实现源码分十层, 层层、逐层剖析, 相信, 更清晰易懂。

或 者 , saturnman 的 这 篇 文

章: <http://saturnman.blog.163.com/blog/static/5576112010969420383/>。

July、二零一零年十二月三十一日、最后更新。

- 1、教你透彻了解红黑树
- 2、红黑树算法的实现与剖析
- 3、红黑树的 c 源码实现与剖析
- 4、一步一图一代码, R-B Tree
- 5、红黑树插入和删除结点的全程演示
- 6、红黑树的 c++完整实现源码

五(续)、红黑树算法的层层剖析与逐步实现

作者 July 二零一零年十二月三十一日

本文主要参考：算法导论第二版

本文主要代码：参考算法导论。

本文图片来源：个人手工画成、算法导论原书。

推荐阅读：[Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的关于红黑树的一篇论文。](#)

- 1、教你透彻了解红黑树**
 - 2、红黑树算法的实现与剖析**
 - 3、红黑树的 c 源码实现与剖析**
 - 4、一步一图一代码，R-B Tree**
 - 5、红黑树插入和删除结点的全程演示**
 - 6、红黑树的 c++完整实现源码**
-

引言：

昨天下午画红黑树画了好几个钟头，总共 10 页纸。

特此，再深入剖析红黑树的算法实现，教你如何彻底实现红黑树算法。

经过我上一篇博文，“教你透彻了解红黑树”后，相信大家对红黑树已经有了一定的了解。

个人觉得，这个红黑树，还是比较容易懂的。

不论是插入、还是删除，不论是左旋还是右旋，最终的目的只有一个：

即保持红黑树的 5 个性质，不得违背。

再次，重述下红黑树的5个性质：

一般的，红黑树，满足一下性质，即只有满足一下性质的树，我们才称之为红黑树：

- 1) 每个结点要么是红的，要么是黑的。
- 2) 根结点是黑的。
- 3) 每个叶结点，即空结点（NIL）是黑的。
- 4) 如果一个结点是红的，那么它的俩个儿子都是黑的。
- 5) 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

抓住了红黑树的那 5 个性质，事情就好办多了。

如，

1. 红黑红黑，要么是红，要么是黑；
2. 根结点是黑；

3. 每个叶结点是黑;
4. 一个红结点, 它的俩个儿子必然都是黑的;
5. 每一条路径上, 黑结点的数目等同。

五条性质, 合起来, 来句顺口溜就是: (1) 红黑 (2) 黑 (3) 黑 (4&5) 红->黑 黑。

本文所有的文字, 都是参照我昨下午画的十张纸(即我拍的照片)与算法导论来写的。
希望, 你依照此文一点一点的往下看, 看懂此文后, 你对红黑树的算法了解程度, 一定大增不少。

ok, 现在咱们来具体深入剖析红黑树的算法, 并教你逐步实现此算法。

此教程分为 10 个部分, 每一个部分作为一个小节。且各小节与我给的十张照片一一对应。

一、左旋与右旋

先明确一点: 为什么要左旋?

因为红黑树插入或删除结点后, 树的结构发生了变化, 从而可能会破坏红黑树的性质。

为了维持插入、或删除结点后的树, 仍然是一颗红黑树, 所以有必要对树的结构做部分调整, 从而恢复红黑树的原本性质。

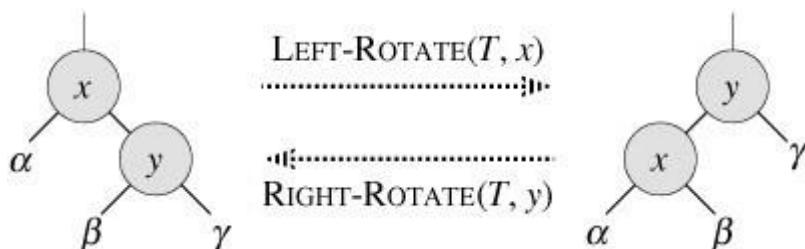
而为了恢复红黑性质而作的动作包括:

结点颜色的改变(重新着色), 和结点的调整。

这部分结点调整工作, 改变指针结构, 即是通过左旋或右旋而达到目的。

从而使插入、或删除结点的树重新成为一颗新的红黑树。

ok, 请看下图:



如上图所示, ‘找茬’

如果你看懂了上述俩幅图有什么区别时, 你就知道什么是“左旋”, “右旋”。

在此, 着重分析左旋算法:

左旋, 如图所示(左->右), 以 x->y 之间的链为“支轴”进行,

使 y 成为该新子树的根, x 成为 y 的左孩子, 而 y 的左孩子则成为 x 的右孩子。

算法很简单, 还有注意一点, 各个结点从左往右, 不论是左旋前还是左旋后, 结点大小都是从小到大。

左旋代码实现, 分三步(注意我给的注释):

The pseudocode for LEFT-ROTATE assumes that $\text{right}[x] \neq \text{nil}[T]$ and that the root's parent is $\text{nil}[T]$.

LEFT-ROTATE(T, x)

1 $y \leftarrow \text{right}[x]$ ▷ Set y .

2 $\text{right}[x] \leftarrow \text{left}[y]$ //开始变化, y 的左孩子成为 x 的右孩子

3 if $\text{left}[y] \neq \text{nil}[T]$

4 then $p[\text{left}[y]] \leftarrow x$

```

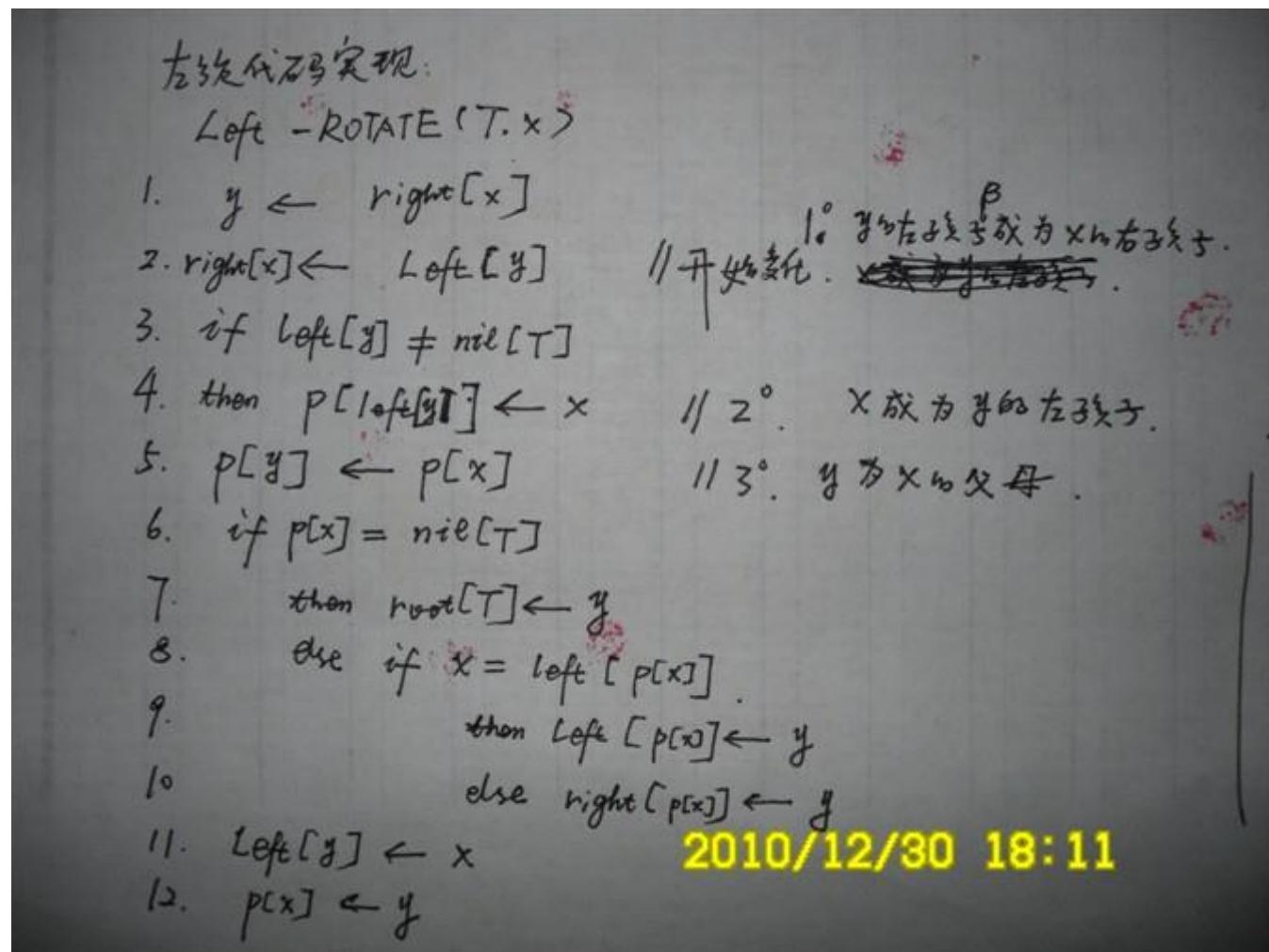
5 p[y] <- p[x]           //y 成为 x 的父母
6 if p[x] = nil[T]
7 then root[T] <- y
8 else if x = left[p[x]]
9     then left[p[x]] <- y
10    else right[p[x]] <- y
11 left[y] <- x          //x 成为 y 的左孩子 (一月三日修正)
12 p[x] <- y

```

//注，此段左旋代码，原书第一版英文版与第二版中文版，有所出入。
//个人觉得，第二版更精准。所以，此段代码以第二版中文版为准。

左旋、右旋都是对称的，且都是在 O(1) 时间内完成。因为旋转时只有指针被改变，而结点中的所有域都保持不变。

最后，贴出昨下午关于此右旋算法所画的图：
左旋（第 2 张图）：

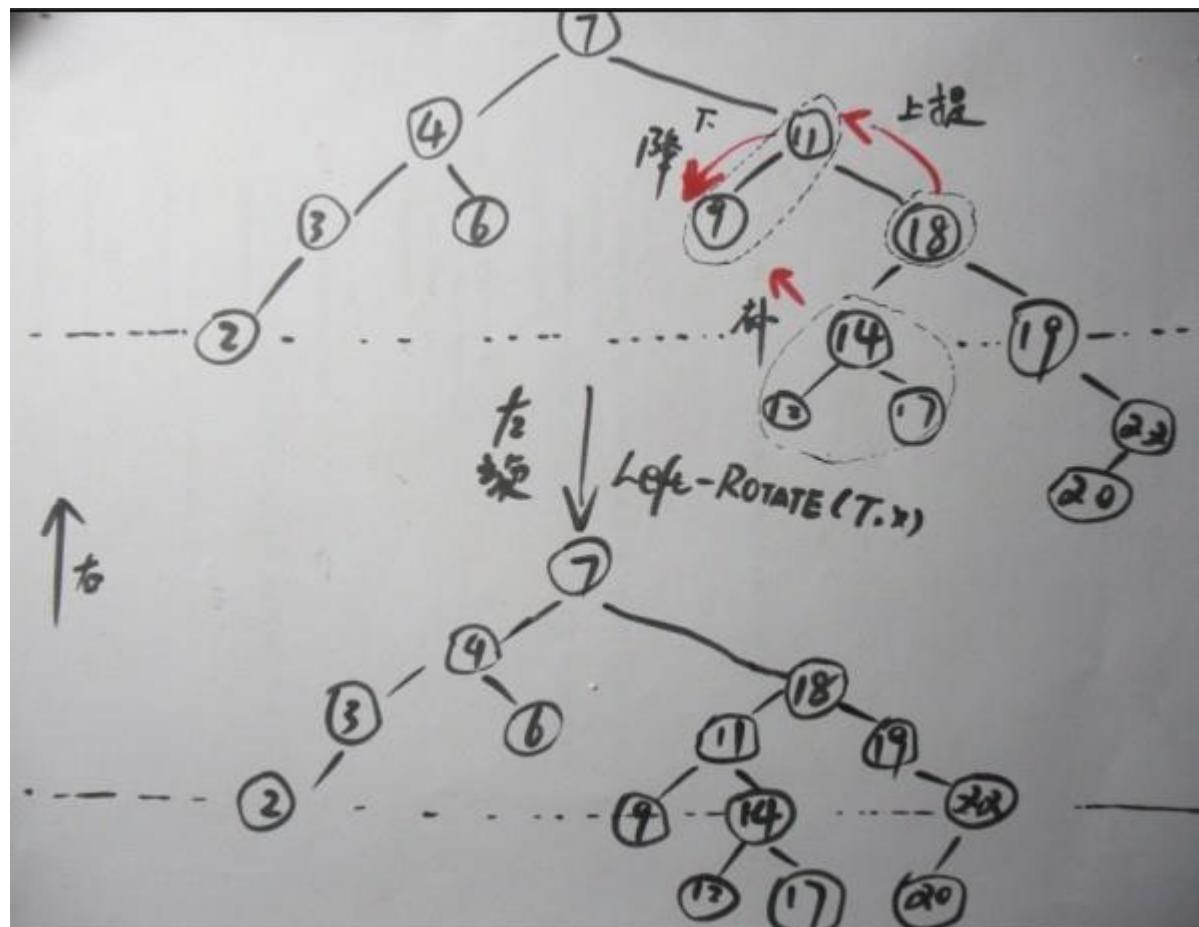


//此图有点 bug。第 4 行的注释移到第 11 行。如上述代码所示。（一月三日修正）

二、左旋的一个实例

不做过多介绍，看下副图，一目了然。

LEFT-ROTATE(T, x)的操作过程（第3张图）：



提醒，看下文之前，请首先务必明确，区别以下俩种操作：

1. 红黑树插入、删除结点的操作

//如插入中，红黑树插入结点操作：RB-INSERT(T, z)。

2. 红黑树已经插入、删除结点之后，

为了保持红黑树原有的红黑性质而做的恢复与保持红黑性质的操作。

//如插入中，为了恢复和保持原有红黑性质，所做的工作：RB-INSERT-FIXUP(T, z)。

ok，请继续。

三、红黑树的插入算法实现

RB-INSERT(T, z) //注意我给的注释...

```
1 y ← nil[T]           // y 始终指向 x 的父结点。  
2 x ← root[T]          // x 指向当前树的根结点，  
3 while x ≠ nil[T]
```

```

4   do y ← x
5     if key[z] < key[x]      //向左, 向右..
6       then x ← left[x]
7       else x ← right[x]    // 为了找到合适的插入点, x 探路跟踪路径, 直到
x 成为 NIL 为止。
8 p[z] ← y      // y 置为 插入结点 z 的父结点。
9 if y = nil[T]
10  then root[T] ← z
11  else if key[z] < key[y]
12    then left[y] ← z
13    else right[y] ← z //此 8-13 行, 置 z 相关的指针。
14 left[z] ← nil[T]
15 right[z] ← nil[T]      //设为空,
16 color[z] ← RED        //将新插入的结点 z 作为红色
17 RB-INSERT-FIXUP(T, z) //因为将 z 着为红色, 可能会违反某一红黑性质,

```

//所以需要调用 RB-INSERT-FIXUP(T, z) 来保持红黑性质。

17 行的 **RB-INSERT-FIXUP(T, z)** , 在下文会得到着重而具体的分析。

还记得, 我开头说的那句话么,

是的, 时刻记住, 不论是左旋还是右旋, 不论是插入、还是删除, 都要记得恢复和保持红黑树的 5 个性质。

四、调用 RB-INSERT-FIXUP(T, z)来保持和恢复红黑性质

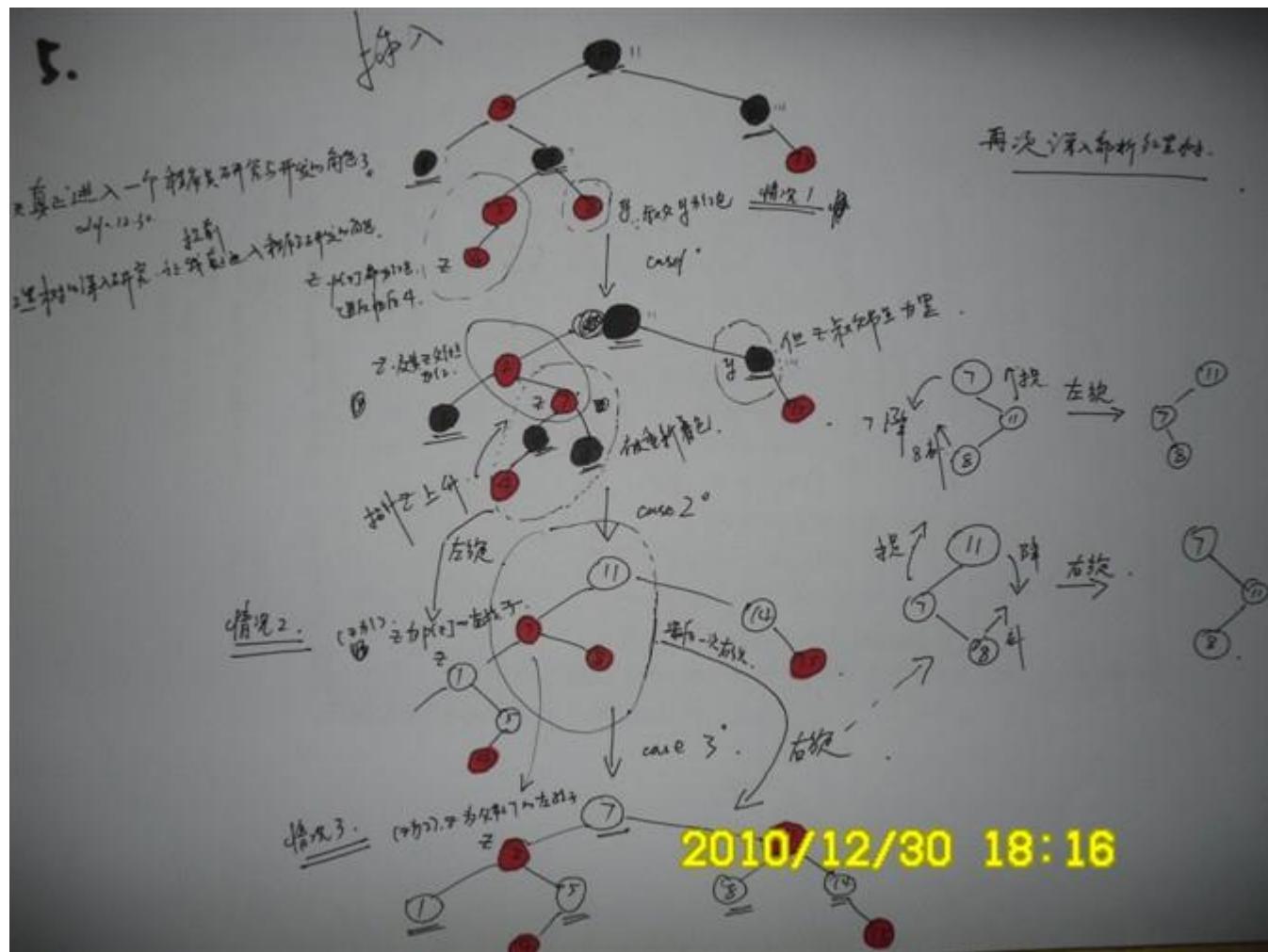
```

RB-INSERT-FIXUP(T, z)
1 while color[p[z]] = RED
2   do if p[z] = left[p[p[z]]]
3     then y ← right[p[p[z]]]
4       if color[y] = RED
5         then color[p[z]] ← BLACK          ▷ Case 1
6           color[y] ← BLACK              ▷ Case 1
7           color[p[p[z]]] ← RED          ▷ Case 1
8           z ← p[p[z]]                  ▷ Case 1
9       else if z = right[p[z]]
10      then z ← p[z]                  ▷ Case 2
11        LEFT-ROTATE(T, z)            ▷ Case 2
12        color[p[z]] ← BLACK          ▷ Case 3
13        color[p[p[z]]] ← RED          ▷ Case 3
14        RIGHT-ROTATE(T, p[p[z]])     ▷ Case 3
15     else (same as then clause
           with "right" and "left" exchanged)
16 color[root[T]] ← BLACK

```

//第4张图略:

五、红黑树插入的三种情况，即 **RB-INSERT-FIXUP(T, z)**。操作过程（第 5 张）：



//这幅图有个小小的问题，读者可能会产生误解。图中左侧所表明的情况 2、情况 3 所标的位置都要标上一点。

//请以图中的标明的 case1、case2、case3 为准。一月三日。

六、红黑树插入的第一种情况（RB-INSERT-FIXUP(T, z)代码的具体分析一）

为了保证阐述清晰，重述下 RB-INSERT-FIXUP(T, z)的源码：

RB-INSERT-FIXUP(T , z)

1 while color[p[z]] = RED

2 do if $p[z] = \text{left}[p[p[z]]]$

3 then $y \leftarrow \text{right}[p[p[2$

4 if color[y] = RED

▷ Case 1

5 then color[p[z]] \leftarrow BLACK ▷ Case 3

```

7           color[p[p[z]]] ← RED          ▷ Case 1
8           z ← p[p[z]]                  ▷ Case 1
9   else if z = right[p[z]]
10      then z ← p[z]                  ▷ Case 2
11          LEFT-ROTATE(T, z)          ▷ Case 2
12          color[p[z]] ← BLACK        ▷ Case 3
13          color[p[p[z]]] ← RED       ▷ Case 3
14          RIGHT-ROTATE(T, p[p[z]])    ▷ Case 3
15   else (same as then clause
           with "right" and "left" exchanged)
16 color[root[T]] ← BLACK

```

//case1 表示情况 1, case2 表示情况 2, case3 表示情况 3.

ok, 如上所示, 相信, 你已看到了。

咱们, 先来透彻分析红黑树插入的第一种情况:

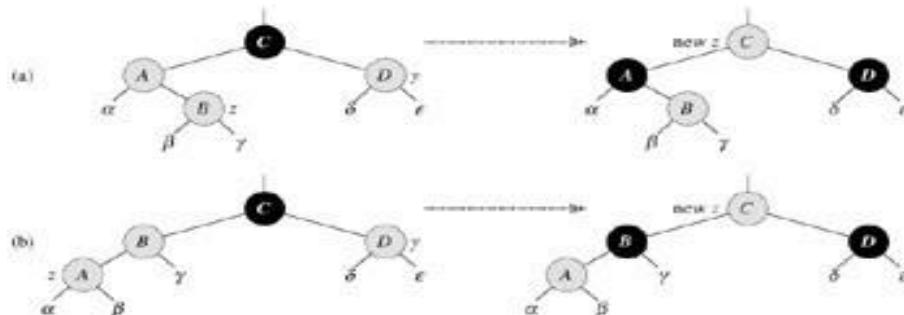
插入情况 1, z 的叔叔 y 是红色的。

第一种情况, 即上述代码的第 5-8 行:

```

5   then color[p[z]] ← BLACK          ▷ Case 1
6   color[y] ← BLACK                 ▷ Case 1
7   color[p[p[z]]] ← RED            ▷ Case 1
8   z ← p[p[z]]                   ▷ Case 1

```



如上图所示, a: z 为右孩子, b: z 为左孩子。

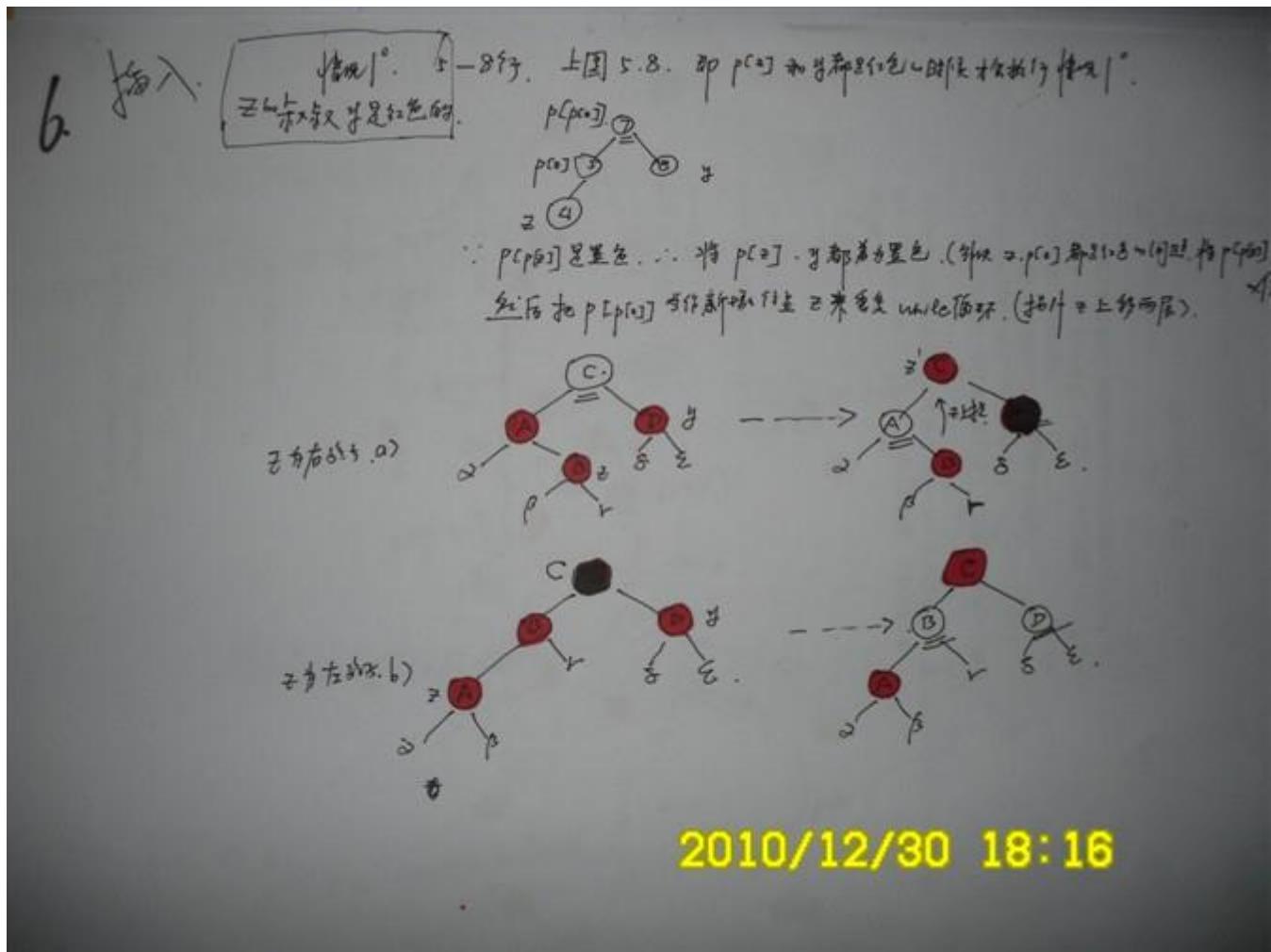
只有 $p[z]$ 和 y (上图 a 中 A 为 $p[z]$, D 为 z , 上图 b 中, B 为 $p[z]$, D 为 y) 都是红色的时候, 才会执行此情况 1.

咱们分析下上图的 a 情况, 即 z 为右孩子时

因为 $p[p[z]]$, 即 c 是黑色, 所以将 $p[z]$ 、 y 都着为黑色 (如上图 a 部分的右边),

此举解决 z 、 $p[z]$ 都是红色的问题, 将 $p[p[z]]$ 着为红色, 则保持了性质 5.

ok, 看下我昨天画的图 (第 6 张):



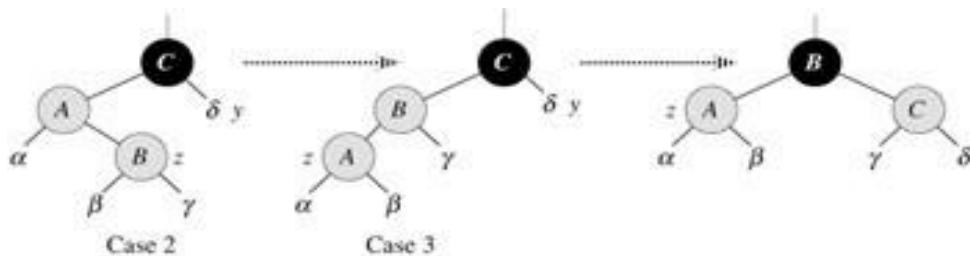
红黑树插入的第一种情况完。

七、红黑树插入的第二种、第三种情况

插入情况 2: z 的叔叔 y 是黑色的, 且 z 是右孩子

插入情况 3: z 的叔叔 y 是黑色的, 且 z 是左孩子

这两种情况, 是通过 z 是 p[z]的左孩子, 还是右孩子区别的。



参照上图, 针对情况 2, z 是她父亲的右孩子, 则为了保持红黑性质, 左旋则变为情况 3, 此时 z 为左孩子,

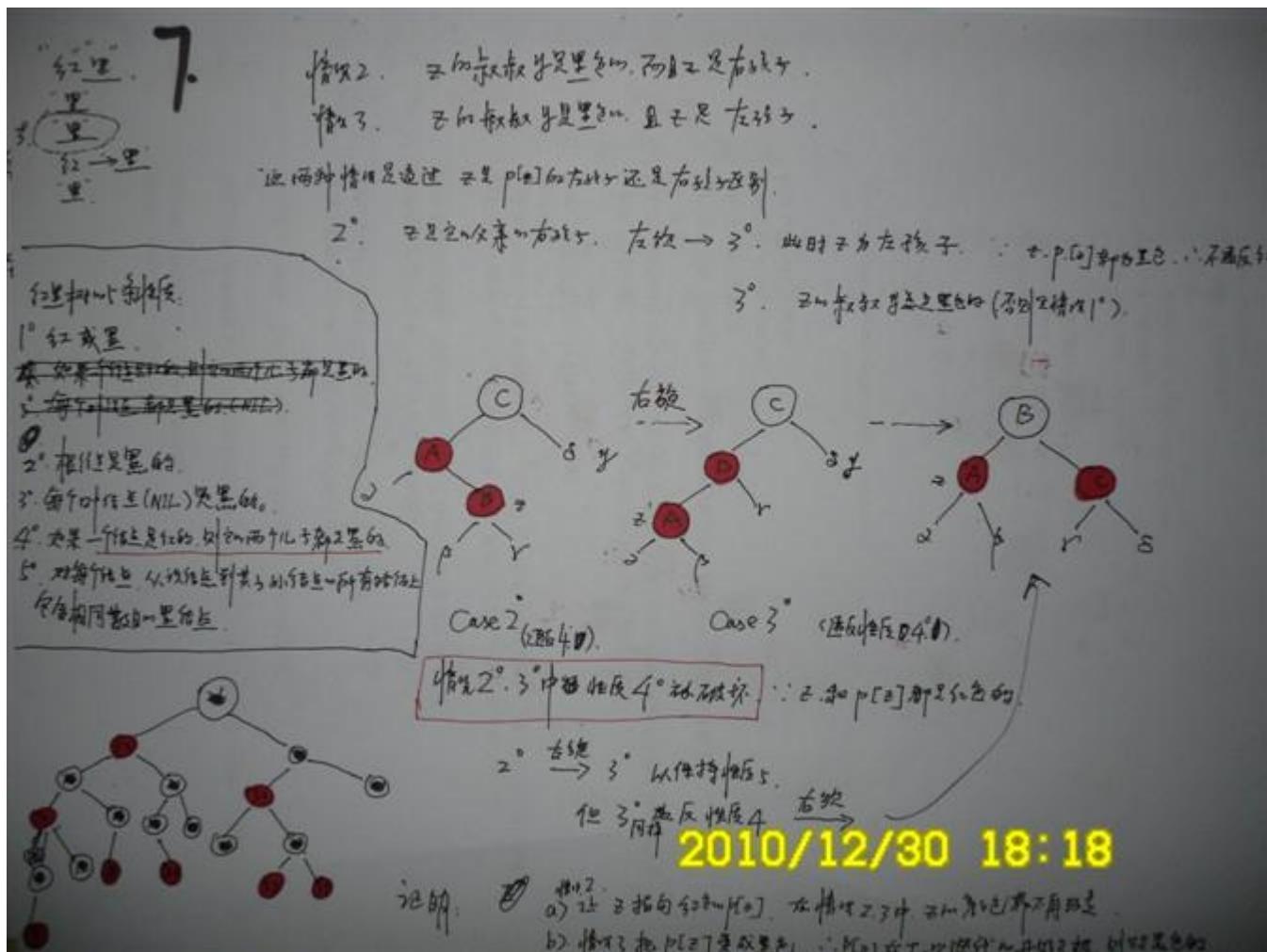
因为 z、p[z]都为黑色, 所以不违反红黑性质 (注, 情况 3 中, z 的叔叔 y 是黑色的, 否则此种情况就变成上述情况 1 了)。

ok, 我们已经看出来了, 情况 2, 情况 3 都违反性质 4 (一个红结点的俩个儿子都是黑

色的)。

所以情况 2->左旋后->情况 3，此时情况 3 同样违反性质 4，所以情况 3->右旋，得到上图的最后那部分。

注：情况 2、3 都只违反性质 4，其它的性质 1、2、3、5 都不违背。
好的，最后，看下我画的图（第 7 张）：



八、接下来，进入红黑树的删除部分。

RB-DELETE(T, z)

- 1 if left[z] = nil[T] or right[z] = nil[T]
- 2 then y \leftarrow z
- 3 else y \leftarrow TREE-SUCCESSOR(z)
- 4 if left[y] \neq nil[T]
- 5 then x \leftarrow left[y]
- 6 else x \leftarrow right[y]
- 7 p[x] \leftarrow p[y]
- 8 if p[y] = nil[T]
- 9 then root[T] \leftarrow x
- 10 else if y = left[p[y]]

```

11      then left[p[y]] ← x
12      else right[p[y]] ← x
13 if y 3≠z
14  then key[z] ← key[y]
15      copy y's satellite data into z
16 if color[y] = BLACK          //如果 y 是黑色的,
17  then RB-DELETE-FIXUP(T, x) //则调用 RB-DELETE-FIXUP(T, x)
18 return y           //如果 y 不是黑色, 是红色的, 则当 y 被删除时, 红黑性质仍然得以保持。
不做操作, 返回。
//因为: 1.树种各结点的黑高度都没有变化。2.不存在俩个相邻的红色结点。
//3.因为入宫 y 是红色的, 就不可能是根。所以, 根仍然是黑色的。
ok, 第 8 张图, 不必贴了。

```

九、红黑树删除之 4 种情况, **RB-DELETE-FIXUP(T, x)**之代码

```

RB-DELETE-FIXUP(T, x)
1 while x ≠ root[T] and color[x] = BLACK
2   do if x = left[p[x]]
3     then w ← right[p[x]]
4       if color[w] = RED
5         then color[w] ← BLACK          ▷ Case 1
6           color[p[x]] ← RED          ▷ Case 1
7           LEFT-ROTATE(T, p[x])      ▷ Case 1
8           w ← right[p[x]]          ▷ Case 1
9       if color[left[w]] = BLACK and color[right[w]] = BLACK
10      then color[w] ← RED          ▷ Case 2
11        x ← p[x]                  ▷ Case 2
12      else if color[right[w]] = BLACK
13        then color[left[w]] ← BLACK    ▷ Case 3
14          color[w] ← RED          ▷ Case 3
15          RIGHT-ROTATE(T, w)      ▷ Case 3
16          w ← right[p[x]]          ▷ Case 3
17          color[w] ← color[p[x]]    ▷ Case 4
18          color[p[x]] ← BLACK      ▷ Case 4
19          color[right[w]] ← BLACK    ▷ Case 4
20          LEFT-ROTATE(T, p[x])      ▷ Case 4
21          x ← root[T]            ▷ Case 4
22      else (same as then clause with "right" and "left" exchanged)
23 color[x] ← BLACK

```

ok, 很清楚, 在此, 就不贴第 9 张图了。

在下文的红黑树删除的 4 种情况, 详细、具体分析了上段代码。

十、红黑树删除的 4 种情况

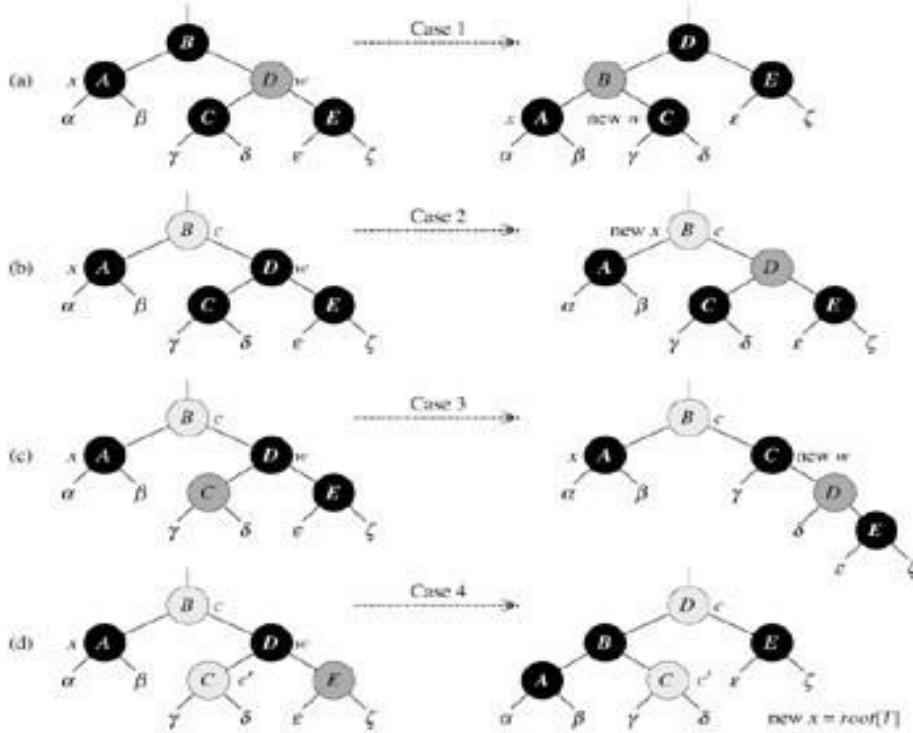
情况 1: x 的兄弟 w 是红色的。

情况 2: x 的兄弟 w 是黑色的, 且 w 的俩个孩子都是黑色的。

情况 3: x 的兄弟 w 是黑色的, w 的左孩子是红色, w 的右孩子是黑色。

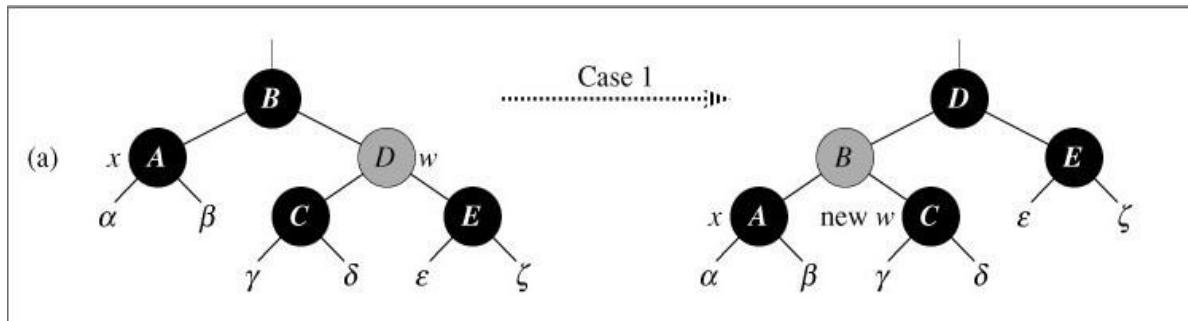
情况 4: x 的兄弟 w 是黑色的, 且 w 的两个孩子都是红色的。

操作流程图:



ok, 简单分析下, 红黑树删除的 4 种情况:

针对情况 1: x 的兄弟 w 是红色的。



```

5      then color[w] ← BLACK           ▷ Case 1
6      color[p[x]] ← RED             ▷ Case 1
7      LEFT-ROTATE(T, p[x])         ▷ Case 1
8      w ← right[p[x]]             ▷ Case 1

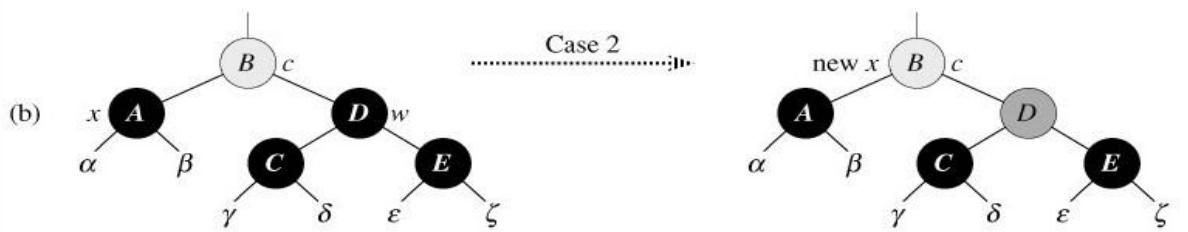
```

对策: 改变 w 、 $p[z]$ 颜色, 再对 $p[x]$ 做一次左旋, 红黑性质得以继续保持。

x 的新兄弟 $\text{new } w$ 是旋转之前 w 的某个孩子, 为黑色。

所以, 情况 1 转化成情况 2 或 3、4。

针对情况 2: x 的兄弟 w 是黑色的, 且 w 的两个孩子都是红色的。



```

10      then color[w] ← RED           ▷ Case 2
11      x <- p[x]                   ▷ Case 2

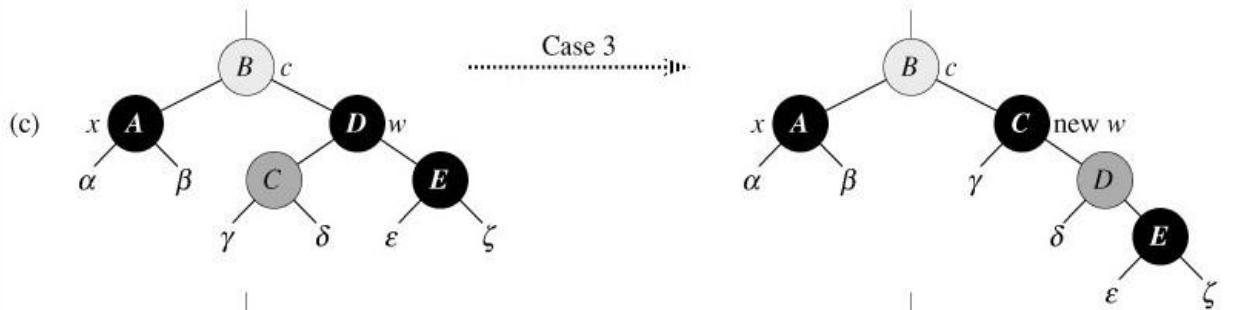
```

如图所示, w 的俩个孩子都是黑色的,

对策: 因为 w 也是黑色的, 所以 x 和 w 中得去掉一黑色, 最后, w 变为红。

$p[x]$ 为新结点 x, 赋给 x, $x <- p[x]$ 。

针对情况 3: x 的兄弟 w 是黑色的, w 的左孩子是红色, w 的右孩子是黑色。



```

13      then color[left[w]] ← BLACK    ▷ Case 3
14          color[w] ← RED            ▷ Case 3
15          RIGHT-ROTATE(T, w)       ▷ Case 3
16          w ← right[p[x]]         ▷ Case 3

```

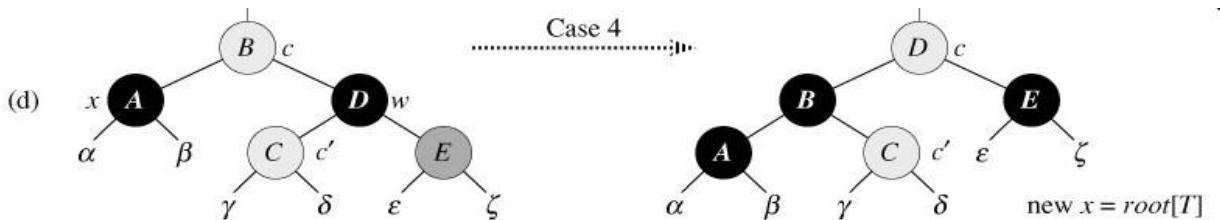
w 为黑, 其左孩子为红, 右孩子为黑

对策: 交换 w 和其左孩子 left[w] 的颜色。即上图的 D、C 颜色互换。:D。

并对 w 进行右旋, 而红黑性质仍然得以保持。

现在 x 的新兄弟 w 是一个有红色右孩子的黑结点, 于是将情况 3 转化为情况 4.

针对情况 4: x 的兄弟 w 是黑色的, 且 w 的右孩子时红色的。



```

17     color[w] ← color[p[x]]           ▷ Case 4
18     color[p[x]] ← BLACK             ▷ Case 4
19     color[right[w]] ← BLACK         ▷ Case 4
20     LEFT-ROTATE(T, p[x])          ▷ Case 4
21     x ← root[T]                  ▷ Case 4

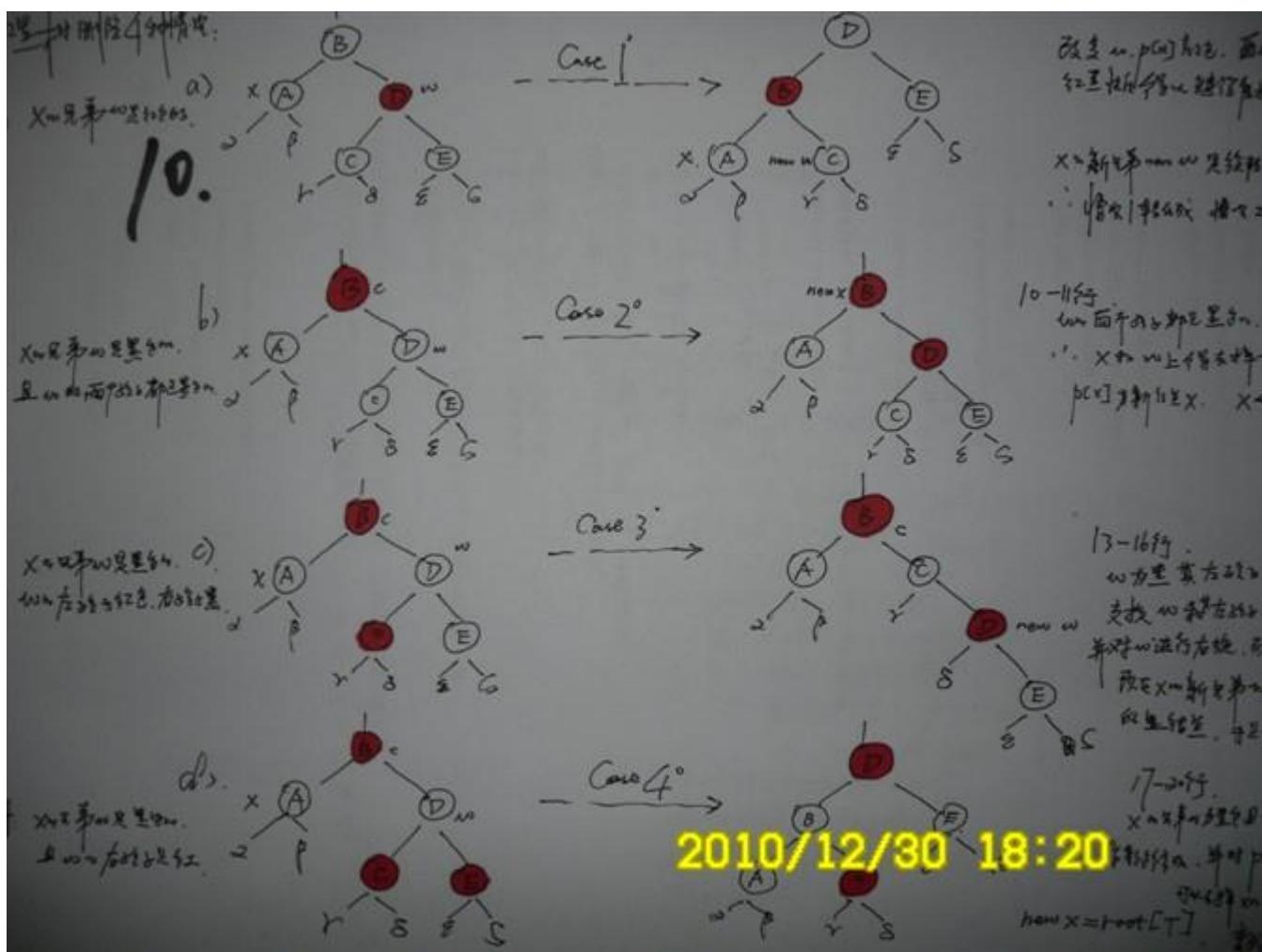
```

x 的兄弟 **w** 为黑色，且 **w** 的右孩子为红色。

对策：做颜色修改，并对 **p[x]** 做一次旋转，可以去掉 **x** 的额外黑色，来把 **x** 变成单独的黑色，此举不破坏红黑性质。

将 **x** 置为根后，循环结束。

最后，贴上最后的第 10 张图：



ok，红黑树删除的 4 中情况，分析完成。

结语：只要牢牢抓住红黑树的 5 个性质不放，而不论是树的左旋还是右旋，不论是红黑树的插入、还是删除，都只为了保持和修复红黑树的 5 个性质而已。

顺祝各位，元旦快乐。完。

July、二零一零年十二月三十日。

扩展阅读：[Left-Leaning Red-Black Trees](#), Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008.

直接下载：<http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>

- 1、教你透彻了解红黑树
 - 2、红黑树算法的实现与剖析
 - 3、红黑树的 c 源码实现与剖析
 - 4、一步一图一代码，R-B Tree
 - 5、红黑树插入和删除结点的全程演示
 - 6、红黑树的 c++ 完整实现源码
-

版权声明

本 BLOG 内的此红黑树系列，总计六篇文章，是整个国内有史以来有关红黑树的最具代表性，最具完整性，最具参考价值的资料。且，本人对此红黑树系列全部文章，享有版权，任何人，任何组织，任何出版社不得侵犯本人版权相关利益，违者追究法律责任。谢谢。

六、教你初步了解 KMP 算法

作者： July、saturnma、上善若水。 时间： 二零一一年一月一日

本文参考：数据结构（c 语言版） 李云清等编著、算法导论

引言：

在文本编辑中，我们经常要在一段文本中某个特定的位置找出 某个特定的字符或模式。

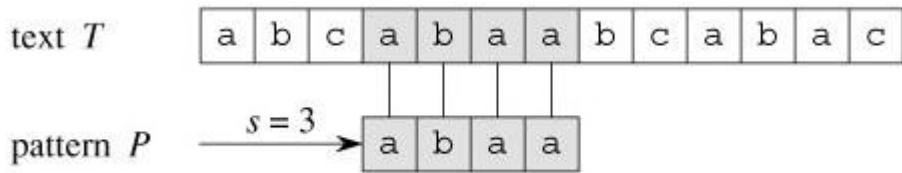
由此，便产生了字符串的匹配问题。

本文由简单的字符串匹配算法开始，再到 KMP 算法，由浅入深，教你从头到尾彻底理解 KMP 算法。

来看算法导论一书上关于此字符串问题的定义：

假设文本是一个长度为 n 的数组 $T[1\dots n]$, 模式是一个长度为 $m \leq n$ 的数组 $P[1\dots m]$ 。

进一步假设 P 和 T 的元素都是属于有限字母表 Σ 中的字符。



依据上图，再来解释下字符串匹配问题。目标是找出所有在文本 $T=abcabaabcaabac$ 中的模式 $P=abaa$ 所有出现。

该模式仅在文本中出现了一次，在位移 $s=3$ 处。位移 $s=3$ 是有效位移。

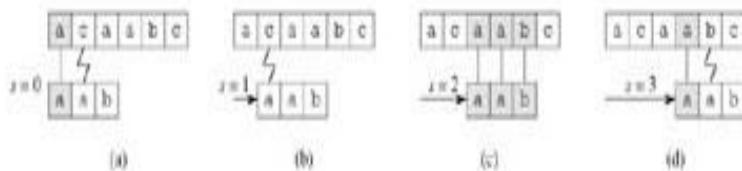
第一节、简单的字符串匹配算法

简单的字符串匹配算法用一个循环来找出所有有效位移，

该循环对 $n-m+1$ 个可能的每一个 s 值检查条件 $P[1\dots m] = T[s+1\dots s+m]$ 。

NAIVE-STRING-MATCHER(T, P)

```
1  $n \leftarrow \text{length}[T]$ 
2  $m \leftarrow \text{length}[P]$ 
3 for  $s \leftarrow 0$  to  $n - m$ 
4   do if  $P[1 \dots m] = T[s + 1 \dots s + m]$ 
      //对  $n-m+1$  个可能的位移  $s$  中的每一个值，比较相应的字符的循环必须执行  $m$  次。
5     then print "Pattern occurs with shift"  $s$ 
```



简单字符串匹配算法，上图针对文本 $T=acaabc$ 和模式 $P=aab$ 。

上述第 4 行代码， $n-m+1$ 个可能的位移 s 中的每一个值，比较相应的字符的循环必须执行

m 次。

所以，在最坏情况下，此简单模式匹配算法的运行时间为 $O((n-m+1)m)$ 。

下面我再来举个具体例子，并给出一具体运行程序：

对于目的字串 target 是 banananobano,要匹配的字串 pattern 是 nano,的情况，

下面是匹配过程，原理很简单，只要先和 target 字串的第一个字符比较，如果相同就比较下一个，如果不同就把 pattern 右移一下，之后再从 pattern 的每一个字符比较，这个算法的运行过程如下图。

//index 表示的每 n 次匹配的情形。

	0	1	2	3	4	5	6	7	8	9	10	11
	b	a	n	a	n	a	n	o	b	a	n	o
index=0	x											
index=1		x										
index=2			n	a	n	x						
index=3				x								
index=4					n	a	n	o				
index=5						x						
index=6							n	x				
index=7								x				
index=8									x			
index=9										x		
index=10											n	x
index=11												x

```
#include<iostream>
#include<string>
using namespace std;
int match(const string& target,const string& pattern)
{
```

```

int target_length = target.size();
int pattern_length = pattern.size();
int target_index = 0;
int pattern_index = 0;
while(target_index < target_length && pattern_index < pattern_length)
{
    if(target[target_index]==pattern[pattern_index])
    {
        ++target_index;
        ++pattern_index;
    }
    else
    {
        target_index -= (pattern_index-1);
        pattern_index = 0;
    }
}
if(pattern_index == pattern_length)
{
    return target_index - pattern_length;
}
else
{
    return -1;
}
}

int main()
{
    cout<<match("banananobano","nano")<<endl;
    return 0;
}

```

//运行结果为 4。

上面的算法时间复杂度是 $O(\text{pattern_length} * \text{target_length})$,

我们主要把时间浪费在什么地方呢,

观看 `index = 2` 那一步, 我们已经匹配了 3 个字符, 而第 4 个字符是不匹配的, 这时我们已经匹配的字符序列是 `nan`,

此时如果向右移动一位, 那么 `nan` 最先匹配的字符序列将是 `an`, 这肯定是不能匹配的,
之后再右移一位, 匹配的是 `nan` 最先匹配的序列是 `n`, 这是可以匹配的。

如果我们事先知道 `pattern` 本身的这些信息就不用每次匹配失败后都把 `target_index` 回退回去,

这种回退就浪费了很多不必要的时间, 如果能事先计算出 `pattern` 本身的这些性质,

那么就可以在失配时直接把 `pattern` 移动到下一个可能的位置,

把其中根本不可能匹配的过程省略掉,

如上表所示我们在 `index=2` 时失配, 此时就可以直接把 `pattern` 移动到 `index=4` 的状态,
`kmp` 算法就是从此出发。

第二节、KMP 算法

2.1、覆盖函数(overlay_function)

覆盖函数所表征的是 `pattern` 本身的性质, 可以让为其表征的是 `pattern` 从左开始的所有连续子串的自我覆盖程度。

比如如下的字串, `abaabcbaba`

子串	值
a	-1
ab	-1
aba	0
abaa	0
abaab	1
abaabc	-1
abaabca	0
abaabcab	1
abaabcbaba	2

由于计数是从 0 始的，因此覆盖函数的值为 0 说明有 1 个匹配，对于从 0 还是从来开始计数是偏好问题，

具体请自行调整，其中 -1 表示没有覆盖，那么何为覆盖呢，下面比较数学的来看一下定义，比如对于序列

$a_0 a_1 \dots a_{j-1} a_j$

要找到一个 k ,使它满足

$a_0 a_1 \dots a_{k-1} a_k = a_{j-k} a_{j-k+1} \dots a_{j-1} a_j$

而没有更大的 k 满足这个条件，就是说要找到尽可能大 k ,使 pattern 前 k 字符与后 k 字符相匹配， k 要尽可能的大，

原因是如果有比较大的 k 存在，而我们选择较小的满足条件的 k ，

那么当失配时，我们就会使 pattern 向右移动的位置变大，而较少的移动位置是存在匹配的，这样我们就会把可能匹配的结果丢失。

比如下面的序列，

target	a	a	b	c	a	a	x	d	n	f	d
pattern	a	a	b	c	a	a	m				
$k=1$					a	a	b	c	a	a	
$k=0$						a	a	b	c	a	a

在红色部分失配，正确的结果是 $k=1$ 的情况，把 pattern 右移 4 位，如果选择 $k=0$,右移 5 位则会产生错误。

计算这个 overlay 函数的方法可以采用递推，可以想象如果对于 pattern 的前 j 个字符，如果覆盖函数值为 k

$a_0 a_1 \dots a_{k-1} a_k = a_{j-k} a_{j-k+1} \dots a_{j-1} a_j$

则对于 pattern 的前 $j+1$ 序列字符，则有如下可能

- (1) $\text{pattern}[k+1] == \text{pattern}[j+1]$ 此时 $\text{overlay}(j+1) = k+1 = \text{overlay}(j)+1$
- (2) $\text{pattern}[k+1] \neq \text{pattern}[j+1]$ 此时只能在 pattern 前 $k+1$ 个子符组所的子串中找到相应的

overlay 函数, $h=overlay(k)$,如果此时 $pattern[h+1]==pattern[j+1]$,则 $overlay(j+1)=h+1$ 否则重复(2)过程.

下面给出一段计算覆盖函数的代码:

```
#include<iostream>
#include<string>
using namespace std;
void compute_overlay(const string& pattern)
{
    const int pattern_length = pattern.size();
    int *overlay_function = new int[pattern_length];
    int index;
    overlay_function[0] = -1;
    for(int i=1;i<pattern_length;++i)
    {
        index = overlay_function[i-1];
        //store previous fail position k to index;

        while(index>=0 && pattern[i]!=pattern[index+1])
        {
            index = overlay_function[index];
        }
        if(pattern[i]==pattern[index+1])
        {
            overlay_function[i] = index + 1;
        }
        else
        {
            overlay_function[i] = -1;
        }
    }
    for(i=0;i<pattern_length;++i)
    {
        cout<<overlay_function[i]<<endl;
    }
}
```

```
    delete[] overlay_function;
}

int main()
{
    string pattern = "abaabcbaba";
    compute_overlay(pattern);
    return 0;
}
```

运行结果为：

```
-1
-1
0
0
1
-1
0
1
2
Press any key to continue
```

2.2、kmp 算法

有了覆盖函数，那么实现 kmp 算法就是很简单的了，我们的原则还是从左向右匹配，但是当失配发生时，我们不用把 `target_index` 向回移动，`target_index` 前面已经匹配过的部分在 `pattern` 自身就能体现出来，只要动 `pattern_index` 就可以了。

当发生在 j 长度失配时，只要把 `pattern` 向右移动 j -`overlay(j)` 长度就可以了。

如果失配时 `pattern_index==0`, 相当于 `pattern` 第一个字符就不匹配, 这时就应该把 `target_index` 加 1, 向右移动 1 位就可以了。

ok, 下图就是 KMP 算法的过程 (红色即是采用 KMP 算法的执行过程) :

下面是kmp匹配过程图

	目标字串annbcdanacadsannannabnna 匹配模式annacanna																				
移动序列	a	n	n	b	c	d	a	n	a	c	a	d	s	a	n	n	a	c	a	n	n
index=0	a	n	n	a	c	a	n	n	a												
index=1		a	n	n	a	c	a	n	n	a											
index=2			a	n	n	a	c	a	n	n	a										
index=3				a	n	n	a	c	a	n	n	a									
index=4					a	n	n	a	c	a	n	n	a								
index=5						a	n	n	a	c	a	n	n	a							
index=6							a	n	n	a	c	a	n	n	a						
index=7								a	n	n	a	c	a	n	n	a					
index=8									a	n	n	a	c	a	n	n	a				
index=9										a	n	n	a	c	a	n	n	a			
index=10											a	n	n	a	c	a	n	n	a		
index=11												a	n	n	a	c	a	n	n	a	
index=12													a	n	n	a	c	a	n	n	
index=13														a	n	n	a	c	a	n	
index=14															a	n	n	a	c	a	
index=15																a	n	n	a	c	
index=16																	a	n	n	a	

图中黄色标注的行为通过 `overlay` 函数跳过的行, 绿色为匹配序列, 红色为失配位置。

另一作者 saturnman 发现, 在上述 KMP 匹配过程图中, `index=8` 和 `index=11` 处画错了。还有, anaven 也早已发现, `index=3` 处也画错了。非常感谢。但图已无法修改, 见谅。

KMP 算法可在 $O(n+m)$ 时间内完成全部的串的模式匹配工作。

ok，最后给出 KMP 算法实现的 c++ 代码：

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

int kmp_find(const string& target,const string& pattern)
{
    const int target_length = target.size();
    const int pattern_length = pattern.size();
    int * overlay_value = new int[pattern_length];
    overlay_value[0] = -1;
    int index = 0;
    for(int i=1;i<pattern_length;++i)
    {
        index = overlay_value[i-1];
        while(index>=0 && pattern[index+1]!=pattern[i])
        {
            index = overlay_value[index];
        }
        if(pattern[index+1]==pattern[i])
        {
            overlay_value[i] = index +1;
        }
        else
        {
            overlay_value[i] = -1;
        }
    }
    //match algorithm start
```

```

int pattern_index = 0;
int target_index = 0;
while(pattern_index<pattern_length&&target_index<target_length)
{
    if(target[target_index]==pattern[pattern_index])
    {
        ++target_index;
        ++pattern_index;
    }
    else if(pattern_index==0)
    {
        ++target_index;
    }
    else
    {
        pattern_index = overlay_value[pattern_index-1]+1;
    }
}
if(pattern_index==pattern_length)
{
    return target_index-pattern_index;
}
else
{
    return -1;
}
delete [] overlay_value;
}

int main()
{
    string source = " annbcdanacadsannannabnna";
    string pattern = " annacanna";
    cout<<kmp_find(source,pattern)<<endl;
    return 0;
}

```

```
}
```

//运行结果为 -1.

第三节、kmp 算法的来源

kmp 如此精巧，那么它是怎么来的呢，为什么要三个人合力才能想出来。其实就算没有 kmp 算法，人们在字符匹配中也能找到相同高效的算法。这种算法，最终相当于 kmp 算法，只是这种算法的出发点不是覆盖函数，不是直接从匹配的内在原理出发，而使用此方法的计算的覆盖函数过程复杂且不易被理解，但是一旦找到这个覆盖函数，那以后使用同一 pattern 匹配时的效率就和 kmp 一样了，其实这种算法找到的函数不应叫做覆盖函数，因为在寻找过程中根本没有考虑是否覆盖的问题。

说了这么半天那么这种方法是什么呢，这种方法是就大名鼎鼎的确定的有限自动机 (Deterministic finite state automaton DFA), DFA 可识别的文法是 3 型文法，又叫正规文法或是正则文法，既然可以识别正则文法，那么识别确定的字串肯定不是问题(确定字串是正则式的一个子集)。对于如何构造 DFA, 是有一个完整的算法，这里不做介绍了。在识别确定的字串时使用 DFA 实在是大材小用，DFA 可以识别更加通用的正则表达式，而用通用的构建 DFA 的方法来识别确定的字串，那这个 overhead 就显得太大了。

kmp 算法的可贵之处是从字符匹配的问题本身特点出发，巧妙使用覆盖函数这一表征 pattern 自身特点的这一概念来快速直接生成识别字串的 DFA, 因此对于 kmp 这种算法，理解这种算法高中数学就可以了，但是如果想从无到有设计出这种算法是要求有比较深的数学功底的。

第四节、精确字符匹配的常见算法的解析

KMP 算法：

KMP 就是串匹配算法

运用自动机原理

比如说

我们在 S 中找 P

设 P={ababbaaba}

我们将 P 对自己匹配

下面是求的过程:{依次记下匹配失败的那一位}

[2]ababbaaba

```

.....ababbaaba[1]
[3]ababbaaba
.....ababbaaba[1]
[4]ababbaaba
.....ababbaaba[2]
[5]ababbaaba
.....ababbaaba[3]
[6]ababbaaba
.....ababbaaba[1]
[7]ababbaaba
.....ababbaaba[2]
[8]ababbaaba
.....ababbaaba[2]
[9]ababbaaba
.....ababbaaba[3]

```

得到 Next 数组『0,1,1,2,3,1,2,2,3』

主过程:

```

[1]i:=1 j:=1
[2]若(j>m)或(i>n)转[4]否则转[3]
[3]若 j=0 或 a[i]=b[j]则 【inc(i)inc(j)转[2]】 否则 【j:=next[j]转 2】
[4]若 j>m 则 return(i-m)否则 return -1;

```

若返回-1 表示失败, 否则表示在 i-m 处成功

BM 算法也是一种快速串匹配算法, **KMP** 算法的主要区别是匹配操作的方向不同。虽然 T 右移的计算方法却发生了较大的变化。

为方便讨论, $T = "dist : c \rightarrow \{dist\}$ 称为滑动距离函数, 它给出了正文中可能出现的任意字符在模式中的位置。函

数

$m - j \quad j$

为

$(m+1 \quad \text{若 } c = t_m)$

$dist$

例如, pattern " ", 则 p) a) t) dist (= 2, r) n) BM 算法的基本思想是: 假设将主串中自位置 $i + dist(si)$ 位置开始重新进行新一轮的匹配, 其效果相当于把模式和主串向右滑过一段距离 si), 即跳过 si 个字符而无需进行比较。

下面是一个 $S = "T = " BM$ 算法可以大大加快串匹配的速度。

下面是 KMP 算法部分, 把调用 BM 函数便可。

```
1. #include <iostream>
```

```

2. using namespace std;
3.
4. int Dist(char *t,char ch)
5. {
6.     int len = strlen(t);
7.     int i = len - 1;
8.     if(ch == t[i])
9.         return len;
10.    i--;
11.    while(i >= 0)
12.    {
13.        if(ch == t[i])
14.            return len - 1 - i;
15.        else
16.            i--;
17.    }
18.    return len;
19. }
20.
21. int BM(char *s,char *t)
22. {
23.     int n = strlen(s);
24.     int m = strlen(t);
25.     int i = m-1;
26.     int j = m-1;
27.     while(j>=0 && i<n)
28.     {
29.         if(s[i] == t[j])
30.         {
31.             i--;
32.             j--;
33.         }
34.         else
35.         {
36.             i += Dist(t,s[i]);
37.             j = m-1;
38.         }
39.     }
40.     if(j < 0)
41.     {
42.         return i+1;
43.     }
44.     return -1;
45. }

```

Horspool 算法

这个算法是由 R.Nigel Horspool 在 1980 年提出的。其滑动思想非常简单，就是从后往前匹配模式串，若在某一位失去匹配，此位对应的文本串字符为 **c**，那就将模式串向右滑动，使模式串之前最近的 **c** 对准这一位，再从新从后往前检查。那如果之前找不到 **c** 怎么办？那好极了，直接将整个模式串滑过这一位。

例如：

文本串：abdabaca

模式串：baca

倒数第 2 位失去匹配，模式串之前又没有 **d**，那模式串就可以整个滑过，变成这样：

文本串：abdabaca

模式串： baca

发现倒数第 1 位就失去匹配，之前 1 位有 **c**，那就向右滑动 1 位：

文本串：abdabaca

模式串： baca

实现代码：

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <cstdlib>
5. using namespace std;
6.
7. int Horspool_match(const string & S,const string & M,int pos)
8. {
9.     int S_len = S.size();
10.    int M_len = M.size();
11.    int Mi = M_len-1,Si= pos+Mi; //这里的串的第一个元素下标是 0
12.    if( (S_len-pos) < M_len )
13.        return -1;
14.    while ( (Mi>-1) && (Si<S_len) )
15.    {
16.        if (S[Si] == M[Mi])
17.        {
18.            --Mi;
19.            --Si;
20.        }
21.        else
22.        {
23.            do
```

```

24.         {
25.             Mi--;
26.         }
27.         while( (S[Si]!=M[Mi]) || (Mi>-1) );
28.         Mi = M_len - 1;
29.         Si += M_len - 1;
30.     }
31. }
32. if(Si < S_len)
33.     return(Si + 1);
34. else
35.     return -1;
36. }
37.
38. int main( )
39. {
40.     string S="abcdefgabcdefghhijklmabc";
41.     string T="hhij";
42.     int    pos = Horspool_match(S,T,3);
43.
44.     cout<<"/n"<<pos<<endl;
45.     system("pause");
46.     return 0;
47. }
```

SUNDAY 算法:

BM 算法的改进的算法 SUNDAY--Boyer-Moore-Horspool-Sunday Aglorithm

BM 算法优于 KMP

SUNDAY 算法描述:

字符串查找算法中，最著名的两个是 KMP 算法 (Knuth-Morris-Pratt)和 BM 算法 (Boyer-Moore)。两个算法在最坏情况下均具有线性的查找时间。但是在实用上，KMP 算法并不比最简单的 c 库函数 strstr()快多少，而 BM 算法则往往比 KMP 算法快上 3—5 倍。但是 BM 算法还不是最快的算法，这里介绍一种比 BM 算法更快一些的查找算法即 Sunday 算法。

例如我们要在"substring searching algorithm"查找"search"，刚开始时，把子串与文本左边对齐：

substring searching algorithm

search

^

结果在第二个字符处发现不匹配，于是要把子串往后移动。但是该移动多少呢？这就是各种算法各显神通的地方了，最简单的做法是移动一个字符位置；**KMP** 是利用已经匹配部分的信息来移动；**BM 算法是做反向比较，并根据已经匹配的部分来确定移动量**。这里要介绍的方法是看紧跟在当前子串之后的那个字符（上图中的 'i'）。

显然，不管移动多少，这个字符是肯定要参加下一步的比较的，也就是说，如果下一步匹配到了，这个字符必须在子串内。所以，可以移动子串，使子串中的最右边的这个字符与它对齐。现在子串'search'中并不存在'i'，则说明可以直接跳过一大片，从'i'之后的那个字符开始作下一步的比较，如下图：

substring searching algorithm

search

^

比较的结果，第一个字符就不匹配，再看子串后面的那个字符，是'r'，它在子串中出现在倒数第三位，于是把子串向前移动三位，使两个'r'对齐，如下：

substring searching algorithm

search

^

哈！这次匹配成功了！回顾整个过程，我们只移动了两次子串就找到了匹配位置，可以证明，用这个算法，每一步的移动量都比 BM 算法要大，所以肯定比 BM 算法更快。

```
1. #include<iostream>
2. #include<fstream>
3. #include<vector>
4. #include<algorithm>
5. #include<string>
6. #include<list>
7. #include<functional>
8.
9. using namespace std;
10.
11. int main()
12. {
13.     char *text=new char[100];
```

```

14.     text="substring searching algorithm search";
15.     char *patt=new char[10];
16.     patt="search";
17.     size_t temp[256];
18.     size_t *shift=temp;
19.
20.     size_t patt_size=strlen(patt);
21.     cout<<"size : "<<patt_size<<endl;
22.     for(size_t i=0;i<256;i++)
23.         *(shift+i)=patt_size+1;//所有值赋于 7, 对这题而言
24.
25.     for(i=0;i<patt_size;i++)
26.         *(shift+unsigned char(*(patt+i) ) )=patt_size-i;
27.         /* //      移动 3 步-->shift['r']=6-3=3;移动三步
28.          //shift['s']=6 步,shift['e']=5 以此类推
29. */
30.
31.     size_t text_size=strlen(text);
32.     size_t limit=text_size-i+1;
33.
34.     for(i=0;i<limit;i+=shift[text[i+patt_size] ] )
35.         if(text[i]==*patt)
36.         {
37.             /*      ^13--这个 r 是位, 从 0 开始算
38.                substring searching algorithm
39.                search
40.                searching-->这个 s 为第 10 位, 从 0 开始算
41.                如果第一个字节匹配, 那么继续匹配剩下的
42.             */
43.
44.             char* match_text=text+i+1;
45.             size_t     match_size=1;
46.             do{
47.                 if(match_size==patt_size)
48.
49.                     cout<<"the no is "<<i<<endl;
50.                     }while( (*match_text++)==patt[match_size++] );
51.             }
52.
53.             cout<<endl;
54.         }
55.         delete []text;
56.         delete []patt;
57.         return 0;

```

```
58. }
59.
60. //运行结果如下:
61. /*
62. size : 6
63. the no is 10
64. the no is 30
65. Press any key to continue
66. */
```

本文完。

版权所有。转载本 BLOG 内任何文章，请以超链接形式注明出处。
否则，一经发现，必定永久谴责+追究法律责任。谢谢，各位。

六之续、由 KMP 算法谈到 BM 算法

作者：滨湖，July、yansha。

说明：初稿由滨湖提供，July 负责 KMP 部分的勘误，yansha 负责 BM 部分的修改。全文由 July 统稿修订完成。

出处：http://blog.csdn.net/v_JULY_v。

引言

在此之前，说明下写作本文的目的：1、之前承诺过，这篇文章**六、教你从头到尾彻底理解 KMP 算法、updated**之后，KMP 算法会写一个续集；2、写这个 KMP 算法的文章很多很多，但真正能把它写明白的少之又少；3、这个 KMP 算法曾经困扰过我很长一段时间。我也必须让读者真真正正彻彻底底的理解它。希望，我能做到。

ok，子串的定位操作通常称做串的**模式匹配**，是各种串处理系统中最重要的操作之一。在很多应用中都会涉及子串的定位问题，如普通的字符串查找问题。如果我们把模式匹配的串看成一字节流的话，那应用空间一下子就广阔了很多，如 HTTP 协议里就是字节流，有各种关键的字节流字段，对 HTTP 数据进行解释就需要用到模式匹配算法。

本文是试图清楚的讲解模式匹配算法里两个最为重要的算法：KMP 与 BM 算法，这两个算法都较为高效，特别是 BM 算法在工程用应用得非常多的，然而网上很多 BM 算法都不算准确的。本文开始讲解简单回溯字符串匹配算法，后面过渡到 KMP 算法，最后再过渡到 BM 算法，希望能够讲得明白易懂。

模式匹配问题抽象为：给定主串 S (Source, 长度为 n)，模式串 P (Pattern, 长度为 m)，要求查找出 P 在 S 中出现的位置，一般即为第一次出现的位置，如果 S 中没有 P 子串，返回相应的结果。如下图 0 查找成功，则查找结果返回 2：

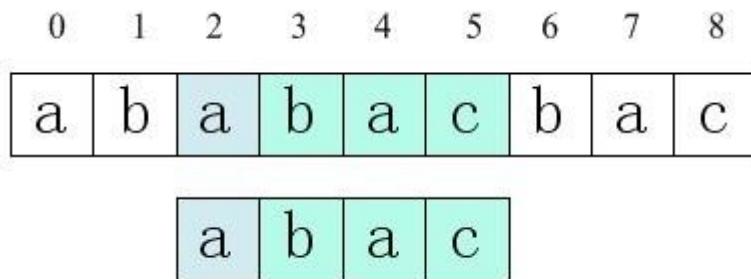


图 0 字符串查找

本文，接下来，将一步一步讲解 KMP 算法。希望看完本文后，读者日后对 Kmp 算法能做到胸中丘壑自成。文章有任何错误，烦请一定指出来。谢谢。

第一部分、KMP 算法

• 1、回溯法字符串匹配算法

回溯法字符串匹配算法就是用一个循环来找出所有有效位移，该循环对 $n-m+1$ 个可能的位移中的每一个 $index$ 值，检查条件为 $P[0...m-1] = S[index...index+m-1]$ (因为模式串的长度是 m ，索引范围为 $0...m-1$)。

S 0..... $index$ $index+m-1$ ($src[i]$ 表示)
 P 0 $m-1$ ($patn[j]$ 表示)

```
1. //代码 1-1
2. //int search(char const*, int, char const*, int)
3. //查找出模式串 patn 在主串 src 中第一次出现的位置
4. //plen 为模式串的长度
5. //返回 patn 在 src 中出现的位置，当 src 中并没有 patn 时，返回-1
6. int search(char const* src, int slen, char const* patn, int plen)
```

```

7.  {
8.      int i = 0, j = 0;
9.      while( i < slen && j < plen )
10.     {
11.         if( src[i] == patn[j] ) //如果相同，则两者++, 继续比较
12.         {
13.             ++i;
14.             ++j;
15.         }
16.         else
17.         {
18.             //否则，指针回溯，重新开始匹配
19.             i = i - j + 1; //退回到最开始时比较的位置
20.             j = 0;
21.         }
22.     }
23.     if( j >= plen )
24.         return i - plen; //如果字符串相同的长度大于模式串的长度，则匹配成功
25.     else
26.         return -1;
27. }
```

该算法思维比较简单（但也常被一些公司做为面试题），很容易分析出本算法的时间复杂度为 $O(pattern_length * target_length)$ ，我们主要是把时间浪费在什么地方呢，相信，你已经看到上面的代码注释中有这么一句话：“指针回溯，重新开始匹配”，这句话的意思就是好比我们乘坐一辆火车已经离站好远了，后来火车司机突然对全部乘客说，你们搭错了列车，要换一辆火车。也就是说在咱们的字符串匹配中，本来已经比较到前面的字符去了，现在又要回到原来的某一个位置重新开始一个个的比较。这就是问题的症结所在。

在继续分析之前，咱们来思考这样一个问题：为什么快排或者堆排序比直接的选择排序快？直接的选择排序，每次都是重复的比较数值的大小，每扫描一次，只得出一个最大（小值），再没有其它的结果信息能给下一次扫描带来便捷。我们看看快排，每扫一次，将数据按某一值分成了两边，至少有右边的数据都大于左边的数据，所以在比较的时候，下一次就不用比较了。再看看堆排序，建堆的过程也是 $O(n)$ 的比较，但比较的结果得到了最大（小）堆这种三角关系，之后的比较就不用再每一个都需要比较了。

由上述思考，咱们总结出了一点优化的归律：采用一种简单的数据结构或者方式，将每次重复性的工作得到的信息记录得尽量多，方便下一次做同样的工作，这样将带来一定的优化（个人性总结）。

回溯法做的多余的工作

以下给出一个例子来启发，如下图 2：

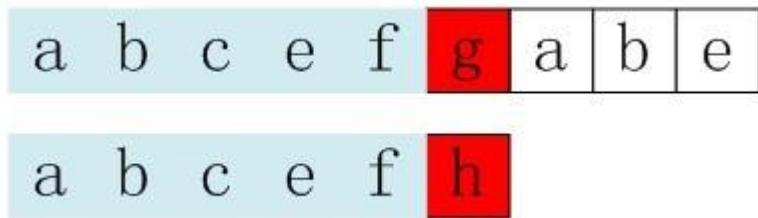


图 1-1 回溯法的一个示例

可以看出当匹配到 **g** 与 **h** 的时候，不匹配了（后面，你将看到，**KMP** 算法会直接从匹配失效的位置，即 **g** 位置处重新开始匹配，这就是 **KMP** 的高效之处），模式串的下一个位置该怎么移动，需要回溯到第二个位置如：

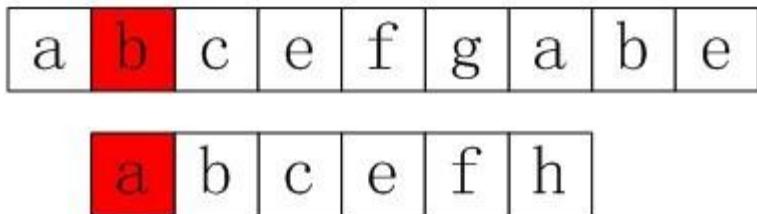


图 1-2 回溯到第二个位置

在第二个位置发现还是不匹配，便再次回溯到第三个位置：

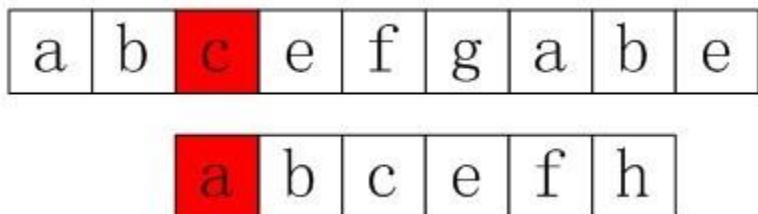


图 1-3 回溯到第三个位置

其实可以分析一下模式串里，每个字符都不相同，如果前面有匹配成功，那移动一位或者几位后，是不可能匹配成功的。

启示：模式串里有蕴含信息的，可以简化扫描。接下来深入的讨论另一算法 **KMP** 算法。

- 2、**KMP** 算法的简介

KMP 算法就是一种基于分析模式串蕴含信息的改进算法，是 D.E.Knuth 与 V.R.Pratt 和 J.H.Morris 同时发现的，因此人们称它为 KMP 算法。

咱们还是以上面的例子为例，如下图 2-1：

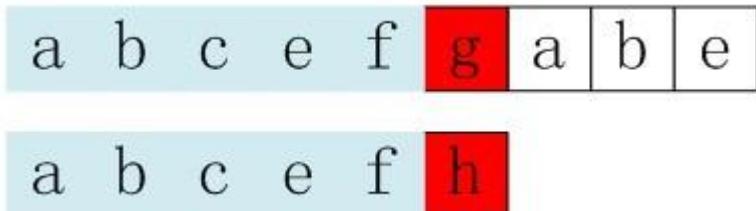


图 2-1 KMP 算法的一个例子

如果是普通的匹配算法，那么接下来，模式串的下一个匹配将如上一节读者所看到的那样，回溯到第二个位置 b 处。而 KMP 算法会怎么做呢？KMP 算法会直接把模式串移到匹配失效的位置上，如下图 2-2，g 处：

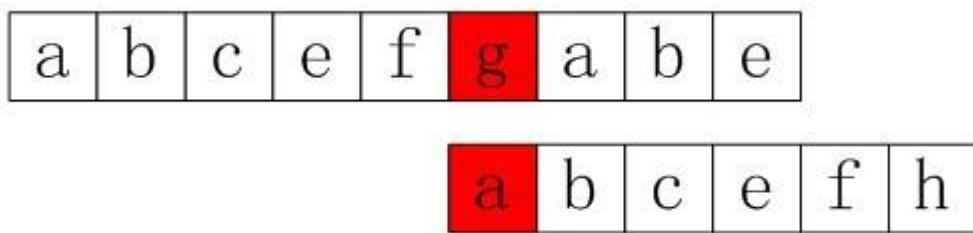


图 2-2 直接移到匹配失效的位置 g 处

Ok，咱们下面再看一个例子，如下图 2-3/4：

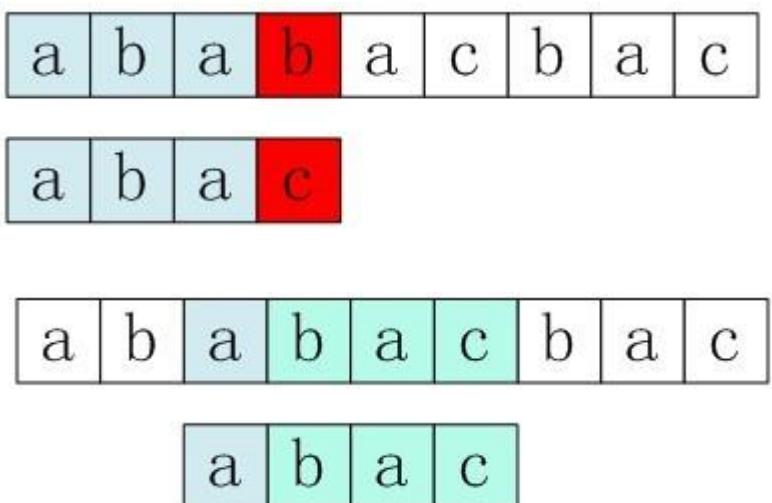


图 2-3/4 另一个例子

我们为什么要这么做呢?如上面的例子,每个字符都不相同,如果前面有匹配成功,那移动一位或者几位后,是不可能匹配成功的,所以我们完全可以就模式串的特点来决定下一次匹配从哪个地方开始。

问题转化成为对于模式串 P ,当 $P[j](0 \leq j < m)$ 与主串匹配到第 i 个字符($S[i], 0 \leq i < n$)失败的时候,接下来应该用什么位置的字符 $P[j_next]$ (我们设 j_next 即匹配失效后下一个匹配的位置)与主串 $S[i]$ 开始匹配呢?重头开始匹配?No,在 $P[j] \neq S[i]$ 之前的时候,有 $S[i-j\dots i-1]$ 与 $P[0\dots j-1]$ 是相同的,所以 S 不用回溯,因为 $S[i]$ 前面的值都已经确切的知道了。

$S \quad 0 \underline{i-j..i-1} \quad i \quad \dots \quad n \quad (S[i] \text{表示, } S[i] \text{处匹配失败})$
 $P \quad \underline{0..j-1} \quad j.. \quad m \quad (P[j] \text{表示, 要找下一个匹配的位置 } P[j_next])$

以上,在 $P[j] \neq S[i]$ 之前的时候,有 $S[i-j\dots i-1]$ 与 $P[0\dots j-1]$ 是匹配即相同的字符,各自都用下划线表示。

咱们先写下算法,你将看到,其实 KMP 算法的代码非常简洁,只有 20 来行而已。如下描述为:

```
1. //代码 2-1
2. //int kmp_search(char const*, int, char const*, int, int const*, int pos) KM
   P 模式匹配函数
3. //输入: src, slen 主串
4. //输入: patn, plen 模式串
5. //输入: nextval KMP 算法中的 next 函数值数组
6. int kmp_search(char const* src, int selen, char const* patn, int plen, int co
   nst* nextval, int pos)
7. {
8.     int i = pos;
9.     int j = 0;
10.    while ( i < selen && j < plen )
11.    {
12.        if( j == -1 || src[i] == patn[j] )
13.        {
14.            ++i;
15.            ++j;    //匹配成功, 就++, 继续比较。
16.        }
17.        else
18.        {
19.            j = nextval[j];
20.            //当在 j 处,P[j]与 S[i]匹配失败的时候直接用 patn[nextval[j]]继续与 S[i]
               比较,
```

```

21.         //所以，Kmp 算法的关键之处就在于怎么求这个值拉，  

22.         //即匹配失效后下一次匹配的位置。下面，具体阐述。  

23.     }  

24. }
25. if( j >= plen )
26.     return i-plen;
27. else
28.     return -1;
29. }

```

• 3、如何求 next 数组各值

现在的问题是 $p[j_next]$ 中的 j_next 即上述代码中的 $nextval[j]$ 怎么求。

当匹配到 $S[i] \neq P[j]$ 的时候有 $S[i-j\dots i-1] = P[0\dots j-1]$ 。如果下面用 j_next 去匹配，则有 $P[0\dots j_next-1] = S[i-j_next\dots i-1] = P[j-j_next\dots j-1]$ 。此过程如下图 3-1 所示。

当匹配到 $S[i] \neq P[j]$ 时， $S[i-j\dots i-1] = P[0\dots j-1]$:

S: 0 ... i-j ... i-1 i ...

P: 0 ... j-1 j ...

如果下面用 j_next 去匹配，则有 $P[0\dots j_next-1] = S[i-j_next\dots i-1] = P[j-j_next\dots j-1]$ 。

所以在 P 中有如下匹配关系（获得这个匹配关系的意义是用来求 next 数组）：

P: 0 ... j-j_next ... j-1 ...

P: 0 ... j_next-1 ...

所以，根据上面两个步骤，推出下一匹配位置 j_next :

S: 0 ... i-j ... i-j_next ... i-1 ... i ...

P: 0 ... j_next-1 ... j_next ...

图 3-1 求 j_next （最大的值）的三个步骤

下面，我们用变量 k 来代表求得的 j_next 的最大值，即 k 表示这 $S[i]、P[j]$ 不匹配时 P 中下一个用来匹配的位置，使得 $P[0\dots k-1] = P[j-k\dots j-1]$ ，而我们要尽量找到这个 k 的最大值。如你所见，当匹配到 $S[i] \neq P[j]$ 的时候，最大的 k 为 1（当 $S[i]$ 与 $P[j]$ 不匹配时，用 $P[k]$ 与 $S[i]$ 匹配，即 $P[1]$ 和 $S[i]$ 匹配，因为 $P[0]=P[2]$ ，所以最大的 $k=1$ ）。

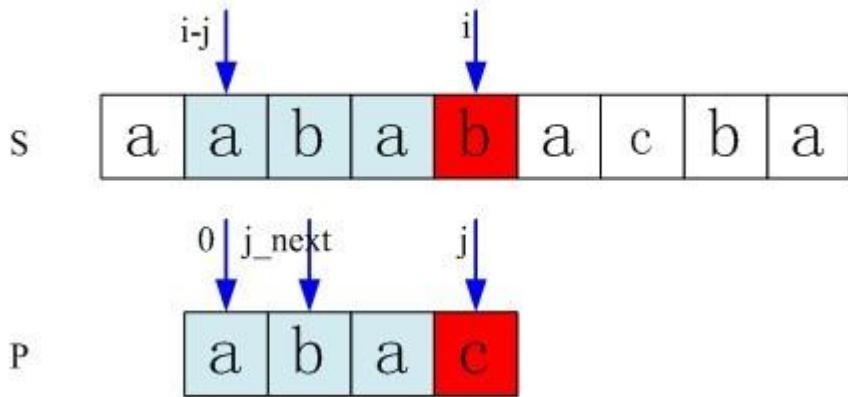


图 3-2 $j_next=1$, 即最大的 k 的值为 1

如上图 3-1, 当 $P[3]=S[i]$, 而 $P[0]=P[2]$ (当 $P[3]=S[i]$, 而 $P[0]=P[2]$, $P[2]=S[i-1]$, 所以肯定有 $P[0]=S[i-1]$) , 所以只需比较 $P[1]$ 与 $S[i]$ 就可以了, 即 k 是 P 可以跳过比较的最大长度, 换句话说, 就是 k 能标示出 $S[i]$ 与 $P[j]$ 不匹配时 P 的下一个匹配的位置。

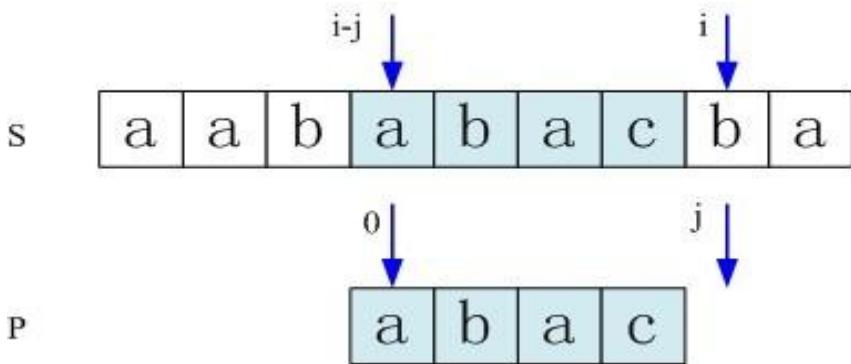


图 3-3 第二步匹配中, 跳过 $P[0]$ (a) , 只需要比较 $P[1]$ 与 $S[3]$ (b) 了

也就是说, 如上图 3-2, 在第一次匹配中, 就是因为 $S[2]=P[0]$, 所以在下一次匹配中, 只需要比较 $S[3]=P[1]$, 跳过了几步?一步。那么 k 等于多少? $k=1$ 。即把 P 右移两个位置后, $P[0]$ 与 $S[2]$ 不必再比较, 因为前一步已经得出他们相等。所以, 此时, 只需要比较 $P[1]$ 与 $S[3]$ 了。

接下来的问题是, 怎么求最大的数 k 使得 $p[0...k-1] = p[j-k...j-1]$ 呢。这就是 KMP 算法中最核心的问题, 即怎么求 $next$ 数组的各元素的值?只有真正弄懂了这个 $next$ 数组的求法, 你才能彻底明白 KMP 算法到底是怎么回事。

那么, 怎么求这个 $next$ 数组呢?咱们一步一步来考虑。

求最大的数 k 使得 $P[0...k-1] = P[j-k...j-1]$, 一个直接的办法是对于 j , 从 $P[j-1]$ 往回查, 看是否有满足 $P[0...k-1] = P[j-k...j-1]$ 的 k 存在, 而且还要最大的一个 k 。下面咱们换一个角度

思考。

当 $P[j+1]$ 与 $S[i+1]$ 不匹配时，分两种情况求 next 数组（注：以下皆有 $k=\text{next}[j]$ ）：

1. $P[j] = p[k]$, 那么 $\text{next}[j+1]=k+1$, 这个很容易理解。采用递推的方式求出 $\text{next}[j+1]=k+1$ (代码 3-1 的 if 部分)。
2. $P[j] \neq p[k]$, 那么 $\text{next}[j+1]=\text{next}[k]+1$ (代码 3-1 的 else 部分)

稍后，你将看到，由这个方法得出的 next 值还不是最优的，也就是说是不能允许 $P[j]=P[\text{next}[j]]$ 出现的。ok，请跟着我们一步一步登上山顶，不要试图一步登天，那是不可能的。由以上，可得如下代码：

```
1. //代码 3-1, 稍后, 你将由下文看到, 此求 next 数组元素值的方法有错误
2. void get_next(char const* ptn, int plen, int* nextval)
3. {
4.     int i = 0;
5.     nextval[i] = -1;
6.     int j = -1;
7.     while( i < plen-1 )
8.     {
9.         if( j == -1 || ptn[i] == ptn[j] )      //循环的 if 部分
10.        {
11.            ++i;
12.            ++j;
13.            nextval[i] = j;
14.        }
15.        else                                //循环的 else 部分
16.            j = nextval[j];                //递推
17.    }
18. }
```

next 数组求值的验证

上述求 next 数组各值的方法(代码)是否正确呢? 我们来举一个例子, 应用上述的 `get_next` 函数来试验一下, 即具体求解一下 next 数组各元素的值 (通过下面的验证, 我们将看到上面的求 next 数组的方法是有问题的, 而后我们会在下文的第 4 小节具体修正上述求 next 数组的方法)。ok, 请看:

首先，模式串如下：字符串 abab 下面对应的数值即是已经求出的对应的 `nextval[i]` 值：

a	b	a	b
-1	0	0	1

图 3-4 求 `next` 数组各值的示例

接下来，咱们来具体解释下上面 `next` 数组中对应的各个 `nextval[i]` 的值是怎么求得来的，因为，理解 KMP 算法的关键就在于这个求 `next` 值的过程。Ok，如下，咱们再次引用一下上述求 `next` 数组各值的核心代码：

```
int i = 0;  
nextval[i] = -1;  
int j = -1;  
while( i < plen-1 )  
{  
    if( j == -1 || ptn[i] == ptn[j] )    //循环的 if 部分  
    {  
        ++i;  
        ++j;  
        nextval[i] = j;  
    }  
    else                    //循环的 else 部分  
        j = nextval[j];      //递推  
}
```

所以，根据上面的代码，咱们首先要初始化 `nextval[0] = -1`，我们得到第一个 `next` 数组元素值即 -1（注意，咱们现在的目标是要求 `nextval[i]` 各个元素的值，`i` 是数组的下标，为 0.1.2.3）；

a	b	a	b
-1			

图 3-5 第一个 `next` 数组元素值-1

首先初始化: $i = 0$, $j = -1$, 由于 $j == -1$, 进入上述函数中循环的 if 部分, $++i$ 得 $i=1$, $++j$ 得 $j=0$, 所以我们得到第二个 next 值即 $\text{nextval}[1] = 0$;

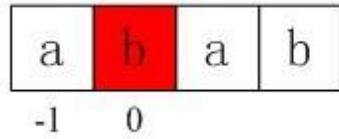


图 3-6 第二个 next 数组元素值 0

$i = 1$, $j = 0$, 由于不满足条件 $j == -1 \parallel \text{ptrn}[i] == \text{ptrn}[j]$ (第一个元素 a 与第二个元素 b 不相同, 所以也不满足第 2 个条件), 所以进入上述循环的 else 部分, 得到 $j = \text{nextval}[j] = -1$ (原来的 $\text{nextval}[0]=-1$ 并没有改变), 得到 $i = 1$, $j = -1$; 此时, 由于 $j == -1$ 且 $i < \text{plen}-1$ 依然成立, 所以再次进入上述循环的 if 部分, $++i$ 的 $i=2$, $++j$ 得 $j=0$, 所以得到第三个 next 值即 $\text{nextval}[2] = 0$;

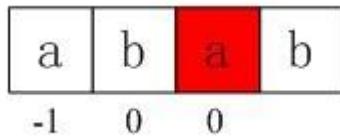


图 3-7 第三个 next 数组元素值 0

此时, $i = 2$, $j = 0$, 由于 $\text{ptrn}[i] == \text{ptrn}[j]$ (第 1 个元素和第 3 个元素都是 a, 相同, 所以, 虽然不满足 $j=-1$ 的第 1 个条件, 但满足第 2 个条件即 $\text{ptrn}[i] == \text{ptrn}[j]$), 进入循环的 if 部分, $++i$ 得 $i=3$, $++j$ 得 $j=1$, 所以得到我们的第四个 next 值即 $\text{nextval}[3] = 1$ (由下文的第 4 小节, 你将看到, 求出的 next 数组之所以有误, 问题就是出在这里。正确的解决办法是, 如下文的第 4 小节所述, $++i$, $++j$ 之后, 还得判断 $\text{patn}[i]$ 与 $\text{patn}[j]$ 是否相等, 即杜绝出现 $P[j]=P[\text{next}[j]]$ 这样的情况);

自此, 我们得到了 $\text{nextval}[i]$ 数组的 4 个元素值, 分别为 **-1, 0, 0, 1**。如下图 3-8 所示:

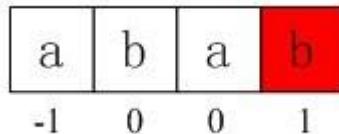


图 3-8 第四个 next 数组元素值 1

求得了相应的 next 数组 (本文约定, next 数组是指一般意义的 next 数组, 而 $\text{nextval}[i]$ 则代表具体求解 next 数组各数值的意义) 各值之后, 接下来的一切工作就好办多了。

第一步：主串和模式串如下，由下图可以看到，我们在 $p[3]$ 处匹配失败（即 $p[3] \neq s[3]$ ）。

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

图 3-9 第一步，在 $p[3]$ 处匹配失败

第二步：接下来要用 $p[next[3]]$ （看到了没，是该我们上面求得的 $next$ 数组各值大显神通的时候了），即 $p[1]$ 与 $s[3]$ 匹配（不要忘了，上面我们已经求得的 $nextval[i]$ 数组的 4 个元素值，分别为 -1, 0, 0, 1）。但在 $p[1]$ 处还是匹配失败（即 $p[1] \neq s[3]$ ）。

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

图 3-10 第二步， $p[1]$ 处还是匹配失败

第三步：接下来模式串指针指向下一位置 $next[1]=0$ 处（注意此过程中主串指针是不动的），即模式串指针指向 $p[0]$ ，即用 $p[0]$ 与 $s[3]$ 匹配（看起来，好像是 k 步步减小，这就是咱们开头所讲到的怎么求最大的数 k 使得 $P[0...k-1] = [j-k...j-1]$ ）。而 $p[0]$ 与 $s[3]$ 还是不匹配。

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

图 3-11 第三步， $p[0]$ 与 $s[3]$ 还是不匹配

第四步：由于上述第三步中， $P[0]$ 与 $S[3]$ 还是不匹配。此时 $i=3, j=nextval[0]=-1$ ，由于满足条件 $j=-1$ ，所以进入循环的 if 部分， $++i=4, ++j=0$ ，即主串指针下移一个位置，从 $p[0]$ 与 $s[4]$ 处开始匹配。

最后 $j==plen$, 跳出循环, 输出结果 $i-plen=4$ (即字串第一次出现的位置)

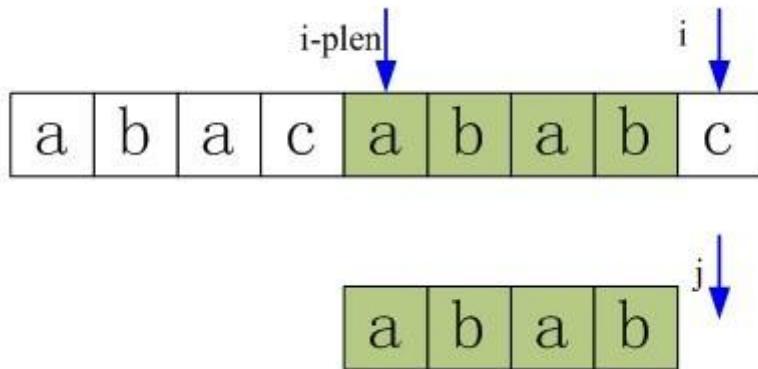


图 3-12 第四步, 跳出循环, 输出结果 $i-plen=4$

所以, 综上, 总结上述四步为:

1. $P[3]! = S[3]$, 匹配失败;
2. $\text{nextval}[3]=1$, 所以 $P[1]$ 继续与 $S[3]$ 匹配, 匹配失败;
3. $\text{nextval}[1]=0$, 所以 $P[0]$ 继续与 $S[3]$ 匹配, 再次匹配失败;
4. $\text{nextval}[0]=-1$, 满足循环 if 部分条件 $j == -1$, 所以, $++i$, $++j$, 主串指针下移一个位置, 从 $P[0]$ 与 $S[4]$ 处开始匹配, 最后 $j == plen$, 跳出循环, 输出结果 $i-plen=4$, 算法结束。

不知, 读者是否已看出, 上面的匹配过程隐藏着一个不容忽略的问题, 即有一个完全可以改进的地方。对的, 问题就出现在上述过程的第二步。

观察上面的匹配过程, 看匹配的第二步, 在第一步的时候已有 $P[3]=b$ 与 $S[3]=c$ 不匹配, 而下一步如果还是要让 $P[\text{next}[3]]=P[1]=b$ 与 $S[3]=c$ 匹配的话, 那么结果很明显, 还是肯定会匹配失败的。由此可以看出我们的 next 值还不是最优的, 也就是说是不能允许 $P[j]=P[\text{next}[j]]$ 出现的, 即上面的求 next 值的算法需要修正。

也就是说上面求得的 $\text{nextval}[i]$ 数组的 4 个元素值, 分别为 -1, 0, 0, 1 是有问题的。有什么问题呢? 就是不容许出现这种情况 $P[j]=P[\text{next}[j]]$ 。为什么?

好比上面的例子。请容许我再次引用上面例子中的两张图。在上面的第一步匹配中, 我们已经得出 $P[3]=b$ 是不等于 $S[3]=c$ 的。而在上面的第二步匹配中, 根据求得的 $\text{nextval}[i]$ 数组值中的 $\text{nextval}[3]=1$, 即让 $P[1]$ 重新与 $S[3]$ 再次匹配。这不是明摆着有问题么? 因为 $P[1]$ 也等于 b 阿, 而在第一步匹配中, 我们已经事先得知 b 是不可能等于 $S[3]$ 的。所以, 第二步匹配之前就已注定是失败的。

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

图 3-13/14 求 next 数组各值的错误解法

这里读者理解可能有困难的是因为文中，时而 `next`，时而 `nextval`，把他们的思维搞混乱了。其实 `next` 用于表达数组索引，而 `nextval` 专用于表达 `next` 数组索引下的具体各值，区别细微。至于文中说不允许 $P[3]=P[next[j]]$ 出现，是因为已经有 $P[3]=b$ 与 $S[1]=b$ 匹配败，

而 $P[next[3]]=P[1]=b$ ，若再拿 $P[1]$ 去与 $S[1]=b$ 匹配则必败。

• 4、求解 `next` 数组各值的方法修正

那么，上面求解 `next` 数组各值的问题到底出现在哪儿呢？我们怎么才能摆脱掉这种情况呢？即不能让 $P[j]=P[next[j]]$ 成立成立。不能再出现上面那样的情况啊！即不能有这种情况出现： $P[3]=b$ ，而竟也有 $P[next[3]]=P[1]=b$ 。

让我们再次回顾一下之前求 `next` 数组的函数代码：

```

1. //引用之前上文第 3 小节中的有错误的求 next 的代码 3-1。
2. void get_next(char const* ptrn, int plen, int* nextval)
3. {
4.     int i = 0;
5.     nextval[i] = -1;
6.     int j = -1;
7.     while( i < plen-1 )

```

```

8.  {
9.    if( j == -1 || ptn[i] == ptn[j] ) //循环的 if 部分
10.   {
11.     ++i;
12.     ++j;
13.     nextval[i] = j; //这里有问题
14.   }
15.   else //循环的 else 部分
16.     j = nextval[j]; //递推
17.   }
18. }
```

由上面之前的代码，我们看到，在求 `next` 值的时候采用的是递推。这里的求法是有问题的。因为在 $s[i] \neq p[j]$ 的时候，如果 $p[j]=p[k]$ ($k=nextval[j]$ ，为之前的错误方法求得的 `next` 值)，那么 $P[k] \neq S[i]$ ，用之前的求法求得的 $next[j]==k$ ，下一步直接导致匹配($S[i]$ 与 $P[k]$ 匹配)失败。

根据上面的分析，我们知道求 `next` 值的时候还要考虑 $P[j]$ 与 $P[k]$ 是否相等。当有 $P[j]=P[k]$ 的时候，只能向前递推出一个 $p[j] \neq p[k]$ ，其中 $k'=next[next[j]]$ 。修正的求 `next` 数组的 `get_nextval` 函数代码如下：

```

1. //代码 4-1
2. //修正后的求 next 数组各值的函数代码
3. void get_nextval(char const* ptn, int plen, int* nextval)
4. {
5.   int i = 0;
6.   nextval[i] = -1;
7.   int j = -1;
8.   while( i < plen-1 )
9.   {
10.     if( j == -1 || ptn[i] == ptn[j] ) //循环的 if 部分
11.     {
12.       ++i;
13.       ++j;
14.       //修正的地方就发生下面这 4 行
15.       if( ptn[i] != ptn[j] ) //++i, ++j 之后，再次判断 ptn[i] 与 ptn[j]
          的关系
16.         nextval[i] = j; //之前的错误解法就在于整个判断只有这一句。
17.       else
18.         nextval[i] = nextval[j];
19.     }
20.     else //循环的 else 部分
21.       j = nextval[j];
```

```
22.      }
23. }
```

举个例子，举例说明下上述求 **next** 数组的方法。

S a b a b a b c

P a b a b c

S[4] != P[4]

那么下一个和 S[4] 匹配的位置是 k=2(也即 P[next[4]])。此处的 k=2 也再次佐证了上文第 3 节开头处关于为了找到下一个匹配的位置时 k 的求法。上面的主串与模式串开头 4 个字符都是“abab”，所以，匹配失效后下一个匹配的位置直接跳两步继续进行匹配。

S a b a b a b c

P a b a b c

匹配成功

P 的 **next** 数组值分别为 -1 0 -1 0 2

next 数组各值怎么求出来的呢？分以下五步：

1. 初始化： i=0, j=-1;

2. i=1, j=0, 进入循环 **esle** 部分, j=nextval[j]=nextval[0]=-1;

3. 进入循环的 **if** 部分, ++i, ++j, i=2, j=0, 因为 pptrn[i]=pptrn[j]=a, 所以

nextval[2]=nextval[0]=-1;

4. i=2, j=0, 由于 pptrn[i]=pptrn[j], 再次进入循环 **if** 部分, 所以 ++i=3, ++j=1, 因为

pptrn[i]=pptrn[j]=b, 所以 nextval[3]=nextval[1]=0;

5. i=3, j=1, 由于 pptrn[i]=pptrn[j]=b, 所以 ++i=4, ++j=2, 因为 pptrn[i] != pptrn[j], 所以

nextval[4]=2。

这样上例中模式串的 **next** 数组各值最终应该为：

a	b	a	b
-1	0	-1	0

图 4-1 正确的 **next** 数组各值

next 数组求解的具体过程如下：

初始化： **nextval[0] = -1**， 我们得到第一个 **next** 值即 -1.

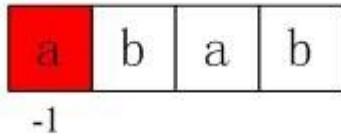


图 4-2 第一个 next 值即-1

$i = 0, j = -1$, 由于 $j == -1$, 进入上述循环的 if 部分, $++i$ 得 $i=1$, $++j$ 得 $j=0$, 且 $\text{ptrn}[i] != \text{ptrn}[j]$ (即 $a != b$) , 所以得到第二个 next 值即 $\text{nextval}[1] = 0$;

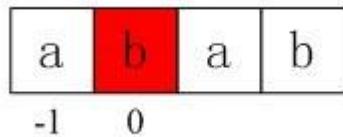


图 4-3 第二个 next 值 0

上面我们已经得到, $i = 1, j = 0$, 由于不满足条件 $j == -1 || \text{ptrn}[i] == \text{ptrn}[j]$, 所以进入循环的 else 部分, 得 $j = \text{nextval}[j] = -1$; 此时, 仍满足循环条件, 由于 $i = 1, j = -1$, 因为 $j == -1$, 再次进入循环的 if 部分, $++i$ 得 $i=2$, $++j$ 得 $j=0$, 由于 $\text{ptrn}[i] == \text{ptrn}[j]$ (即 $\text{ptrn}[2]=\text{ptrn}[0]$, 也就是说第 1 个元素和第三个元素都是 a) , 所以进入循环 if 部分内嵌的 else 部分, 得到 $\text{nextval}[2] = \text{nextval}[0] = -1$;

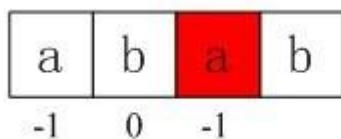


图 4-4 第三个 next 数组元素值-1

$i = 2, j = 0$, 由于 $\text{ptrn}[i] == \text{ptrn}[j]$, 进入 if 部分, $++i$ 得 $i=3$, $++j$ 得 $j=1$, 所以 $\text{ptrn}[i] == \text{ptrn}[j]$ ($\text{ptrn}[3]==\text{ptrn}[1]$, 也就是说第 2 个元素和第 4 个元素都是 b) , 所以进入循环 if 部分内嵌的 else 部分, 得到 $\text{nextval}[3] = \text{nextval}[1] = 0$;

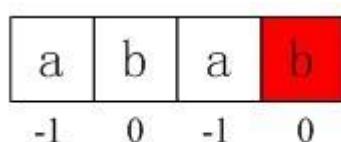


图 4-5 第四个数组元素值 0

如果你还是没有弄懂上述过程是怎么一回事，请现在拿出一张纸和一支笔出来，一步一步的画下上述过程。相信我，把图画出来了之后，你一定能明白它的。

然后，我留一个问题给读者，为什么上述的 next 数组要那么求？有什么原理么？

• 5、利用求得的 next 数组各值运用 Kmp 算法

Ok，next 数组各值已经求得，万事俱备，东风也不欠了。接下来，咱们就要应用求得的 next 值，应用 KMP 算法来匹配字符串了。还记得 KMP 算法是怎么一回事吗？容我再次引用下之前的 KMP 算法的代码，如下：

```
1. //代码 5-1
2. //int kmp_seach(char const*, int, char const*, int, int const*, int pos) KM
   P 模式匹配函数
3. //输入: src, slen 主串
4. //输入: patn, plen 模式串
5. //输入: nextval KMP 算法中的 next 函数值数组
6. int kmp_search(char const* src, int slen, char const* patn, int plen, int co
   nst* nextval, int pos)
7. {
8.     int i = pos;
9.     int j = 0;
10.    while ( i < slen && j < plen )
11.    {
12.        if( j == -1 || src[i] == patn[j] )
13.        {
14.            ++i;
15.            ++j;
16.        }
17.        else
18.        {
19.            j = nextval[j];
20.            //当匹配失败的时候直接用 p[j_next] 与 s[i] 比较,
21.            //下面阐述怎么求这个值, 即匹配失效后下一次匹配的位置
22.        }
23.    }
24.    if( j >= plen )
25.        return i-plen;
26.    else
27.        return -1;
28. }
```

我们上面已经求得的 next 值，如下：

a	b	a	b
-1	0	-1	0

图 5-1 求得的正确的 next 数组元素各值

以下是匹配过程，分三步：

第一步：主串和模式串如下，S[3]与 P[3]匹配失败。

a	b	a	c	a	b	a	b	c
a	b	a	b					

-1	0	-1	0					
----	---	----	---	--	--	--	--	--

图 5-2 第一步，S[3]与 P[3]匹配失败

第二步：S[3]保持不变，P 的下一个匹配位置是 P[next[3]]，而 next[3]=0, 所以 P[next[3]]=P[0]，即 P[0]与 S[3]匹配。在 P[0]与 S[3]处匹配失败。

a	b	a	c	a	b	a	b	c
a	b	a	b					

-1	0	-1	0					
----	---	----	---	--	--	--	--	--

图 5-3 第二步，在 P[0]与 S[3]处匹配失败

第三步：与上文中第 3 小节末的情况一致。由于上述第三步中，P[0]与 S[3]还是不匹配。此时 i=3, j=nextval[0]=-1, 由于满足条件 j== -1，所以进入循环的 if 部分，++i=4, ++j=0, 即主串指针下移一个位置，从 P[0]与 S[4]处开始匹配。最后 j==plen，跳出循环，输出结果 i-plen=4(即字串第一次出现的位置)，匹配成功，算法结束。

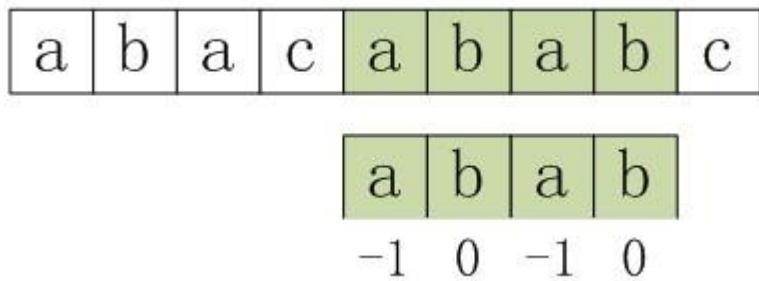


图 5-4 第三步，匹配成功，算法结束

所以，综上，总结上述三步为：

1. 开始匹配，直到 $P[3] \neq S[3]$ ，匹配失败；
2. $\text{nextval}[3]=0$ ，所以 $P[0]$ 继续与 $S[3]$ 匹配，再次匹配失败；
3. $\text{nextval}[0]=-1$ ，满足循环 if 部分条件 $j == -1$ ，所以， $++i$, $++j$ ，主串指针下移一个位置，从 $P[0]$ 与 $S[4]$ 处开始匹配，最后 $j == \text{plen}$ ，跳出循环，输出结果 $i - \text{plen} = 4$ ，算法结束。

与上文中第 3 小节的四步匹配相比，本节运用修正过后的 next 数组，去掉了第 3 小节的第 2 个多余步骤的 $\text{nextval}[3]=1$ ，所以 $P[1]$ 继续与 $S[3]$ 匹配，匹配失败（缘由何在？因为与第 3 小节的 next 数组相比，此时的 next 数组中 $\text{nextval}[3]$ 已等于 0）。所以，才只需要三个匹配步骤了。

ok，KMP 算法已宣告完结，希望已经了却了心中的一块结石。毕竟，这个 KMP 算法此前也困扰了我很长一段时间。耐心点，慢慢来，总会搞懂的。闲不多说，接下来，咱们开始介绍 BM 算法。

第二部分、BM 算法

1、简单的后比对算法

为了更好的理解 BM 算法，我分三步引入 BM 算法。首先看看下面的一个字符串匹配算法，它与前面的回溯法差不多，看看差别在哪儿。

```

1. /*! int search_reverse(char const*, int, char const*, int)
2. */bref 查找出模式串 patn 在主串 src 中第一次出现的位置
3. */return patn 在 src 中出现的位置，当 src 中并没有 patn 时，返回-1
4. */

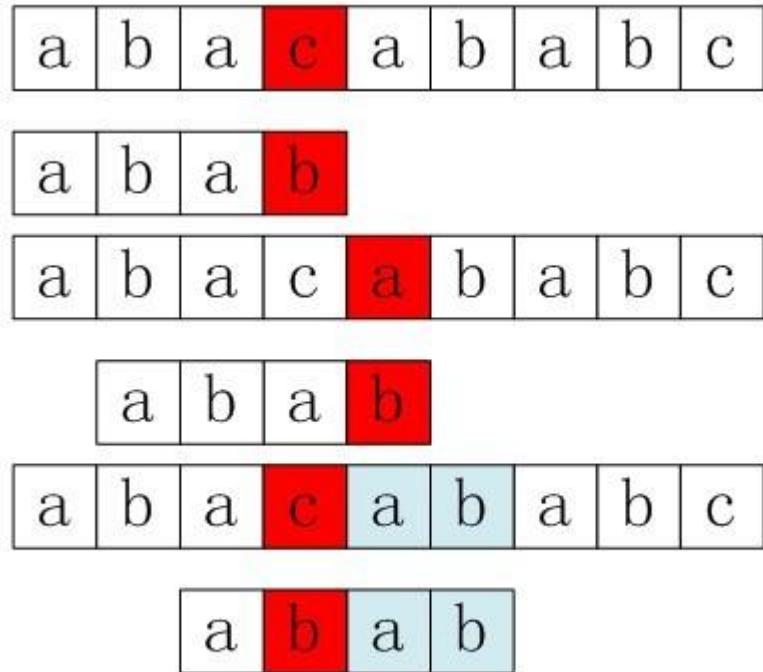
```

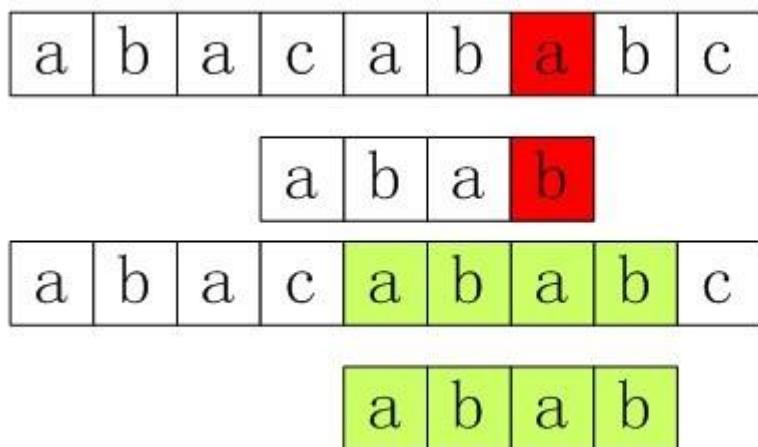
```

5. int search_reverse(char const* src, int slen, char const* patn, int plen)
6. {
7.     int s_idx = plen, p_idx;
8.     if (plen == 0)
9.         return -1;
10.    while (s_idx <= slen)//计算字符串是否匹配到了尽头
11.    {
12.        p_idx = plen;
13.        while (src[--s_idx] == patn[--p_idx])//开始匹配
14.        {
15.            //if (s_idx < 0)
16.            //return -1;
17.            if (p_idx == 0)
18.            {
19.                return s_idx;
20.            }
21.        }
22.        s_idx += (plen - p_idx)+1;
23.    }
24.    return -1
25. }

```

仔细分析上面的代码，可以看出该算法的思路是从模式串的后面向前面匹配的，如果后面的几个都不匹配了，就可以直接往前面跳了，直觉上这样匹配更快些。是否真是如此呢？请先看下面的例子。



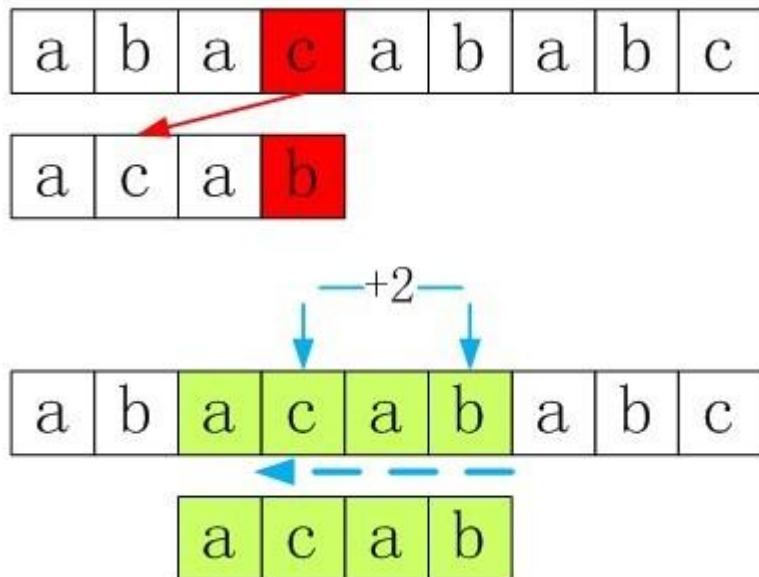


上面是详细的算法流程，接下来我们就用上面的例子，来引出坏 2、字符规则，3、最好后缀规则，最终引出 4、BM 算法。

2、坏字符规则

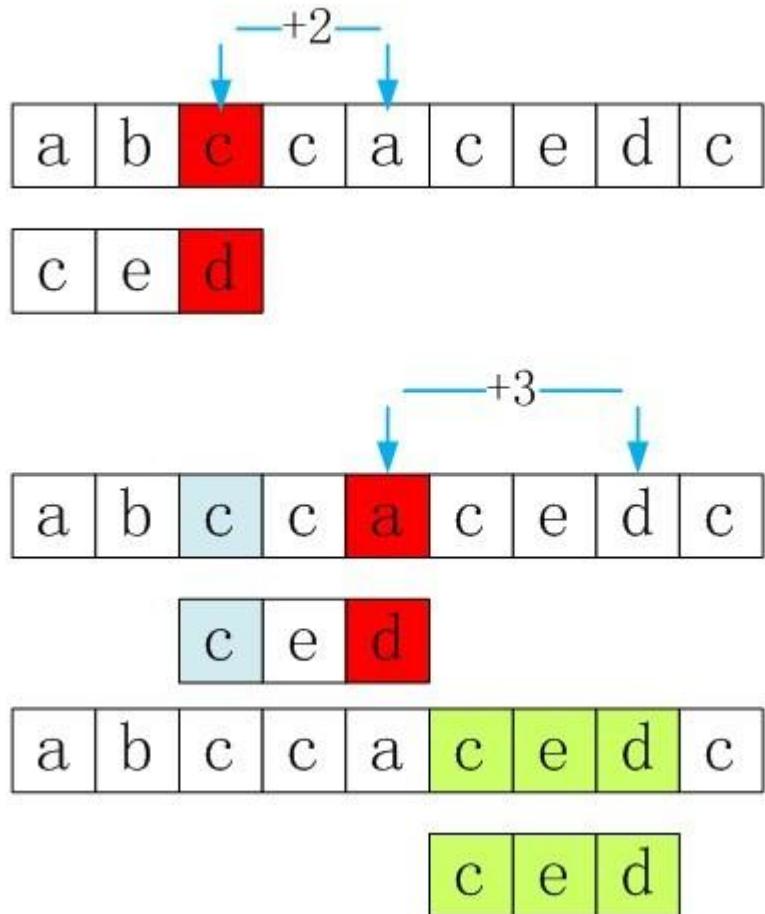
在上面的例子里面，第一步的时候， $S[3] = 'c' \neq P[3]$ ，下一步应该当整个模式串移过 $S[3]$ 即可，因为 $S[3]$ 已经不可能与 P 中的任何一个部分相匹配了。那是不是只是对于 P 中不存在的字符就这样直接跳过呢，如果 P 中存在的字符该怎么定位呢？

如模式串为 $P = "acab"$ ，基于坏字符规则匹配步骤分解图如下：



从上面的例子可以看出，我们需要建一张表，表示 P 中字符存在的情况，不存在，则 s_idx 直接加上 $plen$ 跳过该字符，如果存在，则需要找到从后往前最近的一个字符对齐匹配，如上面的例子便已经说明了坏字符规则匹配方法。

再看下面的例子：



由此可见，第一个匹配失败的时候 $S[i] = 'c'$ ，主串指针需要+2 才有可能在下一次匹配成功，同理第二次匹配失败的时候， $S[i] = 'a'$ ，主串指针需要+3 直接跳过'a'才能下一次匹配成功。

对于 $S[i]$ 字符，有 256 种可能，所以需要对于模式串建立一张长度为 256 的坏字符表，其中当 P 中没出现的字符，表值为 $plen$ ，如果出现了，则设置为最近的一个对齐的值。具体算法比较简单如下：

```

1. /*
2.
3.   函数: void BuildBadCharacterShift(char *, int, int*)
4.   目的: 根据好后缀规则做预处理, 建立一张好后缀表
5.   参数:
6.   pattern => 模式串 P
7.   plen => 模式串 P 长度
8.   shift => 存放坏字符规则表, 长度为的 int 数组
9.   返回: void
10. */
11. void BuildBadCharacterShift(char const* pattern, int plen, int* shift)
12. {
13.     for( int i = 0; i < 256; i++ )
14.         *(shift+i) = plen;

```

```

15.     while ( plen >0 )
16.     {
17.         *(shift+(unsigned char)*pattern++) = --plen;
18.     }
19. }
```

这个时候整个算法的匹配算法该是怎么样的呢，是将上面的 search_reverse 函数中的 `s_idx+=(plen-p_idx)+1` 改成 `s_idx+= shift[(unsigned char)patn[p_idx]] +1` 吗？ 不是的，代码给出如下，具体原因读者可自行分析。

```

1. /*! int search_badcharacter(char const*, int, char const*, int)
2.
3.     */bref 查找出模式串 patn 在主串 src 中第一次出现的位置
4.
5.     */return patn 在 src 中出现的位置，当 src 中并没有 patn 时，返回-1
6.
7. */
8. int search_badcharacter(char const* src, int slen, char const* patn, int plen,
   int* shift)
9. {
10.     int s_idx = plen, p_idx;
11.     int skip_stride;
12.     if (plen == 0)
13.         return -1;
14.     while (s_idx <= slen)//计算字符串是否匹配到了尽头
15.     {
16.         p_idx = plen;
17.         while (src[--s_idx] == patn[--p_idx])//开始匹配
18.         {
19.             //if (s_idx < 0)
20.             //Return -1;
21.             if (p_idx == 0)
22.             {
23.                 return s_idx;
24.             }
25.         }
26.         skip_stride = shift[(unsigned char)src[s_idx]];
27.         s_idx += (skip_stride>plen-p_idx ? skip_stride: plen-p_idx)+1;
28.     }
29.     return -1;
30. }
```

• 3、最好后缀规则

在讲最好后缀规则之前，我们先回顾一下本部分第 1 小节中所举的一个简单后比对算法的例子：

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

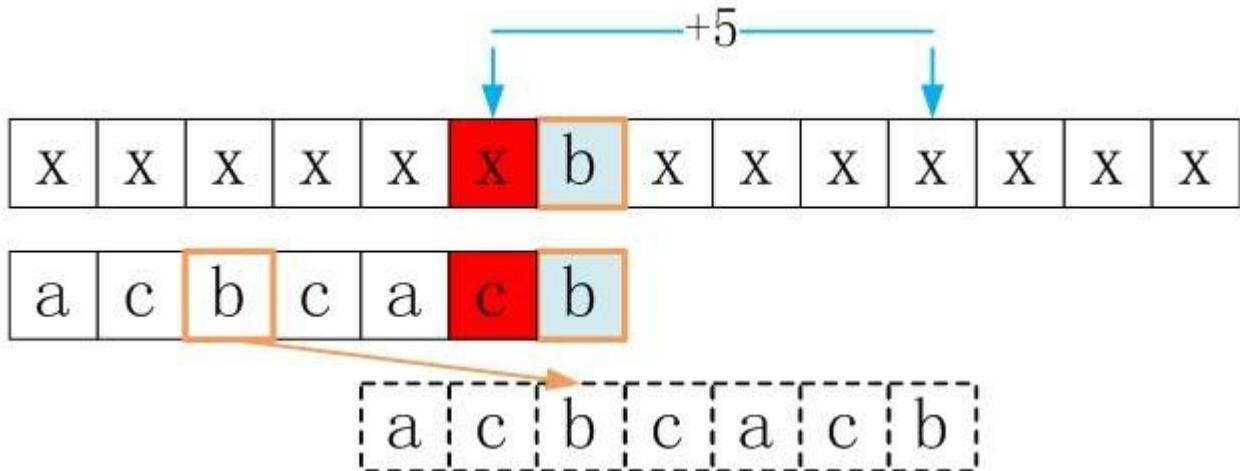
a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

上面倒数第二步匹配是没必要的。为什么呢？在倒数第三步匹配过程中，已有最后两个字符与模式串 P 中匹配，而模式串中有前两个与后两个字符相同的，所以可以直接在接下来将 P 中的前两个与主串中匹配过的'ab'对齐，做为下一次匹配的开始。

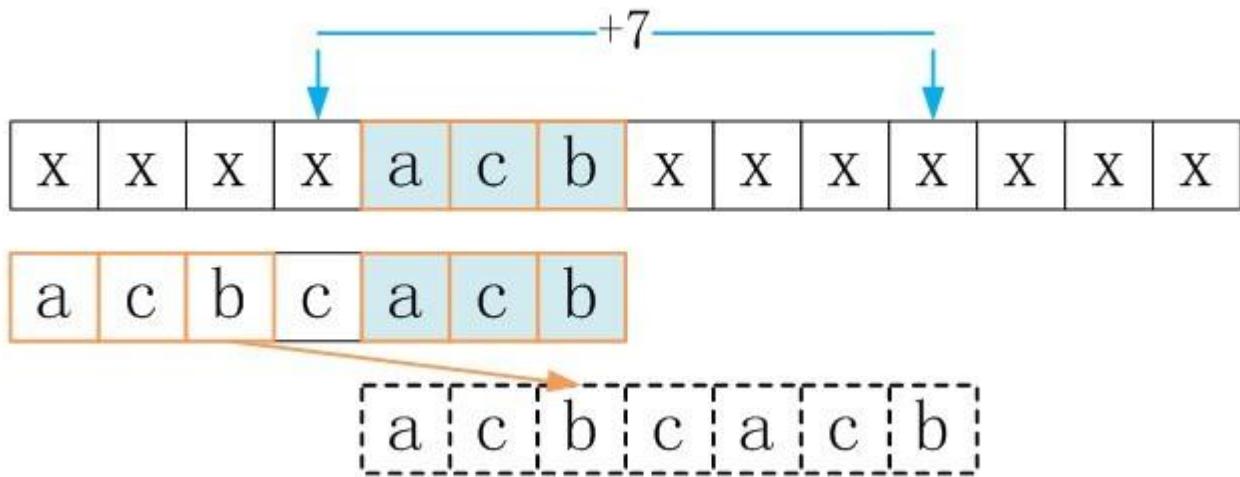
其实思路与本文第一部分讲过的 KMP 算法差不多，也是利用主串与模式串已匹配成功的部分来找一个合适的位置方便下一次最有效的匹配。只是这里是需要寻找一个位置，让已匹配过的后缀与模式串中从后往前最近的一个相同的子串对齐。（理解这句话就理解了 BM 算法的原理）这里就不做数学描述了。

ok，主体思想有了，怎么具体点呢？下面，直接再给一个例子，说明这种匹配过程。看下图吧。



由图可以 $\text{goodsuffixshift}[5] = 5$

下面看 $\text{goodsuffixshift}[3]$ 的求解



求解最好后缀数组是 BM 算法之所以难的根本，所以建议多花时间理清思路。网上有很多方法，我也试过两个，一经测试，很多都不算准确，最好后缀码的求解不像 KMP 的“最好前缀数组”那样可以用递推的方式求解，而是有很多细节。

代码如下：

```

1. /*
2.  函数: void BuildGoodSuffixShift(char *, int, int*)
3.  目的: 根据最好后缀规则做预处理, 建立一张好后缀表
4.  参数:
5.  pattern => 模式串 P
6.  plen => 模式串 P 长度
7.  shift => 存放最好后缀表数组
8.  返回: void
9. */
10. void BuildGoodSuffixShift(char const* pattern, int plen, int* shift)
11. {
12.     shift[plen-1] = 1;           // 右移动一位

```

```

13.     char end_val = pattern[plen-1];
14.     char const* p_prev, const* p_next, const* p_temp;
15.     char const* p_now = pattern + plen - 2;           // 当前配匹不相符字符,
求其对应的 shift
16.     bool isgoodsuffixfind = false;                  // 指示是否找到了最好后
缀子串,修正 shift 值
17.     for( int i = plen - 2; i >=0; --i, --p_now)
18.     {
19.         p_temp = pattern + plen - 1;
20.         isgoodsuffixfind = false;
21.         while ( true )
22.         {
23.             while (p_temp >= pattern && *p_temp-- != end_val);      //
从 p_temp 从右往左寻找和 end_val 相同的字符串子串
24.             p_prev = p_temp;          // 指向与 end_val 相同的字符的前一个
25.             p_next = pattern + plen - 2;           // 指向 end_val 的前一个
26.             // 开始向前匹配有以下三种情况
27.             //第一: p_prev 已经指向 pattern 的前方,即没有找到可以满足条件的最好后缀
子串
28.             //第二: 向前匹配最好后缀子串的时候,p_next 开始的子串先到达目的地
p_now,
29.             //需要判断 p_next 与 p_prev 是否相等,如果相等,则继续往前找最好后缀子
串
30.             //第三: 向前匹配最好后缀子串的时候,p_prev 开始的子串先到达端点
pattern, 这个可以算是最好的子串
31.
32.             if( p_prev < pattern && *(p_temp+1) != end_val )        // 没有
找到与 end_val 相同字符
33.             break;
34.
35.             bool match_flag = true;           //连续匹配失败标志
36.             while( p_prev >= pattern && p_next > p_now )
37.             {
38.                 if( *p_prev --!= *p_next-- )
39.                 {
40.                     match_flag = false;       //匹配失败
41.                     break;
42.                 }
43.             }
44.
45.             if( !match_flag )
46.                 continue;            //继续向前寻找最好后缀子串
47.             else
48.             {

```

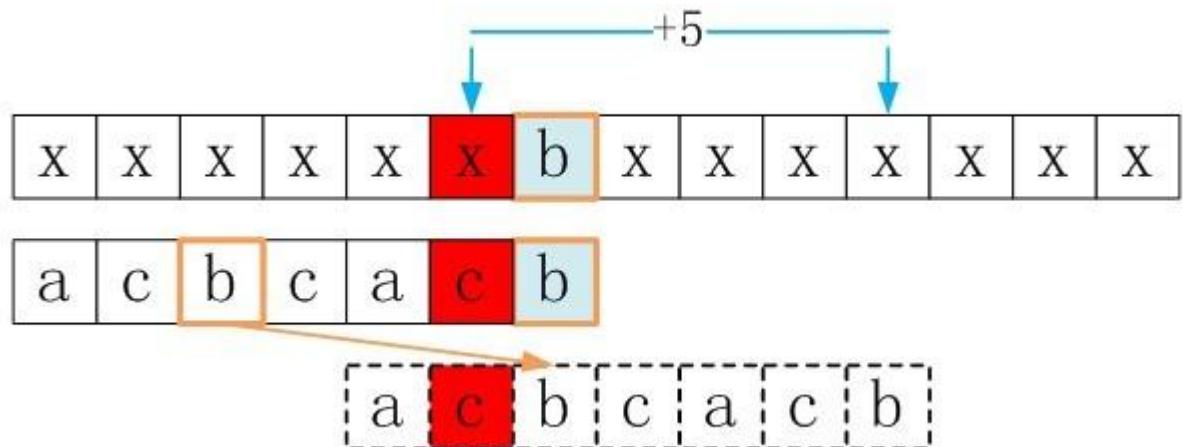
```

49.             // 匹配没有问题，是边界问题
50.             if( p_prev < pattern || *p_prev != *p_next)
51.             {
52.                 // 找到最好后缀子串
53.                 isgoodsuffixfind = true;
54.                 break;
55.             }
56.             // *p_prev == * p_next 则继续向前找
57.         }
58.     }
59.     shift[i] = plen - i + p_next - p_prev;
60.     if( isgoodsuffixfind )
61.         shift[i]--;
62.     }
63. }

```

注：代码里求得到的 goodsuffixshift 值与上述图解中有点不同，这也是我看网上代码时做的一个小的改进。请注意。另外，如上述代码的注释里所述，开始向前匹配有以下三种情况：

- 第一：p_prev 已经指向 pattern 的前方,即没有找到可以满足条件的最好后缀子串
- 第二：向前匹配最好后缀子串的时候,p_next 开始的子串先到达目的地 p_now, 需要判断 p_next 与 p_prev 是否相等,如果相等,则继续往前找最好后缀子串
- 第三：向前匹配最好后缀子串的时候,p_prev 开始的子串先到达端点 pattern, 这个可以算是最好的子串。下面，咱们分析这个例子：

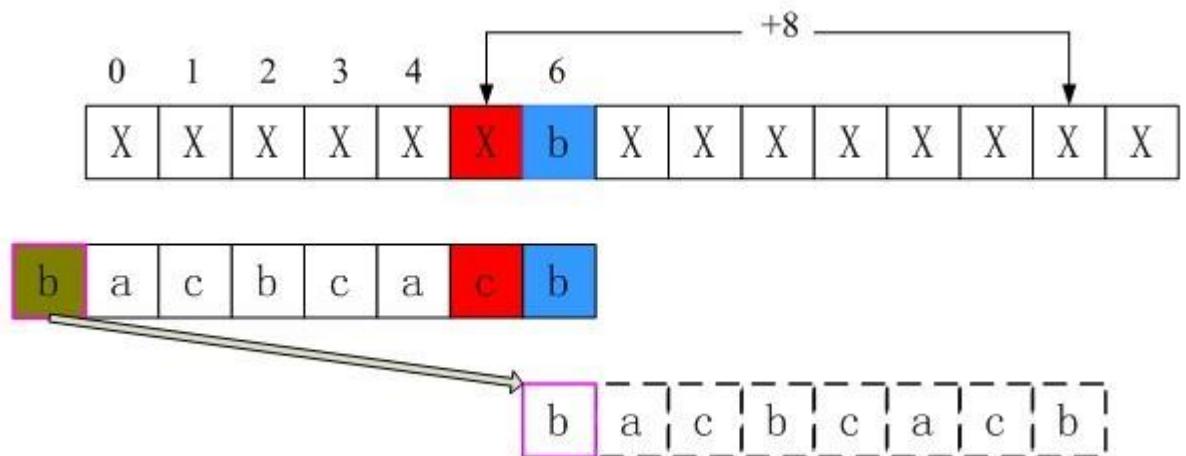


从图中可以看出，在模式串 P 中，P[2]=P[6]但 P[1]也等于 P[5]，所以如果只移 5 位让 P[2]与 S[6]对齐是没必要的，因为 P[1]不可能与 S[5]相等（如红体字符表示），对于这

种情况， $P[2]=P[6]$ 就不算最好后缀码了，所以应该直接将整个 P 滑过 $S[6]$ ，所以 $\text{goodsuffixshift}[5]=8$ 而不是 5。也就是说，在匹配过程中已经得出 $P[1]$ 是不可能等于 $S[5]$ 的，所以，就算为了达到 $P[2]$ 与 $S[6]$ 匹配的效果，让模式串 P 右移 5 位，但在 $P[1]$ 处与 $S[5]$ 处还是会导至匹配失败。所以，必定会匹配失败的事，我们又何必多次一举呢？

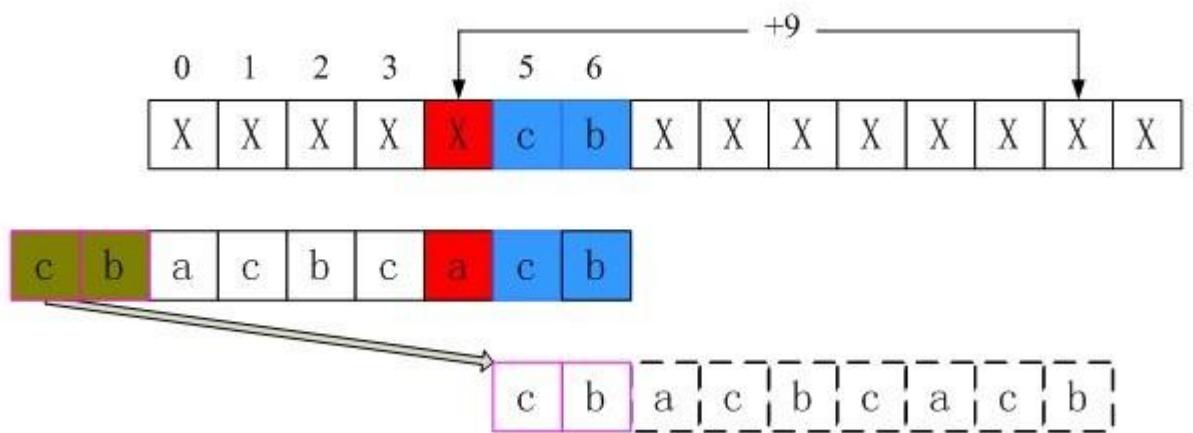
那么，我们到底该怎么做呢？如果我现在直接给出代码的话，可能比较难懂，为了进一步说明，以下图解是将 BM 算法的好后缀表数组 shift （不匹配时直接跳转长度数组）的求解过程。其中第一行为 src 数组，第二行为 patn 数组，第三行为匹配失败时下一次匹配时的 patn 数组（粉色框的元素实际不存在）。

1、 $i=5$ 时不匹配的情况



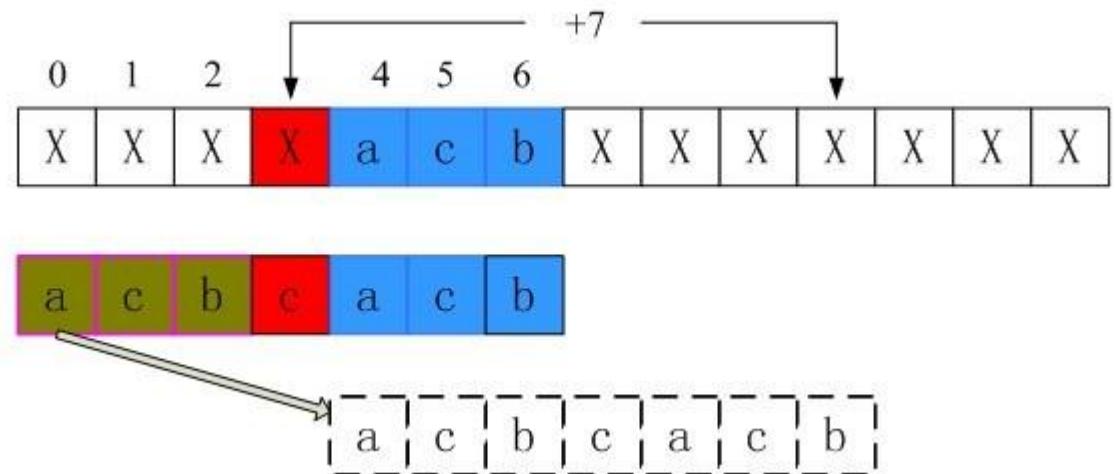
ok，现在咱们定位于 $P[5]$ 处，当 $i = 5$ 时 $\text{src}[5] \neq \text{patn}[5]$ ， p_now 指向 $\text{patn}[5]$ ，而 p_prev 指向 $\text{patn}[1]$ ，即情况二。由于此时 $*p_prev == *p_now$ ，则继续往前找最好后缀子串。循环直到 p_prev 指向 $\text{patn}[0]$ 的前一个位置（实际不存在，为了好理解加上去的）。此时 p_prev 指向 $\text{patn}[0]$ 的前方，即情况一。此时条件 $p_prev < \text{pattern} \&& *(p_temp+1) \neq \text{end_val}$ 满足，所以跳出循环。计算 $\text{shift}[5]=\text{plen}-i+p_next-p_prev=8$ （实际上是第三行的长度）。

2、 $i=4$ 时不匹配的情况



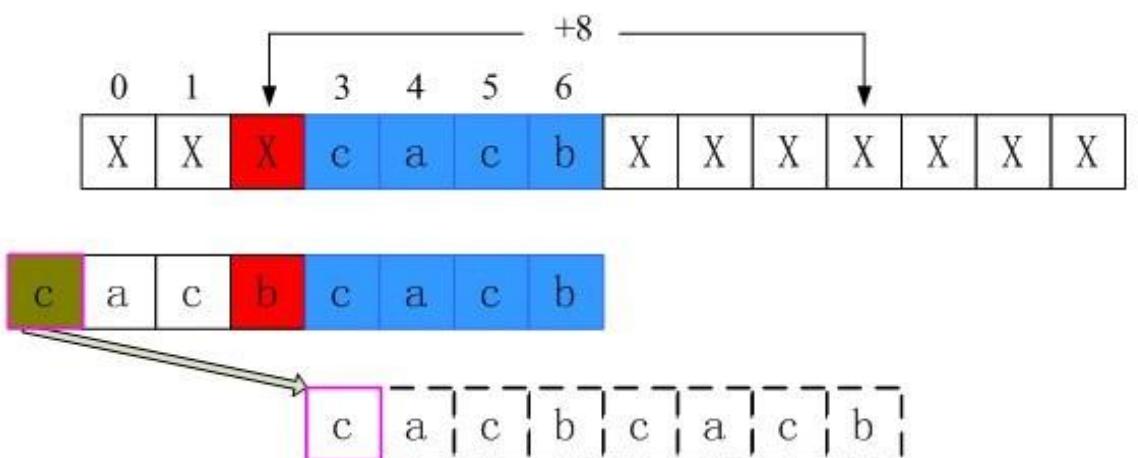
$i = 4$ 时, $\text{src}[4] \neq \text{patn}[4]$, 此时 p_prev 指向 $\text{patn}[0]$, p_now 指向 $\text{patn}[4]$, 即情况二。由于此时 $*p_prev == *p_now$, 则继续往前找最好后缀子串。循环直到 p_prev 指向 $\text{patn}[0]$ 的前一个位置。此时 p_prev 指向 $\text{patn}[0]$ 的前方, 即情况一。此时条件 $p_prev < \text{pattern} \&& *(p_temp+1) \neq \text{end_val}$ 满足, 所以跳出循环。计算 $\text{shift}[4] = \text{plen} - i + p_next - p_prev = 9$ (实际上第三行的长度)。

3、 $i = 3$ 时不匹配的情况



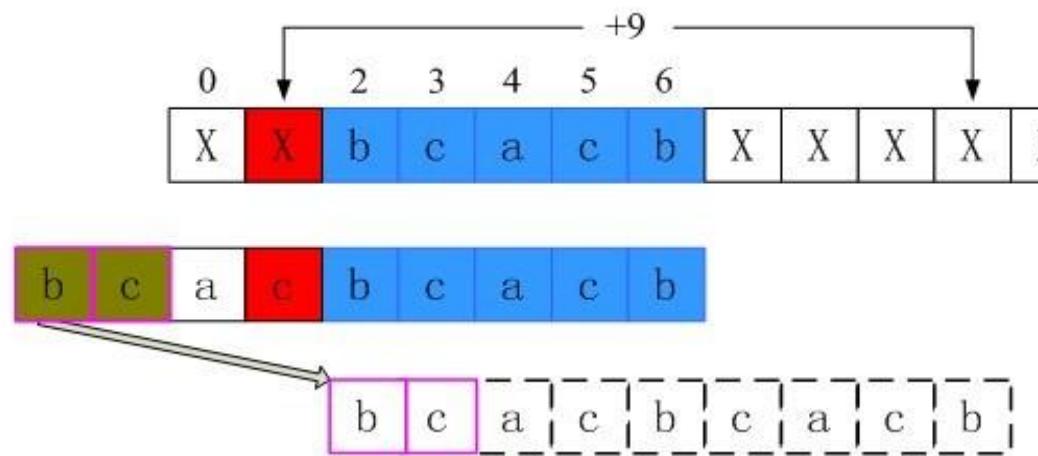
同样的过程可以得到, $i = 3$ 时 $\text{shift}[3]$ 也为第三行的长度 7。

4、 $i = 2$ 时不匹配的情况



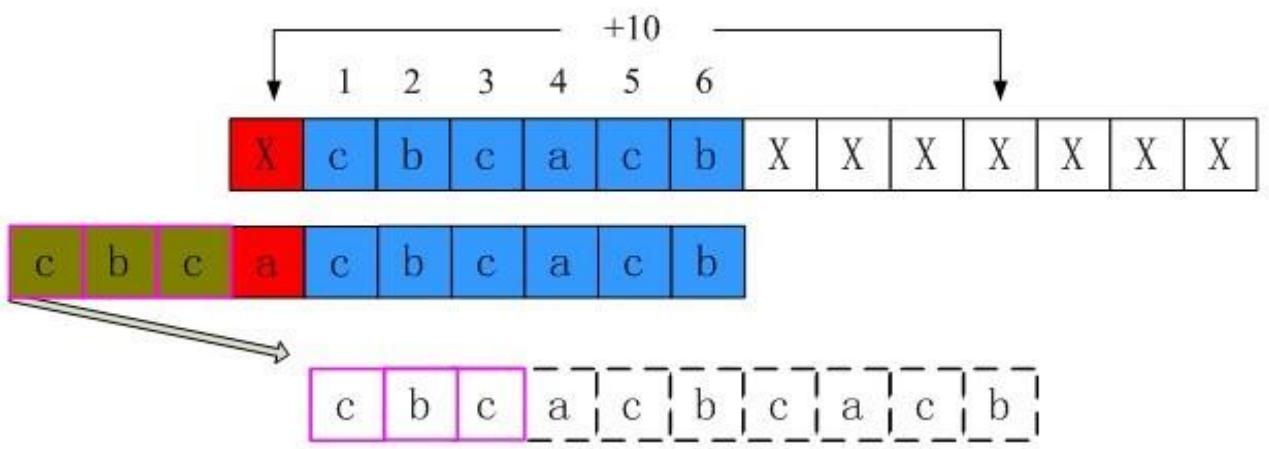
同样的过程可以得到, $i = 2$ 时 $\text{shift}[2]$ 也为第三行的长度 8。

5、 $i = 1$ 时不匹配的情况



同样的过程可以得到， $i = 1$ 时 $\text{shift}[1]$ 也为第三行的长度 9。

6、 $i = 0$ 时不匹配的情况



同样的过程可以得到， $i = 0$ 时 $\text{shift}[0]$ 也为第三行的长度 10。

计算好后缀表数组后，这种情况下的字模式匹配算法为：

```

1. /*! int search_goodsuffix(char const*, int, char const*, int)
2.
3.     */brief 查找出模式串 patn 在主串 src 中第一次出现的位置
4.
5.     */return patn 在 src 中出现的位置，当 src 中并没有 patn 时，返回-1
6.
7. */
8. int search_goodsuffix(char const* src, int slen, char const* patn, int plen,
    int* shift)
9. {
10.     int s_idx = plen, p_idx;
11.     int skip_stride;
12.     if (plen == 0)

```

```

13.         return -1;
14.
15.     while (s_idx <= sLen)//计算字符串是否匹配到了尽头
16.     {
17.         p_idx = plen;
18.         while (src[--s_idx] == patn[--p_idx])//开始匹配
19.         {
20.             //if (s_idx < 0)
21.             //return -1;
22.             if (p_idx == 0)
23.             {
24.                 return s_idx;
25.             }
26.         }
27.         skip_stride = shift[p_idx];
28.         s_idx += skip_stride +1;
29.     }
30.     return -1;
31. }
```

4、BM 算法

有了前面的三个步骤的算法的基础，BM 算法就比较容易理解了，其实 BM 算法就是将坏字符规则与最好后缀规则的综合具体代码如下，相信一看就会明白。

```

1. /*
2. 函数: int* BMSearch(char *, int , char *, int, int *, int *)
3. 目的: 判断文本串 T 中是否包含模式串 P
4. 参数:
5. src => 文本串 T
6. sLen => 文本串 T 长度
7. ptn => 模式串 P
8. pLen => 模式串 P 长度
9. bad_shift => 坏字符表
10. good_shift => 最好后缀表
11. 返回:
12. int - 1 表示匹配失败, 否则返回
13. */
14. int BMSearch(char const*src, int sLen, char const*ptn, int plen, int const*
   bad_shift, int const*good_shift)
15. {
16.     int s_idx = plen;
17.     if (plen == 0)
18.         return 1;
19.
20.     while (s_idx <= sLen)//计算字符串是否匹配到了尽头
21.     {
```

```

22.         int p_idx = plen, bad_stride, good_stride;
23.         while (src[--s_idx] == ptn[--p_idx])//开始匹配
24.         {
25.             //if (s_idx < 0)
26.             //return -1;
27.
28.             if (p_idx == 0)
29.             {
30.                 return s_idx;
31.             }
32.         }
33.
34.         // 当匹配失败的时候，向前滑动
35.         bad_stride = bad_shift[(unsigned char)src[s_idx]];      //根据坏字符
规则计算跳跃的距离
36.         good_stride = good_shift[p_idx];                      ///
根据好后缀规则计算跳跃的距离
37.         s_idx += ((good_stride > bad_stride) ? good_stride : bad_stride )+1;
//取大者
38.     }
39.     return -1;
40. }
```

版权所有，侵权必究。严禁用于任何商业用途，转载请注明出处。

六(三续): KMP 算法之总结篇(必懂 KMP)

作者: July。

出处: http://blog.csdn.net/v_JULY_v/。

引记

此前一天，一位 MS 的朋友邀我一起去与他讨论快速排序，红黑树，字典树，B 树、后缀树，包括 KMP 算法，唯独在讲解 KMP 算法的时候，言语磕磕碰碰，我想，原因有二：1、博客内的东西不常回顾，忘了不少；2、便是我对 KMP 算法的理解还不够彻底，自不用说讲解自如，运用自如了。所以，特再写本篇文章。由于此前，个人已经写过关于 KMP 算法的两篇文章，所以，本文名为：KMP 算法之总结篇。

本文分为如下六个部分：

1. 第一部分、再次回顾普通的 BF 算法与 KMP 算法各自的时间复杂度，并两相对照各自的匹配原理；
2. 第二部分、通过我此前第二篇文章的引用，用图从头到尾详细阐述 KMP 算法中的 next 数组求法，并运用求得的 next 数组写出 KMP 算法的源码；
3. 第三部分、KMP 算法的两种实现，代码实现一是根据本人关于 KMP 算法的第二篇文章所写，代码实现二是根据本人的关于 KMP 算法的第一篇文章所写；
4. 第四部分、测试，分别对第三部分的两种实现中 next 数组的求法进行测试，挖掘其区别之所在；
5. 第五部分、KMP 完整准确源码，给出 KMP 算法的准确的完整源码；
6. 第六部分、一眼看出字符串的 next 数组各值，通过几个例子，让读者能根据字符串本身一眼判断出其 next 数组各值。

力求让此文彻底让读者洞穿此 KMP 算法，所有原理，来龙去脉，让读者搞个通通透透（注意，本文中第二部分及第三部分的代码实现一的字符串下标 i 从 0 开始计算，其它部分如第三部分的代码实现二，第五部分，和第六部分的字符串下标 i 皆是从 1 开始的）。

在看本文之前，你心中如若对前缀和后缀这个两个概念有自己的理解，便最好了。有些东西比如此 KMP 算法需要我们反复思考，反复求解才行。个人写的关于 KMP 算法的第二篇文章为：[六（续）、从 KMP 算法一步一步谈到 BM 算法](#)；第一篇为：[六、教你初步了解 KMP 算法、updated](#)（文末链接）。ok，若有任何问题，恳请不吝指正。多谢。

第一部分、KMP 算法初解

1、普通字符串匹配 BF 算法与 KMP 算法的时间复杂度比较

KMP 算法是一种线性时间复杂的字符串匹配算法，它是对 BF 算法（Brute-Force，最基本的字符串匹配算法的）改进。对于给的原始串 S 和模式串 P，需要从字符串 S 中找到字符串 P 出现的位置的索引。

BF 算法的时间复杂度 $O(\text{strlen}(S) * \text{strlen}(T))$, 空间复杂度 $O(1)$ 。

KMP 算法的时间复杂度 $O(\text{strlen}(S) + \text{strlen}(T))$, 空间复杂度 $O(\text{strlen}(T))$ 。

2、BF 算法与 KMP 算法的区别

假设现在 S 串匹配到 i 位置, T 串匹配到 j 位置。那么总的来说, 两种算法的主要区别在于失配的情况下, 对 $\text{next}[j]$ 的值做的处理:

BF 算法中, 如果当前字符匹配成功, 即 $s[i+j] == T[j]$, 令 $j++$, 继续匹配下一个字符; 如果失配, 即 $s[i + j] != T[j]$, 需要让 $i++$, 并且 $j = 0$, 即每次匹配失败的情况下, 模式串 T 相对于原始串 S 向右移动了一位。

而 KMP 算法中, 如果当前字符匹配成功, 即 $S[i]==T[j]$, 令 $i++$, $j++$, 继续匹配下一个字符; 如果匹配失败, 即 $S[i] != T[j]$, 需要保持 i 不变, 并且让 $j = \text{next}[j]$, 这里 $\text{next}[j] <= j - 1$, 即模式串 T 相对于原始串 S 向右移动了至少 1 位(移动的实际位数 $j - \text{next}[j] >= 1$)。

如果下次匹配是基于 T 向右移动一位, 那么 i 之前的部分(即 $S[i-j+1 ~ i-1]$), 和 $j=\text{next}[j]$ 之前的部分(即 $T[0 ~ j-2]$)仍然相等。显然, 相对于 BF 算法来说, KMP 移动更多的位数, 起到了一个加速的作用! (失配的特殊情形, 令 $j=\text{next}[j]$ 导致 $j==0$ 的时候, 需要将 $i++$, 否则此时没有移动模式串)。

3、BF 算法为什么要回溯

首先说一下为什么 BF 算法要回溯。如下两字符串匹配(恰如上面所述: BF 算法中, 如果当前字符匹配成功, 即 $s[i+j] == T[j]$, 令 $j++$, 继续匹配下一个字符) :

$i+j$ (j 随 T 中的 $j++$ 变, 而动)

S: aaa~~a~~cefg~~h~~ij

$j++$

T: aaac

如果不回溯的话就是从下一位开始比起:

aaa~~a~~cefg~~h~~ij

aaac

看到上面红颜色的没，如果不回溯的话，那么从 **a** 的下一位 **c** 比起。然而下述这种情况就漏了（正确的做法当然是要回溯：如果失配，即 $S[i + j] \neq T[j]$ ，需要让 $i++$ ，并且 $j = 0$ ）：

aaaacefghij

aaac

所以，BF 算法要回溯，其代码如下：

```
1.     int Index(SString S, SString T, int pos) {
2.         //返回 T 在 S 中第 pos 个字符之后的位置
3.         i=pos; j=1;k=0;
4.         while ( i <= S[0] && j <= T[0] ) {
5.             if (S[i+k] == T[j] ) {++k; ++j;} //继续比较后续字符
6.             else {i=i+1; j=1; k=0;} //指针回溯到 下一位，重新开始
7.         }
8.         if(j>T[0]) return i; //子串结束，说明匹配成功
9.         else return 0;
10.    } //Index
```

不过，也有特殊情况可以不回溯，如下：

abcdefghijkl(主串)

abcdefg(模式串)

即(模式串)没有相同的才不需要回溯。

4、KMP 算法思想

普通的字符串匹配算法必须要回溯。但回溯就影响了效率，回溯是由 **T** 串本身的性质决定的，是因为 **T** 串本身有前后'部分匹配'的性质。像上面所说如果主串为 **abcdef** 这样的，大没有回溯的必要。

改进的地方也就是这里，我们从 **T** 串本身出发，事先就找准了 **T** 自身前后部分匹配的位置，那就可以改进算法。

如果不用回溯，那模式串下一个位置从哪里开始呢？

还是上面那个例子，**T**(模式串)为 **ababc**，如果 **c** 失配，那就可以往前移到 **aba** 最后一个 **a** 的位置，像这样：

...ababd...

ababc

->ababc

这样 i 不用回溯， j 跳到前 2 个位置，继续匹配的过程，这就是 KMP 算法所在。这个当 $T[j]$ 失配后， j 应该往前跳的值就是 j 的 **next** 值，它是由 T 串本身固有决定的，与 S 串(主串)无关。

5、next 数组的含义

重点来了。下面解释一下 next 数组的含义，这个也是 KMP 算法中比较不好理解的一点。

令原始串为: $S[i]$, 其中 $0 \leq i \leq n$; 模式串为: $T[j]$, 其中 $0 \leq j \leq m$ 。

假设目前匹配到如下位置

$S_0, S_1, S_2, \dots, S_{i-j}, S_{i-j+1}, \dots, S_{i-1}, S_i, S_{i+1}, \dots, S_n$

$T_0, T_1, \dots, T_{j-1}, T_j, \dots$

S 和 T 的绿色部分匹配成功，恰好到 S_i 和 T_j 的时候失配，如果要保持 i 不变，同时达到让模式串 T 相对于原始串 S 右移的话，可以更新 j 的值，让 S_i 和新的 T_j 进行匹配，假设新的 j 用 $next[j]$ 表示，即让 S_i 和 $next[j]$ 匹配，显然新的 j 值要小于之前的 j 值，模式串才会是右移的效果，也就是说应该有 $next[j] \leq j - 1$ 。那新的 j 值也就是 $next[j]$ 应该是多少呢？我们观察如下的匹配：

1) 如果模式串右移 1 位（从简单的思考起，移动一位会怎么样），即 $next[j] = j - 1$ ，即让蓝色的 S_i 和 T_{j-1} 匹配(注：省略号为未匹配部分)

$S_0, S_1, S_2, \dots, S_{i-j}, S_{i-j+1}, \dots, S_{i-1}, S_i, S_{i+1}, \dots, S_n$

$T_0, T_1, \dots, T_{j-2}, T_{j-1}, T_j, \dots$ (T 的划线部分和 S 划线部分相等
【1】)

$T_0, T_1, \dots, T_{j-2}, T_{j-1}, \dots$ (移动后的 T 的划线部分和 S 的划线部分相等 【2】)

根据【1】【2】可以知道当 $\text{next}[j] = j - 1$, 即模式串右移一位的时候, 有 $T[0 \sim j-2] == T[1 \sim j-1]$, 而这两部分恰好是字符串 $T[0 \sim j-1]$ 的前缀和后缀, 也就是说 $\text{next}[j]$ 的值取决于模式串 T 中 j 前面部分的前缀和后缀相等部分的长度(好好揣摩这两个关键字概念: 前缀、后缀, 或者再想想, 我的上一篇文章, 从 Trie 树谈到后缀树中, 后缀树的概念)。

2)如果模式串右移 2 位, 即 $\text{next}[j] = j - 2$, 即让蓝色的 S_i 和 T_{j-2} 匹配

$S_0, S_1, \dots, \textcolor{blue}{S_{i-j}}, \textcolor{blue}{S_{i-j+1}}, \textcolor{blue}{S_{i-j+2}}, \dots, \textcolor{blue}{S_{i-1}}, \textcolor{blue}{S_i}, S_{i+1}, \dots, S_n$

$T_0, T_1, T_2, \dots, \textcolor{blue}{T_{j-3}}, \textcolor{blue}{T_{j-2}}, \dots, T_j, \dots, (T \text{ 的划线部分和 } S \text{ 划线部分相等})$ 【3】)

$T_0, T_1, \dots, \textcolor{blue}{T_{j-3}}, \textcolor{blue}{T_{j-2}}, \dots, (移动后的 T \text{ 的划线部分和 } S \text{ 的划线部分相等})$ 【4】)

同样根据【3】【4】可以知道当 $\text{next}[j] = j - 2$, 即模式串右移两位的时候, 有 $T[0 \sim j-3] == T[2 \sim j-1]$ 。而这两部分也恰好是字符串 $T[0 \sim j-1]$ 的前缀和后缀, 也就是说 $\text{next}[j]$ 的值取决于模式串 T 中 j 前面部分的前缀和后缀相等部分的长度。

3)依次类推, 可以得到如下结论: 当发生失配的情况下, j 的新值 $\text{next}[j]$ 取决于模式串中 $T[0 \sim j-1]$ 中前缀和后缀相等部分的长度, 并且 $\text{next}[j]$ 恰好等于这个最大长度。

为此, 请再允许我引用上文中的一段原文: “KMP 算法中, 如果当前字符匹配成功, 即 $S[i] == T[j]$, 令 $i++, j++$, 继续匹配下一个字符; 如果匹配失败, 即 $S[i] != T[j]$, 需要保持 i 不变, 并且让 $j = \text{next}[j]$, 这里 $\text{next}[j] <= j - 1$, 即模式串 T 相对于原始串 S 向右移动了至少 1 位(移动的实际位数 $j - \text{next}[j] >= 1$),

同时移动之后, i 之前的部分 (即 $S[i-j+1 \sim i-1]$), 和 $j=\text{next}[j]$ 之前的部分 (即 $T[0 \sim j-2]$) 仍然相等。显然, 相对于 BF 算法来说, KMP 移动更多的位数, 起到了一个加速的作用! (失配的特殊情形, 令 $j=\text{next}[j]$ 导致 $j==0$ 的时候, 需要将 $i++$, 否则此时没有移动模式串)。”

于此, 也就不难理解了我的关于 KMP 算法的第二篇文章之中: “当匹配到 $S[i] != P[j]$ 的时候有 $S[i-j \dots i-1] = P[0 \dots j-1]$. 如果下面用 j_next 去匹配, 则有 $P[0 \dots j_next-1] = S[i-j_next \dots i-1] = P[j-j_next \dots j-1]$ 。此过程如下图 3-1 所示。

当匹配到 $S[i] != P[j]$ 时, $S[i-j \dots i-1] = P[0 \dots j-1]$:

S: 0 ... $i-j$... $i-1$ $i \dots$

P: 0 ... j-1 j ...

如果下面用 j_{next} 去匹配，则有 $P[0 \dots j_{\text{next}}-1] = S[i-j_{\text{next}} \dots i-1] = P[j-j_{\text{next}} \dots j-1]$ 。

所以在 P 中有如下匹配关系（获得这个匹配关系的意义是用来求 next 数组）：

P: 0 ... j-1 j_{\text{next}} ... j-1 ...

P: 0 ... j j_{\text{next}-1} ...

所以，根据上面两个步骤，推出下一匹配位置 j_{next} :

S: 0 ... i-j ... i-j j_{\text{next}} ... i-1 ... i ...

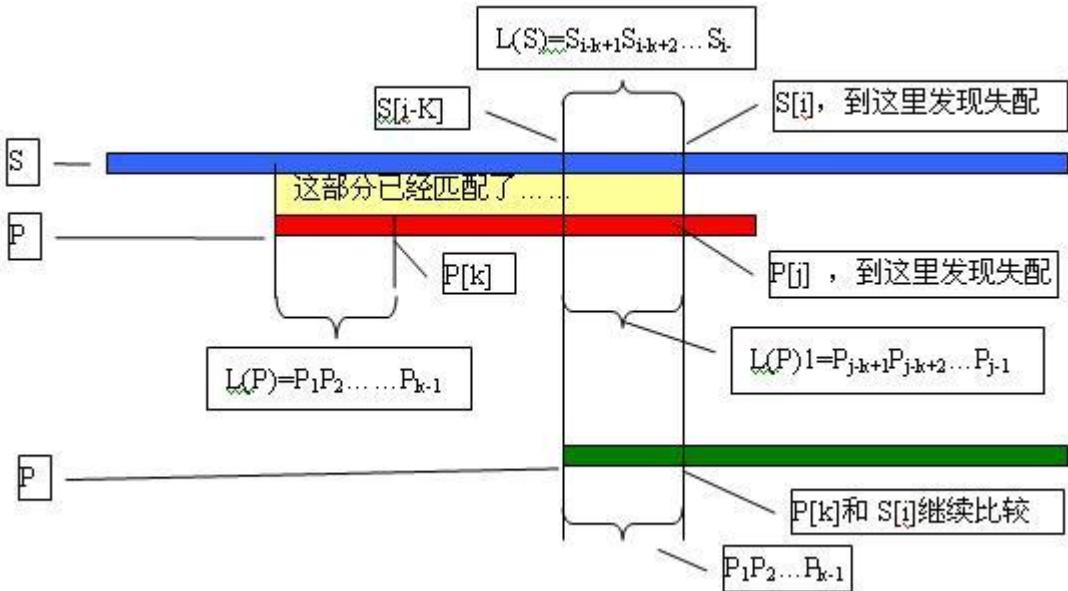
P: 0 ... j j_{\text{next}-1} j_{\text{next}} ...

图 3-1 求 j_{next} (最大的值) 的三个步骤

下面，我们用变量 k 来代表求得的 j_{next} 的最大值，即 k 表示这 $S[i], P[j]$ 不匹配时 P 中下一个用来匹配的位置，使得 $P[0 \dots k-1] = P[j-k \dots j-1]$ ，而我们要尽量找到这个 k 的最大值。”。

根据上文的【1】与【2】的匹配情况，可得第二篇文章之中所谓的 $k=1$ (如 $aaaa$ 的形式)，根据上文的【3】与【4】的匹配情况， $k=2$ (如 $abab$ 的形式)。

再次总结下，如下图：



从上图中我们看到，当 S 移动到 i , P 到 j 的时候失配。这时候 i 不回溯，而只是将 P 向前移动尽可能的距离，继续比较。

假设， P 向右移动一定距离后，第 k 个字符 $P[k]$ 和 $S[i]$ 进行比较。此时如上图，当 $P[j]$ 和 $S[i]$ 失配后， i 不动，将 P 前移到 K ，让 $P[k]$ 和 $S[i]$ 继续匹配。现在的关键是 K 的值是多少？

通过上图，我们发现，因为黄色部分表示已经匹配了的结果（因为是到了 $S[i]$ 和 $P[j]$ 的时候才

失配，所以 $S_{i-j+1}S_{i-j+2}\dots S_{i-1} = P_1P_2\dots P_{j-1}$ ，见黄色的部分）。所以有：

1、 $S_{i-k+1}S_{i-k+2}\dots S_{i-1} = P_{j-k+1}P_{j-k+2}\dots P_{j-1}$ 。

所以当 P 前移到 K 时，有：

2、 $S_{i-k+1}S_{i-k+2}\dots S_{i-1} = P_1P_2\dots P_{k-1}$ 。

通过 1, 2=>

$P_{j-k+1}P_{j-k+2}\dots P_{j-1} = P_1P_2\dots P_{k-1}$ 。

而 $P_1P_2\dots P_{k-1}$ 和 $P_{j-k+1}P_{j-k+2}\dots P_{j-1}$ 就相当于 P 串的前缀和后缀，前已说过，你心中一定要有前缀和后缀的概念或意识。

所以，归根究底，**KMP 算法的本质**便是：每一次匹配都是基于前一次匹配的结果，如何更好地利用这前一次匹配的结果呢？针对待匹配的模式串的特点，判断它是否有重复的字符，从而找到它的前缀与后缀，进而求出相应的 **Next** 数组，最终根据 **Next** 数组而进行 **KMP** 匹配。接下来，进入本文的第二部分。

第二部分、next 数组求法的来龙去脉与 KMP 算法的源码

本部分引自个人此前的关于 **KMP** 算法的第二篇文章：六之续、由 **KMP** 算法谈到 **BM** 算法。前面，我们已经知道即不能让 $P[j]=P[\text{next}[j]]$ 成立成立。不能再出现上面那样的情况啊！即不能有这种情况出现： $P[3]=b$ ，而竟也有 $P[\text{next}[3]]=P[1]=b$ 。

正如在第二篇文章中，所提到的那样：“这里读者理解可能有困难的是因为文中，时而 **next**，时而 **nextval**，把他们的思维搞混乱了。其实 **next** 用于表达数组索引，而 **nextval** 专用于表达 **next** 数组索引下的具体各值，区别细微。至于文中说不允许 $P[T]=P[\text{next}[j]]$ 出现，是因为已经有 $P[3]=b$ 与 $S[1]=b$ 匹配败，而 $P[\text{next}[3]]=P[1]=b$ ，若再拿 $P[1]=b$ 去与 $S[1]=b$ 匹配则必败。”--六之续、由 **KMP** 算法谈到 **BM** 算法。

又恰恰如上文中所述：“模式串 T 相对于原始串 S 向右移动了至少 1 位(**移动的实际位数 $j - \text{next}[j] >= 1$**)”。

ok，求 **next** 数组的 **get_nextval** 函数正确代码如下：

```
1.      //代码 4-1
2.      //修正后的求 next 数组各值的函数代码
3.      void get_nextval(char const* ptn, int plen, int* nextval)
4.      {
5.          int i = 0;
6.          nextval[i] = -1;
```

```

7.         int j = -1;
8.         while( i < plen-1 )
9.         {
10.             if( j == -1 || pptrn[i] == pptrn[j] ) //循环的 if 部分
11.             {
12.                 ++i;
13.                 ++j;
14.                 //修正的地方就发生下面这 4 行
15.                 if( pptrn[i] != pptrn[j] ) //++i, ++j 之后, 再次判断 pptrn[i]与 pptrn[j]
的关系
16.                     nextval[i] = j; //之前的错误解法就在于整个判断只有这一
句。
17.             else
18.                 nextval[i] = nextval[j];
19.             }
20.             else //循环的 else 部分
21.                 j = nextval[j];
22.             }
23.         }

```

举个例子，举例说明下上述求 `next` 数组的方法。

S a b a b a b c

P a b a b c

S[4] != P[4]

那么下一个和 S[4] 匹配的位置是 k=2(也即 P[next[4]])。此处的 k=2 也再次佐证了上文第 3 节开头处关于为了找到下一个匹配的位置时 k 的求法。上面的主串与模式串开头 4 个字符都是“abab”，所以，匹配失效后下一个匹配的位置直接跳两步继续进行匹配。

S a b a b a b c

P a b a b c

匹配成功

P 的 `next` 数组值分别为-1 0 -1 0 2

`next` 数组各值怎么求出来的呢？分以下五步：

1. 初始化： i=0, j=-1, `nextval[0] = -1`。由于 j == -1，进入上述循环的 if 部分，`++i` 得 i=1, `++j` 得 j=0，且 `pptrn[i] != pptrn[j]`(即 `a != b`)，所以得到第二个 `next` 值即 `nextval[1] = 0`；
2. i=1, j=0，进入循环 else 部分，`j=nextval[j]=nextval[0]=-1`；

3. 进入循环的 if 部分, $++i$, $++j$, $i=2$, $j=0$, 因为 $\text{ptrn}[i]=\text{ptrn}[j]=a$, 所以 $\text{nextval}[2]=\text{nextval}[0]=-1$;
4. $i=2$, $j=0$, 由于 $\text{ptrn}[i]=\text{ptrn}[j]$, 再次进入循环 if 部分, 所以 $++i=3$, $++j=1$, 因为 $\text{ptrn}[i]=\text{ptrn}[j]=b$, 所以 $\text{nextval}[3]=\text{nextval}[1]=0$;
5. $i=3, j=1$, 由于 $\text{ptrn}[i]=\text{ptrn}[j]=b$, 所以 $++i=4$, $++j=2$, 退出循环。

这样上例中模式串的 next 数组各值最终应该为:

a	b	a	b
-1	0	-1	0

图 4-1 正确的 next 数组各值

next 数组求解的具体过程如下:

初始化: $\text{nextval}[0] = -1$, 我们得到第一个 next 值即-1.

a	b	a	b
-1			

图 4-2 初始化第一个 next 值即-1

$i = 0$, $j = -1$, 由于 $j == -1$, 进入上述循环的 if 部分, $++i$ 得 $i=1$, $++j$ 得 $j=0$, 且 $\text{ptrn}[i] != \text{ptrn}[j]$ (即 $a != b$) , 所以得到第二个 next 值即 $\text{nextval}[1] = 0$;

a	b	a	b
-1	0		

图 4-3 第二个 next 值 0

上面我们已经得到, $i = 1$, $j = 0$, 由于不满足条件 $j == -1 || \text{ptrn}[i] == \text{ptrn}[j]$, 所以进入循环的 else 部分, 得 $j = \text{nextval}[j] = -1$; 此时, 仍满足循环条件, 由于 $i = 1$, $j = -1$, 因为 $j == -1$, 再次进入循环的 if 部分, $++i$ 得 $i=2$, $++j$ 得 $j=0$, 由于 $\text{ptrn}[i] == \text{ptrn}[j]$ (即 $\text{ptrn}[2]=\text{ptrn}[0]$,

也就是说第 1 个元素和第三个元素都是 a) , 所以进入循环 if 部分内嵌的 else 部分, 得到 nextval[2] = nextval[0] = -1;

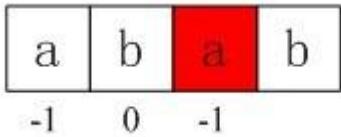


图 4-4 第三个 next 数组元素值-1

i = 2, j = 0, 由于 pptrn[i] == pptrn[j], 进入 if 部分, ++i 得 i=3, ++j 得 j=1, 所以 pptrn[i] == pptrn[j] (pptrn[3]==pptrn[1], 也就是说第 2 个元素和第 4 个元素都是 b) , 所以进入循环 if 部分内嵌的 else 部分, 得到 nextval[3] = nextval[1] = 0;

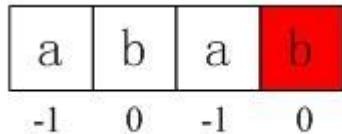


图 4-5 第四个数组元素值 0

如果你还是没有弄懂上述过程是怎么一回事, 请现在拿出一张纸和一支笔出来, 一步一步的画下上述过程。相信我, 把图画出来了之后, 你一定能明白它的。

然后, 我留一个问题给读者, 为什么上述的 next 数组要那么求?有什么原理么?

提示: 我们从上述字符串 abab 各字符的 next 值-1 0 -1 0, 可以看出来, 根据求得的 next 数组值, 偷用前缀、后缀的概念, 一定可以判断出在 abab 之中, 前缀和后缀相同, 即都是 ab, 反过来, 如果一个字符串的前缀和后缀相同, 那么根据前缀和后缀依次求得的 next 各值也是相同的。

• 5、利用求得的 next 数组各值运用 Kmp 算法

Ok, next 数组各值已经求得, 万事俱备, 东风也不欠了。接下来, 咱们就要应用求得的 next 值, 应用 KMP 算法来匹配字符串了。还记得 KMP 算法是怎么一回事吗?容我再次引用下之前的 KMP 算法的代码, 如下:

```
1.      //代码 5-1
2.      //int kmp_seach(char const*, int, char const*, int, int const*, int pos) KM
P 模式匹配函数
```

```

3.      //输入: src, slen 主串
4.      //输入: patn, plen 模式串
5.      //输入: nextval KMP 算法中的 next 函数值数组
6.      int kmp_search(char const* src, int slen, char const* patn, int plen, int co
    nst* nextval, int pos)
7.      {
8.          int i = pos;
9.          int j = 0;
10.         while ( i < slen && j < plen )
11.         {
12.             if( j == -1 || src[i] == patn[j] )
13.             {
14.                 ++i;
15.                 ++j;
16.             }
17.             else
18.             {
19.                 j = nextval[j];
20.                 //当匹配失败的时候直接用 p[j-next] 与 s[i] 比较,
21.                 //下面阐述怎么求这个值, 即匹配失效后下一次匹配的位置
22.             }
23.         }
24.         if( j >= plen )
25.             return i-plen;
26.         else
27.             return -1;
28.     }

```

我们上面已经求得的 next 值, 如下:

a	b	a	b
-1	0	-1	0

图 5-1 求得的正确的 next 数组元素各值

以下是匹配过程, 分三步:

第一步: 主串和模式串如下, S[3]与 P[3]匹配失败。

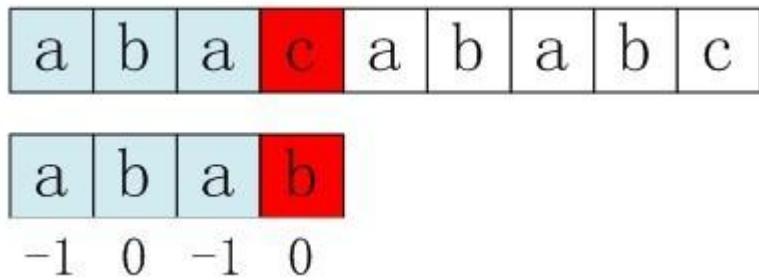


图 5-2 第一步， $S[3]$ 与 $P[3]$ 匹配失败

第二步： $S[3]$ 保持不变， P 的下一个匹配位置是 $P[\text{next}[3]]$ ，而 $\text{next}[3]=0$ ，所以 $P[\text{next}[3]]=P[0]$ ，即 $P[0]$ 与 $S[3]$ 匹配。在 $P[0]$ 与 $S[3]$ 处匹配失败。

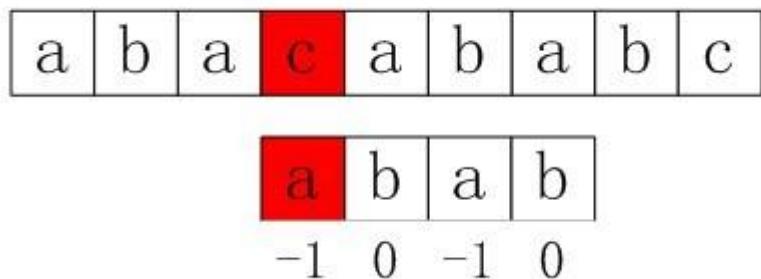


图 5-3 第二步，在 $P[0]$ 与 $S[3]$ 处匹配失败

第三步：与上文中第 3 小节末的情况一致。由于上述第三步中， $P[0]$ 与 $S[3]$ 还是不匹配。此时 $i=3, j=\text{nextval}[0]=-1$ ，由于满足条件 $j == -1$ ，所以进入循环的 if 部分， $++i=4, ++j=0$ ，即主串指针下移一个位置，从 $P[0]$ 与 $S[4]$ 处开始匹配。最后 $j==\text{plen}$ ，跳出循环，输出结果 $i-\text{plen}=4$ （即字串第一次出现的位置），匹配成功，算法结束。

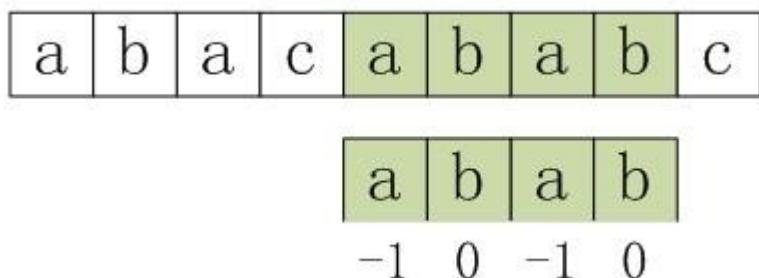


图 5-4 第三步，匹配成功，算法结束

所以，综上，总结上述三步为：

1. 开始匹配，直到 $P[3] \neq S[3]$ ，匹配失败；
2. $nextval[3]=0$ ，所以 $P[0]$ 继续与 $S[3]$ 匹配，再次匹配失败；
3. $nextval[0]=-1$ ，满足循环 if 部分条件 $j == -1$ ，所以， $++i$, $++j$ ，主串指针下移一个位置，从 $P[0]$ 与 $S[4]$ 处开始匹配，最后 $j==plen$ ，跳出循环，输出结果 $i-plen=4$ ，算法结束。

第三部分、KMP 算法的两种实现

代码实现一：

根据上文中第二部分内容的解析，完整写出 KMP 算法的代码已经不是难事了，如下：

```

1.      //copyright@2011 binghu and july
2.      #include "StdAfx.h"
3.      #include <string>
4.      #include <iostream>
5.      using namespace std;
6.
7.      //代码 4-1
8.      //修正后的求 next 数组各值的函数代码
9.      void get_nextval(char const* ptn, int plen, int* nextval)
10.     {
11.         int i = 0; //注，此处与下文的代码实现二不同的是，i 是从 0 开始的（代码实现二 i
从 1 开始）
12.         nextval[i] = -1;
13.         int j = -1;
14.         while( i < plen-1 )
15.         {
16.             if( j == -1 || ptn[i] == ptn[j] ) //循环的 if 部分
17.             {
18.                 ++i;
19.                 ++j;
20.                 //修正的地方就发生下面这 4 行
21.                 if( ptn[i] != ptn[j] ) //++i, ++j 之后，再次判断 ptn[i] 与 ptn[j]
的关系
22.                     nextval[i] = j; //之前的错误解法就在于整个判断只有这一
句。
23.             else
24.                 nextval[i] = nextval[j];
25.             }
26.             else //循环的 else 部分
27.                 j = nextval[j];

```

```

28.         }
29.     }
30.
31.     void print_progress(char const* src, int src_index, char const* pstr, int ps
tr_index)
32.     {
33.         cout<<src_index<<"\t"<<src<<endl;
34.         cout<<pstr_index<<"\t";
35.         for( int i = 0; i < src_index-pstr_index; ++i )
36.             cout<<" ";
37.         cout<<pstr<<endl;
38.         cout<<endl;
39.     }
40.
41. //代码 5-1
42. //int kmp_seach(char const*, int, char const*, int, int const*, int pos) KM
P 模式匹配函数
43. //输入: src, slen 主串
44. //输入: patn, plen 模式串
45. //输入: nextval KMP 算法中的 next 函数值数组
46. int kmp_search(char const* src, int slen, char const* patn, int plen, int co
nst* nextval, int pos)
47. {
48.     int i = pos;
49.     int j = 0;
50.     while ( i < slen && j < plen )
51.     {
52.         if( j == -1 || src[i] == patn[j] )
53.         {
54.             ++i;
55.             ++j;
56.         }
57.         else
58.         {
59.             j = nextval[j];
60.             //当匹配失败的时候直接用 p[j_next] 与 s[i] 比较,
61.             //下面阐述怎么求这个值, 即匹配失效后下一次匹配的位置
62.         }
63.     }
64.     if( j >= plen )
65.         return i-plen;
66.     else
67.         return -1;
68. }

```

```
69.
70.     int    main()
71. {
72.     std::string src = "aabcabcebafabcabceabcaefabcacdacab";
73.     std::string prn = "abac";
74.
75.     int* nextval = new int[prn.size()];
76.     //int* next = new int[prn.size()];
77.     get_nextval(prn.data(), prn.size(), nextval);
78.     //get_next(prn.data(), prn.size(), next);
79.
80.     for( int i = 0; i < prn.size(); ++i )
81.         cout<<nextval[i]<<"\t";
82.     cout<<endl;
83.
84.     cout<<"result sub str: "<<src.substr( kmp_search(src.data(), src.size(),
85.                                         prn.data(), prn.size(), nextval, 0) )<<endl;
86.     system("pause");
87.
88.     delete[] nextval;
89.     return 0;
}
```

运行结果，如下图所示：

The screenshot shows the Microsoft Visual Studio IDE interface. The title bar reads "algorithms (正在运行) - Microsoft Visual Studio". The menu bar includes File, Edit, View, VAssistX, Project, Build, Debug, Tools, Data, Windows, Analysis, View, and Help. The toolbar has various icons for file operations like Open, Save, and Build. The Solution Explorer on the left shows the project structure for "algorithms" with files like stdafx.h, targetver.h, algorithms.cpp, and stdafx.cpp. The code editor window displays the "algorithms.cpp" file with handwritten Chinese annotations explaining the logic of a string matching algorithm. The Output window on the right shows the results of a console application run, including command-line arguments and the output of the program.

```
int j = 0;
while ( i < slen && j < plen )
{
    if( j == -1 || src[i] == patn[j] )
    {
        ++i;
        ++j;
    }
    else
    {
        j = nextval[j];
        //当匹配失败的时候直接用p[j.next]与s[i]比较
        //下面阐述怎么求这个值，即匹配失效后下一次
    }
}
if( j >= plen )
    return i-plen;
else
    return -1;
}
```

代码实现二：

再给出代码实现二之前，让我们再次回顾下关于 KMP 算法的第一篇文章中的部分内容：

“第二节、KMP 算法

2.1、覆盖函数(overlay_function)

覆盖函数所表征的是 pattern 本身的性质，可以让为其表征的是 pattern 从左开始的所有连续子串的自我覆盖程度。比如如下的字串，abaabcaba

子串	值
a	-1
ab	-1
aba	0
abaa	0
abaab	1
abaabc	-1
abaabca	0
abaabcab	1
abaabcaba	2

可能上面的图令读者理解起来还是不那么清晰易懂，其实很简单，针对字符串 abaabcaba

a (-1) b (-1) a (0) a (0) b (1) c (-1) a (0) b (1) a (2)

解释：

1. 初始化为-1
2. b 与 a 不同为-1
3. 与第一个字符 a 相同为 0
4. 还是 a 为 0
5. 后缀 ab 与前缀 ab 两个字符相同为 1
6. 前面并无前缀 c 为-1
7. 与第一个字符同为 0
8. 后缀 ab 前缀 ab 为 1
9. 前缀 aba 后缀 aba 为 2

由于计数是从 0 开始的，因此覆盖函数的值为 0 说明有 1 个匹配，对于从 0 还是从来开始计数是偏好问题，具体请自行调整，其中-1 表示没有覆盖，那么何为覆盖呢，下面比较数学的来看一下定义，比如对于序列

$a_0 a_1 \dots a_{j-1} a_j$

要找到一个 k, 使它满足

$$a_0a_1\dots a_{k-1}a_k = a_{j-k}a_{j-k+1}\dots a_{j-1}a_j$$

而没有更大的 k 满足这个条件，就是说要找到尽可能大 k ，使 pattern 前 k 字符与后 k 字符相匹配， k 要尽可能的大，原因是如果有比较大的 k 存在。

但若我们选择较小的满足条件的 k ，那么当失配时，我们就会使 pattern 向右移动的位置变大，而较少的移动位置是存在匹配的，这样我们就会把可能匹配的结果丢失。比如下面的序列，

target	a	a	b	c	a	a	x	d	n	f	d
pattern	a	a	b	c	a	a	n				
$k=1$					a	a	b	c	a	a	
$k=0$						a	a	b	c	a	a

在红色部分失配，正确的结果是 $k=1$ 的情况，把 pattern 右移 4 位，如果选择 $k=0$ ，右移 5 位则会产生错误。计算这个 overlay 函数的方法可以采用递推，可以想象如果对于 pattern 的前 j 个字符，如果覆盖函数值为 k

$$a_0a_1\dots a_{k-1}a_k = a_{j-k}a_{j-k+1}\dots a_{j-1}a_j$$

则对于 pattern 的前 $j+1$ 序列字符，则有如下可能

- (1) $\text{pattern}[k+1] == \text{pattern}[j+1]$ 此时 $\text{overlay}(j+1) = k+1 = \text{overlay}(j)+1$
- (2) $\text{pattern}[k+1] \neq \text{pattern}[j+1]$ 此时只能在 pattern 前 $k+1$ 个字符组所的子串中找到相应的 overlay 函数， $h = \text{overlay}(k)$ ，如果此时 $\text{pattern}[h+1] == \text{pattern}[j+1]$ ，则 $\text{overlay}(j+1) = h+1$ 否则重复(2)过程。

下面给出一段计算覆盖函数的代码：

```

1.      //copyright@ staurman
2.      //updated@2011 July
3.      #include "StdAfx.h"
4.      #include<iostream>
5.      #include<string>
6.      using namespace std;
7.
8.      //solve to the next array
9.      void compute_overlay(const string& pattern)
10.     {
11.         const int pattern_length = pattern.size();
12.         int *overlay_function = new int[pattern_length];

```

```

13.         int index;
14.         overlay_function[0] = -1;
15.         for(int i=1;i<pattern_length;++i)
16.             //注，与上文代码段一不同的是，此处 i 是从 1 开始的，所以，下文中运用俩种方法求
出来的 next 数组各值会有所不同
17.         {
18.             index = overlay_function[i-1];
19.             //store previous fail position k to index;
20.
21.             while(index>=0 && pattern[i]!=pattern[index+1])
22.             {
23.                 index = overlay_function[index];
24.             }
25.             if(pattern[i]==pattern[index+1])
26.             {
27.                 overlay_function[i] = index + 1;
28.             }
29.             else
30.             {
31.                 overlay_function[i] = -1;
32.             }
33.         }
34.         for(int i=0;i<pattern_length;++i)
35.         {
36.             cout<<overlay_function[i]<<endl;
37.         }
38.         delete[] overlay_function;
39.     }
40.
41.     //abaabcaba
42.     int main()
43.     {
44.         string pattern = "abaabcaba";
45.         compute_overlay(pattern);
46.         system("pause");
47.         return 0;
48.     }

```

运行结果如下所示：

```
algorithms (正在运行) - Microsoft Visual Studio
文件(F) 编辑(E) 视图(V) VAssistX 项目(P) 生成(B) 调试(D) 团队(M) 数据(A) 工具(T) 体系结构(C) 测试(S) 分析(N) 窗口(W) 帮助(H)
cpp.cpp main.cpp index.cpp xstring_dbgmalloc.c xthrow.cpp algorithms.cpp
main int main()
{
    {
        overlay_function[i] = -1;
    }
}
for(int i=0;i<pattern_length;++i)
{
    cout<<overlay_function[i]<<endl;
}
delete[] overlay_function;
}

//abaabcbaba
int main()
{
    string pattern = "abaabcbaba";
    compute_overlay(pattern);
    system("pause");
    return 0;
}
```

1
1
0
0
1
1
0
1
2
请按任意键继续...

2.2、kmp 算法

有了覆盖函数，那么实现 kmp 算法就是很简单的了，我们的原则还是从左向右匹配，但是当失配发生时，我们不用把 `target_index` 向回移动，`target_index` 前面已经匹配过的部分在 `pattern` 自身就能体现出来，只要动 `pattern_index` 就可以了。

当发生在 j 长度失配时，只要把 `pattern` 向右移动 j -`overlay(j)` 长度就可以了。

如果失配时 `pattern_index==0`，相当于 `pattern` 第一个字符就不匹配，这时就应该把 `target_index` 加 1，向右移动 1 位就可以了。

ok，下图就是 KMP 算法的过程（红色即是采用 KMP 算法的执行过程）：

下面是kmp匹配过程图

目标字串annbcdanacadsannannabnna 匹配模式annacanna

移动序列	a n n b c d a n a c a d s a n n a n n a c a n n a
index=0	a n n a c a n n a
index=1	a n n a c a n n a
index=2	a n n a c a n n a
index=3	a n n a c a n n a
index=4	a n n a c a n n a
index=5	a n n a c a n n a
index=6	a n n a c a n n a
index=7	a n n a c a n n a
index=8	a n n a c a n n a
index=9	a n n a c a n n a
index=10	a n n a c a n n a
index=11	a n n a c a n n a
index=12	a n n a c a n n a
index=13	a n n a c a n n a
index=14	a n n a c a n n a
index=15	a n n a c a n n a
index=16	a n n a c a n n a

图中黄色标注的行为通过overlay函数跳过的行，绿色为匹配序列，红色为失配位置。

(另一作者 saturnman 发现，在上述 KMP 匹配过程图中，index=8 和 index=11 处画错了。
还有，anaven 也早已发现，index=3 处也画错了。非常感谢。但图已无法修改，见谅)

KMP 算法可在 $O(n+m)$ 时间内完成全部的串的模式匹配工作。”

OK，下面此前写的关于 KMP 算法的第一篇文章中的源码：

```
1. //copyright@ saturnman
2. //updated@ 2011 July
3. #include "stdafx.h"
4. #include<iostream>
5. #include<string>
6. #include <vector>
```

```

7.     using namespace std;
8.
9.     int kmp_find(const string& target,const string& pattern)
10.    {
11.        const int target_length=target.size();
12.        const int pattern_length=pattern.size();
13.        int* overlay_value=new int[pattern_length];
14.        overlay_value[0]=-1;           //remember:next array's first number was -1.
15.
16.        int index=0;
17.
18.        //next array
19.        for (int i=1;i<pattern_length;++i)
20.            //注，此处的 i 是从 1 开始的
21.        {
22.            index=overlay_value[i-1];
23.            while (index>=0 && pattern[index+1]!=pattern[i]) //remember:!=
24.            {
25.                index=overlay_value[index];
26.            }
27.            if(pattern[index+1] == pattern[i])
28.            {
29.                overlay_value[i]=index+1;
30.            }
31.            else
32.            {
33.                overlay_value[i]=-1;
34.            }
35.
36.            //match algorithm start
37.            int pattern_index=0;
38.            int target_index=0;
39.            while (pattern_index<pattern_length && target_index<target_length)
40.            {
41.                if (target[target_index] == pattern[pattern_index])
42.                {
43.                    ++target_index;
44.                    ++pattern_index;
45.                }
46.                else if(pattern_index==0)
47.                {
48.                    ++target_index;
49.                }

```

```

50.         else
51.         {
52.             pattern_index=overlay_value[pattern_index-1]+1;
53.         }
54.     }
55.     if (pattern_index==pattern_length)
56.     {
57.         return target_index-pattern_index;
58.     }
59.     else
60.     {
61.         return -1;
62.     }
63.     delete [] overlay_value;
64. }
65.
66. int main()
67. {
68.     string sourc="ababc";
69.     string pattern="abc";
70.     cout<<kmp_find(sourc,pattern)<<endl;
71.     system("pause");
72.     return 0;
73. }
```

由于是 abc 跟 ababc 匹配，那么将返回匹配的位置“2”，运行结果如所示：

The screenshot shows the Microsoft Visual Studio IDE interface. The title bar reads "cpp (正在运行) - Microsoft Visual Studio". The menu bar includes File, Edit, View, VAssistX, Project, Build, Debug, Team, Data, Tools, Architecture, Test, Analyze, Window, Help. The toolbar has icons for New, Open, Save, Print, etc. The solution explorer shows files main.cpp, xstring, xthrow.cpp, and cpp.cpp*. The cpp.cpp file is open in the code editor, displaying the following C++ code:

```
int kmp_find(const string& target, const string& pattern)
{
    const int target_length=target.size();
    const int pattern_length=pattern.size();
    int* overlay_value=new int[pattern_length];
    overlay_value[0]=-1; //remember:next array's first number was -1.
    int index=0;

    //next array
    for (int i=1;i<pattern_length;++i)
        //注，此处的i是从1开始的
    {
        index=overlay_value[i-1];
        while (index>=0 && pattern[index+1]!=pattern[i]) //remember:i-
        {
            index=overlay_value[index];
        }
        if(pattern[index+1] == pattern[i])
        {
            overlay_value[i]=index+1;
        }
        else
        {
            overlay_value[i]=-1;
        }
    }

    //match algorithm start
    int pattern_index=0;
}
```

The status bar at the bottom shows "自动窗口" and "就绪". A message box is visible in the top right corner with the text "2 请按任意键继续...".

第四部分、测试

针对上文中第三部分的两段代码测试了下，纠结了，两种求 next 数组的方法对同一个字符串求 next 数组各值，得到的结果竟然不一样，如下二图所示：

1、两种方法对字符串 **abab** 求 next 数组各值比较（下图左边为代码实现一内求 next 数组方法的结果，右边为代码实现二内求 next 数组方法的结果）：

The screenshot shows two Microsoft Visual Studio windows side-by-side. The left window displays the file `main.cpp` with the following code:

```
if( j >= plen )
    return i-plen;
else
    return -1;
}

int main()
{
    std::string src = "ababa";
    std::string prn = "abab";

    int* nextval = new int[prn.size()];
    //int* next = new int[prn.size()];
    get_nextval(prn.data(), prn.size(), nextval);
    //get_next(prn.data(), prn.size(), next);

    for( int i = 0; i < prn.size(); i++ )
        cout<<nextval[i]<<"\t";
    cout<<endl;

    cout<<"result sub str: "<<src;
    system("pause");

    delete[] nextval;
    return 0;
}
```

The right window displays the file `algorithms.cpp` with the following code:

```
else
{
    overlay_function[i] = -1;
}
for(i=0;i<pattern_length;++i)
{
    cout<<overlay_function[i]<<endl;
}
delete[] overlay_function;
}

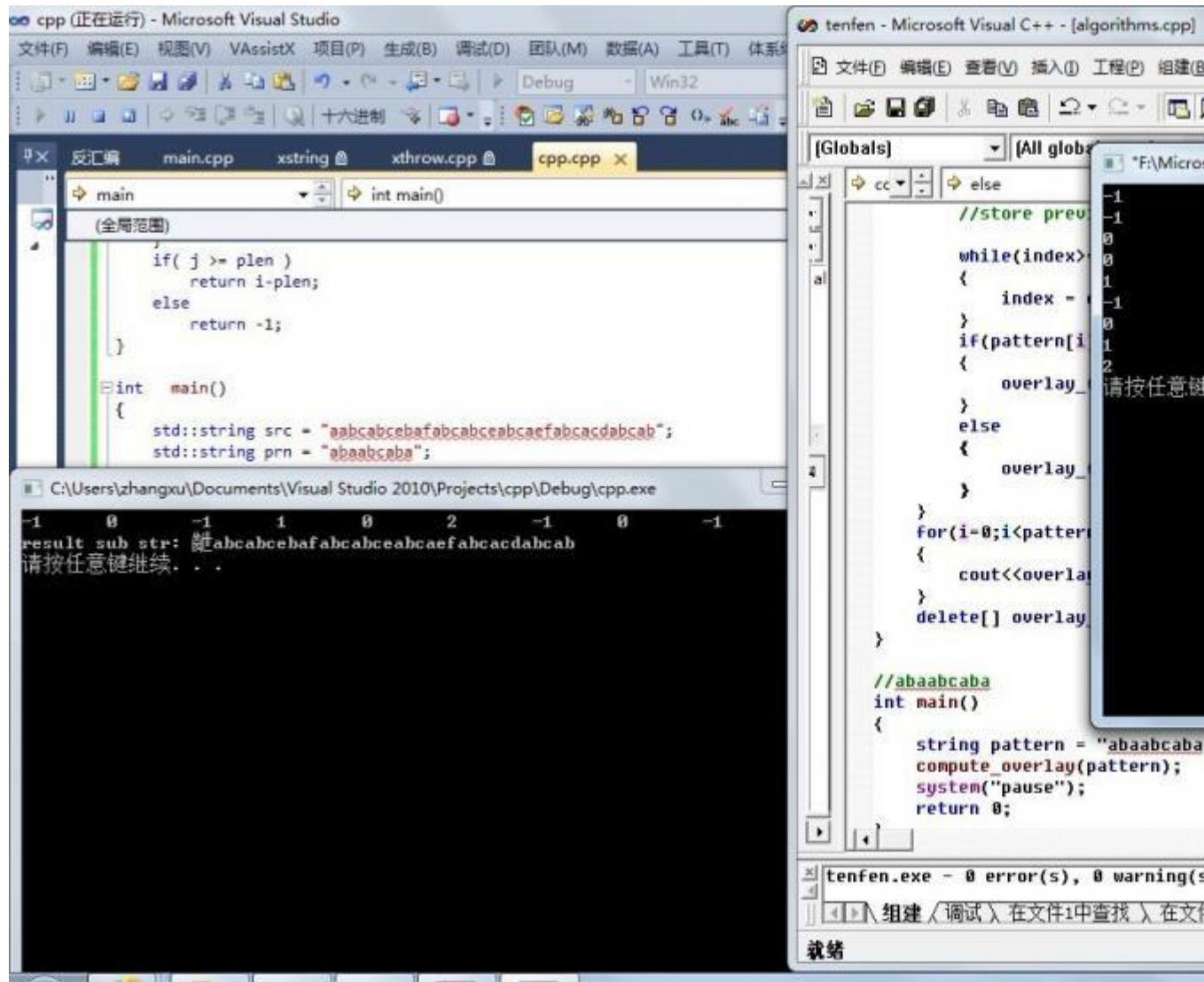
//abab
int main()
{
    string pattern = "abab";
    compute_overlay(pattern);
    system("pause");
    return 0;
}
```

The output window at the bottom shows the results of the program execution:

```
-1     0     -1     0
result sub str: ababa
请按任意键继续... .
```

The status bar at the bottom indicates: `tenfen.exe - 0 error(s), 0 warning(s)`.

2、两种对字符串 **abaabcaba** 求 next 数组各值比较（下图左边为代码实现一内求 next 数组方法的结果，右边为代码实现二内求 next 数组方法的结果）：



为何会这样呢，其实很简单，上文中已经有所说明了，**代码实现一的 i 是从 0 开始的，代码实现二的 i 是从 1 开始的**。但从最终的运行结果来看，暂时还是以代码实现段二为准。

第五部分、KMP 完整准确源码

求 next 数组各值的方法为：

```
1.          //copyright@ staurman
2.          //updated@2011 July
3.          #include "StdAfx.h"
4.          #include<iostream>
5.          #include<string>
6.          using namespace std;
```

```

7.
8.     //solve to the next array
9.     void compute_overlay(const string& pattern)
10.    {
11.        const int pattern_length = pattern.size();
12.        int *overlay_function = new int[pattern_length];
13.        int index;
14.        overlay_function[0] = -1;
15.        for(int i=1;i<pattern_length;++i)
16.        {
17.            index = overlay_function[i-1];
18.            //store previous fail position k to index;
19.
20.            while(index>=0 && pattern[i]!=pattern[index+1])
21.            {
22.                index = overlay_function[index];
23.            }
24.            if(pattern[i]==pattern[index+1])
25.            {
26.                overlay_function[i] = index + 1;
27.            }
28.            else
29.            {
30.                overlay_function[i] = -1;
31.            }
32.        }
33.        for(int i=0;i<pattern_length;++i)
34.        {
35.            cout<<overlay_function[i]<<endl;
36.        }
37.        delete[] overlay_function;
38.    }
39.
40.    //abaabcbaba
41.    int main()
42.    {
43.        string pattern = "abaabcbaba";
44.        compute_overlay(pattern);
45.        system("pause");
46.        return 0;
47.    }

```

运行结果如下图所示：abab 的 next 数组各值是 -1, -1, 0, 1，而非本文第二部分所述的 -1, 0, -1, 0。为什么呢？难道是搬石头砸了自己的脚？

NO，上文第四部分末已经详细说明，上处代码 i 从 0 开始，本文第二部分代码 i 从 1 开始。

The screenshot shows the Microsoft Visual Studio IDE running a project named 'cpp'. The main window displays the 'main.cpp' file with the following code:

```
while(index>=0 && pattern[i]!=pattern[index+1])
{
    index = overlay_function[index];
}
if(pattern[i]==pattern[index+1])
{
    overlay_function[i] = index + 1;
}
else
{
    overlay_function[i] = -1;
}
for(int i=0;i<pattern_length;++i)
{
    cout<<overlay_function[i]<<endl;
}
delete[] overlay_function;
```

Below the code, there is a note: //ababcbaba. The main function is defined as:

```
int main()
{
    string pattern = "abab";
    compute_overlay(pattern);
    system("pause");
    return 0;
}
```

To the right of the code editor, a terminal window shows the output of the program:

```
-1
-1
0
1
请按任意键继续...
```

KMP 算法完整源码，如下：

```
1.      //copyright@ saturnman
2.      //updated@ 2011 July
3.      #include "stdafx.h"
4.      #include<iostream>
5.      #include<string>
```

```

6.     #include <vector>
7.     using namespace std;
8.
9.     int kmp_find(const string& target,const string& pattern)
10.    {
11.        const int target_length=target.size();
12.        const int pattern_length=pattern.size();
13.        int* overlay_value=new int[pattern_length];
14.        overlay_value[0]=-1;           //remember:next array's first number was -1.
15.
16.        int index=0;
17.
18.        //next array
19.        for (int i=1;i<pattern_length;++i)
20.            //注，此处的 i 是从 1 开始的
21.        {
22.            index=overlay_value[i-1];
23.            while (index>=0 && pattern[index+1]!=pattern[i])
24.            {
25.                index=overlay_value[index];
26.            }
27.            if(pattern[index+1] == pattern[i])
28.            {
29.                overlay_value[i]=index+1;
30.            }
31.            else
32.            {
33.                overlay_value[i]=-1;
34.            }
35.
36.            //mach algorithm start
37.            int pattern_index=0;
38.            int target_index=0;
39.            while (pattern_index<pattern_length && target_index<target_length)
40.            {
41.                if (target[target_index] == pattern[pattern_index])
42.                {
43.                    ++target_index;
44.                    ++pattern_index;
45.                }
46.                else if(pattern_index==0)
47.                {
48.                    ++target_index;

```

```
49.         }
50.     else
51.     {
52.         pattern_index=overlay_value[pattern_index-1]+1;
53.     }
54. }
55. if (pattern_index==pattern_length)
56. {
57.     return target_index-pattern_index;
58. }
59. else
60. {
61.     return -1;
62. }
63. delete [] overlay_value;
64. }
65.
66. int main()
67. {
68.     string sourc="ababc";
69.     string pattern="abc";
70.     cout<<kmp_find(sourc,pattern)<<endl;
71.     system("pause");
72.     return 0;
73. }
```

运行结果如下：

```
int kmp_find(const string& target,const string& pattern)
{
    const int target_length=target.size();
    const int pattern_length=pattern.size();
    int* overlay_value=new int[pattern_length];
    overlay_value[0]=-1; //remember:next array's first number was -1.
    int index=0;

    //next array
    for (int i=1;i<pattern_length;++i)
        //注，此处的i是从1开始的
    {
        index=overlay_value[i-1];
        while (index>=0 && pattern[index+1]!=pattern[i])
        {
            index=overlay_value[index];
        }
        if(pattern[index+1] == pattern[i])
        {
            overlay_value[i]=index+1;
        }
        else
        {
            overlay_value[i]=-1;
        }
    }

    //match algorithm start
    int pattern_index=0;
}
```

第六部分、一眼看出字符串的 next 数组各值

上文已经用程序求出了一个字符串的 next 数组各值，接下来，稍稍演示下，如何一眼大致判断出 next 数组各值，以及初步判断某个程序求出的 next 数组各值是不是正确的。有一点务必注意：下文中的代码全部采取代码实现二，即 i 是从 1 开始的。

- 1、对字符串 aba 求 next 数组各值，各位可以先猜猜，-1, ..., aba 中，a 初始化为 -1，第二个字符 b 与 a 不同也为-1，最后一个字符和第一个字符都是 a，所以，我猜其 next 数组各值应该是-1, -1, 0，结果也不出所料，如下图所示：

```
index = overlay_function[index];
    }
    if(pattern[i]==pattern[index+1])
    {
        overlay_function[i] = index + 1;
    }
    else
    {
        overlay_function[i] = -1;
    }
}
for(int i=0;i<pattern_length;++i)
{
    cout<<overlay_function[i]<<endl;
}
delete[] overlay_function;
}

//abaabcbaba
int main()
{
    //i是从1开始的
    string pattern = "aba";
    compute_overlay(pattern);
    system("pause");
    return 0;
}
```

-1
-1
0
请按任意键继续...

- 2、字符串“abab”呢，不用猜了，我已经看出来了，当然上文中代码实现一和代码实现二都已经求出来了。如果 i 是 1 开始的话，那么 $next$ 数组各值将如代码实现二所运行的那样，将是：-1, -1, 0, 1;
- 3、字符串“abaabcaba”呢， $next$ 数组如上第三部分代码实现二所述，为-1, -1, 0, 0, 1, -1, 0, 1, 2;
- 4、字符串“abcdab”呢， $next$ 数组各值将是-1, -1, -1, -1, 0, 1;
- 5、字符串“abcdabc”呢， $next$ 数组各值将是-1, -1, -1, -1, 0, 1, 2;
- 6、字符串“abcdabcd”呢，那么 $next$ 数组各值将是-1, -1, -1, -1, 0, 1, 2, 3;

```
void compute_overlay(const string& pattern)
{
    const int pattern_length = pattern.size();
    int *overlay_function = new int[pattern_length];
    int index;
    overlay_function[0] = -1;
    for(int i=1;i<pattern_length;++i) //i是从1开始的
    {
        index = overlay_function[i-1];
        //store previous fail position k to index;

        while(index>=0 && pattern[i]!=pattern[index+1])
        {
            index = overlay_function[index];
        }
        if(pattern[i]==pattern[index+1])
        {
            overlay_function[i] = index + 1;
        }
        else
        {
            overlay_function[i] = -1;
        }
    }
    for(int i=0;i<pattern_length;++i)
    {
        cout<<overlay_function[i]<<endl;
    }
    delete[] overlay_function;
}

int main()
{
    string pattern = "abcdabcd"; // -1 -1 -1 -1 0 1 2 3
}
```

怎么样，看出规律来了没？呵呵，可以用上述第五部分中求 next 数组的方法自个多试探几次，相信，很快，你也会跟我一样，不用计算，一眼便能看出某个字符串的 next 数组各值了。如此便恭喜你，理解了 next 数组的求法，KMP 算法也就算是真真正正彻彻底底的理解了（至于如何运用求得的 next 数组各值来进行 kmp 算法的匹配的具体方法与过程，请转到本文第二部分。不过，需要你注意的是，本文第二部分的 i 是从 0 开始的）。完。

相关链接

1. KMP 之第二篇文章：[六（续）、从 KMP 算法一步一步谈到 BM 算法。](#)
2. KMP 之第一篇文章：[六、教你初步了解 KMP 算法、updated。](#)

我的微博

在结束全文之前，引用下自个微博上（@周磊 July, <http://weibo.com/julyweibo>）的两

段话：

1. 语言->数据结构->算法：语言是基础，够啃一辈子，基本的常见的数据结构得了如指掌，最后才是算法。除了算法之外，有更多更重要且更值得学习的东西（最重要的是，学习如何编程）。切勿盲目跟风，找准自己的兴趣点，和领域才是关键。这跟选择职位、与领域并持久做下去，比选择公司更重要一样。选择学什么东西不重要，重要的是你的兴趣。
2. 修订这篇文章之时，个人接触 KMP 都有一年了，学算法也刚好快一年。想想阿，我弄一个 KMP，弄了近一年了，到今天才算是真正彻底理解其思想，可想而知，当初创造这个算法的 k、m、p 三人是何等不易。我想，有不少读者是因为我的出现而想学算法的，但不可急功近利，切勿妄想算法速成。早已说过，学算法先修心。

以下是发自本人微博上的对此书：MySQL 性能调优与架构设计，简朝阳著，做的读书笔记，聊做书斋录，以供闲时翻翻：

1. Hash 索引在 MySQL 中使用并不多，目前在 Memory 和 NDB Cluster 存储引擎使用。所谓 Hash 索引，实际上就是通过一定的 Hash 算法，将须要索引的键值进行 Hash 运算，然后将得到的 Hash 值存入 Hash 表中。检索时，根据 Hash 表中的 Hash 值逆 Hash 运算反馈原键值；
2. InnoDB 存储引擎的 B-Tree 索引使用的存储结构实际上是 B+Tree，在 B-Tree 的基础上做了很小的改造，在每一个 LeafNode 上除了存放索引键的相关信息，还存储了指向与该 LeafNode 相邻的后一个 LeafNode 的指针，此举为了加快检索多个相邻 LeafNode 的效率；
3. 在我的那篇[从 B 树、B+树、B*树谈到 R 树](#)的文章中介绍到了 B 树与 B+树的差别，B+树的叶子节点中除了跟 B 树一样包含了关键字的信息之外，还包含了指向相邻叶子节点的指针，如此，叶子节点之间就有了联系、有序了。而 B*树则更进一步，兄弟节点间指针；
4. 无处不透露着数据结构、与算法思想，数据库也不例外。尤其当涉及到数据库性能优化，则更是如此；
5. 又喝了半碗白酒，吃完火锅，叼根烟，同学的手艺实在太好了。来北京初带的钱也即将马上用完了，工作一时还无法定。多亏了同学。再趁着微微酒力，提个问题：我们知道，Hash 索引的效率比 B-Tree 高很多，而为什么大家都不用 Hash 索引而还要使用 B-Tree 索引呢？稳定？你能说出几个原因呢？

后记

相信，看过此文后，无论是谁，都一定可以把 KMP 算法搞懂了（但万一还是有读者没有搞懂，那怎么办呢？还有最后一个办法：把本文打印下来，再仔细琢磨。如果是真真正正想彻底弄懂某一个东西，那么必须付出些代价。但万一要是打印下来了却还是没有弄懂呢？那就来北京找我吧，我手把手教你。祝好运）。

OK，扯远了。本文文中有关任何问题或错误，烦请不吝赐教与指正。谢谢，完。

July、二零一一年十二月五日中午。

七、遗传算法初探

---深入浅出、透析 GA 本质

作者:July 二零一一年一月十二日。

本文参考：维基百科 华南理工大学电子讲义 互联网

一、初探遗传算法

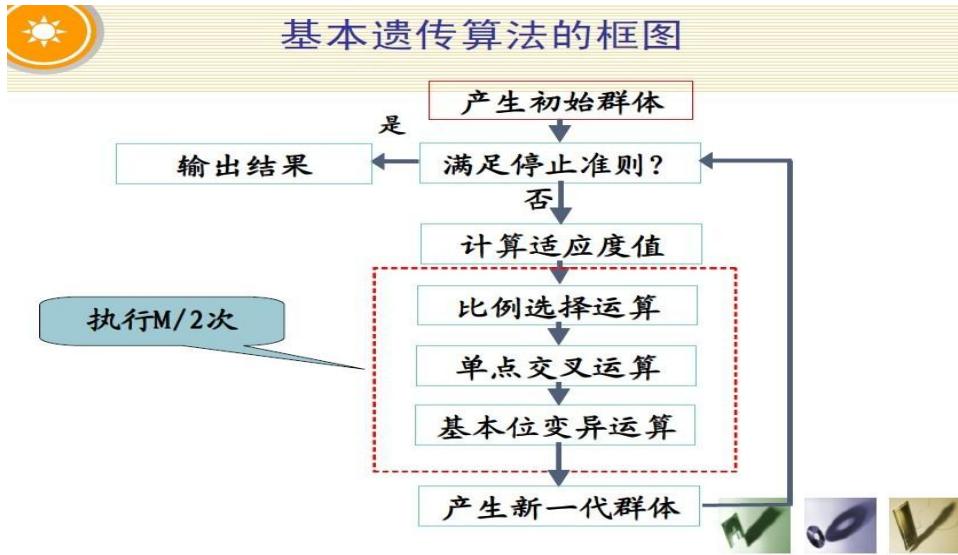
Ok，先看维基百科对遗传算法所给的解释：

遗传算法是计算数学中用于解决最优化的搜索算法，是进化算法的一种。进化算法最初是借鉴了进化生物学中的一些现象而发展起来的，这些现象包括遗传、突变、自然选择以及杂交等。

遗传算法通常实现方式为一种计算机模拟。对于一个最优化问题，一定数量的候选解（称为个体）的抽象表示（称为染色体）的种群向更好的解进化。传统上，解用二进制表示（即 0 和 1 的串），但也可以用其他表示方法。进化从完全随机个体的种群开始，之后一代一代发生。在每一代中，整个种群的适应度被评价，从当前种群中随机地选择多个个体（基于它们的适应度），通过自然选择和突变产生新的生命种群，该种群在算法的下一次迭代中成为当前种群。

光看定义，可能思路并不清晰，咱们来几个清晰的图解、步骤、公式：

基本遗传算法的框图：



所以，遗传算法基本步骤是：

- 1) 初始化 $t \leftarrow 0$ 进化代数计数器； T 是最大进化代数；随机生成 M 个个体作为初始群体 $P(t)$ ；
- 2) 个体评价 计算 $P(t)$ 中各个个体的适应度值；
- 3) 选择运算 将选择算子作用于群体；
- 4) 交叉运算 将交叉算子作用于群体；
- 5) 变异运算 将变异算子作用于群体，并通过以上运算得到下一代群体 $P(t+1)$ ；
- 6) 终止条件判断 $t \leq T$: $t \leftarrow t+1$ 转到步骤 2;
 $t > T$: 终止 输出解。

好的，看下遗传算法的伪代码实现：

▲Procedures GA: 伪代码

```

begin
    initialize P(0);
    t = 0; //t 是进化的代数，一代、二代、三代...
    while(t <= T) do
        for i = 1 to M do //M 是初始种群的个体数
            Evaluate fitness of P(t); //计算 P(t) 中各个个体
            的适应度
        end for
        for i = 1 to M do
            Select operation to P(t); //将选择算子作用于群体
        end for
        for i = 1 to M/2 do
            Crossover operation to P(t); //将交叉算子作用于群体
        end for
        for i = 1 to M do
            Mutation operation to P(t); //将变异算子作用于群体
        end for
    end

```

```

for i = 1 to M do
    P(t+1) = P(t); //得到下一代群体 P
    (t + 1)
    end for
    t = t + 1; //终止条件判断 t≤T: t← t+1 转到步骤 2
end while
end

```

二、深入遗传算法

1、智能优化算法概述

智能优化算法又称现代启发式算法，是一种具有全局优化性能、通用性强且适合于并行处理的算法。

这种算法一般具有严密的理论依据，而不是单纯凭借专家经验，理论上可以在一定的时间内找到最优解或近似最优解。

遗传算法属于智能优化算法之一。

常用的智能优化算法有：

遗传算法、模拟退火算法、禁忌搜索算法、粒子群算法、蚁群算法。

(本经典算法研究系列，日后将陆续阐述模拟退火算法、粒子群算法、蚁群算法。)

2、遗传算法概述

遗传算法是由美国的 J. Holland 教授于 1975 年在他的专著《自然界和人工系统的适应性》中首先提出的。

借鉴生物界自然选择和自然遗传机制的随机化搜索算法。

模拟自然选择和自然遗传过程中发生的繁殖、交叉和基因突变现象。

在每次迭代中都保留一组候选解，并按某种指标从解群中选取较优的个体，利用遗传算子(选择、交叉和变异)对这些个体进行组合，产生新一代的候选解群，重复此过程，直到满足某种收敛指标为止。

基本遗传算法 (Simple Genetic Algorithms, GA)又称简单遗传算法或标准遗传算法)，是由 Goldberg 总结出的一种最基本的遗传算法，其遗传进化操作过程简单，容易理解，是其它一些遗传算法的雏形和基础。

3、基本遗传算法的组成

- (1) 编码 (产生初始种群)
- (2) 适应度函数
- (3) 遗传算子 (选择、交叉、变异)
- (4) 运行参数

接下来，咱们分门别类，分别阐述着基本遗传算法的五个组成部分：

1、编码

遗传算法（GA）通过某种编码机制把对象抽象为由特定符号按一定顺序排成的串。正如研究生物遗传是从染色体着手，而染色体则是由基因排成的串。基本遗传算法（SGA）使用二进制串进行编码。
初始种群：基本遗传算法（SGA）采用随机方法生成若干个个体的集合，该集合称为初始种群。
初始种群中个体的数量称为种群规模。

2、适应度函数

遗传算法对一个个体（解）的好坏用适应度函数值来评价，适应度函数值越大，解的质量越好。

适应度函数是遗传算法进化过程的驱动力，也是进行自然选择的唯一标准，它的设计应结合求解问题本身的要求而定。

3.1、选择算子

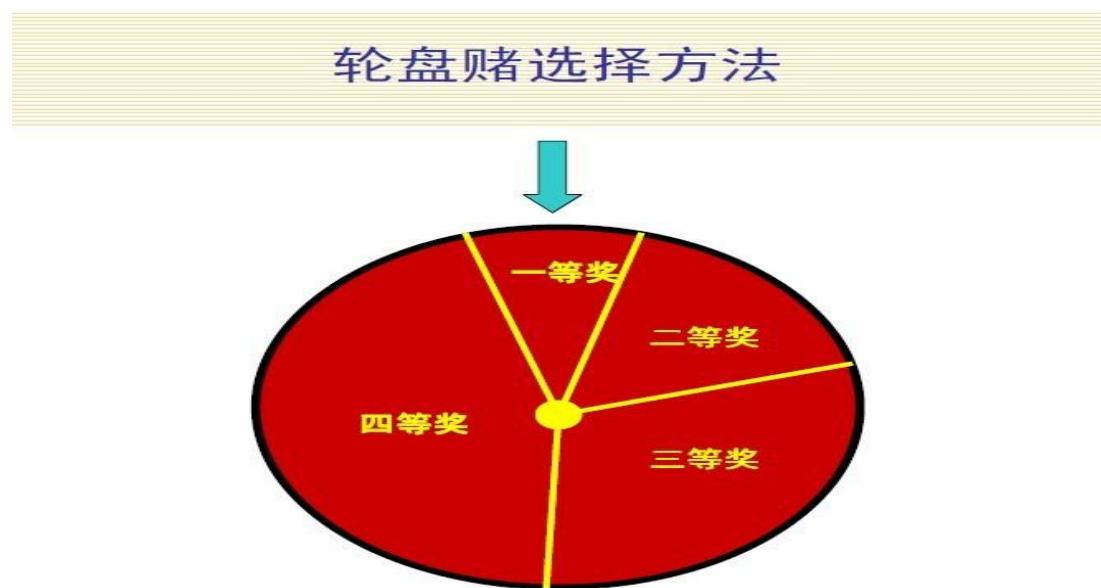
遗传算法使用选择运算对个体进行优胜劣汰操作。

适应度高的个体被遗传到下一代群体中的概率大；适应度低的个体，被遗传到下一代群体中的概率小。

选择操作的任务就是从父代群体中选取一些个体，遗传到下一代群体。

基本遗传算法（SGA）中选择算子采用轮盘赌选择方法。

Ok，下面就来看下这个[轮盘赌](#)的例子，这个例子通俗易懂，[对理解选择算子帮助很大](#)。



轮盘赌选择方法

轮盘赌选择又称比例选择算子，其基本思想是：
各个个体被选中的概率与其适应度函数值大小成正比。

设群体大小为 N，个体 x_i 的适应度为 $f(x_i)$ ，则个体 x_i 的选择概率为：

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^N f(x_j)}$$

轮盘赌选择法可用如下过程模拟来实现：

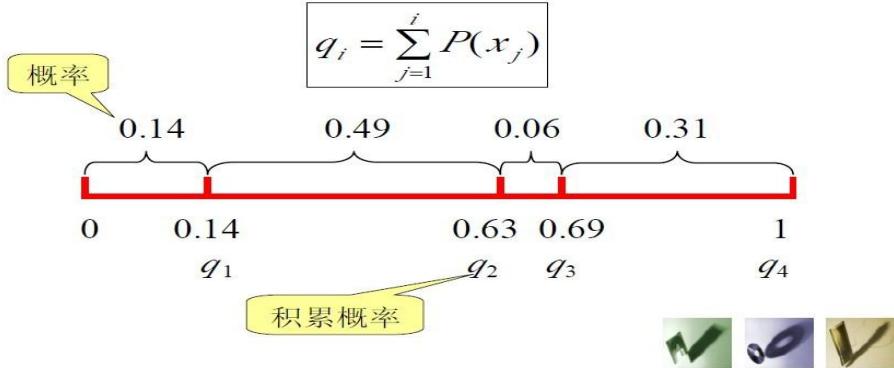
- (1) 在 $[0, 1]$ 内产生一个均匀分布的随机数 r 。
- (2) 若 $r \leq q_1$, 则染色体 x_1 被选中。
- (3) 若 $q_{k-1} < r \leq q_k$ ($2 \leq k \leq N$), 则染色体 x_k 被选中。

其中的 q_i 称为染色体 x_i ($i=1, 2, \dots, n$) 的累积概率，其计算公式为：

$$q_i = \sum_{j=1}^i P(x_j)$$

累积概率实例：

累积概率实例：



轮盘赌选择方法的实现步骤：

- (1) 计算群体中所有个体的适应度值；
- (2) 计算每个个体的选择概率；
- (3) 计算累积概率；
- (4) 采用模拟赌盘操作（即生成 0 到 1 之间的随机数与每个个体遗传到下一代群体的概率进行匹配）来确定各个个体是否遗传到下一代群体中。

例如，有染色体

$s1 = 13$ (01101)

$s2 = 24$ (11000)

s3= 8 (01000)

s4= 19 (10011)

假定适应度为 $f(s) = s^2$ ，则

$$f(s_1) = f(13) = 13^2 = 169$$

$$f(s_2) = f(24) = 24^2 = 576$$

$$f(s_3) = f(8) = 8^2 = 64$$

$$f(s_4) = f(19) = 19^2 = 361$$

染色体的选择概率为：

染色体的选择概率为

$$P(s_1) = \frac{f(s_1)}{\sum_{j=1}^N f(s_j)} = \frac{169}{169 + 576 + 64 + 361} = 0.14$$

$$P(s_2) = \frac{f(s_2)}{\sum_{j=1}^N f(s_j)} = \frac{576}{169 + 576 + 64 + 361} = 0.49$$

$$P(s_3) = \frac{f(s_3)}{\sum_{j=1}^N f(s_j)} = \frac{64}{169 + 576 + 64 + 361} = 0.06$$

$$P(s_4) = \frac{f(s_4)}{\sum_{j=1}^N f(s_j)} = \frac{361}{169 + 576 + 64 + 361} = 0.31$$

染色体的累计概率为：

染色体的累计概率为

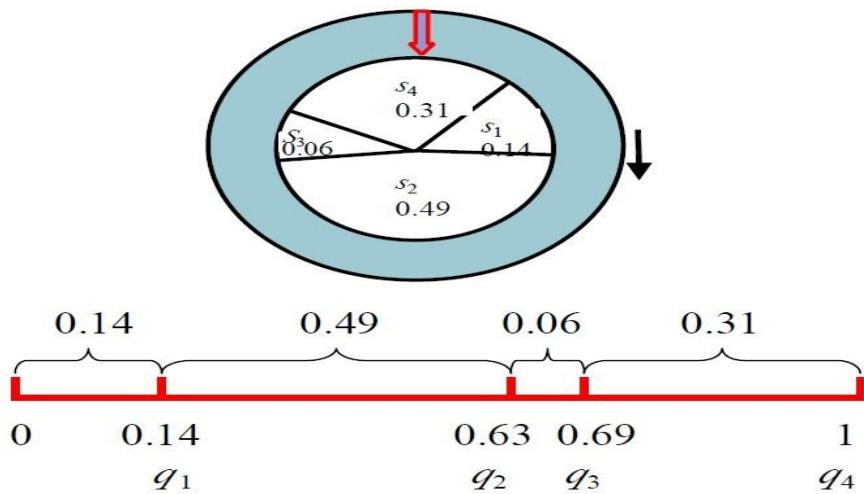
$$Q(s_1) = \sum_{j=1}^N P(s_j) = 0.14$$

$$Q(s_2) = \sum_{j=1}^N P(s_j) = 0.14 + 0.49 = 0.63$$

$$Q(s_3) = \sum_{j=1}^N P(s_j) = 0.14 + 0.49 + 0.06 = 0.69$$

$$Q(s_4) = \sum_{j=1}^N P(s_j) = 0.14 + 0.49 + 0.06 + 0.31 = 1$$

根据上面的式子，可得到：



例如设从区间 $[0, 1]$ 中产生 4 个随机数:

$$\begin{array}{ll} r_1 = 0.450126, & r_2 = 0.110347 \\ r_3 = 0.572496, & r_4 = 0.98503 \end{array}$$

染色体	适应度	选择概率	积累概率	选中次数
$s_1=01101$	169	0.14	0.14	1
$s_2=11000$	576	0.49	0.63	2
$s_3=01000$	64	0.06	0.69	0
$s_4=10011$	361	0.31	1.00	1

3. 2、交叉算子

交叉运算，是指对两个相互配对的染色体依据交叉概率 P_c 按某种方式相互交换其部分基因，从而形成两个新的个体。

交叉运算是遗传算法区别于其他进化算法的重要特征，它在遗传算法中起关键作用，是产生新个体的主要方法。

基本遗传算法（SGA）中交叉算子采用单点交叉算子。

单点交叉运算



3.3、变异算子

变异运算，是指改变个体编码串中的某些基因值，从而形成新的个体。

变异运算是产生新个体的辅助方法，决定遗传算法的局部搜索能力，保持种群多样性。

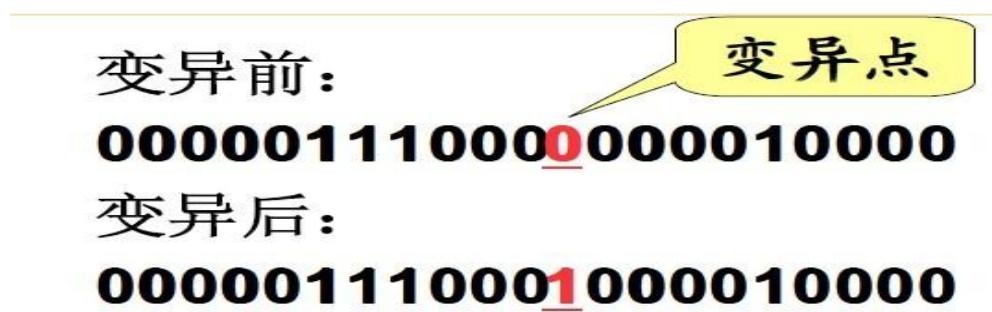
交叉运算和变异运算的相互配合，共同完成对搜索空间的全局搜索和局部搜索。

基本遗传算法（SGA）中变异算子采用基本位变异算子。

基本位变异算子是指对个体编码串随机指定的某一位或某几位基因作变异运算。

对于二进制编码符号串所表示的个体，若需要进行变异操作的某一基因座上的原有基因值为 0，则将其变为 1；反之，若原有基因值为 1，则将其变为 0。

基本位变异算子的执行过程：



4、运行参数

- (1) M : 种群规模
- (2) T : 遗传运算的终止进化代数
- (3) P_c : 交叉概率
- (4) P_m : 变异概率

三、浅出遗传算法

遗传算法的本质

遗传算法本质上是对染色体模式所进行的一系列运算，即通过选择算子将当前种群中的优良模式遗传到下一代种群中，利用交叉算子进行模式重组，利用变异算子进行模式突变。

通过这些遗传操作，模式逐步向较好的方向进化，最终得到问题的最优解。

遗传算法的主要有以下八方面的应用：

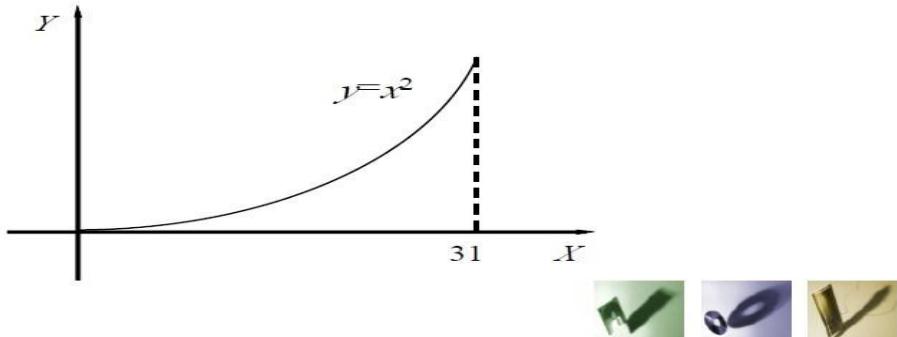
- | | | | |
|----------|----------|----------|----------|
| (1) 组合优化 | (2) 函数优化 | (3) 自动控制 | (4) 生产调度 |
| (5) 图像处理 | (6) 机器学习 | (7) 人工生命 | (8) 数据挖掘 |

四、遗传算法的应用

遗传算法的应用举例、透析本质（这个例子简明、但很重要）

已知 x 为整数，利用遗传算法求解区间 $[0, 31]$ 上的二次函数 $y=x^2$ 的最大值。

已知 x 为整数，利用遗传算法求解区间 $[0, 31]$ 上的二次函数 $y=x^2$ 的最大值。



[分析]

原问题可转化为在区间 $[0, 31]$ 中搜索能使 y 取最大值的点 a 的问题。

个体： $[0, 31]$ 中的任意点 x

适应度： 函数值 $f(x)=x^2$

解空间： 区间 $[0, 31]$

这样，只要能给出个体 x 的适当染色体编码，该问题就可以用遗传算法来解决。

[解]

(1) 设定种群规模, 编码染色体, 产生初始种群。

将种群规模设定为 4; 用 5 位二进制数编码染色体; 取下列个体组成初始种群 S1

$$\begin{aligned}s1 &= 13 (01101), \quad s2 = 24 (11000) \\s3 &= 8 (01000), \quad s4 = 19 (10011)\end{aligned}$$

(2) 定义适应度函数, 取适应度函数

$$f(x) = x^2$$

(3) 计算各代种群中的各个体的适应度, 并对其染色体进行遗传操作,
直到适应度最高的个体, 即 31 (11111) 出现为止。

首先计算种群 S1 中各个体:

$$\begin{aligned}s1 &= 13(01101), \quad s2 = 24(11000) \\s3 &= 8(01000), \quad s4 = 19(10011)\end{aligned}$$

的适应度 $f(s_i)$, 容易求得:

$$\begin{aligned}f(s1) &= f(13) = 13^2 = 169 \\f(s2) &= f(24) = 24^2 = 576 \\f(s3) &= f(8) = 8^2 = 64 \\f(s4) &= f(19) = 19^2 = 361\end{aligned}$$

再计算种群 S1 中各个体的选择概率:

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^N f(x_j)}$$

由此可求得

$$\begin{aligned}P(s1) &= P(13) = 0.14 \\P(s2) &= P(24) = 0.49 \\P(s3) &= P(8) = 0.06 \\P(s4) &= P(19) = 0.31\end{aligned}$$

再计算种群 S1 中各个体的积累概率:

$$q_i = \sum_{j=1}^i P(x_j)$$

选择-复制

设从区间 [0, 1] 中产生 4 个随机数如下：

$$\begin{array}{ll} r_1 = 0.450126, & r_2 = 0.110347 \\ r_3 = 0.572496, & r_4 = 0.98503 \end{array}$$

设从区间 [0, 1] 中产生 4 个随机数如下：

$$\begin{array}{ll} r_1 = 0.450126, & r_2 = 0.110347 \\ r_3 = 0.572496, & r_4 = 0.98503 \end{array}$$

染色体	适应度	选择概率	积累概率	选中次数
s₁=01101	169	0.14	0.14	1
s₂=11000	576	0.49	0.63	2
s₃=01000	64	0.06	0.69	0
s₄=10011	361	0.31	1.00	1

于是，经复制得群体：

$$\begin{array}{ll} s'_1 = 11000 (24), & s'_2 = 01101 (13) \\ s'_3 = 11000 (24) \text{ (24 被选中俩次)}, & s'_4 = 10011 (19) \end{array}$$

交叉

设交叉率 $pc=100\%$ ，即 S_1 中的全体染色体都参加交叉运算。

设 s'_1 与 s'_2 配对， s'_3 与 s'_4 配对。

$$\begin{array}{ll} s'_1 = 11000 (24), & s'_2 = 01101 (13) \\ s'_3 = 11000 (24), & s'_4 = 10011 (19) \end{array}$$

分别交换后两位基因，得新染色体：

$$\begin{array}{ll} s'_1' = 11001 (25), & s'_2' = 01100 (12) \\ s'_3' = 11011 (27), & s'_4' = 10000 (16) \end{array}$$

变异

设变异率 $pm=0.001$ 。

这样，群体 S_1 中共有

$$5 \times 4 \times 0.001 = 0.02$$

位基因可以变异。

0.02 位显然不足 1 位，所以本轮遗传操作不做变异。

于是，得到第二代种群 S_2 :

$$\begin{aligned}s_1 &= 11001 \text{ (25)}, & s_2 &= 01100 \text{ (12)} \\s_3 &= 11011 \text{ (27)}, & s_4 &= 10000 \text{ (16)}\end{aligned}$$

第二代种群 S_2 中各染色体的情况:

第二代种群 S_2 中各染色体的情况

染色体	适应度	选择概率	积累概率	估计的选中次数
$s_1=11001$	625	0.36	0.36	1
$s_2=01100$	144	0.08	0.44	0
$s_3=11011$	729	0.41	0.85	2
$s_4=10000$	256	0.15	1.00	1

假设这一轮选择-复制操作中，种群 S_2 中的 4 个染色体都被选中，则得到群体:

$$\begin{aligned}s'_1 &= 11001 \text{ (25)}, & s'_2 &= 01100 \text{ (12)} \\s'_3 &= 11011 \text{ (27)}, & s'_4 &= 10000 \text{ (16)}\end{aligned}$$

做交叉运算，让 s'_1 与 s'_2 , s'_3 与 s'_4 分别交换后三位基因，得

$$\begin{aligned}s''_1 &= 11100 \text{ (28)}, & s''_2 &= 01001 \text{ (9)} \\s''_3 &= 11000 \text{ (24)}, & s''_4 &= 10011 \text{ (19)}\end{aligned}$$

这一轮仍然不会发生变异。

于是，得第三代种群 S_3 :

$$\begin{aligned}s_1 &= 11100 \text{ (28)}, & s_2 &= 01001 \text{ (9)} \\s_3 &= 11000 \text{ (24)}, & s_4 &= 10011 \text{ (19)}\end{aligned}$$

第三代种群 S_3 中各染色体的情况:

第三代种群 S_3 中各染色体的情况

染色体	适应度	选择概率	积累概率	估计的选中次数
$s_1=11100$	784	0.44	0.44	2
$s_2=01001$	81	0.04	0.48	0
$s_3=11000$	576	0.32	0.80	1
$s_4=10011$	361	0.20	1.00	1

设这一轮的选择-复制结果为：

$s'_1 = 11100$ (28), $s'_2 = 11100$ (28)

$s'_3 = 11000$ (24), $s'_4 = 10011$ (19)

做交叉运算，让 s'_1 与 s'_4 , s'_2 与 s'_3 分别交换后两位基因，得

$s'_1' = 11111$ (31), $s'_2' = 11100$ (28)

$s'_3' = 11000$ (24), $s'_4' = 10000$ (16)

这一轮仍然不会发生变异。

于是，得第四代种群 S4:

$s_1=11111$ (31) (出现最优解), $s_2=11100$ (28)

$s_3=11000$ (24), $s_4=10000$ (16)

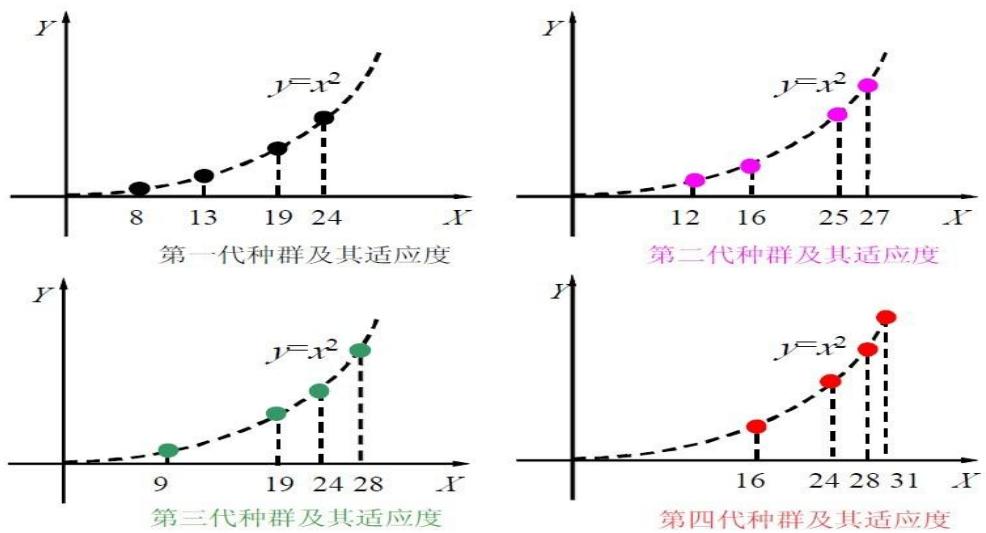
显然，在这一代种群中已经出现了适应度最高的染色体 $s_1=11111$ 。

于是，遗传操作终止，将染色体 (11111) 作为最终结果输出。

然后，将染色体“11111”解码为表现型，即得所求的最优解：31。

将 31 代入函数 $y=x^2$ 中，即得原问题的解，即函数 $y=x^2$ 的最大值为 961。

所以，综合以上各代群的情况，如下：



ok, 完。

文章有误之处，一经发现，立马修正。同时，也望各位不吝指正。

本人 July 对本博客所有文章和资料享有版权，转载或引用任何文章、资料请注明出处。
向您的厚道致敬。谢谢。 July、二零一一年一月十二日。

八、再谈启发式搜索算法

作者:July 二零一一年二月十日

本文参考:

- I、维基百科、
 - II、人工智能-09 启发式搜索、
 - III、本 BLOG 内，经典算法研究系列：一、A*搜索算法
-

引言：

A*搜索算法，作为经典算法研究系列的开篇文章，之前已在本 BLOG 内有所阐述。
但要真正理解 A*搜索算法，还是得先从启发式搜索算法谈起。

毕竟，A*搜索算法也是启发式算法中的一种。ok，切入正题。

一、何谓启发式搜索

启发式搜索算法有点像广度优先搜索，不同的是，它会优先顺着有启发性和具有特定信息的节点搜索下去，这些节点可能是到达目标的最好路径。

我们称这个过程为最优（best-first）或启发式搜索。

下面是其基本思想：

1) 假定有一个启发式(评估)函数 f ，可以帮助确定下一个要扩展的最优节点，我们采用一个约定，即 f 的值小表示找到了好的节点。

这个函数基于指定问题域的信息，它是状态描述的一个实数值函数。

2) 下一个要扩展的节点 n 是 $f(n)$ 值最小的节点(假定节点扩展产生一个节点的所有后继)。

3) 当下一个要扩展的节点是目标节点时过程终止。

我们常常可以为最优搜索指定好的评估函数。

如在**8数码问题**中，可以用不正确位置的数字个数作为状态描述好坏的一个度量：

$f(n)$ = 位置不正确的数字个数(和目标相比)

在搜索过程中采用这个启发式函数将产生图 9 - 1 所示的图，每个节点的数值是该节点的值。

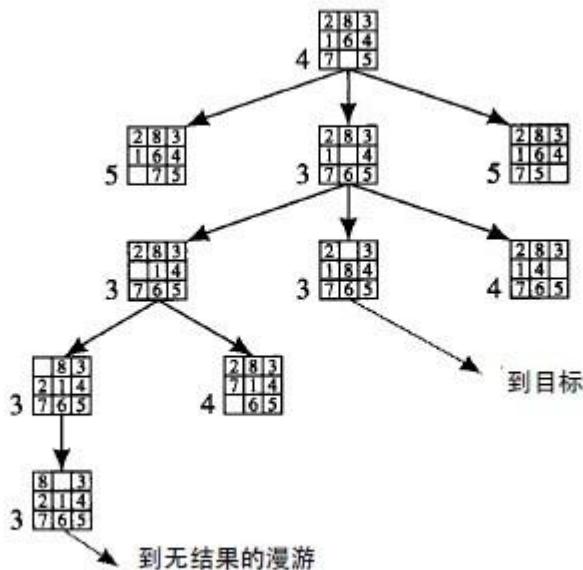


图9-1 一个启发式搜索过程的一个可能结果

图 9-1 一个启发式搜索过程的一个可能结果

上述例子表明，在搜索过程中我们需要偏向有利于回溯到早期路径的搜索(为了避免由于过分的优化试探而陷入“花园小径”)。

因此我们加了一个“深度因子”给 f : $f(n) = g^*(n) + h^*(n)$, $g^*(n)$ 是对图中节点 n 的“深度”估计(即从开始节点到 n 的最短路径长度), $h^*(n)$ 是对节点 n 的启发或评估。

像前面一样,如果 $g^*(n)=$ 搜索图中节点 n 的深度, $h^*(n)=$ 不正确位置的数字个数(和目标相比), 我们可以得到图 9-2。

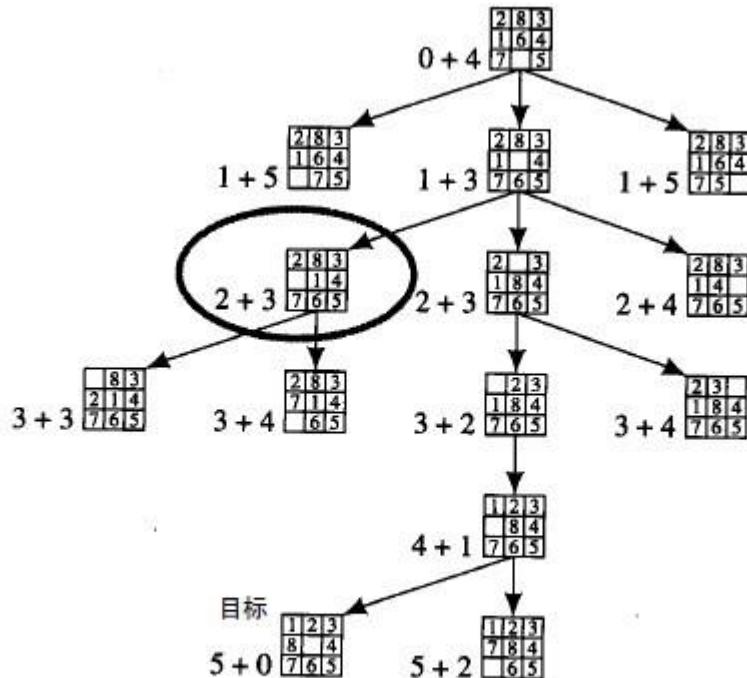


图9-2 使用 $f(n) = g^*(n) + h^*(n)$ 的启发式搜索

在这个图中, 把 $g^*(n)$ 和 $h^*(n)$ 的值写在每个节点的旁边, 在这种情况下, 可以看到搜索相当直接地朝着目标进行(除了用圆圈标注的节点外)。

这些例子提出了两个重要的问题。

第一, 我们如何为最优搜索决定评估函数?

第二, 最优搜索的特性是什么? 它能找到到达目标节点的好路径吗?

本文主要讨论最优搜索的形式表示。作为例子, 下面介绍一个包括最优搜索版本的一般图搜索算法

(为了更详细地了解启发式搜索, 可以参考引用的文章和 Pearl 写的书[pearl 1984])。

二、一个通用的图搜索算法

为了更准确地解释启发式搜索过程, 这里提出一个通用的图搜索算法, 它允许各种用户—偏爱启发式的或盲目的, 进行定制。我把这个算法叫做图搜索 (GRAPHSEARCH)。

下面是（第一个版本）它的定义。

GRAPHSEARCH:

- 1) 生成一个仅包含开始节点 n_0 的搜索树 Tr 。把 n_0 放在一个称为 $OPEN$ 的有序列表中。
- 2) 生成一个初始值为空的列表 $CLOSED$ 。
- 3) 如果 $OPEN$ 为空，则失败并退出。
- 4) 选出 $OPEN$ 中的第一个节点，并将它从 $OPEN$ 中移出，放入 $CLOSED$ 中。称该节点为 n 。
- 5) 如果 n 是目标节点，顺着 Tr 中的弧从 n 回溯到 n_0 找到一条路径，获得解决方案，则成功退出（弧在第 6 步产生）。
- 6) 扩展节点 n ，生成 n 的后继节点集 M 。通过在 Tr 中建立从 n 到 M 中每个成员的弧生成 n 的后继。
- 7) 按照任意的模式或启发式方式对列表 $OPEN$ 重新排序。
- 8) 返回步骤 3。

这个算法可用来执行最优搜索、广度优先搜索或深度优先搜索。

在广度优先搜索中，新节点只要放在 $OPEN$ 的尾部即可（先进先出，FIFO），节点不用重排。

在深度优先搜索中，新节点放在 $OPEN$ 的开始（后进先出，LIFO）。

在最优（也叫启发式）搜索中，按照节点的启发式方式来重排 $OPEN$ 。

三、略谈 A*搜索算法

用最优搜索算法详细说明 GRAPHSEARCH。

最优搜索算法根据函数的增加值，(在上述第 7 步)重排 $OPEN$ 中的节点，如 8 数码问题。把 GRAPHSEARCH 的这种算法称为算法 A*。

下面将会看到，定义使 A*执行广度搜索或相同代价搜索的函数是可行的。为了确定要用的函数族，必须先

介绍一些其他符号。

设 $g(n)$ = 从开始节点 n_0 到节点 n 的一个最小代价路径的代价。

设 $h(n)$ = 节点 n 和目标节点（遍及所有可能的目标节点以及从 n 到它们的所有可能路径）之间的最小代价路径的实际代价。

那么 $f(n) = g(n) + h(n)$ 就是从 n_0 到目标节点并且经过节点 n 的最小代价路径的代价。

注意 $f(n_0) = h(n_0)$ 是从 n_0 到目标节点的一个（不受限的）最小代价路径的代价。

对每个节点 n ，设 $g^*(n)$ （深度因子）是由 A*发现的到节点 n 的最小代价路径的代价， $h^*(n)$ （启发因子）是 $h(n)$ 的某个估计。

在算法 A*中，我们用 $f^*(n) = g^*(n) + h^*(n)$ 。

注意，如果算法 A*中的恒等于 0，就成为相同代价搜索。这些定义示例在图 3 中。

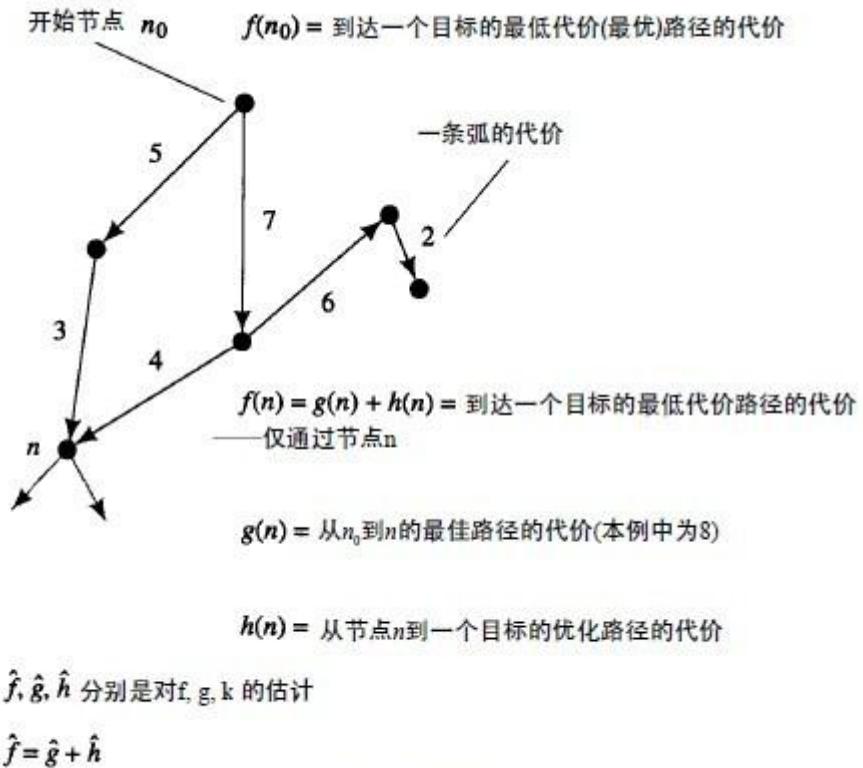


图9-3 启发式搜索符号

算法 A*:

- 1) 生成一个只包含开始节点 n_0 的搜索图 G , 把 n_0 放在一个叫 OPEN 的列表上。
- 2) 生成一个列表 CLOSED, 它的初始值为空。
- 3) 如果 OPEN 为空, 则失败退出。
- 4) 选择 OPEN 上的第一个节点, 把它从 OPEN 中移入 CLOSED, 称该节点为 n 。
- 5) 如果 n 是目标节点, 顺着 G 中, 从 n 到 n_0 的指针找到一条路径, 获得解决方案, 成功退出 (该指针定义了一个搜索树, 在第 7 步建立)。
- 6) 扩展节点 n , 生成其后继节点集 M , 在 G 中, n 的祖先不能在 M 中。在 G 中安置 M 的成员, 使它们成为 n 的后继。
- 7) 从 M 的每一个不在 G 中的成员建立一个指向 n 的指针 (例如, 既不在 OPEN 中, 也不在 CLOSED 中)。把 M 的这些成员加到 OPEN 中。对每一个已在 OPEN 中或 CLOSED 中的成员 m , 如果到目前为止找到的到达 m 的最好路径通过 n , 就把它的指针指向 n 。对已在 CLOSE 中的 M 的每一个成员, 重定向它在 G 中的每一个后继, 以使它们顺着到目前为止发现的最好路径指向它们的祖先。
- 8) 按递增值, 重排 OPEN (相同最小值可根据搜索树中的最深节点来解决)。
- 9) 返回第 3 步。

在第 7 步中, 如果搜索过程发现一条路径到达一个节点的代价比现存的路径代价低, 我们就要重定向指向该节点的指针。已经在 CLOSED 中的节点子孙的重定向保存了后面的搜索结果, 但是可能需要指数级的计算代价。因此, 第 7 步常常不会实现。随着搜索的向前推进, 其中有些指针最终将会被重定向。

更多, 可参考: 经典算法研究系列: 一、A*搜索算法:

四、启发式算法相关问题

4.1、启发式算法与最短路径问题

启发式通常用于资讯充份的搜寻算法，例如最好优先贪婪算法与 A*。

最好优先贪婪算法会为启发式函数选择最低代价的节点；

A*则会为 $g(n) + h(n)$ 选择最低代价的节点，此 $g(n)$ 是从起始节点到目前节点的路径的确
实代价。

如果 $h(n)$ 是可接受的 (admissible) 意即 $h(n)$ 未曾付出超过达到目标的代价，则 A*
一定会找出最佳解。

最能感受到启发式算法好处的经典问题是 n-puzzle。此问题在计算错误的拼图图形，
与计算任两块拼图的曼哈顿距离的总和以及它距离目的有多远时，使用了本算法。注意，上
述两条件都必须在可接受的范围内。

曼哈顿距离是一个简单版本的 n-puzzle 问题，因为我们假设可以独立移动一个方块到
我们想要的位置，而暂不考虑会移到其他方块的问题。

给我们一群合理的启发式函式 $h_1(n), h_2(n), \dots, h_i(n)$ ，而函式 $h(n) = \max\{h_1(n), h_2(n), \dots, h_i(n)\}$ 则是个可预测这些函式的启发式函式。

一个在 1993 年由 A.E. Prieditis 写出的程式 ABSOLVER 就运用了这些技术，这程式
可以自动为问题产生启发式算法。ABSOLVER 为 8-puzzle 产生的启发式算法优于任何先
前存在的！而且它也发现了第一个有用的解魔术方块的启发式程式。

4.2、启发式算法对运算效能的影响

任何的搜寻问题中，每个节点都有 b 个选择以及到达目标的深度 d ，一个毫无技巧的算
法通常都要搜寻 bd 个节点才能找到答案。

启发式算法借由使用某种切割机制降低了分叉率 (branching factor) 以改进搜寻效率，
由 b 降到较低的 b' 。分叉率可以用来定义启发式算法的偏序关系，例如：若在一个 n 节点
的搜寻树上， $h_1(n)$ 的分叉率较 $h_2(n)$ 低，则 $h_1(n) < h_2(n)$ 。

启发式为每个要解决特定问题的搜寻树的每个节点提供了较低的分叉率，因此它们拥有
较佳效率的计算能力。

完。

九、图像特征提取与匹配之 SIFT 算法

作者:July、二零一一年二月十五日。

推荐阅读:

David G. Lowe, "Distinctive image features from scale-invariant keypoints,"

International Journal of Computer Vision, 60, 2 (2004), pp. 91-110

尺度不变特征转换(**Scale-invariant feature transform** 或 **SIFT**)是一种电脑视觉的算法用来侦测与描述影像中的局部性特征, 它在空间尺度中寻找极值点, 并提取出其位置、尺度、旋转不变量, 此算法由 David Lowe 在 1999 年所发表, 2004 年完善总结。

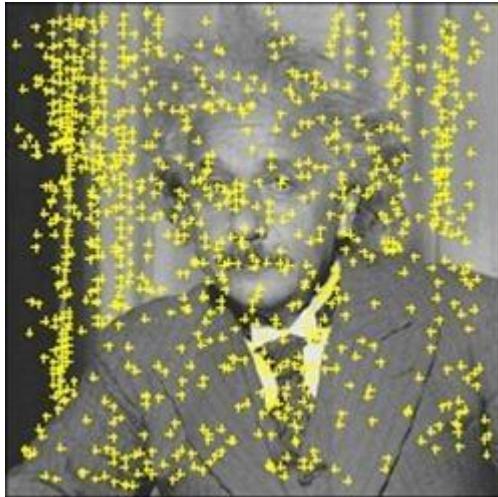
Sift 算法就是用不同尺度(标准差)的高斯函数对图像进行平滑, 然后比较平滑后图像的差别, 差别大的像素就是特征明显的点。

一、Sift 算法的步骤

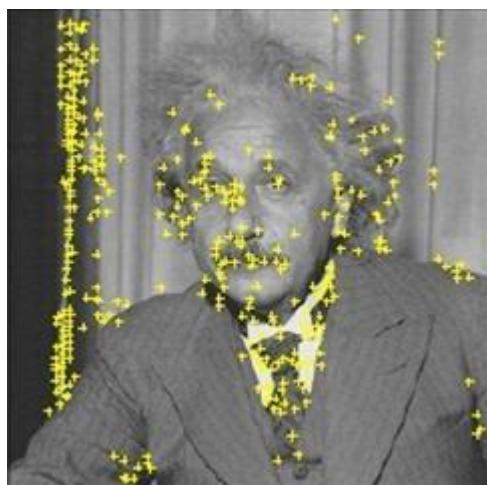
Sift (Scale Invariant Feature Transform) 是一个很好的图像匹配算法, 同时能处理亮度、平移、旋转、尺度的变化, 利用特征点来提取特征描述符, 最后在特征描述符之间寻找匹配。

该算法主要包括 5 个步骤进行匹配:

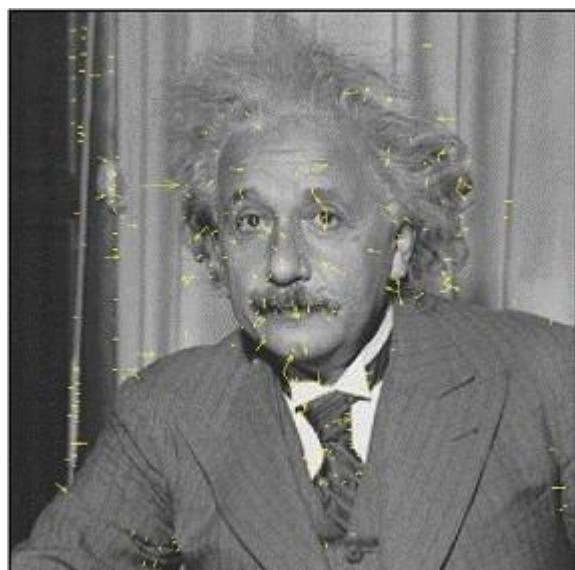
1、构建尺度空间, 检测极值点, 获得尺度不变性;



2、特征点过滤并进行精确定位, 剔除不稳定的特征点;



3、在特征点处提取特征描述符，为特征点分配方向值；



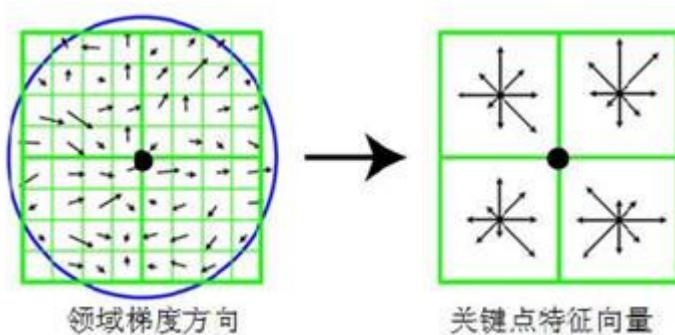
4、生成特征描述子，利用特征描述符寻找匹配点；

以特征点为中心取 **16*16** 的邻域作为采样窗口，

将采样点与特征点的相对方向通过高斯加权后归入包含 8 个 bin 的方向直方图，

最后获得 **4*4*8** 的 **128** 维特征描述子。

示意图如下：



5、计算变换参数。

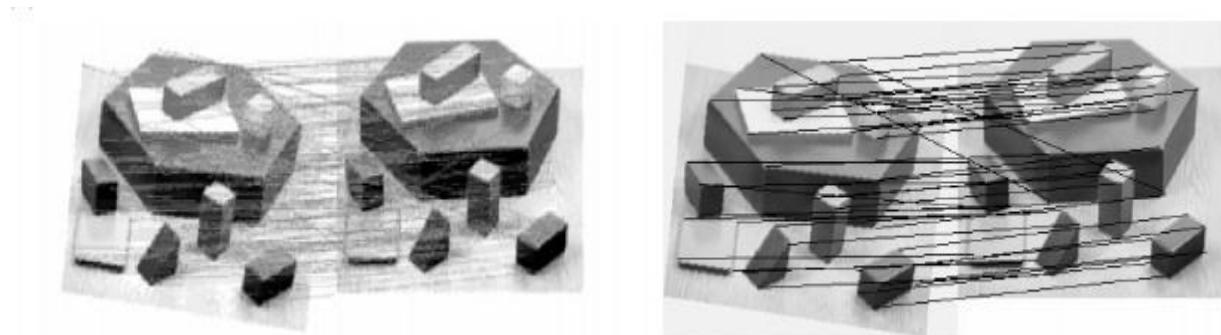
当两幅图像的 Sift 特征向量生成以后，下一步就可以采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。

取图 1 的某个关键点，通过遍历找到图像 2 中的距离最近的两个关键点。

在这两个关键点中，如果次近距离除以最近距离小于某个阈值，则判定为一对匹配点。

最后，看下 Sift 算法效果图：

下图左边部分 Sift 算法匹配结果，右边部分是其它算法匹配结果：



二、Sift 算法的描述

在上述的 Sift 算法步骤一中，提到了尺度空间，那么什么是尺度和尺度空间呢？

尺度就是受 δ 这个参数控制的表示。

而不同的 $L(x,y,\delta)$ 就构成了尺度空间，实际上，具体计算的时候，即使连续的高斯函数，都要被离散为（一般为奇数大小） $(2*k+1) * (2*k+1)$ 矩阵，来和数字图像进行卷积运算。

David Lowe 关于 Sift 算法，2004 年发表在 Int. Journal of Computer Vision 的经典论文中，对尺度空间（scal space）是这样定义的：

It has been shown by Koenderink (1984) and Lindeberg (1994) that under a variety of reasonable assumptions the only possible scale-space kernel is the Gaussian function.

Therefore, the scale space of an image is defined as a function, $L(x; y; \delta)$ that is

produced from the convolution of a variable-scale Gaussian, $G(x; y; \delta)$, with an input

image, $I(x; y)$:

因此，一个图像的尺度空间， $L(x, y, \delta)$ ，

定义为原始图像 $I(x, y)$ 与一个可变尺度的 2 维高斯函数 $G(x, y, \delta)$ 卷积运算。

即，原始影像 $I(x, y)$ 在不同的尺度 δ 下，与高斯滤波器 $G(x, y, \delta)$ 进行卷积，得到 $L(x, y, \delta)$ ，如下：

$$L(x, y, \delta) = G(x, y, \delta) * I(x, y)$$

其中 $G(x, y, \delta)$ 是尺度可变高斯函数，

$$G(x, y, \delta) = [1/2 * \pi * \delta^2] * \exp[-(x^2 + y^2)/2\delta^2]$$

(x, y) 是空间坐标， δ 是尺度坐标。

为了更有效的在尺度空间检测到稳定的关键点，提出了高斯差分尺度空间（DOG scale-space）。

利用不同尺度的高斯差分核与原始图像 $I(x, y)$ ，卷积生成。

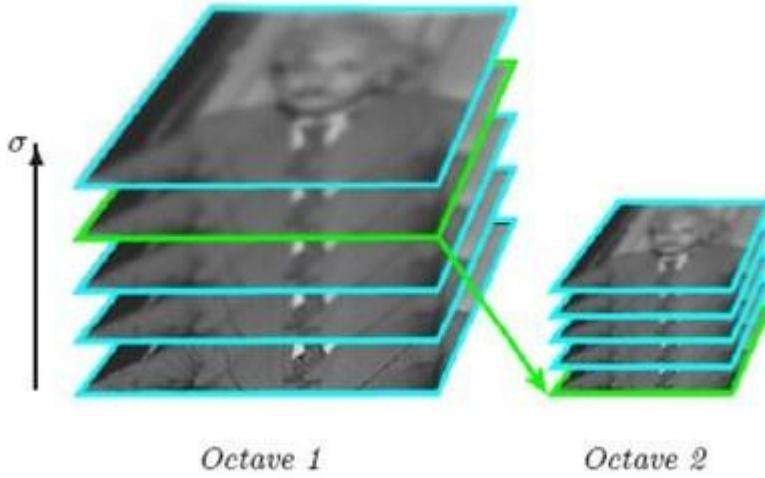
$$\begin{aligned} D(x, y, \delta) &= ((G(x, y, \delta) - G(x, y, \delta))) * I(x, y) \\ &= L(x, y, \delta) - L(x, y, \delta) \end{aligned}$$

DOG 算子计算简单，是尺度归一化的 LoG 算子的近似。

Gaussian 卷积是有尺寸大小的，使用同一尺寸的滤波器对两幅包含有不同尺寸的同一物体的图像求局部最值将有可能出现一方求得最值而另一方却没有的情况，但是容易知道假如物体的尺寸都一致的话它们的局部最值将会相同。

SIFT 的精妙之处在于采用图像金字塔的方法解决这一问题，我们可以把两幅图像想象成是连续的，分别以它们作为底面作四棱锥，就像金字塔，那么每一个 截面与原图像相似，那么两个金字塔中必然会有包含大小一致的物体的无穷个截面，但应用只能是离散的，所以我们只能构造有限层，层数越多当然越好，但处理时间会相应增加，层数太少不行，因为向下采样的截面中可能找不到尺寸大小一致的两个物体的图像。

有了图像金字塔就可以对每一层求出局部最值，但是这样的稳定 点数目将会十分可观，所以需要使用某种方法抑制去除一部分点，但又使得同一尺度下的稳定点得以保存图像金字塔的构建：图像金字塔共 O 组，每组有 S 层，下一组的图像由上一组图像降采样得到。如下图：



三、Sift 算法的实现

作为一种匹配能力较强的局部描述算子，SIFT 算法的实现相当复杂，不过 David Lowe 到底也还是用 c++ 实现了它，下面，阐述下其中的两个关键函数。

关键函数一：

```
int sift_features( IplImage* img, struct feature** feat )
```

这个函数就是用来提取图像中的特征向量。

参数 `img` 为一个指向 `IplImage` 数据类型的指针，用来表示需要进行特征提取的图像。

`IplImage` 是 opencv 库定义的图像基本类型（关于 opencv 是一个著名的图像处理类库，详细的介绍可以参见

<http://www.opencv.org.cn>

）。

参数 `feat` 是一个数组指针，用来存储图像的特征向量。

函数调用成功将返回特征向量的数目，否则返回-1.

函数完整表述如下：

```
int sift_features( IplImage* img, struct feature** feat )
{
    return _sift_features( img, feat, SIFT_INTVLS, SIFT_SIGMA,
SIFT_CONTR_THR,
    SIFT_CURV THR, SIFT_IMG_DBL, SIFT_DESCR_WIDTH,
    SIFT_DESCR_HIST_BINS );
}
```

关键函数二：

```
int _sift_features( IplImage* img, struct feature** feat, int intvls,double sigma,
double
```

```
contr_thr, int curv_thr, int img_dbl, int descr_width, int descr_hist_bins )
```

稍微介绍下此函数的几个参数：

intvls: 每个尺度空间的采样间隔数，默认值为 3.

sigma: 高斯平滑的数量，默认值 1.6.

contr_thr: 判定特征点是否稳定，取值 (0, 1)，默认为 0.04，这个值越大，被剔除的特征点就越多。

curv_thr: 判定特征点是否边缘点，默认为 6.

img_dbl: 在建立尺度空间前如果图像被放大了 1 倍则取值为 1，否则为 0.

descr_width: 计算特征描述符时邻域子块的宽度，默认为 4.

descr_hist_bins: 计算特征描述符时将特征点邻域进行投影的方向数，默认为 8，分别是 0, 45, 90, 135, 180, 215, 270, 315 共 8 个方向。

以下是此函数的完整表述：

```
int _sift_features( IplImage* img, struct feature** feat, int intvl,
                     double sigma, double contr_thr, int curv_thr,
                     int img_dbl, int descr_width, int descr_hist_bins )
{
    IplImage* init_img;
    IplImage*** gauss_pyr, *** dog_pyr;
    CvMemStorage* storage;
    CvSeq* features;
    int octvs, i, n = 0;

    /* check arguments */
    if( ! img )
        fatal_error( "NULL pointer error, %s, line %d", __FILE__, __LINE__ );

    if( ! feat )
        fatal_error( "NULL pointer error, %s, line %d", __FILE__, __LINE__ );

    /* build scale space pyramid; smallest dimension of top level is ~4 pixels */
    init_img = create_init_img( img, img_dbl, sigma );
    octvs = log( MIN( init_img->width, init_img->height ) ) / log(2) - 2;
    gauss_pyr = build_gauss_pyr( init_img, octvs, intvl, sigma );
    dog_pyr = build_dog_pyr( gauss_pyr, octvs, intvl );

    storage = cvCreateMemStorage( 0 );
    features = scale_space_extrema( dog_pyr, octvs, intvl, contr_thr,
                                    curv_thr, storage );
    calc_feature_scales( features, sigma, intvl );
    if( img_dbl )
        adjust_for_img_dbl( features );
    calc_feature_olis( features, gauss_pyr );
    compute_descriptors( features, gauss_pyr, descr_width, descr_hist_bins );
```

```

/* sort features by decreasing scale and move from CvSeq to array */
cvSeqSort( features, (CvCmpFunc)feature_cmp, NULL );
n = features->total;
*feat = calloc( n, sizeof(struct feature) );
*feat = cvCvtSeqToArray( features, *feat, CV_WHOLE_SEQ );
for( i = 0; i < n; i++ )
{
    free( (*feat)[i].feature_data );
    (*feat)[i].feature_data = NULL;
}

cvReleaseMemStorage( &storage );
cvReleaseImage( &init_img );
release_pyr( &gauss_pyr, octvs, intvls + 3 );
release_pyr( &dog_pyr, octvs, intvls + 2 );
return n;
}

```

这个函数是上述函数一的重载，作用是一样的，实际上函数一只不过是使用默认参数调用了函数二，核心的代码都是在函数二中实现的。

sift 创始人 David Lowe 的完整代码，包括他的论文，请到此处下载：

<http://www.cs.ubc.ca/~lowe/keypoints>

日后，本 BLOG 内，会具体剖析下上述 David Lowe 的 Sift 算法代码。

Rob Hess 维护的 sift 库：

<http://blogs.oregonstate.edu/hess/code/sift/>

还可，参考这里：

sift 图像特征提取与匹配算法代码(友人，onezeros 博客)：

<http://blog.csdn.net/onezeros/archive/2011/01/05/6117704.aspx>

完。

本人 **July** 对本博客所有任何文章、内容和资料享有版权。

转载务必注明作者本人及出处，并通知本人。**July**、二零一一年二月十五日。

九（续）、SIFT 算法的编译

作者：July 、二零一一年三月一日。

代码：Rob Hess 维护的 sift 库，July updated。

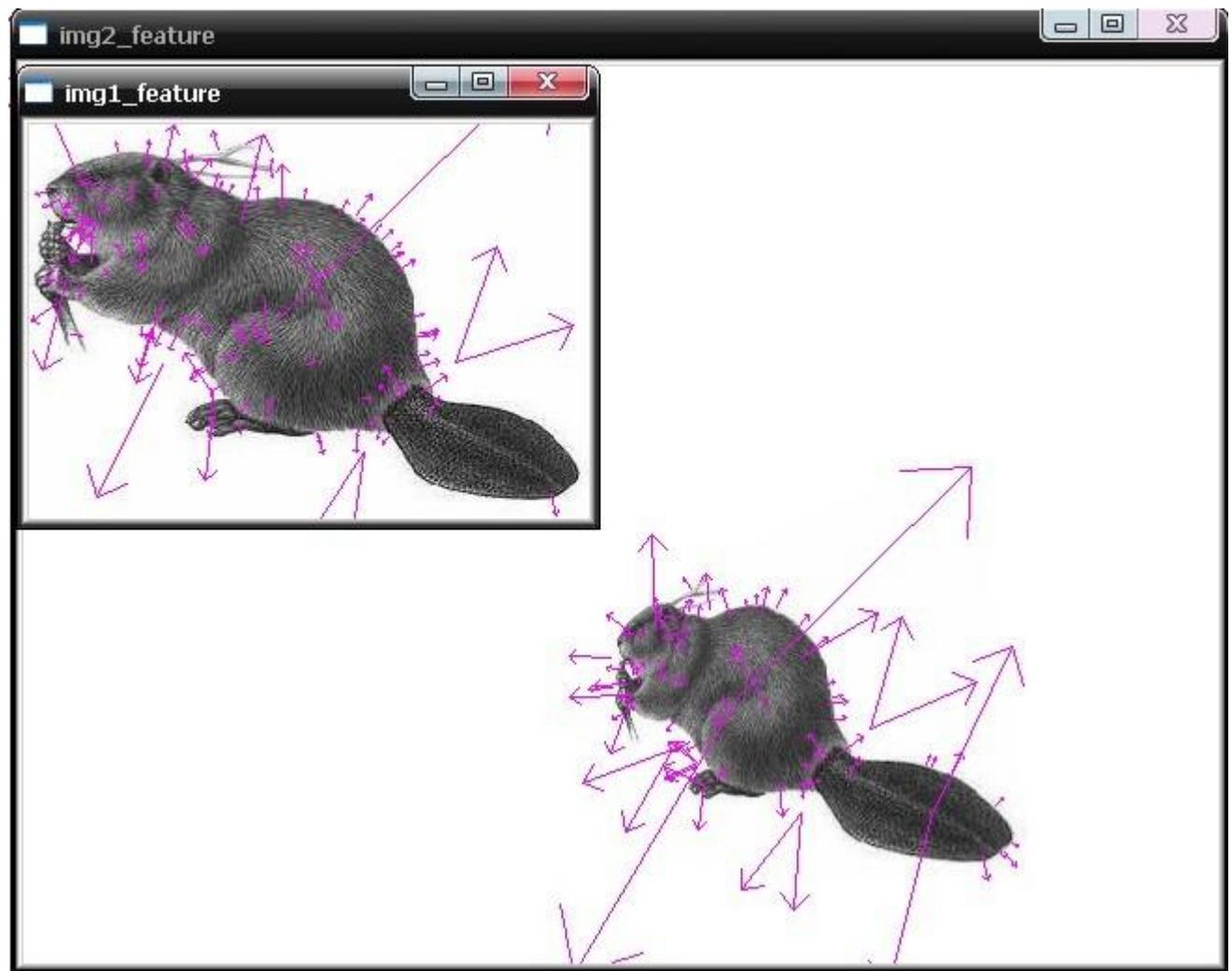
环境：windows xp+vc6.0。

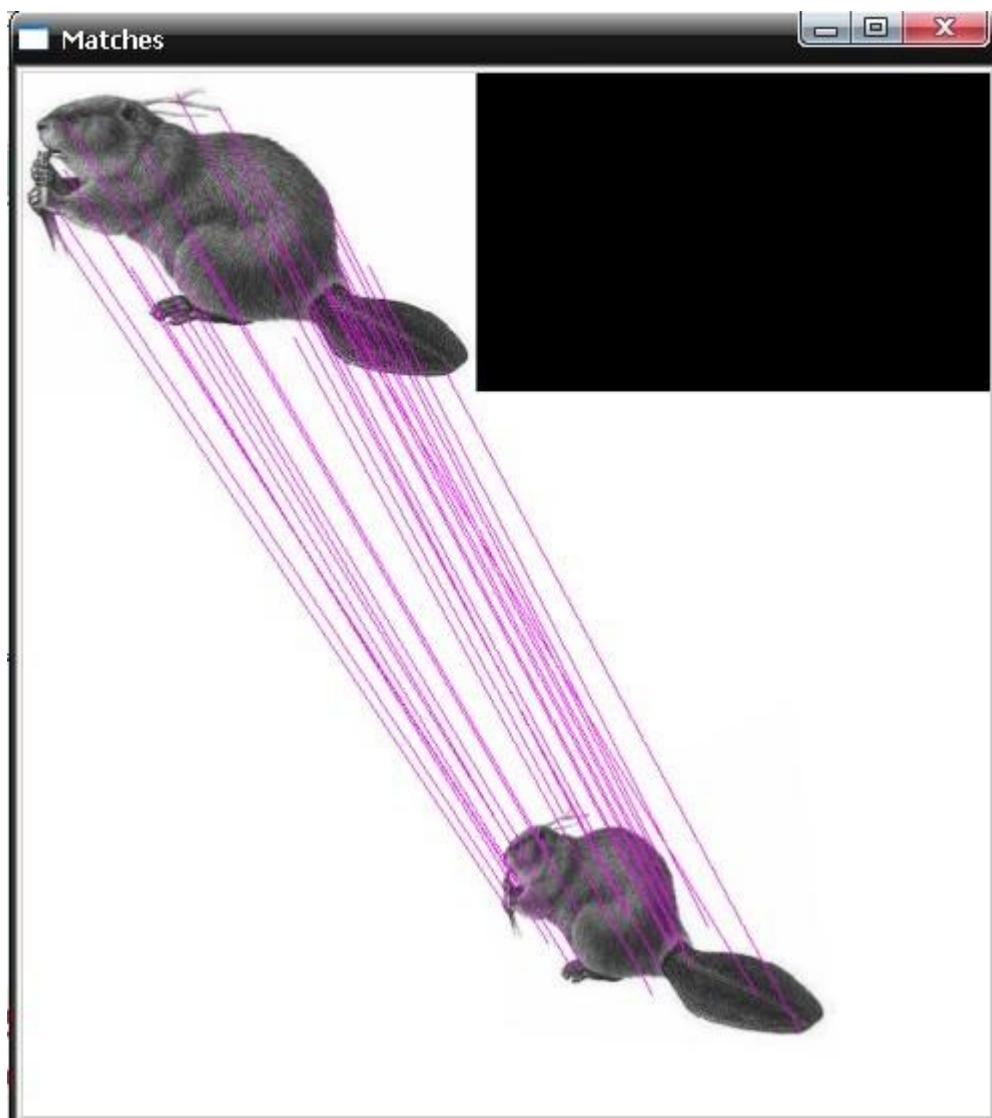
条件：opencv1.0、gsl-1.8.exe

昨日，下载了 Rob Hess 的 sift 库，将其源码粗略的看了看，想要编译时，遇到了不少问题，先修改了下代码，然后下载 opencv、gsl。最后，几经周折，才最终编译成功。

以下便是 sift 源码库编译后的效果图：

```
c:\ "C:\Documents and Settings\Administrator\桌面\傅里叶算法\sift-latest_win\SIFT_VC\Debug\MatchImage.exe" -1 C:\Users\Administrator\Desktop\beaver.png -2 C:\Users\Administrator\Desktop\beaver_xform.png  
Finding features in .\beaver.png...  
Finding features in .\beaver_xform.png...  
Found 30 total matches  
30 个匹配的点的坐标已分别保存到文件：  
upimg_matched_points.xml 和 downimg_matched_points.xml  
同时按ALT键和鼠标右键按下将缩小匹配结果图  
同时按ALT键和鼠标左键按下将放大匹配结果图  
  
搜狗拼音 半：
```





为了给有兴趣实现 **sift** 算法的朋友提供个参考，特整理此文如下。要了解什么是 **sift** 算法，请参考：[九、图像特征提取与匹配之 SIFT 算法](#)。ok，咱们下面，就来利用 Rob Hess 维护的 **sift** 库来实现 **sift** 算法：

首先，请下载 **Rob Hess** 维护的 **sift** 库：

<http://blogs.oregonstate.edu/hess/code/sift/>

下载 **Rob Hess** 的这个压缩包后，如果直接解压缩，直接编译，那么会出现下面的错误提示：

编译提示:error C1083: Cannot open include file: 'cxcore.h': No such file or directory，找不到这个头文件。

这个错误，是因为你还没有安装 **opencv**，因为：**cxcore.h** 和 **cv.h** 是开源的 **OPEN CV** 头文件，不是 **VC++** 的默认安装文件，所以你还得下载 **OpenCV** 并进行安装。然后，可以在 **OpenCV** 文件夹下找到你所需要的头文件了。

据网友称，截止 2010 年 4 月 4 日，还没有在 **VC6.0** 下成功使用 **opencv2.0** 的案例。所以，如果你是 **VC6.0** 的用户请下载 **opencv1.0** 版本。**vs** 的话，**opencv2.0,1.0** 任意下载。

以下，咱们就以 **vc6.0** 为平台举例，下载并安装 **opencv1.0** 版本、**gsl** 等。当然，你也可以用 **vs** 编译，同样下载 **opencv**（具体版本不受限制）、**gsl** 等。

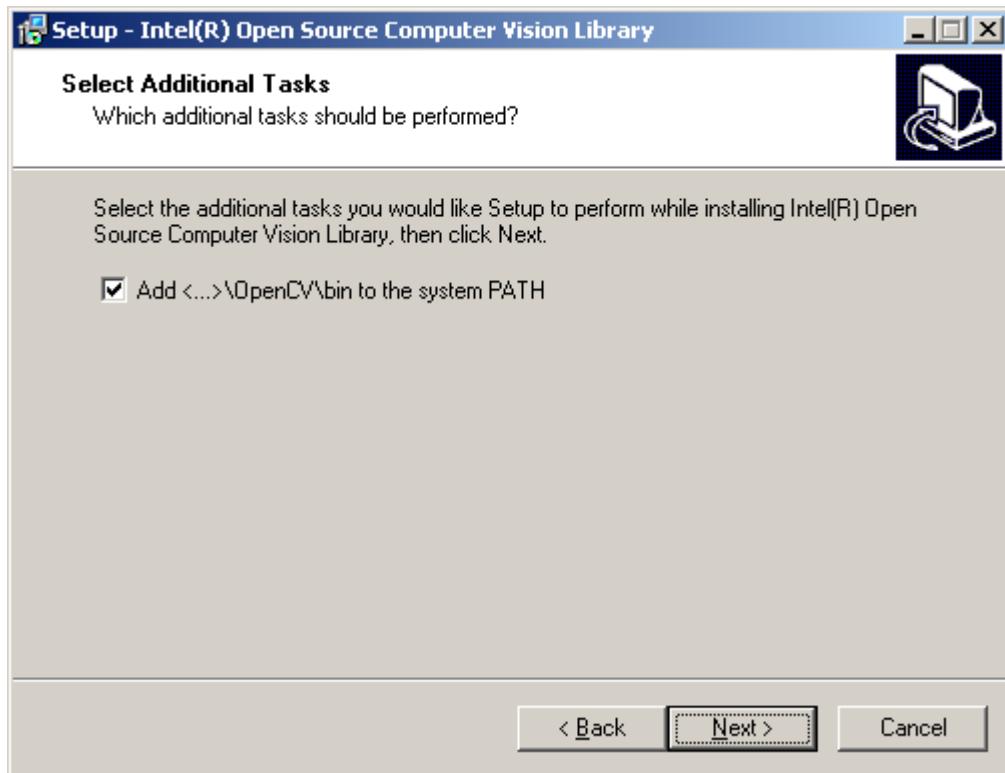
请按以下步骤操作：

一、下载 **opencv1.0**

<http://sourceforge.net/projects/opencvlibrary/files/opencv-win/1.0/OpenCV1.0.exe/download>

二、安装 **opencv1.0**，配置 **Windows** 环境变量

1、安装注意：假如你是将 OpenCV 安装到 **C:\Program Files\OpenCV**（如果你安装的时候选择不是安装在 C 盘，则下面所有对应的 C 盘都改为你所安装的那个“X 盘”，即可），在安装时选择“将\OpenCV\bin 加入系统变量”，打上“勾”。（Add\OpenCV\bin to the system PATH。这一步确认选上了之后，下面的检查环境变量的步骤，便可免去）



2、检查环境变量。为了确保上述步骤中，加入了系统变量，在安装 **opencv1.0** 成功后，还得检查 **C:\Program Files\OpenCV\bin** 是否已经被加入到环境变量 **PATH**，如果没有，请加入。

3、最后是配置 **Visual C++ 6.0**。

全局设置

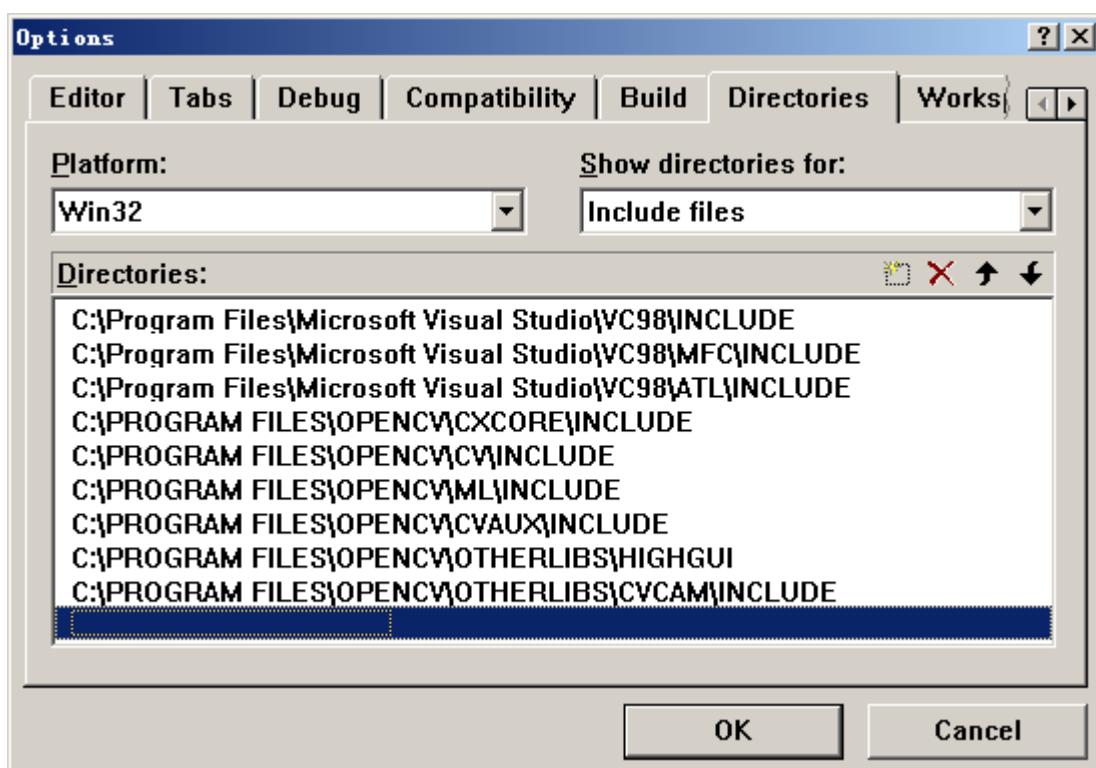
菜单 Tools->Options->Directories：先设置 lib 路径，选择 Library files，在下方

填入路径:

C:\Program Files\OpenCV\lib

然后选择 include files, 在下方填入路径(参考下图):

**C:\Program Files\OpenCV\cxcore\include
C:\Program Files\OpenCV\cv\include
C:\Program Files\OpenCV\cvaux\include
C:\Program Files\OpenCV\ml\include
C:\Program Files\OpenCV\otherlibs\highgui
C:\Program Files\OpenCV\otherlibs\cvcam\include**



最后选择 source files, 在下方填入路径:

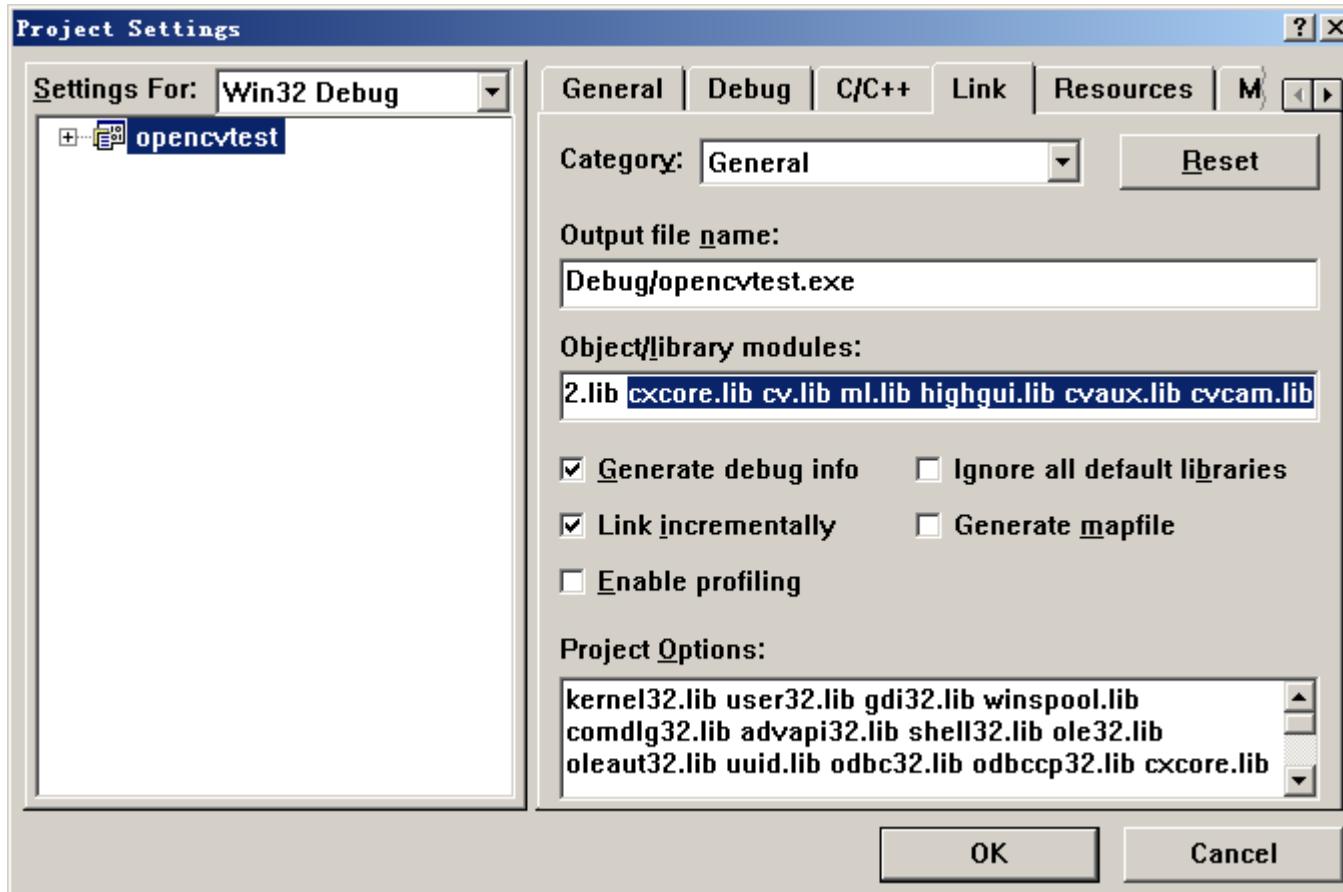
**C:\Program Files\OpenCV\cv\src
C:\Program Files\OpenCV\cxcore\src
C:\Program Files\OpenCV\cvaux\src
C:\Program Files\OpenCV\otherlibs\highgui
C:\Program Files\OpenCV\otherlibs\cvcam\src\windows**

项目设置

每创建一个将要使用 OpenCV 的 VC Project, 都需要给它指定需要的 lib。菜单: Project->Settings, 然后将 Setting for 选为 All Configurations, 然后选择右边的 link 标签, 在 Object/library modules 附加上:

cxcore.lib cv.lib ml.lib cvaux.lib highgui.lib cvcam.lib

当然, 你不需要这么多 lib, 你可以只添加你需要的 lib(见下图)



三、下载 gsl, gsl 也是一个库，也需要下载：

<http://sourceforge.net/projects/gnuwin32/files/gsl/1.8/gsl-1.8.exe/download>。在编译时候 GSL 也是和 OpenCV 一样要把头文件和 lib 的路径指定好。

四、配置 gsl

将 C:\WinGsl\bin 中的 WinGsl.dll 和 WinGslD.dll 复制到 C:\VC6.0\Bin；将整个 Gsl 目录复制到 C:\VC6.0\Bin 下；lib 目录下的所有.lib 文件全部复制到 C:\VC6.0\Lib 下。

然后，在 tools—options—directories 中，将 C:\WinGsl 下的 lib, gsl 分别加入到库文件和头文件的搜索路径中。

以下是可能会出现的错误情况处理：

I、OpenCV 安装后“没有找到 cxcore100.dll”的错误处理

在安装时选择“将\OpenCV\bin 加入系统变量”（Add\OpenCV\bin to the system PATH）。但该选项并不一定能成功添加到系统变量，如果编写的程序在运行时出现“没有

找到 cxcore100.dll, 因为这个应用程序未能启动。重新安装应用程序可能会修复此问题。”的错误。

手动在我的电脑->属性->高级->环境变量->系统变量->path 添加 c:\program files\opencv\bin;添加完成后需要重启计算机。

II、vc6.0 下配置了一下，可是编译程序时遇到如下一个错误：

Linking... LINK : fatal error LNK1104: cannot open file "odbccp32.libcxcore.lib"

可能是：在工程设置的时候添加连接库时没加空格或.来把两个文件名（odbccp32.lib cxcore.lib）分开。注意每一次操作后，记得保存。

若经过以上所有的步骤之后，如果还不能正常编译，那就是还要稍微修改下你下载的 Rob Hess 代码。ok，日后，若有空，再好好详细剖析下此 sift 的源码。最后，祝你编译顺利。

完。

版权声明：原创文章，若需转载，请标明出处。谢谢。

九（再续）、教你一步一步用 c 语言实现 sift 算法、上

作者：July、二零一一年三月十二日

出处：http://blog.csdn.net/v_JULY_v

参考：Rob Hess 维护的 sift 库

环境：windows xp+vc6.0

条件：c 语言实现。

说明：本 BLOG 内会陆续一一实现所有经典算法。

引言：

在我写的关于 sift 算法的前俩篇文章里头，已经对 sift 算法有了初步的介绍：[九、图像特征提取与匹配之 SIFT 算法](#)，而后在：[九（续）、sift 算法的编译与实现](#)里，我也简单记录下了如何利用 opencv, gsl 等库编译运行 sift 程序。

但据一朋友表示，是否能用 c 语言实现 sift 算法，同时，尽量不用到 opencv, gsl 等第三方库之类的东西。而且，Rob Hess 维护的 sift 库，也不好懂，有的人根本搞不懂是怎么一回事。

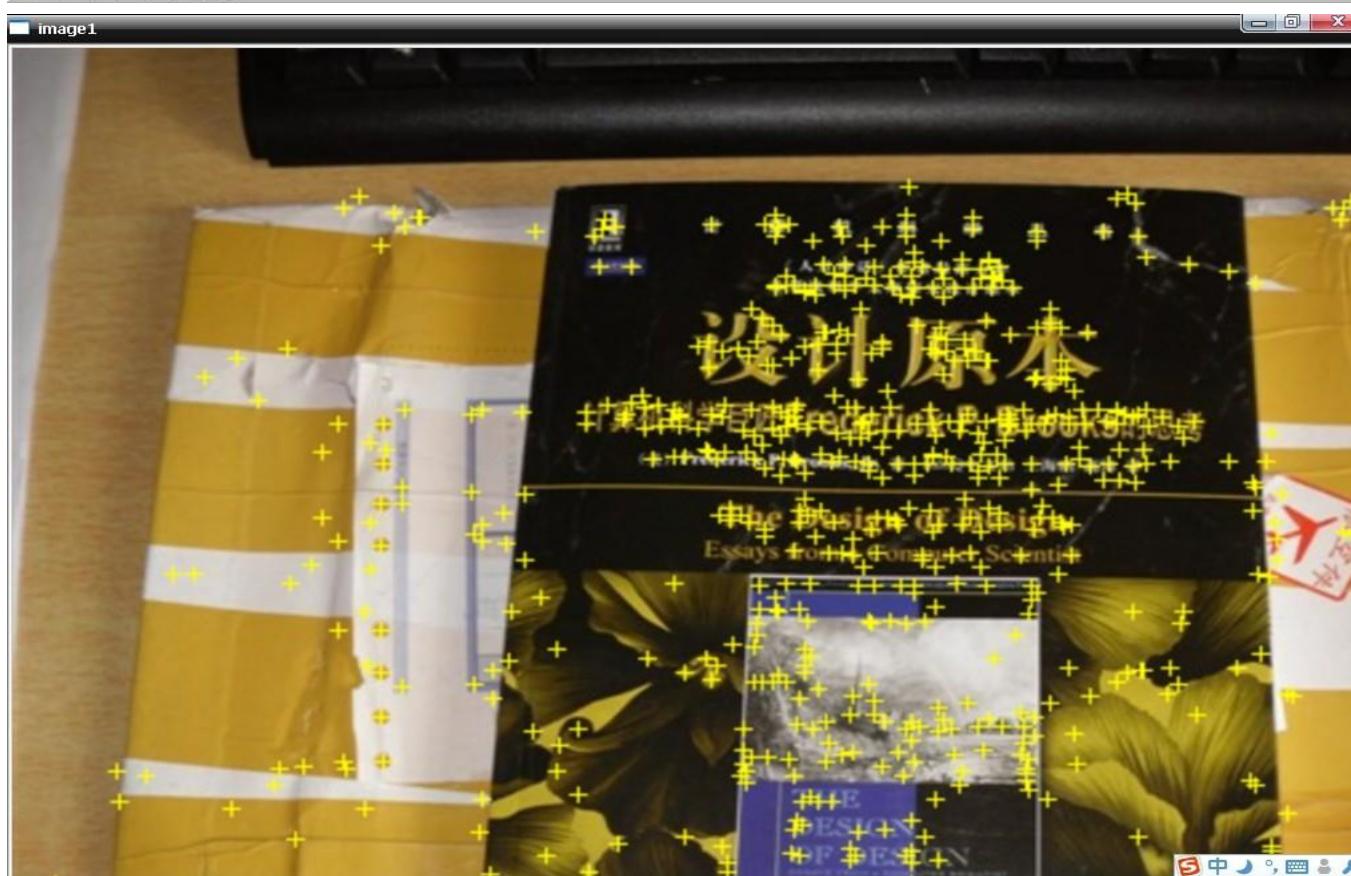
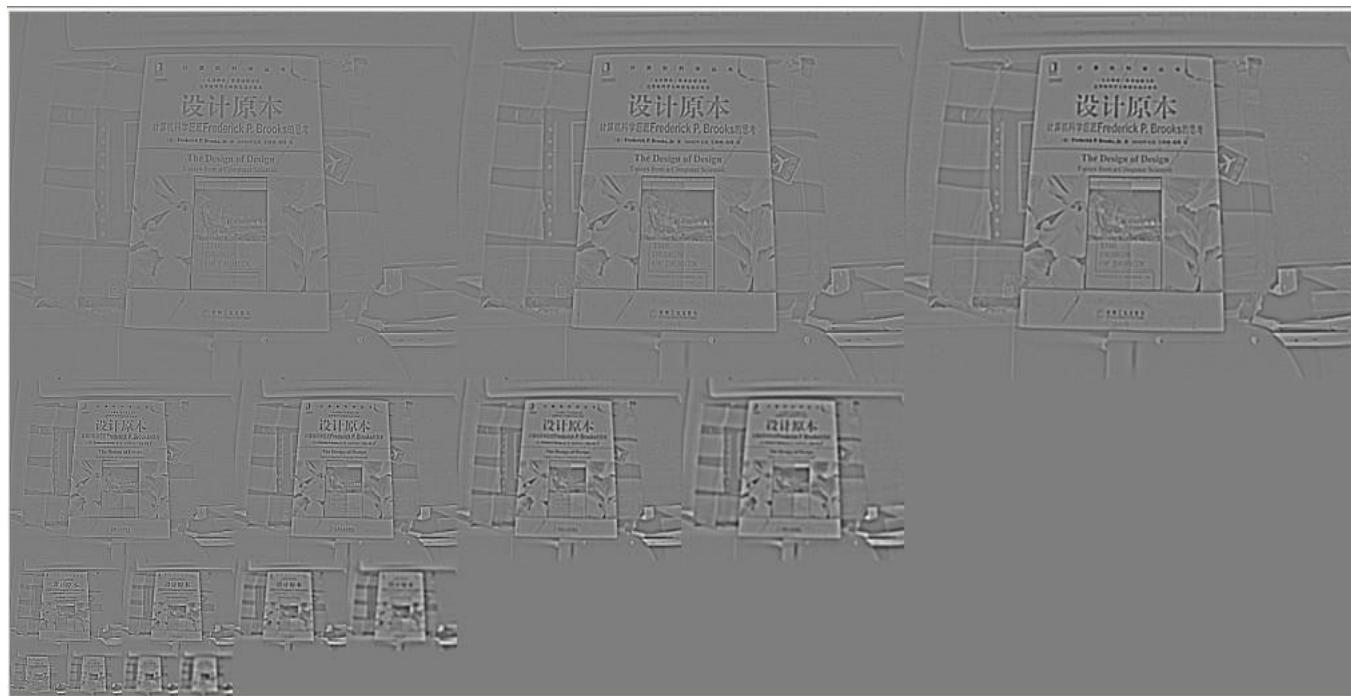
那么本文，就教你如何利用 c 语言一步一步实现 sift 算法，同时，你也就能真正明白 sift 算法到底是怎么一回事了。

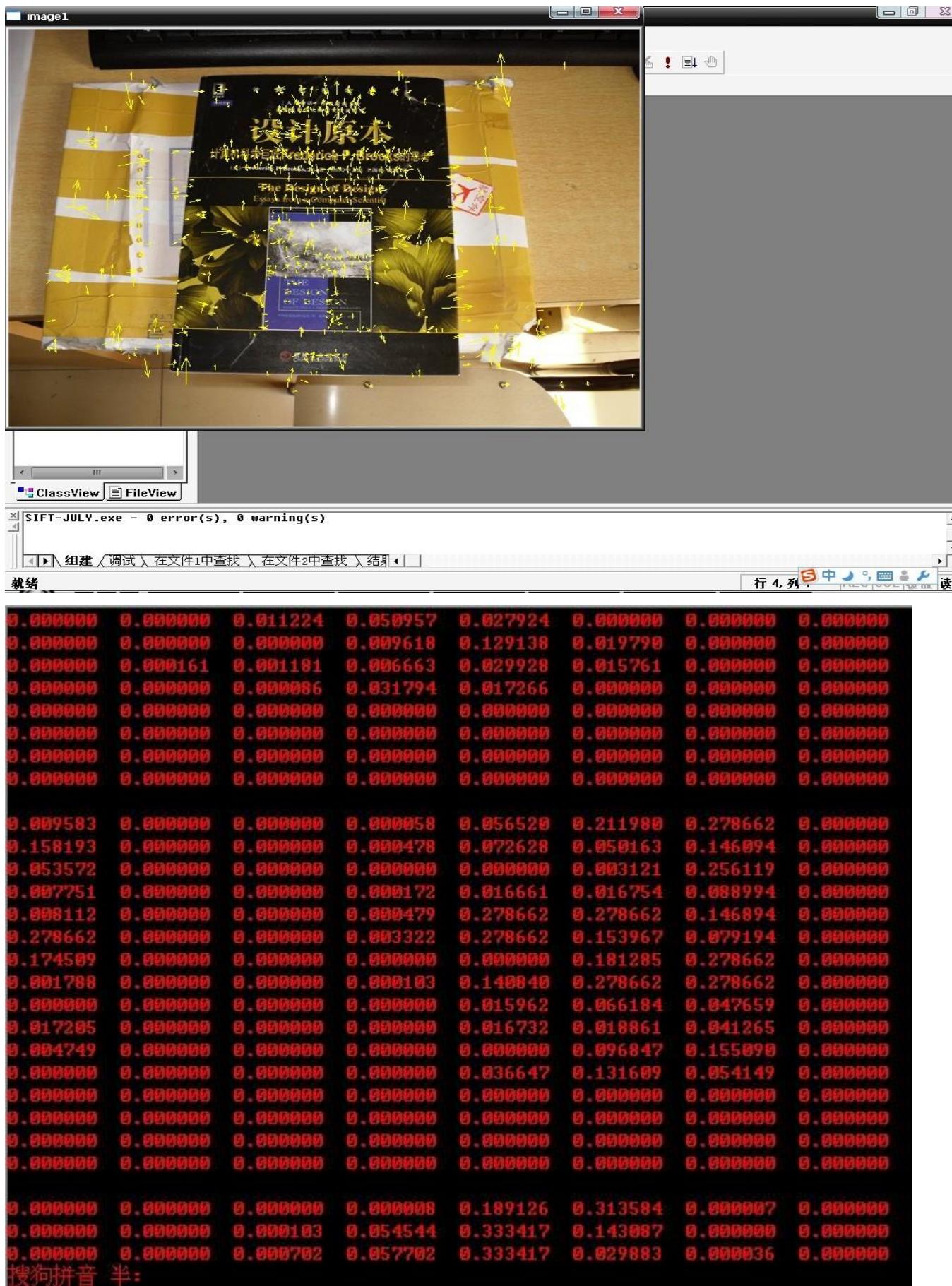
ok，先看一下，本程序最终运行的效果图，sift 算法分为五个步骤（下文详述），对应以 下 第 二 -- 第 六 幅 图 ：

```
BuildGaussianOctaves(): Base image dimension is 670x502
BuildGaussianOctaves(): Building 4 octaves
Building octave 0 of dimesion (1340, 1004)
0 scale and blur sigma : 0.707107
1 scale and Blur sigma: 1.000000
2 scale and Blur sigma: 1.414214
3 scale and Blur sigma: 2.000000
4 scale and Blur sigma: 2.828427
Building octave 1 of dimesion (670, 502)
0 scale and blur sigma : 1.414214
1 scale and Blur sigma: 2.000000
2 scale and Blur sigma: 2.828427
3 scale and Blur sigma: 4.000000
4 scale and Blur sigma: 5.656854
Building octave 2 of dimesion (335, 251)
0 scale and blur sigma : 2.828427
1 scale and Blur sigma: 4.000000
2 scale and Blur sigma: 5.656854
3 scale and Blur sigma: 8.000000
4 scale and Blur sigma: 11.313708
Building octave 3 of dimesion (167, 125)
0 scale and blur sigma : 5.656854
1 scale and Blur sigma: 8.000000
2 scale and Blur sigma: 11.313708
3 scale and Blur sigma: 16.000000
4 scale and Blur sigma: 22.627417
the time of build Gaussian pyramid and DOG pyramid is 4005.2
the keypoints number are 747 ;
```

搜狗拼音 半：







sift 算法的步骤

要实现一个算法，首先要完全理解这个算法的原理或思想。咱们先来简单了解下，什么叫 sift 算法：

sift，尺度不变特征转换，是一种电脑视觉的算法用来侦测与描述影像中的局部性特征，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变量，此算法由 David Lowe 在 1999 年所发表，2004 年完善总结。

所谓，**Sift** 算法就是用不同尺度（标准差）的高斯函数对图像进行平滑，然后比较平滑后图像的差别，差别大的像素就是特征明显的点。

以下是 sift 算法的五个步骤：

一、建立图像尺度空间(或高斯金字塔)，并检测极值点

首先建立尺度空间，要使得图像具有尺度空间不变形，就要建立尺度空间，sift 算法采用了高斯函数来建立尺度空间，高斯函数公式为：

$$G(x,y,e) = [1/2*pi*e^2] * \exp[-(x^2 + y^2)/2e^2]$$

上述公式 $G(x,y,e)$ ，即为尺度可变高斯函数。

而，一个图像的尺度空间 $L(x,y,e)$ ，定义为原始图像 $I(x,y)$ 与上述的一个可变尺度的 2 维高斯函数 $G(x,y,e)$ 卷积运算。

即，原始影像 $I(x,y)$ 在不同的尺度 e 下，与高斯函数 $G(x,y,e)$ 进行卷积，得到 $L(x,y,e)$ ，如下：

$$L(x,y,e) = G(x,y,e)*I(x,y)$$

以上的 (x, y) 是空间坐标， e ，是尺度坐标，或尺度空间因子， e 的大小决定平滑程度，大尺度对应图像的概貌特征，小尺度对应图像的细节特征。大的 e 值对应粗糙尺度(低分辨率)，反之，对应精细尺度(高分辨率)。

尺度，受 e 这个参数控制的表示。而不同的 $L(x,y,e)$ 就构成了尺度空间，具体计算的时候，即使连续的高斯函数，都被离散为（一般为奇数大小） $(2*k+1)*(2*k+1)$ 矩阵，来和数字图像进行卷积运算。

随着 e 的变化，建立起不同的尺度空间，或称之为建立起图像的高斯金字塔。

但，像上述 $L(x,y,e) = G(x,y,e)*I(x,y)$ 的操作，在进行高斯卷积时，整个图像就要遍历所有的像素进行卷积(边界点除外)，于此，就造成了时间和空间上的很大浪费。

为了更有效的在尺度空间检测到稳定的关键点，也为了缩小时间和空间复杂度，对上述的操作作了一个改建：即，提出了高斯差分尺度空间（DOG scale-space）。利用不同尺度的高斯差分与原始图像 $I(x,y)$ 相乘，卷积生成。

$$\begin{aligned} D(x,y,e) &= ((G(x,y,ke) - G(x,y,e)) * I(x,y)) \\ &= L(x,y,ke) - L(x,y,e) \end{aligned}$$

DOG 算子计算简单，是尺度归一化的 LOG 算子的近似。

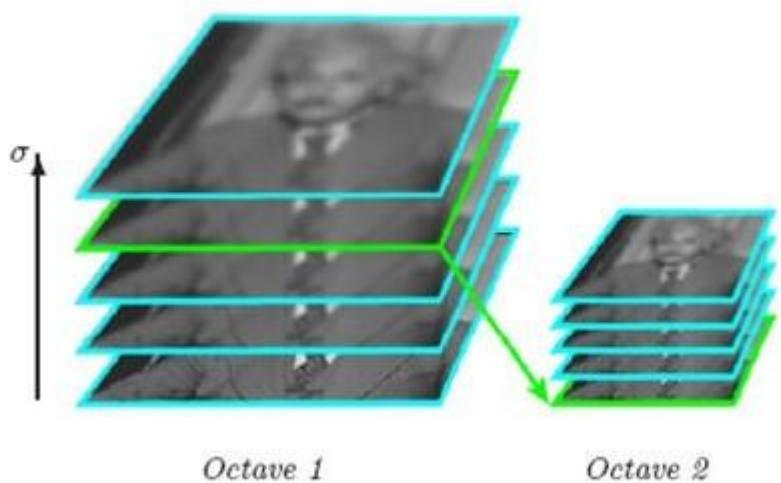
ok，耐心点，咱们再来总结一下上述内容：

1、高斯卷积

在组建一组尺度空间后，再组建下一组尺度空间，对上一组尺度空间的最后一幅图像进行二分之一采样，得到下一组尺度空间的第一幅图像，然后进行像建立第一组尺度空间那样的操作，得到第二组尺度空间，公式定义为

$$L(x,y,e) = G(x,y,e) * I(x,y)$$

图像金字塔的构建：图像金字塔共 O 组，每组有 S 层，下一组的图像由上一组图像降采样得到，效果图，图 A 如下(左为上一组，右为下一组)：

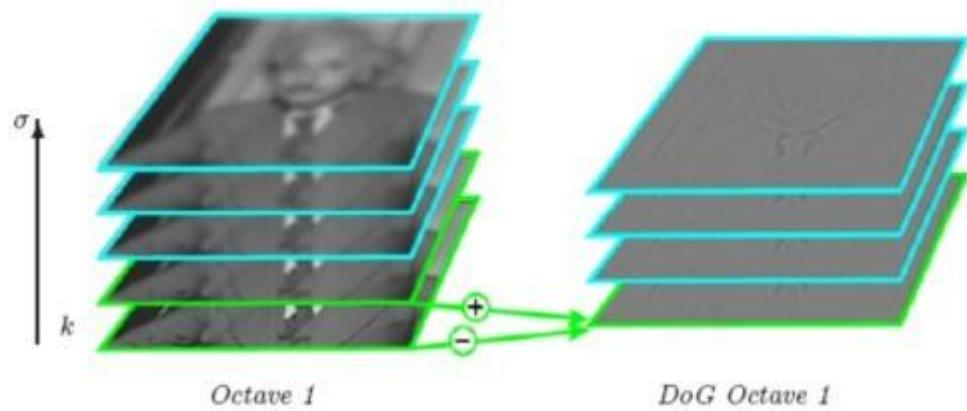


2、高斯差分

在尺度空间建立完毕后，为了能够找到稳定的关键点，采用高斯差分的方法来检测那些在局部位置的极值点，即采用两个相邻的尺度中的图像相减，即公式定义为：

$$\begin{aligned} D(x,y,e) &= ((G(x,y,ke) - G(x,y,e)) * I(x,y) \\ &= L(x,y,ke) - L(x,y,e) \end{aligned}$$

效果图，图 B：



SIFT 的精妙之处在于采用图像金字塔的方法解决这一问题，我们可以把两幅图像想象成是连续的，分别以它们作为底面作四棱锥，就像金字塔，那么每一个 截面与原图像相似，那么两个金字塔中必然会有包含大小一致的物体的无穷个截面，但应用只能是离散的，所以我们只能构造有限层，层数越多当然越好，但处理时间会相应增加，层数太少不行，因为向下采样的截面中可能找不到尺寸大小一致的两个物体的图像。

咱们再来具体阐述下构造 $\mathbf{D}(\mathbf{x}, \mathbf{y}, \mathbf{e})$ 的详细步骤：

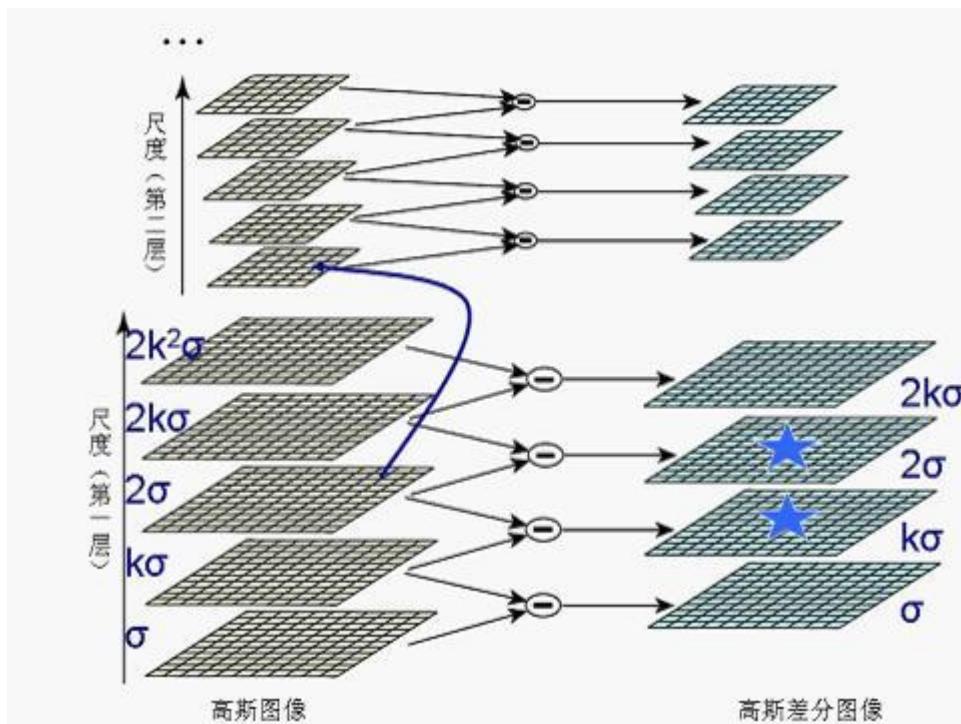
1、首先采用不同尺度因子的高斯核对图像进行卷积以得到图像的不同尺度空间，将这一组图像作为金子塔图像的第一层。

2、接着对第一层图像中的 2 倍尺度图像（相对于该层第一幅图像的 2 倍尺度）以 2 倍像素距离进行下采样来得到金子塔图像的第二层中的第一幅图像，对该图像采用不同尺度因子的高斯核进行卷积，以获得金字塔图像中第二层的一组图像。

3、再以金字塔图像中第二层中的 2 倍尺度图像（相对于该层第一幅图像的 2 倍尺度）以 2 倍像素距离进行下采样来得到金字塔图像的第三层中的第一幅图像，对该图像采用不同尺度因子的高斯核进行卷积，以获得金字塔图像中第三层的一组图像。这样依次类推，从而获得了金字塔图像的每一层中的一组图像，如下图所示：



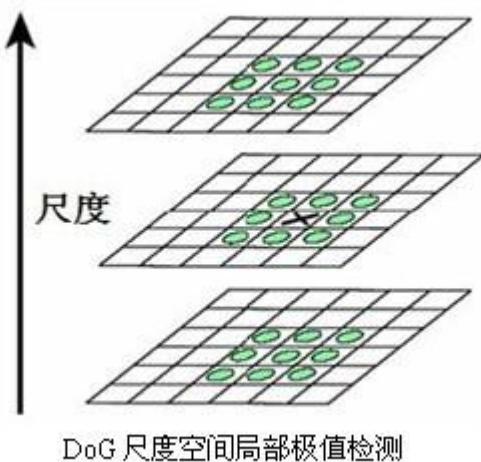
4、对上图得到的每一层相邻的高斯图像相减，就得到了高斯差分图像，如下述第一幅图所示。下述第二幅图中的右列显示了将每组中相邻图像相减所生成的高斯差分图像的结果，限于篇幅，图中只给出了第一层和第二层高斯差分图像的计算(下述俩幅图统称为图 2)：



5、因为高斯差分函数是归一化的高斯拉普拉斯函数的近似，所以可以从高斯差分金字塔分层结构提取出图像中的极值点作为候选的特征点。对 DOG 尺度空间每个点与相邻尺度和相邻位置的点逐个进行比较，得到的局部极值位置即为特征点所处的位置和对应的尺度。

二、检测关键点

为了寻找尺度空间的极值点，每一个采样点要和它所有的相邻点比较，看其是否比它的图像域和尺度域的相邻点大或者小。如下图，图 3 所示，中间的检测点和它同尺度的 8 个相邻点和上下相邻尺度对应的 9×2 个点共 26 个点比较，以确保在尺度空间和二维图像空间都检测到极值点。



因为需要同相邻尺度进行比较，所以在一组高斯差分图像中只能检测到两个尺度的极值点（如上述第二幅图中右图的五角星标识），而其它尺度的极值点检测则需要在图像金字塔的上一层高斯差分图像中进行。依次类推，最终在图像金字塔中不同层的高斯差分图像中完成不同尺度极值的检测。

当然这样产生的极值点并不都是稳定的特征点，因为某些极值点响应较弱，而且 DOG 算子会产生较强的边缘响应。

三、关键点方向的分配

为了使描述符具有旋转不变性，需要利用图像的局部特征为给每一个关键点分配一个方向。利用关键点邻域像素的梯度及方向分布的特性，可以得到梯度模值和方向如下：

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\theta(x, y) = \tan^{-1}((L(x, y+1) - L(x, y-1)) / (L(x+1, y) - L(x-1, y)))$$

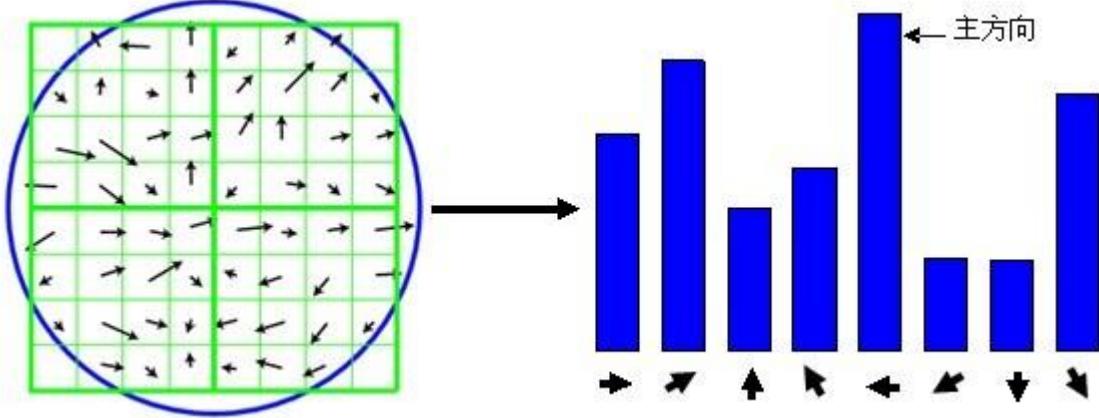
其中，尺度为每个关键点各自所在的尺度。

在以关键点为中心的邻域窗口内采样，并用直方图统计邻域像素的梯度方向。梯度直方图的范围是 0~360 度，其中每 10 度一个方向，总共 36 个方向。

直方图的峰值则代表了该关键点处邻域梯度的主方向，即作为该关键点的方向。

在计算方向直方图时，需要用一个参数等于关键点所在尺度 1.5 倍的高斯权重窗对方向直方图进行加权，上图中用蓝色的圆形表示，中心处的蓝色较重，表示权值最大，边缘处颜色渐变，表示权值小。如下图所示，该示例中为了简化给出了 8 方向的方向直方图计算结

果，实际 sift 创始人 David Lowe 的原论文中采用 36 方向的直方图。



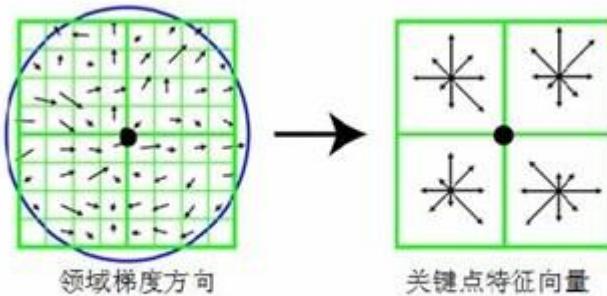
方向直方图的峰值则代表了该特征点处邻域梯度的方向，以直方图中最大值作为该关键点的主方向。为了增强匹配的鲁棒性，只保留峰值大于主方向峰值 80% 的方向作为该关键点的辅方向。因此，对于同一梯度值的多个峰值的关键点位置，在相同位置和尺度将会有多个关键点被创建但方向不同。仅有 15% 的关键点被赋予多个方向，但可以明显的提高关键点匹配的稳定性。

至此，图像的关键点已检测完毕，每个关键点有三个信息：位置、所处尺度、方向。由此可以确定一个 SIFT 特征区域。

四、特征点描述符

通过以上步骤，对于每一个关键点，拥有三个信息：位置、尺度以及方向。接下来就是为每个关键点建立一个描述符，使其不随各种变化而改变，比如光照变化、视角变化等等。并且描述符应该有较高的独特性，以便于提高特征点正确匹配的概率。

首先将坐标轴旋转为关键点的方向，以确保旋转不变性。



接下来以关键点为中心取 8×8 的窗口。

上图，图 5 中左部分的中央黑点为当前关键点的位置，每个小格代表关键点邻域所在尺度空间的一个像素，箭头方向代表该像素的梯度方向，箭头长度代表梯度模值，图中蓝色的圈代表高斯加权的范围（越靠近关键点的像素梯度方向信息贡献越大）。

然后在每 4×4 的小块上计算 8 个方向的梯度方向直方图，绘制每个梯度方向的累加值，即可形成一个种子点，如图 5 右部分所示。此图中一个关键点由 2×2 共 4 个种子点组成，

每个种子点有 **8** 个方向向量信息。这种邻域方向性信息联合的思想增强了算法抗噪声的能力，同时对于含有定位误差的特征匹配也提供了较好的容错性。

实际计算过程中，为了增强匹配的稳健性，**Lowe** 建议对每个关键点使用 4×4 共 **16** 个种子点来描述，这样对于一个关键点就可以产生 **128** 个数据，即最终形成 **128** 维的 **SIFT** 特征向量。此时 **SIFT** 特征向量已经去除了尺度变化、旋转等几何变形因素的影响，再继续将特征向量的长度归一化，则可以进一步去除光照变化的影响。

五、最后一步：当两幅图像的 **SIFT** 特征向量生成后，下一步我们采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。取上图中，**图像 A** 中的某个关键点，并找出其与**图像 B** 中欧式距离最近的前两个关键点，在这两个关键点中，如果最近的距离除以次近的距离少于某个比例阈值，则接受这一对匹配点。降低这个比例阈值，**SIFT** 匹配点数目会减少，但更加稳定。

sift 算法的逐步 c 实现

ok，上文搅了那么多的理论，如果你没有看懂它，咋办列？没关系，下面，咱们来一步一步实现此 **sift** 算法，即使你没有看到上述的理论，慢慢的，你也会明白 **sift** 算法到底是怎么一回事，**sift** 算法到底是怎么实现的....。

yeah，请看：

前期工作：

在具体编写核心函数之前，得先做几个前期的准备工作：

0、头文件：

[view plain](#)[copy to clipboard](#)[print](#)

```
1. #ifndef _CH_
2. #pragma package <opencv>
3. #endif
4.
5. #ifndef _EiC
6. #include <stdio.h>
7.
8. #include "stdlib.h"
9. #include "string.h"
10. #include "malloc.h"
11. #include "math.h"
12. #include <assert.h>
13. #include <ctype.h>
14. #include <time.h>
15. #include <cv.h>
```

```
16. #include <cxcore.h>
17. #include <highgui.h>
18. #include <vector>
19. #endif
20.
21. #ifdef _EIC
22. #define WIN32
23. #endif
```

1、定义几个宏，及变量，以免下文函数中，突然冒出一个变量，而您却不知道怎么一回事：

[view plain](#)[copy to clipboard](#)[print](#)

```
1. #define NUMSIZE 2
2. #define GAUSSKERN 3.5
3. #define PI 3.14159265358979323846
4.
5. //Sigma of base image -- See D.L.'s paper.
6. #define INITSIGMA 0.5
7. //Sigma of each octave -- See D.L.'s paper.
8. #define SIGMA sqrt(3)//1.6//
9.
10. //Number of scales per octave. See D.L.'s paper.
11. #define SCALESPEROCTAVE 2
12. #define MAXOCTAVES 4
13. int numoctaves;
14.
15. #define CONTRAST_THRESHOLD 0.02
16. #define CURVATURE_THRESHOLD 10.0
17. #define DOUBLE_BASE_IMAGE_SIZE 1
18. #define peakRelThresh 0.8
19. #define LEN 128
20.
21. // temporary storage
22. CvMemStorage* storage = 0;
```

2、然后，咱们还得，声明几个变量，以及建几个数据结构（数据结构是一切程序事物的基础嘛，：D。）：

[view plain](#)[copy to clipboard](#)[print](#)

```
1. //Data structure for a float image.
2. typedef struct ImageSt {           /*金字塔每一层*/
3.
4.     float levelsigma;
```

```

5. int levelsigmaLength;
6. float absolute_sigma;
7. CvMat *Level; //CvMat 是 OPENCV 的矩阵类, 其元素可以是图像的像素值
8. } ImageLevels;
9.

10. typedef struct ImageSt1 { /*金字塔每一阶梯*/
11.     int row, col; //Dimensions of image.
12.     float subsample;
13.     ImageLevels *Octave;
14. } ImageOctaves;
15.

16. ImageOctaves *DOGOctaves;
17. //DOG pyr, DOG 算子计算简单, 是尺度归一化的 Log 算子的近似。
18.

19. ImageOctaves *mag_thresh ;
20. ImageOctaves *mag_pyr ;
21. ImageOctaves *grad_pyr ;
22.

23. //keypoint 数据结构, Lists of keypoints are linked by the "next" field.
24. typedef struct KeypointSt
25. {
26.     float row, col; /* 反馈回原图像大小, 特征点的位置 */
27.     float sx,sy; /* 金字塔中特征点的位置*/
28.     int octave,level; /*金字塔中, 特征点所在的阶梯、层次*/
29.

30.     float scale, ori,mag; /*所在层的尺度 sigma, 主方向
   orientation (range [-PI,PI]), 以及幅值*/
31.     float *descrip; /*特征描述字指针: 128 维或 32 维等*/
32.     struct KeypointSt *next; /* Pointer to next keypoint in list. */
33. } *Keypoint;
34.

35. //定义特征点具体变量
36. Keypoint keypoints=NULL; //用于临时存储特征点的位置等
37. Keypoint keyDescriptors=NULL; //用于最后的确定特征点以及特征描述字

```

3、声明几个图像的基本处理函数:

图像处理基本函数, 其实也可以用 OPENCV 的函数代替, 但本文, 咱们选择了用 c 语言实现, 尽量不用第三方库的东西, 所以, 还得自己编写这些函数:

[view plain](#) [copy to clipboard](#) [print](#)

```

1. CvMat * halfSizeImage(CvMat * im); //缩小图像: 下采样
2. CvMat * doubleSizeImage(CvMat * im); //扩大图像: 最近邻方法
3. CvMat * doubleSizeImage2(CvMat * im); //扩大图像: 线性插值
4. float getPixelBI(CvMat * im, float col, float row); //双线性插值函数

```

```

5. void normalizeVec(float* vec, int dim); //向量归一化
6. CvMat* GaussianKernel2D(float sigma); //得到 2 维高斯核
7. void normalizeMat(CvMat* mat); //矩阵归一化
8. float* GaussianKernel1D(float sigma, int dim); //得到 1 维高斯核
9.
10. //在具体像素处宽度方向进行高斯卷积
11. float ConvolveLocWidth(float* kernel, int dim, CvMat * src, int x, int y);

12. //在整个图像宽度方向进行 1D 高斯卷积
13. void Convolve1DWidth(float* kern, int dim, CvMat * src, CvMat * dst);

14. //在具体像素处高度方向进行高斯卷积
15. float ConvolveLocHeight(float* kernel, int dim, CvMat * src, int x, int y);

16. //在整个图像高度方向进行 1D 高斯卷积
17. void Convolve1DHeight(float* kern, int dim, CvMat * src, CvMat * dst);

18. //用高斯函数模糊图像
19. int BlurImage(CvMat * src, CvMat * dst, float sigma);

```

算法核心

本程序中，**sift** 算法被分为以下五个步骤及其相对应的函数（可能表述与上，或与前俩篇文章有所偏差，但都一个意思）：

[view plain](#)[copy to clipboard](#)[print](#)

```

1. //SIFT 算法第一步：图像预处理
2. CvMat *ScaleInitImage(CvMat * im); //金字塔初始化
3.
4. //SIFT 算法第二步：建立高斯金字塔函数
5. ImageOctaves* BuildGaussianOctaves(CvMat * image); //建立高斯金字塔
6.
7. //SIFT 算法第三步：特征点位置检测，最后确定特征点的位置
8. int DetectKeypoint(int numoctaves, ImageOctaves *GaussianPyr);
9. void DisplayKeypointLocation(IplImage* image, ImageOctaves *GaussianPyr);
10.
11. //SIFT 算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征点的主方向
12. void ComputeGrad_DirecandMag(int numoctaves, ImageOctaves *GaussianPyr);
13.
14. int FindClosestRotationBin (int binCount, float angle); //进行方向直方图统计
15. void AverageWeakBins (double* bins, int binCount); //对方向直方图滤波
16. //确定真正的主方向
17. bool InterpolateOrientation (double left, double middle, double right, double
    *degreeCorrection, double *peakValue);

```

```

18. //确定各个特征点处的主方向函数
19. void AssignTheMainOrientation(int numoctaves, ImageOctaves *GaussianPyr, ImageOctaves *mag_pyr, ImageOctaves *grad_pyr);
20. //显示主方向
21. void DisplayOrientation (IplImage* image, ImageOctaves *GaussianPyr);
22.
23. //SIFT 算法第五步：抽取各个特征点处的特征描述字
24. void ExtractFeatureDescriptors(int numoctaves, ImageOctaves *GaussianPyr);
25.
26. //为了显示图象金字塔，而作的图像水平、垂直拼接
27. CvMat* MosaicHorizen( CvMat* im1, CvMat* im2 );
28. CvMat* MosaicVertical( CvMat* im1, CvMat* im2 );
29.
30. //特征描述点，网格
31. #define GridSpacing 4

```

主体实现

ok，以上所有的工作都就绪以后，那么接下来，咱们就先来编写 `main` 函数，因为你一看主函数之后，你就立马能发现 `sift` 算法的工作流程及其原理了。

（主函数中涉及到的函数，下一篇章：[一、教你一步一步用 c 语言实现 sift 算法](#)、下，咱们自会一个一个编写）：

[view plain](#)[copy to clipboard](#)[print?](#)

```

1. int main( void )
2. {
3.     //声明当前帧 IplImage 指针
4.     IplImage* src = NULL;
5.     IplImage* image1 = NULL;
6.     IplImage* grey_im1 = NULL;
7.     IplImage* DoubleSizeImage = NULL;
8.
9.     IplImage* mosaic1 = NULL;
10.    IplImage* mosaic2 = NULL;
11.
12.    CvMat* mosaicHorizen1 = NULL;
13.    CvMat* mosaicHorizen2 = NULL;
14.    CvMat* mosaicVertical1 = NULL;
15.
16.    CvMat* image1Mat = NULL;
17.    CvMat* tempMat=NULL;
18.
19.    ImageOctaves *Gaussianpyr;
20.    int rows,cols;

```

```

21.
22. #define Im1Mat(ROW,COL) ((float *) (image1Mat->data.fl + image1Mat->step/size
   of(float) *(ROW)))[(COL)]
23.
24. //灰度图象像素的数据结构
25. #define Im1B(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3]
26. #define Im1G(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3+1]
27. #define Im1R(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3+2]
28.
29. storage = cvCreateMemStorage(0);
30.
31. //读取图片
32. if( (src = cvLoadImage( "street1.jpg", 1)) == 0 ) // test1.jpg einstein.jpg
   m back1.bmp
33. return -1;
34.
35. //为图像分配内存
36. image1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U,3);
37. grey_im1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U,1);

38. DoubleSizeImage = cvCreateImage(cvSize(2*(src->width), 2*(src->height)), I
   PL_DEPTH_8U,3);
39.
40. //为图像阵列分配内存, 假设两幅图像的大小相同, tempMat 跟随 image1 的大小
41. image1Mat = cvCreateMat(src->height, src->width, CV_32FC1);
42. //转化成单通道图像再处理
43. cvCvtColor(src, grey_im1, CV_BGR2GRAY);
44. //转换进入 Mat 数据结构, 图像操作使用的是浮点型操作
45. cvConvert(grey_im1, image1Mat);
46.
47. double t = (double)cvGetTickCount();
48. //图像归一化
49. cvConvertScale( image1Mat, image1Mat, 1.0/255, 0 );
50.
51. int dim = min(image1Mat->rows, image1Mat->cols);
52. numoctaves = (int) (log((double) dim) / log(2.0)) - 2; //金字塔阶数
53. numoctaves = min(numoctaves, MAXOCTAVES);
54.
55. //SIFT 算法第一步, 预滤波除噪声, 建立金字塔底层
56. tempMat = ScaleInitImage(image1Mat) ;
57. //SIFT 算法第二步, 建立 Gaussian 金字塔和 DOG 金字塔

```

```

58. Gaussianpyr = BuildGaussianOctaves(tempMat) ;
59.
60. t = (double)cvGetTickCount() - t;
61. printf( "the time of build Gaussian pyramid and DOG pyramid is %.1f\n", t/
   cvGetTickFrequency()*1000.) );
62.
63. #define ImLevels(OCTAVE,LEVEL,ROW,COL) ((float *) (Gaussianpyr[(OCTAVE)].Octa
   ve[(LEVEL)].Level->data.fl + Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level->st
   ep/sizeof(float) *(ROW)))[(COL)]
64. //显示高斯金字塔
65. for (int i=0; i<numoctaves;i++)
66. {
67.   if (i==0)
68.   {
69.     mosaicHorizen1=MosaicHorizen( (Gaussianpyr[0].Octave)[0].Level, (Gaussian
   pyr[0].Octave)[1].Level );
70.     for (int j=2;j<SCALESPEROCTAVE+3;j++)
71.       mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[0].Octave)[j]
   .Level );
72.     for ( j=0;j<NUMSIZE;j++)
73.       mosaicHorizen1=halfSizeImage(mosaicHorizen1);
74.   }
75.   else if (i==1)
76.   {
77.     mosaicHorizen2=MosaicHorizen( (Gaussianpyr[1].Octave)[0].Level, (Gaussian
   pyr[1].Octave)[1].Level );
78.     for (int j=2;j<SCALESPEROCTAVE+3;j++)
79.       mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (Gaussianpyr[1].Octave)[j]
   .Level );
80.     for ( j=0;j<NUMSIZE;j++)
81.       mosaicHorizen2=halfSizeImage(mosaicHorizen2);
82.     mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );
83.   }
84.   else
85.   {
86.     mosaicHorizen1=MosaicHorizen( (Gaussianpyr[i].Octave)[0].Level, (Gaussian
   pyr[i].Octave)[1].Level );
87.     for (int j=2;j<SCALESPEROCTAVE+3;j++)
88.       mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[i].Octave)[j]
   .Level );
89.     for ( j=0;j<NUMSIZE;j++)
90.       mosaicHorizen1=halfSizeImage(mosaicHorizen1);
91.     mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );
92.   }

```

```

93. }
94. mosaic1 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1->height), IPL_DEPTH_8U,1);
95. cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0, 0 );
96. cvConvertScaleAbs( mosaicVertical1, mosaic1, 1, 0 );
97.
98. // cvSaveImage("GaussianPyramid of me.jpg",mosaic1);
99. cvNamedWindow("mosaic1",1);
100. cvShowImage("mosaic1", mosaic1);
101. cvWaitKey(0);
102. cvDestroyWindow("mosaic1");
103. //显示DOG金字塔
104. for ( i=0; i<numoctaves;i++)
105. {
106. if (i==0)
107. {
108. mosaicHorizen1=MosaicHorizen( (DOGoctaves[0].Octave)[0].Level, (DOGoctaves[0].Octave)[1].Level );
109. for (int j=2;j<SCALESPEROCTAVE+2;j++)
110. mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[0].Octave)[j].Level );
111. for ( j=0;j<NUMSIZE;j++)
112. mosaicHorizen1=halfSizeImage(mosaicHorizen1);
113. }
114. else if (i==1)
115. {
116. mosaicHorizen2=MosaicHorizen( (DOGoctaves[1].Octave)[0].Level, (DOGoctaves[1].Octave)[1].Level );
117. for (int j=2;j<SCALESPEROCTAVE+2;j++)
118. mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (DOGoctaves[1].Octave)[j].Level );
119. for ( j=0;j<NUMSIZE;j++)
120. mosaicHorizen2=halfSizeImage(mosaicHorizen2);
121. mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );
122. }
123. else
124. {
125. mosaicHorizen1=MosaicHorizen( (DOGoctaves[i].Octave)[0].Level, (DOGoctaves[i].Octave)[1].Level );
126. for (int j=2;j<SCALESPEROCTAVE+2;j++)
127. mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[i].Octave)[j].Level );
128. for ( j=0;j<NUMSIZE;j++)
129. mosaicHorizen1=halfSizeImage(mosaicHorizen1);

```

```

130.    mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );
131. }
132. }
133. //考虑到 DOG 金字塔各层图像都会有正负, 所以, 必须寻找最负的, 以将所有图像抬高一个台阶去显示
134. double min_val=0;
135. double max_val=0;
136. cvMinMaxLoc( mosaicVertical1, &min_val, &max_val,NULL, NULL, NULL );
137. if ( min_val<0.0 )
138. cvAddS( mosaicVertical1, cvScalarAll( (-1.0)*min_val ), mosaicVertical1,
NULL );
139. mosaic2 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1->height),
IPL_DEPTH_8U,1);
140. cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0/(max_val-min_val),
0 );
141. cvConvertScaleAbs( mosaicVertical1, mosaic2, 1, 0 );
142.
143. // cvSaveImage("DOGPyramid of me.jpg",mosaic2);
144. cvNamedWindow("mosaic1",1);
145. cvShowImage("mosaic1", mosaic2);
146. cvWaitKey(0);
147.
148. //SIFT 算法第三步: 特征点位置检测, 最后确定特征点的位置
149. int keycount=DetectKeypoint(numoctaves, Gaussianpyr);
150. printf("the keypoints number are %d ;\n", keycount);
151. cvCopy(src,image1,NULL);
152. DisplayKeypointLocation( image1 ,Gaussianpyr);
153.
154. cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
155. cvNamedWindow("image1",1);
156. cvShowImage("image1", DoubleSizeImage);
157. cvWaitKey(0);
158. cvDestroyWindow("image1");
159.
160. //SIFT 算法第四步: 计算高斯图像的梯度方向和幅值, 计算各个特征点的主方向
161. ComputeGrad_DirecandMag(numoctaves, Gaussianpyr);
162. AssignTheMainOrientation( numoctaves, Gaussianpyr,mag_pyr,grad_pyr);
163. cvCopy(src,image1,NULL);
164. DisplayOrientation ( image1, Gaussianpyr);
165.
166. // cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
167. cvNamedWindow("image1",1);
168. // cvResizeWindow("image1", 2*(image1->width), 2*(image1->height) );
169. cvShowImage("image1", image1);

```

```
170. cvWaitKey(0);
171.
172. //SIFT 算法第五步：抽取各个特征点处的特征描述字
173. ExtractFeatureDescriptors( numoctaves, Gaussianpyr);
174. cvWaitKey(0);
175.
176. //销毁窗口
177. cvDestroyWindow("image1");
178. cvDestroyWindow("mosaic1");
179. //释放图像
180. cvReleaseImage(&image1);
181. cvReleaseImage(&grey_im1);
182. cvReleaseImage(&mosaic1);
183. cvReleaseImage(&mosaic2);
184. return 0;
185. }
```

更多见下文：[一、教你一步一步用 c 语言实现 sift 算法、下](#)。本文完。

版权声明：

本文版权归本人和 CSDN 共同拥有。转载，请注明出处及作者本人。

侵犯版权者，无论任何人，任何网站，**1、永久追踪，2、永久谴责，3、永久追究法律责任的权利。**

July、二零一一年三月十二日声明。

九（再续）、教你一步一步用 c 语言实现 sift 算法、下

作者：July、二零一一年三月十二日

出处：http://blog.csdn.net/v_JULY_v。

参考：Rob Hess 维护的 sift 库

环境：windows xp+vc6.0

条件：c 语言实现。

说明：本 BLOG 内会陆续一一实现所有经典算法。

本文接上，[教你一步一步用 c 语言实现 sift 算法、上](#)，而来：

函数编写

ok，接上文，咱们一个一个的来编写 main 函数中所涉及到所有函数，这也是本文的关键部分：

[view plain](#)[copy to clipboard](#)[print?](#)

```
1. //下采样原来的图像，返回缩小 2 倍尺寸的图像
2. CvMat * halfSizeImage(CvMat * im)
3. {
4.     unsigned int i,j;
5.     int w = im->cols/2;
6.     int h = im->rows/2;
7.     CvMat *imnew = cvCreateMat(h, w, CV_32FC1);
8.
9. #define Im(ROW,COL) ((float *) (im->data.fl + im->step/sizeof(float) *(ROW)))
   [(COL)]
10. #define Imnew(ROW,COL) ((float *) (imnew->data.fl + imnew->step/sizeof(float)
    *(ROW)))[(COL)]
11. for ( j = 0; j < h; j++)
12.     for ( i = 0; i < w; i++)
13.         Imnew(j,i)=Im(j*2, i*2);
14.     return imnew;
15. }
16.
17. //上采样原来的图像，返回放大 2 倍尺寸的图像
18. CvMat * doubleSizeImage(CvMat * im)
19. {
20.     unsigned int i,j;
21.     int w = im->cols*2;
22.     int h = im->rows*2;
23.     CvMat *imnew = cvCreateMat(h, w, CV_32FC1);
24.
25. #define Im(ROW,COL) ((float *) (im->data.fl + im->step/sizeof(float) *(ROW)))
   [(COL)]
26. #define Imnew(ROW,COL) ((float *) (imnew->data.fl + imnew->step/sizeof(float)
    *(ROW)))[(COL)]
27.
28. for ( j = 0; j < h; j++)
29.     for ( i = 0; i < w; i++)
30.         Imnew(j,i)=Im(j/2, i/2);
31.
32.     return imnew;
33. }
34.
```

```

35. //上采样原来的图像，返回放大 2 倍尺寸的线性插值图像
36. CvMat * doubleSizeImage2(CvMat * im)
37. {
38.     unsigned int i,j;
39.     int w = im->cols*2;
40.     int h = im->rows*2;
41.     CvMat *imnew = cvCreateMat(h, w, CV_32FC1);
42.
43. #define Im(ROW,COL) ((float *) (im->data.fl + im->step/sizeof(float) *(ROW)))
44. #define Imnew(ROW,COL) ((float *) (imnew->data.fl + imnew->step/sizeof(float)
45.                         *(ROW)))[(COL)]
46. // fill every pixel so we don't have to worry about skipping pixels later
47. for ( j = 0; j < h; j++)
48. {
49.     for ( i = 0; i < w; i++)
50.     {
51.         Imnew(j,i)=Im(j/2, i/2);
52.     }
53. }
54. /*
55. A B C
56. E F G
57. H I J
58. pixels A C H J are pixels from original image
59. pixels B E G I F are interpolated pixels
60. */
61. // interpolate pixels B and I
62. for ( j = 0; j < h; j += 2)
63.     for ( i = 1; i < w - 1; i += 2)
64.         Imnew(j,i)=0.5*(Im(j/2, i/2)+Im(j/2, i/2+1));
65. // interpolate pixels E and G
66. for ( j = 1; j < h - 1; j += 2)
67.     for ( i = 0; i < w; i += 2)
68.         Imnew(j,i)=0.5*(Im(j/2, i/2)+Im(j/2+1, i/2));
69. // interpolate pixel F
70. for ( j = 1; j < h - 1; j += 2)
71.     for ( i = 1; i < w - 1; i += 2)
72.         Imnew(j,i)=0.25*(Im(j/2, i/2)+Im(j/2+1, i/2)+Im(j/2, i/2+1)+Im(j/2+1,
73.                         i/2+1));
73.     return imnew;
74. }
75.

```

```
76. //双线性插值，返回像素间的灰度值
77. float getPixelBI(CvMat * im, float col, float row)
78. {
79.     int irow, icol;
80.     float rfrac, cfrac;
81.     float row1 = 0, row2 = 0;
82.     int width=im->cols;
83.     int height=im->rows;
84. #define ImMat(ROW,COL) ((float *) (im->data.fl + im->step/sizeof(float) *(ROW
85. )))[(COL)]
86.     irow = (int) row;
87.     icol = (int) col;
88.
89.     if (irow < 0 || irow >= height
90.         || icol < 0 || icol >= width)
91.         return 0;
92.     if (row > height - 1)
93.         row = height - 1;
94.     if (col > width - 1)
95.         col = width - 1;
96.     rfrac = 1.0 - (row - (float) irow);
97.     cfrac = 1.0 - (col - (float) icol);
98.     if (cfrac < 1)
99.     {
100.         row1 = cfrac * ImMat(irow,icol) + (1.0 - cfrac) * ImMat(irow,icol+1);
101.     }
102.     else
103.     {
104.         row1 = ImMat(irow,icol);
105.     }
106.     if (rfrac < 1)
107.     {
108.         if (cfrac < 1)
109.         {
110.             row2 = cfrac * ImMat(irow+1,icol) + (1.0 - cfrac) * ImMat(irow+1,icol+1)
111.             ;
112.         } else
113.         {
114.             row2 = ImMat(irow+1,icol);
115.         }
116.     return rfrac * row1 + (1.0 - rfrac) * row2;
117. }
```

```

118.
119. //矩阵归一化
120. void normalizeMat(CvMat* mat)
121. {
122. #define Mat(ROW,COL) ((float * )(mat->data.fl + mat->step/sizeof(float) *(RO
   W)))[(COL)]
123. float sum = 0;
124.
125. for (unsigned int j = 0; j < mat->rows; j++)
126. for (unsigned int i = 0; i < mat->cols; i++)
127. sum += Mat(j,i);
128. for ( j = 0; j < mat->rows; j++)
129. for (unsigned int i = 0; i < mat->rows; i++)
130. Mat(j,i) /= sum;
131. }
132.
133. //向量归一化
134. void normalizeVec(float* vec, int dim)
135. {
136.     unsigned int i;
137.     float sum = 0;
138.     for ( i = 0; i < dim; i++)
139.         sum += vec[i];
140.     for ( i = 0; i < dim; i++)
141.         vec[i] /= sum;
142. }
143.
144. //得到向量的欧式长度, 2-范数
145. float GetVecNorm( float* vec, int dim )
146. {
147.     float sum=0.0;
148.     for (unsigned int i=0;i<dim;i++)
149.         sum+=vec[i]*vec[i];
150.     return sqrt(sum);
151. }
152.
153. //产生 1D 高斯核
154. float* GaussianKernel1D(float sigma, int dim)
155. {
156.
157.     unsigned int i;
158.     //printf("GaussianKernel1D(): Creating 1x%d vector for sigma=%.3f gaussian
   kernel\n", dim, sigma);
159.

```

```

160. float *kern=(float*)malloc( dim*sizeof(float) );
161. float s2 = sigma * sigma;
162. int c = dim / 2;
163. float m= 1.0/(sqrt(2.0 * CV_PI) * sigma);
164.     double v;
165. for ( i = 0; i < (dim + 1) / 2; i++)
166. {
167.     v = m * exp(-(1.0*i*i)/(2.0 * s2)) ;
168.     kern[c+i] = v;
169.     kern[c-i] = v;
170. }
171. // normalizeVec(kern, dim);
172. // for ( i = 0; i < dim; i++)
173. // printf("%f ", kern[i]);
174. // printf("\n");
175. return kern;
176. }
177.
178. //产生 2D 高斯核矩阵
179. CvMat* GaussianKernel2D(float sigma)
180. {
181. // int dim = (int) max(3.0f, GAUSSKERN * sigma);
182.     int dim = (int) max(3.0f, 2.0 * GAUSSKERN *sigma + 1.0f);
183. // make dim odd
184. if (dim % 2 == 0)
185.     dim++;
186. //printf("GaussianKernel(): Creating %dx%d matrix for sigma=%.3f gaussian\
n", dim, dim, sigma);
187. CvMat* mat=cvCreateMat(dim, dim, CV_32FC1);
188. #define Mat(ROW,COL) ((float *) (mat->data.fl + mat->step/sizeof(float) *(RO\
W)))[(COL)]
189. float s2 = sigma * sigma;
190. int c = dim / 2;
191. //printf("%d %d\n", mat.size(), mat[0].size());
192.     float m= 1.0/(sqrt(2.0 * CV_PI) * sigma);
193. for (int i = 0; i < (dim + 1) / 2; i++)
194. {
195.     for (int j = 0; j < (dim + 1) / 2; j++)
196.     {
197.         //printf("%d %d %d\n", c, i, j);
198.         float v = m * exp(-(1.0*i*i + 1.0*j*j) / (2.0 * s2));
199.         Mat(c+i,c+j) =v;
200.         Mat(c-i,c+j) =v;
201.         Mat(c+i,c-j) =v;

```

```

202.     Mat(c-i,c-j) =v;
203.   }
204. }
205. // normalizeMat(mat);
206. return mat;
207. }
208.
209. //x 方向像素处作卷积
210. float ConvolveLocWidth(float* kernel, int dim, CvMat * src, int x, int y)

211. {
212. #define Src(ROW,COL) ((float * )(src->data.fl + src->step/sizeof(float) *(RO
    W)))[(COL)]
213.     unsigned int i;
214.     float pixel = 0;
215.     int col;
216.     int cen = dim / 2;
217. //printf("ConvolveLoc(): Applying convolution at location (%d, %d)\n", x,
    y);
218. for ( i = 0; i < dim; i++)
219. {
220.     col = x + (i - cen);
221.     if (col < 0)
222.         col = 0;
223.     if (col >= src->cols)
224.         col = src->cols - 1;
225.     pixel += kernel[i] * Src(y,col);
226. }
227. if (pixel > 1)
228.     pixel = 1;
229. return pixel;
230. }
231.
232. //x 方向作卷积
233. void Convolve1DWidth(float* kern, int dim, CvMat * src, CvMat * dst)
234. {
235. #define DST(ROW,COL) ((float * )(dst->data.fl + dst->step/sizeof(float) *(RO
    W)))[(COL)]
236.     unsigned int i,j;
237.
238. for ( j = 0; j < src->rows; j++)
239. {
240.     for ( i = 0; i < src->cols; i++)
241.     {

```

```

242. //printf("%d, %d\n", i, j);
243. DST(j,i) = ConvolveLocWidth(kern, dim, src, i, j);
244. }
245. }
246. }
247.
248. //y 方向像素处作卷积
249. float ConvolveLocHeight(float* kernel, int dim, CvMat * src, int x, int y)

250. {
251. #define Src(ROW,COL) ((float *) (src->data.fl + src->step/sizeof(float) *(RO
   W)))[(COL)]
252.     unsigned int j;
253.     float pixel = 0;
254.     int cen = dim / 2;
255. //printf("ConvolveLoc(): Applying convolution at location (%d, %d)\n", x,
   y);
256. for ( j = 0; j < dim; j++)
257. {
258.     int row = y + (j - cen);
259.     if (row < 0)
260.         row = 0;
261.     if (row >= src->rows)
262.         row = src->rows - 1;
263.     pixel += kernel[j] * Src(row,x);
264. }
265. if (pixel > 1)
266.     pixel = 1;
267. return pixel;
268. }
269.
270. //y 方向作卷积
271. void Convolve1DHeight(float* kern, int dim, CvMat * src, CvMat * dst)
272. {
273. #define Dst(ROW,COL) ((float *) (dst->data.fl + dst->step/sizeof(float) *(RO
   W)))[(COL)]
274.     unsigned int i,j;
275.     for ( j = 0; j < src->rows; j++)
276.     {
277.         for ( i = 0; i < src->cols; i++)
278.         {
279.             //printf("%d, %d\n", i, j);
280.             Dst(j,i) = ConvolveLocHeight(kern, dim, src, i, j);
281.         }

```

```

282. }
283. }
284.
285. //卷积模糊图像
286. int BlurImage(CvMat * src, CvMat * dst, float sigma)
287. {
288.     float* convkernel;
289.     int dim = (int) max(3.0f, 2.0 * GAUSSKERN * sigma + 1.0f);
290.     CvMat *tempMat;
291.     // make dim odd
292.     if (dim % 2 == 0)
293.         dim++;
294.     tempMat = cvCreateMat(src->rows, src->cols, CV_32FC1);
295.     convkernel = GaussianKernel1D(sigma, dim);
296.
297.     Convolve1DWidth(convkernel, dim, src, tempMat);
298.     Convolve1DHeight(convkernel, dim, tempMat, dst);
299.     cvReleaseMat(&tempMat);
300.     return dim;
301. }

```

五个步骤

ok, 接下来, 进入重点部分, 咱们依据上文介绍的 **sift** 算法的几个步骤, 来一一实现这些函数。

为了表述清晰, 再贴一下, 主函数, 顺便再加强下对 **sift** 算法的五个步骤的认识:

1、SIFT 算法第一步: 图像预处理

CvMat *ScaleInitImage(CvMat * im) ; //金字塔初始化

2、SIFT 算法第二步: 建立高斯金字塔函数

ImageOctaves* BuildGaussianOctaves(CvMat * image) ; //建立高斯金字塔

3、SIFT 算法第三步: 特征点位置检测, 最后确定特征点的位置

int DetectKeypoint(int numoctaves, ImageOctaves *GaussianPyr);

4、SIFT 算法第四步: 计算高斯图像的梯度方向和幅值, 计算各个特征点的主方向

void ComputeGrad_DirecandMag(int numoctaves, ImageOctaves *GaussianPyr);

5、SIFT 算法第五步: 抽取各个特征点处的特征描述字

void ExtractFeatureDescriptors(int numoctaves, ImageOctaves *GaussianPyr);

ok, 接下来一一具体实现这几个函数:

SIFT 算法第一步

SIFT 算法第一步: 扩大图像, 预滤波剔除噪声, 得到金字塔的最底层-第一阶的第一层:

[view plain](#)[copy to clipboard](#)[print](#)

```

1. CvMat *ScaleInitImage(CvMat * im)
2. {
3.     double sigma, preblur_sigma;
4.     CvMat *imMat;
5.     CvMat * dst;
6.     CvMat *tempMat;
7.     //首先对图像进行平滑滤波，抑制噪声
8.     imMat = cvCreateMat(im->rows, im->cols, CV_32FC1);
9.     BlurImage(im, imMat, INITSIGMA);
10.    //针对两种情况分别进行处理：初始化放大原始图像或者在原图像基础上进行后续操作
11.    //建立金字塔的最底层
12.    if (DOUBLE_BASE_IMAGE_SIZE)
13.    {
14.        tempMat = doubleSizeImage2(imMat); //对扩大两倍的图像进行二次采样，采样率为 0.5,
15.        //采用线性插值
16.        #define TEMPMAT(ROW,COL) ((float *) (tempMat->data.fl + tempMat->step/sizeof(
17.            float) * (ROW)))[(COL)]
18.        dst = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1);
19.        preblur_sigma = 1.0; //sqrt(2 - 4*INITSIGMA*INITSIGMA);
20.        BlurImage(tempMat, dst, preblur_sigma);
21.        // The initial blurring for the first image of the first octave of the pyr
22.        // amid.
23.        sigma = sqrt( (4*INITSIGMA*INITSIGMA) + preblur_sigma * preblur_sigma
24.            );
25.        // sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4);
26.        //printf("Init Sigma: %f\n", sigma);
27.        BlurImage(dst, tempMat, sigma); //得到金字塔的最底层-放大 2 倍的图像
28.        cvReleaseMat( &dst );
29.        return tempMat;
30.    }
31.    else
32.    {
33.        dst = cvCreateMat(im->rows, im->cols, CV_32FC1);
34.        //sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA);
35.        preblur_sigma = 1.0; //sqrt(2 - 4*INITSIGMA*INITSIGMA);
36.        sigma = sqrt( (4*INITSIGMA*INITSIGMA) + preblur_sigma * preblur_sigma );
37.        //printf("Init Sigma: %f\n", sigma);
38.        BlurImage(imMat, dst, sigma); //得到金字塔的最底层：原始图像大小
39.    }
}

```

SIFT 算法第二步

SIFT 第二步，建立 Gaussian 金字塔，给定金字塔第一阶第一层图像后，计算高斯金字塔其他尺度图像，

每一阶的数目由变量 SCALESPECTROCTAVE 决定，给定一个基本图像，计算它的高斯金字塔图像，返回外部向量是阶梯指针，内部向量是每一个阶梯内部的不同尺度图像。

[view plain](#)[copy to clipboard](#)[print?](#)

```
1. //SIFT 算法第二步
2. ImageOctaves* BuildGaussianOctaves(CvMat * image)
3. {
4.     ImageOctaves *octaves;
5.     CvMat *tempMat;
6.     CvMat *dst;
7.     CvMat *temp;
8.
9.     int i,j;
10.    double k = pow(2, 1.0/((float)SCALESPECTROCTAVE)); //方差倍数
11.    float preblur_sigma, initial_sigma , sigma1,sigma2,sigma,absolute_sigma,sigma_f;
12.    //计算金字塔的阶梯数目
13.    int dim = min(image->rows, image->cols);
14.    int numoctaves = (int) (log((double) dim) / log(2.0)) - 2; //金字塔阶
    数
15.    //限定金字塔的阶梯数
16.    numoctaves = min(numoctaves, MAXOCTAVES);
17.    //为高斯金字塔和 DOG 金字塔分配内存
18.    octaves=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );
19.    DOGoctaves=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );
20.
21.    printf("BuildGaussianOctaves(): Base image dimension is %dx%d\n", (int)(0.5
        *(image->cols)), (int)(0.5*(image->rows)) );
22.    printf("BuildGaussianOctaves(): Building %d octaves\n", numoctaves);
23.
24.    // start with initial source image
25.    tempMat=cvCloneMat( image );
26.    // preblur_sigma = 1.0;//sqrt(2 - 4*INITSIGMA*INITSIGMA);
27.    initial_sigma = sqrt(2);//sqrt( (4*INITSIGMA*INITSIGMA) + preblur_sigma
        * preblur_sigma );
28.    // initial_sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4);
29.
30.    //在每一阶金字塔图像中建立不同的尺度图像
```

```

31. for ( i = 0; i < numoctaves; i++)
32. {
33.     //首先建立金字塔每一阶梯的最底层，其中 0 阶梯的最底层已经建立好
34.     printf("Building octave %d of dimesion (%d, %d)\n", i, tempMat->cols,tempM
at->rows);
35.         //为各个阶梯分配内存
36.     octaves[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE + 3) * sizeof(I
mageLevels) );
37.     DOGoctaves[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE + 2) * sizeof(I
mageLevels) );
38.     //存储各个阶梯的最底层
39.     (octaves[i].Octave)[0].Level=tempMat;
40.
41.     octaves[i].col=tempMat->cols;
42.     octaves[i].row=tempMat->rows;
43.     DOGoctaves[i].col=tempMat->cols;
44.     DOGoctaves[i].row=tempMat->rows;
45.     if (DOUBLE_BASE_IMAGE_SIZE)
46.         octaves[i].subsample=pow(2,i)*0.5;
47.     else
48.         octaves[i].subsample=pow(2,i);
49.
50.     if(i==0)
51.     {
52.         (octaves[0].Octave)[0].levelsigma = initial_sigma;
53.         (octaves[0].Octave)[0].absolute_sigma = initial_sigma;
54.         printf("0 scale and blur sigma : %f \n", (octaves[0].subsample) * ((octav
es[0].Octave)[0].absolute_sigma));
55.     }
56.     else
57.     {
58.         (octaves[i].Octave)[0].levelsigma = (octaves[i-1].Octave)[SCALESPEROCTAVE
].levelsigma;
59.         (octaves[i].Octave)[0].absolute_sigma = (octaves[i-1].Octave)[SC
ALESPEROCTAVE].absolute_sigma;
60.         printf( "0 scale and blur sigma : %f \n", ((octaves[i].Octave)[0].absolut
e_sigma) );
61.     }
62.     sigma = initial_sigma;
63.         //建立本阶梯其他层的图像
64.     for ( j = 1; j < SCALESPEROCTAVE + 3; j++)
65.     {
66.         dst = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1); //用于存储高斯
层

```

```

67.     temp = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1); //用于存储 DOG
    层
68.     // 2 passes of 1D on original
69.     // if(i!=0)
70.     // {
71.         // sigma1 = pow(k, j - 1) * ((octaves[i-1].Octave)[j-1].levelsigma)
72.         ;
73.         // sigma2 = pow(k, j) * ((octaves[i].Octave)[j-1].levelsigma);
74.         sigma = sqrt(sigma2*sigma2 - sigma1*sigma1);
75.         sigma_f= sqrt(k*k-1)*sigma;
76.     }
77.     // else
78.     // {
79.         // sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4)*pow(k,j)
80.         ;
81.         sigma = k*sigma;
82.         absolute_sigma = sigma * (octaves[i].subsample);
83.         printf("%d scale and Blur sigma: %f \n", j, absolute_sigma);
84.         (octaves[i].Octave)[j].levelsigma = sigma;
85.         (octaves[i].Octave)[j].absolute_sigma = absolute_sigma;
86.         //产生高斯层
87.         int length=BlurImage((octaves[i].Octave)[j-1].Level, dst, sigma_f); //相应
    尺度
88.         (octaves[i].Octave)[j].levelsigmalength = length;
89.         (octaves[i].Octave)[j].Level=dst;
90.         //产生 DOG 层
91.         cvSub( ((octaves[i].Octave)[j]).Level, ((octaves[i].Octave)[j-1]
    ).Level, temp, 0 );
92.         // cvAbsDiff( ((octaves[i].Octave)[j]).Level, ((octaves[i].Octave
    )[j-1]).Level, temp );
93.         ((DOGoctaves[i].Octave)[j-1]).Level=temp;
94.     }
95.     // halve the image size for next iteration
96.     tempMat = halfSizeImage( ( (octaves[i].Octave)[SCALESPEROCTAVE].Level ) )
    ;
97. }
98. return octaves;
99. }

```

SIFT 算法第三步

SIFT 算法第三步，特征点位置检测，最后确定特征点的位置检测 DOG 金字塔中的局

部最大值，找到之后，还要经过两个检验才能确认为特征点：一是它必须有明显的差异，二是他不应该是边缘点，（也就是说，在极值点处的主曲率比应该小于某一个阈值）。

[view plain](#)[copy to clipboard](#)[print](#)

```
1. //SIFT 算法第三步，特征点位置检测,
2. int DetectKeypoint(int numoctaves, ImageOctaves *GaussianPyr)
3. {
4.     //计算用于 DOG 极值点检测的主曲率比的阈值
5.     double curvature_threshold;
6.     curvature_threshold= ((CURVATURE_THRESHOLD + 1)*(CURVATURE_THRESHOLD + 1))/CURVATURE_THRESHOLD;
7.     #define ImLevels(OCTAVE, LEVEL, ROW, COL) ((float *) (DOGoctaves[(OCTAVE)].Octave[(LEVEL)].Level->data.fl + DOGoctaves[(OCTAVE)].Octave[(LEVEL)].Level->step / sizeof(float) *(ROW)))[(COL)]
8.
9.     int keypoint_count = 0;
10.    for (int i=0; i<numoctaves; i++)
11.    {
12.        for(int j=1;j<SCALESPEROCTAVE+1;j++)//取中间的 scaleperoctave 个层
13.        {
14.            //在图像的有效区域内寻找具有显著性特征的局部最大值
15.            //float sigma=(GaussianPyr[i].Octave)[j].levelsigma;
16.            //int dim = (int) (max(3.0f, 2.0*GAUSSKERN *sigma + 1.0f)*0.5);
17.            int dim = (int)(0.5*((GaussianPyr[i].Octave)[j].levelsigmalength)+0.5);
18.            for (int m=dim;m<((DOGoctaves[i].row)-dim);m++)
19.                for(int n=dim;n<((DOGoctaves[i].col)-dim);n++)
20.                {
21.                    if ( fabs(ImLevels(i,j,m,n))>= CONTRAST_THRESHOLD )
22.                    {
23.
24.                        if ( ImLevels(i,j,m,n)!=0.0 ) //1、首先是非零
25.                        {
26.                            float inf_val=ImLevels(i,j,m,n);
27.                            if(( (inf_val <= ImLevels(i,j-1,m-1,n-1))&&
28.                                (inf_val <= ImLevels(i,j-1,m ,n-1))&&
29.                                (inf_val <= ImLevels(i,j-1,m+1,n-1))&&
30.                                (inf_val <= ImLevels(i,j-1,m-1,n ))&&
31.                                (inf_val <= ImLevels(i,j-1,m ,n ))&&
32.                                (inf_val <= ImLevels(i,j-1,m+1,n ))&&
33.                                (inf_val <= ImLevels(i,j-1,m-1,n+1))&&
34.                                (inf_val <= ImLevels(i,j-1,m ,n+1))&&
35.                                (inf_val <= ImLevels(i,j-1,m+1,n+1))&& //底层的小尺度 9
36.
37.                                (inf_val <= ImLevels(i,j,m-1,n-1))&&
```

```

38.         (inf_val <= ImLevels(i,j,m ,n-1))&&
39.         (inf_val <= ImLevels(i,j,m+1,n-1))&&
40.         (inf_val <= ImLevels(i,j,m-1,n ))&&
41.         (inf_val <= ImLevels(i,j,m+1,n ))&&
42.         (inf_val <= ImLevels(i,j,m-1,n+1))&&
43.         (inf_val <= ImLevels(i,j,m ,n+1))&&
44.         (inf_val <= ImLevels(i,j,m+1,n+1))&&           //当前层 8
45.
46.         (inf_val <= ImLevels(i,j+1,m-1,n-1))&&
47.         (inf_val <= ImLevels(i,j+1,m ,n-1))&&
48.         (inf_val <= ImLevels(i,j+1,m+1,n-1))&&
49.         (inf_val <= ImLevels(i,j+1,m-1,n ))&&
50.         (inf_val <= ImLevels(i,j+1,m ,n ))&&
51.         (inf_val <= ImLevels(i,j+1,m+1,n ))&&
52.         (inf_val <= ImLevels(i,j+1,m-1,n+1))&&
53.         (inf_val <= ImLevels(i,j+1,m ,n+1))&&
54.         (inf_val <= ImLevels(i,j+1,m+1,n+1))           //下一层大尺度 9
55.     ) ||
56.     ( (inf_val >= ImLevels(i,j-1,m-1,n-1))&&
57.     (inf_val >= ImLevels(i,j-1,m ,n-1))&&
58.     (inf_val >= ImLevels(i,j-1,m+1,n-1))&&
59.     (inf_val >= ImLevels(i,j-1,m-1,n ))&&
60.     (inf_val >= ImLevels(i,j-1,m ,n ))&&
61.     (inf_val >= ImLevels(i,j-1,m+1,n ))&&
62.     (inf_val >= ImLevels(i,j-1,m-1,n+1))&&
63.     (inf_val >= ImLevels(i,j-1,m ,n+1))&&
64.     (inf_val >= ImLevels(i,j-1,m+1,n+1))&&
65.
66.     (inf_val >= ImLevels(i,j,m-1,n-1))&&
67.     (inf_val >= ImLevels(i,j,m ,n-1))&&
68.     (inf_val >= ImLevels(i,j,m+1,n-1))&&
69.     (inf_val >= ImLevels(i,j,m-1,n ))&&
70.     (inf_val >= ImLevels(i,j,m+1,n ))&&
71.     (inf_val >= ImLevels(i,j,m-1,n+1))&&
72.     (inf_val >= ImLevels(i,j,m ,n+1))&&
73.     (inf_val >= ImLevels(i,j,m+1,n+1))&&
74.
75.     (inf_val >= ImLevels(i,j+1,m-1,n-1))&&
76.     (inf_val >= ImLevels(i,j+1,m ,n-1))&&
77.     (inf_val >= ImLevels(i,j+1,m+1,n-1))&&
78.     (inf_val >= ImLevels(i,j+1,m-1,n ))&&
79.     (inf_val >= ImLevels(i,j+1,m ,n ))&&
80.     (inf_val >= ImLevels(i,j+1,m+1,n ))&&
81.     (inf_val >= ImLevels(i,j+1,m-1,n+1))&&

```

```

82.         (inf_val >= ImLevels(i,j+1,m ,n+1))&&
83.         (inf_val >= ImLevels(i,j+1,m+1,n+1))
84.     ) ) //2、满足 26 个中极值点
85.     {
86.         //此处可存储
87.         //然后必须具有明显的显著性，即必须大于 CONTRAST_THRESHOLD=0.02
88.         if ( fabs(ImLevels(i,j,m,n))>= CONTRAST_THRESHOLD )
89.         {
90.             //最后显著处的特征点必须具有足够的曲率比, CURVATURE_THRESHOLD=10.0, 首先
91.             // Compute the entries of the Hessian matrix at the extrema locatio
92.             n.
93.             /*
94.             1   0   -1
95.             0   0   0
96.             -1  0   1           *0.25
97.             */
98.             // Compute the trace and the determinant of the Hessian.
99.             //Tr_H = Dxx + Dyy;
100.            //Det_H = Dxx*Dyy - Dxy^2;
101.            float Dxx,Dyy,Dxy,Tr_H,Det_H,curvature_ratio;
102.            Dxx = ImLevels(i,j,m,n-1) + ImLevels(i,j,m,n+1)-2.0*ImLevels(i,j,m
103.                ,n);
104.            Dyy = ImLevels(i,j,m-1,n) + ImLevels(i,j,m+1,n)-2.0*ImLevels(i,j,m
105.                ,n);
106.            Dxy = ImLevels(i,j,m-1,n-1) + ImLevels(i,j,m+1,n+1) - ImLevels(i,j
107.                ,m+1,n-1) - ImLevels(i,j,m-1,n+1);
108.            Tr_H = Dxx + Dyy;
109.            Det_H = Dxx*Dyy - Dxy*Dxy;
110.            // Compute the ratio of the principal curvatures.
111.            curvature_ratio = (1.0*Tr_H*Tr_H)/Det_H;
112.            if ( (Det_H>=0.0) && (curvature_ratio <= curvature_threshold) ) /
113.                //最后得到最具有显著性特征的特征点
114.                {
115.                    //将其存储起来, 以计算后面的特征描述字
116.                    keypoint_count++;
117.                    Keypoint k;
118.                    /* Allocate memory for the keypoint. */
119.                    k = (Keypoint) malloc(sizeof(struct KeypointSt));

```

```

120.         k->sx = n;      //列
121.         k->octave=i;
122.         k->level=j;
123.         k->scale = (GaussianPyr[i].Octave)[j].absolute_sigma;
124.         } //if >curvature_thresh
125.         } //if >contrast
126.         } //if inf value
127.         } //if non zero
128.         } //if >contrast
129.     } //for concrete image level col
130. } //for levels
131. } //for octaves
132. return keypoint_count;
133. }
134.
135. //在图像中，显示 SIFT 特征点的位置
136. void DisplayKeypointLocation(IplImage* image, ImageOctaves *GaussianPyr)
137. {
138.
139.     Keypoint p = keypoints; // p 指向第一个结点
140.     while(p) // 没到表尾
141.     {
142.         cvLine( image, cvPoint((int)((p->col)-3),(int)(p->row)),
143.             cvPoint((int)((p->col)+3),(int)(p->row)), CV_RGB(255,255,0),
144.             1, 8, 0 );
145.         cvLine( image, cvPoint((int)(p->col),(int)((p->row)-3)),
146.             cvPoint((int)(p->col),(int)((p->row)+3)), CV_RGB(255,255,0),
147.             1, 8, 0 );
148.         // cvCircle(image,cvPoint((uchar)(p->col),(uchar)(p->row)),
149.         // (int)((GaussianPyr[p->octave].Octave)[p->level].absolute_sigma),
150.         // CV_RGB(255,0,0),1,8,0);
151.         p=p->next;
152.     }
153. }
154.
155. // Compute the gradient direction and magnitude of the gaussian pyramid ima
ges
156. void ComputeGrad_DirecandMag(int numoctaves, ImageOctaves *GaussianPyr)
157. {
158.     // ImageOctaves *mag_thresh ;
159.     mag_pyr=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );
160.     grad_pyr=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );
161.     // float sigma=( (GaussianPyr[0].Octave)[SCALESPEROCTAVE+2].absolute_sigma
) / GaussianPyr[0].subsample;

```

```

162. // int dim = (int) (max(3.0f, 2 * GAUSSKERN *sigma + 1.0f)*0.5+0.5);
163. #define ImLevels(OCTAVE,LEVEL,ROW,COL) ((float *) (GaussianPyr[(OCTAVE)].Octave[(LEVEL)].Level->data.fl + GaussianPyr[(OCTAVE)].Octave[(LEVEL)].Level->step/sizeof(float) *(ROW)))[(COL)]
164. for (int i=0; i<numoctaves; i++)
165. {
166.     mag_pyr[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE) * sizeof(ImageLevels) );
167.     grad_pyr[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE) * sizeof(ImageLevels) );
168.     for(int j=1;j<SCALESPEROCTAVE+1;j++)//取中间的 scaleperoctave 个层
169.     {
170.         CvMat *Mag = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col
171. , CV_32FC1);
172.         CvMat *Ori = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV_32FC
173. 1);
174.         CvMat *tempMat1 = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV
175. _32FC1);
176.         CvMat *tempMat2 = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV
177. _32FC1);
178. #define MAG(ROW,COL) ((float *) (Mag->data.fl + Mag->step/sizeof(float) *(RO
179. W)))[(COL)]
180. #define ORI(ROW,COL) ((float *) (Ori->data.fl + Ori->step/sizeof(float) *(RO
181. W)))[(COL)]
182. #define TEMPMAT1(ROW,COL) ((float *) (tempMat1->data.fl + tempMat1->step/siz
183. eof(float) *(ROW)))[(COL)]
184. #define TEMPMAT2(ROW,COL) ((float *) (tempMat2->data.fl + tempMat2->step/siz
185. eof(float) *(ROW)))[(COL)]
186.     for (int m=1;m<(GaussianPyr[i].row-1);m++)
187.         for(int n=1;n<(GaussianPyr[i].col-1);n++)
188.     {
189.         //计算幅值
190.         TEMPMAT1(m,n) = 0.5*( ImLevels(i,j,m,n+1)-ImLevels(i,j,m,n-1) ); //dx
191.         TEMPMAT2(m,n) = 0.5*( ImLevels(i,j,m+1,n)-ImLevels(i,j,
192. m-1,n) ); //dy
193.         MAG(m,n) = sqrt(TEMPMAT1(m,n)*TEMPMAT1(m,n)+TEMPMAT2(m,
194. n)*TEMPMAT2(m,n)); //mag
195.         //计算方向
196.         ORI(m,n) =atan( TEMPMAT2(m,n)/TEMPMAT1(m,n) );

```

```

191.             if (ORI(m,n)==CV_PI)
192.                 ORI(m,n)=-CV_PI;
193.             }
194.             ((mag_pyr[i].Octave)[j-1]).Level=Mag;
195.             ((grad_pyr[i].Octave)[j-1]).Level=Ori;
196.             cvReleaseMat(&tempMat1);
197.             cvReleaseMat(&tempMat2);
198.         }//for levels
199.     }//for octaves
200. }
```

SIFT 算法第四步

[view plain](#)[copy to clipboard](#)[print?](#)

```

1. //SIFT 算法第四步：计算各个特征点的主方向，确定主方向
2. void AssignTheMainOrientation(int numoctaves, ImageOctaves *GaussianPyr, ImageOctaves *mag_pyr, ImageOctaves *grad_pyr)
3. {
4.     // Set up the histogram bin centers for a 36 bin histogram.
5.     int num_bins = 36;
6.     float hist_step = 2.0*PI/num_bins;
7.     float hist_orient[36];
8.     for (int i=0;i<36;i++)
9.     hist_orient[i]=-PI+i*hist_step;
10.    float sigma1=( ((GaussianPyr[0].Octave)[SCALESPEROCTAVE].absolute_sigma)
11.        ) / (GaussianPyr[0].subsample); //SCALESPEROCTAVE+2
12.    int zero_pad = (int) (max(3.0f, 2 * GAUSSKERN * sigma1 + 1.0f)*0.5+0.5);
13.    //Assign orientations to the keypoints.
14. #define ImLevels(OCTAVES,LEVELS,ROW,COL) ((float *)((GaussianPyr[(OCTAVES)].Octave[(LEVELS)].Level)->data.fl + (GaussianPyr[(OCTAVES)].Octave[(LEVELS)].Level)->step/sizeof(float) *(ROW)))[(COL)])
15.    int keypoint_count = 0;
16.    Keypoint p = keypoints; // p 指向第一个结点
17.
18.    while(p) // 没到表尾
19.    {
20.        int i=p->octave;
21.        int j=p->level;
22.        int m=p->sy;    //行
23.        int n=p->sx;    //列
24.        if ((m>zero_pad)&&(m<GaussianPyr[i].row-zero_pad)&&
25.            (n>zero_pad)&&(n<GaussianPyr[i].col-zero_pad) )
26.    {
```

```

27.     float sigma=( ((GaussianPyr[i].Octave)[j].absolute_sigma) ) / (GaussianPy
r[i].subsample);
28.     //产生二维高斯模板
29.     CvMat* mat = GaussianKernel2D( sigma );
30.     int dim=(int)(0.5 * (mat->rows));
31.     //分配用于存储 Patch 幅值和方向的空间
32. #define MAT(ROW,COL) ((float *) (mat->data.fl + mat->step/sizeof(float) *(ROW
)))[(COL)]
33.
34.     //声明方向直方图变量
35.     double* orienthist = (double *) malloc(36 * sizeof(double));
36.     for ( int sw = 0 ; sw < 36 ; ++sw)
37.     {
38.         orienthist[sw]=0.0;
39.     }
40.     //在特征点的周围统计梯度方向
41.         for ( int x=m-dim,mm=0;x<=(m+dim);x++,mm++)
42.             for(int y=n-dim,nn=0;y<=(n+dim);y++,nn++)
43.             {
44.                 //计算特征点处的幅值
45.                 double dx = 0.5*(ImLevels(i,j,x,y+1)-ImLevels(i,j,x,y-1)); //dx
46.                 double dy = 0.5*(ImLevels(i,j,x+1,y)-ImLevels(i,j,x-1,y)); //dy
47.                 double mag = sqrt(dx*dx+dy*dy); //mag
48.                 //计算方向
49.                 double Ori =atan( 1.0*dy/dx );
50.                 int binIdx = FindClosestRotationBin(36, Ori);           //得到离
现有方向最近的直方块
51.                 orienthist[binIdx] = orienthist[binIdx] + 1.0* mag * MAT(mm,nn); //利用高
斯加权累加进直方图相应的块
52.             }
53.             // Find peaks in the orientation histogram using nonmax suppression.
54.             AverageWeakBins (orienthist, 36);
55.             // find the maximum peak in gradient orientation
56.             double maxGrad = 0.0;
57.             int maxBin = 0;
58.             for (int b = 0 ; b < 36 ; ++b)
59.             {
60.                 if (orienthist[b] > maxGrad)
61.                 {
62.                     maxGrad = orienthist[b];
63.                     maxBin = b;
64.                 }
65.             }
66.             // First determine the real interpolated peak high at the maximum bin

```

```

67.     // position, which is guaranteed to be an absolute peak.
68.     double maxPeakValue=0.0;
69.     double maxDegreeCorrection=0.0;
70.     if ( InterpolateOrientation ( orienthist[maxBin == 0 ? (36 - 1) : (maxB
    in - 1)],
71.                                         orienthist[maxBin], orienthist[(maxBin + 1) % 36],
72.                                         &maxDegreeCorrection, &maxPeakValue)) == false)
73.     printf("BUG: Parabola fitting broken");
74.
75.     // Now that we know the maximum peak value, we can find other keypoint
76.     // orientations, which have to fulfill two criterias:
77.     //
78.     // 1. They must be a local peak themselves. Else we might add a very
79.     //     similar keypoint orientation twice (imagine for example the
80.     //     values: 0.4 1.0 0.8, if 1.0 is maximum peak, 0.8 is still added
81.     //     with the default threshold, but the maximum peak orientation
82.     //     was already added).
83.     // 2. They must have at least peakRelThresh times the maximum peak
84.     //     value.
85.     bool binIsKeypoint[36];
86.     for ( b = 0 ; b < 36 ; ++b)
87.     {
88.         binIsKeypoint[b] = false;
89.         // The maximum peak of course is
90.         if (b == maxBin)
91.         {
92.             binIsKeypoint[b] = true;
93.             continue;
94.         }
95.         // Local peaks are, too, in case they fulfill the threshold
96.         if (orienthist[b] < (peakRelThresh * maxPeakValue))
97.             continue;
98.         int leftI = (b == 0) ? (36 - 1) : (b - 1);
99.         int rightI = (b + 1) % 36;
100.        if (orienthist[b] <= orienthist[leftI] || orienthist[b] <= orienthist[
    rightI])
101.            continue; // no local peak
102.        binIsKeypoint[b] = true;
103.    }
104.    // find other possible locations
105.    double oneBinRad = (2.0 * PI) / 36;
106.    for ( b = 0 ; b < 36 ; ++b)
107.    {
108.        if (binIsKeypoint[b] == false)

```

```

109.     continue;
110.     int bLeft = (b == 0) ? (36 - 1) : (b - 1);
111.     int bRight = (b + 1) % 36;
112.     // Get an interpolated peak direction and value guess.
113.     double peakValue;
114.     double degreeCorrection;
115.
116.     double maxPeakValue, maxDegreeCorrection;
117.     if (InterpolateOrientation ( orienthist[maxBin == 0 ? (36 - 1) : (maxB
118.         in - 1)],
119.             orienthist[maxBin], orienthist[(maxBin + 1) % 36],
120.             &degreeCorrection, &peakValue) == false)
121.     {
122.         printf("BUG: Parabola fitting broken");
123.     }
124.     double degree = (b + degreeCorrection) * oneBinRad - PI;
125.     if (degree < -PI)
126.         degree += 2.0 * PI;
127.     else if (degree > PI)
128.         degree -= 2.0 * PI;
129.     //存储方向，可以直接利用检测到的链表进行该步主方向的指定;
130.     //分配内存重新存储特征点
131.     Keypoint k;
132.     /* Allocate memory for the keypoint Descriptor. */
133.     k = (Keypoint) malloc(sizeof(struct KeypointSt));
134.     k->next = keyDescriptors;
135.     keyDescriptors = k;
136.     k->descrip = (float*)malloc(LEN * sizeof(float));
137.     k->row = p->row;
138.     k->col = p->col;
139.     k->sy = p->sy;      //行
140.     k->sx = p->sx;      //列
141.     k->octave = p->octave;
142.     k->level = p->level;
143.     k->scale = p->scale;
144.     k->ori = degree;
145.     k->mag = peakValue;
146.     }//for
147.     free(orienthist);
148. }
149. p=p->next;
150. }
151. }

```

```

152.
153. // 寻找与方向直方图最近的柱，确定其 index
154. int FindClosestRotationBin (int binCount, float angle)
155. {
156.     angle += CV_PI;
157.     angle /= 2.0 * CV_PI;
158.     // calculate the aligned bin
159.     angle *= binCount;
160.     int idx = (int) angle;
161.     if (idx == binCount)
162.         idx = 0;
163.     return (idx);
164. }
165.
166. // Average the content of the direction bins.
167. void AverageWeakBins (double* hist, int binCount)
168. {
169.     // TODO: make some tests what number of passes is the best. (its clear
170.     // one is not enough, as we may have something like
171.     // ( 0.4, 0.4, 0.3, 0.4, 0.4 ))
172.     for (int sn = 0 ; sn < 2 ; ++sn)
173.     {
174.         double firstE = hist[0];
175.         double last = hist[binCount-1];
176.         for (int sw = 0 ; sw < binCount ; ++sw)
177.         {
178.             double cur = hist[sw];
179.             double next = (sw == (binCount - 1)) ? firstE : hist[(sw + 1) % binCount
180. ];
181.             hist[sw] = (last + cur + next) / 3.0;
182.             last = cur;
183.         }
184.     }
185.
186. // Fit a parabol to the three points (-1.0 ; left), (0.0 ; middle) and
187. // (1.0 ; right).
188. // Formulas:
189. // f(x) = a (x - c)^2 + b
190. // c is the peak offset (where f'(x) is zero), b is the peak value.
191. // In case there is an error false is returned, otherwise a correction
192. // value between [-1 ; 1] is returned in 'degreeCorrection', where -1
193. // means the peak is located completely at the left vector, and -0.5 just
194. // in the middle between left and middle and > 0 to the right side. In

```

```

195. // 'peakValue' the maximum estimated peak value is stored.
196. bool InterpolateOrientation (double left, double middle,double right, doubl
   e *degreeCorrection, double *peakValue)
197. {
198.   double a = ((left + right) - 2.0 * middle) / 2.0;    //抛物线拟合系数 a
199.   // degreeCorrection = peakValue = Double.NaN;
200.
201.   // Not a parabol
202.   if (a == 0.0)
203.     return false;
204.   double c = (((left - middle) / a) - 1.0) / 2.0;
205.   double b = middle - c * c * a;
206.   if (c < -0.5 || c > 0.5)
207.     return false;
208.   *degreeCorrection = c;
209.   *peakValue = b;
210.   return true;
211. }
212.
213. //显示特征点处的主方向
214. void DisplayOrientation (IplImage* image, ImageOctaves *GaussianPyr)
215. {
216.   Keypoint p = keyDescriptors; // p 指向第一个结点
217.   while(p) // 没到表尾
218.   {
219.     float scale=(GaussianPyr[p->octave].Octave)[p->level].absolute_sigma;
220.     float autoscale = 3.0;
221.     float uu=autoscale*scale*cos(p->ori);
222.     float vv=autoscale*scale*sin(p->ori);
223.     float x=(p->col)+uu;
224.     float y=(p->row)+vv;
225.     cvLine( image, cvPoint((int)(p->col),(int)(p->row)),
226.             cvPoint((int)x,(int)y), CV_RGB(255,255,0),
227.             1, 8, 0 );
228.     // Arrow head parameters
229.     float alpha = 0.33; // Size of arrow head relative to the length of
   the vector
230.     float beta = 0.33; // Width of the base of the arrow head relative
   to the length
231.
232.     float xx0= (p->col)+uu-alpha*(uu+beta*vv);
233.     float yy0= (p->row)+vv-alpha*(vv-beta*uu);
234.     float xx1= (p->col)+uu-alpha*(uu-beta*vv);
235.     float yy1= (p->row)+vv-alpha*(vv+beta*uu);

```

```

236.         cvLine( image, cvPoint((int)xx0,(int)yy0),
237.             cvPoint((int)x,(int)y), CV_RGB(255,255,0),
238.             1, 8, 0 );
239.         cvLine( image, cvPoint((int)xx1,(int)yy1),
240.             cvPoint((int)x,(int)y), CV_RGB(255,255,0),
241.             1, 8, 0 );
242.         p=p->next;
243.     }
244. }
```

SIFT 算法第五步

SIFT 算法第五步：抽取各个特征点处的特征描述字，确定特征点的描述字。描述字是 Patch 网格内梯度方向的描述，旋转网格到主方向，插值得到网格处梯度值。

一个特征点可以用 $2*2*8=32$ 维的向量，也可以用 $4*4*8=128$ 维的向量更精确的进行描述。

[view plain](#)[copy to clipboard](#)[print?](#)

```

1. void ExtractFeatureDescriptors(int numoctaves, ImageOctaves *GaussianPyr)
2. {
3.     // The orientation histograms have 8 bins
4.     float orient_bin_spacing = PI/4;
5.     float orient_angles[8]={-PI,-PI+orient_bin_spacing,-PI*0.5, -orient_bin_
spacing,
6.     0.0, orient_bin_spacing, PI*0.5, PI+orient_bin_spacing};
7.     //产生描述字中心各点坐标
8.     float *feat_grid=(float *) malloc( 2*16 * sizeof(float));
9.     for (int i=0;i<GridSpacing;i++)
10.    {
11.        for (int j=0;j<2*GridSpacing;++j,++j)
12.        {
13.            feat_grid[i*2*GridSpacing+j]=-6.0+i*GridSpacing;
14.            feat_grid[i*2*GridSpacing+j+1]=-6.0+0.5*j*GridSpacing;
15.        }
16.    }
17.    //产生网格
18.    float *feat_samples=(float *) malloc( 2*256 * sizeof(float));
19.    for ( i=0;i<4*GridSpacing;i++)
20.    {
21.        for (int j=0;j<8*GridSpacing;j+=2)
22.        {
23.            feat_samples[i*8*GridSpacing+j]=-(2*GridSpacing-0.5)+i;
24.            feat_samples[i*8*GridSpacing+j+1]=-(2*GridSpacing-0.5)+0.5*j;
25.        }
26.    }
```

```

27. float feat_window = 2*GridSpacing;
28. Keypoint p = keyDescriptors; // p 指向第一个结点
29. while(p) // 没到表尾
30. {
31.     float scale=(GaussianPyr[p->octave].Octave)[p->level].absolute_sigma;
32.
33.     float sine = sin(p->ori);
34.     float cosine = cos(p->ori);
35.         //计算中心点坐标旋转之后的位置
36.     float *featcenter=(float *) malloc( 2*16 * sizeof(float));
37.     for (int i=0;i<GridSpacing;i++)
38.     {
39.         for (int j=0;j<2*GridSpacing;j+=2)
40.         {
41.             float x=feat_grid[i*2*GridSpacing+j];
42.             float y=feat_grid[i*2*GridSpacing+j+1];
43.             featcenter[i*2*GridSpacing+j]=((cosine * x + sine * y) + p->sx);
44.             featcenter[i*2*GridSpacing+j+1]=((-sine * x + cosine * y) + p->sy);
45.         }
46.     }
47.     // calculate sample window coordinates (rotated along keypoint)
48.     float *feat=(float *) malloc( 2*256 * sizeof(float));
49.     for ( i=0;i<64*GridSpacing;i++,i++)
50.     {
51.         float x=feat_samples[i];
52.         float y=feat_samples[i+1];
53.         feat[i]=((cosine * x + sine * y) + p->sx);
54.         feat[i+1]=((-sine * x + cosine * y) + p->sy);
55.     }
56.     //Initialize the feature descriptor.
57.     float *feat_desc = (float *) malloc( 128 * sizeof(float));
58.     for (i=0;i<128;i++)
59.     {
60.         feat_desc[i]=0.0;
61.         // printf("%f ",feat_desc[i]);
62.     }
63.         //printf("\n");
64.     for ( i=0;i<512;++i,++i)
65.     {
66.         float x_sample = feat[i];
67.         float y_sample = feat[i+1];
68.             // Interpolate the gradient at the sample position
69.             /*
70.              0   1   0

```

```

71.      1 * 1
72.      0 1 0 具体插值策略如图示
73.      */
74.      float sample12=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Lev
el, x_sample, y_sample-1);
75.      float sample21=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Lev
el, x_sample-1, y_sample);
76.      float sample22=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Lev
el, x_sample, y_sample);
77.      float sample23=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Lev
el, x_sample+1, y_sample);
78.      float sample32=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Lev
el, x_sample, y_sample+1);
79.      //float diff_x = 0.5*(sample23 - sample21);
80.      //float diff_y = 0.5*(sample32 - sample12);
81.      float diff_x = sample23 - sample21;
82.      float diff_y = sample32 - sample12;
83.      float mag_sample = sqrt( diff_x*diff_x + diff_y*diff_y );
84.      float grad_sample = atan( diff_y / diff_x );
85.      if(grad_sample == CV_PI)
86.          grad_sample = -CV_PI;
87.      // Compute the weighting for the x and y dimensions.
88.      float *x_wght=(float *) malloc( GridSpacing * GridSpacing * sizeof(float));
89.      float *y_wght=(float *) malloc( GridSpacing * GridSpacing * sizeof(float));
90.      float *pos_wght=(float *) malloc( 8*GridSpacing * GridSpacing * sizeof(float));
91.      for (int m=0;m<32;++m,++m)
92.      {
93.          float x=featcenter[m];
94.          float y=featcenter[m+1];
95.          x_wght[m/2] = max(1 - (fabs(x - x_sample)*1.0/GridSpacing), 0);
96.          y_wght[m/2] = max(1 - (fabs(y - y_sample)*1.0/GridSpacing), 0);
97.      }
98.      }
99.      for ( m=0;m<16;++)
100.          for (int n=0;n<8;++)
101.              pos_wght[m*8+n]=x_wght[m]*y_wght[m];
102.      free(x_wght);
103.      free(y_wght);
104.      //计算方向的加权, 首先旋转梯度场到主方向, 然后计算差异
105.      float diff[8],orient_wght[128];

```

```

106.     for ( m=0;m<8;++m)
107.     {
108.         float angle = grad_sample-(p->ori)-orient_angles[m]+CV_PI;
109.         float temp = angle / (2.0 * CV_PI);
110.         angle -= (int)(temp) * (2.0 * CV_PI);
111.         diff[m]= angle - CV_PI;
112.     }
113.     // Compute the gaussian weighting.
114.     float x=p->sx;
115.     float y=p->sy;
116.     float g = exp(-((x_sample-x)*(x_sample-x)+(y_sample-y)*(y_sample-y))/(2
117.                   *feat_window*feat_window))/(2*CV_PI*feat_window*feat_window);
118.
119.     for ( m=0;m<128;++m)
120.     {
121.         orient_wght[m] = max((1.0 - 1.0*fabs(diff[m%8])/orient_bin_spacing),0)
122. ;
123.         feat_desc[m] = feat_desc[m] + orient_wght[m]*pos_wght[m]*g*mag_sample;
124.     }
125.     free(pos_wght);
126.     free(feat);
127.     free(featcenter);
128.     float norm=GetVecNorm( feat_desc, 128);
129.     {
130.         feat_desc[m]/=norm;
131.         if (feat_desc[m]>0.2)
132.             feat_desc[m]=0.2;
133.     }
134.     norm=GetVecNorm( feat_desc, 128);
135.     for ( m=0;m<128;m++)
136.     {
137.         feat_desc[m]/=norm;
138.         printf("%f ",feat_desc[m]);
139.     }
140.     printf("\n");
141.     p->descrip = feat_desc;
142.     p=p->next;
143. }
144. free(feat_grid);
145. free(feat_samples);
146. }

```

```

147.
148. //为了显示图象金字塔, 而作的图像水平拼接
149. CvMat* MosaicHorizen( CvMat* im1, CvMat* im2 )
150. {
151.   int row,col;
152.   CvMat *mosaic = cvCreateMat( max(im1->rows,im2->rows),(im1->cols+im2->cols
153. ),CV_32FC1);
154. #define Mosaic(ROW,COL) ((float*)(mosaic->data.fl + mosaic->step/sizeof(float)
155.   at)*(ROW))[(COL)]
156. #define Im11Mat(ROW,COL) ((float *) (im1->data.fl + im1->step/sizeof(float)
157.   *(ROW)))[(COL)]
158. #define Im22Mat(ROW,COL) ((float *) (im2->data.fl + im2->step/sizeof(float)
159.   *(ROW)))[(COL)]
160. cvZero(mosaic);
161. /* Copy images into mosaic1. */
162. for ( row = 0; row < im1->rows; row++)
163.   for ( col = 0; col < im1->cols; col++)
164.     Mosaic(row,col)=Im11Mat(row,col) ;
165.   for ( row = 0; row < im2->rows; row++)
166.     for ( col = 0; col < im2->cols; col++)
167.       Mosaic(row, (col+im1->cols) )= Im22Mat(row,col) ;
168.   return mosaic;
169.
170. }
171. //为了显示图象金字塔, 而作的图像垂直拼接
172. CvMat* MosaicVertical( CvMat* im1, CvMat* im2 )
173. {
174.   int row,col;
175.   CvMat *mosaic = cvCreateMat(im1->rows+im2->rows,max(im1->cols,im2->cols),
176.   CV_32FC1);
177. #define Mosaic(ROW,COL) ((float*)(mosaic->data.fl + mosaic->step/sizeof(float)
178.   at)*(ROW))[(COL)]
179. #define Im11Mat(ROW,COL) ((float *) (im1->data.fl + im1->step/sizeof(float)
180.   *(ROW)))[(COL)]
181. #define Im22Mat(ROW,COL) ((float *) (im2->data.fl + im2->step/sizeof(float)
182.   *(ROW)))[(COL)]
183. cvZero(mosaic);
184. /* Copy images into mosaic1. */
185. for ( row = 0; row < im1->rows; row++)
186.   for ( col = 0; col < im1->cols; col++)
187.     Mosaic(row,col)= Im11Mat(row,col) ;
188.   for ( row = 0; row < im2->rows; row++)
189.     for ( col = 0; col < im2->cols; col++)
190.       Mosaic(row, (row+im1->rows) )= Im22Mat(row,col) ;
191.   return mosaic;
192.

```

```
183.     Mosaic((row+im1->rows),col)=Im22Mat(row,col) ;
184.
185.     return mosaic;
186. }
```

ok, 为了版述清晰, 再贴一下上文所述的主函数 (注, 上文已贴出, 此是为了版述清晰, 重复造轮) :

[view plain](#)[copy to clipboard](#)[print](#)

```
1. int main( void )
2. {
3.     //声明当前帧 IplImage 指针
4.     IplImage* src = NULL;
5.     IplImage* image1 = NULL;
6.     IplImage* grey_im1 = NULL;
7.     IplImage* DoubleSizeImage = NULL;
8.
9.     IplImage* mosaic1 = NULL;
10.    IplImage* mosaic2 = NULL;
11.
12.    CvMat* mosaicHorizen1 = NULL;
13.    CvMat* mosaicHorizen2 = NULL;
14.    CvMat* mosaicVertical1 = NULL;
15.
16.    CvMat* image1Mat = NULL;
17.    CvMat* tempMat=NULL;
18.
19.    ImageOctaves *Gaussianpyr;
20.    int rows,cols;
21.
22. #define Im1Mat(ROW,COL) ((float *) (image1Mat->data.fl + image1Mat->step/size
   of(float) *(ROW)))[(COL)]
23.
24. //灰度图象像素的数据结构
25. #define Im1B(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3]
26. #define Im1G(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3+1]
27. #define Im1R(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW))
   )[(COL)*3+2]
28.
29. storage = cvCreateMemStorage(0);
```

```

30.
31. //读取图片
32. if( (src = cvLoadImage( "street1.jpg", 1)) == 0 ) // test1.jpg einstein.jpg
   m back1.bmp
33. return -1;
34.
35. //为图像分配内存
36. image1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U,3);
37. grey_im1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U,1);

38. DoubleSizeImage = cvCreateImage(cvSize(2*(src->width), 2*(src->height)), IPL_DEPTH_8U,3);
39.
40. //为图像阵列分配内存, 假设两幅图像的大小相同, tempMat 跟随 image1 的大小
41. image1Mat = cvCreateMat(src-> height, src-> width, CV_32FC1);
42. //转化成单通道图像再处理
43. cvCvtColor(src, grey_im1, CV_BGR2GRAY);
44. //转换进入 Mat 数据结构, 图像操作使用的是浮点型操作
45. cvConvert(grey_im1, image1Mat);
46.
47. double t = (double)cvGetTickCount();
48. //图像归一化
49. cvConvertScale( image1Mat, image1Mat, 1.0/255, 0 );
50.
51. int dim = min(image1Mat->rows, image1Mat->cols);
52. numoctaves = (int) (log((double) dim) / log(2.0)) - 2; //金字塔阶数
53. numoctaves = min(numoctaves, MAXOCTAVES);
54.
55. //SIFT 算法第一步, 预滤波除噪声, 建立金字塔底层
56. tempMat = ScaleInitImage(image1Mat) ;
57. //SIFT 算法第二步, 建立 Gaussian 金字塔和 DOG 金字塔
58. Gaussianpyr = BuildGaussianOctaves(tempMat) ;
59.
60. t = (double)cvGetTickCount() - t;
61. printf( "the time of build Gaussian pyramid and DOG pyramid is %.1f\n", t/
   cvGetTickFrequency()*1000. );
62.
63. #define ImLevels(OCTAVE,LEVEL,ROW,COL) ((float *) (Gaussianpyr[(OCTAVE)].Octa
   ve[(LEVEL)].Level->data.fl + Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level->st
   ep/sizeof(float) *(ROW)))[(COL)]
64. //显示高斯金字塔
65. for (int i=0; i<numoctaves;i++)
66. {
67.   if (i==0)

```

```

68.  {
69.      mosaicHorizen1=MosaicHorizen( (Gaussianpyr[0].Octave)[0].Level, (Gaussian
    pyr[0].Octave)[1].Level );
70.      for (int j=2;j<SCALESPEROCTAVE+3;j++)
71.          mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[0].Octave)[j]
    .Level );
72.      for ( j=0;j<NUMSIZE;j++)
73.          mosaicHorizen1=halfSizeImage(mosaicHorizen1);
74.  }
75. else if (i==1)
76.  {
77.      mosaicHorizen2=MosaicHorizen( (Gaussianpyr[1].Octave)[0].Level, (Gaussian
    pyr[1].Octave)[1].Level );
78.      for (int j=2;j<SCALESPEROCTAVE+3;j++)
79.          mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (Gaussianpyr[1].Octave)[j]
    .Level );
80.      for ( j=0;j<NUMSIZE;j++)
81.          mosaicHorizen2=halfSizeImage(mosaicHorizen2);
82.      mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );
83.  }
84. else
85.  {
86.      mosaicHorizen1=MosaicHorizen( (Gaussianpyr[i].Octave)[0].Level, (Gaussian
    pyr[i].Octave)[1].Level );
87.      for (int j=2;j<SCALESPEROCTAVE+3;j++)
88.          mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[i].Octave)[j]
    .Level );
89.      for ( j=0;j<NUMSIZE;j++)
90.          mosaicHorizen1=halfSizeImage(mosaicHorizen1);
91.      mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );
92.  }
93.  }
94. mosaic1 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1->hei
    ght), IPL_DEPTH_8U,1);
95. cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0, 0 );
96. cvConvertScaleAbs( mosaicVertical1, mosaic1, 1, 0 );
97.
98. // cvSaveImage("GaussianPyramid of me.jpg",mosaic1);
99. cvNamedWindow("mosaic1",1);
100. cvShowImage("mosaic1", mosaic1);
101. cvWaitKey(0);
102. cvDestroyWindow("mosaic1");
103. //显示DOG金字塔
104. for ( i=0; i<numoctaves;i++)

```

```

105.  {
106.    if (i==0)
107.    {
108.      mosaicHorizen1=MosaicHorizen( (DOOctaves[0].Octave)[0].Level, (DOOctav
es[0].Octave)[1].Level );
109.      for (int j=2;j<SCALESPEROCTAVE+2;j++)
110.        mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOOctaves[0].Octave)[j]
.Level );
111.      for ( j=0;j<NUMSIZE;j++)
112.        mosaicHorizen1=halfSizeImage(mosaicHorizen1);
113.    }
114.    else if (i==1)
115.    {
116.      mosaicHorizen2=MosaicHorizen( (DOOctaves[1].Octave)[0].Level, (DOOctav
es[1].Octave)[1].Level );
117.      for (int j=2;j<SCALESPEROCTAVE+2;j++)
118.        mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (DOOctaves[1].Octave)[j]
.Level );
119.      for ( j=0;j<NUMSIZE;j++)
120.        mosaicHorizen2=halfSizeImage(mosaicHorizen2);
121.      mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );
122.    }
123.    else
124.    {
125.      mosaicHorizen1=MosaicHorizen( (DOOctaves[i].Octave)[0].Level, (DOOctav
es[i].Octave)[1].Level );
126.      for (int j=2;j<SCALESPEROCTAVE+2;j++)
127.        mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOOctaves[i].Octave)[j]
.Level );
128.      for ( j=0;j<NUMSIZE;j++)
129.        mosaicHorizen1=halfSizeImage(mosaicHorizen1);
130.      mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );
131.    }
132.  }
133. //考虑到 DOG 金字塔各层图像都会有正负, 所以, 必须寻找最负的, 以将所有图像抬高一个台
阶去显示
134. double min_val=0;
135. double max_val=0;
136. cvMinMaxLoc( mosaicVertical1, &min_val, &max_val,NULL, NULL, NULL );
137. if ( min_val<0.0 )
138.   cvAddS( mosaicVertical1, cvScalarAll( (-1.0)*min_val ), mosaicVertical1,
NULL );
139. mosaic2 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1->he
ight), IPL_DEPTH_8U,1);

```

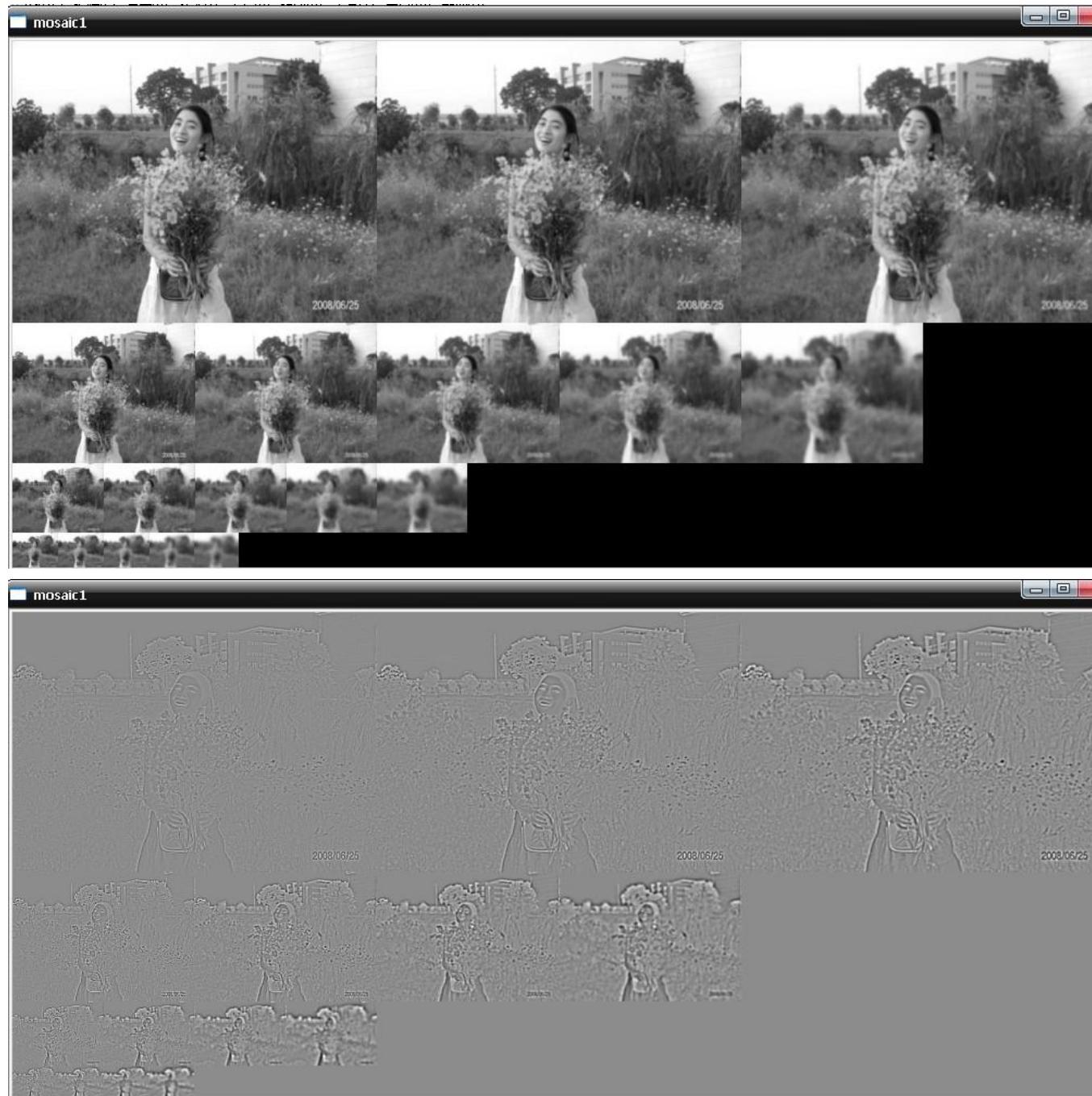
```

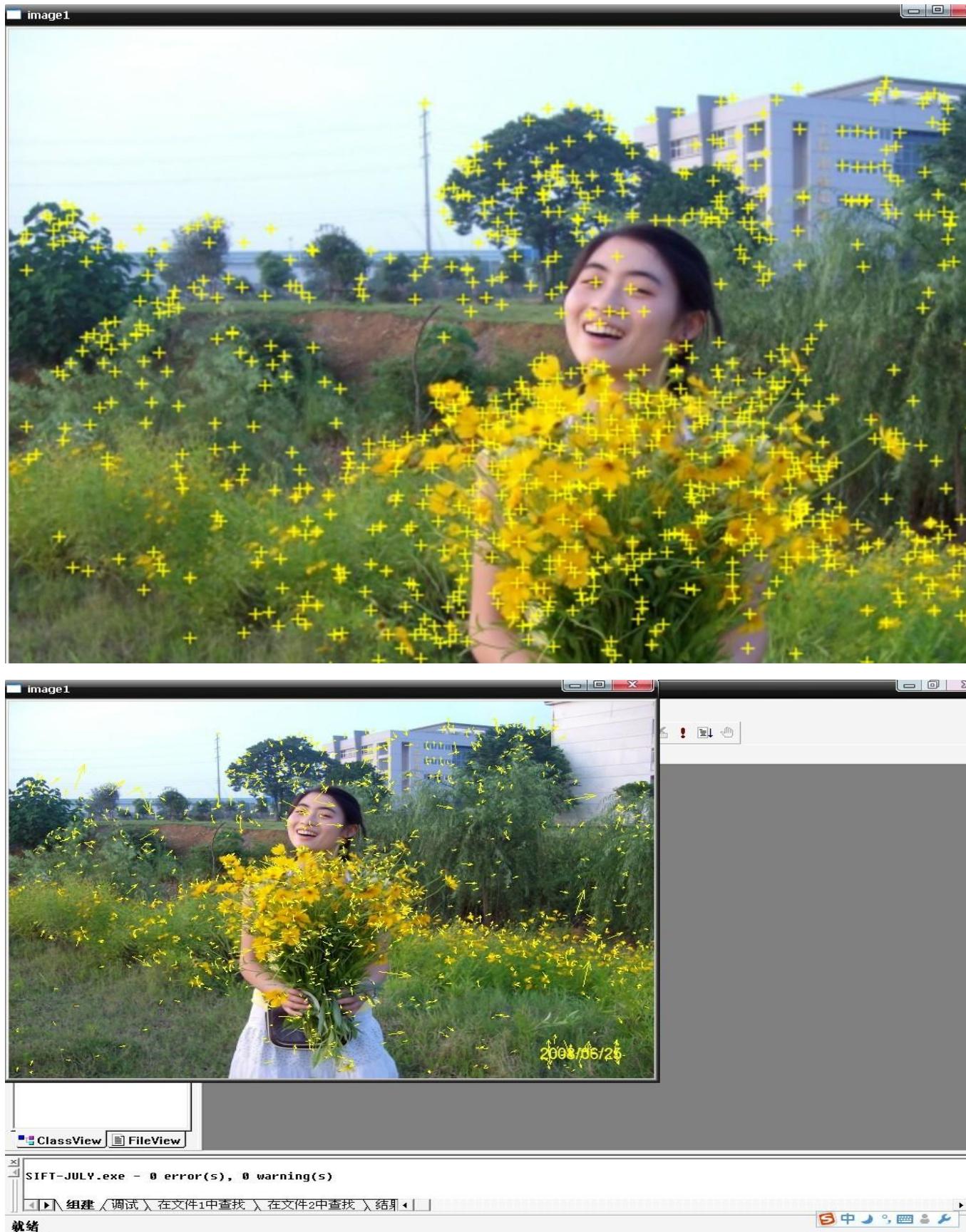
140. cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0/(max_val-min_val),
    0 );
141. cvConvertScaleAbs( mosaicVertical1, mosaic2, 1, 0 );
142.
143. // cvSaveImage("DOGPyramid of me.jpg",mosaic2);
144. cvNamedWindow("mosaic1",1);
145. cvShowImage("mosaic1", mosaic2);
146. cvWaitKey(0);
147.
148. //SIFT 算法第三步：特征点位置检测，最后确定特征点的位置
149. int keycount=DetectKeypoint(numoctaves, Gaussianpyr);
150. printf("the keypoints number are %d ;\n", keycount);
151. cvCopy(src,image1,NULL);
152. DisplayKeypointLocation( image1 ,Gaussianpyr);
153.
154. cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
155. cvNamedWindow("image1",1);
156. cvShowImage("image1", DoubleSizeImage);
157. cvWaitKey(0);
158. cvDestroyWindow("image1");
159.
160. //SIFT 算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征点的主方向
161. ComputeGrad_DirecandMag(numoctaves, Gaussianpyr);
162. AssignTheMainOrientation( numoctaves, Gaussianpyr,mag_pyr,grad_pyr);
163. cvCopy(src,image1,NULL);
164. DisplayOrientation ( image1, Gaussianpyr);
165.
166. // cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
167. cvNamedWindow("image1",1);
168. // cvResizeWindow("image1", 2*(image1->width), 2*(image1->height) );
169. cvShowImage("image1", image1);
170. cvWaitKey(0);
171.
172. //SIFT 算法第五步：抽取各个特征点处的特征描述字
173. ExtractFeatureDescriptors( numoctaves, Gaussianpyr);
174. cvWaitKey(0);
175.
176. //销毁窗口
177. cvDestroyWindow("image1");
178. cvDestroyWindow("mosaic1");
179. //释放图像
180. cvReleaseImage(&image1);
181. cvReleaseImage(&grey_im1);
182. cvReleaseImage(&mosaic1);

```

```
183. cvReleaseImage(&mosaic2);
184. return 0;
185. }
```

最后，再看一下，运行效果（图中美女为老乡 + 朋友，何姐 08 年照）：





完。

版权声明：

- 1、本文版权归本人和 CSDN 共同拥有。转载，请注明出处及作者本人。**
2、版权侵犯者，无论任何人，任何网站，1、永久追踪，2、永久谴责，3、永久追究法律责任的权利****

July、二零一一年三月十二日唐明。

九（三续）：SIFT 算法的应用—目标识别之 Bag-of-words 模型

作者: wawayu, July。编程艺术室出品。

出处：http://blog.csdn.net/v_JULY_v

- 引言

本 blog 之前已经写了四篇关于 SIFT 的文章, 请参考[九、图像特征提取与匹配之 SIFT 算法](#), [九\(续\)、sift 算法的编译与实现](#), [九\(再续\)、教你一步一步用 c 语言实现 sift 算法、上](#), 及[九\(再续\)、教你一步一步用 c 语言实现 sift 算法、下](#)。

上述这 4 篇文章对 SIFT 算法的原理和 C 语言实现都做了详细介绍, 用 SIFT 做图像匹配效果不错。现在考虑更为高层的应用, 将 SIFT 算法应用于目标识别: 发现图像中包含的物体类别, 这是计算机视觉领域最基本也是最重要的任务之一。

且原经典算法研究系列可能将改名为[算法珠玑—经典算法的通俗演义](#)。改名考虑到三点: 1、不求面面俱到所有算法, 所以掏炼, 谓之“珠玑”; 2、突出本博客内算法内容的特色—通俗易懂、简明直白, 谓之“通俗”; 3、侧重经典算法的研究与实现, 以及实际应用, 谓之“演义”。

OK, 闲话少说, 上一篇我们介绍了[六\(续\)、从 KMP 算法一步一步谈到 BM 算法](#)。下面我们来介绍有关 SIFT 算法的目标识别的应用—Bag-of-words 模型。

• **Bag-of-words 模型简介**

Bag-of-words 模型是信息检索领域常用的文档表示方法。在信息检索中, BOW 模型假定对于一个文档, 忽略它的单词顺序和语法、句法等要素, 将其仅仅看作是若干个词汇的集合, 文档中每个单词的出现都是独立的, 不依赖于其它单词是否出现。也就是说, 文档中任意一个位置出现的任何单词, 都不受该文档语意影响而独立选择的。例如有如下两个文档:

1: Bob likes to play basketball, Jim likes too.
2: Bob also likes to play football games.

基于这两个文本文档, 构造一个词典:

Dictionary = {1: "Bob", 2: "like", 3: "to", 4: "play", 5: "basketball", 6: "also", 7: "football", 8: "games", 9: "Jim", 10: "too"}。

这个词典一共包含 10 个不同的单词, 利用词典的索引号, 上面两个文档每一个都可以用一个 10 维向量表示 (用整数数字 $0 \sim n$ (n 为正整数) 表示某个单词在文档中出现的次数) :

1: [1, 2, 1, 1, 1, 0, 0, 0, 1, 1]

2: [1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

向量中每个元素表示词典中相关元素在文档中出现的次数(下文中，将用单词的直方图表示)。不过，在构造文档向量的过程中可以看到，我们并没有表达单词在原来句子中出现的次序(这是本 Bag-of-words 模型的缺点之一，不过瑕不掩瑜甚至在此处无关紧要)。

- **Bag-of-words 模型的应用**

Bag-of-words 模型的适用场合

现在想象在一个巨大的文档集合 D，里面一共有 M 个文档，而文档里面的所有单词提取出来后，一起构成一个包含 N 个单词的词典，利用 Bag-of-words 模型，每个文档都可以被表示成为一个 N 维向量，计算机非常擅长于处理数值向量。这样，就可以利用计算机来完成海量文档的分类过程。

考虑将 Bag-of-words 模型应用于图像表示。为了表示一幅图像，我们可以将图像看作文档，即若干个“视觉词汇”的集合，同样的，视觉词汇相互之间没有顺序。

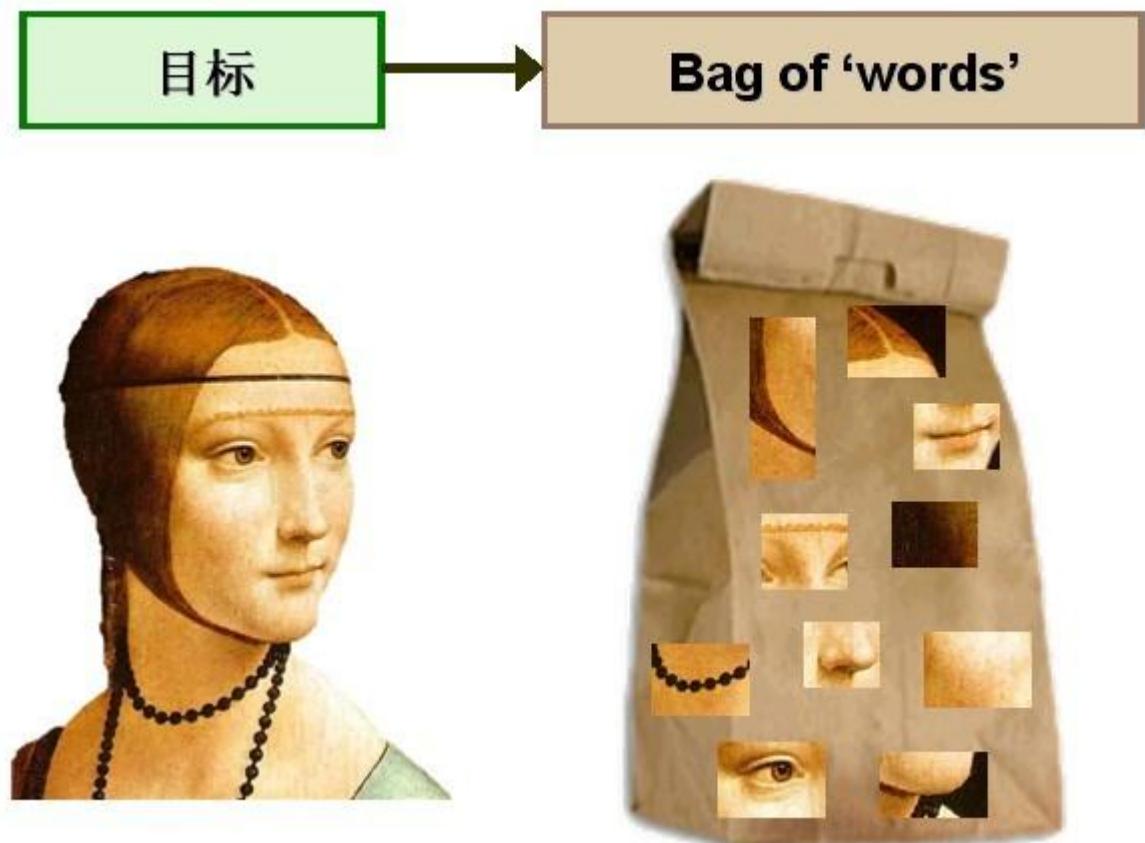


图 1 将 Bag-of-words 模型应用于图像表示

由于图像中的词汇不像文本文档中的那样是现成的，我们需要首先从图像中提取出相互独立的视觉词汇，这通常需要经过三个步骤：（1）特征检测，（2）特征表示，（3）单词本的生成，，请看下图 2：



图 2 从图像中提取出相互独立的视觉词汇

通过观察会发现，同一类目标的不同实例之间虽然存在差异，但我们仍然可以找到它们之间的一些共同的地方，比如说人脸，虽然说不同人的脸差别比较大，但眼睛，嘴，鼻子等一些比较细小的部位，却观察不到太大差别，我们可以把这些不同实例之间共同的部位提取出来，作为识别这一类目标的视觉词汇。

而 SIFT 算法是提取图像中局部不变特征的应用最广泛的算法，因此我们可以用 SIFT 算法从图像中提取不变特征点，作为视觉词汇，并构造单词表，用单词表中的单词表示一幅图像。

Bag-of-words 模型应用三步

接下来，我们通过上述图像展示如何通过 Bag-of-words 模型，将图像表示成数值向量。现在有三个目标类，分别是人脸、自行车和吉他。

Bag-of-words 模型的第一步是利用 SIFT 算法，从每类图像中提取视觉词汇，将所有的视觉词汇集合在一起，如下图 3 所示：



图 3 从每类图像中提取视觉词汇

第二步是利用 K-Means 算法构造单词表。K-Means 算法是一种基于样本间相似性度量的间接聚类方法，此算法以 K 为参数，把 N 个对象分为 K 个簇，以使簇内具有较高的相似度，而簇间相似度较低。SIFT 提取的视觉词汇向量之间根据距离的远近，可以利用 K-Means 算法将词义相近的词汇合并，作为单词表中的基础词汇，假定我们将 K 设为 4，那么单词表的构 造 过 程 如 下 图 4 所 示 ：

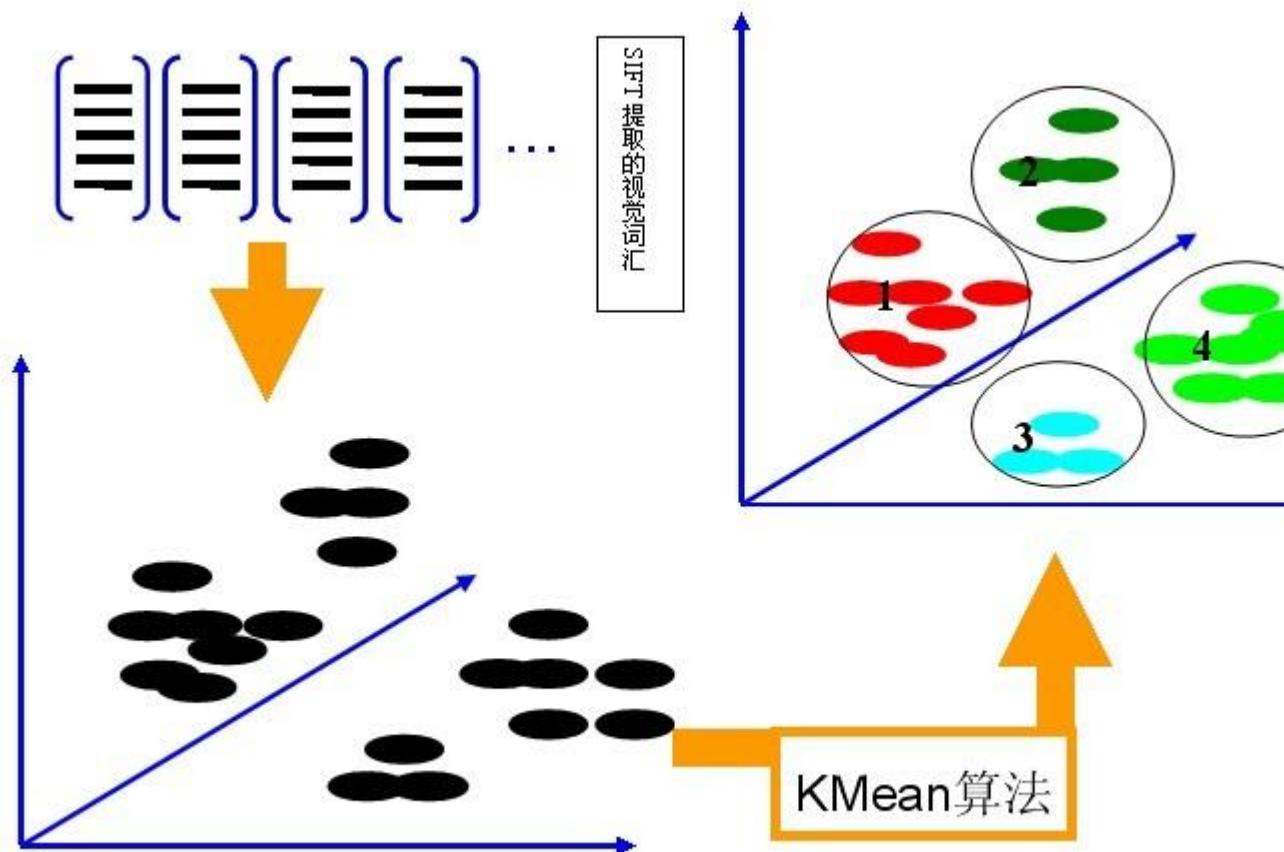


图 4 利用 K-Means 算法构造单词表

第三步是利用单词表的中词汇表示图像。利用 SIFT 算法，可以从每幅图像中提取很多个特征点，这些特征点都可以用单词表中的单词近似代替，通过统计单词表中每个单词在图

像中出现的次数，可以将图像表示成为一个 $K=4$ 维数值向量。请看下图 5：



图 5 每幅图像的直方图表示

上图 5 中，我们从人脸、自行车和吉他三个目标类图像中提取出的不同视觉词汇，而构造的词汇表中，会把词义相近的视觉词汇合并为同一类，经过合并，词汇表中只包含了四个视觉单词，分别按索引值标记为 1, 2, 3, 4。通过观察可以看到，它们分别属于自行车、人脸、吉他、人脸类。统计这些词汇在不同目标类中出现的次数可以得到每幅图像的直方图表示（我们假定存在误差，实际情况亦不外如此）：

人脸： [3, 30, 3, 20]

自行车： [20, 3, 3, 2]

吉他： [8, 12, 32, 7]

其实这个过程非常简单，就是针对人脸、自行车和吉他这三个文档，抽取出相似的部分（或者词义相近的视觉词汇合并为同一类），构造一个词典，词典中包含 4 个视觉单词，即 $\text{Dictionary} = \{1: \text{"自行车"}, 2: \text{"人脸"}, 3: \text{"吉他"}, 4: \text{"人脸类"}\}$ ，最终人脸、自行车和吉他这三个文档皆可以用一个 4 维向量表示，最后根据三个文档相应部分出现的次数画成了上面对应的直方图。

需要说明的是，以上过程只是针对三个目标类非常简单的一个示例，实际应用中，为了达到较好的效果，单词表中的词汇数量 K 往往非常庞大，并且目标类数目越多，对应的 K 值也越大，一般情况下， K 的取值在几百到上千，在这里取 $K=4$ 仅仅是为了方便说明。

下面，我们再来总结一下如何利用 Bag-of-words 模型将一幅图像表示成为数值向量：

- 第一步：利用 SIFT 算法从不同类别的图像中提取视觉词汇向量，这些向量代表的是图像中局部不变的特征点；
- 第二步：将所有特征点向量集合到一块，利用 K-Means 算法合并词义相近的视觉词汇，构造一个包含 K 个词汇的单词表；

- 第三步：统计单词表中每个单词在图像中出现的次数，从而将图像表示成为一个 K 维数值向量。

下面我们按照以上步骤，用 C++一步步实现上述过程。

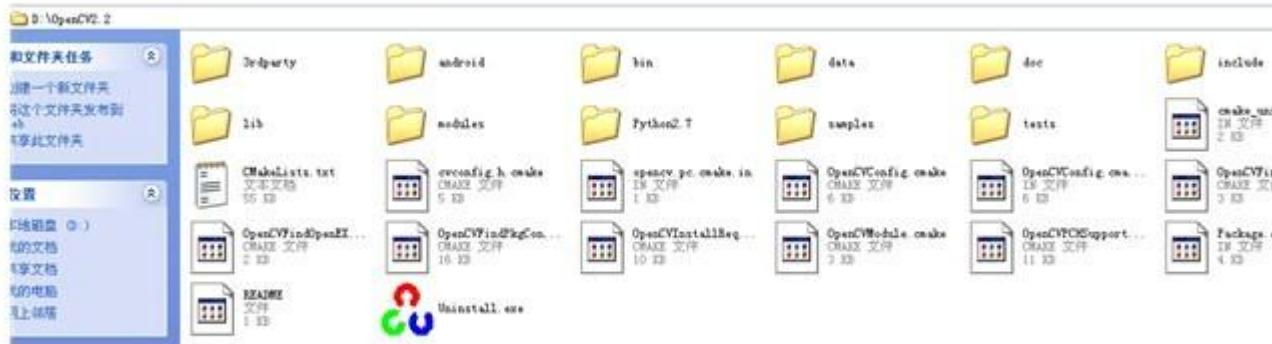
• C++逐步实现：Bag-of-words 模型表示一幅图像

在具体编码之前，我们需要事先搭配开发环境。

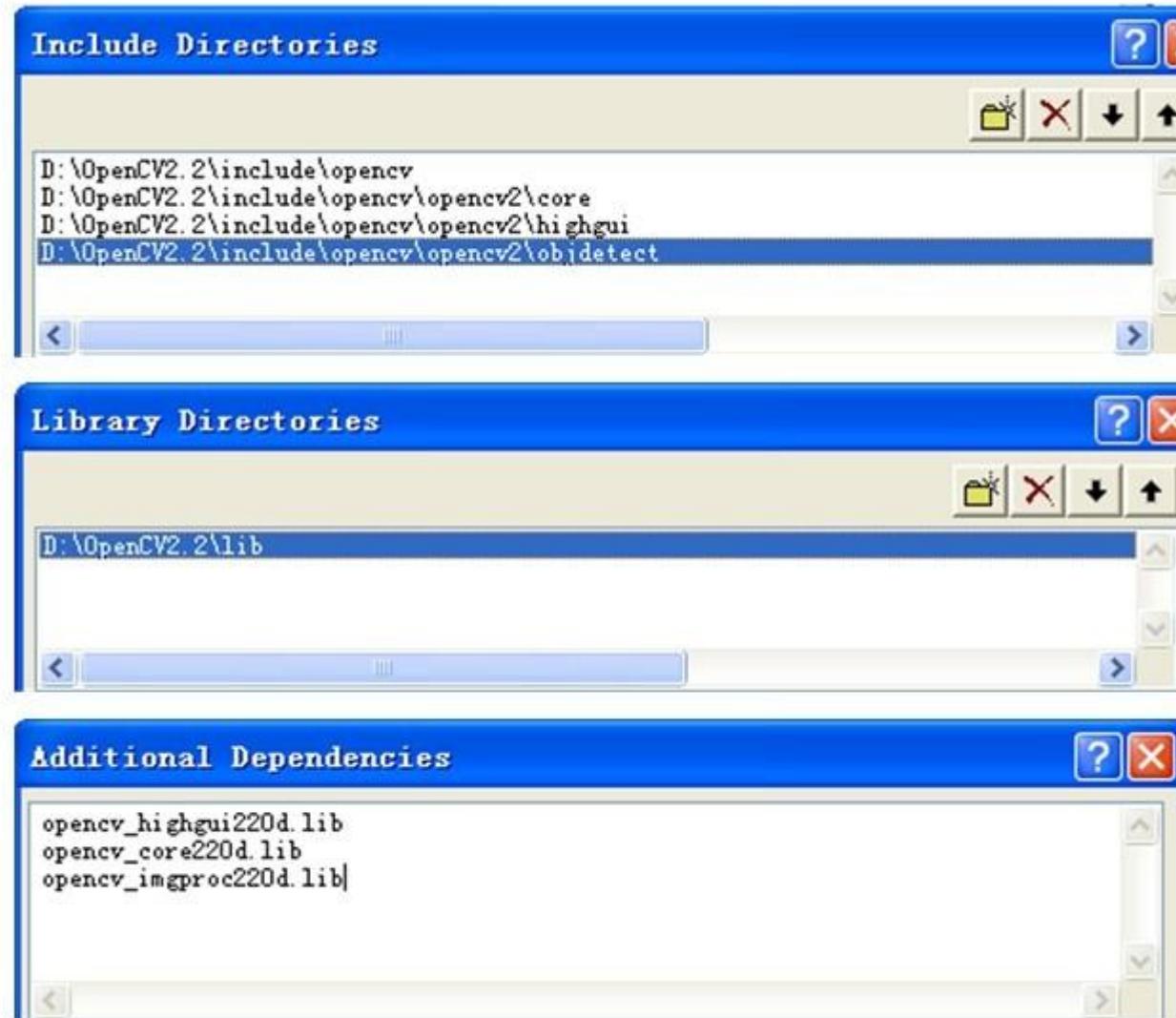
一. 搭建开发环境

使用的开发平台是 windows xp sp3 + vs2010 (windows xp sp3 + vc6.0 的情况，请参考此文：[九（续）、sift 算法的编译与实现](#))

1. 从 Rob Hess 的个人主页：<http://blogs.oregonstate.edu/hess/code/sift/>，下载最新版本的 sift 开源库源代码 sift-latest_win.zip；
2. 由于 sift-latest_win.zip 要求的 opencv 版本是 2.0 以上，也下载最新版本 OpenCV-2.2.0-win32-vs2010.exe，运行安装程序将 opencv 安装在本地某路径下。例如，我安装在 D 盘根目录下。



3. 运行 vs2010，建立一个空的控制台应用程序，取名 bow。
4. 配置 opencv 环境。在 vs2010 下选择 project 菜单下的 bow property 子菜单，调出 bow property pages 对话框，需要配置的地方有三处：在 vc++ Directory 选项里需要配置 Include Directories 和 Library Directories，在 Linker 选项卡的 Input 选项里需要配置 Additional Dependencies。



至此，开发环境全部搭建并配置完毕。

二. 创建 c++类 CSIFTDescriptor

为了方便使用，我们将 SIFT 库用 C++类 CSIFTDescriptor 封装，该类可以计算并获取指定图像的特征点向量集合。类的声明在 SIFTDescriptor.h 文件中，内容如下：

```
1. #ifndef _SIFT_DESCRIPTOR_H_
2. #define _SIFT_DESCRIPTOR_H_
3. #include <string>
4. #include <highgui.h>
5. #include <cv.h>
6.
7. extern "C"
8. {
9. #include "../sift/sift.h"
10. #include "../sift/imgfeatures.h"
11. #include "../sift/utils.h"
```

```

12. };
13.
14. class CSIFTDescriptor
15. {
16. public:
17.     int GetInterestPointNumber()
18.     {
19.         return m_nInterestPointNumber;
20.     }
21.     struct feature *GetFeatureArray()
22.     {
23.         return m_pFeatureArray;
24.     }
25. public :
26.     void SetImgName(const std::string &strImgName)
27.     {
28.         m_strInputImgName = strImgName;
29.     }
30.     int CalculateSIFT();
31. public:
32.     CSIFTDescriptor(const std::string &strImgName);
33.     CSIFTDescriptor()
34.     {
35.         m_nInterestPointNumber = 0;
36.         m_pFeatureArray = NULL;
37.     }
38.     ~CSIFTDescriptor();
39. private:
40.     std::string m_strInputImgName;
41.     int m_nInterestPointNumber;
42.     feature *m_pFeatureArray;
43. };
44. #endif

```

成员函数实现在 SIFTDescriptor.cpp 文件中，其中，CalculateSIFT 函数完成特征点的提取和计算，其主要内部流程如下：

- 1) 调用 OpenCV 函数 cvLoadImage 加载输入图像；
- 2) 为了统一输入图像的尺寸，CalculateSIFT 函数的第二步是调整输入图像的尺寸，这通过调用 cvResize 函数实现；
- 3) 如果输入图像是彩色图像，我们需要首先将其转化成灰度图，这通过调用 cvCvtColor 函数实现；
- 4) 调用 SIFT 库函数 sift_feature 获取输入图像的特征点向量集合和特征点个数。

```

1. #include "SIFTDescriptor.h"
2. int CSIFTDescriptor::CalculateSIFT()
3. {
4.     IplImage *pInputImg = cvLoadImage(m_strInputImgName.c_str());
5.     if (!pInputImg)
6.     {
7.         return -1;
8.     }
9.     int nImgWidth = 320;      //训练用标准图像大小
10.    double dbScaleFactor = pInputImg->width / 300.0;      //缩放因子
11.    IplImage *pTmpImg = cvCreateImage(cvSize(pInputImg->width / dbScaleFacto
r, pInputImg->height / dbScaleFactor),
12.        pInputImg->depth, pInputImg->nChannels);
13.    cvResize(pInputImg, pTmpImg);      //缩放
14.    cvReleaseImage(&pInputImg);
15.
16.    if (pTmpImg->nChannels != 1)      //非灰度图
17.    {
18.        IplImage *pGrayImg = cvCreateImage(cvSize(pTmpImg->width, pTmpImg->h
eight),
19.            pTmpImg->depth, 1);
20.        cvCvtColor(pTmpImg, pGrayImg, CV_RGB2GRAY);
21.        m_nInterestPointNumber = sift_features(pGrayImg, &m_pFeatureArray);

22.        cvReleaseImage(&pGrayImg);
23.    }
24.    else
25.    {
26.        m_nInterestPointNumber = sift_features(pTmpImg, &m_pFeatureArray);
27.    }
28.    cvReleaseImage(&pTmpImg);
29.    return m_nInterestPointNumber;
30. }
31. CSIFTDescriptor::CSIFTDescriptor(const std::string &strImgName)
32. {
33.     m_strInputImgName = strImgName;
34.     m_nInterestPointNumber = 0;
35.     m_pFeatureArray = NULL;
36.     CalculateSIFT();
37. }
38. CSIFTDescriptor::~CSIFTDescriptor()
39. {
40.     if (m_pFeatureArray)
41.     {

```

```
42.         free(m_pFeatureArray);
43.     }
44. }
```

三. 创建 c++类 CImgSet，管理实验图像集合

Bag-of-words 模型需要从多个目标类图像中提取视觉词汇，不同目标类的图像存储在不同子文件夹中，为了方便操作，我们设计了一个专门的类 CImgSet 用来管理图像集合，声明在文件 ImgSet.h 中：

```
1. #ifndef _IMG_SET_H_
2. #define _IMG_SET_H_
3. #include <vector>
4. #include <string>
5. #pragma comment(lib, "shlwapi.lib")
6. class CImgSet
7. {
8. public:
9.     CImgSet (const std::string &strImgDirName) : m_strImgDirName(strImgDirNa
    me+"//"), m_nImgNumber(0){}
10.    int GetTotalImageNumber()
11.    {
12.        return m_nImgNumber;
13.    }
14.    std::string GetImgName(int nIndex)
15.    {
16.        return m_szImgName.at(nIndex);
17.    }
18.    int LoadImgFromDir()
19.    {
20.        return LoadImgFromDir("");
21.    }
22. private:
23.     int LoadImgFromDir(const std::string &strDirName);
24. private:
25.     typedef std::vector<std::string> IMG_SET;
26.     IMG_SET m_szImgName;
27.     int m_nImgNumber;
28.     const std::string m_strImgDirName;
29. };
30. #endif
31.
32. //成员函数实现在文件 ImgSet.cpp 中:
33. #include "ImgSet.h"
34. #include <windows.h>
35. #include <Shlwapi.h>
```

```
36. /**
37. strSubDirName: 子文件夹名
38. */
39. int CImgSet::LoadImgsFromDir(const std::string &strSubDirName)
40. {
41.     WIN32_FIND_DATAA stFD = {0};
42.     std::string strDirName;
43.     if ("\" == strSubDirName)
44.     {
45.         strDirName = m_strImgDirName;
46.     }
47.     else
48.     {
49.         strDirName = strSubDirName;
50.     }
51.     std::string strFindName = strDirName + "/*";
52.     HANDLE hFile = FindFirstFileA(strFindName.c_str(), &stFD);
53.     BOOL bExist = FindNextFileA(hFile, &stFD);
54.
55.     for (;bExist;)
56.     {
57.         std::string strTmpName = strDirName + stFD.cFileName;
58.         if (strDirName + "." == strTmpName || strDirName + ".." == strTmpName)
59.     }
60.         bExist = FindNextFileA(hFile, &stFD);
61.         continue;
62.     }
63.     if (PathIsDirectoryA(strTmpName.c_str()))
64.     {
65.         strTmpName += "/";
66.         LoadImgsFromDir(strTmpName);
67.         bExist = FindNextFileA(hFile, &stFD);
68.         continue;
69.     }
70.     std::string strSubImg = strDirName + stFD.cFileName;
71.     m_szImgs.push_back(strSubImg);
72.     bExist = FindNextFileA(hFile, &stFD);
73. }
74. m_nImgNumber = m_szImgs.size();
75. return m_nImgNumber;
76. }
```

LoadImgFromDir 递归地从图像文件夹中获取所有实验用图像名，包括子文件夹。
该函数内部通过循环调用 windows API 函数 FindFirstFile 和 FindNextFile 来找到文件夹中所有图像的名称。

四. 创建 CHistogram，生成图像的直方图表示

```
1. //ImgHistogram.h
2.
3. #ifndef _IMG_HISTOGRAM_H_
4. #define _IMG_HISTOGRAM_H_
5.
6. #include <string>
7. #include "SIFTDescriptor.h"
8. #include "ImgSet.h"
9.
10. const int cnClusterNumber = 1500;
11. const int ciMax_D = FEATURE_MAX_D;
12.
13. class CHistogram
14. {
15. public:
16.     void SetTrainingImgSetName(const std::string strTrainingImgSet)
17.     {
18.         m_strTrainingImgSetName = strTrainingImgSet;
19.     }
20.     int FormHistogram();
21.     CvMat CalculateImgHistogram(const string strImgName, int *pszImgHistogram
22.     []);
22.     CvMat *GetObservedData();
23.     CvMat *GetCodebook()
24.     {
25.         return m_pCodebook;
26.     }
27.     void SetCodebook(CvMat *pCodebook)
28.     {
29.         m_pCodebook = pCodebook;
30.         m_bSet = true;
31.     }
32. public:
33.     CHistogram():m_pszHistogram(0), m_nImgNumber(0), m_pObservedData(0), m_p
34.     Codebook(0), m_bSet(false){}
34.     ~CHistogram()
35.     {
36.         if (m_pszHistogram)
```

```

37.         {
38.             delete m_pszHistogram;
39.             m_pszHistogram = 0;
40.         }
41.         if (m_pObservedData)
42.         {
43.             cvReleaseMat(&m_pObservedData);
44.             m_pObservedData = 0;
45.         }
46.         if (m_pCodebook && !m_bSet)
47.         {
48.             cvReleaseMat(&m_pCodebook);
49.             m_pCodebook = 0;
50.         }
51.     }
52. private :
53.     bool m_bSet;
54.     CvMat *m_pCodebook;
55.     CvMat *m_pObservedData;
56.     std::string m_strTrainingImgSetName;
57.     int (*m_pszHistogram)[cnClusterNumber];
58.     int m_nImgNumber;
59. };
60. #endif
61.
62. #include "ImgHistogram.h"
63. int CHistogram::FormHistogram()
64. {
65.     int nRet = 0;
66.     CIImgSet iImgSet(m_strTrainingImgSetName);
67.     nRet = iImgSet.LoadImgsFromDir();
68.
69.     const int cnTrainingImgNumber = iImgSet.GetTotalImageNumber();
70.     m_nImgNumber = cnTrainingImgNumber;
71.     CSIFTDescriptor *pDescriptor = new CSIFTDescriptor[cnTrainingImgNumber];
72.
73.     int nIPNumber(0) ;
74.     for (int i = 0; i < cnTrainingImgNumber; ++i) //计算每一幅训练图像的 SIFT
描述符
75.     {
76.         const string strImgName = iImgSet.GetImgName(i);
77.         pDescriptor[i].SetImgName(strImgName);
78.         pDescriptor[i].CalculateSIFT();
79.         nIPNumber += pDescriptor[i].GetInterestPointNumber();

```

```

79.      }
80.
81.      double (*pszDescriptor)[FEATURE_MAX_D] = new double[nIPNumber][FEATURE_M
   AX_D]; //存储所有描述符的数组。每一行代表一个 IP 的描述符
82.      ZeroMemory(pszDescriptor, sizeof(int) * nIPNumber * FEATURE_MAX_D);
83.      int nIndex = 0;
84.      for (int i = 0; i < cnTrainingImgNumber; ++i) //遍历所有图像
85.      {
86.          struct feature *pFeatureArray = pDescriptor[i].GetFeatureArray();
87.          int nFeatureNumber = pDescriptor[i].GetInterestPointNumber();
88.          for (int j = 0; j < nFeatureNumber; ++j) //遍历一幅图像中所有的
   IP(Interesting Point 兴趣点
89.          {
90.              for (int k = 0; k < FEATURE_MAX_D; k++)//初始化一个 IP 描述符
91.              {
92.                  pszDescriptor[nIndex][k] = pFeatureArray[j].descr[k];
93.              }
94.              ++nIndex;
95.          }
96.      }
97.      CvMat *pszLabels = cvCreateMat(nIPNumber, 1, CV_32SC1);
98.
99.      //对所有 IP 的描述符，执行 KMeans 算法，找到 cnClusterNumber 个聚类中心，存储在
   pszClusterCenters 中
100.     if (!m_pCodebook) //构造码元表
101.     {
102.         CvMat szSamples,
103.         *pszClusterCenters = cvCreateMat(cnClusterNumber, FEATURE_MAX_D
   , CV_32FC1);
104.         cvInitMatHeader(&szSamples, nIPNumber, FEATURE_MAX_D, CV_32FC1, psz
   Descriptor);
105.         cvKMeans2(&szSamples, cnClusterNumber, pszLabels,
106.             cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ),
107.             1, (CvRNG *)0, 0, pszClusterCenters); //
108.         m_pCodebook = pszClusterCenters;
109.     }
110.
111.     m_pszHistogram = new int[cnTrainingImgNumber][cnClusterNumber]; //存储
   每幅图像的直方图表示，每一行对应一幅图像
112.     ZeroMemory(m_pszHistogram, sizeof(int) * cnTrainingImgNumber * cnCluste
   rNumber);
113.
114.     //计算每幅图像的直方图
115.     nIndex = 0;

```

```

116.     for (int i = 0; i < cnTrainingImgNumber; ++i)
117.     {
118.         struct feature *pFeatureArray = pDescriptor[i].GetFeatureArray();
119.         int nFeatureNumber = pDescriptor[i].GetInterestPointNumber();
120.         //      int nIndex = 0;
121.         for (int j = 0; j < nFeatureNumber; ++j)
122.         {
123.             //          CvMat szFeature;
124.             //          cvInitMatHeader(&szFeature, 1, FEATURE_MAX_D, CV_32
125.             //          FC1, pszDescriptor[nIndex++]);
126.             //          double dbMinimum = 1.79769e308;
127.             //          int nCodebookIndex = 0;
128.             //          for (int k = 0; k < m_pCodebook->rows; ++k)//找到距
129.                 离最小的码元，用最小码元代替原//来的词汇
130.             //          {
131.                 //          CvMat szCode = cvMat(1, m_pCodebook->cols, m_pC
132.                 //          odebook->type);
133.                 //          cvGetRow(m_pCodebook, &szCode, k);
134.                 //          double dbDistance = cvNorm(&szFeature, &szCode,
135.                 //          CV_L2);
136.                 //          if (dbDistance < dbMinimum)
137.                 //          {
138.                     //          nCodebookIndex = pszLabels->data.i[nIndex++]; //找到第 i 幅
139.                     //          图像中第 j 个 IP 在 Codebook 中的索引值 nCodebookIndex
140.                     //          ++m_pszHistogram[i][nCodebookIndex]; //0<nCodebookIndex<cnClu
141.                     //          sterNumber;
142.                 }
143.             //资源清理，函数返回
144.             // delete []m_pszHistogram;
145.             // m_pszHistogram = 0;
146.
147.             cvReleaseMat(&pszLabels);
148.             // cvReleaseMat(&pszClusterCenters);
149.             delete []pszDescriptor;
150.             delete []pDescriptor;
151.
152.             return nRet;
153.     }

```

```

154.
155. //double descr_dist_sq( struct feature* f1, struct feature* f2 );
156. CvMat CHistogram::CalculateImgHistogram(const string strImgName, int pszImg
   Histogram[])
157. {
158.     if ("\" == strImgName || !m_pCodebook || !pszImgHistogram)
159.     {
160.         return CvMat();
161.     }
162.     CSIFTDescriptor iImgDisp;
163.     iImgDisp.SetImgName(strImgName);
164.     iImgDisp.CalculateSIFT();
165.     struct feature *pImgFeature = iImgDisp.GetFeatureArray();
166.     int cnIPNumber = iImgDisp.GetInterestPointNumber();
167.     // int *pszImgHistogram = new int[cnClusterNumber];
168.     // ZeroMemory(pszImgHistogram, sizeof(int)*cnClusterNumber);
169.     for (int i = 0; i < cnIPNumber; ++i)
170.     {
171.         double *pszDistance = new double[cnClusterNumber];
172.         CvMat iIP = cvMat(FEATURE_MAX_D, 1, CV_32FC1, pImgFeature[i].descr)
173.         ;
174.         for (int j = 0; j < cnClusterNumber; ++j)
175.         {
176.             CvMat iCode = cvMat(1, FEATURE_MAX_D, CV_32FC1);
177.             cvGetRow(m_pCodebook, &iCode, j);
178.             CvMat *pTmpMat = cvCreateMat(FEATURE_MAX_D, 1, CV_32FC1);
179.             cvTranspose(&iCode, pTmpMat);
        double dbDistance = cvNorm(&iIP, pTmpMat); //计算第 i 个 IP 与第 j
        个 code 之间的距离
180.             pszDistance[j] = dbDistance;
181.             cvReleaseMat(&pTmpMat);
182.         }
183.         double dbMinDistance = pszDistance[0];
184.         int nCodebookIndex = 0; //第 i 个 IP 在 codebook 中距离最小的 code 的索引
        值
185.         for (int j = 1; j < cnClusterNumber; ++j)
186.         {
187.             if (dbMinDistance > pszDistance[j])
188.             {
189.                 dbMinDistance = pszDistance[j];
190.                 nCodebookIndex = j;
191.             }
192.         }
193.         ++pszImgHistogram[nCodebookIndex];

```

```

194.         delete []pszDistance;
195.     }
196.     CvMat iImgHistogram = cvMat(cnClusterNumber, 1, CV_32SC1, pszImgHistogram);
197.     return iImgHistogram;
198. }
199.
200. CvMat *CHistogram::GetObservedData()
201. {
202.     CvMat iHistogram;
203.     cvInitMatHeader(&iHistogram, m_nImgNumber, cnClusterNumber, CV_32SC1, m_pszHistogram);
204.     CvMat *m_pObservedData = cvCreateMat(iHistogram.cols, iHistogram.rows,
205.                                         CV_32SC1);
206.     cvTranspose(&iHistogram, m_pObservedData);
207.     return m_pObservedData;

```

本文完。

版权所有，侵权必究。严禁用于任何商业用途，转载请注明出处。

十、从头到尾彻底理解傅里叶变换算法、上

作者：July、dznlong 二零一一年二月二十日

推荐阅读：The Scientist and Engineer's Guide to Digital Signal Processing, By Steven W. Smith, Ph.D。此书地址：<http://www.dspguide.com/pdfbook.htm>。

博主说明：

I、本文中阐述离散傅里叶变换方法，是根据此书：The Scientist and Engineer's Guide to Digital Signal Processing，By Steven W. Smith, Ph.D.而翻译而成的，此书地址：<http://www.dspguide.com/pdfbook.htm>。

II、同时，有相当一部分内容编辑整理自 dznlong 的博客，也贴出其博客地址，向原创的作者表示致敬：<http://blog.csdn.net/dznlong>。这年头，真正静下心写来原创文章的人，很少了。

从头到尾彻底理解傅里叶变换算法、上

前言

第一部分、 DFT

第一章、傅立叶变换的由来

第二章、实数形式离散傅立叶变换 (Real DFT)

从头到尾彻底理解傅里叶变换算法、下

第三章、复数

第四章、复数形式离散傅立叶变换

前言：

“关于傅立叶变换，无论是书本还是在网上可以很容易找到关于傅立叶变换的描述，但是大都是些故弄玄虚的文章，太过抽象，尽是一些让人看了就望而生畏的公式的罗列，让人很难能够从感性上得到理解”--dznlong，

那么，到底什么是傅里叶变换算法？傅里叶变换所涉及到的公式具体有多复杂？

傅里叶变换 (Fourier transform) 是一种线性的积分变换。因其基本思想首先由法国学者傅里叶系统地提出，所以以其名字来命名以示纪念。

哦，傅里叶变换原来就是一种变换而已，只是这种变换是从时间转换为频率的变化。这下，你就知道了，傅里叶就是一种变换，一种什么变换？就是一种从时间到频率的变化或其相互转化。

ok，咱们再来总体了解下傅里叶变换，让各位对其有个总体大概的印象，也顺便看看傅里叶变换所涉及到的公式，究竟有多复杂：

以下就是傅里叶变换的 4 种变体（摘自，维基百科）

连续傅里叶变换

一般情况下，若“傅里叶变换”一词不加任何限定语，则指的是“连续傅里叶变换”。连续傅里叶变换将平方可积的函数 $f(t)$ 表示成复指数函数的积分或级数形式。

$$F(\omega) = \mathcal{F}[f(t)] = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt.$$

这是将频率域的函数 $F(\omega)$ 表示为时间域的函数 $f(t)$ 的积分形式。

连续傅里叶变换的逆变换 (inverse Fourier transform) 为：

$$f(t) = \mathcal{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega.$$

即将时间域的函数 $f(t)$ 表示为频率域的函数 $F(\omega)$ 的积分。

一般可称函数 $f(t)$ 为原函数，而称函数 $F(\omega)$ 为傅里叶变换的像函数，原函数和像函数构成一个傅里叶变换对（transform pair）。

除此之外，还有其它型式的变换对，以下两种型式亦常被使用。在通信或是信号处理方面，

常以 $f = \frac{\omega}{2\pi}$ 来代换，而形成新的变换对：

$$\begin{aligned} X(f) &= \mathcal{F}[x(t)] = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft} dt \\ x(t) &= \mathcal{F}^{-1}[X(f)] = \int_{-\infty}^{\infty} X(f) e^{i2\pi ft} df. \end{aligned}$$

或者是因系数重分配而得到新的变换对：

$$\begin{aligned} F(\omega) &= \mathcal{F}[f(t)] = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \\ f(t) &= \mathcal{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega. \end{aligned}$$

一种对连续傅里叶变换的推广称为分数傅里叶变换（Fractional Fourier Transform）。分数傅里叶变换(fractional Fourier transform,FRFT)指的就是傅里叶变换(Fourier transform,FT)的广义化。

分数傅里叶变换的物理意义即做傅里叶变换 a 次，其中 a 不一定要为整数；而做了分数傅里叶变换之后，信号或输入函数便会在介于时域(time domain)与频域(frequency domain)之间的分数域(fractional domain)。

当 $f(t)$ 为偶函数(或奇函数)时，其正弦(或余弦)分量将消亡，而可以称这时的变换为余弦变换(cosine transform)或正弦变换(sine transform)。

另一个值得注意的性质是，当 $f(t)$ 为纯实函数时， $F(-\omega) = F^*(\omega)$ 成立。

傅里叶级数

连续形式的傅里叶变换其实是傅里叶级数 (Fourier series) 的推广，因为积分其实是一种极限形式的求和算子而已。对于周期函数，其傅里叶级数是存在的：

$$f(x) = \sum_{n=-\infty}^{\infty} F_n e^{inx},$$

其中 F_n 为复幅度。对于实值函数，函数的傅里叶级数可以写成：

$$f(x) = a_0 + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)]$$

其中 a_n 和 b_n 是实频率分量的幅度。

离散时域傅里叶变换

离散傅里叶变换是离散时间傅里叶变换 (DTFT) 的特例 (有时作为后者的近似)。DTFT 在时域上离散，在频域上则是周期的。DTFT 可以被看作是傅里叶级数的逆变换。

离散傅里叶变换

离散傅里叶变换 (DFT)，是连续傅里叶变换在时域和频域上都离散的形式，将时域信号的采样变换为在离散时间傅里叶变换 (DTFT) 频域的采样。在形式上，变换两端 (时域和频域上) 的序列是有限长的，而实际上这两组序列都应当被认为是离散周期信号的主值序列。即使对有限长的离散信号作 DFT，也应当将其看作经过周期延拓成为周期信号再作变换。在实际应用中通常采用快速傅里叶变换以高效计算 DFT。

为了在科学计算和数字信号处理等领域使用计算机进行傅里叶变换，必须将函数 x_n 定义在离散点而非连续域内，且须满足有限性或周期性条件。这种情况下，使用[离散傅里叶变换 \(DFT\)](#)，将函数 x_n 表示为下面的求和形式：

$$x_n = \sum_{k=0}^{N-1} X_k e^{i \frac{2\pi}{N} kn} \quad n = 0, \dots, N-1$$

其中 X_k 是傅里叶幅度。直接使用这个公式计算的计算复杂度为 $O(n^2)$ ，而[快速傅里叶变换 \(FFT\)](#) 可以将复杂度改进为 $O(n \log n)$ 。（后面会具体阐述 FFT 是如何将复杂度降为 $O(n \log n)$ 的。）计算复杂度的降低以及数字电路计算能力的发展使得 DFT 成为在信号处理领域十分实用且重要的方法。

下面，比较下上述傅立叶变换的 4 种变体，

变换	时间	频率
连续傅里叶变换	连续, 非周期性	连续, 非周期性
傅里叶级数	连续, 周期性	离散, 非周期性
离散时间傅里叶变换	离散, 非周期性	连续, 周期性
离散傅里叶变换	离散, 周期性	离散, 周期性

如上，容易发现：函数在时（频）域的离散对应于其像函数在频（时）域的周期性。反之连续则意味着在对应域的信号的非周期性。也就是说，时间上的离散性对应着频率上的周期性。同时，注意，离散时间傅里叶变换，时间离散，频率不离散，它在频域依然是连续的。

如果，读到此，你不甚明白，大没关系，不必纠结于以上 4 种变体，继续往下看，你自会豁然开朗。（有什么问题，也恳请提出，或者批评指正）

ok，本文，接下来，由傅里叶变换入手，后重点阐述离散傅里叶变换、快速傅里叶算法，到最后彻底实现 FFT 算法，全篇力求通俗易懂、阅读顺畅，教你从头到尾彻底理解傅里叶变换算法。由于傅里叶变换，也称傅立叶变换，下文所称为 [傅立叶变换](#)，同一个变换，不同叫法，读者不必感到奇怪。

第一部分、DFT

第一章、傅立叶变换的由来

要理解傅立叶变换，先得知道傅立叶变换是怎么变换的，当然，也需要一定的高等数学基础，最基本的是级数变换，其中傅立叶级数变换是傅立叶变换的基础公式。

一、傅立叶变换的提出

傅立叶是一位法国数学家和物理学家，原名是 Jean Baptiste Joseph Fourier(1768-1830)，Fourier 于 1807 年在法国科学学会上发表了一篇论文，论文里描述运用正弦曲线来描述温度分布，论文里有个在当时具有争议性的决断：任何连续周期信号都可以由一组适当的正弦曲线组合而成。

当时审查这个论文拉格朗日坚决反对此论文的发表，而后在近 50 年的时间里，拉格朗日坚持认为傅立叶的方法无法表示带有棱角的信号，如在方波中出现非连续变化斜率。直到拉格朗日死后 15 年这个论文才被发表出来。

谁是对的呢？拉格朗日是对的：正弦曲线无法组合成一个带有棱角的信号。但是，我们可以用正弦曲线来非常逼近地表示它，逼近到两种表示方法不存在能量差别，基于此，傅立叶是对的。

为什么我们要用正弦曲线来代替原来的曲线呢？如我们也还可以用方波或三角波来代替呀，分解信号的方法是无穷多的，但分解信号的目的是为了更加简单地处理原来的信号。

用正余弦来表示原信号会更加简单，因为正余弦拥有原信号所不具有的性质：正弦曲

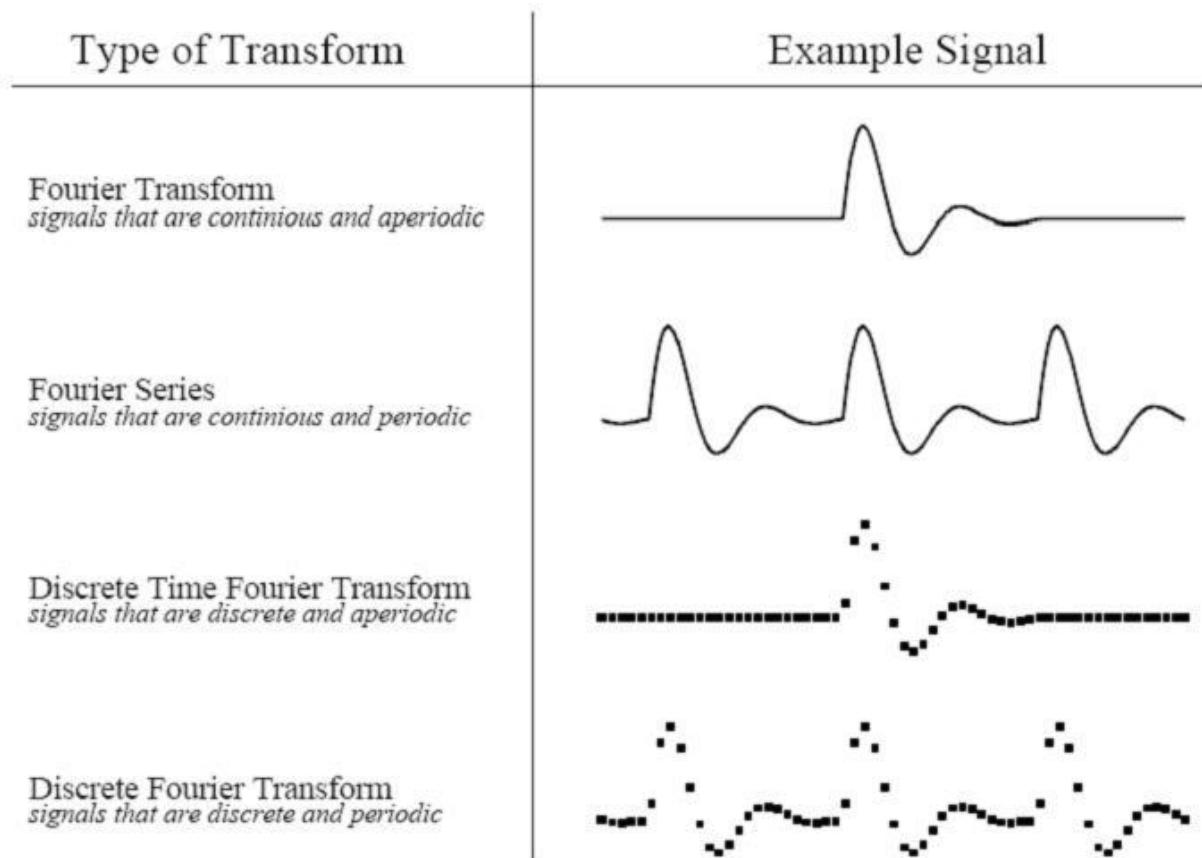
线保真度。一个正余弦曲线信号输入后，输出的仍是正余弦曲线，只有幅度和相位可能发生变化，但是频率和波的形状仍是一样的。且只有正余弦曲线才拥有这样的性质，正因如此我们才不用方波或三角波来表示。

二、傅立叶变换分类

根据原信号的不同类型，我们可以把傅立叶变换分为四种类别：

- | | |
|------------|---|
| 1、非周期性连续信号 | 傅立叶变换 (Fourier Transform) |
| 2、周期性连续信号 | 傅立叶级数(Fourier Series) |
| 3、非周期性离散信号 | 离散时域傅立叶变换 (Discrete Time Fourier Transform) |
| 4、周期性离散信号 | 离散傅立叶变换(Discrete Fourier Transform) |

下图是四种原信号图例（从上到下，依次是 FT, FS, DTFT, DFT）：



这四种傅立叶变换都是针对正无穷大和负无穷大的信号，即信号的长度是无穷大的，我们知道这对于计算机处理来说是不可能的，那么有没有针对长度有限的傅立叶变换呢？没有。因为正余弦波被定义成从负无穷小到正无穷大，我们无法把一个长度无限的信号组合成长度有限的信号。

面对这种困难，方法是：把长度有限的信号表示成长度无限的信号。如，可以把信号无限地从左右进行延伸，延伸的部分用零来表示，这样，这个信号就可以被看成是非周期性离散信号，我们可以用到[离散时域傅立叶变换 \(DTFT\)](#)的方法。也可以把信号用复制的方法进行延伸，这样信号就变成了周期性离散信号，这时我们就可以用[离散傅立叶变换方法 \(DFT\)](#)进行变换。本章我们要讲的是离散信号，对于连续信号我们不作讨论，因为计算

机只能处理离散的数值信号，我们的最终目的是运用计算机来处理信号的。

但是对于非周期性的信号，我们需要用无穷多不同频率的正弦曲线来表示，这对于计算机来说是不可能实现的。所以对于离散信号的变换只有**离散傅立叶变换（DFT）**才能被适用，对于计算机来说只有离散的和有限长度的数据才能被处理，对于其它的变换类型只有在数学演算中才能用到，在计算机面前我们只能用**DFT**方法，后面我们要理解的也正是**DFT**方法。

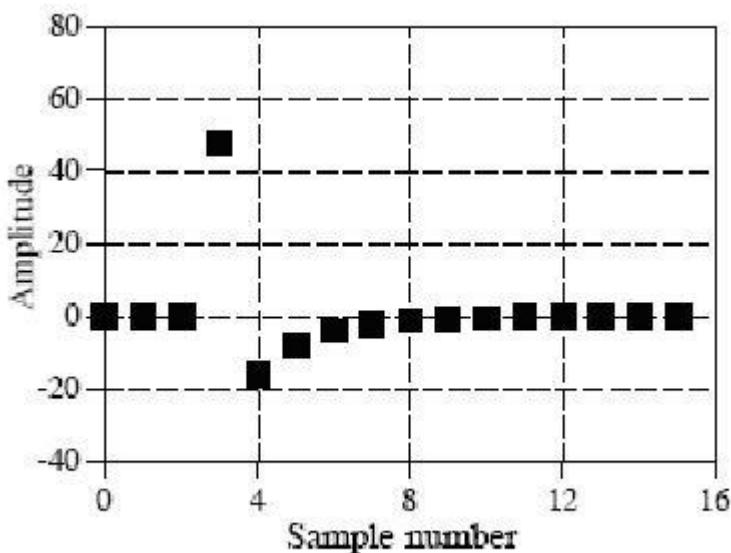
这里要理解的是我们使用周期性的信号目的是为了能够用数学方法来解决问题，至于考虑周期性信号是从哪里得到或怎样得到是无意义的。

每种傅立叶变换都分成实数和复数两种方法，对于实数方法是最好理解的，但是复数方法就相对复杂许多了，需要懂得有关复数的理论知识，不过，如果理解了实数离散傅立叶变换(**real DFT**)，再去理解复数傅立叶变换就更容易了，所以我们先把复数的傅立叶变换放到一边去，先来理解实数傅立叶变换，在后面我们会先讲讲关于复数的基本理论，然后在理解了实数傅立叶变换的基础上再来理解复数傅立叶变换。

还有，这里我们所要说的变换(**transform**)虽然是数学意义上的变换，但跟函数变换是不同的，函数变换是符合一一映射准则的，对于离散数字信号处理(**DSP**)，有许多的变换：傅立叶变换、拉普拉斯变换、Z变换、希尔伯特变换、离散余弦变换等，这些都扩展了函数变换的定义，允许输入和输出有多种的值，简单地说变换就是把一堆的数据变成另一堆的数据的方法。

三、一个关于实数离散傅立叶变换(**Real DFT**)的例子

先来看一个变换实例，下图是一个原始信号图像：

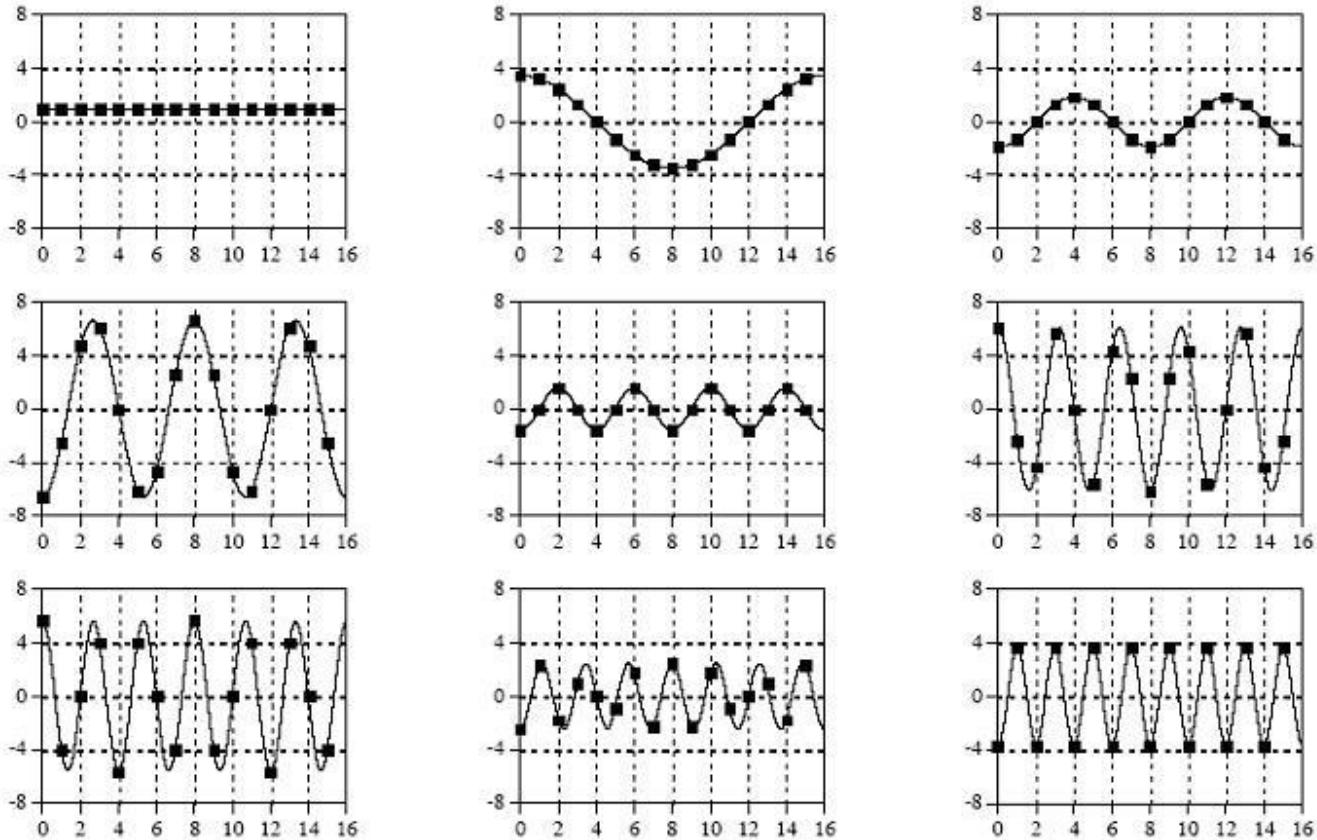


这个信号的长度是 16，于是可以把这个信号分解 9 个余弦波和 9 个正弦波（一个长度为 N 的信号可以分解成 $N/2+1$ 个正余弦信号，这是为什么呢？结合下面的 18 个正余弦图，

我想从计算机处理精度上就不难理解，一个长度为 N 的信号，最多只能有 $N/2+1$ 个不同频率，再多的频率就超过了计算机所能所处理的精度范围），如下图：

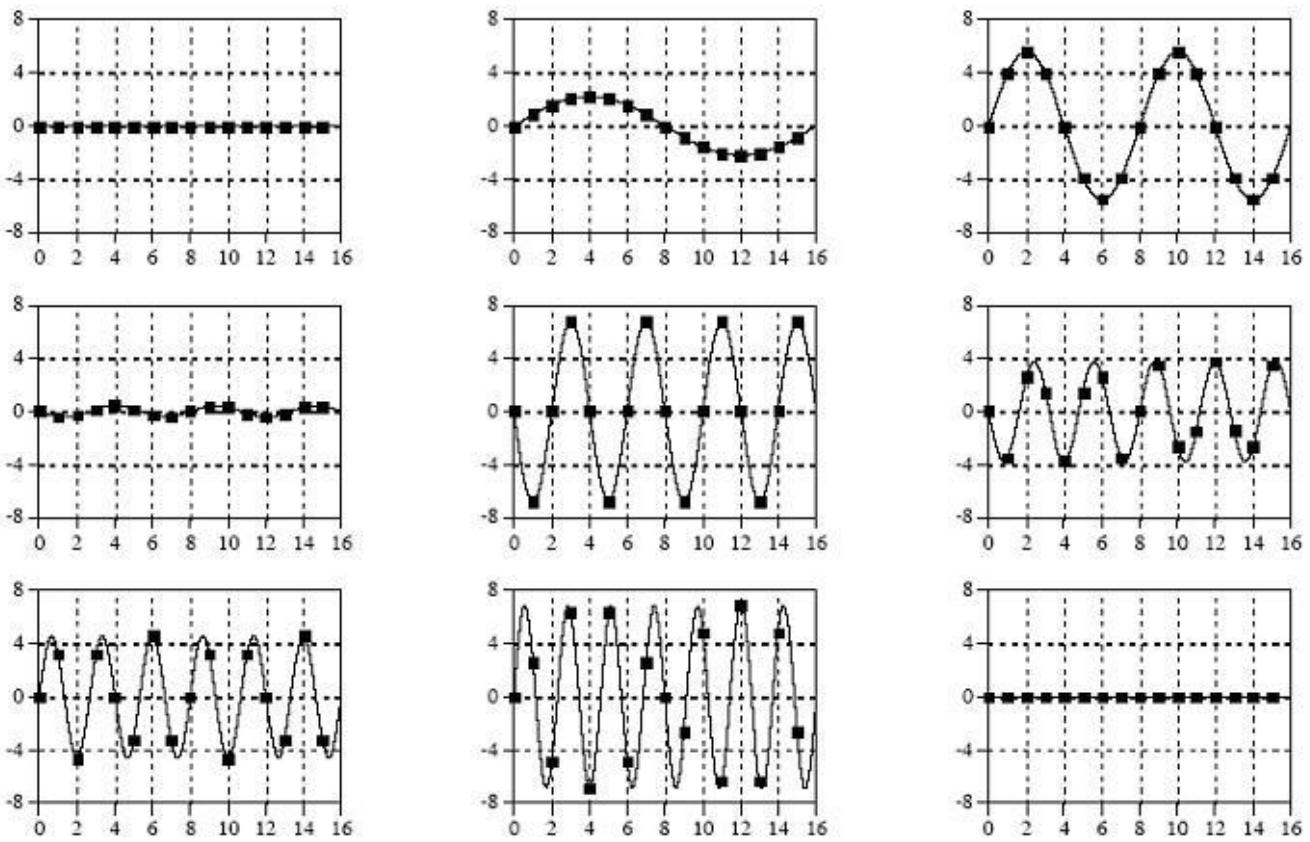
9 个余弦信号：

Cosine Waves

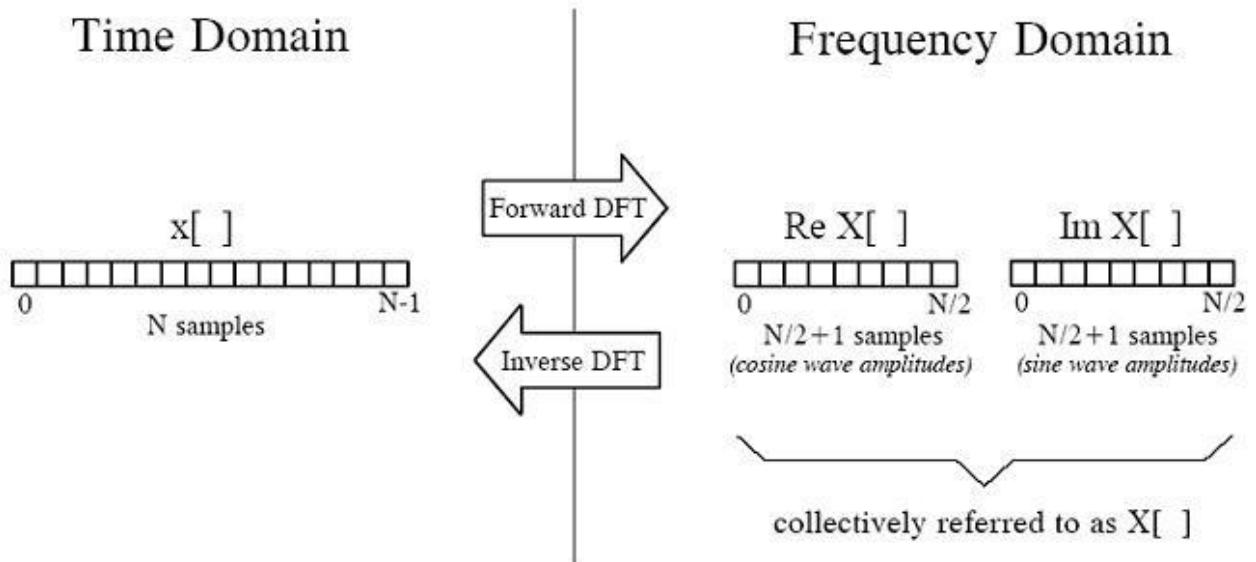


9 个正弦信号：

Sine Waves



把以上所有信号相加即可得到原始信号，至于是怎么分别变换出 9 种不同频率信号的，我们先不急，先看看对于以上的变换结果，在程序中又是该怎么表示的，我们可以看看下面这个示例图：



上图中左边表示时域中的信号，右边是频域信号表示方法，从左向右， \rightarrow ，表示正向转换(Forward DFT)，从右向左， \leftarrow ，表示逆向转换(Inverse DFT)，用小写 $x[]$ 表示信号在每个时间点上的幅度值数组，用大写 $X[]$ 表示每种频率的副度值数组（即时间 $x \rightarrow$ 频率 X ），因为有 $N/2+1$ 种频率，所以该数组长度为 $N/2+1$ ， $X[]$ 数组又分两种，一种是表示余弦波的不同频率幅度值： $\text{Re } X[]$ ，另一种是表示正弦波的不同频率幅度值： $\text{Im } X[]$ 。

Re 是实数(Real)的意思， Im 是虚数(Imagine)的意思，采用复数的表示方法把正余弦波组合起来进行表示，但这里我们不考虑复数的其它作用，只记住是一种组合方法而已，目的是为了便于表达（在后面我们会知道，复数形式的傅立叶变换长度是 N ，而不是 $N/2+1$ ）。如此，再回过头去，看上面的正余弦各 9 种频率的变化，相信，问题不大了。

第二章、实数形式离散傅立叶变换 (Real DFT)

上一章，我们看到了一个实数形式离散傅立叶变换的例子，通过这个例子能够让我们先对傅立叶变换有一个较为形象的感性认识，现在就让我们来看看实数形式离散傅立叶变换的正向和逆向是怎么进行变换的。在此，我们先来看一下频率的多种表示方法。

一、频域中关于频率的四种表示方法

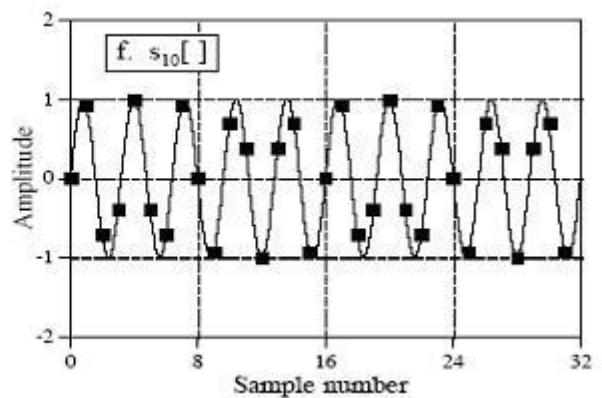
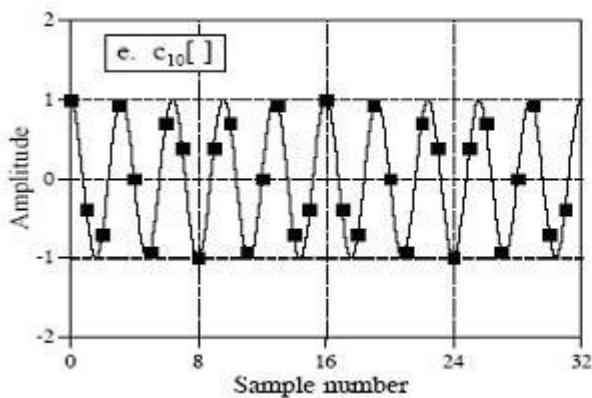
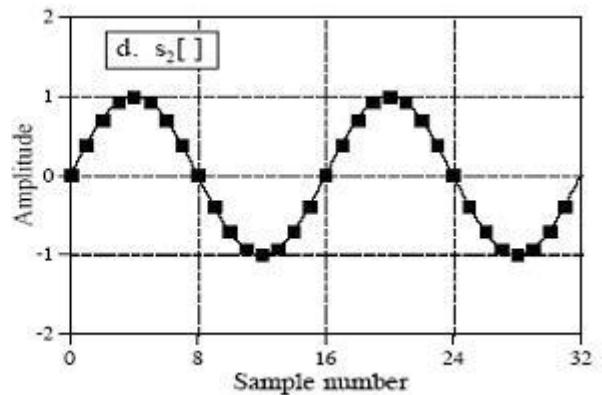
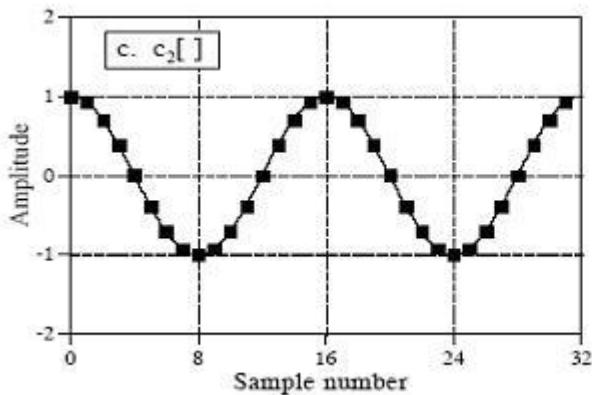
- 1、序号表示方法，根据时域中信号的样本数取 $0 \sim N/2$ ，用这种方法在程序中使用起来可以更直接地取得每种频率的幅度值，因为频率值跟数组的序号是一一对应的： $X[k]$ ，取值范围是 $0 \sim N/2$ ；
- 2、分数表示方法，根据时域中信号的样本数的比例值取 $0 \sim 0.5$ ： $X[f]$ ， $f = k/N$ ，取值范围是 $0 \sim 1/2$ ；
- 3、用弧度值来表示，把 f 乘以一个 2π 得到一个弧度值，这种表示方法叫做自然频率 (natural frequency)： $X[\omega]$ ， $\omega = 2\pi f = 2\pi k/N$ ，取值范围是 $0 \sim \pi$ ；
- 4、以赫兹(Hz)为单位来表示，这个一般是应用于一些特殊应用，如取样率为 10 kHz 表示每秒有 10,000 个样本数：取值范围是 0 到取样率的一半。

二、DFT 基本函数

$$ck[i] = \cos(2\pi ki/N)$$

$$sk[i] = \sin(2\pi ki/N)$$

其中 k 表示每个正余弦波的频率，如为 2 表示在 0 到 N 长度中存在两个完整的周期，10 即有 10 个周期，如下图：



上图中至于每个波的振幅(amplitude)值($\text{Re } X[k], \text{Im } X[k]$)是怎么算出来的,这个是DFT的核心,也是最难理解的部分,我们先来看看如何把分解出来的正余弦波合成原始信号(Inverse DFT)。

三、合成运算方法(Real Inverse DFT)

DFT 合成等式 (合成原始时间信号, 频率-->时间, 逆向变换) :

$$x[i] = \sum_{k=0}^{N/2} \text{Re } \bar{X}[k] \cos(2\pi ki/N) + \sum_{k=0}^{N/2} \text{Im } \bar{X}[k] \sin(2\pi ki/N)$$

如果有学过傅立叶级数,对这个等式就会有似曾相识的感觉,不错!这个等式跟傅立叶级数是非常相似的:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k \cos kx + b_k \sin kx)$$

当然，差别是肯定存在的，因为这两个等式是在两个不同条件下运用的，至于怎么证明 DFT 合成公式，这个我需要非常强的高等数学理论知识了，这是研究数学的人的工作，对于普通应用者就不需要如此的追根究底了，但是傅立叶级数是好理解的，我们起码可以从傅立叶级数公式中看出 DFT 合成公式的合理性。

DFT 合成等式中的 $\bar{\text{Im } X[k]}$ 和 $\bar{\text{Re } X[k]}$ 跟之前提到的 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 是不一样的，下面是转换方法（[关于此公式的解释，见下文](#)）：

$$\bar{\text{Re } X}[k] = \frac{\text{Re } X[k]}{N/2}$$

$$\bar{\text{Im } X}[k] = -\frac{\text{Im } X[k]}{N/2}$$

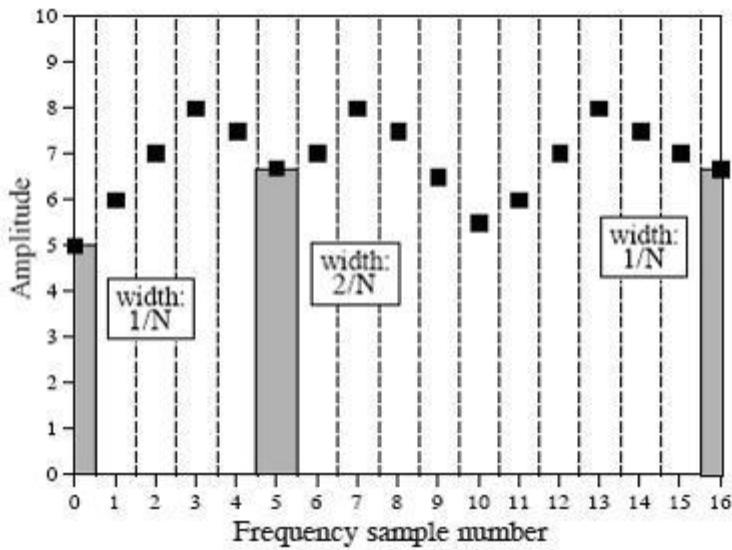
但 k 等于 0 和 $N/2$ 时，实数部分的计算要用下面的等式：

$$\bar{\text{Re } X}[0] = \frac{\text{Re } X[0]}{N}$$

$$\bar{\text{Re } X}[N/2] = \frac{\text{Re } X[N/2]}{N}$$

上面四个式中的 N 是时域中点的总数， k 是从 0 到 $N/2$ 的序号。

为什么要这样进行转换呢？这个可以从频谱密度(**spectral density**)得到理解，如下图就是个频谱图：



这是一个频谱图，横坐标表示频率大小，纵坐标表示振幅大小，原始信号长度为 N （这里是 32），经 DFT 转换后得到的 17 个频率的频谱，频谱密度表示每单位带宽中为多大的振幅，那么带宽是怎么计算出来的呢？看上图，除了头尾两个，其余点的所占的宽度是 $2/N$ ，这个宽度便是每个点的带宽，头尾两个点的带宽是 $1/N$ ，而 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 表示的是频谱密度，即每一个单位带宽的振幅大小，但 $\text{Im } \bar{X}[k]$ 和 $\text{Re } \bar{X}[k]$ 表示 $2/N$ （或 $1/N$ ）带宽的振幅大小，所以 $\text{Im } \bar{X}[k]$ 和 $\text{Re } \bar{X}[k]$ 分别应当是 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 的 $2/N$ （或 $1/N$ ）。

频谱密度就象物理中物质密度，原始信号中的每一个点就象是一个混合物，这个混合物是由不同密度的物质组成的，混合物中含有的每种物质的质量是一样的，除了最大和最小两个密度的物质外，这样我们只要把每种物质的密度加起来就可以得到该混合物的密度了，又该混合物的质量是单位质量，所以得到的密度值跟该混合物的质量值是一样的。

至于为什么虚数部分是负数，这是为了跟复数 DFT 保持一致，这个我们将在后面会知道这是数学计算上的需要（ $\text{Im } X[k]$ 在计算时就已经加上了一个负号（[稍后，由下文，便可知](#)）， $\text{Im } \bar{X}[k]$ 再加上负号，结果便是正的，等于没有变化）。

如果已经得到了 DFT 结果，这时要进行逆转换，即合成原始信号，则可按如下步骤进行转换：

1、先根据上面四个式子计算得出 $\text{Im } \bar{X}[k]$ 和 $\text{Re } \bar{X}[k]$ 的值；

2、再根据 DFT 合成等式得到原始信号数据。

下面是用 BASIC 语言来实现的转换源代码：

100 'DFT 逆转换方法

```

110 '/XX[]数组存储计算结果（时域中的原始信号）
120 '/REX[]数组存储频域中的实数分量，IMX[]为虚分量
130 '
140 DIM XX[511]
150 DIM REX[256]
160 DIM IMX[256]
170 '
180 PI = 3.14159265
190 N% = 512
200 '
210 GOSUB XXXX '转到子函数去获取 REX[]和 IMX[]数据
220 '
230 '
240 '
250 FOR K% = 0 TO 256
260   REX[K%] = REX[K%] / (N%/2)
270   IMX[K%] = -IMX[K%] / (N%/2)
280 NEXT k%
290 '
300 REX[0] = REX[0] / N
310 REX[256] = REX[256] / N
320 '
330 ' 初始化 XX[]数组
340 FOR I% = 0 TO 511
350   XX[I%] = 0
360 NEXT I%
370 '
380 '
390 '
400 '
410 '
420 FOR K% =0 TO 256
430   FOR I%=0 TO 511
440 '
450     XX[I%] = XX[I%] + REX[K%] * COS(2 * PI * K% * I% / N%)
460     XX[I%] = XX[I%] + IMX[K%] * SIN(2 * PI * K% * I% / N%)
470 '
480   NEXT I%
490 NEXT K%
500 '
510 END

```

上面代码中 420 至 490 换成如下形式也许更好理解，但结果都是一样的：

420 FOR I% =0 TO 511

```

430 FOR K%=0 TO 256
440 '
450     XX[I%] = XX[I%] + REX[K%] * COS(2 * PI * K% * I% / N%)
460     XX[I%] = XX[I%] + IMX[K%] * SIN(2 * PI * K% * I% / N%)
470 '
480 NEXT I%
490 NEXT K%

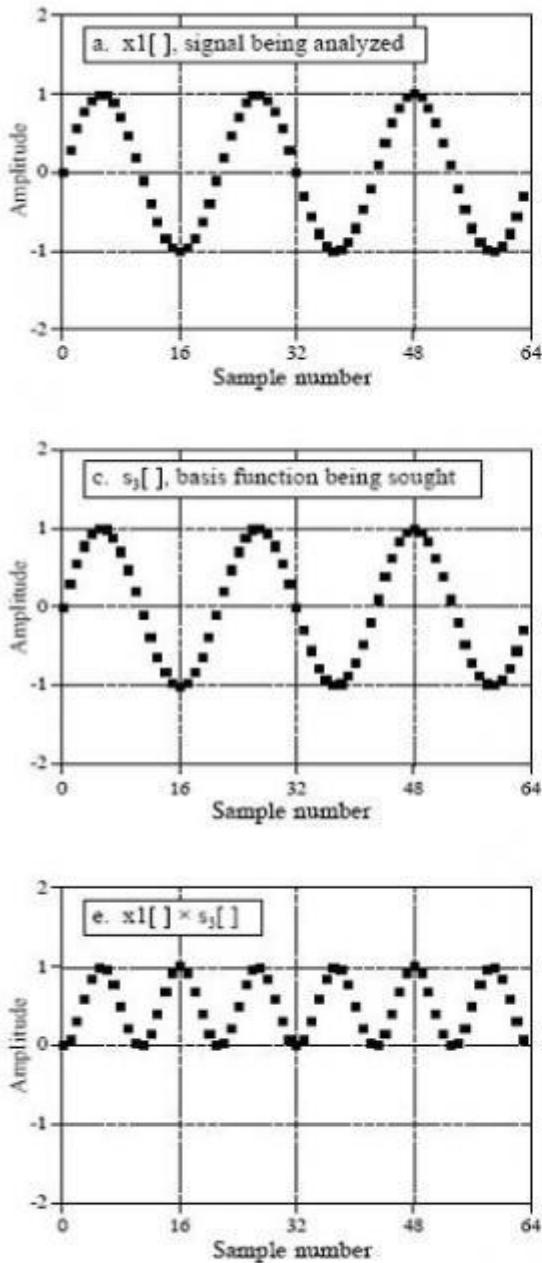
```

四、 分解运算方法 (DFT)

有三种完全不同的方法进行 DFT: 一种方法是通过联立方程进行求解, 从代数的角度看, 要从 N 个已知值求 N 个未知值, 需要 N 个联立方程, 且 N 个联立方程必须是线性独立的, 但这是这种方法计算量非常的大且极其复杂, 所以很少被采用; 第二种方法是利用信号的相关性 (correlation) 进行计算, 这个是我们后面将要介绍的方法; 第三种方法是快速傅立叶变换 (FFT), 这是一个非常具有创造性和革命性的方法, 因为它大大提高了运算速度, 使得傅立叶变换能够在计算机中被广泛应用, 但这种算法是根据复数形式的傅立叶变换来实现的, 它把 N 个点的信号分解成长度为 N 的频域, 这个跟我们现在所进行的实域 DFT 变换不一样, 而且这种方法也较难理解, 这里我们先不去理解, 等先理解了复数 DFT 后, 再来看一下 FFT。有一点很重要, 那就是这三种方法所得的变换结果是一样的, 经过实践证明, 当频域长度为 32 时, 利用相关性方法进行计算效率最好, 否则 FFT 算法效率较高。现在就让我们来看一下相关性算法。

利用第一种方法、信号的相关性(correlation)可以从噪声背景中检测出已知的信号, 我们也可以利用这个方法检测信号波中是否含有某个频率的信号波: 把一个待检测信号波乘以另一个信号波, 得到一个新的信号波, 再把这个新的信号波所有的点进行相加, 从相加的结果就可以判断出这两个信号的相似程度。如下图:

Example 1



Example 2

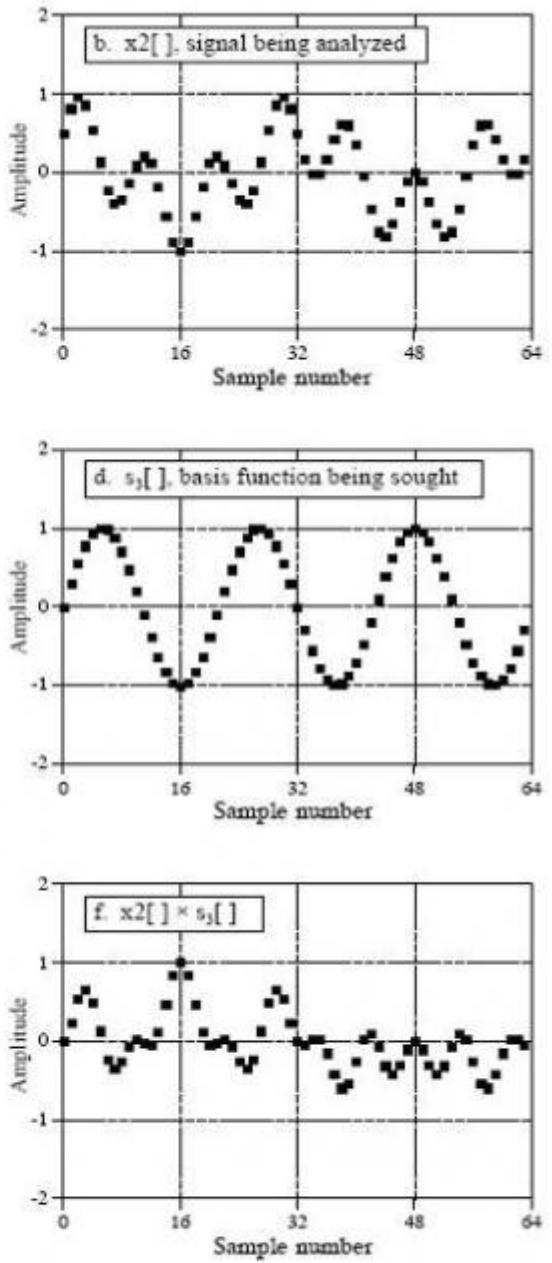


FIGURE 8-8

Two example signals, (a) and (b), are analyzed for containing the specific basis function shown in (c) and (d). Figures (e) and (f) show the result of multiplying each example signal by the basis function. Figure (e) has an average of 0.5, indicating that $x1[]$ contains the basis function with an amplitude of 1.0. Conversely, (f) has a zero average, indicating that $x2[]$ does not contain the basis function.

上面 a 和 b 两个图是待检测信号波, 图 a 很明显可以看出是个 3 个周期的正弦信号波, 图 b 的信号波则看不出是否含有正弦或余弦信号, 图 c 和 d 都是个 3 个周期的正弦信号波, 图 e 和 f 分别是 a、b 两图跟 c、d 两图相乘后的结果, 图 e 所有点的平均值是 0.5, 说明信号 a 含有振幅为 1 的正弦信号 c, 但图 f 所有点的平均值是 0, 则说明信号 b 不含有信号 d。这个就是通过信号相关性来检测是否含有某个信号的方法。

第二种方法：相应地，我也可以通过把输入信号和每一种频率的正余弦信号进行相乘（**关联操作**），从而得到原始信号与每种频率的关联程度（即总和大小），这个结果便是我们所要的傅立叶变换结果，下面两个等式便是我们所要的计算方法：

$$ReX[k] = \sum_{i=0}^{N-1} x[i] \cos(2\pi k i / N)$$

$$ImX[k] = - \sum_{i=0}^{N-1} x[i] \sin(2\pi k i / N)$$

第二个式子中加了个负号，是为了保持复数形式的一致，[前面我们知道在计算 \$\text{Im } \bar{x}\[k\]\$ 时又加了个负号](#)，所以这只是个形式的问题，并没有实际意义，你也可以把负号去掉，并在计算 $\text{Im } \bar{x}[k]$ 时也不加负号。

这里有一点必须明白一个正交的概念：两个函数相乘，如果结果中的每个点的总和为 0，则可认为这两个函数为正交函数。要确保关联性算法是正确的，则必须使得跟原始信号相乘的信号的函数形式是正交的，我们知道所有的正弦或余弦函数是正交的，这一点我们可以通过简单的高数知识就可以证明它，所以我们可以通过关联的方法把原始信号分离出正余弦信号。当然，其它的正交函数也是存在的，如：方波、三角波等形式的脉冲信号，所以原始信号也可被分解成这些信号，但这只是说可以这样做，却是没有用的。

下面是实域傅立叶变换的 **BASIC** 语言代码：

```

100 'THE DISCRETE FOURIER TRANSFORM
110 'The frequency domain signals, held in REX[ ] and IMX[ ], are calculated from
120 'the time domain signal, held in XX[ ].'
130 '
140 DIM XX[511]           'XX[ ] holds the time domain signal
150 DIM REX[256]          'REX[ ] holds the real part of the frequency domain
160 DIM IMX[256]          'IMX[ ] holds the imaginary part of the frequency domain
170 '
180 PI = 3.14159265      'Set the constant, PI
190 N% = 512              'N% is the number of points in XX[ ]
200 '
210 GOSUB XXXX           'Mythical subroutine to load data into XX[ ]
220 '
230 '
240 FOR K% = 0 TO 256    'Zero REX[ ] & IMX[ ] so they can be used as accumulators
250   REX[K%] = 0
260   IMX[K%] = 0
270 NEXT K%
280 '
290 'Correlate XX[ ] with the cosine and sine waves, Eq. 8-4
300 '
310 FOR K% = 0 TO 256    'K% loops through each sample in REX[ ] and IMX[ ]
320   FOR I% = 0 TO 511   'I% loops through each sample in XX[ ]
330   '
340   REX[K%] = REX[K%] + XX[I%] * COS(2*PI*K%*I%/N%)
350   IMX[K%] = IMX[K%] - XX[I%] * SIN(2*PI*K%*I%/N%)
360   '
370 NEXT I%
380 NEXT K%
390 '
400 END

```

到此为止，我们对傅立叶变换便有了感性的认识了吧。但要记住，这只是在实域上的离散傅立叶变换，其中虽然也用到了复数的形式，但那只是个替代的形式，并无实际意义，现实中一般使用的是复数形式的离散傅立叶变换，且**快速傅立叶变换**是根据复数离散傅立叶变换来设计算法的，在后面我们先来复习一下有关复数的内容，然后再在理解实域离散傅立叶变换的基础上来理解复数形式的离散傅立叶变换。

更多见下文：[十、从头到尾彻底理解傅里叶变换算法、下](#) (July、dznlong)

本人 **July** 对本博客所有任何文章、内容和资料享有版权。

转载务必注明作者本人及出处，并通知本人。二零一一年二月二十一日。

十、从头到尾彻底理解傅里叶变换算法、下

作者：July、dznlong 二零一一年二月二十二日

推荐阅读: ***The Scientist and Engineer's Guide to Digital Signal Processing***,
By Steven W. Smith, Ph.D。此书地址: <http://www.dspguide.com/pdfbook.htm>。

从头到尾彻底理解傅里叶变换算法、上

前言

第一部分、 DFT

第一章、傅立叶变换的由来

第二章、实数形式离散傅立叶变换 (Real DFT)

从头到尾彻底理解傅里叶变换算法、下

第三章、复数

第四章、复数形式离散傅立叶变换

前期回顾，在上一篇：**十、从头到尾彻底理解傅里叶变换算法、上**里，我们讲了傅立叶变换的由来、和实数形式离散傅立叶变换 (Real DFT) 两个问题，本文接上文，着重讲下复数、和复数形式离散傅立叶变换等两个问题。

第三章、复数

复数扩展了我们一般所能理解的数的概念，复数包含了实数和虚数两部分，利用复数的形式可以把由两个变量表示的表达式变成由一个变量(复变量)来表达，使得处理起来更加自然和方便。

我们知道傅立叶变换的结果是由两部分组成的，使用复数形式可以缩短变换表达式，使得我们可以单独处理一个变量（这个在后面的描述中我们就可以更加确切地知道），而且快速傅立叶变换正是基于复数形式的，所以几乎所有描述的傅立叶变换形式都是复数的形式。

但是复数的概念超过了我们日常生活中所能理解的概念，要理解复数是较难的，所以我们在理解复数傅立叶变换之前，先来专门复习一下有关复数的知识，这对后面的理解非常重要。

一、复数的提出

在此，先让我们看一个物理实验：把一个球从某点向上抛出，然后根据初速度和时间来计算球所在高度，这个方法可以根据下面的式子计算得出：

$$h = \frac{-gt^2}{2} + vt$$

其中 h 表示高度， g 表示重力加速度(9.8m/s²)， v 表示初速度， t 表示时间。现在反过来，假如知道了高度，要求计算到这个高度所需要的时间，这时我们又可以通过下式来计算：

$$t = \sqrt{1 - h/4.9}$$

(多谢 JERRY_PRI 提出：

1、根据公式 $h=-(gt^2/2)+vt$ (gt 后面的 2 表示 t 的平方)，我们可以讨论最终情况，也就是说小球运动到最高点时， $v=gt$ ，所以，可以得到 $t=sqrt(2h/g)$
且在您给的公式中，根号下为 $1-(2h)/g$ ，化成分数形式为 $(g-2h)/g$ ， g 和 h 不能直接做加减运算。

2、 g 是重力加速度，单位是 m/s^2 ， h 的单位是 m ，他们两个相减的话在物理上没有意义，而且使用您给的那个公式反向回去的话推出的是 $h=-(gt^2/2)+gt$ 啊 (gt 后面的 2 表示 t 的平方)。

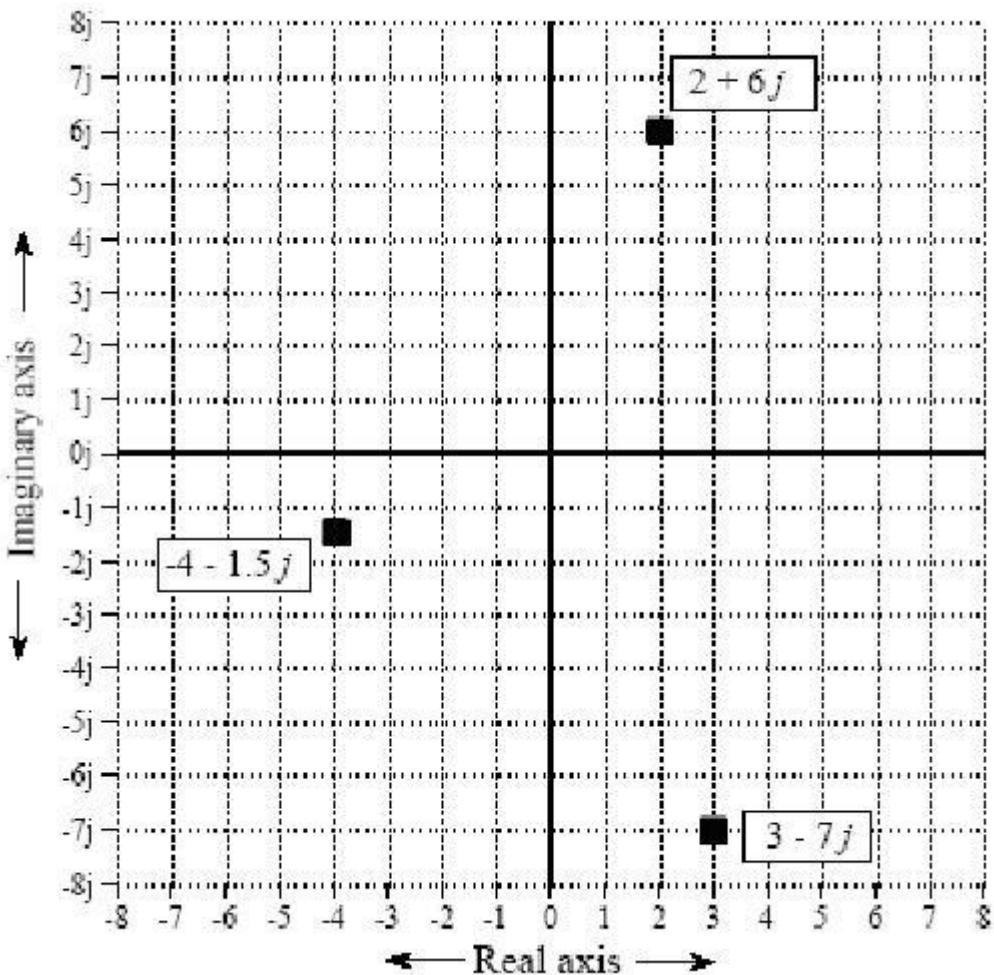
3、直接推到可以得出 $t=v/g \pm sqrt((v^2-2hg)/g^2)$ (v 和 g 后面的 2 都表示平方)，那么也就是说当 $v^2 < 2hg$ 时会产生复数，但是如果从实际的 v^2 是不可能小于 $2hg$ 的，所以我感觉复数不能从实际出发去推到，只能从抽象的角度说明一下。

)

经过计算我们可以知道，当高度是 3 米时，有两个时间点到达该高度：球向上运动时的时间是 0.38 秒，球向下运动时的时间是 1.62 秒。但是如果高度等于 10 时，结果又是什么呢？根据上面的式子可以发现存在对负数进行开平方运算，我们知道这肯定是不现实的。

第一次使用这个不一般的式子的人是意大利数学家 Girolamo Cardano (1501-1576)，两个世纪后，德国伟大数学家 Carl Friedrich Gause (1777-1855) 提出了复数的概念，为后来的应用铺平了道路，他对复数进行这样表示：复数由实数 (real) 和虚数 (imaginary) 两部分组成，虚数中的根号负 1 用 i 来表示（在这里我们用 j 来表示，因为 i 在电力学中表示电流的意思）。

我们可以把横坐标表示成实数，纵坐标表示成虚数，则坐标中的每个点的向量就可以用复数来表示，如下图：



上图中的 ABC 三个向量可以表示成如下的式子：

$$A = 2 + 6j$$

$$B = -4 - 1.5j$$

$$C = 3 - 7j$$

这样子来表达方便之处在于运用一个符号就能把两个原来难以联系起来的数组合起来了，不方便的是我们要分辨哪个是实数和哪个是虚数，我们一般是用 $\text{Re}()$ 和 $\text{Im}()$ 来表示实数和虚数两部分，如：

$$\text{Re } A = 2 \quad \text{Im } A = 6$$

$$\text{Re } B = -4 \quad \text{Im } B = -1.5$$

$$\text{Re } C = 3 \quad \text{Im } C = -7$$

复数之间也可以进行加减乘除运算：

$$(a + bj) + (c + dj) = (a + c) + j(b + d)$$

$$(a + bj) - (c + dj) = (a - c) + j(b - d)$$

$$(a + bj)(c + dj) = (ac - bd) + j(bc + ad)$$

$$\frac{(a + bj)}{(c + dj)} = \left(\frac{ac + bd}{c^2 + d^2} \right) + j \left(\frac{bc - ad}{c^2 + d^2} \right)$$

这里有个特殊的地方是 j^2 等于-1，上面第四个式子的计算方法是把分子和分母同时乘以 $c - dj$ ，这样就可消去分母中的 j 了。

复数也符合代数运算中的交换律、结合律、分配律：

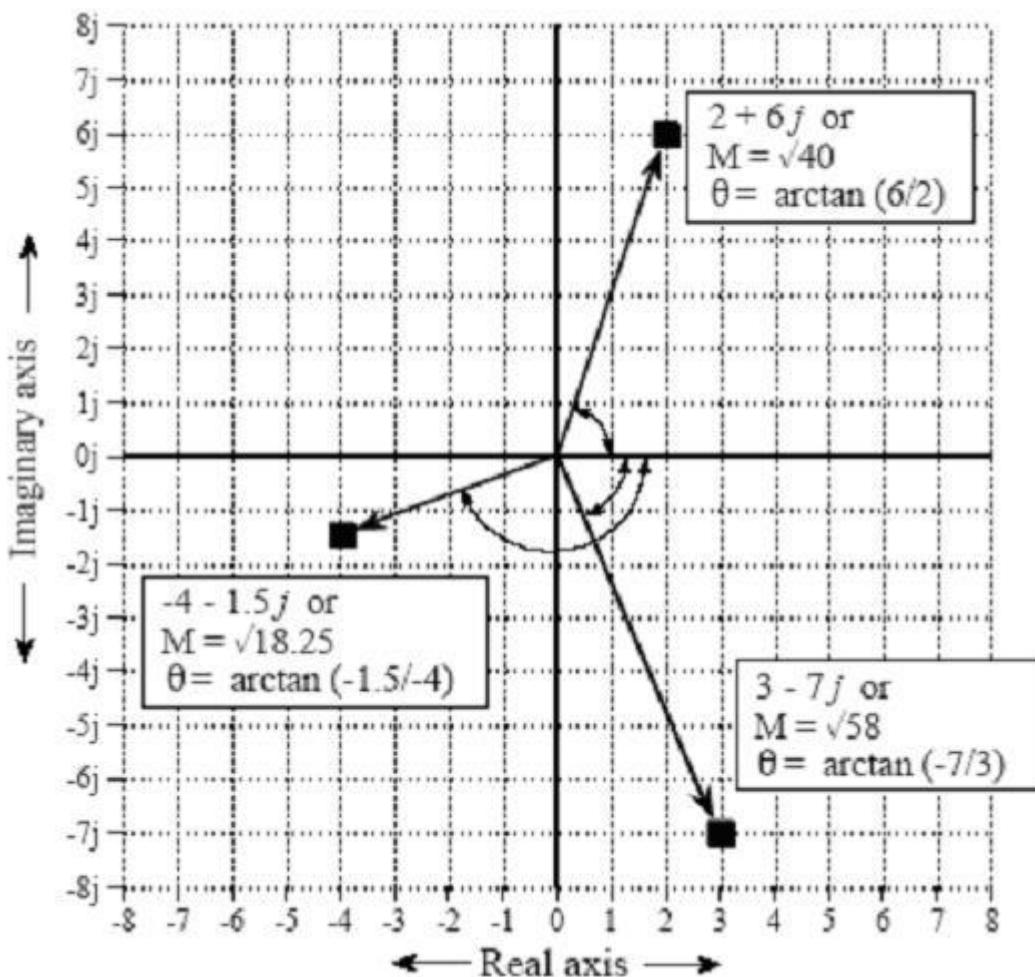
$$A B = B A$$

$$(A + B) + C = A + (B + C)$$

$$A(B + C) = AB + AC$$

二、复数的极坐标表示形式

前面提到的是运用直角坐标来表示复数，其实更为普遍应用的是极坐标的表示方法，如下图：



上图中的 M 即是数量积(magnitude)，表示从原点到坐标点的距离， θ 是相位角(phase angle)，表示从 X 轴正方向到某个向量的夹角，下面四个式子是计算方法：

$$M = \sqrt{(Re A)^2 + (Im A)^2}$$

$$\Theta = \arctan \left[\frac{Im A}{Re A} \right]$$

$$Re A = M \cos(\Theta)$$

$$Im A = M \sin(\Theta)$$

我们还可以通过下面的式子进行极坐标到直角坐标的转换：

$$\mathbf{a} + j\mathbf{b} = M (\cos\theta + j \sin\theta)$$

上面这个等式中左边是直角坐标表达式，右边是极坐标表达式。

还有一个更为重要的等式——欧拉等式（欧拉，瑞士的著名数学家，Leonhard Euler, 1707-1783）：

$$ejx = \cos x + j \sin x$$

这个等式可以从下面的级数变换中得到证明：

$$e^{jx} = \sum_{n=0}^{\infty} \frac{(jx)^n}{n!} = \left[\sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} \right] + j \left[\sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \right]$$

上面中右边的两个式子分别是 $\cos(x)$ 和 $\sin(x)$ 的泰勒(Taylor)级数。

这样子我们又可以把复数的表达式表示成指数的形式了：

$$\mathbf{a} + j\mathbf{b} = M ej\theta \quad (\text{这便是复数的两个表达式})$$

指数形式是数字信号处理中数学方法的支柱，也许是因为用指数形式进行复数的乘除运算极为简单的缘故吧：

$$M_1 e^{j\theta_1} M_2 e^{j\theta_2} = M_1 M_2 e^{j(\theta_1 + \theta_2)}$$

$$\frac{M_1 e^{j\theta_1}}{M_2 e^{j\theta_2}} = \left[\frac{M_1}{M_2} \right] e^{j(\theta_1 - \theta_2)}$$

三、复数是数学分析中的一个工具

为什么要使用复数呢？其实它只是个工具而已，就如钉子和锤子的关系，复数就象那锤子，作为一种使用的工具。我们把要解决的问题表达成复数的形式（因为有些问题用复数的形式进行运算更加方便），然后对复数进行运算，最后再转换回来得到我们所需要的结果。

有两种方法使用复数，一种是用复数进行简单的替换，如前面所说的向量表达式方法和前一节中我们所讨论的实域 DFT，另一种是更高级的方法：数学等价(**mathematical equivalence**)，复数形式的傅立叶变换用的便是数学等价的方法，但在这里我们先不讨论这种方法，这里我们先来看一下用复数进行替换中的问题。

用复数进行替换的基本思想是：把所要分析的物理问题转换成复数的形式，其中只是简单地添加一个复数的符号 j ，当返回到原来的物理问题时，则只是把符号 j 去掉就可以了。

有一点要明白的是并不是所有问题都可以用复数来表示，必须看用复数进行分析是否适用，有个例子可以看出用复数来替换原来问题的表达方式明显是谬误的：假设一箱的苹果是 5 美元，一箱的桔子是 10 美元，于是我们把它表示成 $5 + 10j$ ，有一个星期你买了 6 箱苹果和 2 箱桔子，我们又把它表示成 $6 + 2j$ ，最后计算总共花的钱是 $(5 + 10j)(6 + 2j) = 10 + 70j$ ，结果是买苹果花了 10 美元的，买桔子花了 70 美元，这样的结果明显是错了，所以复数的形式不适合运用于对这种问题的解决。

四、用复数来表示正余弦函数表达式

对于象 $M \cos(\omega t + \phi)$ 和 $A \cos(\omega t) + B \sin(\omega t)$ 表达式，用复数来表示，可以变得非常简洁，对于直角坐标形式可以按如下形式进行转换：

$$A \cos(\omega t) + B \sin(\omega t) \rightleftharpoons a + jb$$

(conventional representation) (complex number)

上式中余弦幅值 A 经变换生成 a , 正弦幅值 B 的相反数经变换生成 b : $A <=> a, B <=> -b$, 但要注意的是, 这不是个等式, 只是个替换形式而已。

对于极坐标形式可以按如下形式进行转换:

$$M \cos(\omega t + \phi) \rightleftharpoons Me^{j\theta}$$

(conventional representation) (complex number)

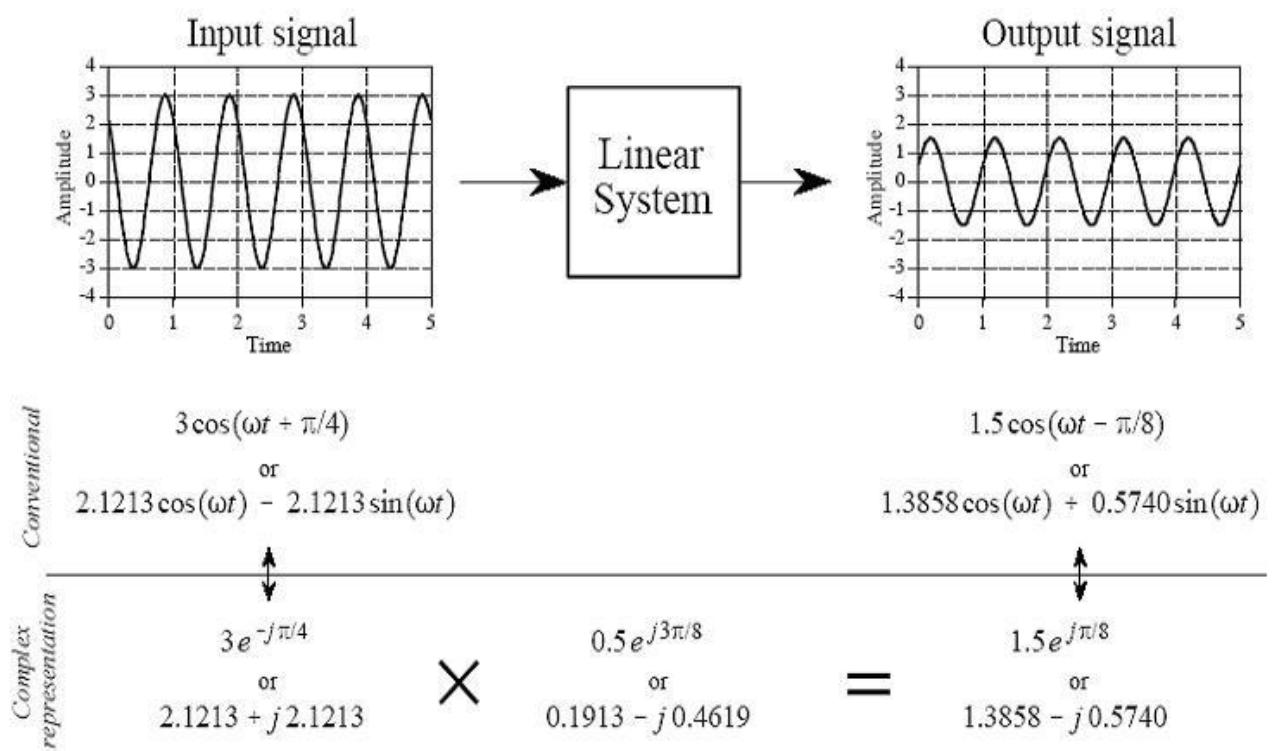
上式中, $M <=> M, \theta <=> \phi$ 。

这里虚数部分采用负数的形式主要是为了跟复数傅立叶变换表达式保持一致, 对于这种替换的方法来表示正余弦, 符号的变换没有什么好处, 但替换时总会被改变掉符号以跟更高级的等价变换保持形式上的一致。

在离散信号处理中, 运用复数形式来表示正余弦波是个常用的技术, 这是因为利用复数进行各种运算得到的结果跟原来的正余弦运算结果是一致的, 但是, 我们要小心使用复数操作, 如加、减、乘、除, 有些操作是不能用的, 如两个正弦信号相加, 采用复数形式进行相加, 得到的结果跟替换前的直接相加的结果是一样的, 但是如果两个正弦信号相乘, 则采用复数形式来相乘结果是不一样的。幸运的是, 我们已严格定义了正余弦复数形式的运算操作条件:

- 1、参加运算的所有正余弦的频率必须是一样的;
- 2、运算操作必须是线性的, 如两个正弦信号可以进行相加减, 但不能进行乘除, 象信号的放大、衰减、高低通滤波等系统都是线性的, 象平方、缩短、取限等则不是线性的。要记住的是卷积和傅立叶分析也只有线性操作才可以进行。

下图是一个相量变换(我们把正弦或余弦波变成复数的形式称为相量变换, **Phasor transform**)的例子, 一个连续信号波经过一个线性处理系统生成另一个信号波, 从计算过程我们可以看出采用复数的形式使得计算变化十分的简洁:



在第二章中我们描述的实数形式傅立叶变换也是一种替换形式的复数变换，但要注意的是那还不是复数傅立叶变换，只是一种代替方式而已。下一章、即，第四章，我们就会知道复数傅立叶变换是一种更高级的变换，而不是这种简单的替换形式。

第四章、复数形式离散傅立叶变换

复数形式的离散傅立叶变换非常巧妙地运用了复数的方法，使得傅立叶变换变换更加自然和简洁，它并不是只是简单地运用替换的方法来运用复数，而是完全从复数的角度来分析问题，这一点跟实数 DFT 是完全不一样的。

一、把正余弦函数表示成复数的形式

通过欧拉等式可以把正余弦函数表示成复数的形式：

$$\begin{aligned}\cos(x) &= 1/2 e^{j(-x)} + 1/2 e^{jx} \\ \sin(x) &= j(1/2 e^{j(-x)} - 1/2 e^{jx})\end{aligned}$$

从这个等式可以看出，如果把正余弦函数表示成复数后，它们变成了由正负频率组成的正余弦波，相反地，一个由正负频率组成的正余弦波，可以通过复数的形式来表示。

我们知道，在实数傅立叶变换中，它的频谱是 $0 \sim \pi (0 \sim N/2)$ ，但无法表示 $-\pi \sim 0$ 的频谱，可以预见，如果把正余弦表示成复数形式，则能够把负频率包含进来。

二、把变换前后的变量都看成复数的形式

复数形式傅立叶变换把原始信号 $x[n]$ 当成是一个用复数来表示的信号，其中实数部分表示原始信号值，虚数部分为 0，变换结果 $X[k]$ 也是个复数的形式，但这里的虚数部分是有值的。

在这里要用复数的观点来看原始信号，是理解复数形式傅立叶变换的关键（如果有学过复变函数则可能更好理解，即把 $x[n]$ 看成是一个复数变量，然后象对待实数那样对这个复数变量进行相同的变换）。

三、对复数进行相关性算法（正向傅立叶变换）

从实数傅立叶变换中可以知道，我们可以通过原始信号乘以一个正交函数形式的信号，然后进行求总和，最后就能得到这个原始信号所包含的正交函数信号的分量。

现在我们的原始信号变成了复数，我们要得到的当然是复数的信号分量，我们是不是可以把它乘以一个复数形式的正交函数呢？答案是肯定的，正余弦函数都是正交函数，变成如下形式的复数后，仍旧还是正交函数（这个从正交函数的定义很容易得到证明）：

$$\cos x + j \sin x, \cos x - j \sin x, \dots$$

这里我们采用上面的第二个式子进行相关性求和，为什么用第二个式子呢？，我们在后面会知道，正弦函数在虚数中变换后得到的是负的正弦函数，这里我们再加上一个负号，使得最后的得到的是正的正弦波，根据这个于是我们很容易就可以得到了复数形式的 **DFT 正向变换等式**：

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] \left(\cos(2\pi kn/N) - j \sin(2\pi kn/N) \right)$$

这个式子很容易可以得到欧拉变换式子：

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

其实我们是为了表达上的方便才用到欧拉变换式，在解决问题时我们还是较多地用到正余弦表达式。

对于上面的等式，我们要清楚如下几个方面（也是区别于实数 DFT 的地方）：

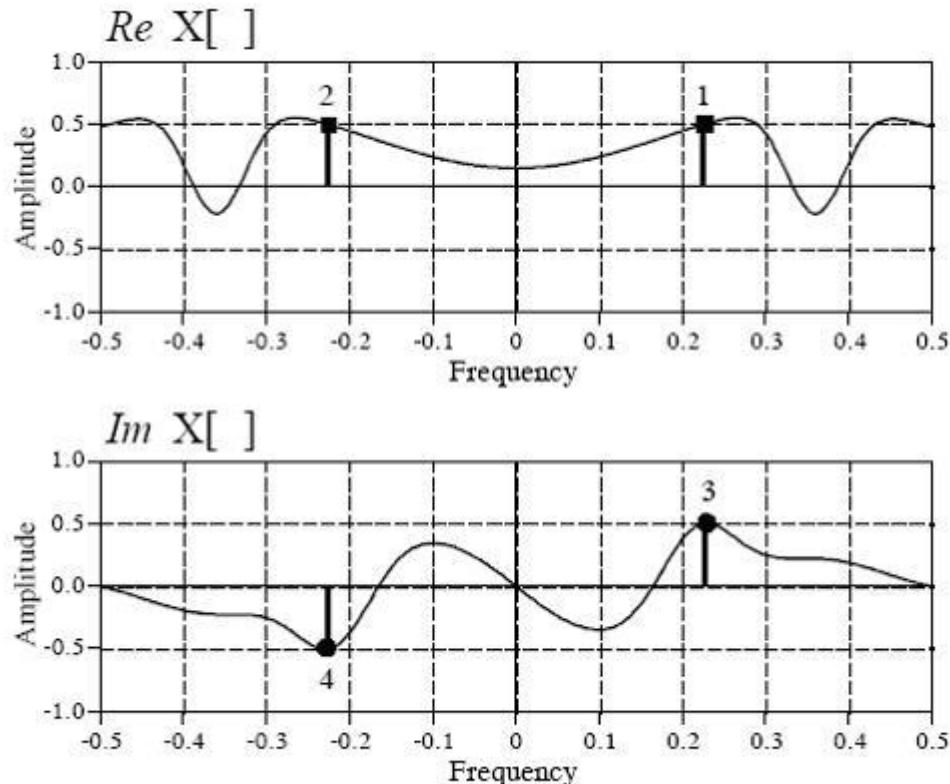
1、 $X[k]$ 、 $x[n]$ 都是复数，但 $x[n]$ 的虚数部分都是由 0 组成的，实数部分表示原始信号；

2、 k 的取值范围是 $0 \sim N-1$ (也可以表达成 $0 \sim 2\pi$)，其中 $0 \sim N/2$ (或 $0 \sim \pi$) 是正频部分，

$N/2 \sim N-1$ ($\pi \sim 2\pi$) 是负频部分，由于正余弦函数的对称性，所以我们把 $-\pi \sim 0$ 表示成 $\pi \sim 2\pi$ ，这是出于计算上方便的考虑。

3、其中的 j 是一个不可分离的组成部分，就象一个等式中的变量一样，不能随便去掉，去掉之后意义就完全不一样了，但我们知道在实数 DFT 中， j 只是个符号而已，把 j 去掉，整个等式的意义不变；

4、下图是个连续信号的频谱，但离散频谱也是与此类似的，所以不影响我们对问题的分析：



上面的频谱图把负频率放到了左边，是为了迎合我们的思维习惯，但在实际实现中我们一般是把它移到正的频谱后面的。

从上图可以看出，时域中的正余弦波（用来组成原始信号的正余弦波）在复数 DFT 的频谱中被分成了正、负频率的两个组成部分，基于此等式中前面的比例系数是 $1/N$ (或 $1/2\pi$)，而不是 $2/N$ ，这是因为现在把频谱延伸到了 2π ，但把正负两个频率相加即又得到了 $2/N$ ，又还原到了实数 DFT 的形式，这个在后面的描述中可以更清楚地看到。

由于复数 DFT 生成的是一个完整的频谱，原始信号中的每一个点都是由正、负两个频率组合而成的，所以频谱中每一个点的带宽是一样的，都是 $1/N$ ，相对实数 DFT，两端带宽比其它点的带宽少了一半；复数 DFT 的频谱特征具有周期性： $-N/2 \sim 0$ 与 $N/2 \sim N-1$

是一样的，实域频谱呈偶对称性（表示余弦波频谱），虚域频谱呈奇对称性（表示正弦波频谱）。

四、逆向傅立叶变换

假设我们已经得到了复数形式的频谱 $X[k]$ ，现在要把它还原到复数形式的原始信号 $x[n]$ ，当然应该是把 $X[k]$ 乘以一个复数，然后再进行求和，最后得到原始信号 $x[n]$ ，这个跟 $X[k]$ 相乘的复数首先让我们想到的应该是上面进行相关性计算的复数：

$$\cos(2\pi kn/N) - j \sin(2\pi kn/N),$$

但其中的负号其实是为了使得进行逆向傅立叶变换时把正弦函数变为正的符号，因为虚数 j 的运算特殊性，使得原来应该是正的正弦函数变为了负的正弦函数（我们从后面的推导会看到这一点），所以这里的负号只是为了纠正符号的作用，在进行逆向 DFT 时，我们可以把负号去掉，于是我们便得到了这样的逆向 DFT 变换等式：

$$x[n] = X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N))$$

我们现在来分析这个式子，会发现这个式其实跟实数傅立叶变换是可以得到一样结果的。我们先把 $X[k]$ 变换一下：

$$X[k] = \operatorname{Re} X[k] + j \operatorname{Im} X[k]$$

这样我们就可以对 $x[n]$ 再次进行变换，如：

$$\begin{aligned} x[n] &= (\operatorname{Re} X[k] + j \operatorname{Im} X[k]) (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) \\ &= (\operatorname{Re} X[k] \cos(2\pi kn/N) + j \operatorname{Im} X[k] \cos(2\pi kn/N) + j \operatorname{Re} X[k] \\ &\quad \sin(2\pi kn/N) - \operatorname{Im} X[k] \sin(2\pi kn/N)) \\ &= (\operatorname{Re} X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) \\ &+ \text{-----} (1) \\ &\quad \operatorname{Im} X[k] (\text{-----} - \sin(2\pi kn/N) + j \\ &\quad \cos(2\pi kn/N))) \text{-----} (2) \end{aligned}$$

这时我们就把原来的等式分成了两个部分，第一个部分是跟实域中的频谱相乘，第二个部分是跟虚域中的频谱相乘，根据频谱图我们可以知道， $\operatorname{Re} X[k]$ 是个偶对称的变量， $\operatorname{Im} X[k]$ 是个奇对称的变量，即

$$\begin{aligned} \operatorname{Re} X[k] &= \operatorname{Re} X[-k] \\ \operatorname{Im} X[k] &= -\operatorname{Im} X[-k] \end{aligned}$$

但 k 的范围是 $0 \sim N-1$, $0 \sim N/2$ 表示正频率, $N/2 \sim N-1$ 表示负频率, 为了表达方便我们把 $N/2 \sim N-1$ 用 $-k$ 来表示, 这样在从 0 到 $N-1$ 的求和过程中对于(1)和(2)式分别有 $N/2$ 对的 k 和 $-k$ 的和, 对于 (1) 式有:

$$\operatorname{Re} X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \operatorname{Re} X[-k] (\cos(-2\pi kn/N) + j \sin(-2\pi kn/N))$$

根据偶对称性和三角函数的性质, 把上式化简得到:

$$\operatorname{Re} X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \operatorname{Re} X[-k] (\cos(2\pi kn/N) - j \sin(2\pi kn/N))$$

这个式子最后的结果是:

$$2 \operatorname{Re} X[k] \cos(2\pi kn/N)。$$

再考虑到求 $\operatorname{Re} X[k]$ 等式中有个比例系数 $1/N$, 把 $1/N$ 乘以 2, 这样的结果不就是跟实数 DFT 中的式子一样了吗?

对于(2)式, 用同样的方法, 我们也可以得到这样的结果:

$$-2 \operatorname{Im} X[k] \sin(2\pi kn/N)$$

注意上式前面多了个负符号, 这是由于虚数变换的特殊性造成的, 当然我们肯定不能把负符号的正弦函数跟余弦来相加, 还好, 我们前面是用 $\cos(2\pi kn/N) - j \sin(2\pi kn/N)$ 进行相关性计算, 得到的 $\operatorname{Im} X[k]$ 中有个负的符号, 这样最后的结果中正弦函数就没有负的符号了, 这就是为什么在进行相关性计算时虚数部分要用到负符号的原因 (我觉得这也许是复数形式 DFT 美中不足的地方, 让人有一种拼凑的感觉)。

从上面的分析中可以看出, 实数傅立叶变换跟复数傅立叶变换, 在进行逆变换时得到的结果是一样的, 只不过是殊途同归吧。本文完。 (July、dznlong)

本人 **July** 对本博客所有任何文章、内容和资料享有版权。

转载务必注明作者本人及出处, 并通知本人。二零一一年二月二十二日。

十一、从头到尾彻底解析 Hash 表算法

作者: July、wuliming、pkuoliver

出处: http://blog.csdn.net/v_JULY_v。

说明: 本文分为三部分内容,

第一部分为一道百度面试题 Top K 算法的详解; 第二部分为关于 Hash 表算法的详细

阐述；第三部分为打造一个最快的 Hash 表算法。

第一部分：Top K 算法详解

问题描述

百度面试题：

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

必备知识：

什么是哈希表？

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

哈希表的做法其实很简单，就是把 Key 通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将 value 存储在以该数字为下标的数组空间里。

而当使用哈希表进行查询的时候，就是再次使用哈希函数将 key 转换为对应的数组下标，并定位到该空间获取 value，如此一来，就可以充分利用到数组的定位性能进行数据定位（文章第二、三部分，会针对 Hash 表详细阐述）。

问题解析：

要统计最热门查询，首先就是要统计每个 Query 出现的次数，然后根据统计结果，找出 Top 10。所以我们可以基于这个思路分两步来设计该算法。

即，此问题的解决分为以下两个步骤：

第一步：Query 统计

Query 统计有以下两个方法，可供选择：

1、直接排序法

首先我们最先想到的的算法就是排序了，首先对这个日志里面的所有 Query 都进行排序，然后再遍历排好序的 Query，统计每个 Query 出现的次数了。

但是题目中有明确要求，那就是内存不能超过 1G，一千万条记录，每条记录是 225Byte，很显然要占据 2.55G 内存，这个条件就不满足要求了。

让我们回忆一下数据结构课程上的内容，当数据量比较大而且内存无法装下的时候，我们可以采用外排序的方法来进行排序，这里我们可以采用归并排序，因为归并排序有一个比较好的时间复杂度 $O(N \lg N)$ 。

排完序之后我们再对已经有序的 **Query** 文件进行遍历，统计每个 **Query** 出现的次数，再次写入文件中。

综合分析一下，排序的时间复杂度是 $O(N \lg N)$ ，而遍历的时间复杂度是 $O(N)$ ，因此该算法的总体时间复杂度就是 $O(N + N \lg N) = O(N \lg N)$ 。

2、Hash Table 法

在第 1 个方法中，我们采用了排序的办法来统计每个 **Query** 出现的次数，时间复杂度是 $N \lg N$ ，那么能不能有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，虽然有一千万个 **Query**，但是由于重复度比较高，因此事实上只有 300 万的 **Query**，每个 **Query** 255Byte，因此我们可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，**Hash Table** 绝对是我们优先的选择，因为 **Hash Table** 的查询速度非常的快，几乎是 $O(1)$ 的时间复杂度。

那么，我们的算法就有了：维护一个 **Key** 为 **Query** 字串，**Value** 为该 **Query** 出现次数的 **HashTable**，每次读取一个 **Query**，如果该字串不在 **Table** 中，那么加入该字串，并且将 **Value** 值设为 1；如果该字串在 **Table** 中，那么将该字串的计数加一即可。最终我们在 $O(N)$ 的时间复杂度内完成了对该海量数据的处理。

本方法相比算法 1：在时间复杂度上提高了一个数量级，为 $O(N)$ ，但不仅仅是时间复杂度上的优化，该方法只需要 IO 数据文件一次，而算法 1 的 IO 次数较多的，因此该算法 2 比算法 1 在工程上有更好的可操作性。

第二步：找出 Top 10

算法一：普通排序

我想对于排序算法大家都已经不陌生了，这里不在赘述，我们要注意的是排序算法的时间复杂度是 $N \lg N$ ，在本题目中，三百万条记录，用 1G 内存是可以存下的。

算法二：部分排序

题目要求是求出 **Top 10**，因此我们没有必要对所有的 **Query** 都进行排序，我们只需要维护一个 10 个大小的数组，初始化放入 10 个 **Query**，按照每个 **Query** 的统计次数由大到小排序，然后遍历这 300 万条记录，每读一条记录就和数组最后一个 **Query** 对比，如果小于这个 **Query**，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的 **Query**。最后当所有的数据都遍历完毕之后，那么这个数组中的 10 个 **Query** 便是我们要找的 **Top 10** 了。

不难分析出，这样，算法的最坏时间复杂度是 $N * K$ ，其中 **K** 是指 top 多少。

算法三：堆

在算法二中，我们已经将时间复杂度由 $N \log N$ 优化到 NK ，不得不说这是一个比较大的改进了，可是有没有更好的办法呢？

分析一下，在算法二中，每次比较完成之后，需要的操作复杂度都是 K ，因为要把元素插入到一个线性表之中，而且采用的是顺序比较。这里我们注意一下，该数组是有序的，一次我们每次查找的时候可以采用二分的方法查找，这样操作的复杂度就降到了 $\log K$ ，可是，随之而来的问题就是数据移动，因为移动数据次数增多了。不过，这个算法还是比算法二有了改进。

基于以上的分析，我们想想，有没有一种既能快速查找，又能快速移动元素的数据结构呢？回答是肯定的，那就是堆。

借助堆结构，我们可以在 \log 量级的时间内查找和调整/移动。因此到这里，我们的算法可以改进为这样，维护一个 K （该题目中是 10）大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比。

思想与上述算法二一致，只是算法在算法三，我们采用了最小堆这种数据结构代替数组，把查找目标元素的时间复杂度有 $O(K)$ 降到了 $O(\log K)$ 。

那么这样，采用堆数据结构，算法三，最终的时间复杂度就降到了 $N' \log K$ ，和算法二相比，又有了比较大的改进。

总结：

至此，算法就完全结束了，经过上述第一步、先用 Hash 表统计每个 Query 出现的次数， $O(N)$ ；然后第二步、采用堆数据结构找出 Top 10， $N * O(\log K)$ 。所以，我们最终的时间复杂度是： $O(N) + N' * O(\log K)$ 。（ N 为 1000 万， N' 为 300 万）。如果各位有什么更好的算法，欢迎留言评论。第一部分，完。

第二部分、Hash 表 算法的详细解析

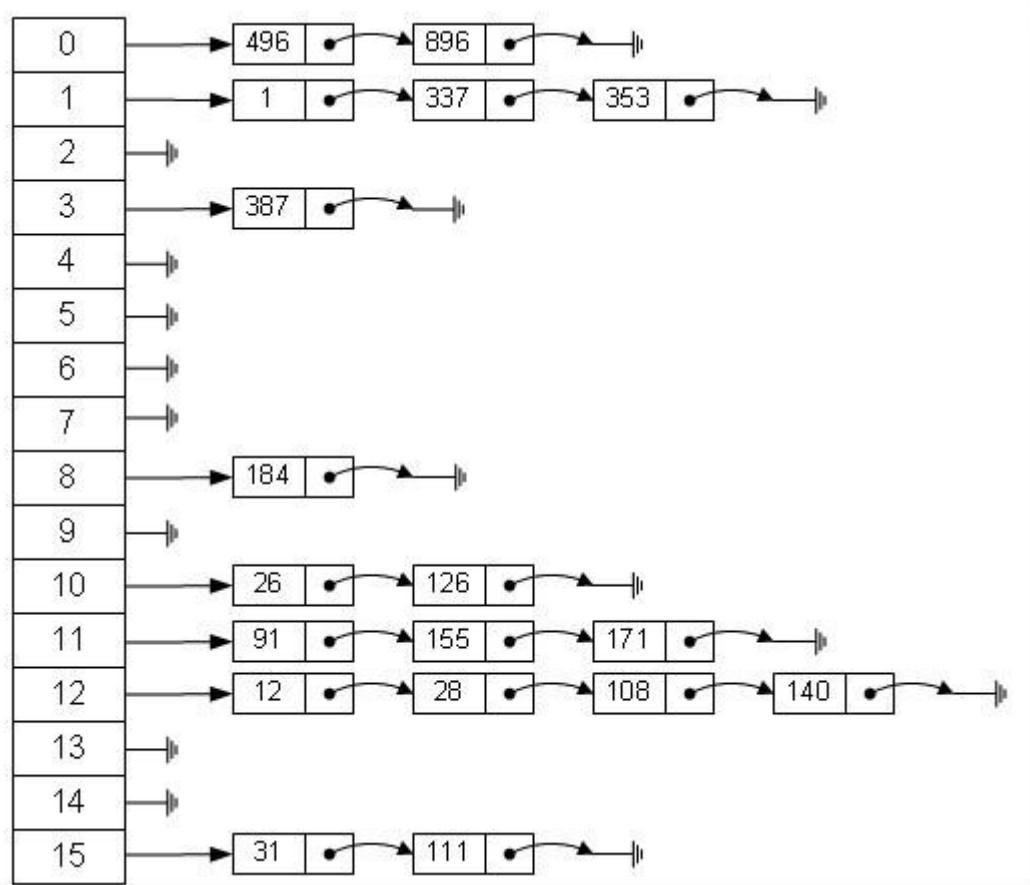
什么是 Hash

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射， *pre-image*），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

HASH 主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的 128 位的编码，这些编码值叫做 HASH 值。也可以说，hash 就是找到一种数据内容和数据存放地址之间的映射关系。

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？

答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，我们可以理解为“链表的数组”，如图：



左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。

元素特征转变为数组下标的方法就是散列法。散列法当然不止一种，下面列出三种比较常用的：

1. 除法散列法

最直观的一种，上图使用的就是这种散列法，公式：

$$\text{index} = \text{value \% 16}$$

学过汇编的都知道，求模数其实是通过一个除法运算得到的，所以叫“除法散列法”。

2. 平方散列法

求 `index` 是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的 CPU 来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：

`index = (value * value) >> 28` （右移，除以 2^{28} 。记法：左移变大，是乘。右移变小，是除。）

如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的 `index` 都是 0——非常失败。也许你还有个问题，`value` 如果很大，`value`

* value 不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取 index。

3. 斐波那契 (Fibonacci) 散列法

平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿 value 本身当作乘数呢？答案是肯定的。

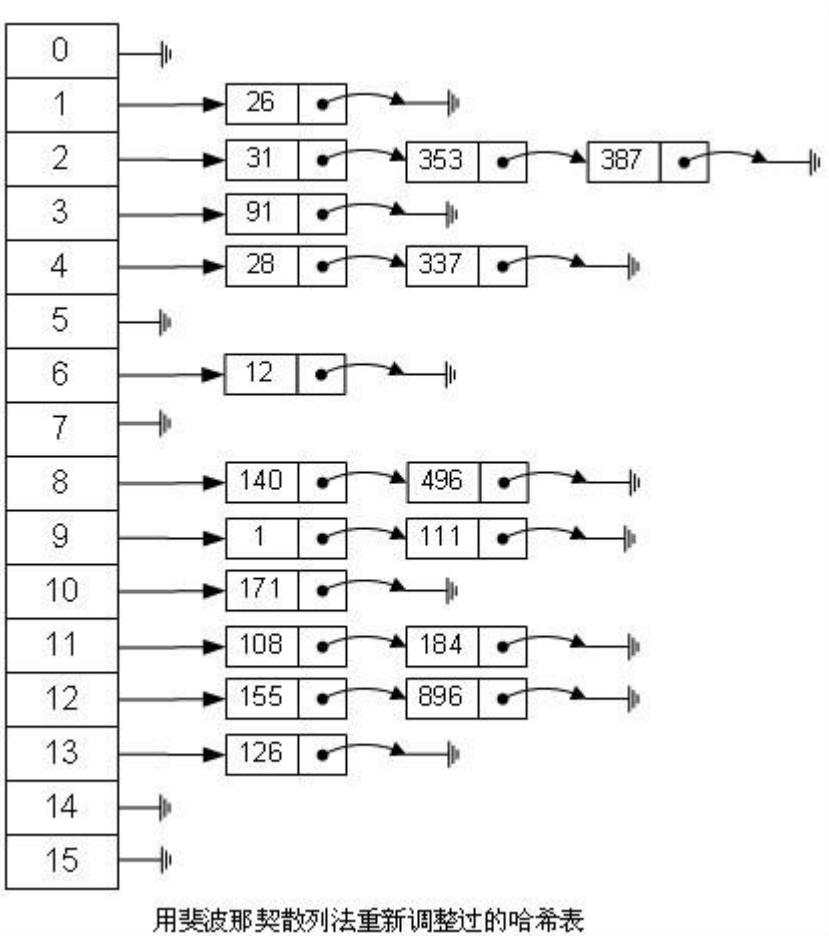
- 1, 对于 16 位整数而言，这个乘数是 40503
- 2, 对于 32 位整数而言，这个乘数是 2654435769
- 3, 对于 64 位整数而言，这个乘数是 11400714819323198485

这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，即如此形式的序列：
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...。另外，斐波那契数列的值和太阳系八大行星的轨道半径的比例出奇吻合。

对我们常见的 32 位整数而言，公式：

```
index = (value * 2654435769) >> 28
```

如果用这种斐波那契散列法的话，那上面的图就变成这样了：



很明显，用斐波那契散列法调整之后要比原来的取摸散列法好很多。

适用范围

快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。

基本原理及要点

hash 函数选择，针对字符串，整数，排列，具体相应的 hash 方法。

碰撞处理，一种是 open hashing，也称为拉链法；另一种就是 closed hashing，也称开地址法，opened addressing。

扩展

d-left hashing 中的 d 是多个的意思，我们先简化这个问题，看一看 2-left hashing。
2-left hashing 指的是将一个哈希表分成长度相等的两半，分别叫做 T1 和 T2，给 T1 和 T2 分别配备一个哈希函数， h_1 和 h_2 。在存储一个新的 key 时，同时用两个哈希函数进行计算，得出两个地址 $h_1[key]$ 和 $h_2[key]$ 。这时需要检查 T1 中的 $h_1[key]$ 位置和 T2 中的 $h_2[key]$ 位置，哪一个位置已经存储的（有碰撞的）key 比较多，然后将新 key 存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个 key，就把新

key 存储在左边的 T1 子表中, **2-left** 也由此而来。在查找一个 **key** 时, 必须进行两次 **hash**, 同时查找两个位置。

问题实例（海量数据处理）

我们知道 **hash** 表在海量数据处理中有着广泛的应用, 下面, 请看另一道百度面试题: 题目: 海量日志数据, 提取出某日访问百度次数最多的那个 **IP**。

方案: **IP** 的数目还是有限的, 最多 2^{32} 个, 所以可以考虑使用 **hash** 将 **ip** 直接存入内存, 然后进行统计。

第三部分、最快的 Hash 表算法

接下来, 咱们来具体分析一下一个最快的 **Hasb** 表算法。

我们由一个简单的问题逐步入手: 有一个庞大的字符串数组, 然后给你一个单独的字符串, 让你从这个数组中查找是否有这个字符串并找到它, 你会怎么做? 有一个方法最简单, 老老实实从头查到尾, 一个一个比较, 直到找到为止, 我想只要学过程序设计的人都能把这样一个程序作出来, 但要是有程序员把这样的程序交给用户, 我只能用无语来评价, 或许它真的能工作, 但...也只能如此了。

最合适的算法自然是使用 **HashTable** (哈希表), 先介绍介绍其中的基本知识, 所谓 **Hash**, 一般是一个整数, 通过某种算法, 可以把一个字符串"压缩"成一个整数。当然, 无论如何, 一个 32 位整数是无法对应回一个字符串的, 但在程序中, 两个字符串计算出的 **Hash** 值相等的可能非常小, 下面看看在 **MPQ** 中的 **Hash** 算法:

函数一、以下的函数生成一个长度为 0x500(合 10 进制数: 1280)的 **cryptTable[0x500]**

```
void prepareCryptTable()
{
    unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;

    for( index1 = 0; index1 < 0x100; index1++ )
    {
        for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
        {
            unsigned long temp1, temp2;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp1 = (seed & 0xFFFF) << 0x10;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp2 = (seed & 0xFFFF);

            cryptTable[index2] = ( temp1 | temp2 );
        }
    }
}
```

```
    }
}
}
```

函数二、以下函数计算 `IpszFileName` 字符串的 `hash` 值，其中 `dwHashType` 为 `hash` 的类型，在下面的**函数三**、`GetHashTablePos` 函数中调用此函数二，其可以取的值为 0、1、2；该函数返回 `IpszFileName` 字符串的 `hash` 值：

```
unsigned long HashString( char *IpszFileName, unsigned long dwHashType )
{
    unsigned char *key = (unsigned char *)IpszFileName;
    unsigned long seed1 = 0x7FED7FED;
    unsigned long seed2 = 0xFFFFFFFF;
    int ch;

    while( *key != 0 )
    {
        ch = toupper(*key++);

        seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
    }
    return seed1;
}
```

Blizzard 的这个算法是非常高效的，被称为"One-Way Hash"(A one-way hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串 "unitneutralacritter.grp" 通过这个算法得到的结果是 0xA26067F3。

是不是把第一个算法改进一下，改成逐个比较字符串的 `Hash` 值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个哈希表(`Hash Table`)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，例如 1024，每一个 `Hash` 值通过取模运算 (`mod`) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置有没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的 $O(1)$ ，现在仔细看看这个算法吧：

```
typedef struct
{
    int nHashA;
    int nHashB;
    char bExists;
    ....
```

```
 } SOMESTRUCTURE;
```

一种可能的结构体定义？

函数三、下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则，return -1.

```
int GetHashTablePos( har *IpszString, SOMESTRUCTURE *IpTable )
//IpszString 要在 Hash 表中查找的字符串，IpTable 为存储字符串 Hash 值的 Hash 表。
{
    int nHash = HashString(IpszString); //调用上述函数二，返回要查找字符串
    IpszString 的 Hash 值。
    int nHashPos = nHash % nTableSize;

    if ( IpTable[nHashPos].bExists && !strcmp( IpTable[nHashPos].pString,
    IpszString ) )
        { //如果找到的 Hash 值在表中存在，且要查找的字符串与表中对应位置的字符串相
        同，
            return nHashPos; //则返回上述调用函数二后，找到的 Hash 值
        }
    else
    {
        return -1;
    }
}
```

看到此，我想大家都在想一个很严重的问题：“如果两个字符串在哈希表中对应的位置相同怎么办？”，毕竟一个数组容量是有限的，这种可能性很大。解决该问题的方法很多，我首先想到的就是用“链表”，感谢大学里学的数据结构教会了这个百试百灵的法宝，我遇到的很多算法都可以转化成链表来解决，只要在哈希表的每个入口挂一个链表，保存所有对应的字符串就 OK 了。事情到此似乎有了完美的结局，如果是把问题独自交给我解决，此时我可能就要开始定义数据结构然后写代码了。

然而 Blizzard 的程序员使用的方法则是更精妙的方法。基本原理就是：他们在哈希表中不是用一个哈希值而是用三个哈希值来校验字符串。

MPQ 使用文件名哈希表来跟踪内部的所有文件。但是这个表的格式与正常的哈希表有一些不同。首先，它没有使用哈希作为下标，把实际的文件名存储在表中用于验证，实际上它根本就没有存储文件名。而是使用了 3 种不同的哈希：一个用于哈希表的下标，两个用于验证。这两个验证哈希替代了实际文件名。

当然了，这样仍然会出现 2 个不同的文件名哈希到 3 个同样的哈希。但是这种情况发生的概率平均是：1:18889465931478580854784，这个概率对于任何人来说应该都是足够小的。现在再回到数据结构上，Blizzard 使用的哈希表没有使用链表，而采用“顺延”的方式来解决问题，看看这个算法：

函数四、`IpszString` 为要在 `hash` 表中查找的字符串；`IpTable` 为存储字符串 `hash` 值的 `hash` 表；`nTableSize` 为 `hash` 表的长度：

```
int GetHashTablePos( char *IpszString, MPQHASHTABLE *IpTable, int nTableSize )
{
    const int HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;

    int nHash = HashString( IpszString, HASH_OFFSET );
    int nHashA = HashString( IpszString, HASH_A );
    int nHashB = HashString( IpszString, HASH_B );
    int nHashStart = nHash % nTableSize;
    int nHashPos = nHashStart;

    while ( IpTable[nHashPos].bExists )
    {
        /*如果仅仅是判断在该表中时候存在这个字符串，就比较这两个 hash 值就可以了，不用对
         *结构体中的字符串进行比较。这样会加快运行的速度？减少 hash 表占用的空间？这种
         *方法一般应用在什么场合？*/
        if ( IpTable[nHashPos].nHashA == nHashA
            && IpTable[nHashPos].nHashB == nHashB )
        {
            return nHashPos;
        }
        else
        {
            nHashPos = (nHashPos + 1) % nTableSize;
        }

        if (nHashPos == nHashStart)
            break;
    }
    return -1;
}
```

上述程序解释：

1. 计算出字符串的三个哈希值（一个用来确定位置，另外两个用来校验）
2. 察看哈希表中的这个位置
3. 哈希表中这个位置为空吗？如果为空，则肯定该字符串不存在，返回-1。
4. 如果存在，则检查其他两个哈希值是否也匹配，如果匹配，则表示找到了该字符串，返回其 Hash 值。
5. 移到下一个位置，如果已经移到了表的末尾，则反绕到表的开始位置起继续查询

6. 看看是不是又回到了原来的位置，如果是，则返回没找到
7. 回到 3

ok，这就是本文中所说的最快的 Hash 表算法。什么？不够快？:D。欢迎，各位批评指正。

补充 1、一个简单的 hash 函数：

```
/*key 为一个字符串， nTableLength 为哈希表的长度
*该函数得到的 hash 值分布比较均匀*/
unsigned long getHashIndex( const char *key, int nTableLength )
{
    unsigned long nHash = 0;

    while (*key)
    {
        nHash = (nHash<<5) + nHash + *key++;
    }

    return ( nHash % nTableLength );
}
```

补充 2、一个完整测试程序：

哈希表的数组是定长的，如果太大，则浪费，如果太小，体现不出效率。合适的数组大小是哈希表的性能的关键。哈希表的尺寸最好是一个质数。当然，根据不同的数据量，会有不同的哈希表的大小。对于数据量时多时少的应用，最好的设计是使用动态可变尺寸的哈希表，那么如果你发现哈希表尺寸太小了，比如其中的元素是哈希表尺寸的 2 倍时，我们就需要扩大哈希表尺寸，一般是扩大一倍。

下面是哈希表尺寸大小的可能取值：

```
17,      37,      79,      163,      331,
673,      1361,     2729,     5471,     10949,
21911,     43853,    87719,    175447,    350899,
701819,    1403641,   2807303,   5614657,   11229331,
22458671,   44917381,   89834777,   179669557,  359339171,
718678369,  1437356741, 2147483647
```

以下为该程序的完整源码，已在 linux 下测试通过：

[view plain](#) [copy to clipboard](#) [print?](#)

```
1. #include <stdio.h>
2. #include <ctype.h>      //多谢 citylove 指正。
3. //cryptTable[]里面保存的是 HashString 函数里面将会用到的一些数据，在
   prepareCryptTable
4. //函数里面初始化
5. unsigned long cryptTable[0x500];
6.
7. //以下的函数生成一个长度为 0x500 (含 10 进制数: 1280) 的 cryptTable[0x500]
8. void prepareCryptTable()
9. {
10.     unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
11.
12.     for( index1 = 0; index1 < 0x100; index1++ )
13.     {
14.         for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
15.         {
16.             unsigned long temp1, temp2;
17.
18.             seed = (seed * 125 + 3) % 0x2AAAAB;
19.             temp1 = (seed & 0xFFFF) << 0x10;
20.
21.             seed = (seed * 125 + 3) % 0x2AAAAB;
22.             temp2 = (seed & 0xFFFF);
23.
24.             cryptTable[index2] = ( temp1 | temp2 );
25.         }
26.     }
27. }
28.
29. //以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，
30. //在下面 GetHashTablePos 函数里面调用本函数，其可以取的值为 0、1、2；该函数
31. //返回 lpszFileName 字符串的 hash 值；
32. unsigned long HashString( char *lpszFileName, unsigned long dwHashType )
33. {
34.     unsigned char *key = (unsigned char *)lpszFileName;
35.     unsigned long seed1 = 0x7FED7FED;
36.     unsigned long seed2 = 0xFFFFFFFF;
37.     int ch;
38.
39.     while( *key != 0 )
40.     {
```

```
41.         ch = toupper(*key++);
42.
43.         seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
44.         seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
45.     }
46.     return seed1;
47. }
48.
49. //在 main 中测试 argv[1]的三个 hash 值:
50. //./hash "arr\units.dat"
51. //./hash "unit\neutral\acritter.grp"
52. int main( int argc, char **argv )
53. {
54.     unsigned long ulHashValue;
55.     int i = 0;
56.
57.     if ( argc != 2 )
58.     {
59.         printf("please input two arguments\n");
60.         return -1;
61.     }
62.
63.     /*初始化数组: cryptTable[0x500]*/
64.     prepareCryptTable();
65.
66.     /*打印数组 cryptTable[0x500]里面的值*/
67.     for ( ; i < 0x500; i++ )
68.     {
69.         if ( i % 10 == 0 )
70.         {
71.             printf("\n");
72.         }
73.
74.         printf("%-12X", cryptTable[i] );
75.     }
76.
77.     ulHashValue = HashString( argv[1], 0 );
78.     printf("\n----%X ----\n", ulHashValue );
79.
80.     ulHashValue = HashString( argv[1], 1 );
81.     printf("----%X ----\n", ulHashValue );
82.
83.     ulHashValue = HashString( argv[1], 2 );
84.     printf("----%X ----\n", ulHashValue );
```

```
85.  
86.     return 0;  
87. }
```

致谢：

- 1、<http://blog.redfox66.com/>。
- 2、http://blog.csdn.net/wuliming_sc/。完。

版权所有，法律保护。转载，请以链接形式，注明出处。

十一、从头到尾彻底解析 Hash 表算法

作者：July、yansha。编程艺术室出品。

出处：结构之法算法之道。

前言

本文阐述两个问题，第二十三章是杨氏矩阵查找问题，第二十四章是有关倒排索引中关键词 Hash 编码的问题，主要要解决不重复以及追加的功能，同时也是经典算法研究系列十一、从头到尾彻底解析 Hash 表算法之续。

OK，有任何问题，也欢迎随时交流或批评指正。谢谢。

第二十三章、杨氏矩阵查找 杨氏矩阵查找

先看一个来自算法导论习题里 6-3 与剑指 offer 的一道编程题（也被经常用作面试题，本人此前去搜狗二面时便遇到了）：

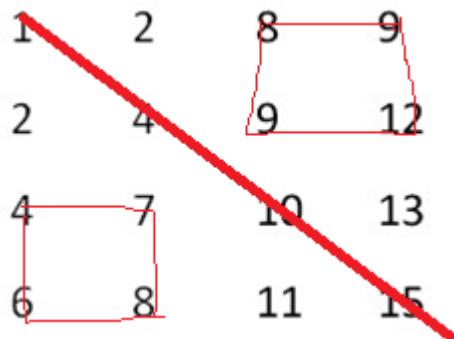
在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字 6，则返回 `true`；如果查找数字 5，由于数组不含有该数字，则返回 `false`。

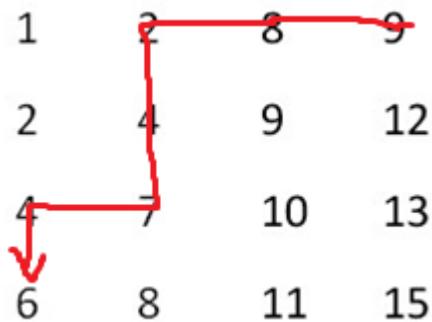
1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

本 Young 问题解法有二（如查找数字 6）：

1、分治法，分为四个矩形，配以二分查找，如果要找的数是 6 介于对角线上相邻的两个数 4、10，可以排除掉左上和右下的两个矩形，而递归在左下和右上的两个矩形继续找，如下图所示：



2、首先直接定位到最右上角的元素，再配以二分查找，比要找的数（6）大就往左走，比要找数（6）的小就往下走，直到找到要找的数字（6）为止，如下图所示：



上述方法二的关键代码+程序运行如下图所示：

```

1:存在
2:存在
3:不存在
4:存在
5:不存在
6:存在
7:存在
8:存在
9:存在
10:存在
11:存在
12:存在
13:存在
14:不存在
15:存在
请按任意键继续. . .

```

```

#include <iostream>
#define ROW 4
#define COL 4

bool Young(int array[][COL], int search)
{
    int i = 0, j = COL-1;
    int var = array[i][j];
    while (true) {
        if (var == search)
            return true;
        else if (var < search && i < ROW - 1)
            var = array[++i][j];
        else if (var > search && j > 0)
            var = array[i][--j];
        else
            return false;
    }
}

int main()
{
    int array[ROW][COL] = {{1,2,8,9},{2,4,9,12},{4,7,10,13},{6,8,11,15}};

    for (int search = 1; search <= 15; search++) {
        cout << search;
        if (Young(array, search))
            cout << ":存在";
        else
            cout << ":不存在";
    }
}

```

试问，上述算法复杂么？不复杂，只要稍微动点脑筋便能想到，还可以参看友人老梦的文章，Young 氏矩阵：<http://blog.csdn.net/zhanglei8893/article/details/6234564>，以及

IT 练兵场的：<http://www.jobcoding.com/array/matrix/young-tableau-problem/>，除此之外，何海涛先生一书剑指 offer 中也收集了此题，感兴趣的朋友也可以去看看。

十一（续）、倒排索引关键词 Hash 不重复编码实践

本章要介绍这样一个问题，对倒排索引中的关键词进行编码。那么，这个问题将分为两个个步骤：

1. 首先，要提取倒排索引内词典文件中的关键词；
2. 对提取出来的关键词进行编码。本章采取 hash 编码的方式。既然要用 hash 编码，那么最重要的就是要解决 hash 冲突的问题，下文会详细介绍。

有一点必须提醒读者的是，**倒排索引包含词典和倒排记录表两个部分**，词典一般有词项（或称为关键词）和词项频率（即这个词项或关键词出现的次数），倒排记录表则记录着上述词项（或关键词）所出现的位置，或出现的文档及网页 ID 等相关信息。

1、正排索引与倒排索引

咱们先来看什么是倒排索引，以及倒排索引与正排索引之间的区别：

我们知道，搜索引擎的关键步骤就是建立倒排索引，所谓倒排索引一般表示为一个关键词，然后是它的频度（出现的次数），位置（出现在哪一篇文章或网页中，及有关的日期，作者等信息），它相当于为互联网上几千亿页网页做了一个索引，好比一本书的目录、标签一般。读者想看哪一个主题相关的章节，直接根据目录即可找到相关的页面。不必再从书的第一页到最后一页，一页一页的查找。

接下来，阐述下正排索引与倒排索引的区别：

一般索引（正排索引）

正排表是以文档的 ID 为关键字，表中记录文档中每个字的位置信息，查找时扫描表中每个文档中字的信息直到找出所有包含查询关键字的文档。正排表结构如图 1 所示，这种组织方法在建立索引的时候结构比较简单，建立比较方便且易于维护；因为索引是基于文档建立的，若是有新的文档假如，直接为该文档建立一个新的索引块，挂接在原来索引文件的后面。若是有文档删除，则直接找到该文档号文档对应的索引信息，将其直接删除。但是在

查询的时候需对所有的文档进行扫描以确保没有遗漏，这样就使得检索时间大大延长，检索效率低下。

尽管正排表的工作原理非常的简单，但是由于其检索效率太低，除非在特定情况下，否则实用性价值不大。

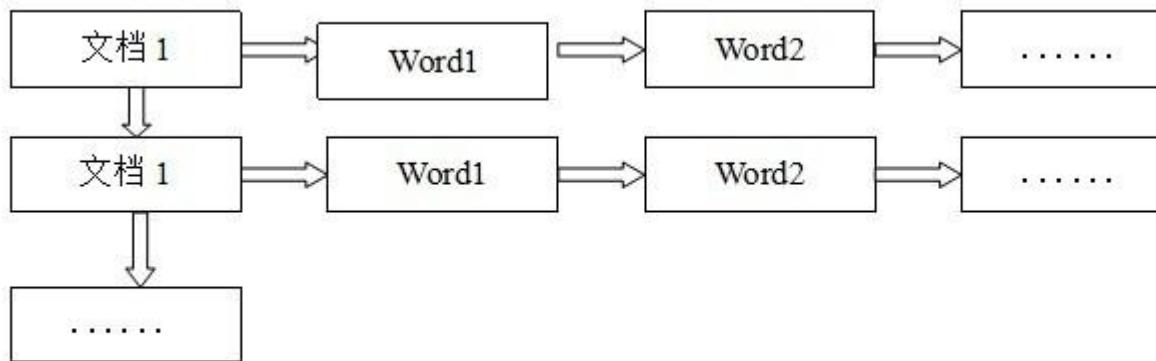


图 1 正排表结构图

倒排索引

倒排表以字或词为关键字进行索引，表中关键字所对应的记录表项记录了出现这个字或词的所有文档，一个表项就是一个字表段，它记录该文档的 ID 和字符在该文档中出现的位置情况。由于每个字或词对应的文档数量在动态变化，所以倒排表的建立和维护都较为复杂，但是在查询的时候由于可以一次得到查询关键字所对应的所有文档，所以效率高于正排表。在全文检索中，检索的快速响应是一个最为关键的性能，而索引建立由于在后台进行，尽管效率相对低一些，但不会影响整个搜索引擎的效率。

倒排表的结构图如图 2:

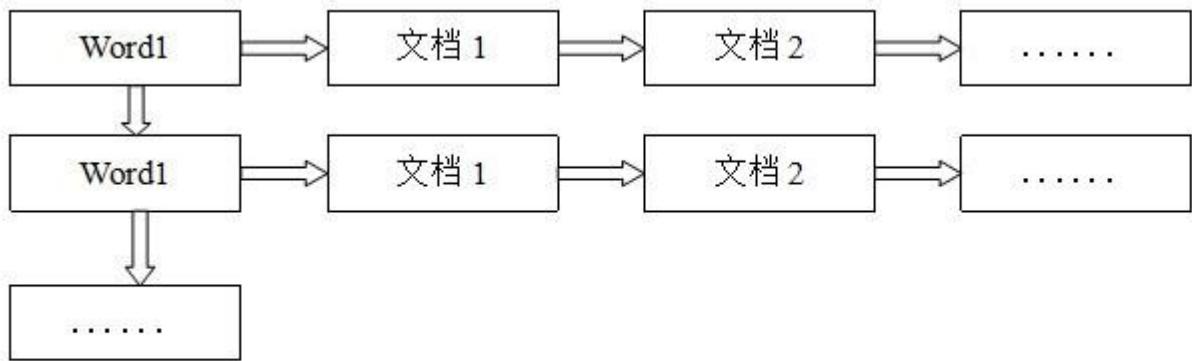


图 2 一倒排表结构图

倒排表的索引信息保存的是字或词后继数组模型、互关联后继数组模型条在文档内的位置，在同一篇文档内相邻的字或词条的前后关系没有被保存到索引文件内。

2、倒排索引中提取关键词

倒排索引是搜索引擎之基石。建成了倒排索引后，用户要查找某个 query，如在搜索框输入某个关键词：“结构之法”后，搜索引擎不会再使用爬虫又一个一个去抓取每一个网页，从上到下扫描网页，看这个网页有没有出现这个关键词，而是会在它预生成的倒排索引文件中查找和匹配包含这个关键词“结构之法”的所有网页。找到了之后，再按相关性度排序，最终把排序后的结果显示给用户。

所有结果	结构之法算法之道 - 博客频道 - CSDN.NET blog.csdn.net/v_JULY_v?utm_source=weibolife - 网页快照
图片	置顶]程序员面试、算法研究、编程艺术、红黑树4大系列集锦与总结. 程序员面试、算法研究、编程艺术、红黑树4大经典原创系列集锦与总结作者：July--结构之法算法之 ...
地图	



如下，即是一个倒排索引文件（不全），我们把它取名为 big_index，

文件中每一较短的，不包含有“#####”符号的便是某个关键词，及这个关键词的出现次数。现在要从这个大索引文件中提取出这些关键词，--Firelf--，-11，-Winter-，.，007，007：天降杀机，02Chan.. 如何做到呢？一行一行的扫描整个索引文件么？

何意？之前已经说过：倒排索引包含词典和倒排记录表两个部分，词典一般有词项（或称为关键词）和词项频率（即这个词项或关键词出现的次数），倒排记录表则记录着上述词项（或关键词）所出现的位置，或出现的文档及网页 ID 等相关信息。

最简单的讲，就是要提取词典中的词项（关键词）：--Fireelf--，-11，-Winter-，.，007，007：天降杀机，02Chan...。

—Fireelf— (关键词) 8 (出现次数)

我们可以试着这么解决：通过查找#####便可判断某一行出现的词是不是关键词，但如果这样做的话，便要扫描整个索引文件的每一行，代价实在巨大。如何提高速度呢？对了，关键词后面的那个出现次数为我们问题的解决起到了很好的作用，如下注释所示：

```
//      本身没有##### 的行判定为关键词行，后跟这个关键词的行数 N (即词项频率)
//      接下来，截取关键词--Firelf--，然后读取后面关键词的行数 N
//      再跳过 N 行（滤过和避免扫描中间的倒排记录表信息）
//      读取下一个关键词..
```

有朋友指出，上述方法虽然减少了扫描的行数，但并没有减少 I/O 开销。读者是否有更好地办法？欢迎随时交流。

34.2、为提取出来的关键词编码

爱思考的朋友可能会问，上述从倒排索引文件中提取出那些关键词（词项）的操作是为了什么呢？其实如我个人微博上 12 月 12 日所述的 Hash 词典编码：

词典文件的编码：1、词典怎么生成（存储和构造词典）；2、如何运用 hash 对输入的汉字进行编码；3、如何更好的解决冲突，即不重复以及追加功能。具体例子为：事先构造好词典文件后，输入一个词，要求找到这个词的编码，然后将其编码输出。且要有不断能添加词的功能，不得重复。

步骤应该是如下：1、读索引文件；2、提取索引中的词出来；3、词典怎么生成，存储和构造词典；4、词典文件的编码：不重复与追加功能。编码比如，输入中国，他的编码可以为 10001，然后输入银行，他的编码可以为 10002。只要实现不断添加词功能，以及不重复即可，词典类的大文件，hash 最重要的是怎样避免冲突。

也就是说，现在我要对上述提取出来后的关键词进行编码，采取何种方式编码呢？暂时用 hash 函数编码。编码之后的效果将是每一个关键词都有一个特定的编码，如下图所示（与上文 big_index 文件比较一下便知）：

--Firelf-- 对应编码为：135942

-11 对应编码为：106101

....

1	-Firelf - 135942
2	-11 106101
3	-Winter - 90114
4	4 140059
5	007 106205
6	007: 天降杀机 37634
7	02Chan 8200
8	08:30 194768
9	09:34 195980
10	09:43 195981
11	10+10 25270
12	10086 25452
13	10:33 25797
14	10月团购报告 193308
15	11:14 26987
16	11月8日 30982
17	12580生活播报 18158
18	139说客 97635
19	14:51 30587
20	15:19 31755
21	1626潮流双周刊 100430
22	163 106246
23	163邮箱 142794
24	17173 33830
25	178游戏网 119117
26	1881 138811
27	1967-liu 28210
28	1983组合龙飞龙革 53522
29	199it 36544
30	1ging 104981
31	1号店 111818
32	2001国足十强赛全场 120106
33	2010年新疆gdp 101348
34	2011-11-01 41963
35	2011-11-02 41963
36	2011-11-03 41963
37	2011-11-04 41963
38	2011-11-05 41963
39	2011-11-07 41963

但细心的朋友一看上图便知，其中第 34~39 行显示，有重复的编码，那么如何解决这个不重复编码的问题呢？

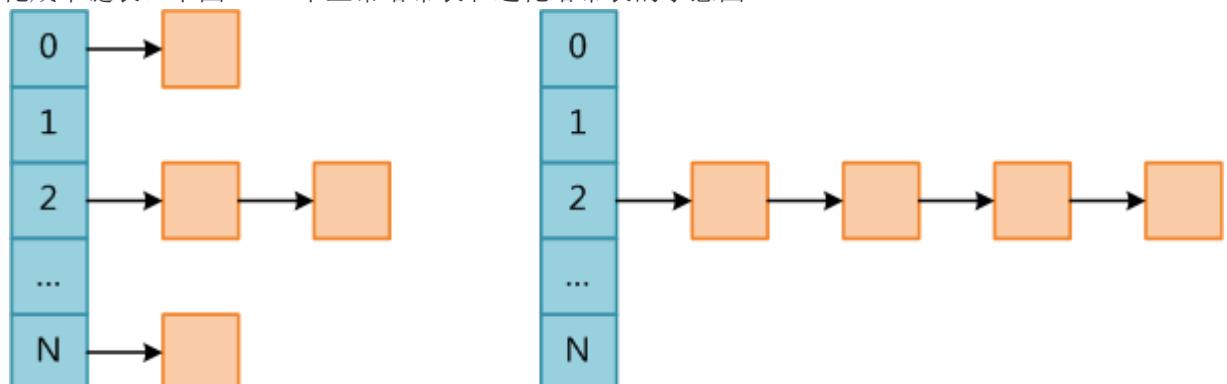
用 hash 表编码？但其极易产生冲突碰撞，为什么？请看：

哈希表是一种查找效率极高的数据结构，很多语言都在内部实现了哈希表。PHP 中的哈希表是一种极为重要的数据结构，不但用于表示 **Array** 数据类型，还在 Zend 虚拟机内部用于存储上下文环境信息（执行上下文的变量及函数均使用哈希表结构存储）。

理想情况下哈希表插入和查找操作的时间复杂度均为 $O(1)$ ，任何一个数据项可以在一个与哈希表长度无关的时间内计算出一个哈希值（key），然后在常量时间内定位到一个桶（术语 **bucket**，表示哈希表中的一个位置）。当然这是理想情况下，因为任何哈希表的长度都是有限的，所以一定存在不同的数据项具有相同哈希值的情况，此时不同数据项被定为到同一个桶，称为碰撞（collision）。哈希表的实现需要解决碰撞问题，碰撞解决大体有两种思路，第一种是根据某种原则将被碰撞数据定为到其它桶，例如线性探测——如果数据在插入时发生了碰撞，则顺序查找这个桶后面的桶，将其放入第一个没有被使用的桶；第二种策略是每个桶不是一个只能容纳单个数据项的位置，而是一个可容纳多个数据的数据结构（例如链表或红黑树），所有碰撞的数据以某种数据结构的形式组织起来。

不论使用了哪种碰撞解决策略，都导致插入和查找操作的时间复杂度不再是 $O(1)$ 。以查找为例，不能通过 key 定位到桶就结束，必须还要比较原始 key（即未做哈希之前的 key）是否相等，如果不相等，则要使用与插入相同的算法继续查找，直到找到匹配的值或确认数据不在哈希表中。

PHP 是使用单链表存储碰撞的数据，因此实际上 PHP 哈希表的平均查找复杂度为 $O(L)$ ，其中 L 为桶链表的平均长度；而最坏复杂度为 $O(N)$ ，此时所有数据全部碰撞，哈希表退化成单链表。下图 PHP 中正常哈希表和退化哈希表的示意图。



正常哈希表

退化哈希表

哈希表碰撞攻击就是通过精心构造数据，使得所有数据全部碰撞，人为将哈希表变成一个退化的单链表，此时哈希表各种操作的时间均提升了一个数量级，因此会消耗大量 CPU 资源，导致系统无法快速响应请求，从而达到拒绝服务攻击（DoS）的目的。

可以看到，进行哈希碰撞攻击的前提是哈希算法特别容易找出碰撞，如果是 MD5 或者 SHA1 那基本就没戏了，幸运的是（也可以说不幸的是）大多数编程语言使用的哈希算法都十分简单（这是为了效率考虑），因此可以不费吹灰之力构造出攻击数据（引自：<http://www.codinglabs.org/html/hash-collisions-attack-on-php.html>）。

3、暴雪的 Hash 算法

值得一提的是，在解决 Hash 冲突的时候，搞的焦头烂额，结果今天上午在自己的博客内的一篇文章（十一、从头到尾彻底解析 Hash 表算法）内找到了解决办法：网上流传甚广的暴雪的 Hash 算法。OK，接下来，咱们回顾下暴雪的 hash 表算法：

“ 接下来，咱们来具体分析一下一个最快的 Hash 表算法。

我们由一个简单的问题逐步入手：有一个庞大的字符串数组，然后给你一个单独的字符串，让你从这个数组中查找是否有这个字符串并找到它，你会怎么做？

有一个方法最简单，老老实实从头查到尾，一个一个比较，直到找到为止，我想只要学过程序设计的人都能把这样一个程序作出来，但要是有程序员把这样的程序交给用户，我只能用无语来评价，或许它真的能工作，但...也只能如此了。

最合适的算法自然是使用 HashTable(哈希表)，先介绍介绍其中的基本知识，所谓 Hash，一般是一个整数，通过某种算法，可以把一个字符串“压缩”成一个整数。当然，无论如何，一个 32 位整数是无法对应回一个字符串的，但在程序中，两个字符串计算出的 Hash 值相等的可能非常小，下面看看在 MPQ 中的 Hash 算法：

函数 prepareCryptTable 以下的函数生成一个长度为 0x500 (合 10 进制数: 1280) 的 cryptTable[0x500]

```
1.      //函数 prepareCryptTable 以下的函数生成一个长度为 0x500 (合 10 进制数: 1280) 的
2.      cryptTable[0x500]
3.      void prepareCryptTable()
4.      {
5.          unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
6.
7.          for( index1 = 0; index1 < 0x100; index1++ )
8.          {
9.              for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
10.             {
11.
12.                 unsigned long temp1, temp2;
13.
14.                 seed = (seed * 125 + 3) % 0x2AAAAB;
15.                 temp1 = (seed & 0xFFFF) << 0x10;
16.
17.                 seed = (seed * 125 + 3) % 0x2AAAAB;
18.                 temp2 = (seed & 0xFFFF);
19.
20.                 cryptTable[index2] = ( temp1 | temp2 );
21.             }
22.         }
23.     }
```

函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值, 其中 dwHashType 为 hash 的类型,

```
1.      //函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值, 其中 dwHashType 为 hash
2.      的类型,
3.      unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
4.
5.      {
6.          unsigned char *key   = (unsigned char *)lpszkeyName;
7.          unsigned long seed1 = 0x7FED7FED;
8.          unsigned long seed2 = 0xFFFFFFFF;
9.          int ch;
10.
11.         while( *key != 0 )
12.         {
13.             ch = *key++;
14.             seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
15.             seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
```

```

14.         }
15.         return seed1;
16.     }

```

Blizzard 的这个算法是非常高效的，被称为"One-Way Hash"(A one-way hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串"unitneutralacritter.grp"通过这个算法得到的结果是 0xA26067F3。

是不是把第一个算法改进一下，改成逐个比较字符串的 Hash 值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个哈希表(Hash Table)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，

例如 1024，每一个 Hash 值通过取模运算 (mod) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置有没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的 O(1)，现在仔细看看这个算法吧：

```

1.     typedef struct
2.     {
3.         int nHashA;
4.         int nHashB;
5.         char bExists;
6.         .....
7.     } SOMESTRUCTURE;
8.     //一种可能的结构体定义？

```

函数 GetHashTablePos 下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则， return -1.

```

1.     //函数 GetHashTablePos 下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则， return -1.
2.     int GetHashTablePos( char *lpszString, SOMESTRUCTURE *lpTable )
3.     //lpszString 要在 Hash 表中查找的字符串， lpTable 为存储字符串 Hash 值的 Hash 表。
4.     {
5.         int nHash = HashString(lpszString); //调用上述函数 HashString, 返回要查找字符串 lpszString 的 Hash 值。
6.         int nHashPos = nHash % nTableSize;
7.
8.         if ( lpTable[nHashPos].bExists && !strcmp( lpTable[nHashPos].pString,
lpszString ) )
9.         { //如果找到的 Hash 值在表中存在，且要查找的字符串与表中对应位置的字符串相同，
10.             return nHashPos; //返回找到的 Hash 值
11.         }
12.         else
13.         {
14.             return -1;
15.         }
16.     }

```

看到此，我想大家都在想一个很严重的问题：“如果两个字符串在哈希表中对应的位置相同怎么办？”，毕竟一个数组容量是有限的，这种可能性很大。解决该问题的方法很多，我首

先想到的就是用“链表”，感谢大学里学的数据结构教会了这个百试百灵的法宝，我遇到的很多算法都可以转化成链表来解决，只要在哈希表的每个入口挂一个链表，保存所有对应的字符串就OK了。事情到此似乎有了完美的结局，如果是把问题独自交给我解决，此时我可能就要开始定义数据结构然后写代码了。

然而 Blizzard 的程序员使用的方法则是更精妙的方法。基本原理就是：他们在哈希表中不是用一个哈希值而是用三个哈希值来校验字符串。
“

MPQ 使用文件名哈希表来跟踪内部的所有文件。但是这个表的格式与正常的哈希表有一些不同。首先，它没有使用哈希作为下标，把实际的文件名存储在表中用于验证，实际上它根本就没有存储文件名。而是使用了 3 种不同的哈希：一个用于哈希表的下标，两个用于验证。这两个验证哈希替代了实际文件名。

当然了，这样仍然会出现 2 个不同的文件名哈希到 3 个同样的哈希。但是这种情况发生的概率平均是：1:18889465931478580854784，这个概率对于任何人来说应该都是足够小的。现在再回到数据结构上，Blizzard 使用的哈希表没有使用链表，而采用“顺延”的方式来解决问题。”下面，咱们来看看这个网上流传甚广的暴雪 hash 算法：

函数 GetHashTablePos 中，lpszString 为要在 hash 表中查找的字符串；lpTable 为存储字符串 hash 值的 hash 表；nTableSize 为 hash 表的长度：

```
1.      //函数 GetHashTablePos 中，lpszString 为要在 hash 表中查找的字符串；lpTable 为存储
2.      //字符串 hash 值的 hash 表；nTableSize 为 hash 表的长度：
3.      int GetHashTablePos( char *lpszString, MPQHASHTABLE *lpTable, int nTableSize
4.      )
5.      {
6.          const int HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;
7.
8.          int nHash = HashString( lpszString, HASH_OFFSET );
9.          int nHashA = HashString( lpszString, HASH_A );
10.         int nHashB = HashString( lpszString, HASH_B );
11.         int nHashStart = nHash % nTableSize;
12.         int nHashPos = nHashStart;
13.
14.         while ( lpTable[nHashPos].bExists )
15.         {
16.             // 如果仅仅是判断在该表中时候存在这个字符串，就比较这两个 hash 值就可以了，不用对
17.             // 结构体中的字符串进行比较。
18.             // 这样会加快运行的速度？减少 hash 表占用的空间？这种方法一般应用在什么场
19.             // 合？
20.             if ( lpTable[nHashPos].nHashA == nHashA
21.                 && lpTable[nHashPos].nHashB == nHashB )
22.             {
23.                 return nHashPos;
24.             }
25.         }
```

```

21.         else
22.         {
23.             nHashPos = (nHashPos + 1) % nTableSize;
24.         }
25.
26.         if (nHashPos == nHashStart)
27.             break;
28.         }
29.     return -1;
30. }
```

上述程序解释：

- 1、计算出字符串的三个哈希值（一个用来确定位置，另外两个用来校验）
- 2、察看哈希表中的这个位置
- 3、哈希表中这个位置为空吗？如果为空，则肯定该字符串不存在，返回-1。
- 4、如果存在，则检查其他两个哈希值是否也匹配，如果匹配，则表示找到了该字符串，返回其 Hash 值。
- 5、移到下一个位置，如果已经移到了表的末尾，则反绕到表的开始位置起继续查询
- 6、看看是不是又回到了原来的位置，如果是，则返回没找到
- 7、回到 3。

4、不重复 Hash 编码

有了上面的暴雪 Hash 算法。咱们的问题便可解决了。不过，有两点必须先提醒读者：
 1、Hash 表起初要初始化；2、暴雪的 Hash 算法对于查询那样处理可以，但对插入就不能那么解决。

关键主体代码如下：

```

1.      //函数 prepareCryptTable 以下的函数生成一个长度为 0x500 (合 10 进制数：1280) 的
cryptTable[0x500]
2.      void prepareCryptTable()
3.      {
4.          unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
5.
6.          for( index1 = 0; index1 < 0x100; index1++ )
```

```

7.        {
8.            for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100)
9.            {
10.                unsigned long temp1, temp2;
11.                seed = (seed * 125 + 3) % 0x2AAAAB;
12.                temp1 = (seed & 0xFFFF)<<0x10;
13.                seed = (seed * 125 + 3) % 0x2AAAAB;
14.                temp2 = (seed & 0xFFFF);
15.                cryptTable[index2] = ( temp1 | temp2 );
16.            }
17.        }
18.    }
19.
20.    //函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值,其中 dwHashType 为 hash
   的类型,
21.    unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
22.
23.    {
24.        unsigned char *key = (unsigned char *)lpszkeyName;
25.        unsigned long seed1 = 0x7FED7FED;
26.        unsigned long seed2 = 0xFFFFFFFF;
27.        int ch;
28.
29.        while( *key != 0 )
30.        {
31.            ch = *key++;
32.            seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
33.            seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
34.        }
35.        return seed1;
36.    }
37.    /////////////////////////////////
38.    //function: 哈希词典 编码
39.    //parameter:
40.    //author: lei.zhou
41.    //time: 2011-12-14
42.    /////////////////////////////////
43.    MPQHASHTABLE TestHashTable[nTableSize];
44.    int TestHashCTable[nTableSize];
45.    int TestHashDTable[nTableSize];
46.    key_list test_data[nTableSize];
47.
48.    //直接调用上面的 hashstring, nHashPos 就是对应的 HASH 值。

```

```

49.     int insert_string(const char *string_in)
50.     {
51.         const int HASH_OFFSET = 0, HASH_C = 1, HASH_D = 2;
52.         unsigned int nHash = HashString(string_in, HASH_OFFSET);
53.         unsigned int nHashC = HashString(string_in, HASH_C);
54.         unsigned int nHashD = HashString(string_in, HASH_D);
55.         unsigned int nHashStart = nHash % nTableSize;
56.         unsigned int nHashPos = nHashStart;
57.         int ln, ires = 0;
58.
59.         while (TestHashTable[nHashPos].bExists)
60.         {
61.             //      if (TestHashCTable[nHashPos] == (int) nHashC && TestHashDTable[nHas-
62. hPos] == (int) nHashD)
63.                 //      break;
64.                 //...
65.                 //      else
66.                     //如之前所提示读者的那般，暴雪的 Hash 算法对于查询那样处理可以，但对插入就不
67.                     //能那么解决
68.             nHashPos = (nHashPos + 1) % nTableSize;
69.
70.             if (nHashPos == nHashStart)
71.                 break;
72.
73.             ln = strlen(string_in);
74.             if (!TestHashTable[nHashPos].bExists && (ln < nMaxStrLen))
75.             {
76.                 TestHashCTable[nHashPos] = nHashC;
77.                 TestHashDTable[nHashPos] = nHashD;
78.
79.                 test_data[nHashPos] = (KEYNODE *) malloc (sizeof(KEYNODE) * 1);
80.                 if(test_data[nHashPos] == NULL)
81.                 {
82.                     printf("10000 EMS ERROR !!!!\n");
83.                     return 0;
84.                 }
85.                 test_data[nHashPos]->pkey = (char *)malloc(ln+1);
86.                 if(test_data[nHashPos]->pkey == NULL)
87.                 {
88.                     printf("10000 EMS ERROR !!!!\n");
89.                     return 0;
90.                 }

```

```

91.         memset(test_data[nHashPos]->pkey, 0, ln+1);
92.         strncpy(test_data[nHashPos]->pkey, string_in, ln);
93.         *((test_data[nHashPos]->pkey)+ln) = 0;
94.         test_data[nHashPos]->weight = nHashPos;
95.
96.         TestHashTable[nHashPos].bExists = 1;
97.     }
98.     else
99.     {
100.         if(TestHashTable[nHashPos].bExists)
101.             printf("30000 in the hash table %s !!!\n", string_in);
102.         else
103.             printf("90000 strkey error !!!\n");
104.     }
105.     return nHashPos;
106. }
107. 
```

接下来要读取索引文件 big_index 对其中的关键词进行编码（为了简单起见，直接一行一行扫描读写，没有跳过行数了）：

```

1.     void bigIndex_hash(const char *docpath, const char *hashpath)
2.     {
3.         FILE *fr, *fw;
4.         int len;
5.         char *pbuf, *p;
6.         char dockey[TERM_MAX LENG];
7.
8.         if(docpath == NULL || *docpath == '\0')
9.             return;
10.
11.        if(hashpath == NULL || *hashpath == '\0')
12.            return;
13.
14.        fr = fopen(docpath, "rb"); //读取文件 docpath
15.        fw = fopen(hashpath, "wb");
16.        if(fr == NULL || fw == NULL)
17.        {
18.            printf("open read or write file error!\n");
19.            return;
20.        }
21.
22.        pbuf = (char*)malloc(BUFF_MAX LENG);
23.        if(pbuf == NULL) 
```

```

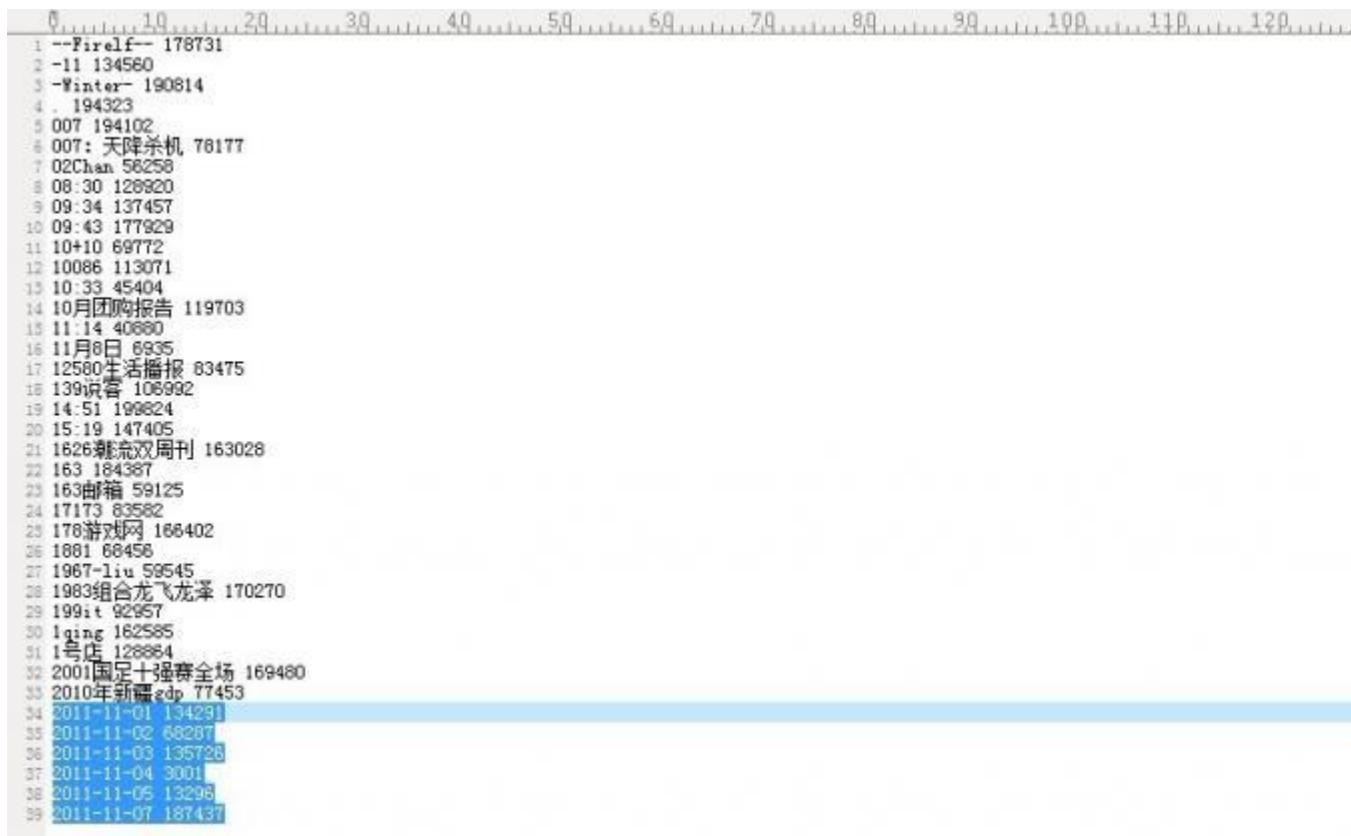
24.         {
25.             fclose(fr);
26.             return ;
27.         }
28.
29.         memset(pbuf, 0, BUFF_MAX_LEN);
30.
31.         while(fgets(pbuf, BUFF_MAX_LEN, fr))
32.         {
33.             len = GetRealString(pbuf);
34.             if(len <= 1)
35.                 continue;
36.             p = strstr(pbuf, "#####");
37.             if(p != NULL)
38.                 continue;
39.
40.             p = strstr(pbuf, " ");
41.             if (p == NULL)
42.             {
43.                 printf("file contents error!");
44.             }
45.
46.             len = p - pbuf;
47.             dockey[0] = 0;
48.             strncpy(dockey, pbuf, len);
49.
50.             dockey[len] = 0;
51.
52.             int num = insert_string(dockey);
53.
54.             dockey[len] = ' ';
55.             dockey[len+1] = '\0';
56.             char str[20];
57.             itoa(num, str, 10);
58.
59.             strncat(dockey, str);
60.             dockey[len+strlen(str)+1] = '\0';
61.             fprintf (fw, "%s\n", dockey);
62.
63.         }
64.         free(pbuf);
65.         fclose(fr);
66.         fclose(fw);
67.     }

```

主函数已经很简单了，如下：

```
1.     int main()
2.     {
3.         prepareCryptTable(); //Hash 表起初要初始化
4.
5.         //现在要把整个 big_index 文件插入 hash 表，以取得编码结果
6.         bigIndex_hash("big_index.txt", "hashpath.txt");
7.         system("pause");
8.
9.     return 0;
10. }
```

程序运行后生成的 hashpath.txt 文件如下：

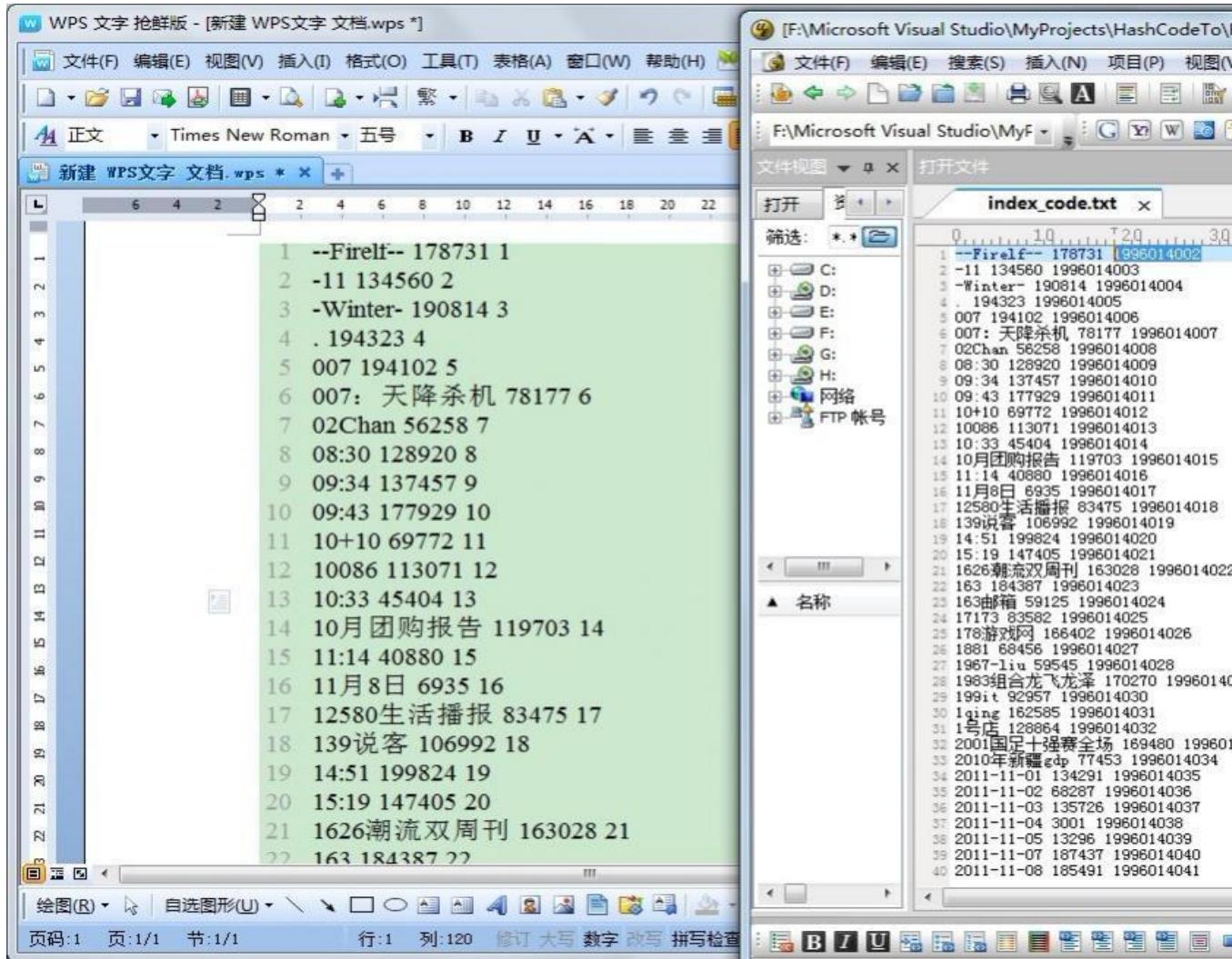


1	--Firelf--	178731
2	-11	134580
3	-Winter-	190814
4	.	194323
5	007	194102
6	007: 天降杀机	78177
7	02Chan	58258
8	08:30	128920
9	09:34	137457
10	09:43	177929
11	10+10	69772
12	10086	113071
13	10:33	45404
14	10月团购报告	119703
15	11:14	40680
16	11月8日	6935
17	12580生活播报	83475
18	139说客	106992
19	14:51	199824
20	15:19	147405
21	1626潮流双周刊	163028
22	163	184387
23	163邮箱	59125
24	17173	83582
25	178游戏网	166402
26	1881	68458
27	1967-liu	58545
28	1983组合龙飞龙革	170270
29	1991t	92957
30	1ging	162585
31	1号店	128884
32	2001国足十强赛全场	169480
33	2010年新疆	edp 77453
34	2011-11-01	134291
35	2011-11-02	69287
36	2011-11-03	135728
37	2011-11-04	3001
38	2011-11-05	13298
39	2011-11-07	187437

如上所示，采取暴雪的 Hash 算法并在插入的时候做适当处理，当再次对上文中的索引文件 big_index 进行 Hash 编码后，冲突问题已经得到初步解决。当然，还有待更进一步更深入的测试。

后续添上数目索引 1~10000...

后来又为上述文件中的关键词编了码一个计数的内码，不过，奇怪的是，同样的代码，在 Dev C++ 与 VS2010 上运行结果却不同（左边 dev 上计数从“1”开始，VS 上计数从“1994014002”开始），如下图所示：



在上面的 bigIndex_hashcode 函数的基础上，修改如下，即可得到上面的效果：

```

1. void bigIndex_hashcode(const char *in_file_path, const char *out_file_path)

2. {
3.     FILE *fr, *fw;
4.     int len, value;
5.     char *pbuf, *pleft, *p;
6.     char keyvalue[TERM_MAX LENG], str[WORD_MAX LENG];
7.
8.     if(in_file_path == NULL || *in_file_path == '\0') {
9.         printf("input file path error!\n");
10.        return;

```

```

11.        }
12.
13.        if(out_file_path == NULL || *out_file_path == '\0') {
14.            printf("output file path error!\n");
15.            return;
16.        }
17.
18.        fr = fopen(in_file_path, "r"); //读取 in_file_path 路径文件
19.        fw = fopen(out_file_path, "w");
20.
21.        if(fr == NULL || fw == NULL)
22.        {
23.            printf("open read or write file error!\n");
24.            return;
25.        }
26.
27.        pbuf = (char*)malloc(BUFF_MAX LENG);
28.        pleft = (char*)malloc(BUFF_MAX LENG);
29.        if(pbuf == NULL || pleft == NULL)
30.        {
31.            printf("allocate memory error!");
32.            fclose(fr);
33.            return ;
34.        }
35.
36.        memset(pbuf, 0, BUFF_MAX LENG);
37.
38.        int offset = 1;
39.        while(fgets(pbuf, BUFF_MAX LENG, fr))
40.        {
41.            if (--offset > 0)
42.                continue;
43.
44.            if(GetRealString(pbuf) <= 1)
45.                continue;
46.
47.            p = strstr(pbuf, "#");
48.            if(p != NULL)
49.                continue;
50.
51.            p = strstr(pbuf, " ");
52.            if (p == NULL)
53.            {
54.                printf("file contents error!");

```

```

55.         }
56.
57.         len = p - pbuf;
58.
59.         // 确定跳过行数
60.         strcpy(pleft, p+1);
61.         offset = atoi(pleft) + 1;
62.
63.         strncpy(keyvalue, pbuf, len);
64.         keyvalue[len] = '\0';
65.         value = insert_string(keyvalue);
66.
67.         if (value != -1) {
68.
69.             // key value 中插入空格
70.             keyvalue[len] = ' ';
71.             keyvalue[len+1] = '\0';
72.
73.             itoa(value, str, 10);
74.             strcat(keyvalue, str);
75.
76.             keyvalue[len+strlen(str)+1] = ' ';
77.             keyvalue[len+strlen(str)+2] = '\0';
78.
79.             keysize++;
80.             itoa(keysize, str, 10);
81.             strcat(keyvalue, str);
82.
83.             // 将 key value 写入文件
84.             fprintf (fw, "%s\n", keyvalue);
85.
86.         }
87.     }
88.     free(pbuf);
89.     fclose(fr);
90.     fclose(fw);
91. }

```

小结

本文有一点值得一提的是，在此前的这篇文章（十一、从头到尾彻底解析 Hash 表算法）之中，只是对 Hash 表及暴雪的 Hash 算法有过学习和了解，但尚未真正运用过它，而在本章中体现，证明还是之前写的文章，及之前对 Hash 表等算法的学习还是有一定作用的。

同时，也顺便对暴雪的 Hash 函数算是做了个测试，其的确能解决一般的冲突性问题，创造这个算法的人不简单呐。

后记

再次感谢老大 xiaoqi，以及艺术室内朋友 xiaolin, 555, yansha 的指导。没有他们的帮助，我将寸步难行。日后，自己博客内的文章要经常回顾，好好体会。同时，写作本文时，刚接触倒排索引等相关问题不久，若有任何问题，欢迎随时交流或批评指正。

最后，基于本 blog 的分为程序语言，数据结构，算法讨论，面试题库，编程技巧五大板块交流的论坛正在加紧建设当中（总负责人：scott && yinhex && 网络骑士），相信不久以后便会与大家见面。谢谢。完。

十二、快速排序算法

作者 July 二零一一年一月四日

写此八大排序算法系列之前，先说点题外话。

每写一篇文章，我都会遵循以下几点原则：

- 一、保持版面的尽量清晰，力保排版良好。
- 二、力争所写的东西，清晰易懂，图文并茂
- 三、尽最大可能确保所写的东西精准，有实用价值。

因为，我觉得，你既然要把你的文章，公布出来，那么你就一定要为你的读者负责。
不然，就不要发表出来。一切，为读者服务。

ok，闲不多说。接下来，咱们立刻进入本文章的主题，排序算法。

众所周知，快速排序算法是排序算法中的重头戏。

因此，本系列，本文就从快速排序开始。

一、快速排序算法的基本特性

时间复杂度： $O(n \lg n)$

最坏： $O(n^2)$

空间复杂度: $O(n \lg n)$

不稳定。

快速排序是一种排序算法，对包含 n 个数的输入数组，平均时间为 $O(n \lg n)$ ，最坏情况是 $O(n^2)$ 。

通常是用于排序的最佳选择。因为，排序最快，也只能达到 $O(n \lg n)$ 。

二、快速排序算法的描述

算法导论，第 7 章

快速排序时基于分治模式处理的，

对一个典型子数组 $A[p \dots r]$ 排序的分治过程为三个步骤：

1. 分解：

$A[p..r]$ 被划分为两个（可能空）的子数组 $A[p \dots q-1]$ 和 $A[q+1 \dots r]$ ，使得

$A[p \dots q-1] \leq A[q] \leq A[q+1 \dots r]$

2. 解决：通过递归调用快速排序，对子数组 $A[p \dots q-1]$ 和 $A[q+1 \dots r]$ 排序。

3. 合并。

三、快速排序算法

版本一：

QUICKSORT(A, p, r)

```
1 if p < r
2   then q ← PARTITION(A, p, r) //关键
3   QUICKSORT(A, p, q - 1)
4   QUICKSORT(A, q + 1, r)
```

数组划分

快速排序算法的关键是 PARTITION 过程，它对 $A[p..r]$ 进行就地重排：

PARTITION(A, p, r)

```
1 x ← A[r]
2 i ← p - 1
3 for j ← p to r - 1
4   do if A[j] ≤ x
5     then i ← i + 1
```

```

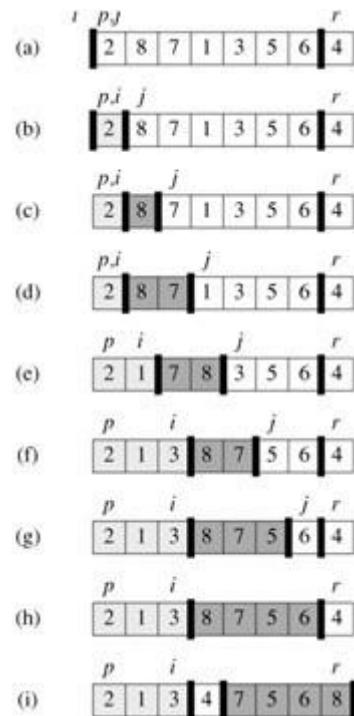
6           exchange A[i] <-> A[j]
7 exchange A[i + 1] <-> A[r]
8 return i + 1

```

ok, 咱们来举一个具体而完整的例子。

来对以下数组，进行快速排序，

2 8 7 1 3 5 6 4(主元)



一、

i p/j

2 8 7 1 3 5 6 4(主元)

j 指的 $2 \leq 4$, 于是 $i++$, i 也指到 2, 2 和 2 互换, 原数组不变。

j 后移, 直到指向 1..

二、

j (指向 1) ≤ 4 , 于是 $i++$

i 指向了 8, 所以 8 与 1 交换。

数组变成了:

i j

2 1 7 8 3 5 6 4

三、j后移，指向了3,3<=4，于是i++

i这是指向了7，于是7与3交换。

数组变成了：

i j
2 1 3 8 7 5 6 4

四、j继续后移，发现没有再比4小的数，所以，执行到了最后一步，

即上述PARTITION(A, p, r)代码部分的第7行。

因此，i后移一个单位，指向了8

i j
2 1 3 8 7 5 6 4

A[i + 1] <-> A[r]，即8与4交换，所以，数组最终变成了如下形式，

2 1 3 4 7 5 6 8

ok，快速排序第一趟完成。

4把整个数组分成了俩部分，2 1 3, 7 5 6 8，再递归对这俩部分分别快速排序。

i p/j

2 1 3(主元)

2与2互换，不变，然后又是1与1互换，还是不变，最后，3与3互换，不变，
最终，3把2 1 3，分成了俩部分，2 1，和3.

再对2 1，递归排序，最终结果成为了1 2 3.

7 5 6 8(主元)，7、5、6、都比8小，所以第一趟，还是7 5 6 8，

不过，此刻8把7 5 6 8，分成了7 5 6，和8.[7 5 6->5 7 6->5 6 7]

再对7 5 6，递归排序，最终结果变成5 6 7 8。

ok，所有过程，全部分析完成。

最后，看下我画的图：

快速排序

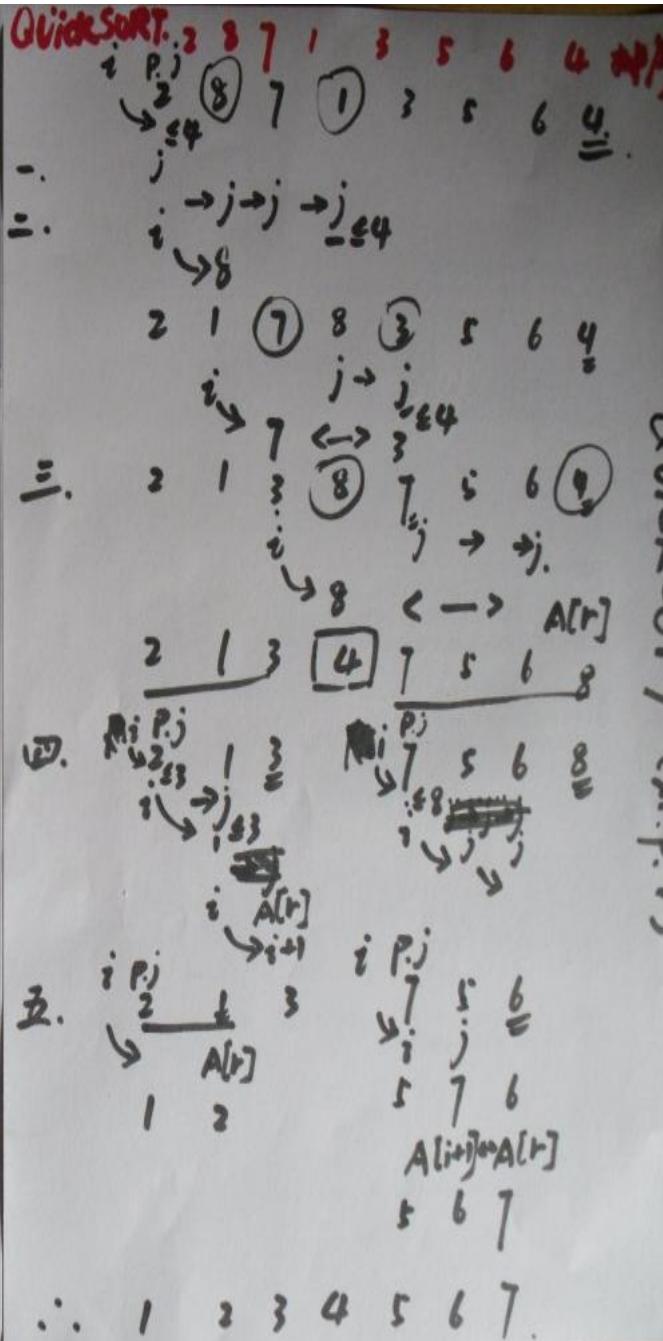
QUICKSORT(A, P, R)

1. if $P = R$
2. then $q \leftarrow \text{PARTITION}(A, P, R)$
3. $\text{QUICKSORT}(A, P, q-1)$
4. $\text{QUICKSORT}(A, q+1, R)$.

PARTITION(A, P, R)

1. $x \leftarrow A[R]$
2. $i \leftarrow P-1$
3. for $j \leftarrow P$ to $R-1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i+1$
6. Exchange $A[i] \leftrightarrow A[j]$
7. Exchange $A[i+1] \leftrightarrow A[R]$
8. return $i+1$

July. 二〇一二年一月一日



快速排序算法版本二

不过，这个版本不再选取（如上第一版本的）数组的最后一个元素为主元，而是选择，数组中的第一个元素为主元。

```
/*
 * 函数功能：快速排序算法
 */
```

```

/* 函数参数：结构类型 table 的指针变量 tab      */
/*          整型变量 left 和 right 左右边界的下标   */
/* 函数返回值：空          */
/* 文件名：quicksort.c 函数名：quicksort ()      */
/*********************************************************/
void quicksort(table *tab,int left,int right)
{
    int i,j;
    if(left<right)
    {
        i=left;j=right;
        tab->r[0]=tab->r[i]; //准备以本次最左边的元素值为标准进行划分，先保存其值
        do
        {
            while(tab->r[j].key>tab->r[0].key&&i<j)
                j--; //从右向左找第 1 个小于标准值的位置 j
            if(i<j) //找到了，位置为 j
            {
                tab->r[i].key=tab->r[j].key;i++;
            } //将第 j 个元素置于左端并重置 i
            while(tab->r[i].key<tab->r[0].key&&i<j)
                i++; //从左向右找第 1 个大于标准值的位置 i
            if(i<j) //找到了，位置为 i
            {
                tab->r[j].key=tab->r[i].key;j--;
            } //将第 i 个元素置于右端并重置 j
        }while(i!=j);

        tab->r[i]=tab->r[0]; //将标准值放入它的最终位置,本次划分结束
        quicksort(tab,left,i-1); //对标准值左半部递归调用本函数
        quicksort(tab,i+1,right); //对标准值右半部递归调用本函数
    }
}

```

ok, 咱们, 还是以上述相同的数组, 应用此快排算法的版本二, 来演示此排序过程:
这次, 以数组中的第一个元素 2 为主元。

2(主) 8 7 1 3 5 6 4

请细看:

2 8 7 1 3 5 6 4

i-> <-j
(找大) (找小)

一、 j

j 找第一个小于 2 的元素 1,1 赋给(覆盖重置)i 所指元素 2

得到:

1 8 7 3 5 6 4

i j

二、 i

i 找到第一个大于 2 的元素 8,8 赋给(覆盖重置)j 所指元素(NULL<-8)

1 7 8 3 5 6 4

i <-j

三、 j

j 继续左移, 在与 i 碰头之前, 没有找到比 2 小的元素, 结束。

最后, 主元 2 补上。

第一趟快排结束之后, 数组变成:

1 2 7 8 3 5 6 4

第二趟,

7 8 3 5 6 4
i-> <-j
(找大) (找小)

一、 j

j 找到 4, 比主元 7 小, 4 赋给 7 所处位置

得到:

4 8 3 5 6
i-> j

二、 i

i 找比 7 大的第一个元素 8,8 覆盖 j 所指元素(NULL)

4 3 5 6 8

i j

4 6 3 5 8

i-> j

i 与 j 碰头，结束。

第三趟：

4 6 3 5 7 8

.....

以下，分析原理，一致，略过。

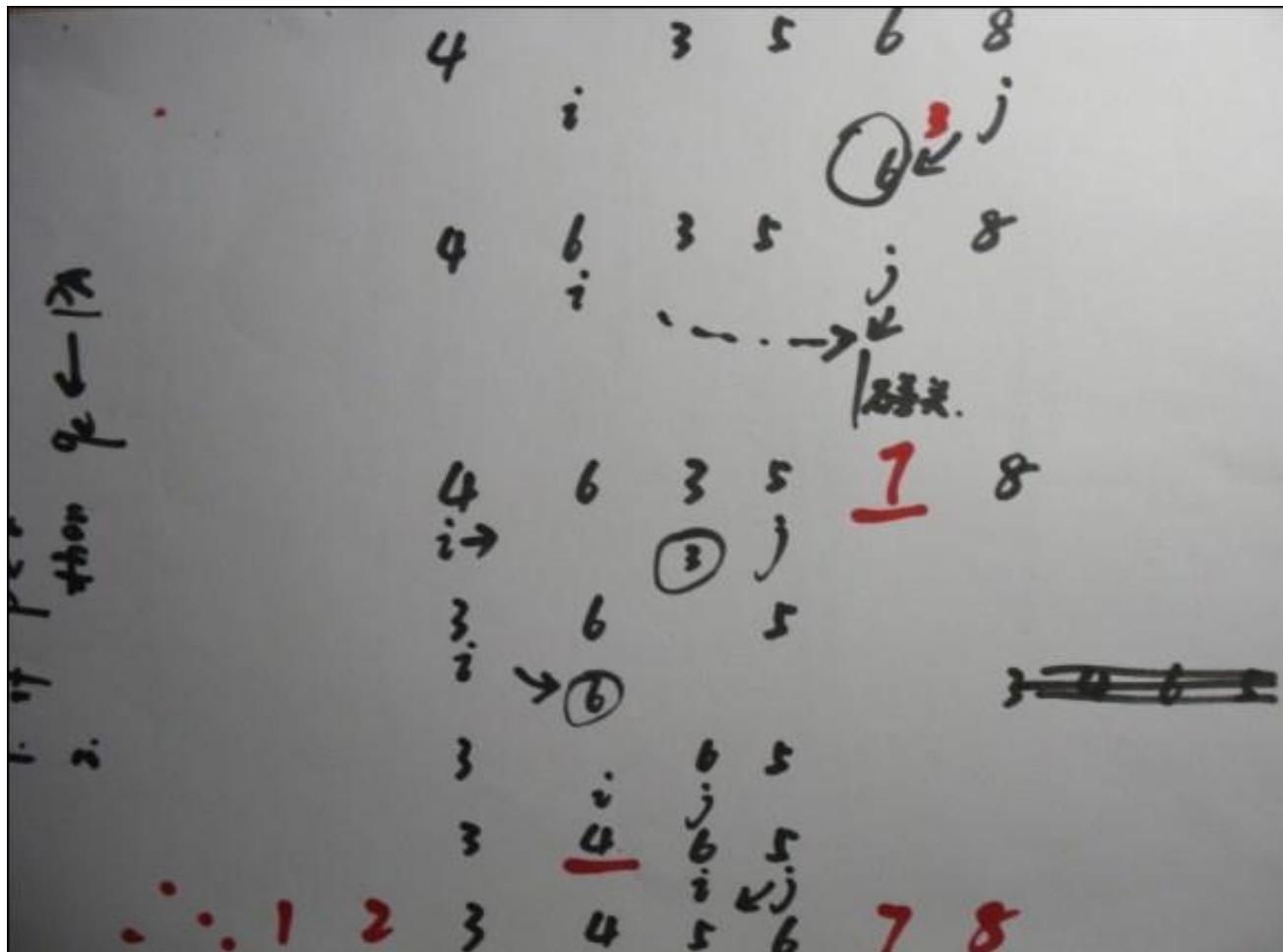
最后的结果，如下图所示：

1 2 3 4 5 6 7 8

相信，经过以上内容的具体分析，你一定明了了。

最后，贴一下我画的关于这个排序过程的图：

第二.	2	8	7	1	3	5	6	4	12345678
	2主.	8	7	1	3	5	6	4	
	i → (找大)			①	j ← 1				← j (找小)
					j				
	1	8	7		3	5	6	4	
	i	8		j					
	2 i								
	1	7	8	3	5	6	4		
	i	7		j					
	2	8							
	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								
	50								
	51								
	52								
	53								
	54								
	55								
	56								
	57								
	58								
	59								
	60								
	61								
	62								
	63								
	64								
	65								
	66								
	67								
	68								
	69								
	70								
	71								
	72								
	73								
	74								
	75								
	76								
	77								
	78								
	79								
	80								
	81								
	82								
	83								
	84								
	85								
	86								
	87								
	88								
	89								
	90								
	91								
	92								
	93								
	94								
	95								
	96								
	97								
	98								
	99								
	100								



完。一月五日补充。

OK, 上述两种算法，明白一种即可。

五、快速排序的最坏情况和最快情况。

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 $O(n^2)$ 。

亦即，快速排序算法的最坏情况并不比插入排序的更好。

此外，当数组完全排好序之后，快速排序的运行时间为 $O(n^2)$ 。

而在同样情况下，插入排序的运行时间为 $O(n)$ 。

//注，请注意理解这句话。我们说一个排序的时间复杂度，是仅仅针对一个元素的。

//意思是，把一个元素进行插入排序，即把它插入到有序的序列里，花的时间为 n 。

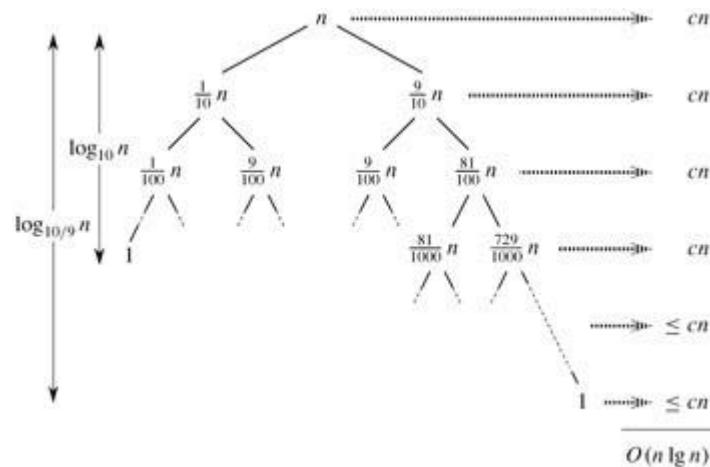
再来证明，最快情况下，即 **PARTITION** 可能做的最平衡的划分中，得到的每个子问题都不能大于 $n/2$ 。

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 $\lceil n/2 \rceil - 1$ 。

在这种情况下，快速排序的速度要快得多。为，

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

直观上，看，快速排序就是一颗递归树，其中，**PARTITION** 总是产生 9:1 的划分，总的运行时间为 $O(n \lg n)$ 。各结点中示出了子问题的规模。每一层的代价在右边显示。每一层包含一个常数 c 。



完。

July、二零一一年一月四日。

=====

请各位自行，思考以下这个版本，对应于上文哪个版本？

HOARE-PARTITION(A, p, r)

```

1 x ← A[p]
2 i ← p - 1
3 j ← r + 1
4 while TRUE
5   do repeat j ← j - 1
6     until A[j] ≤ x
7   repeat i ← i + 1
8     until A[i] ≥ x
9   if i < j
10    then exchange A[i] ↔ A[j]
11 else return j

```

我常常思考，为什么有的人当时明明读懂明白了一个算法，而一段时间过后，它又对此算法完全陌生而不了解了呢？

我想，究其根本，还是没有彻底明白此快速排序算法的原理，与来龙去脉...
那作何改进呢，只能找发明那个算法的原作者了，从原作者身上，再多挖掘点有用的东西出来。

July、二零一一年二月十五日更新。

最后，再给出一个快速排序算法的简洁示例：

Quicksort 函数

```

void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
}

```

```
    quicksort(m+1, u);  
}
```

如果函数的调用形式是 `quicksort(0, n-1)`, 那么这段代码将对一个全局数组 `x[n]` 进行排序。函数的两个参数分别是将要进行排序的子数组的下标: `l` 是较低的下标, 而 `u` 是较高的下标。

函数调用 `swap(i,j)` 将会交换 `x[i]` 与 `x[j]` 这两个元素。

第一次交换操作将会按照均匀分布的方式在 `l` 和 `u` 之间随机地选择一个划分元素。

ok, 更多请参考我写的关于快速排序算法的第二篇文章: [一之续、快速排序算法的深入分析](#), 第三篇文章: [十二、一之再续: 快速排序算法之所有版本的 c/c++ 实现](#)。

[July](#)、二零一一年二月二十日更新。

十二（续）、快速排序算法的深入分析

作者:July 二零一一年二月二十七日

前言

- 一、快速排序最初的版本
- 二、快速排序名字的由来
- 三、Hoare 版本的具体分析
- 四、快速排序的优化版本
- 五、快速排序的深入分析
- 六、Hoare 版本与优化后版本的比较
- 七、快速排序算法的时间复杂度
- 八、由快速排序所想到的

前言

之前，曾在本 BLOG 内写过一篇文章，十二、[快速排序算法](#)，不少网友反映此文好懂。然，后来有网友 [algorithm__](#)，指出，“快速排序算法怎么一步一步想到的列？就如一个 P 与 NP 问题。知道了解，证明不难。可不知道解之前，要一点一点、一步一步推导出来，好难阿？”

其实，这个问题，我也想过很多次了。之前，也曾在博客里多次提到过。那么，到底为什么，有不少人看了我写的快速排序算法，过了一段时间后，又不清楚快排是怎么一回事了？

以下是我十、[从头到尾彻底理解傅里叶变换算法、下](#)，一文里回复 [algorithm__](#) 的评论：

“很大一部分原因，就是只知其表，不知其里，只知其用，不知其本质。很多东西，都是可以从本质看本质的。而大部分人没有做到这一点。从而看了又忘，忘了再看，如此，在对知识的一次一次重复记忆中，始终未能透析本质，从而，形成不好的循环。

所以，归根究底，学一个东西，不但要运用自如，还要通晓其原理，来龙去脉与本质。正如侯捷先生所言，只知一个东西的用法，却不知其原理，实在不算高明。你提出的问题，非常好。我会再写一篇文章，彻底阐述快速排序算法是如何设计的，以及怎么一步一步来的。”

ok，那么现在，我就来彻底分析下此快速排序算法，希望能读者真正理解此算法，通晓其来龙去脉，明白其内部原理。本文着重分析快速排序算法的过程来源及其时间复杂度，要了解什么是快速排序算法，请参考此文：[精通八大排序算法系列：一、快速排序算法](#)。

一、快速排序最初的版本

快速排序的算法思想(此时，还不叫做快速排序)最初是由，一个名叫 [R.Sedgewick](#) 提出的，他的算法思想为：

I、 取两个指针 i, j ，开始时， $i=2, j=n$ ，且我们确定，最终完成排序时，左边子序列的数 \leq 右边子序列。

II、 纠正位置，不断交换

[i-->\(i 从左至右，右移，找比第一个元素要大的\)](#)

通过比较 k_i 与 k_1 ，如果 R_i 在分区之后最终要成为左边子序列的一部分，则 $i++$ ，且不断 $i++$ ，直到遇到一个该属于右边子序列 R_i (较大)为止。

[--j\(i 从右至左，左移，找比第一个元素要小的\)](#)

类似的， $j--$ ，直到遇到一个该属于左边子序列的较小元素 R_j (较小)为止，如此，当 $i < j$ 时，交换 R_i 与 R_j ，即摆正位置嘛，把较小的放左边，较大的放右边。

III、然后以同样的方式处理划分的记录，直到 i 与 j 碰头为止。这样，不断的通过交换 R_i , R_j 摆正位置，最终完成整个序列的排序。

举个例子，如下(2 为主元)：

$i \rightarrow$ $<-j$ (找小)

2 8 7 1 3 5 6 4

j 所指元素 4，大于 2，所以， $j--$,

i $<-j$

2 8 7 1 3 5 6 4

此过程中，若 j 没有遇到比 2 小的元素，则 j 不断--，直到 j 指向了 1，

i j

2 8 7 1 3 5 6 4

此刻，8 与 1 互换，

i j

2 1 7 8 3 5 6 4

此刻， i 所指元素 1，小于 2，所以 i 不动， j 继续--，始终没有遇到再比 2 小的元素，最终停至 7。

i j

2 1 7 8 3 5 6 4

最后， i 所指元素 1，与数列中第一个元素 $k1$ ，即 2 交换，

i

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

这样，2 就把整个序列，排成了 2 个部分，接下来，再对剩余待排序的数递归进行第二趟、第三趟排序....。

由以上的过程，还是可以大致看出此算法的拙陋之处的。如一，在上述第一步过程中， j 没有找到比 2 小的元素时，需不断的前移， $j--$ 。二，当 i 所指元素交换后，无法确保此刻 i 所指的元素就一定小于 2，即 i 指针，还有可能继续停留在原处，不能移动。如此停留，势必应该会耗费不少时间。

二、快速排序名字的由来

后来，**C.A.R.Hoare** 根据上述方案，因其排序速率较快，便称之为“快速排序”，快速排序也因此而诞生了。并最终一举成名，成为了**二十世纪最伟大的 10 大算法**之一。

他给出的算法思想描述的具体版本，如下：

HOARE-PARTITION(A, p, r)

```
1 x ← A[p]
2 i ← p - 1
3 j ← r + 1
4 while TRUE
5   do repeat j ← j - 1
6     until A[j] ≤ x
7   repeat i ← i + 1
8     until A[i] ≥ x
9   if i < j
10    then exchange A[i] ↔ A[j]
11 else return j
```

此程序的原理，与上文中 **R.Sedgewick** 的思想，已明显不同。确切的说，是有了不小的完善。后来，此版本又有不少的类似变种。下面，会具体分析。

三、Hoare 版本的具体分析

在上面，我们已经知道，**Hoare** 的快速排序版本可以通过前后俩个指针，分别指向首尾，分别比较而进行排序。

下面，分析一下此版本，或其它变种问题：

I、俩个指针， i 指向序列的首部， j 指着尾部，即 $i=1, j=n$ ，取数组中第一个元素 k_i 为主元，即 $key \leftarrow k_i$ (赋值)。

II、赋值操作（注，以下“->”，表示的是赋值）：

j (找小)，从右至左，不断 $--$ ，直到遇到第一个比 key 小的元素 k_j ， $k_i \leftarrow k_j$ 。

i (找大)，从左至右，不断 $++$ ，直到遇到第一个比 key 大的元素 k_i ， $k_j \leftarrow k_i$ 。

III、按上述方式不断进行，直到 i, j 碰头， $k_i = key$ ，第一趟排序完成接下来重复 II 步骤，递归进行。

再举一个例子：对序列 3 8 7 1 2 5 6 4，进行排序：

1、 $j--$ ，直到遇到了序列中第一个比 key 值 3 小的元素 2，把 2 赋给 k_i ， j 此刻指向了空元素。

i j
3 8 7 1 2 5 6 4
i j
=> 2 8 7 1 5 6 4

2、 $i++$ ，指向 8，把 8 重置给 j 所指元素空白处， i 所指元素又为空：

i j
2 8 7 1 5 6 4
i j
=> 2 7 1 8 5 6 4

3、 j 继续 $--$ ，遇到了 1，还是比 3(事先保存的 key 值)小，1 赋给 i 所指空白处：

i j
2 7 1 8 5 6 4
=> 2 1 7 8 5 6 4

4、同理， i 又继续 $++$ ，遇到了 7，比 key 大，7 赋给 j 所指空白处，此后， i , j 碰头。

第一趟结束：

i j
2 1 7 8 5 6 4
i j
=> 2 1 7 8 5 6 4

5、最后，事先保存的 key，即 3 赋给 k_i ，即 i 所指空白处，得：

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

所以，整趟下来，便是这样：

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

后续补充：

如果待排序的序列是逆序数列列？ok，为了说明的在清楚点，再举个例子，对序列 9 8 7 6 5 4 3 2 1 排序：

9 8 7 6 5 4 3 2 1 //9 为主元

1 8 7 6 5 4 3 2 //j 从右向左找小，找到 1，赋给第一个

1 8 7 6 5 4 3 2 //i 从左向右找大，没有找到，直到与 j 碰头

1 8 7 6 5 4 3 2 9 //最后，填上 9.

如上，当数组已经是逆序排列好的话，我们很容易就能知道，此时数组排序需要 $O(N^2)$ 的时间复杂度。稍后下文，会具体分析快速排序的时间复杂度。

最后，写程序实现此算法，如下，相信，不用我过多解释了：

```
int partition(int data[],int lo,int hi) //引自 whatever。
{
    int key=data[lo];
    int l=lo;
    int h=hi;
    while(l<h)
    {
        while(key<=data[h] && l<h) h--; //高位找小，找到了，就把它弄到前面去
        data[l]=data[h];
        while(data[l]<=key && l<h) l++; //低位找大，找到了，就把它弄到后面去
        data[h]=data[l];
    }
    data[l]=key;
    return l;
}
```

后续补充：举个例子，如下（只说明了第一趟排序）：

3 8 7 1 2 5 6 4

2 8 7 1 5 6 4

2 7 1 8 5 6 4

2 1 7 8 5 6 4

2 1 7 8 5 6 4

2 1 3 7 8 5 6 4 //最后补上, 关键字 3

看到这, 不知各位读者, 有没有想到我上一篇文章里头, 一、[快速排序算法](#), 的那快速排序的第二个版本?对, 上述程序, 即那篇文章里头的第二个版本。我把程序揪出来, 你一看, 就明白了:

```
void quicksort(table *tab,int left,int right)
{
    int i,j;
    if(left<right)
    {
        i=left;j=right;
        tab->r[0]=tab->r[i]; //准备以本次最左边的元素值为标准进行划分, 先保存其值
        do
        {
            while(tab->r[j].key>tab->r[0].key&&i<j)
                j--; //从右向左找第 1 个小于标准值的位置 j
            if(i<j) //找到了, 位置为 j
            {
                tab->r[i].key=tab->r[j].key;i++;
            } //将第 j 个元素置于左端并重置 i
            while(tab->r[i].key<tab->r[0].key&&i<j)
                i++; //从左向右找第 1 个大于标准值的位置 i
            if(i<j) //找到了, 位置为 i
            {
                tab->r[j].key=tab->r[i].key;j--;
            } //将第 i 个元素置于右端并重置 j
        }while(i!=j);

        tab->r[i]=tab->r[0]; //将标准值放入它的最终位置,本次划分结束
        quicksort(tab,left,i-1); //对标准值左半部递归调用本函数
        quicksort(tab,i+1,right); //对标准值右半部递归调用本函数
    }
}
```

我想，至此，已经讲的足够明白了。如果，你还不懂的话，好吧，伸出你的手指，数数吧....ok，再问读者一个问题：像这种 i 从左至右找大，找到第一个比 key 大(数组中第一个元素置为 key)，便重置 kj ， j 从右向左找小，找到第一个比 key 小的元素，则重置 ki ，当然此过程中， i ， j 都在不断的变化，通过 $++$ ，或 $--$ ，指向着不同的元素。

你是否联想到了，现实生活中，有什么样的情形，与此快速排序算法思想相似?ok，等你想到了，再告诉我，亦不迟。:D。

四、快速排序的优化版本

再到后来，**N.Lomuto** 又提出了一种新的版本，此版本即为本文**快速排序算法**，中阐述的第一个版本，即优化了 **PARTITION** 程序，它现在写在了 **算法导论** 一书上，

快速排序算法的关键是 **PARTITION** 过程，它对 $A[p..r]$ 进行就地重排：

```
PARTITION(A, p, r)
1 x ← A[r]      //以最后一个元素, A[r]为主元
2 i ← p - 1
3 for j ← p to r - 1  //注, j 从 p 指向的是 r-1, 不是 r。
4   do if A[j] ≤ x
5     then i ← i + 1
6     exchange A[i] <-> A[j]
7 exchange A[i + 1] <-> A[r]
8 return i + 1
```

然后，对整个数组进行递归排序：

```
QUICKSORT(A, p, r)
1 if p < r
2  then q ← PARTITION(A, p, r)  //关键
3  QUICKSORT(A, p, q - 1)
4  QUICKSORT(A, q + 1, r)
```

举最开头的那个例子：2 8 7 1 3 5 6 4，不过与上不同的是：它不再以第一个元素为主元，而是以最后一个元素 4 为主元，且 i , j 俩指针都从头出发， j 一前， i 一后。 i 指元素的前一个位置， j 指待排序数列中的第一个元素。

一、

i p/j->

2 8 7 1 3 5 6 4(主元)

j 指的 $2 \leq i < r$, 于是 $i++$, i 也指到 2, 2 和 2 互换, 原数组不变。

j 后移, 直到指向 1..

二、

j (指向 1) $\leq i < r$, 于是 $i++$, i 指向了 8,

i j
2 8 7 1 3 5 6 4

所以 8 与 1 交换, 数组变成了:

i j
2 1 7 8 3 5 6 4

三、j 后移, 指向了 3, $3 \leq i < r$, 于是 $i++$

i 这时指向了 7,

i j
2 1 7 8 3 5 6 4

于是 7 与 3 交换, 数组变成了:

i j
2 1 3 8 7 5 6 4

四、j 继续后移, 发现没有再比 4 小的数, 所以, 执行到了最后一步,

即上述 PARTITION(A, p, r) 代码部分的 第 7 行。

因此, i 后移一个单位, 指向了 8

i j
2 1 3 8 7 5 6 4

$A[i + 1] \leftrightarrow A[r]$, 即 8 与 4 交换, 所以, 数组最终变成了如下形式,

2 1 3 4 7 5 6 8

ok, 快速排序第一趟完成。接下来的过程, 略, 详细, 可参考此文: [快速排序算法](#)。不过, 有个问题, 你能发现此版本与上述版本的优化之处么?

五、快速排序的深入分析

咱们，再具体分析下上述的优化版本，

PARTITION(A, p, r)

```
1 x ← A[r]
2 i ← p - 1
3 for j ← p to r - 1
4   do if A[j] ≤ x
5     then i ← i + 1
6     exchange A[i] <-> A[j]
7 exchange A[i + 1] <-> A[r]
8 return i + 1
```

咱们以下数组进行排序，每一步的变化过程是：

i p/j
2 8 7 1 3 5 6 4(主元)

i j
2 1 7 8 3 5 6 4

i j
2 1 3 8 7 5 6 4

i
2 1 3 4 7 5 6 8

由上述过程，可看出，j 扫描了整个数组一遍，只要一旦遇到比 4 小的元素，i 就++，然后，kj、ki 交换。那么，为什么当 j 找到比 4 小的元素后，i 要++列？你想嘛，如果 i 始终停在原地不动，与 kj 每次交换的 ki 不就是同一个元素了么？如此，还谈什么排序？。

所以，j 在前面开路，i 跟在 j 后，j 只要遇到比 4 小的元素，i 就向前前进一步，然后把 j 找到的比 4 小的元素，赋给 i，然后，j 才再前进。

打个比喻就是，你可以这么认为，i 所经过的每一步，都必须是比 4 小的元素，否则，i 就不能继续前行。好比 j 是先行者，为 i 开路搭桥，把小的元素作为跳板放到 i 跟前，为其铺路前行啊。

于此， j 扫描到最后，也已经完全排查出了比 4 小的元素，只有最后一个主元 4，则交给 i 处理，因为最后一步， $\text{exchange } A[i + 1] \leftrightarrow A[r]$ 。这样，不但完全确保了只要是比 4 小的元素，都被交换到了数组的前面，且 j 之前未处理的比较大的元素则被交换到了后面，而且还是 $O(N)$ 的时间复杂度，你不得不佩服此算法设计的巧妙。

这样，我就有一个问题了，上述的 $\text{PARTITION}(A, p, r)$ 版本，可不可以改成这样咧？
望读者思考：

$\text{PARTITION}(A, p, r)$ //请读者思考版本。

```
1 x ← A[r]
2 i ← p - 1
3 for j ← p to r    //让 j 指向了最后一个元素 r
4     do if A[j] ≤ x
5         then i ← i + 1
6         exchange A[i] <-> A[j]
//7 exchange A[i + 1] <-> A[r] 去掉此最后的步骤
8 return i    //返回 i，不再返回 i+1.
```

六、Hoare 版本与优化后版本的比较

现在，咱们来讨论一个问题，快速排序中，其中对于序列的划分，我们可以看到，已经有以上俩个版本，那么这俩个版本孰优孰劣呢？OK，不急，咱们来比较下：

为了看着方便，再贴一下各自的算法，

Hoare 版本：

$\text{HOARE-PARTITION}(A, p, r)$

```
1 x ← A[p]    //以第一个元素为主元
2 i ← p - 1
3 j ← r + 1
4 while TRUE
5     do repeat j ← j - 1
6         until A[j] ≤ x
7         repeat i ← i + 1
8             until A[i] ≥ x
9         if i < j
```

```
10      then exchange A[i] ↔ A[j]
11      else return j
```

优化后的算法导论上的版本：

```
PARTITION(A, p, r)
1 x ← A[r]      //以最后一个元素, A[r]为主元
2 i ← p - 1
3 for j ← p to r - 1
4   do if A[j] ≤ x
5     then i ← i + 1
6   exchange A[i] <-> A[j]
7 exchange A[i + 1] <-> A[r]
8 return i + 1
```

咱们，先举上述说明 Hoare 版本的这个例子，对序列 3 8 7 1 2 5 6 4，进行排序：

Hoare 版本（以 3 为主元，**红体**为主元）：

```
3 8 7 1 2 5 6 4
2 8 7 1 5 6 4 //交换 1 次，比较 4 次
2 7 1 8 5 6 4 //交换 1 次，比较 1 次
2 1 7 8 5 6 4 //交换 1 次，比较 1 次
2 1 7 8 5 6 4 //交换 1 次，比较 0 次
2 1 3 7 8 5 6 4 //总计交换 4 次，比较 6 次。
//移动了元素 3、8、7、1、2. 移动范围为：2+3+1+2+4=12.
```

优化版本（以 4 为主元）：

```
3 8 7 1 2 5 6 4 //3 与 4 交换，不用移动元素，比较 1 次
3 1 7 8 2 5 6 4 //交换 1 次，比较 3 次
3 1 2 8 7 5 6 4 //交换 1 次，比较 1 次
3 1 2 4 7 5 6 8 //交换 1 次，比较 2 次。
//即完成，总计交换 4 次，比较 7 次。
//移动了元素 8、7、1、2、4. 移动范围为：6+2+2+2+4=16.
```

再举一个例子：对序列 2 8 7 1 3 5 6 4 排序：

Hoare 版本:

```
2 8 7 1 3 5 6 4
1 8 7 3 5 6 4 //交换 1 次，比较 5 次
1 7 8 3 5 6 4 //交换 1 次，比较 1 次
1 2 7 8 3 5 6 4 //交换 0 次，比较 1 次。2 填上，完成，总计交换 2 次，比较 7 次。
```

优化版本:

```
2 8 7 1 3 5 6 4 //2 与 2 交换，比较 1 次
2 1 7 8 3 5 6 4 //交换 1 次，比较 3 次
2 1 3 8 7 5 6 4 //交换 1 次，比较 1 次
2 1 3 4 7 5 6 8 //交换 1 次，比较 2 次。完成，总计交换 4 次，比较 7 次。
```

各位，已经看出来了，这俩个例子说明不了任何问题。到底哪个版本效率更高，还有待进一步验证或者数学证明。ok，等我日后再发现更有利的证据再来论证下。

七、快速排序算法的时间复杂度

ok，我想你已经完全理解了此快速排序，那么，我想你应该也能很快的判断出：快速排序算法的平均时间复杂度，即为 $O(n \lg n)$ 。为什么列?因为你看， j, i 扫描一遍数组，花费用时多少?对了，扫描一遍，当然是 $O(n)$ 了，那样，扫描多少遍列， $\lg n$ 到 n 遍，最快 $\lg n$ ，最慢 n 遍。且可证得，**快速排序的平均时间复杂度即为 $O(n^* \lg n)$** 。

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 $\lceil n/2 \rceil - 1$.

在这种情况下，快速排序的速度要快得多。为：

$$T(n) \leq 2T(n/2) + O(n) . \text{可以证得， } T(n) = O(n \lg n) .$$

以下给出一个递归树的简单证明：

在分治算法中的三个步骤中， 我们假设分解和合并过程所用的时间分别为 $D(n), C(n)$ ，设 $T(n)$ 为处理一个规模为 n 的序列所消耗的时间为子序列个数，每一个子序列是原序列的 $1/b$ ， a 为把每个问题分解成 a 个子问题，则所消耗的时间为：

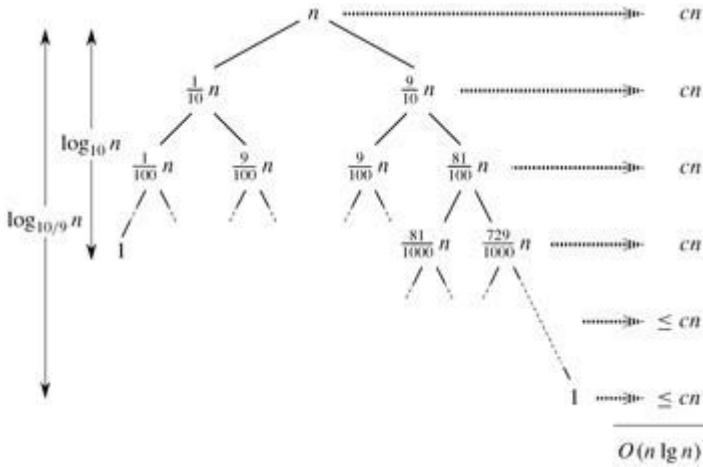
$$O(1) \quad \text{如果 } n \leq c$$

$$T(n) =$$

$$aT(n/b) + D(n) + C(n)$$

在快速排序中, α 是为 2 的, β 也为 2, 则分解(就是取参照点, 可以认为是 1), 合并(把数组合并, 为 n), 因此 $D(n) + C(n)$ 是一个线性时间 $O(n)$. 这样时间就变成了:

$$T(n) = 2T(n/2) + O(n).$$



如上图所示, 在每个层的时间复杂度为: $O(n)$, 一共有 $\lg n$ 层, 每一层上都是 cn , 所以共消耗时间为 $cn \cdot \lg n$; 则总时间:

$$cn \cdot \lg 2n + cn = cn(1 + \lg n) \text{ 即 } O(n \lg n).$$

关于 $T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$ 的严格数学证明, 可参考[算法导论 第四章 递归式](#)。

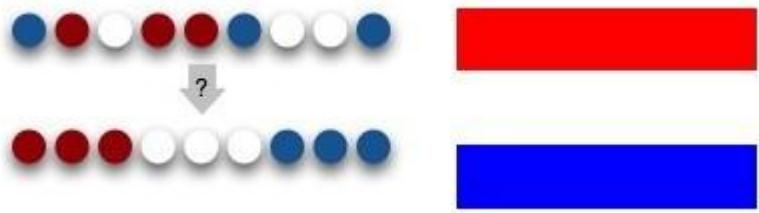
而后, 我想问读者一个问题, 由此快速排序算法的时间复杂度, 你想到了什么, 是否想到了归并排序的时间复杂度, 是否想到了二分查找, 是否想到了一棵 n 个结点的红黑树的高度 $\lg n$, 是否想到了.....

八、由快速排序所想到的

上述提到的东西, 很早以前就想过了。此刻, 我倒是想到了前几天看到的一个荷兰国旗问题。当时, 和 `algorithm__`、`gnuhpc` 简单讨论过这个问题。现在, 我也来具体解决下此问题:

问题描述:

我们将乱序的红白蓝三色小球排列成有序的红白蓝三色的同颜色在一起的小球组。这个问题之所以叫[荷兰国旗](#), 是因为我们可以将红白蓝三色小球想象成条状物, 有序排列后正好组成荷兰国旗。如下图所示:



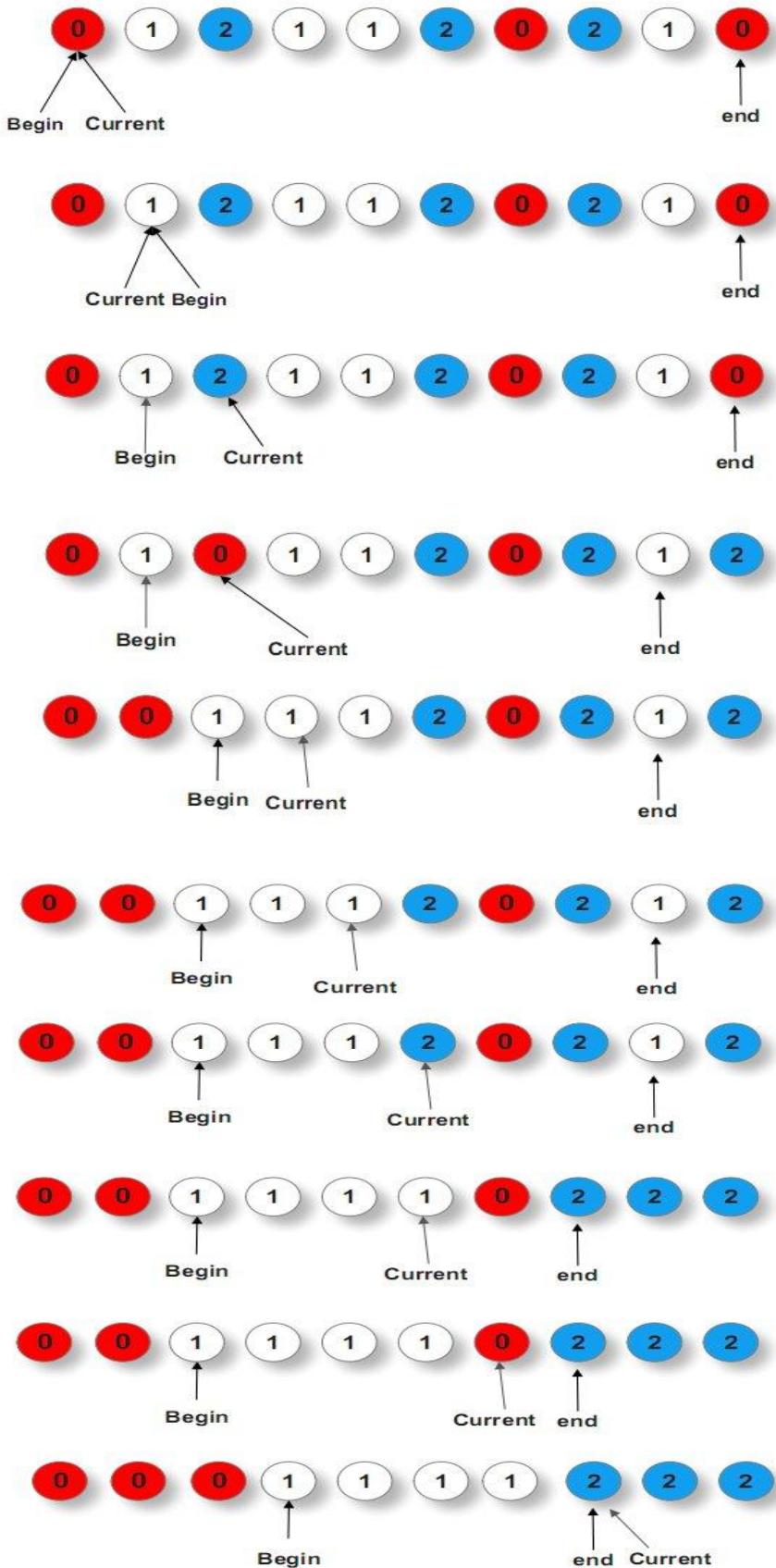
这个问题，类似快排中 `partition` 过程。不过，要用三个指针，一前 `begin`，一中 `current`，一后 `end`，俩俩交换。

- 1、`current` 遍历，整个数组序列，`current` 指 1 不动，
- 2、`current` 指 0，与 `begin` 交换，而后 `current++`, `begin++`,
- 3、`current` 指 2，与 `end` 交换，而后，`current` 不动，`end--`。

为什么，第三步，`current` 指 2，与 `end` 交换之后，`current` 不动了列，对的，正如 `algorithm_` 所说：`current` 之所以与 `begin` 交换后，`current++`、`begin++`，是因为此无后顾之忧。而 `current` 与 `end` 交换后，`current` 不动，`end--`，是因有后顾之忧。

为什么啊，因为你想想啊，你最终的目的无非就是为了让 0、1、2 有序排列，试想，如果第三步，`current` 与 `end` 交换之前，万一 `end` 之前指的是 0，而 `current` 交换之后，`current` 此刻指的是 0 了，此时，`current` 能动么？不能动啊，指的是 0，还得与 `begin` 交换列。

ok，说这么多，你可能不甚明了，直接引用下 `gnuhpc` 的图，就一目了然了：



本程序主体的代码是：

```

//引用自 gnuhpc
while( current<=end )
{
    if( array[current] ==0 )
    {
        swap(array[current],array[begin]);
        current++;
        begin++;
    }
    else if( array[current] == 1 )
    {
        current++;
    }

    else //When array[current] =2
    {
        swap(array[current],array[end]);
        end--;
    }
}

```

看似，此问题与本文关系不大，但是，一来因其余本文中快速排序 `partition` 的过程类似，二来因为此问题引发了一段小小的思考，并最终成就了本文。差点忘了，还没回答本文开头提出的问题。**所以，快速排序算法是如何想到的，如何一步一步发明的列？答案很简单：多观察，多思考。**

ok，测试一下，看看你平时有没有多观察、多思考的习惯：我不知道是否有人真正思考过冒泡排序，如果思考过了，你是否想过，怎样改进冒泡排序？ok，其它的，我就不多说了，只贴以下这张图：

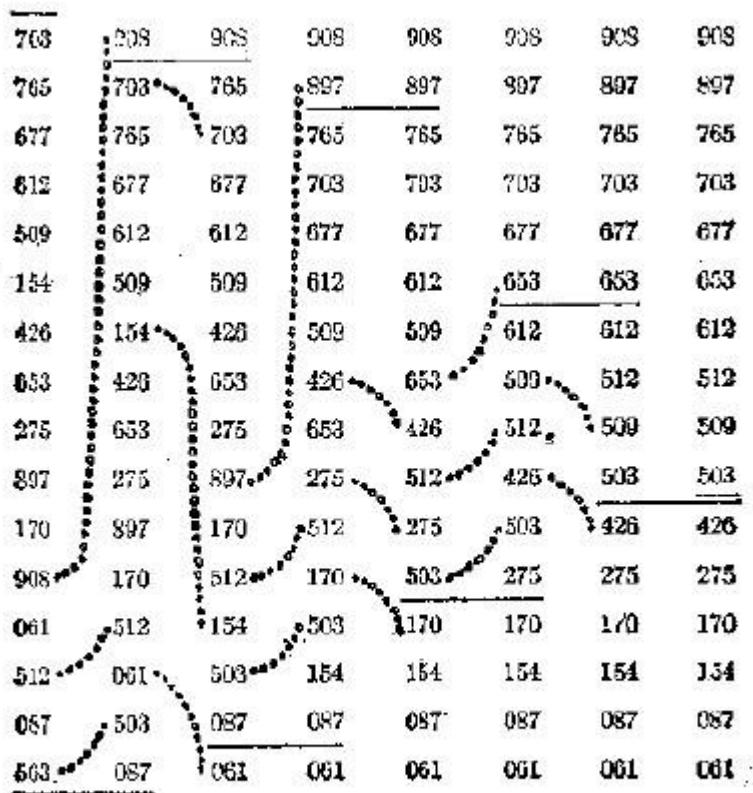


图16 “鸡尾混合排序”

更多，请参考：[十二、快速排序算法之所有版本的 c/c++实现](#)。本文完。

十二（再续）：快速排序算法之所有版本的 c/c++ 实现

作者: July、二零一一年三月三十日。

出处：http://blog.csdn.net/v_JULY_v

前言：

相信，经过本人之前写的前俩篇关于快速排序算法的文章：第一篇、[一、快速排序算法](#)，及第二篇、[一之续、快速排序算法的深入分析](#)，各位，已经对快速排序算法有了足够的了解与认识。但仅仅停留在对一个算法的认识层次上，显然是不够的，即便你认识的有多透彻与深入。最好是，编程实现它。

而网上，快速排序的各种写法层次不清，缺乏统一、整体的阐述与实现，即，没有个一锤定音，如此，我便打算自己去实现它了。

于是，今花了一个上午，把快速排序算法的各种版本全部都写程序一一实现了一下。包括网上有的，没的，算法导论上的，国内教材上通用的，随机化的，三数取中分割法的，递归的，非递归的，所有版本都用 **c/c++** 全部写了个遍。

鉴于时间仓促下，一个人考虑问题总有不周之处，以及水平有限等等，不正之处，还望各位不吝赐教。不过，以下，所有全部 **c/c++** 源码，都经本人一一调试，若有任何问题，恳请指正。

ok，本文主要分为以下几部分内容：

第一部分、递归版

一、算法导论上的单向扫描版本

二、国内教材双向扫描版

 2.1、Hoare 版本

 2.2、Hoare 的几个变形版本

三、随机化版本

四、三数取中分割法

第二部分、非递归版

好的，请一一细看。

第一部分、快速排序的递归版本

一、算法导论上的版本

在我写的第二篇文章中，我们已经知道：

“再到后来，N.Lomuto 又提出了一种新的版本，此版本....，即优化了 PARTITION 程序,它现在写在了 算法导论 一书上”：

快速排序算法的关键是 PARTITION 过程，它对 A[p..r] 进行就地重排：

```

PARTITION(A, p, r)
1 x ← A[r]      //以最后一个元素, A[r]为主元
2 i ← p - 1
3 for j ← p to r - 1  //注, j 从 p 指向的是 r-1, 不是 r。
4   do if A[j] ≤ x
5     then i ← i + 1
6     exchange A[i] <-> A[j]
7 exchange A[i + 1] <-> A[r]  //最后, 交换主元
8 return i + 1

```

然后, 对整个数组进行递归排序:

```

QUICKSORT(A, p, r)
1 if p < r
2  then q ← PARTITION(A, p, r)  //关键
3  QUICKSORT(A, p, q - 1)
4  QUICKSORT(A, q + 1, r)

```

根据上述伪代码, 我们不难写出以下的 c/c++ 程序:

首先是, PARTITION 过程:

```

int partition(int data[],int lo,int hi)
{
    int key=data[hi]; //以最后一个元素, data[hi]为主元
    int i=lo-1;
    for(int j=lo;j<hi;j++)  //注, j 从 p 指向的是 r-1, 不是 r。
    {
        if(data[j]<=key)
        {
            i=i+1;
            swap(&data[i],&data[j]);
        }
    }
    swap(&data[i+1],&data[hi]); //不能改为 swap(&data[i+1],&key)
    return i+1;
}

```

补充说明：举个例子，如下为第一趟排序（更多详尽的分析请参考第二篇文章）：

第一趟(4步)：

a: 3 8 7 1 2 5 6 4 //以最后一个元素，`data[hi]`为主元

b: 3 1 7 8 2 5 6 4

c: 3 1 2 8 7 5 6 4

d: 3 1 2 4 7 5 6 8 //最后，`swap(&data[i+1],&data[hi])`

而其中 `swap` 函数的编写，是足够简单的：

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

然后是，调用 `partition`，对整个数组进行递归排序：

```
void QuickSort(int data[], int lo, int hi)
{
    if (lo<hi)
    {
        int k = partition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}
```

现在，我有一个问题要问各位了，保持其它的不变，稍微修改一下上述的 `partition` 过程，如下：

```
int partition(int data[],int lo,int hi) //请读者思考
{
    int key=data[hi]; //以最后一个元素，data[hi]为主元
    int i=lo-1;
```

```

for(int j=lo;j<=hi;j++) //现在，我让 j 从 lo 指向了 hi，不是 hi-1。
{
    if(data[j]<=key)
    {
        i=i+1;
        swap(&data[i],&data[j]);
    }
}
//swap(&data[i+1],&data[hi]); //去掉这行
return i; //返回 i，非 i+1.
}

```

如上，其它的不变，请问，让 j 扫描到了最后一个元素，后与 $\text{data}[i+1]$ 交换，去掉最后的 $\text{swap}(\&\text{data}[i+1], \&\text{data}[hi])$ ，然后，再返回 i 。请问，如此，是否可行？

其实这个问题，就是我第二篇文章中，所提到的：

“上述的 PARTITION(A, p, r) 版本，可不可以改成这样咧？以下这样列”：

PARTITION(A, p, r) //请读者思考版本。

```

1 x ← A[r]
2 i ← p - 1
3 for j ← p to r //让 j 从 p 指向了最后一个元素 r
4     do if A[j] ≤ x
5         then i ← i + 1
6             exchange A[i] <-> A[j]
//7 exchange A[i + 1] <-> A[r] 去掉此最后的步骤
8 return i //返回 i，不再返回 i+1.

```

望读者思考，后把结果在评论里告知我。

我这里简单论述下：上述请读者思考版本，只是代码做了以下三处修改而已：**1、 j 从 $p \rightarrow r$ ；2、去掉最后的交换步骤；3、返回 i** 。首先，无论是我的版本，还是算法导论上的原装版本，都是准确无误的，且我都已经编写程序测试通过了。但，其实这两种写法，思路是完全一致的。

为什么这么说呢？请具体看以下的请读者思考版本，

```

int partition(int data[],int lo,int hi) //请读者思考
{
    int key=data[hi]; //以最后一个元素， data[hi]为主元
    int i=lo-1;
    for(int j=lo;j<=hi;j++)
    {
        if(data[j]<=key) //如果让 j 从 lo 指向 hi, 那么当 j 指到 hi 时, 是一定会有 A[j]<=x 的
        {
            i=i+1;
            swap(&data[i],&data[j]);
        }
    }
    //swap(&data[i+1],&data[hi]); //事实是, 应该加上这句, 直接交换, 即可。
    return i;
}

```

我们知道当 j 最后指到了 r 之后，是一定会有 $A[j] \leq x$ 的（即 $=$ ），所以这个 `if` 判断就有点多余，没有意义。所以应该如算法导论上的版本那般，最后直接交换 `swap(&data[i+1],&data[hi]);` 即可，返回 $i+1$ 。所以，总体说来，算法导论上的版本那样写，比请读者思考版本更规范，更合乎情理。ok，请接着往下阅读。

当然，上述 `partition` 过程中，也可以去掉 `swap` 函数的调用，直接写在分割函数里：

```

int partition(int data[],int lo,int hi)
{
    int i,j,t;
    int key = data[hi]; //还是以最后一个元素作为哨兵，即主元元素
    i = lo-1;
    for (j = lo;j<=hi;j++)
        if(data[j]<key)
        {
            i++;
            t = data[j];
            data[j] = data[i];
            data[i] = t;
        }
}

```

```

    data[i] = t;
}
data[hi] = data[i+1]; //先,data[i+1]赋给 data[hi]
data[i+1] = key; //后, 把事先保存的 key 值, 即 data[hi]赋给 data[i+1]
//不可调换这两条语句的顺序。
return i+1;
}

```

提醒:

- 1、程序中尽量不要有任何多余的代码。
- 2、你最好绝对清楚的知道，程序的某一步，是该用 **if**，还是该用 **while**，等任何细节的东西。

ok，以上程序的测试用例，可以简单编写如下：

```

int main()
{
    int a[8]={3,8,7,1,2,5,6,4};
    QuickSort(a,0,N-1);
    for(int i=0;i<8;i++)
        cout<<a[i]<<endl;
    return 0;
}

```

当然，如果，你如果对以上的测试用例不够放心，可以采取 1~10000 的随机数进行极限测试，保证程序的万无一失（主函数中测试用的随机数例子，即所谓的“**极限**”测试，下文会给出）。

至于上述程序是什么结果，相信，不用我再啰嗦。:D。

补充一种写法：

```

void quickSort(int p, int q)
{
    if(p < q)
    {
        int x = a[p]; //以第一个元素为主元
        int i = p;
        for(int j = p+1; j < q; j++)

```

```

{
    if(a[j] < x)
    {
        i++;
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}

int temp = a[p];
a[p] = a[i];
a[i] = temp;
quickSort(p, i);
quickSort(i+1, q);
}
}

```

二、国内教材双向扫描版

国内教材上一般所用的通用版本，是我写的第二篇文章中所提到的霍尔排序或其变形，[而非上述所述的算法导论上的版本](#)。而且，现在网上一般的朋友，也是更倾向于采用此种思路来实现快速排序算法。ok，请看：

2.1、Hoare 版本

那么，什么是霍尔提出的快速排序版本列？如下，即是：

HOARE-PARTITION(A, p, r)

```

1 x ← A[p]
2 i ← p - 1
3 j ← r + 1
4 while TRUE
5   do repeat j ← j - 1
6     until A[j] ≤ x
7   repeat i ← i + 1
8     until A[i] ≥ x
9   if i < j

```

```
10      then exchange A[i] <-> A[j]
11      else return j
```

同样，根据以上伪代码，不难写出以下的 c/c++ 代码：

```
1. //此处原来的代码有几点错误，后听从了 Joshua 的建议，现修改如下：
2. int partition(int data[],int lo,int hi) //。
3. {
4.     int key=data[lo];
5.     int l=lo-1;
6.     int h=hi+1;
7.     for(;;)
8.     {
9.         do{
10.             h--;
11.         }while(data[h]>key);
12.
13.         do{
14.             l++;
15.         }while(data[l]<key);
16.
17.         if(l<h)
18.         {
19.             swap(data[l],data[h]);
20.         }
21.         else
22.         {
23.             return h;
24.         }
25.     }
26. }
27. }
```

或者原来的代码改成这样（已经过测试，有误）：

```
int partition(int data[],int lo,int hi) //。
{
    int key=data[lo];
    int l=lo;
    int h=hi;
    for(;;)
    {
        while(data[h]>key) //不能加“=”
```

```

    h--;
    while(data[l]<key) //不能加“-”
    ++
    if(l<h)
    {
        swap(data[l],data[h]);
    }
    else
    {
        return h; //各位注意了，这里的返回值是 h。不是返回各位习以为常的枢纽元素，即不是
        //之之类的。
    }
}
//这个版本，已经证明有误，因为当 data[l] == data[h] == key 的时候，将会进
入死循环，所以淘汰。因此，使用上面的 do-while 形式吧。

```

读者可以试下，对这个序列进行排序，用上述淘汰版本将立马进入死循环：int data[16]={ 1000, 0, 6, 5, 4, 3, 2, 1, 7, 156, 44, 23, 123, 11, 5 }。

或者，如朋友颜沙所说：

如果 `data` 数组有相同元素就可能陷入死循环，比如：

2 3 4 5 6 2
| -> | <-h

交换 `l` 和 `h` 单元后重新又回到：

2 3 4 5 6 2
| -> | <-h

而第一个程序不存在这种情况，因为它总是在 `l` 和 `h` 调整后比较，也就是 `l` 终究会大于等于 `h`。

相信，你已经看出来了，上述的第一个程序中 `partition` 过程的返回值 `h` 并不是枢纽元的位置，但是仍然保证了 `A[p..j] <= A[j+1..q]`。

这种方法在效率上与以下将要介绍的 `Hoare` 的几个变形版本差别甚微，只不过是上述代码相对更为紧凑点而已。

2.2、Hoare 的几个变形版本

ok，可能，你对上述的最初的霍尔排序 partition 过程，理解比较费力，没关系，我再写几种变形，相信，你立马就能了解此双向扫描是怎么一回事了。

```
int partition(int data[],int lo,int hi) //双向扫描。
```

```
{  
    int key=data[lo]; //以第一个元素为主元  
    int l=lo;  
    int h=hi;  
    while(l<h)  
    {  
        while(key<=data[h] && l<h)  
            h--;  
        data[l]=data[h];  
        while(data[l]<=key && l<h)  
            l++;  
        data[h]=data[l];  
    }  
    data[l]=key; //1.key。只有出现要赋值的情况，才事先保存好第一个元素的值。  
    return l; //这里和以下所有的 Hoare 的变形版本都是返回的是枢纽元素，即主元元素 l。  
}
```

补充说明：同样，还是举上述那个例子，如下为第一趟排序（更多详尽的分析请参考第二篇文章）：

第一趟(五步曲):

```
a: 3 8 7 1 2 5 6 4 //以第一个元素为主元  
      2 8 7 1 5 6 4  
b: 2 7 1 8 5 6 4  
c: 2 1 7 8 5 6 4  
d: 2 1 7 8 5 6 4  
e: 2 1 3 7 8 5 6 4 //最后补上，关键字 3
```

然后，对整个数组进行递归排序：

```
void QuickSort(int data[], int lo, int hi)  
{
```

```

if (lo<hi)
{
    int k = partition(data, lo, hi);
    QuickSort(data, lo, k-1);
    QuickSort(data, k+1, hi);
}
}

```

当然，你也可以这么写，把递归过程写在同一个排序过程里：

```

void QuickSort(int data[],int lo,int hi)
{
    int i,j,temp;
    temp=data[lo]; //还是以第一个元素为主元。
    i=lo;
    j=hi;
    if(lo>hi)
        return;
    while(i!=j)
    {
        while(data[j]>=temp && j>i)
            j--;
        if(j>i)
            data[i++]=data[j];
        while(data[i]<=temp && j>i)
            i++;
        if(j>i)
            data[j--]=data[i];
    }
    data[i]=temp; //2.temp。同上，返回的是枢纽元素，即主元元素。
    QuickSort(data,lo,i-1); //递归左边
    QuickSort(data,i+1,hi); //递归右边
}

```

或者，如下：

```
1. void quicksort (int[] a, int lo, int hi)
```

```

2. {
3. // lo is the lower index, hi is the upper index
4. // of the region of array a that is to be sorted
5. int i=lo, j=hi, h;
6.
7. // comparison element x
8. int x=a[(lo+hi)/2];
9.
10. // partition
11. do
12. {
13.     while (a[i]<x) i++;
14.     while (a[j]>x) j--;
15.     if (i<=j)
16.     {
17.         h=a[i]; a[i]=a[j]; a[j]=h;
18.         i++; j--;
19.     }
20. } while (i<=j);
21.
22. // recursion
23. if (lo<j) quicksort(a, lo, j);
24. if (i<hi) quicksort(a, i, hi);
25. }

```

另，本人在一本国内的数据结构教材上（[注](#)，此处非指严那本），看到的一种写法，发现如下问题：一、冗余繁杂，二、错误之处无所不在，除了会犯一些注释上的错误，一些最基本的代码，都会弄错。详情，如下：

```

void QuickSort(int data[],int lo,int hi)
{
int i,j,key;
if(lo<hi)
{
    i=lo;
    j=hi;
    key=data[lo];
    //已经测试：原教材上，原句为“data[0]=data[lo];”，有误。
    //因为只能用一个临时变量 key 保存着主元，data[lo]，而若为以上，则相当于覆盖原元素
    //data[0]的值了。
    do

```

```

{
while(data[j]>=key&&i<j)
j--;
if(i<j)
{
    data[i]=data[j];
//i++; 这是教材上的语句，为使代码简洁，我特意去掉。
}
while(data[i]<=key&&i<j)
i++;
if(i<j)
{
    data[j]=data[i];
//j--; 这是教材上的语句，为使代码简洁，我特意去掉。
}
}while(i!=j);
data[i]=key; //3.key。

//已经测试：原教材上，原句为“data[i]=data[0];”，有误。
QuickSort(data,lo,i-1); //对标准值左半部递归调用本函数
QuickSort(data,i+1,hi); //对标准值右半部递归调用本函数
}
}

```

然后，你能很轻易的看到，这个写法，与上是同一写法，之所以写出来，是希望各位慎看国内的教材，多多质疑+思考，勿轻信。

ok，再给出一种[取中间元素为主元](#)的实现：

```

void QuickSort(int data[],int lo,int hi)
{
int pivot,l,r,temp;
l = lo;
r = hi;
pivot=data[(lo+hi)/2]; //取中位值作为分界值
while(l<r)
{

```

```

while(data[l]<pivot)
    ++l;
while(data[r]>pivot)
    --r;
if(l>=r)
    break;
temp = data[l];
data[l] = data[r];
data[r] = temp;
++l;
--r;
}
if(l==r)
    l++;
if(lo<r)
    QuickSort(data,lo,l-1);
if(l<hi)
    QuickSort(data,r+1,hi);
}

```

或者，这样写：

```

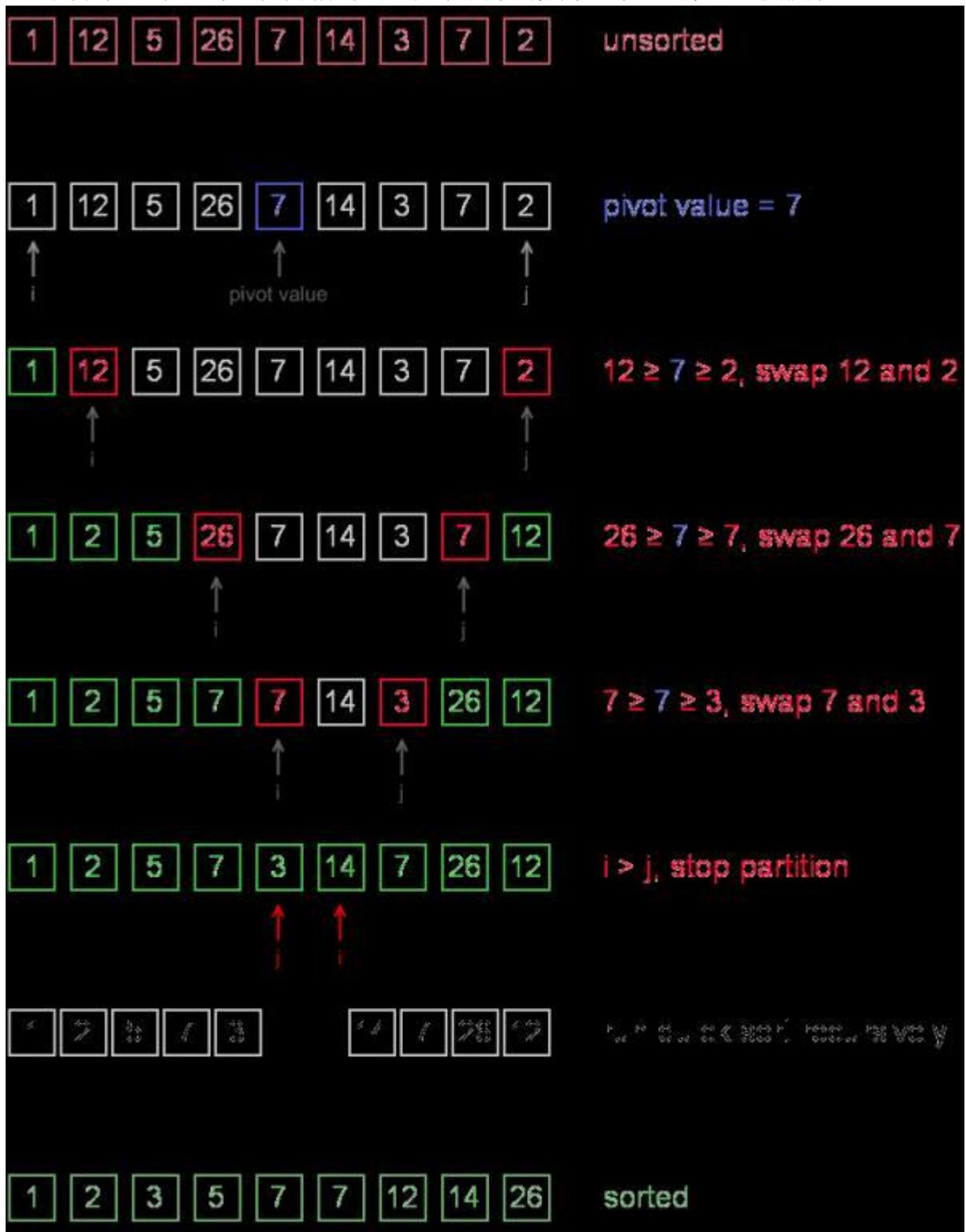
void quickSort(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2]; //取中间元素为主元

    /* partition */
    while (i <= j)
    {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j)

```

```
{  
    tmp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = tmp;  
    i++;  
    j--;  
}  
}  
}
```

上述演示过程，如下图所示（取中间元素为主元，第一趟排序）：



三、快速排序的随机化版本

以下是完整测试程序，由于给的注释够详尽了，就再做多余的解释了：

```
//交换两个元素值，咱们换一种方式，采取引用“&”
void swap(int& a , int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

//返回属于[lo,hi)的随机整数
int rand(int lo,int hi)
{
    int size = hi-lo+1;
    return lo+ rand()%size;
}

//分割，换一种方式，采取指针 a 指向数组中第一个元素
int RandPartition(int* data, int lo , int hi)
{
    //普通的分割方法和随机化分割方法的区别就在于下面三行
    swap(data[rand(lo,hi)], data[lo]);
    int key = data[lo];
    int i = lo;

    for(int j=lo+1; j<=hi; j++)
    {
        if(data[j]<=key)
        {
            i = i+1;
            swap(data[i], data[j]);
        }
    }
    swap(data[i],data[lo]);
}
```

```

return i;
}

//逐步分割排序
void RandQuickSortMid(int* data, int lo, int hi)
{
if(lo<hi)
{
int k = RandPartition(data,lo,hi);
RandQuickSortMid(data,lo,k-1);
RandQuickSortMid(data,k+1,hi);
}
}

int main()
{
const int N = 100; //此就是上文说所的“极限”测试。为了保证程序的准确无误，你也可以让
N=10000。
int *data = new int[N];
for(int i =0; i<N; i++)
data[i] = rand(); //同样，随机化的版本，采取随机输入。
for(i=0; i<N; i++)
cout<<data[i]<<" ";
RandQuickSortMid(data,0,N-1);
cout<<endl;
for(i=0; i<N; i++)
cout<<data[i]<<" ";
cout<<endl;
return 0;
}

```

四、三数取中分割法

我想，如果你爱思考，可能你已经在想一个问题了，那就是，像上面的程序版本，其中算法导论上采取单向扫描中，是以最后一个元素为枢纽元素，即主元，而在 Hoare 版本及其几个变形中，都是以第一个元素、或中间元素为主元，最后，上述给的快速排序算法的随

机化版本，则是以序列中任一个元素作为主元。

那么，枢纽元素的选取，即主元元素的选取是否决定快速排序最终的效率列？

答案是肯定的，当我们采取 `data[lo],data[mid],data[hi]` 三者之中的那个第二大的元素为主元时，便能尽最大限度保证快速排序算法不会出现 $O(N^2)$ 的最坏情况。这就是所谓的三数取中分割方法。当然，针对的还是那个 `Partition` 过程。

ok，直接写代码：

```
//三数取中分割方法
int RandPartition(int* a, int p , int q)
{
    //三数取中方法的关键就在于下述六行,
    int m=(p+q)/2;
    if(a[p]<a[m])
        swap(a[p],a[m]);
    if(a[q]<a[m])
        swap(a[q],a[m]);
    if(a[q]<a[p])
        swap(a[q],a[p]);
    int key = a[p];
    int i = p;

    for(int j = p+1; j <= q; j++)
    {
        if(a[j] <= key)
        {
            i = i+1;
            if(i != j)
                swap(a[i], a[j]);
        }
    }

    swap(a[i],a[p]);
    return i;
}
```

```

void QuickSort(int data[], int lo, int hi)
{
    if (lo<hi)
    {
        int k = RandPartition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}

```

经过测试，这种方法可行且有效，不过到底其性能、效率有多好，还有待日后进一步的测试。

第二部分、快速排序的非递归版

ok，相信，您已经看到，上述所有的快速排序算法，都是递归版本的，那还有什么办法可以实现此快速排序算法呢？对了，递归，与之相对的，就是非递归了。

以下，就是快速排序算法的非递归实现：

```

template <class T>
int RandPartition(T data[],int lo,int hi)
{
    T v=data[lo];
    while(lo<hi)
    {
        while(lo<hi && data[hi]>=v)
            hi--;
        data[lo]=data[hi];
        while(lo<hi && data[lo]<=v)
            lo++;
        data[hi]=data[lo];
    }
    data[lo]=v;
    return lo;
}

```

//快速排序的非递归算法

```
template <class T>
void QuickSort(T data[],int lo,int hi)
{
    stack<int> st;
    int key;
    do{
        while(lo<hi)
        {
            key=partition(data,lo,hi);
            //递归的本质是什么?对了，就是借助栈，进栈，出栈来实现的。
            if( (key-lo)<(key-key) )
            {
                st.push(key+1);
                st.push(hi);
                hi=key-1;
            }
            else
            {
                st.push(lo);
                st.push(key-1);
                lo=key+1;
            }
        }
        if(st.empty())
            return;
        hi=st.top();
        st.pop();
        lo=st.top();
        st.pop();
    }while(1);
}
```

```
void QuickSort(int data[], int lo, int hi)
```

```
{  
    if (lo<hi)
```

```

{
    int k = RandPartition(data, lo, hi);
    QuickSort(data, lo, k-1);
    QuickSort(data, k+1, hi);
}
}

```

如果你还尚不知道快速排序算法的原理与算法思想，请参考本人写的关于快速排序算法的前俩篇文章：[一之续、快速排序算法的深入分析](#)，及[一、快速排序算法](#)。如果您看完了此篇文章后，还是不知如何从头实现快速排序算法，那么好吧，伸出手指，数数，1,2,3,4,5....数到 100 之后，再来看此文。

据本文评论里头网友 ybt631 的建议，表示非常感谢，并补充阐述下所谓的**并行快速排序**：

Intel Threading Building Blocks(简称 TBB)是一个 C++的并行编程模板库，它能使你的程序充分利用多核 CPU 的性能优势，方便使用，效率很高。

以下是，**parallel_sort.h** 头文件中的关键代码：

```

1. 00039 template<typename RandomAccessIterator, typename Compare>
2. 00040 class quick_sort_range: private no_assign {
3. 00041
4. 00042     inline size_t median_of_three(const RandomAccessIterator &array, size_t l, size_t m, size_t r) const {
5. 00043         return comp(array[l], array[m]) ? ( comp(array[m], array[r]) ?
6. 00044             m : ( comp( array[l], array[r] ) ? r : l ) ) : ( comp(array[r], array[m]) ?
7. 00045             m : ( comp( array[r], array[l] ) ? r : l ) );
8. 00046
9. 00047     inline size_t pseudo_median_of_nine( const RandomAccessIterator &a-
10. 00048         rray, const quick_sort_range &range ) const {
11. 00049         size_t offset = range.size/8u;
12. 00050             return median_of_three(array,
13. 00051                         median_of_three(array, 0, offset, offset
14. 00052                         *4, offset*5),
15. 00053                         median_of_three(array, offset*6, offset
16. 00054                         *7, range.size - 1) );

```

```

16. 00054      }
17. 00055
18. 00056 public:
19. 00057
20. 00058     static const size_t grainsize = 500;
21. 00059     const Compare &comp;
22. 00060     RandomAccessIterator begin;
23. 00061     size_t size;
24. 00062
25. 00063     quick_sort_range( RandomAccessIterator begin_, size_t size_, const
    Compare &comp_ ) :
26. 00064         comp(comp_), begin(begin_), size(size_) {}
27. 00065
28. 00066     bool empty() const {return size==0;}
29. 00067     bool is_divisible() const {return size>=grainsize;}
30. 00068
31. 00069     quick_sort_range( quick_sort_range& range, split ) : comp(range.co
        mp) {
32. 00070         RandomAccessIterator array = range.begin;
33. 00071         RandomAccessIterator key0 = range.begin;
34. 00072         size_t m = pseudo_median_of_nine(array, range);
35. 00073         if (m) std::swap ( array[0], array[m] );
36. 00074
37. 00075         size_t i=0;
38. 00076         size_t j=range.size;
39. 00077         // Partition interval [i+1,j-1] with key *key0.
40. 00078         for(;;) {
41. 00079             __TBB_ASSERT( i<j, NULL );
42. 00080             // Loop must terminate since array[1]==*key0.
43. 00081             do {
44. 00082                 --j;
45. 00083                 __TBB_ASSERT( i<=j, "bad ordering relation?" );
46. 00084             } while( comp( *key0, array[j] ) );
47. 00085             do {
48. 00086                 __TBB_ASSERT( i<=j, NULL );
49. 00087                 if( i==j ) goto partition;
50. 00088                 ++i;
51. 00089             } while( comp( array[i],*key0 ) );
52. 00090             if( i==j ) goto partition;
53. 00091             std::swap( array[i], array[j] );
54. 00092         }
55. 00093 partition:
56. 00094         // Put the partition key were it belongs
57. 00095         std::swap( array[j], *key0 );

```

```

58. 00096      // array[l..j) is less or equal to key.
59. 00097      // array(j..r) is greater or equal to key.
60. 00098      // array[j] is equal to key
61. 00099      i=j+1;
62. 00100      begin = array+i;
63. 00101      size = range.size-i;
64. 00102      range.size = j;
65. 00103      }
66. 00104 };
67. 00105
68. ....
69. 00218 #endif

```

再贴一下插入排序、快速排序之其中的俩种版本、及插入排序与快速排序结合运用的实现代码，如下：

```

1.   /// 插入排序,最坏情况下为 O(n^2)
2.   template< typename InPos, typename ValueType >
3.   void _isort( InPos posBegin_, InPos posEnd_, ValueType* )
4.   {
5.   /*****
6.   * 伪代码如下:
7.   *      for i = [1, n)
8.   *          t = x
9.   *          for( j = i; j > 0 && x[j-1] > t; j-- )
10.  *              x[j] = x[j-1]
11.  *          x[j] = t
12.  ****/
13.  if( posBegin_ == posEnd_ )
14.  {
15.  return;
16.  }
17.
18.  /// 循环迭代, 将每个元素插入到合适的位置
19.  for( InPos pos = posBegin_; pos != posEnd_; ++pos )
20.  {
21.  ValueType Val = *pos;
22.  InPos posPrev = pos;
23.  InPos pos2 = pos;
24.  /// 当元素比前一个元素大时, 交换
25.  for( ;pos2 != posBegin_ && *(--posPrev) > Val ; --pos2 )
26.  {

```

```

27.     *pos2 = *posPrev;
28.   }
29.   *pos2 = Val;
30. }
31. }
32.
33. /// 快速排序 1, 平均情况下需要 O(nlogn) 的时间
34. template< typename InPos >
35. inline void qsort1( InPos posBegin_, InPos posEnd_ )
36. {
37. /*****
38. * 伪代码如下:
39. *      void qsort(l, u)
40. *          if(l >= u)
41. *              return;
42. *          m = l
43. *          for i = [l+1, u]
44. *              if( x < x[1]
45. *                  swap(++m, i)
46. *              swap(l, m)
47. *              qsort(l, m-1)
48. *              qsort(m+1, u)
49. ****/
50. if( posBegin_ == posEnd_ )
51. {
52.   return;
53. }
54.
55. /// 将比第一个元素小的元素移至前半部
56. InPos pos = posBegin_;
57. InPos posLess = posBegin_;
58. for( ++pos; pos != posEnd_; ++pos )
59. {
60.   if( *pos < *posBegin_ )
61.   {
62.     swap( *pos, *(++posLess) );
63.   }
64. }
65.
66. /// 把第一个元素插到两快元素中央
67. swap( *posBegin_, *(posLess) );
68.

```

```
69. // 对前半部、后半部执行快速排序
70. qsort1(posBegin_, posLess);
71. qsort1(++posLess, posEnd_);
72. };
73.

74. // 快速排序 2, 原理与 1 基本相同, 通过两端同时迭代加快平均速度
75. template<typename InPos>
76. void qsort2( InPos posBegin_, InPos posEnd_ )
77. {
78.     if( distance(posBegin_, posEnd_) <= 0 )
79.     {
80.         return;
81.     }
82.

83.     InPos posL = posBegin_;
84.     InPos posR = posEnd_;
85.

86.     while( true )
87.     {
88.         // 找到不小于第一个元素的数
89.         do
90.         {
91.             ++posL;
92.         }while( *posL < *posBegin_ && posL != posEnd_ );
93.

94.         // 找到不大于第一个元素的数
95.         do
96.         {
97.             --posR;
98.         } while( *posR > *posBegin_ );
99.

100.        // 两个区域交叉时跳出循环
101.        if( distance(posL, posR) <= 0 )
102.        {
103.            break;
104.        }
105.        // 交换找到的元素
106.        swap(*posL, *posR);
107.    }
108.

109.    // 将第一个元素换到合适的位置
110.    swap(*posBegin_, *posR);
111.    // 对前半部、后半部执行快速排序 2
112.    qsort2(posBegin_, posR);
```

```

113. qsort2(++posR, posEnd_);
114. }
115.
116. /// 当元素个数小与 g_iSortMax 时使用插入排序, g_iSortMax 是根据 STL 库选取的
117. const int g_iSortMax = 32;
118. /// 该排序算法是快速排序与插入排序的结合
119. template<typename InPos>
120. void qsort3( InPos posBegin_, InPos posEnd_ )
121. {
122.   if( distance(posBegin_, posEnd_) <= 0 )
123.   {
124.     return;
125.   }
126.
127.   /// 小与 g_iSortMax 时使用插入排序
128.   if( distance(posBegin_, posEnd_) <= g_iSortMax )
129.   {
130.     return isort(posBegin_, posEnd_);
131.   }
132.
133.   /// 大与 g_iSortMax 时使用快速排序
134.   InPos posL = posBegin_;
135.   InPos posR = posEnd_;
136.
137.   while( true )
138.   {
139.     do
140.     {
141.       ++posL;
142.     }while( *posL < *posBegin_ && posL != posEnd_ );
143.
144.     do
145.     {
146.       --posR;
147.     } while ( *posR > *posBegin_ );
148.
149.     if( distance(posL, posR) <= 0 )
150.     {
151.       break;
152.     }
153.     swap(*posL, *posR);
154.   }
155.   swap(*posBegin_, *posR);
156.   qsort3(posBegin_, posR);

```

```
157. qsort3(++posR, posEnd_);  
158. }
```

十三、通过浙大上机复试试题学 SPFA 算法

作者: July、sunbaigui。二零一一年三月二十五日。

出处: http://blog.csdn.net/v_JULY_v。

前言:

本人不喜欢写诸如“**如何学算法**”此类的文章，一来怕被人认为是自以为是，二来话题太泛，怕扯得太远，反而不着边际。所以，一直不打算写怎么学习算法此类的文章。

不过，鉴于读者的热心支持与关注，给出以下几点小小的建议，仅供参考：

1、算法，浩如烟海，找到自己感兴趣的那个分支，或那个点来学习，然后，一往无前的深入探究下去。

2、兴趣第一，一切，由着你的兴趣走，忌浮躁。

3、思维敏捷。给你一道常见的题目，你的头脑中应该立刻能冒出解决这道问题的最适用的数据结构，以及算法。

4、随兴趣，多刷题。ACM 题。poj，面试题，包括下文将出现的研究生复试上机考试题，都可以作为你的编程练习题库。

5、多实践，多思考。学任何一个算法，反复研究，反复思考，反复实现。

6、数据结构是一切的基石。不必太过专注于算法，一切算法的实现，原理都是依托数据结构来实现的。弄懂了一个数据结构，你也就通了一大片算法。

7、学算法，重优化。

8、**学习算法的高明之处**不在于某个算法运用得有多自如，而在于通晓一个算法的内部原理，运作机制，及其来龙去脉。

ok，话不再多。希望，对你有用。

接下来，咱们来通过最近几年的浙大研究生复试上机试题，来学习或巩固常用的算法。

浙大研究生复试 2010 年上机试题-最短路径问题

问题描述:

给你 n 个点， m 条无向边，每条边都有长度 d 和花费 p ，给你起点 s 终点 t ，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：输入 n, m ，点的编号是 $1 \sim n$ ，然后是 m 行，每行 4 个数 a, b, d, p ，表示 a 和 b 之间有一条边，且其长度为 d ，花费为 p 。最后一行是两个数 s, t ；起点 s ，终点 t 。 n 和 m 为 0 时输入结束。

($1 < n \leq 1000$, $0 < m < 100000$, $s \neq t$)

输出：一行有两个数， 最短距离及其花费。 (下文，会详细解决)

几个最短路径算法的比较

ok, 怎么解决上述的最短路径问题列? 提到最短路径问题, 想必大家会立马想到 Dijkstra 算法, 但 Dijkstra 算法的效率如何列?

我们知道, Dijkstra 算法的运行时间, 依赖于其最小优先队列的采取何种具体实现决定, 而最小优先队列可有以下三种实现方法:

1、利用从 1 至 $|V|$ 编好号的顶点, 简单地将每一个 $d[v]$ 存入一个数组中对应的第 v 项, 如上述 DIJKSTRA (G, w, s) 所示, Dijkstra 算法的运行时间为 $O(V^2+E)$ 。

2、如果是二叉/项堆实现最小优先队列的话, EXTRACT-MIN(Q) 的运行时间为 $O(V \cdot \lg V)$,

所以, Dijkstra 算法的运行时间为 $O(V \cdot \lg V + E \cdot \lg V)$,

若所有顶点都是从源点可达的话, $O((V+E) \cdot \lg V) = O(E \cdot \lg V)$ 。

当是稀疏图时, 则 $E = O(V^2/\lg V)$, 此 Dijkstra 算法的运行时间为 $O(V^2)$ 。

3、采用斐波那契堆实现最小优先队列的话, EXTRACT-MIN(Q) 的运行时间为 $O(V \cdot \lg V)$,

所以, 此 Dijkstra 算法的运行时间即为 $O(V \cdot \lg V + E)$ 。

综上所述, 此最小优先队列的三种实现方法比较如下:

EXTRACT-MIN + RELAX

I、简单方式: $O(V \cdot V + E \cdot 1)$

II、二叉/项堆: $O(V \cdot \lg V + |E| \cdot \lg V)$

源点可达: $O(E \cdot \lg V)$

稀疏图时, 有 $E = O(V^2/\lg V)$,

$\Rightarrow O(V^2)$

III、斐波那契堆: $O(V \cdot \lg V + E)$

是的, 由上, 我们已经看出来了, Dijkstra 算法最快的实现是, 采用斐波那契堆作最小优先队列, 算法时间复杂度, 可达到 $O(V \cdot \lg V + E)$ 。

但是列? 如果题目有时间上的限制列? $V \lg V + E$ 的时间复杂度, 能否一定满足要求? 我们试图寻找一种解决此最短路径问题更快的算法。

这个时候, 我们想到了 Bellman-Ford 算法: 求单源最短路, 可以判断有无负权回路(若有, 则不存在最短路), 时效性较好, 时间复杂度 $O(VE)$ 。不仅时效性好于上述的 Dijkstra 算法, 还能判断回路中是否有无负权回路。

既然, 想到了 Bellman-Ford 算法, 那么时间上, 是否还能做进一步的突破。对了, 我们中国人自己的算法--SPFA 算法: SPFA 算法, Bellman-Ford 的队列优化, 时效性相对好, 时间复杂度 $O(kE)$ 。 $(k \ll V)$ 。

是的, 线性的时间复杂度, 我想, 再苛刻的题目, 或多或少, 也能满足要求了。

什么是 SPFA 算法

在上一篇文章二之三续、**Dijkstra 算法+Heap 堆的完整 c 实现源码**中，我们给出了 Dijkstra+Heap 堆的实现。其实，在稀疏图中对单源问题来说 SPFA 的效率略高于 Heap+Dijkstra；对于无向图上的 APSP(All Pairs Shortest Paths)问题，SPFA 算法加上优化后效率更是远高于 Heap+Dijkstra。

那么，究竟什么是 SPFA 算法呢？

SPFA 算法，Shortest Path Faster Algorithm，是由西南交通大学段凡丁于 1994 年发表的。正如上述所说，Dijkstra 算法无法用于负权回路，很多时候，如果给定的图存在负权边，这时类似 Dijkstra 等算法便没有了用武之地，而 Bellman-Ford 算法的复杂度又过高，SPFA 算法便派上用场了。

简洁起见，我们约定有向加权图 G 不存在负权回路，即最短路径一定存在。当然，我们可以在执行该算法前做一次拓扑排序，以判断是否存在负权回路。

我们用数组 d 记录每个结点的最短路径估计值，而且用邻接表来存储图 G 。

我们采取的方法是动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点 u ，并且用 u 点当前的最短路径估计值对离开 u 点所指向的结点 v 进行松弛操作，如果 v 点的最短路径估计值有所调整，且 v 点不在当前的队列中，就将 v 点放入队尾。

这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

定理：只要最短路径存在，上述 SPFA 算法必定能求出最小值。

期望的时间复杂度 $O(ke)$ ，其中 k 为所有顶点进队的平均次数，可以证明 k 一般小于等于 2。

SPFA 实际上是 Bellman-Ford 基础上的优化，以下，是此算法的伪代码：

```
//这里的 q 数组表示的是节点是否在队列中，如 q[v]=1 则点 v 在队列中
SPFA(G,w,s)
1. INITIALIZE-SINGLE-SOURCE(G,s)
2. Q ← &emptyset;;
3. for each vertex ∈ V[G]
4. q[v]=0
5. ENQUEUE(Q,s)
6. q[s]=1
7. while Q≠&emptyset;;
8. do u ← DEQUEUE(Q)
9. q[u]=0
10. for each edge(u,v) ∈ E[G]
11. do t ← d[v]
12. RELAX(u,v,w)
```

13. $n[v] \leftarrow u$
14. if ($d[v] < t$)
15. ENQUEUE(Q, v)
16. $q[v] = 1$

SPFA 是 **Bellman-Ford** 的队列优化，时效性相对好，时间复杂度 $O(kE)$ 。 $(k \ll V)$ 。与 **Bellman-ford** 算法类似，**SPFA** 算法采用一系列的松弛操作以得到从某一个节点出发到达图中其它所有节点的最短路径。所不同的是，**SPFA** 算法通过维护一个队列，使得一个节点的当前最短路径被更新之后没有必要立刻去更新其他的节点，从而大大减少了重复的操作次数。

与 **Dijkstra** 算法与 **Bellman-ford** 算法不同，**SPFA** 的算法时间效率是不稳定的，即它对于不同的图所需要的时间有很大的差别。在最好情形下，每一个节点都只入队一次，则算法实际上变为广度优先遍历，其时间复杂度仅为 $O(E)$ 。另一方面，存在这样的例子，使得每一个节点都被入队 $(V-1)$ 次，此时算法退化为 **Bellman-ford** 算法，其时间复杂度为 $O(VE)$ 。有研究指出在随机情形下平均一个节点入队的次数不超过 2 次，因此算法平均的时间复杂度为 $O(E)$ ，甚至优于使用堆优化过的 **Dijkstra** 算法。

最短路径问题的解决

浙大研究生复试 2010 年上机试题-最短路径问题

问题描述：

给你 n 个点， m 条无向边，每条边都有长度 d 和花费 p ，给你起点 s 终点 t ，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：输入 n, m ，点的编号是 $1 \sim n$ ，然后是 m 行，每行 4 个数 a, b, d, p ，表示 a 和 b 之间有一条边，且其长度为 d ，花费为 p 。最后一行是两个数 s, t ；起点 s ，终点 t 。 n 和 m 为 0 时输入结束。

$(1 < n \leq 1000, 0 < m \leq 100000, s \neq t)$

输出：一行有两个数，最短距离及其花费。

接下来，咱们便利用 **SPFA** 算法来解决此最短路径问题。

以下，便引用 **sunbaigui** 的代码来说明此问题：

声明几个变量：

[view plain](#) [copy to clipboard](#) [print?](#)

```

1. int d[1005][2];
2. bool used[1005];
3. vector<Node>map[1005];
4. int N,M,S,T;

```

建个数据结构：

[view plain](#) [copy to clipboard](#) [print?](#)

```

1. struct Node
2. {
3.     int x,y,z;
4.     Node(int a=0,int b=0,int c=0):x(a),y(b),z(c){}
5. };

```

以下是关键代码

[view plain](#)[copy to clipboard](#)[print](#)

```

1. //sunbaigui:
2. //首先将起点弹入队列，用 used 数组标记 i 节点是否在队列中，
3. //然后从队列中弹出节点，判断从这个弹出节点能到达的每个节点的距离是否小于已得到的距
4. //离。
5. void spfa()
6. {
7.     queue<int>q;    //构造一个队列
8.     q.push(S);
9.     memset(used,false,sizeof(used));
10.    int i;
11.    for(i=1;i<=N;i++)
12.        d[i][0]=d[i][1]=-1;
13.    d[S][0]=d[S][1]=0;           //初始化
14.    while(!q.empty())
15.    {
16.        int node=q.front();
17.        q.pop();
18.        used[node]=false;
19.        int t,dis,p;
20.        for(i=0;i<map[node].size();i++) //遍历
21.        {
22.            t=map[node][i].x;
23.            dis=map[node][i].y;
24.            p=map[node][i].z;
25.            if(d[t][0]==-1||d[t][0]>d[node][0]+dis)
26.            {
27.                d[t][0]=d[node][0]+dis; //松弛操作
28.                d[t][1]=d[node][1]+p;
29.                if(!used[t])
30.                {
31.                    used[t]=true;
32.                    q.push(t);      //t 点不在当前队列中，放入队尾。

```

```

33.         }
34.     }
35.     else if(d[t][0]!=-1&&d[t][0]==d[node][0]+dis)
36.     {
37.         if(d[t][1]>d[node][1]+p)
38.         {
39.             d[t][1]=d[node][1]+p;
40.             if(!used[t])
41.             {
42.                 q.push(t);
43.                 used[t]=true;
44.             }
45.         }
46.     }
47. }
48.
49. }
50. }
```

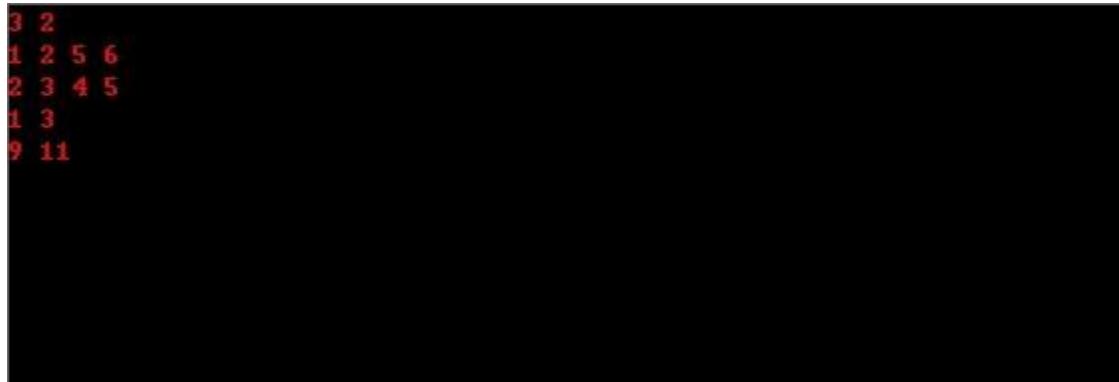
主函数测试用例：

[view plain](#)[copy to clipboard](#)[print?](#)

```

1. int main()
2. {
3.     while(scanf("%d %d",&N,&M)!=EOF)
4.     {
5.         if(N==0&&M==0)break;
6.         int s,t,dis,p;
7.         int i;
8.         for(i=1;i<=N;i++)
9.             map[i].clear();
10.        while(M--)
11.        {
12.            scanf("%d %d %d %d",&s,&t,&dis,&p);
13.            map[s].push_back(Node(t,dis,p));
14.            map[t].push_back(Node(s,dis,p));
15.        }
16.        scanf("%d %d",&S,&T);
17.        spfa();
18.        printf("%d %d\n",d[T][0],d[T][1]);
19.    }
20.    return 0;
21. }
```

运行结果，是：



最后，总结一下 (**Lunatic Princess**) :

(1)对于稀疏图，当然是 SPFA 的天下，不论是单源问题还是 APSP 问题，SPFA 的效率都是最高的，写起来也比 Dijkstra 简单。对于无向图的 APSP 问题还可以加入优化使效率提高 2 倍以上。

(2)对于稠密图，就得分情况讨论了。单源问题明显还是 Dijkstra 的势力范围，效率比 SPFA 要高 2-3 倍。APSP 问题，如果对时间要求不是那么严苛的话简简单单的 Floyd 即可满足要求，又快又不容易写错；否则就得使用 Dijkstra 或其他更高级的算法了。如果是无向图，则可以把 Dijkstra 扔掉了，加上优化的 SPFA 绝对是必然的选择。

稠密图

稀疏图

有负权边

单源问题	Dijkstra+heap	SPFA(或 Dijkstra+heap,根据稀疏程度)	SPFA
APSP(无向图)	SPFA(优化)/Floyd	SPFA(优化)	SPFA(优化)
APSP(有向图)	Floyd	SPFA (或 Dijkstra+heap,根据稀疏程度)	SPFA

完。

十四：快速选择 SELECT 算法的深入分析与实现

作者：July。

出处：http://blog.csdn.net/v_JULY_v。

前言

经典算法研究系列已经写了十三个算法，共计 22 篇文章（详情，见这：[十三个经典算法研究与总结、目录+索引](#)），我很怕我自己不再把这个算法系列给继续写下去了。沉思良久，到底是不想因为要创作狂想曲系列而耽搁这个经典算法研究系列，何况它，至今反响还不错。

ok，狂想曲第三章提出了一个算法，就是快速选择 SELECT 算法，关于这个 SELECT 算法通过选取数组中中位数的中位数作为枢纽元能保证在最坏情况下，亦能做到线性 $O(N)$ 的时间复杂度的证明，在狂想曲第三章也已经给出。

本文咱们从快速排序算法分析开始（因为如你所知，快速选择算法与快速排序算法在 partition 划分过程上是类似的），参考 Mark 的数据结构与算法分析-c 语言描述一书，而后逐步深入分析快速选择 SELECT 算法，最后，给出 SELECT 算法的程序实现。

同时，本文有部分内容来自狂想曲系列第三章，也算是对[第三章、寻找最小的 k 个数](#)的一个总结。yeah，有任何问题，欢迎各位批评指正，如果你挑出了本文章或[本 blog](#) 任何一个问题或错误，当即免费给予单独赠送本 blog 最新一期第 6 期的博文集锦 CHM 文件，谢谢。

第一节、快速排序

1.1、快速排序算法的介绍

关于快速排序算法，本人已经写了 3 篇文章（可参见其中的两篇：1、[十二、快速排序算法之所有版本的 c/c++ 实现](#)，2、[一之续、快速排序算法的深入分析](#)），为何又要旧事重提列？正如很多事物都有相似的地方，而咱们面临的问题--快速选择算法中的划分过程等同于快速排序，所以，在分析快速选择 SELECT 算法之前，咱们先再来简单回顾和分析下快速排序，ok，今天看到 Mark 的数据结构与算法分析-c 语言描述一书上对快速排序也有不错的介绍，所以为了增加点新鲜感，就不用自己以前的文章而改为直接引用 Mark 的叙述了：

As its name implies, quicksort is the fastest known sorting algorithm in practice. Its average running time is $O(n \log n)$ (快速排序是实践中已知的最快的排序算法，他的平均运行时间为 $O(N \log N)$) . It is very fast, mainly due to a very tight and highly optimized inner loop. It has

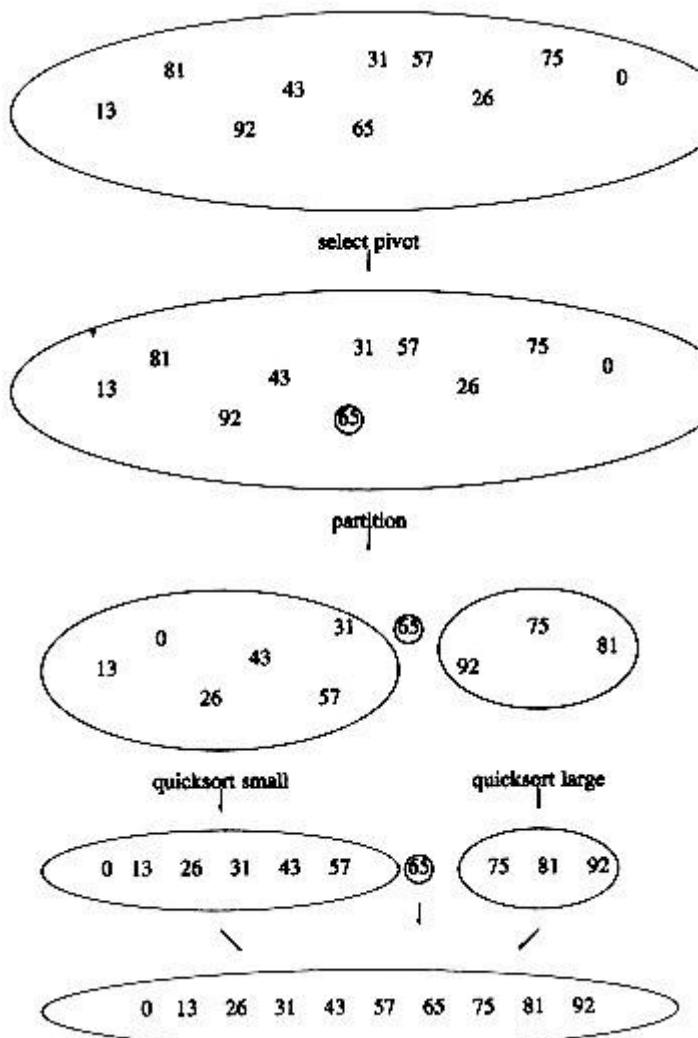
$O(n^2)$ worst-case performance (最坏情形的性能为 $O(N^2)$) , but this can be made exponentially unlikely with a little effort.

The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly (no doubt because of FORTRAN).

Like mergesort, quicksort is a divide-and-conquer recursive algorithm (像归并排序一样，快速排序也是一种采取分治方法的递归算法) . The basic algorithm to sort an array S consists of the following four easy steps (通过下面的 4 个步骤将数组 S 排序的算法如下) :

1. If the number of elements in S is 0 or 1, then return (如果 S 中元素个数是 0 或 1, 则返回) .
2. Pick any element v in S. This is called the pivot (取 S 中任一元素 v, 作为枢纽元) .
3. Partition $S - \{v\}$ (the remaining elements in S) into two disjoint groups (枢纽元 v 将 S 中其余的元素分成两个不想交的集合) : $S_1 = \{x | x \in S - \{v\} \text{ and } x \leq v\}$, and $S_2 = \{x | x \in S - \{v\} \text{ and } x \geq v\}$.
4. Return { quicksort(S_1) followed by v followed by quicksort(S_2) }.

下面依据上述步骤对序列 13,81,92,43,65,31,57,26,75,0 进行第一趟划分处理, 可得到如下图所示的过程:



1.2、选取枢纽元的几种方法

1、糟糕的方法

通常的做法是选择数组中第一个元素作为枢纽元，如果输入是随机的，那么这是可以接受的。但是，如果输入序列是预排序的或者是反序的，那么依据这样的枢纽元进行划分则会出现相当糟糕的情况，因为可能所有的元素不是被划入 S1，就是都被划入 S2 中。

2、较好的方法

一个比较好的做法是随机选取枢纽元，一般来说，这种策略是比较妥当的。

3、三数选取中值方法

例如，输入序列为 8, 1, 4, 9, 6, 3, 5, 2, 7, 0，它的左边元素为 8，右边元素为 0，中间位置 $\lfloor \text{left+right} \rfloor / 2 \rfloor$ 上的元素为 6，于是枢纽元为 6。显然，使用三数中值分割法消除了预

排序输入的坏情形，并且减少了快速排序大约 5%（此为前人实验所得数据，无法具体证明）的运行时间。

1.3、划分过程

下面，我们再对序列 8, 1, 4, 9, 6, 3, 5, 2, 7, 0 进行第一趟划分，我们要达到的划分目的就是为了把所有小于枢纽元（据三数取中分割法取元素 6 为枢纽元）的元素移到数组的左边，而把所有大于枢纽元的元素全部移到数组的右边。

此过程，如下述几个图所示：

8 1 4 9 0 3 5 2 7 6

i j

8 1 4 9 0 3 5 2 7 6

i j

After First Swap:

2 1 4 9 0 3 5 8 7 6

i j

Before Second Swap:

2 1 4 9 0 3 5 8 7 6

i j

After Second Swap:

2 1 4 5 0 3 9 8 7 6

i j

Before Third Swap

2 1 4 5 0 3 9 8 7 6

j i //i, j 在元素 3 处碰头之后，i++指向了 9，最后与 6 交换后，得到：

```
2 1 4 5 0 3 6 8 7 9  
i      pivot
```

至此，第一趟划分过程结束，枢纽元 6 将整个序列划分成了左小右大两个部分。

1.4、四个细节

下面，是 4 个值得你注意的细节问题：

1、我们要考虑一下，就是如何处理那些等于枢纽元的元素，问题在于当 i 遇到第一个等于枢纽元的关键字时，是否应该停止移动 i ，或者当 j 遇到一个等于枢纽元的元素时是否应该停止移动 j 。

答案是：如果 i, j 遇到等于枢纽元的元素，那么我们就让 i 和 j 都停止移动。

2、对于很小的数组，如数组的大小 $N \leq 20$ 时，快速排序不如插入排序好。

3、只通过元素间进行比较达到排序目的任何排序算法都需要进行 $O(N \log N)$ 次比较，如快速排序算法（最坏 $O(N^2)$ ，最好 $O(N \log N)$ ），归并排序算法（最坏 $O(N \log N)$ ，不过归并排序的问题在于合并两个待排序的序列需要附加线性内存，在整个算法中，还要将数据拷贝到临时数组再拷贝回来这样一些额外的开销，放慢了归并排序的速度）等。

4、下面是实现三数取中的划分方法的程序：

```
//三数取中分割法  
input_type median3( input_type a[], int left, int right )  
//下面的快速排序算法实现之一，及通过三数取中分割法寻找最小的 k 个数的快速选择  
SELECT 算法都要调用这个 median3 函数  
{  
    int center;  
    center = (left + right) / 2;  
  
    if( a[left] > a[center] )  
        swap( &a[left], &a[center] );  
    if( a[left] > a[right] )  
        swap( &a[left], &a[right] );  
    if( a[center] > a[right] )  
        swap( &a[center], &a[right] );  
  
    /* invariant: a[left] <= a[center] <= a[right] */  
    swap( &a[center], &a[right-1] );    /* hide pivot */
```

```
    return a[right-1];           /* return pivot */  
}
```

下面的程序是利用上面的三数取中分割法而运行的快速排序算法：

```
//快速排序的实现之一  
void q_sort( input_type a[], int left, int right )  
{  
    int i, j;  
    input_type pivot;  
    if( left + CUTOFF <= right )  
    {  
        pivot = median3( a, left, right ); //调用上面的实现三数取中分割法的 median3 函数  
        i=left; j=right-1; //第 8 句  
        for(;;)  
        {  
            while( a[++i] < pivot );  
            while( a[--j] > pivot );  
            if( i < j )  
                swap( &a[i], &a[j] );  
            else  
                break; //第 16 句  
        }  
        swap( &a[i], &a[right-1] ); /*restore pivot*/  
        q_sort( a, left, i-1 );  
        q_sort( a, i+1, right );  
  
        //如上所见，在划分过程（partition）后，快速排序需要两次递归，一次对左边递归  
        //一次对右边递归。下面，你将看到，快速选择 SELECT 算法始终只对一边进行递归。  
        //这从直观上也能反应出：此快速排序算法（O（N*logN））明显会比  
        //下面第二节中的快速选择 SELECT 算法（O（N））平均花费更多的运行时间。  
    }  
}
```

如果上面的第 8-16 句，改写成以下这样：

```

i=left+1; j=right-2;
for(;;)
{
    while( a[i] < pivot ) i++;
    while( a[j] > pivot ) j--;
    if( i < j )
        swap( &a[i], &a[j] );
    else
        break;
}

```

那么，当 $a[i] = a[j] = \text{pivot}$ 则会产生无限，即死循环（相信，不用我多余解释，:D）。ok，接下来，咱们将进入正题--快速选择 **SELECT** 算法。

第二节、线性期望时间的快速选择 **SELECT** 算法

2.1、快速选择 **SELECT** 算法的介绍

Quicksort can be modified to solve the selection problem, which we have seen in chapters 1 and 6. Recall that by using a priority queue, we can find the k th largest (or smallest) element in $O(n + k \log n)$ (以用最小堆初始化数组，然后取这个优先队列前 k 个值，复杂度 $O(n)+k*O(\log n)$)。实际上，最好采用最大堆寻找最小的 k 个数，那样，此时复杂度为 $n*\log k$ 。更多详情，请参见：狂想曲系列[第三章、寻找最小的 \$k\$ 个数](#)。For the special case of finding the median, this gives an $O(n \log n)$ algorithm.

Since we can sort the file in $O(n \log n)$ time, one might expect to obtain a better time bound for selection. The algorithm we present to find the k th smallest element in a set S is almost identical to quicksort. In fact, the first three steps are the same. We will call this algorithm quickselect (叫做快速选择)。Let $|S_i|$ denote the number of elements in S_i (令 $|S_i|$ 为 S_i 中元素的个数)。The steps of quickselect are:

1. If $|S| = 1$, then $k = 1$ and return the elements in S as the answer. If a cutoff for small files is being used and $|S| \leq \text{CUTOFF}$, then sort S and return the k th smallest element.
2. Pick a pivot element, v (- S . (选取一个枢纽元 v 属于 S)
3. Partition $S - \{v\}$ into S_1 and S_2 , as was done with quicksort.
(将集合 $S-\{v\}$ 分割成 S_1 和 S_2 ，就像我们在快速排序中所作的那样)

4. If $k \leq |S_1|$, then the k th smallest element must be in S_1 . In this case, return quickselect (S_1, k). If $k = 1 + |S_1|$, then the pivot is the k th smallest element and we can return it as the answer. Otherwise, the k th smallest element lies in S_2 , and it is the $(k - |S_1| - 1)$ st smallest element in S_2 . We make a recursive call and return quickselect ($S_2, k - |S_1| - 1$).

(如果 $k \leq |S_1|$, 那么第 k 个最小元素必然在 S_1 中。在这种情况下, 返回 quickselect (S_1, k)。如果 $k = 1 + |S_1|$, 那么枢纽元素就是第 k 个最小元素, 即找到, 直接返回它。否则, 这第 k 个最小元素就在 S_2 中, 即 S_2 中的第 $(k - |S_1| - 1)$ 个最小元素, 我们递归调用并返回 quickselect ($S_2, k - |S_1| - 1$)) (下面几节的程序关于 k 的表述可能会有所出入, 但无碍, 抓住原理即 **ok**)。

In contrast to quicksort, quickselect makes only one recursive call instead of two. The worst case of quickselect is identical to that of quicksort and is $O(n^2)$. Intuitively, this is because quicksort's worst case is when one of S_1 and S_2 is empty; thus, quickselect (快速选择) is not really saving a recursive call. The average running time, however, is $O(n)$ (不过, 其平均运行时间为 **$O(N)$** 。看到了没, 就是平均复杂度为 $O(N)$ 这句话). The analysis is similar to quicksort's and is left as an exercise.

The implementation of quickselect is even simpler than the abstract description might imply. The code to do this shown in Figure 7.16. When the algorithm terminates, the k th smallest element is in position k . This destroys the original ordering; if this is not desirable, then a copy must be made.

2.2、三数中值分割法寻找第 k 小的元素

第一节, 已经介绍过此三数中值分割法, 有个细节, 你要注意, 即数组元素索引是从“ $0 \dots i$ ”开始计数的, 所以第 k 小的元素应该是返回 $a[i] = a[k-1]$. 即 $k-1=i$ 。换句话就是说, 第 k 小元素, 实际上应该在数组中对应下标为 $k-1$ 。ok, 下面给出三数中值分割法寻找第 k 小的元素的程序的两个代码实现:

```
1. //代码实现一
2. //copyright@ mark allen weiss
3. //July、updated, 2011.05.05 凌晨。
4.
5. //三数中值分割法寻找第 k 小的元素的快速选择 SELECT 算法
6. void q_select( input_type a[], int k, int left, int right )
7. {
8.     int i, j;
9.     input_type pivot;
10.    if( left /*+ CUTOFF*/ <= right ) //去掉 CUTOFF 常量, 无用
11.    {
```

```

12.         pivot = median3( a, left, right ); //调用 1、4 节里的实现三数取中分割法
        的 median3 函数
13.         //取三数中值作为枢纽元，可以消除最坏情况而保证此算法是 O(N) 的。不过，这还
        只局限在理论意义上。
14.         //稍后，您将看到另一种选取枢纽元的方法。
15.
16.         i=left; j=right-1;
17.         for(;;) //此句到下面的九行代码，即为快速排序中的 partition 过程的实现之
        一
18.         {
19.             while( a[++i] < pivot ){}
20.             while( a[--j] > pivot ){}
21.             if (i < j )
22.                 swap( &a[i], &a[j] );
23.             else
24.                 break;
25.         }
26.         swap( &a[i], &a[right-1] ); /* restore pivot */
27.         if( k < i )
28.             q_select( a, k, left, i-1 );
29.         else
30.             if( k-1 > i ) //此条语句相当于: if(k>i+1)
31.                 q-select( a, k, i+1, right );
32.         //1、希望你已经看到，通过上面的 if-else 语句表明，此快速选择 SELECT 算法
        始终只对数组的一边进行递归，
33.         //这也是其与第一节中的快速排序算法的本质性区别。
34.
35.         //2、这个区别则直接决定了：快速排序算法最快能达到 O(N*logN)，
36.         //而快速选择 SELECT 算法则最坏亦能达到 O(N) 的线性时间复杂度。
37.         //3、而确保快速选择算法最坏情况下能做到 O(N) 的根本保障在于枢纽元元素的
        选取，
38.         //即采取稍后的 2.3 节里的五分化中项，或 2.4 节里的中位数的中外位数
        的枢纽元选择方法达到 O(N) 的目的。
39.         //后天老爸生日，孩儿深深祝福。July、updated, 2011.05.19。
40.     }
41.     else
42.         insert_sort(a, left, right-left+1 );
43.     }
44.
45.
46. //代码实现二
47. //copyright @ 飞羽
48. //July、updated, 2011.05.11。
49. //三数中值分割法寻找第 k 小的元素

```

```

50. bool median_select(int array[], int left, int right, int k)
51. {
52.     //第 k 小元素，实际上应该在数组中下标为 k-1
53.     if (k-1 > right || k-1 < left)
54.         return false;
55.
56.     //三数中值作为枢纽元方法，关键代码就是下述六行：
57.     int midIndex=(left+right)/2;
58.     if(array[left]<array[midIndex])
59.         swap(array[left],array[midIndex]);
60.     if(array[right]<array[midIndex])
61.         swap(array[right],array[midIndex]);
62.     if(array[right]<array[left])
63.         swap(array[right],array[left]);
64.     swap(array[midIndex], array[right]);
65.
66.     int pos = partition(array, left, right);
67.
68.     if (pos == k-1)    //第 k 小元素，实际上应该在数组中下标为 k-1
69.         return true;
70.     else if (pos > k-1)
71.         return median_select(array, left, pos-1, k);
72.     else return median_select(array, pos+1, right, k);
73. }

```

上述程序使用三数中值作为枢纽元的方法可以使得最坏情况发生的概率几乎可以忽略不计。然而，稍后，您将看到：通过一种更好的方法，如“五分化中项的中项”，或“中位数的中位数”等方法选取枢纽元，我们将能彻底保证在最坏情况下依然是线性 $O(N)$ 的复杂度。即，如稍后 2.3 节所示。

2.3、五分化中项的中项，确保 $O(N)$

The selection problem requires us to find the k th smallest element in a list S of n elements (要求我们找出含 N 个元素的表 S 中的第 k 个最小的元素). Of particular interest is the special case of finding the median. This occurs when $k = \lceil n/2 \rceil$ (向上取整) . (我们对找出中间元素的特殊情况有着特别的兴趣，这种情况发生在 $k=\lceil n/2 \rceil$ 的时候)

In Chapters 1, 6, 7 we have seen several solutions to the selection problem. The solution in Chapter 7 uses a variation of quicksort and runs in $O(n)$ average time (第 7 章中的解法，即本文上面第 1 节所述的思路 4，用到快速排序的变体并以平均时间 $O(N)$ 运行) . Indeed, it is described in Hoare's original paper on quicksort.

Although this algorithm runs in linear average time, it has a worst case of $O(n^2)$ (但它有一个 $O(N^2)$ 的最快情况). Selection can easily be solved in $O(n \log n)$ worst-case time by sorting the elements, but for a long time it was unknown whether or not selection could be accomplished in $O(n)$ worst-case time. The quickselect algorithm outlined in Section 7.7.6 is quite efficient in practice, so this was mostly a question of theoretical interest.

Recall that the basic algorithm is a simple recursive strategy. Assuming that n is larger than the cutoff point where elements are simply sorted, an element v , known as the pivot, is chosen. The remaining elements are placed into two sets, S_1 and S_2 . S_1 contains elements that are guaranteed to be no larger than v , and S_2 contains elements that are no smaller than v . Finally, if $k \leq |S_1|$, then the k th smallest element in S can be found by recursively computing the k th smallest element in S_1 . If $k = |S_1| + 1$, then the pivot is the k th smallest element. Otherwise, the k th smallest element in S is the $(k - |S_1| - 1)$ st smallest element in S_2 . The main difference between this algorithm and quicksort is that there is only one subproblem to solve instead of two (这个快速选择算法与快速排序之间的主要区别在于，这里求解的只有一个子问题，而不是两个子问题)。

定理 10.9

The running time of quickselect using median-of-median-of-five partitioning is $O(n)$.

The basic idea is still useful. Indeed, we will see that we can use it to improve the expected number of comparisons that quickselect makes. To get a good worst case, however, the key idea is to use one more level of indirection. Instead of finding the median from a sample of random elements, we will find the median from a sample of medians.

The basic pivot selection algorithm is as follows:

1. Arrange the n elements into $\lfloor n/5 \rfloor$ groups of 5 elements, ignoring the (at most four) extra elements.
2. Find the median of each group. This gives a list M of $\lfloor n/5 \rfloor$ medians.
3. Find the median of M . Return this as the pivot, v .

We will use the term **median-of-median-of-five partitioning** to describe the quickselect algorithm that uses the pivot selection rule given above. (我们将用术语“五分化中项的中项”来描述使用上面给出的枢纽元选择法的快速选择算法)。We will now show that median-of-median-of-five partitioning guarantees that each recursive subproblem is at most roughly 70 percent as large as the original (现在我们要证明，“五分化中项的中项”，得保证每个递归子问题的大小最多为原问题的大约 70%)。We will also show that the pivot can be

computed quickly enough to guarantee an $O(n)$ running time for the entire selection algorithm
(我们还要证明，对于整个选择算法，枢纽元可以足够快的算出，以确保 $O(N)$ 的运行时间。
看到了没，这再次佐证了我们的类似快速排序的 partition 过程的分治方法为 $O(N)$ 的观点) (更多详细的证明，请参考：[第三章、寻找最小的 k 个数](#))。

2.4、中位数的中位数， $O(N)$ 的再次论证

以下内容来自算法导论第九章第 9.3 节全部内容（最坏情况线性时间的选择），如下（我酌情对之参考原中文版做了翻译，下文中括号内的中文解释，为我个人添加）：

9.3 Selection in worst-case linear time (最坏情况下线性时间的选择算法)

We now examine a selection algorithm whose running time is $O(n)$ in the worst case (现在来看，一个最坏情况运行时间为 $O(N)$ 的选择算法 SELECT). Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), modified to take the element to partition around as an input parameter (像 RANDOMIZED-SELECT 一样，SELECT 通过输入数组的递归划分来找出所求元素，但是，该算法的基本思想是要保证对数组的划分是个好的划分。SELECT 采用了取自快速排序的确定性划分算法 partition，并做了修改，把划分主元元素作为其参数) .

The SELECT algorithm determines the i th smallest of an input array of $n > 1$ elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the i th smallest.) (算法 SELECT 通过执行下列步骤来确定一个有 $n > 1$ 个元素的输入数组中的第 i 小的元素。(如果 $n=1$ ，则 SELECT 返回它的唯一输入数值作为第 i 个最小值。))

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \bmod 5$ elements.
2. Find the median of each of the $\lfloor n/5 \rfloor$ groups by first insertion sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lfloor n/5 \rfloor$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the k th smallest element and there are $n-k$ elements on the high side of the partition.
(利用修改过的 partition 过程，按中位数的中位数 x 对输入数组进行划分，让 k 比划低去的元素数目多 1，所以， x 是第 k 小的元素，并且有 $n-k$ 个元素在划分的高区)

5. If $i = k$, then return x . Otherwise, use SELECT recursively to find the i th smallest element on the low side if $i < k$, or the $(i - k)$ th smallest element on the high side if $i > k$. (如果要找的第 i 小的元素等于程序返回的 k , 即 $i=k$, 则返回 x 。否则, 如果 $i < k$, 则在低区递归调用 SELECT 以找出第 i 小的元素, 如果 $i > k$, 则在高区间找第 $(i-k)$ 个最小元素)

1) 将输入数组的 n 个元素划分为 $\lceil n/5 \rceil$ 组, 每组 5 个元素, 且至多只有一个组由剩下的 $n \bmod 5$ 个元素组成。

2) 寻找 $\lceil n/5 \rceil$ 个组中每一组的中位数。首先对每组中的元素(至多为 5 个)进行插入排序, 然后从排序过的序列中选出中位数。

3) 对第 2 步中找出的 $\lceil n/5 \rceil$ 个中位数, 递归调用 SELECT 以找出其中位数 x 。(如果有偶数个中位数, 根据约定, x 是下中位数。)

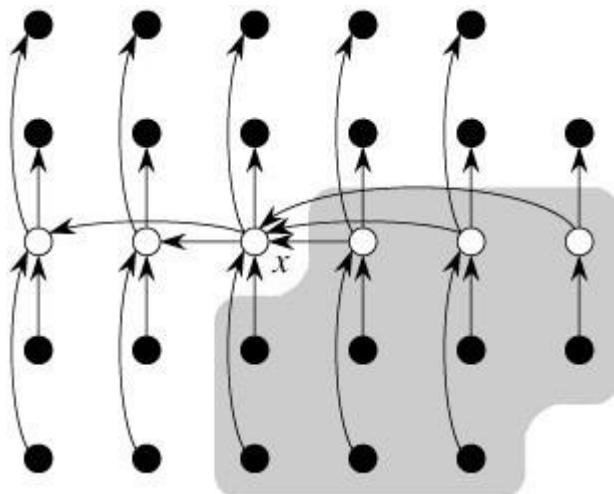
4) 利用修改过的 PARTITION 过程, 按中位数的中位数 x 对输入数组进行划分。让 k 比划分低区的元素数目多 1, 所以 x 是第 k 小的元素, 并且有 $n - k$ 个元素在划分的高区。

5) 如果 $i = k$, 则返回 x 。否则, 如果 $i < k$, 则在低区递归调用 SELECT 以找出第 i 小的元素, 如果 $i > k$, 则在高区找第 $(i-k)$ 个最小元素。

(以上五个步骤, 即本文上面的第四节末中所提到的所谓“五分化中项的中项”的方法。)

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element x . (为了分析 SELECT 的运行时间, 先来确定大于划分主元元素 x 的元素数的一个下界)Figure 9.1 is helpful in visualizing this bookkeeping. At least half of the medians found in step 2 are greater than^[1] the median-of-medians x . Thus, at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than x , except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least:

$$3 \left(\left\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6.$$



(Figure 9.1: 对上图的解释或称对 SELECT 算法的分析: n 个元素由小圆圈来表示, 并且每一个组占一纵列。组的中位数用白色表示, 而各中位数的中位数 x 也被标出。(当寻找偶数数目元素的中位数时, 使用下中位数)。箭头从比较大的元素指向较小的元素, 从中可以看出, 在 x 的右边, 每一个包含 5 个元素的组中都有 3 个元素大于 x, 在 x 的左边, 每一个包含 5 个元素的组中有 3 个元素小于 x。大于 x 的元素以阴影背景表示。)

Similarly, the number of elements that are less than x is at least $3n/10 - 6$. Thus, in the worst case, SELECT is called recursively on at most $7n/10 + 6$ elements in step 5.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of 140 or fewer elements requires $O(1)$ time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140. \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \leq cn$ for some suitably large constant c and all $n > 0$. We begin by assuming that $T(n) \leq cn$ for some suitably large constant c and all $n \leq 140$; this assumption holds if c is large enough. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by an for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) \\ &\quad + an \\ &\leq cn/5 + c + 7cn/10 + \\ &\quad 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an), \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \leq 0.$$

Inequality (9.2) is equivalent to the inequality $c \geq 10a(n/(n - 70))$ when $n > 70$. Because we assume that $n \geq 140$, we have $n/(n - 70) \leq 2$, and so choosing $c \geq 20a$ will satisfy inequality (9.2). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose c accordingly.) The worst-case running time of SELECT is therefore linear (因此, 此 SELECT 的最坏情况的运行时间是线性的).

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine

information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input. They are not subject to the $\Omega(n \lg n)$ lower bound because they manage to solve the selection problem without sorting.

(与比较排序（算法导论 8.1 节）中的一样，SELECT 和 RANDOMIZED-SELECT 仅通过元素间的比较来确定它们之间的相对次序。在算法导论第 8 章中，我们知道在比较模型中，即使在平均情况下，排序仍然要 $O(n \lg n)$ 的时间。第 8 章得线性时间排序算法在输入上做了假设。相反地，本节提到的此类似 **partition** 过程的 SELECT 算法不需要关于输入的任何假设，它们不受下界 $O(n \lg n)$ 的约束，因为它们没有使用排序就解决了选择问题（看到了没，道出了此算法的本质阿）)

Thus, the running time is linear because these algorithms do not sort; the linear-time behavior is not a result of assumptions about the input, as was the case for the sorting algorithms in Chapter 8. Sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1), and thus the method of sorting and indexing presented in the introduction to this chapter is asymptotically inefficient. (所以，本节中的选择算法之所以具有线性运行时间，是因为这些算法没有进行排序；线性时间的结论并不需要在输入上所任何假设，即可得到。.....)

第三节、快速选择 SELECT 算法的实现

本节，咱们将依据下图所示的步骤，采取中位数的中位数选取枢纽元的方法来实现此 SELECT 算法，

- 1) 将输入数组的 n 个元素划分为 $\lfloor n/5 \rfloor$ 组，每组 5 个元素，且至多只有一个组由剩下的 $n \bmod 5$ 个元素组成。
- 2) 寻找 $\lceil n/5 \rceil$ 个组中每一组的中位数。首先对每组中的元素（至多为 5 个）进行插入排序，然后从排序过的序列中选出中位数。
- 3) 对第 2 步中找出的 $\lceil n/5 \rceil$ 个中位数，递归调用 SELECT 以找出其中位数 x 。（如果有偶数个中位数，根据约定， x 是下中位数。）
- 4) 利用修改过的 PARTITION 过程，按中位数的中位数 x 对输入数组进行划分。让 k 比划分低区的元素数目多 1，所以 x 是第 k 小的元素，并且有 $n-k$ 个元素在划分的高区。
- 5) 如果 $i=k$ ，则返回 x 。否则，如果 $i < k$ ，则在低区递归调用 SELECT 以找出第 i 小的元素，如果 $i > k$ ，则在高区找第 $(i-k)$ 个最小元素。

不过，在实现之前，有个细节我还是必须要提醒你，即上文中 2.2 节开头处所述，“数组元素索引是从“0...i”开始计数的，所以第 k 小的元素应该是返回 $a[i]=a[k-1]$ 即 $k-1=i$ 。换句话说就是说，第 k 小元素，实际上应该在数组中对应下标为 $k-1$ ”这句话，我想，你应该明白了：

返回数组中第 k 小的元素，实际上就是返回数组中的元素 $array[i]$ ，即 $array[k-1]$ 。ok，最后请看此快速选择 SELECT 算法的完整代码实现（据我所知，在此之前，从没有人采取中位数的中位数选取枢纽元的方法来实现过这个 SELECT 算法）：

```
1. //copyright@ yansha && July && 飞羽
2. //July、updated, 2011.05.19.清晨。
3. //版权所有，引用必须注明出处：http://blog.csdn.net/v_JULY_v。
4. #include <iostream>
5. #include <time.h>
6. using namespace std;
7.
8. const int num_array = 13;
9. const int num_med_array = num_array / 5 + 1;
10. int array[num_array];
11. int median_array[num_med_array];
12.
13. //冒泡排序（晚些时候将修正为插入排序）
14. /*void insert_sort(int array[], int left, int loop_times, int compare_times)

15. {
16.     for (int i = 0; i < loop_times; i++)
17.     {
18.         for (int j = 0; j < compare_times - i; j++)
19.         {
20.             if (array[left + j] > array[left + j + 1])
21.                 swap(array[left + j], array[left + j + 1]);
22.         }
23.     }
24. }*/
25.
26. /*
27. //插入排序算法伪代码
28. INSERTION-SORT(A)                      cost      times
29. 1   for j ← 2 to length[A]                c1        n
30. 2       do key ← A[j]                   c2        n - 1
31. 3           Insert A[j] into the sorted sequence A[1 .. j - 1].    0...n - 1

32. 4       i ← j - 1                         c4        n - 1
33. 5       while i > 0 and A[i] > key       c5
34. 6           do A[i + 1] ← A[i]            c6
35. 7           i ← i - 1                  c7
36. 8           A[i + 1] ← key              c8        n - 1
37. */
38. //已修正为插入排序，如下:
```

```

39. void insert_sort(int array[], int left, int loop_times)
40. {
41.     for (int j = left; j < left+loop_times; j++)
42.     {
43.         int key = array[j];
44.         int i = j-1;
45.         while ( i>left && array[i]>key )
46.         {
47.             array[i+1] = array[i];
48.             i--;
49.         }
50.         array[i+1] = key;
51.     }
52. }
53.
54. int find_median(int array[], int left, int right)
55. {
56.     if (left == right)
57.         return array[left];
58.
59.     int index;
60.     for (index = left; index < right - 5; index += 5)
61.     {
62.         insert_sort(array, index, 4);
63.         int num = index - left;
64.         median_array[num / 5] = array[index + 2];
65.     }
66.
67.     // 处理剩余元素
68.     int remain_num = right - index + 1;
69.     if (remain_num > 0)
70.     {
71.         insert_sort(array, index, remain_num - 1);
72.         int num = index - left;
73.         median_array[num / 5] = array[index + remain_num / 2];
74.     }
75.
76.     int elem_aux_array = (right - left) / 5 - 1;
77.     if ((right - left) % 5 != 0)
78.         elem_aux_array++;
79.
80.     // 如果剩余一个元素返回, 否则继续递归
81.     if (elem_aux_array == 0)
82.         return median_array[0];

```

```

83.     else
84.         return find_median(median_array, 0, elem_aux_array);
85.     }
86.
87. // 寻找中位数的所在位置
88. int find_index(int array[], int left, int right, int median)
89. {
90.     for (int i = left; i <= right; i++)
91.     {
92.         if (array[i] == median)
93.             return i;
94.     }
95.     return -1;
96. }
97.
98. int q_select(int array[], int left, int right, int k)
99. {
100.    // 寻找中位数的中位数
101.    int median = find_median(array, left, right);
102.
103.    // 将中位数的中位数与最右元素交换
104.    int index = find_index(array, left, right, median);
105.    swap(array[index], array[right]);
106.
107.    int pivot = array[right];
108.
109.    // 申请两个移动指针并初始化
110.    int i = left;
111.    int j = right - 1;
112.
113.    // 根据枢纽元素的值对数组进行一次划分
114.    while (true)
115.    {
116.        while(array[i] < pivot)
117.            i++;
118.        while(array[j] > pivot)
119.            j--;
120.        if (i < j)
121.            swap(array[i], array[j]);
122.        else
123.            break;
124.    }
125.    swap(array[i], array[right]);
126.

```

```

127.     /* 对三种情况进行处理: (m = i - left + 1)
128.     1、如果 m=k, 即返回的主元即为我们要找的第 k 小的元素, 那么直接返回主元 a[i]即可;
129.     2、如果 m>k, 那么接下来要到低区间 A[0....m-1]中寻找, 丢掉高区间;
130.     3、如果 m<k, 那么接下来要到高区间 A[m+1...n-1]中寻找, 丢掉低区间。
131. */
132.     int m = i - left + 1;
133.     if (m == k)
134.         return array[i];
135.     else if(m > k)
136.         //上条语句相当于 if( (i-left+1) >k), 即 if( (i-left) > k-1 ), 于此就与
137.         //2.2 节里的代码实现一、二相对应起来了。
138.         return q_select(array, left, i - 1, k);
139.     else
140.         return q_select(array, i + 1, right, k - m);
141.
142. int main()
143. {
144.     //srand(unsigned(time(NULL)));
145.     //for (int j = 0; j < num_array; j++)
146.     //array[j] = rand();
147.
148.     int array[num_array]={0,45,78,55,47,4,1,2,7,8,96,36,45};
149.     // 寻找第 k 最小数
150.     int k = 4;
151.     int i = q_select(array, 0, num_array - 1, k);
152.     cout << i << endl;
153.
154.     return 0;
155. }

```

十五、多项式乘法与快速傅里叶变换

前言

经典算法研究系列，已经写到第十五章了，本章，咱们来介绍多项式的乘法以及快速傅里叶变换算法。本博客之前也已详细介绍过离散傅里叶变换（请参考：[十、从头到尾彻底理解傅里叶变换算法、上](#)，及[十、从头到尾彻底理解傅里叶变换算法、下](#)），这次咱们从多项式乘法开始，然后介绍FFT算法的原理与实现。同时，本文虽涉及到不少数学公式和定理（当然，我会尽量舍去一些与本文咱们要介绍的中心内容无关的定理或证明，只为保证能让读者易于接受或理解），但尽量保证通俗易懂，以让读者能看个明白。

有朋友建议，算法专一种，就ok，没必要各个都学习。但个人实在抑制不住自己的兴趣，就是想写，当没法做到强迫自己不写时，这个经典算法研究系列便一直这么写下来了。

第一节、多项式乘法

我们知道，有两种表示多项式的方法，即系数表示法和点值表示法。什么是系数表示法？所谓的系数表示法，举个例子如下图所示， $A(x) = 6x^3 + 7x^2 - 10x + 9$, $B(x) = -2x^3 + 4x - 5$, 则 $C(x) = A(x) * B(x)$ 就是普通的多项式相乘的算法，系数与系数相乘，这就是所谓的系数表示法。

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ - 2x^3 \quad \quad \quad + 4x - 5 \\ \hline - 30x^6 - 35x^5 + 50x^4 - 45 \\ 24x^6 + 28x^5 - 40x^4 + 36x \\ \hline - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

那么，何谓点值表示法？一个次数为 n 次的多项式 $A(x)$ 的点值表示就是 n 个点值所形成的集合： $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ 。其中所有 x_k 各不相同，且当 $k=0, 1, \dots, n-1$ 时，有 $y(k) = A(x_k)$ 。

一个多项式可以由很多不同的点值表示，这是由于任意 n 个相异点 x_0, x_1, \dots, x_{n-1} 组成的集合，都可以看做是这种点值法的表示方法。对于一个用系数形式表示的多项式来说，在原则上计算其点值表示是简单易行的，我们只需要选取 n 个相异点 x_0, x_1, \dots, x_{n-1} ，然后对 $k=0, 1, \dots, n-1$ ，求出 $A(x_k)$ ，然后根据霍纳法则，求出这 n 个点的值所需要的时间为 $O(n^2)$ 。

然，稍后，你将看到，如果巧妙的选取 x_k 的话，适当的利用点值表示可以使多项式的乘法可以在线性时间内完成。所以，如果我们能恰到好处的利用系数表示法与点值表示法的相互转化，那么我们可以加速多项式乘法（下面，将详细阐述这个过程），达到 $O(n \log n)$ 的最佳时间复杂度。

前面说了，当用系数表示法，即用一般的算法表示多项式时，两个 n 次多项式的乘法需要用 $O(n^2)$ 的时间才能完成。但采用点值表示法时，多项式乘法的运行时间仅为 $O(n)$ 。接下来，咱们要做的是，如何利用系数表示法与点值表示法的相互转化来实现**多项式系数表示法的 $O(n \log n)$** 的快速乘法。

第二节、多项式的快速乘法

在此之前，我们得明白两个概念，**求值与插值**。通俗的讲，所谓求值（系数->点值），是指系数形式的多项式转换为点值形式的表示方式。而插值（点值->系数）正好是求值的逆过程，即反过来，它是已知点值表示法，而要你转换其为多项式的系数表示法（用 n 个点值对也可以唯一确定一个不超过 $n-1$ 次多项式，这个过程称之为插值）。而这个系数表示法与点值表示法的相互转化，即无论是从系数表示法转化到点值表示法所谓的求值，还是从点值表示法转化到系数表示法所谓的插值，求值和插值两种过程的时间复杂度都是 $O(n \log n)$ 。

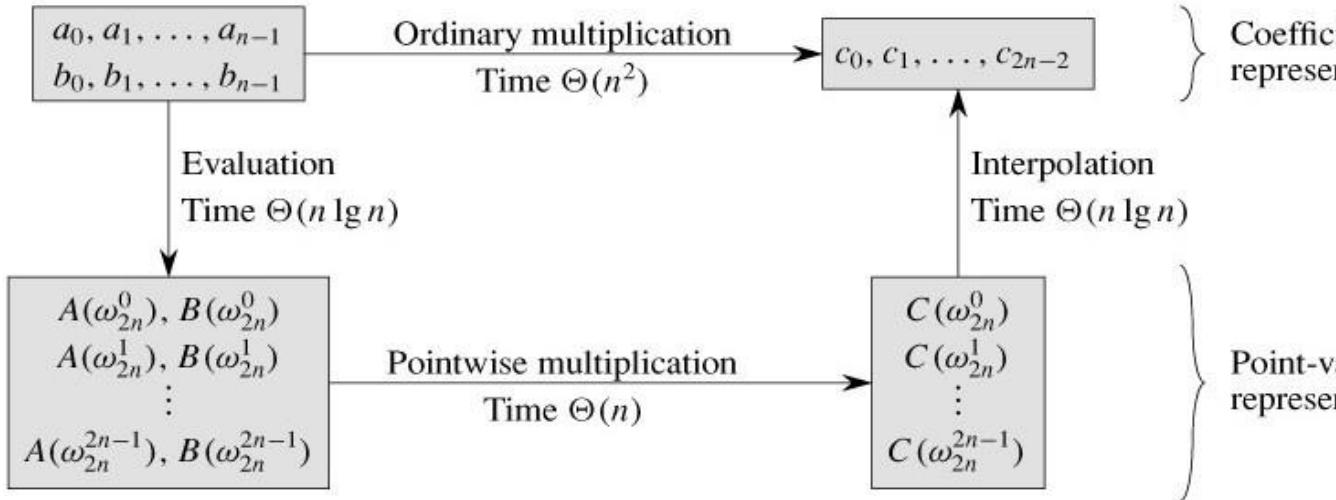
前面，我们已经说了，在多项式乘法中，如果用系数表示法表示多项式，那么多项式乘法的复杂度将为 $O(n^2)$ ，而用点值表示法表示的多项式，那么多项式的乘法的复杂度将是线性复杂度 $O(n)$ ，即：适当的利用点值表示可以使多项式的乘法可以在线性时间内完成。此时读者可以发挥你的想象，假设我们以下面这样一种过程来计算多项式的乘法（不过在此之前，你得先把两个要相乘的多项式 $A(x)$ 和 $B(x)$ 扩充为次数或者说系数为 $2n$ 次的多项式），我们将会得到我们想要的结果：

1. 系数表示法转化为点值表示法。先用系数表示法表示一个多项式，然后对这个多项式进行求值操作，即多项式从系数表示法变成了点值表示法，时间复杂度为 $O(n \log n)$ ；
2. 点值表示法计算多项式乘法。用点值表示法表示多项式后，计算多项式的乘法，线性复杂度为 $O(n)$ ；
3. 点值表示法转化为系数表示法。最终再次将点值表示法表示的多项式转变为系数表示法表示的多项式，这一过程，为 $O(n \log n)$ 。

那么，综上上述三项操作，系数表示法表示的多项式乘法总的时间复杂度已被我们降到了 $O(n \log n + n + n \log n) = O(N \log N)$ 。

如下图所示，从左至右，看过去，这个过程即是完成**多项式乘法的计算过程**。不过，完成这个过程有两种方法，一种就是前面第一节中所说的普通方法，即直接对系数表示法表示的多项式进行乘法运算，复杂度为 $O(n^2)$ ，它体现在下图中得 Ordinary multiplication 过程。还有一种就是本节上文处所述的三个步骤：1、将多项式由系数表示法转化为点值表

示法(点值过程)；2、利用点值表示法完成多项式乘法；3、将点值表示法再转化为系数表示法(插值过程)。三个步骤下来，总的时间复杂度为 $O(N \log N)$ 。它体现在下图中的 Evaluation + Pointwise multiplication + Interpolation 三个合过程。



由上图中，你也已经看到了。我们巧妙的利用了关于点值形式的多项式的线性时间乘法算法，来加快了系数形式表示的多项式乘法的运算速度。在此过程中，我们的工作最关键的就是以 $O(n \log n)$ 的时间快速把一个多项式从系数形式转换为点值形式（求值，我们即称之为 FFT），然后再以 $O(n \log n)$ 的时间从点值形式转换为系数形式（插值，我们即称之为 FFT 逆）。最终达到了我们以最终的系数形式表示的多项式的快速乘法为 $O(N \log N)$ 的时间复杂度。好不令人心生快意。

对上图，有一点必须说明，项 w_{2n} 是 $2n$ 次单位复根。且不容忽视的是，在上述两种表示法即系数表示法和点值表示法相互转换的过程中，都使用了单位复根，才得以在 $O(n \log n)$ 的时间内完成求值与插值运算（至于何谓单位复根，请参考相关资料。因为为了保证本文的通俗易懂性，无意引出一大堆公式或定理）。

第三节、FFT

注：本节有相当部分文字来自算法导论中文版第二版以及维基百科。

在具体介绍 FFT 之前，我们得熟悉知道折半定理是怎么一回事儿：如果 $n > 0$ 且 n 为偶数，那么， n 个 n 次单位复根的平方等于 $n/2$ 个 $n/2$ 个单位复根。我们已经知道，通过使用一种称为快速傅里叶变换 (FFT) 的方法，可以在 $O(n \log n)$ 的时间内计算出 $DFT_n(a)$ ，而若采用直接的方法复杂度为 $O(n^2)$ 。FFT 就是利用了单位复根的特殊性质。

你将看到，FFT方法运用的策略为分治策略，所谓分治，即分而治之。两个多项式 $A(x)$ 与 $B(x)$ 相乘的过程中，FFT用 $A(x)$ 中偶数下标的系数与奇数下标的系数，分别定义了两个新的次数界为 $n/2$ 的多项式 $A[0](x)$ 和 $A[1](x)$ ：

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1},$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}.$$

注意， $A[0]$ 包含了 A 中所有偶数下标的系数（下标的相应二进制数的最后一位为 0），而 $A[1]$ 包含 A 中所有奇数下标的系数（下标的相应二进制数的最后一位为 1）。有下式：

$$A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2),$$

这样，求 $A(x)$ 在 $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ 处的问题就转换为：

1) 求次数界为 $n/2$ 的多项式 $A[0]$ 与 $A[1]$ 在点 $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$ 的值，然后

2) 将上述结果进行组合。

下面，我们用 N 次单位根 W_N 来表示 $e^{-j\frac{2\pi}{N}}$ 。

W_N 的性质：

1. 周期性， W_N 具有周期 N 。
2. 对称性： $W_N^{k+\frac{N}{2}} = -W_N^k$ 。
3. $W_N^{ikn} = W_{\frac{N}{i}}^{kn}$

为了简单起见，我们下面设待变换序列长度 $n = 2^r$ 。根据上面单位根的对称性，求级数

$$y_k = \sum_{n=0}^{N-1} W_N^{kn} x_n$$

时，可以将求和区间分为两部分：

$$\begin{aligned} y_k &= \sum_{n=2i} W_N^{kn} x_n + \sum_{n=2i+1} W_N^{kn} x_n \\ &= \sum_i W_{\frac{N}{2}}^{ki} x_{2i} + W_N^k \sum_i W_{\frac{N}{2}}^{ki} x_{2i+1} \\ &= F_{even}(k) + W_N^k F_{odd}(k) \quad (i \in \mathbb{Z}) \end{aligned}$$

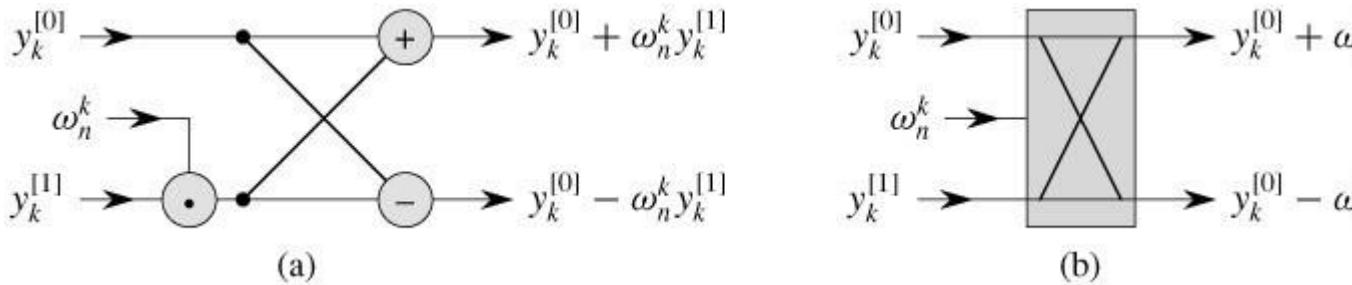
$F_{odd}(k)$ 和 $F_{even}(k)$ 是两个分别关于序列 $\{x_n\}_0^{N-1}$ 奇数号和偶数号序列 $N/2$ 点变换。由此式只能计算出 y_k 的前 $N/2$ 个点，对于后 $N/2$ 个点，注意 $F_{odd}(k)$ 和 $F_{even}(k)$ 都是周期为 $N/2$ 的函数，由单位根的对称性，于是有以下变换公式：

- $y_{k+\frac{N}{2}} = F_{even}(k) - W_N^k F_{odd}(k)$
- $y_k = F_{even}(k) + W_N^k F_{odd}(k)$.

这样，一个 N 点变换就分解成了两个 $N/2$ 点变换，照这样可继续分解下去。这就是库利-图基快速傅里叶变换算法的基本原理。此时，我们已经不难分析出此时算法的时间复杂度将为 $O(N \log N)$ 。

ok，没想到，本文之中还是出现了这么多的公式（没办法，搞这个 FFT 就是个纯数学活儿。之前买的一本小波与傅里叶分析基础也是如此，就是一本数学公式和定理的聚集书。不过，能看懂更好，实在无法弄懂也只权当做个稍稍了解）。

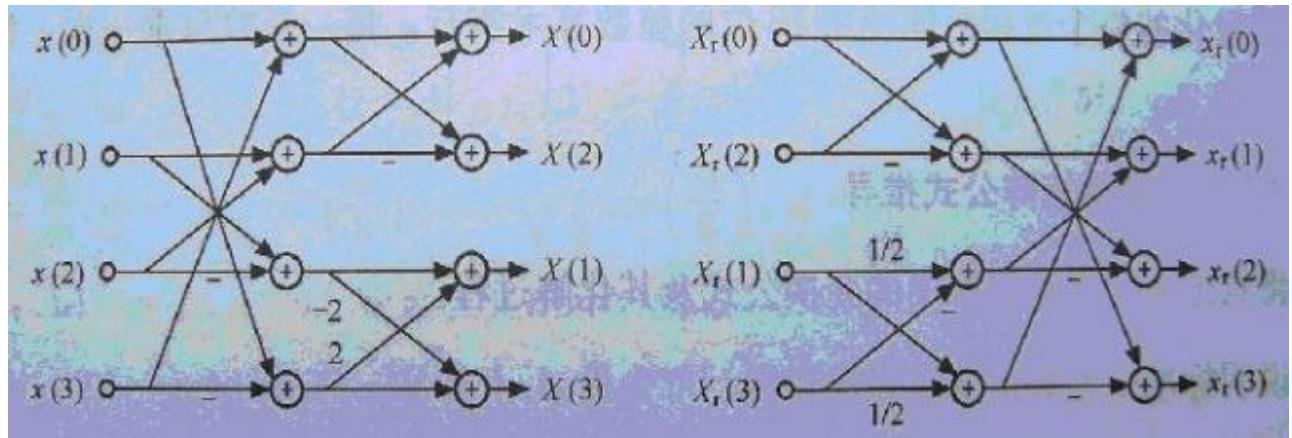
好了，下面，咱们来简单理解下 FFT 中的蝶形算法，本文将告结束。如下图所示：



有人这样解释蝶形算法：对于 N (2 的 x 次方) 点的离散信号，把它按索引位置分成两个序列，分别为 $0, 2, 4, \dots, 2K$ (记为 A) 和 $1, 3, 5, \dots, 2K-1$ (记为 B)，由傅立叶变换可以推出 N 点的傅立叶变换前半部分的结果为 $A+B^*$ 旋转因子，后半部分的结果为 $A-B^*$ 旋转因子。于是求 N 点的傅立叶变换就变成分别求两个 $N/2$ 点序列的傅立叶变换，对每一个 $N/2$ 点的序列，递归前面的步骤，直到只有两点的序列，就只变成简单的加减关系了。把这些点的加减关系用线连接，看上去就是个蝶形。ok，更多可参考算法导论第 30 章。

举一个例子，我们知道， 4×4 的矩阵运算如果按常规算法的话仍要进行 64 次乘法运算和 48 次加法，这将耗费较多的时间，于是在 H.264 中，有一种改进的算法（蝶形算法）可

以减少运算的次数。这种矩阵运算算法构造非常巧妙,利用构造的矩阵的整数性质和对称性,可完全将乘法运算转化为加法运算。如下图所示:



下面的代码来自一本数字图像处理的书上的源代码:

```

1.      VOID WINAPI FFT(complex<double> * TD, complex<double> * FD, int r)
2.      {
3.          // 付立叶变换点数
4.          LONG count;
5.
6.          // 循环变量
7.          int i,j,k;
8.
9.          // 中间变量
10.         int bsize,p;
11.
12.         // 角度
13.         double angle;
14.
15.         complex<double> *W,*X1,*X2,*X;
16.
17.         // 计算付立叶变换点数
18.         count = 1 << r;
19.
20.         // 分配运算所需存储器
21.         W = new complex<double>[count / 2];
22.         X1 = new complex<double>[count];
23.         X2 = new complex<double>[count];
24.
25.         // 计算加权系数
26.         for(i = 0; i < count / 2; i++)
27.     {

```

```

28.         angle = -i * PI * 2 / count;
29.         W[i] = complex<double> (cos(angle), sin(angle));
30.     }
31.
32.     // 将时域点写入 X1
33.     memcpy(X1, TD, sizeof(complex<double>) * count);
34.
35.     // 采用蝶形算法进行快速付立叶变换
36.     for(k = 0; k < r; k++)
37.     {
38.         for(j = 0; j < 1 << k; j++)
39.         {
40.             bsize = 1 << (r-k);
41.             for(i = 0; i < bsize / 2; i++)
42.             {
43.                 p = j * bsize;
44.                 X2[i + p] = X1[i + p] + X1[i + p + bsize / 2];
45.                 X2[i + p + bsize / 2] = (X1[i + p] - X1[i + p + bsize / 2]
46. ) * W[i * (1<<k)];
47.             }
48.         }
49.         X = X1;
50.         X1 = X2;
51.         X2 = X;
52.     }
53.     // 重新排序
54.     for(j = 0; j < count; j++)
55.     {
56.         p = 0;
57.         for(i = 0; i < r; i++)
58.         {
59.             if (j&(1<<i))
60.             {
61.                 p+=1<<(r-i-1);
62.             }
63.         }
64.         FD[j]=X1[p];
65.     }
66.
67.     // 释放内存
68.     delete W;
69.     delete X1;
70.     delete X2;

```

updated: 关于快速傅立叶变换(FFT)的 C++实现与 Matlab 实验, 这里有一篇不错的文章, 读者可以看看: <http://blog.csdn.net/rappy/article/details/1700829>。全系列, 完。
2012.03.03。

后记

关于制作者—花明月暗 & [有鱼网 CEO 吴超](#)

朋友花明月暗此前全权负责了此经典算法研究系列 V0.1 版(十三个经典算法)的制作, 另一朋友吴超则负责了 V0.2 版本(十五个经典算法)的更新, V0.2 版最后的目录+标签工作仍由花明月暗完成。

而在此之前, 就在昨天, 朋友吴超刚刚完成[程序员编程艺术](#) PDF 电子版本的制作, 同样也是 400 多页, 由此, 特别感谢这两位朋友的奉献, 希望我们可以帮助到更多的人。

联系作者

本经典算法研究系列文档中有任何一处错误, bug, 或漏洞, 读者朋友一经发现, 欢迎随时来信指导, 或 blog 内留言, 评论, 我将感激不尽。

我的联系方式如下:

邮箱: zhoulei0907@yahoo.cn

微博: <http://weibo.com/julyweibo>

Blog: http://blog.csdn.net/v_JULY_v。

本结构之法算法之道 blog, 若无意外, 永久更新, 永久勘误, 谢谢。

July、2012 年 4 月 4 日, 于有鱼网公司。

版权所有, 侵权必究。严禁用于任何商业用途, 违者定究法律责任。