Hash 表及其应用

引例 1、hash 表（P1925）

给定一个长度为 n 的整数数组 A[1]、A[2]、…、A[N](-10^9<=A[i]<=10^9)，和 m 个操作:

操作 1: 1 i x 把 A[i]的值增加 x（-10^3<=A[i]<=10^3）;

操作 2: 2 x 查询整数 x 在 A[1]..A[N]中出现的次数。

【样例】

10 9 //N 和 M 3 5 8 17 14 21 7 6 20 5 //A[1]..A[N] 2 9 //查询操作 1 2 -1 //修改操作 1 1 1 2 4 1 5 2 2 5 1 8 -1 1 2 1 2 5	0 //查询的结果 2 1 3

【数据范围】

30%的数据任何时候整数序列中 A[i]满足: -10,000,000<=A[i]<=10,000,000

100%的数据任何时候整数序列中 A[i]满足: -2,000,000,000<=A[i]<=2,000,000,000,

N,M 满足: 1<=N,M<=1,000,000

【讲解】

算法 1、朴素的算法（时间复杂度 O(M*N)）

◆对于操作 1: 只须 A[i]=A[i]+x;

◆对于操作 2: 在 A[1]..A[N]顺序查找 x 出现的次数。

算法 2、排序+二分查找（时间复杂度 O(M*N)）——费力不讨好

```
void solve()
{
    读入 N,M 和 A[1]..A[N];
    qksort(A,1,N);
    for(int k=1;k<=M;k++)
    {
        操作 1: 让 A[i]=A[i]+x;   然后 A[i]插入到 A[1]..A[N]的合适位置;
        操作 2:
            int lower=lower_bsrch(A,1,N,x);   //查找 A[1]..A[N]中大于等于 x 的第一个元素的下标
            int upper=upper_bsrch(A,1,N,x);   //查找 A[1]..A[N]中大于 x 的第一个元素的下标
            printf("%d\n",upper-lower);
    }
}
```

算法 3、标记数组（时间复杂度 O(M)）

设数组: vis[x]=整数 x 在序列中出现的次数，则算法为:

```
#define hash(i) vis[i+10000000]      //平移标记数组
#define hashsize 20000005            //标记数组的大小
int vis[hashsize];                   //标记数组
void solve()
{
      memset(vis,0,sizeof(vis));
      scanf("%d%d",&N,&M);
```

这种以空间换时间的算法时间复杂度相当优秀，但局限性也是显而易见的：因为内存限制，标记数组不能定义的太大！！。因为在本题中 A[i]的范围在[-2*10^9, 2*10^9]，设置一个 4*10^9的 int 数组，占用的

```
        for(int i=1;i<=N;i++)
        {
            scanf("%d",&A[i]);
            hash(A[i])++;
        }
        for(int k=1;k<=M;k++)
        {
             scanf("%d",&op);
             if(op==1)
             {
                 scanf("%d%d",&i,&x);
                 hash(A[i])--;
                 A[i]=A[i]+x;
                 hash(A[i])++;
             }

             if(op==2)
            {
                scanf("%d",&x);
                printf("%d\n",hash(x));
         }
}
```

内存高达：15259M≈15G。只能通过本题 30%的数据。

算法 4、hash 表（时间复杂度 O(kM)，其中 k 为冲突系数）

其实标记数组就是简单的hash，他只是把一个整数x直接映射到数组的下标vis[x]，这样要查找x时只须 O(1)的时间复杂度直接引用 vis[x]即可。例如样例：**3 5 8 17 14 21 7 6 20 5**

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
标记次数	0	0	0	1	0	2	1	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1

我们上面也分析这种直接映射占用庞大内存的缺陷，那么怎么避免这个缺陷呢？

1、hash 函数介绍

Hash 表的想法是这样的：**把序列各整数 x 不直接影射到标记数组的下标，而用函数来实现映射，**例如简单映射函数就是：x 除以质数 p 的余数，即 hash(x)=x%p；假设 p=11，则对于序列：

106、30、69、83、98、284、30、22、45

各数得到 hash 值如下表：

A[i]=	106	30	69	83	98	284	30	22	45
hash(A[i])=	7	8	3	6	10	9	8	0	1

因为 x%11 的范围是：0..10，所以这时标记数组大小只须 11 个元素：

hash(x)映射的下标	0	1	2	3	4	5	6	7	8	9	10
元素 A[i]		45		69			83	106	30	264	98
出现次数	0	1	0	1	0	0	1	1	2	1	1

那么这是要找某元素 x 出现的次数，只须查看 vis[hash(x)]即可。

利用 hash 函数解决了标记数组占用空间大的缺陷，但不幸的是，大多数情况并不是上面的例子那么特殊：**“不同元素的 hash 函数值相异。”**例如 45 和 67 两个元素，他们的 hash 函数值都为 45%11=1,67%11=1，这就出现了冲突，怎么办呢？

2、hash 表中如何解决冲突

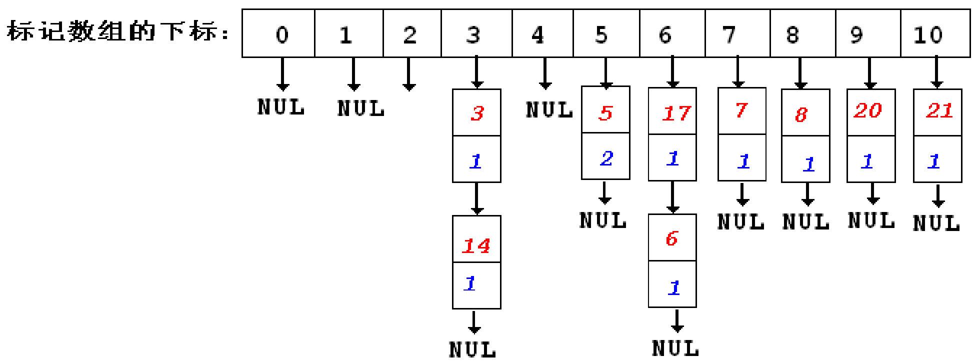
如果两个不同元素的 hash 函数值相同，这就出现冲突，解决冲突的办法很多，其中最简单和常用的是**挂链法**。所谓挂链法，就是把冲突元素组织成一个链表，例如：

序列 A[i]： 3、5、8、17、14、21、7、6、20、5

Hash 函数：hash(x)=x%11

A[i]=	3	5	8	17	14	21	7	6	20	5
hash(A[i])=	3	5	8	6	3	10	7	6	9	5

可以看出冲突的元素是：3 和 14、17 和 6，那么用挂链法得到 hash 表图示如下：



上图中，每个结点的红色数字代表元素，兰色数字代表该元素出现的次数。

现在我们要查询元素 x 出现的次数，只须扫描标记数组 vis[hash[x]]拉出的链表即可！

3、hash 表的存储结构及其操作

Hash 函数解决了标记数组占用空间大的缺陷，挂链法解决了冲突问题，现在来解决代码实现问题。观察上面的挂链图形，不正是图的邻接表存储结构吗？可以用变长数组存储和操作：

1）、存储结构

```
#define hashsize 1000003    //hash 表的大小，尽量取一个大质数
vector<int>g[hashsize],c[hashsize];
//g[i][k]表示 hash(x)为 i 拉出去的链表的第 k 个结点的对应的元素值
//c[i][k] 表示 hash(x)为 i 拉出去的链表的第 k 个结点的元素在序列中出现的次数
```

2）、hash 函数

hash 函数构造方法很多，常用采用**留余法**：hash(x)=|x%p|，其中 p 取一个大质数，例如 1000003，那么为什么要用质数，因为这样可以减少冲突。

```
int hash(int x)
{
    return abs(x%hashsize);
}
```

3）、hash 表的查找

```
int hash_look(int x)    //查找元素 x 的出现次数
{
    int i=hash(x);   //得到 hash 值
    for(int k=0;k<g[i].size();k++)
      if(g[i][k]==x) return c[i][k];    //如果有结点对应的元素与 x 相同，则返回其出现次数
    return 0;
}
```

4）、向 hash 表中插入一个结点

```
void hash_ins(int x)
{
    int i=hash(x);
    for(int k=0;k<g[i].size();k++)
      if(g[i][k]==x) { c[i][k]++; return; }    //如果 x 在 hash 表中出现，则次数加 1

    g[i].push_back(x);         //把 x 插入到 hash 表中
    c[i].push_back(1);
}
```

5）、删除 hash 表的一个结点（懒删除）

```
void hash_del(int x)
{
    int i=hash(x);
    for(int k=0;k<g[i].size();k++)
      if(g[i][k]==x && c[i][k]>0)
      {
           c[i][k]--;
           return;
      }
}
```

4、hash 表解答本题的算法框架

```
void solve()
{
    读入 N,M 和 A[1]..A[N]，并把每个 A[i]插入到 hash 表中;
    for(int k=1;k<=M;k++)
    {
       读入操作 op
       if(op==1){ 删除 hash 表中的 A[i]，A[i]=A[i]+x，并把 A[i]插入到 hash 表中}
       if(op==2) {在 hash 表中查找 x，并输出查询结果}
    }
}
```

例 2、兑换（P1922）

你手头有一张价值为 N 的纸币，每次你可以将你手头的某张价值为 X 的纸币换成三张价值为 X/2,X/3,X/4 的纸币，这里的"/"指的是整除(div)，当然也可以不兑换任何纸比，最后你最多能得到多少钱。

【样例】

2 // T 组数据 12 // N 2	13 //12 最多能得到的钱数 2 //2 最多能得到的钱数

【数据范围】

T<=10 N<=1,000,000,000

【讲解】

由题意可得：

- ●**状态函数：** f(x) = 一张面额为 x 的纸币能能得到钱的最大数量；
- ●**状态转移方程：** f(x) = MAX(**x**,**f(x/2)+f(x/3)+f(x/4)**)；
- ●**边界分析：** f(0) = 0;
- ●**最后的答案：** Ans = f(N)。

可用填表法和记忆化搜索实现上述状态转移方程。但状态记忆表需要：int d[1000000001]，太大了，因为针对一个整数 x，最多会出现的不同状态数量 ：$\lceil \log_2 x \rceil + \lceil \log_3 x \rceil + \lceil \log_4 x \rceil$，由此看来直接用数组作为状态记忆表会浪费大量的空间，所以可以用 hash 表来作为状态记忆表。

以 hash 表作为状态记忆表，每个接点需要的信息有两项：x 和 ans。在记忆化搜索时，每次都检查 x 在 hash 表中是否出现，若出现则返回 ans，否则继续搜索下去！

例 3、八数码问题（P1313）

编号为 1..8 的 8 个正方形滑块被摆成 3 行 3 列（有一个格子留空），如图所示。每次可以把与空格相邻的滑块（有公共边才算相邻）移到空格中，而它原来的位置就成为一个新空格。给定初始局面和目标局面（用 0 表示空格），你的任务是计算出最少的移动步骤。如果无法到达目标局面，则输出-1。

2	6	4
1	3	7
	5	8

初始局面

8	1	5
7	3	6
4		2

目标局面

【样例】

2 6 4 1 3 7 0 5 8 //初始局面 8 1 5 7 3 6 4 0 2 //目标局面	31 //最少的变换步骤，若无解则输出-1

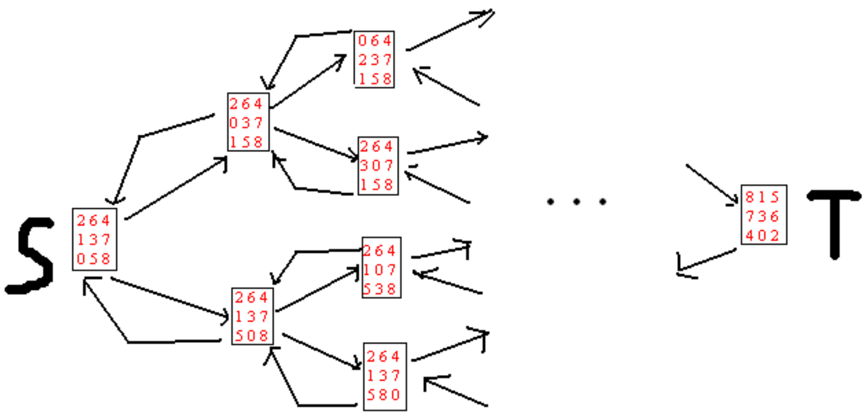
【讲解】

这是一道广度搜索算法计算最短路的典型题目，《算法竞赛入门经典》第 132 页有专门讲述！

◆**分步：**分若干步，计算从初始状态到达目标状态的最少步数（最短路径）；

◆**每步的选择：**空格子可与上、下、左、右格子中的数字交换。

由此可以建立一张图，每个状态当成图的一个结点，一个状态能转换成另一个状态，则把对应结点连一条有向边，问题就是计算图的最短路径。



用 BFS 解答这个问题是最有效的方法（也可用 DFSID 算法），把棋盘的每个状态看成 0..8 的一个全排列，那么最多有 9!=362880 个顶点。所以 BFS 中用到的队列 q 存储信息定义如下：

```
struct node     //队列存储每个元素存储的信息
{
   int x,y,a[3][3];  //0 的位置 x,y 和棋盘状态 a
}q[362885];
int front,rear,dist[362885];
```

BFS 算法的框架如下：

```
int dx[]={-1,1,0,0};
int dy[]={0,0,-1,1};
int S[3][3],T[3][3],x0,y0;    //初始棋盘状态 S 和目标棋盘状态 T，初始时 0 的位置 x0,y0

int BFS(int S[][3],x0,y0)
{
     S 进入队列 q[1]并标记为已被访问;
     dist[1]=0;
     while(队列不为空)
     {
           取队首的棋盘状态和 0 的位置 x,y;
           for(int  i=0;i<4;i++)   //上/下/左/右
           {
              tmp=对 q[front].a 进行 i 变换后的棋盘状态
              if(tmp 状态对应的结点没被访问过)
              {
                  tmp 进队列 ;
                  dist[rear]=dist[front]+1 ;
                  标记 tmp 状态为已经访问;
                  如果 tmp 状态与目标结点 T 对应的棋盘状态相同，则返回 dist[rear]
                  rear++ ;
              }
         }
        front++ ;
     }
     return -1;
}
```

现在的关键问题是如何标记结点是否访问过，即图的 BFS 的 vis 数组。平常见到的图中每个结点有一个确定的编号（1..N），而这个图中每个结点的编号是一个 0..8 的排列。

我们可以把每个棋盘对应的排列看成一个 9 位整数，以这个整数建立 hash 表，每次查找该结点是否访问过，在 hash 表中查找即可！

1)hash 数据结构定义

```
#define hashsize 1000003
vector<int> g[hashsize];
//g[i]k]=hash 函数值为 i 拉出去的链表的第 k 个结点对应的队列 q 的下标;
```

请仔细理解上面红色的字！！

234067518

2）、hash 函数

```
int hash(int a[][3])
{
    int t=0;
    for(int  x=0;x<3;x++)
    for(int  y=0;y<3;y++)
      t=(t*10+a[x][y])%hashsize;
    return t%hashsize;
}
```

3）、hash 查找

```
int hash_look(int a[][3])
{
    int i=hash(a);
    for(int k=0;k<g[i].size();k++)
    {
        int j=g[i][k];   //对应 q[j]
        if(memcmp(q[j].a, a, sizeof(q[j].a))==0) return 1;
    }
    return 0;

}
```

4）、hash 插入

```
void hash_ins(int a[][3],int j)  //把当前棋盘 a[3]3]对应的队列下标 j 插入 hash 表中
{
     int i=hash(a);
     g[i].push_back(j);
}
```

有了上面的 hash 操作，BFS 实现只须把标记修改成 hash_ins()，查询是否访问过只须调用 hash_look()即可！

例 4、平衡的队列（P1840）

FJ 有 N 头奶牛，从左到右排成一列（依次编号 1..N），每头奶牛有 K 个特征。FJ 用一种简单的方法描述一只奶牛所具有的特征，这是一个 K 位二进制数。比如一只奶牛的特征为 13，则 13 转换为二进制是 1101，这就是说这只奶牛显示出的特征是 1,3,4，而没有特征 2。

如果在第 i 到第 j 只奶牛身上每个特征出现次数相同，那么这 i..j 只奶牛是平衡的。FJ 很好奇奶牛们可以平衡的最大范围。看看你能不能解决它。

【讲解】

本题解答没有疑问，前缀和优化：

```
void ready()
{
   scanf("%d",&n);
   memset(s,0,sizeof(s));
   for(int i=1;i<=n;++i)
   {
      int t;
      scanf("%d",&t);
      for(int j=0;j<k;++j)
        s[i][j]=s[i-1][j]+((t>>j)&1);
   }
}
```

```
int Check60(int x,int y)
{
   int t=s[y][0]-s[x-1][0];
   for(int i=0;i<K;++i)
      if(s[y][i]-s[x-1][i]!=t) return 0;
   return 1;
}

void solve60()
{
   for(int j=1;j<=n;j++)
     for(int i=1;i<=j;i++)
       if(check(i,j)) ans=MAX(ans,j-i+1);
   cout<<ans<<'\n';
}
```

hash 优化：（时间复杂度 O(kM)）

check60()判断平衡的条件是：

s[y][i]-s[x-1][i]==s[y][0]-s[x-1][0]→ s[y][i]-g[y][0]==s[x-1][i]-s[x-1][0]　（1<i<K）

设 **sub[y][i]=g[y][i]-g[y][0]**；上述条件变为：**sub[y][i] == sub[x-1][i]**（1<i<K）。

一句话：要想第 s 头牛..第 t 头牛平衡，必须 sub[t][i]=sub[s-1][i]（1<i<K）。比如样例：

	7	6	7	2	1	4	2
属性0	1	1	1	0	0	1	0
属性1	1	1	1	1	0	0	1
属性2	1	0	1	0	1	0	0

→

1	2	3	3	3	4	4
1	2	3	4	4	4	5
1	1	2	2	3	3	3

前缀和g数组

→

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

sub数组

有上可以看出，要想第 i 到第 j 头牛平衡，则 sub 数组的第 i-1 列与第 j 列相同；那么我们可以为 sub[i][1]..sub[k]设计 hash 函数：

hash(i)=(**|sub[i][1]|*prime[1]+|sub[i][1]|*prime[2]+…+|sub[i][k]|*prime[k]**)%**hashsize;**

我们建立如下 hash 函数：

```
int prime[30]={1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,…};
int hash(int p)
{
    int t=0;
    for(int j=1;j<K;j++)
      t=(t+abs(sub[p][j])*prime[j])%hashsize;
    return t;
}
```

设计了这个 hash 函数后，建立 hash 表的含义如下：

```
#define hashsize 1000003
vector<int>g[hashsize];   //g[i][k]表示 hash 值为 i 拉出的链第 k 个结点对应的 sub 数组的下标
```

按此 hash 表的含义，分别写出 hash_look(int p)和 hash_ins(int p)函数后，解答如下：

在预处理时，把每个 sub[i]插入到 hash 表中

```
void ready()
{
   读入并计算 s 数组……;
   memset(sub,0,sizeof(sub));
   hash_ins(0);
   for(int i=1;i<=N;i++)
   {
        for(int j=0;j<K;j++)
          sub[i][j]=s[i][j]-s[i][0];
        hash_ins(i);  //插入 hash 表
    }
}
```

```
void solve100()
{
    int ans=0;
    for(int j=1;j<=N;j++)
    {
        int i=hash_look(j);
        if(ans<j-i) ans=j-i;   //为什么？
    }
    printf("%d\n",ans);
}
```

本题应该还有更好的 hash 函数设计！

作业：P1250、P1923、P1924