



软件基础实践 报告

（ 2022/ 2023 学年 第 1 学期）

题 目： 希尔排序

一般背包问题、0/1背包问题

KMP算法

图像压缩问题-动态规划法

残缺棋盘-分治法

**专 业**  软件工程（嵌入式培养）

**学 生 姓 名**  叶升

**班 级 学 号**  B21031925

**指 导 教 师**  费宁

**指 导 单 位**  软件工程系

**日 期**  2022.11.14

**希尔排序**

1. **概论**

使用希尔排序算法实现如下功能

要求：初始无序序列经过若干趟排序，最终称为关键字递增的有序序列。

分析：

(1) 目标是使用希尔排序方法使所有的数有序排列。

(2) 任务是存储每一个数，写出排序算法，然后调用希尔排序，向屏幕输出。

1. **设计与实现**
2. 程序的整体架构
3. shell函数作为本次研究课题关键函数，该函数使用gap为划分区间的增量，来减少每次比较交换的次数，提高程序的运行效率。
4. 本质上是对插排的优化，也就是说当遇到大规模乱序数据时，插排的效率较低，需要分组进行并行的插排。

(3) shell中的gap使用三分法（也可二分法）（向上取整）进行递减，那么只要最后我的gap递减到1，进行最后一遍插排，整个数组就必然有序.

(4)每组a[i+n\*gap]分开进行部分插排，使之跳跃有序（相同间隔的分为一组内部排序），也就是离他的最终位置越来越近。

(5)其它：使用vector数组动态输入数字，emplace\_back在尾部添加数字。

1. 主要功能的流程

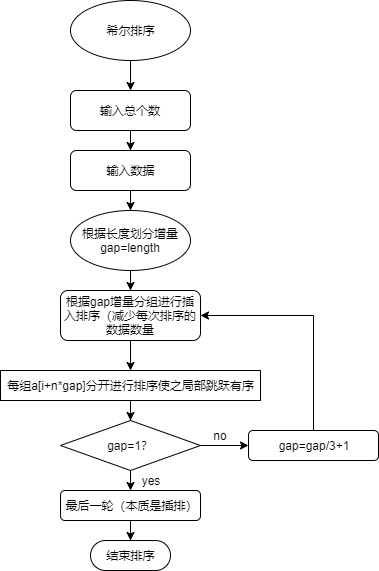


图1 希尔排序

1. 主要数据结构的设计

（1）利用数组存数据；（2）利用字符串数组存数据。（3）循环语句控制交换与比较次数

1. 数据的设计

（1）设待排序元素序列有n个元素，首先取一个整数gap（小于n）作为间隔将全部元素分为gap个子序列，所有距离为gap的元素放在同一个子序列中，在每一个子序列中分别实行直接插入排序。然后缩小间隔gap，重复上述子序列划分和排序工作。直到最后取gap=1，将所有元素放在同一个子序列中排序为止。

（2）由于开始时，gap的取值较大，每个子序列中的元素较少，排序速度较快，到排序后期gap取值逐渐变小，子序列中元素个数逐渐增多，但由于前面工作的基础，大多数元素已经基本有序，所以排序速度仍然很快。

（3）关于gap的选取：最初shell提出取gap=n/2向下取整，gap=gap/2向下取整，直到gap=1。但由于直到最后一步，在奇数位置的元素才会与偶数位置的元素进行比较，这样使用这个序列的效率会很低。后来Knuth提出取gap=n/3向下取整+1.还有人提出都取奇数为好，也有人提出gap互质为好。不过一般来说都分在1/3~1/2处比较合适。应用不同的序列会使希尔排序算法的性能有很大的差异。

1. 部分关键的代码

void **shell2**(**vector**<int>& a);

int limit;

int **main**(){

  int i;

**vector**<int> a{0};

  cout**<<**"请输入总个数："**<<endl**;

  cin**>>**limit;

  cout**<<**"请输入数据："**<<endl**;

  for(i=0;i<limit;i++){

    cin**>>**a**[**i**]**;

    a.**emplace\_back**(a**[**i**]**);

  }

  cout**<<endl<<endl<<**"排序后："**<<endl<<endl**;

**shell2**(a);

  //a.erase(a.end()-1);

  for(i=0;i<limit;i++){

    cout**<<**a**[**i**]<<**" ";

  }

  return 0;

}

void **shell2**(**vector**<int>& a) {

  const int start=0,end=limit-1;

  int gap = end - start + 1;  //初始化划分增量

  int temp{ 0 };

  do {  //每次减小增量，直到gap = 1

    gap = gap / 3 + 1;

    for (int i = start + gap; i <= end; ++i) {  //对每个划分进行直接插入排序

      if (a**[**i - gap**]** > a**[**i**]**) {

        temp = a**[**i**]**;

        int j = i - gap;

        do {  //移动元素并寻找位置

          a**[**j + gap**]** = a**[**j**]**;

          j -= gap;

        } while (j >= start && a**[**j**]** > temp);

        a**[**j + gap**]** = temp;  //插入元素

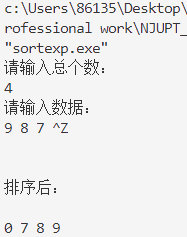
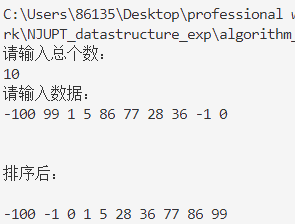
      }

    }

  } while (gap > 1);

}

1. **系统测试**



1. **评价与结论**
2. 希尔排序相较于直接插入排序和折半插入排序，在大多数情况下效率要高，但是它的时间复杂度分析很困难，约等于O(n2)，在特定情况下可以准确的估算排序码的比较次数和元素移动的次数

结论：比较不同算法和应用的场合，可以帮助我们以后在面对不同的需求时选择相对更优的解法提高效率。

1. 与插排的比较：插排在使用时，两个嵌套循环，外面大循环n-1次，里面每次n-i个数据依次比较，这对于小规模数据是可行的，但是若遇到大规模的乱序数据，就显得效率过低。看希尔排序，我写了不同的排序版本，其本质思想都是一样，跳跃的分组排序，使之局部有序，只要我的gap递减至1，那么最后的数据一定是排好的，不论gap是二分法或是三分法，不论循环以正序或是倒序进行，只要核心思想不变，都是希尔排序。从上面希尔排序的过程可以看到，相等的排序码会在排序前后的顺序发生颠倒，所以希尔排序是一种不稳定的排序算法。

结论：算法要研究透彻其核心思想，那么自己就可以写出不同的实现版本，也可以自如地应对各种变式。

1. 每次进行组内的插入排序，都是为了让元素距其最终位置更近一步。减少了大规模的移动，增加了微调的元素。

类似于宏观调控，修马路时多段同时开工，使得工时大大缩短。

希尔排序的优化：分组时，让各组一起进行插入排序，都只进行一次，然后循环进行，代码看起来简洁，但是实际耗时基本相同。

结论：希尔排序适用于大规模数据且乱序的情况，比插排更加好用。

1. 遇到未知的问题首先手动debug，不济再上网查找资料，再不济重新换思路重写。

**一般背包问题**

1. **概论**

（1）题目简述：

要求：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。

（2）分析：

性价比的贪心算法，最后一个物品进行拆分。

1. **设计与实现**
2. 程序的整体架构

以物品的性价比进行降序排序，从高到低选择，每次记录剩余容量，最后一个超容进行分割，填满背包。

1. 主要功能的流程

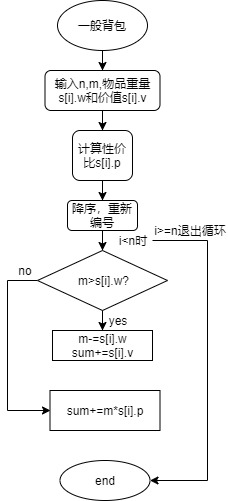


图2 一般背包问题主函数流程图

1. 主要数据结构的设计

自定义比较函数cmp，作为sort函数的第三个值以实现降序；m-=s[i].w来控制背包容量。

1. 数据的设计

定义结构体代表物品，给物品排序后重新编号，然后取物品。

1. 部分关键的代码

const int N =105;

typedef struct **treasure**{

  double w;

  double v;

  double p;

}**treasure**;

//自定义比较函数cmp

bool **cmp**(**treasure** t1,**treasure** t2){

  return t2.p<t1.p;

}

**treasure** s[N];

int **main**(){

  int n,m;

  cout**<<**"请输入不同物品的种数n和最大承载总重量m"**<<endl**;

  cin**>>**n**>>**m;

  cout**<<**"请分别输入这"**<<**n**<<**"种物品所对应的重量与价值"**<<endl**;

  for(int i=0;i<n;i++){

    cin**>>**s[i].w**>>**s[i].v;

    s[i].p=s[i].v/s[i].w;

  }

**sort**(s,s+n,**cmp**);//对物品以性价比进行降序排列

    cout**<<**"性价比排序(重新编号)："**<<endl**;

    for(int i=0;i<n;i++){

        cout**<<**"物品"**<<**i+1**<<**"性价比"**<<**s[i].p**<<endl**;

    }

  double sum=0.0;

  for(int i=0;i<n;i++){

    if(m>s[i].w){

      m-=s[i].w;

      sum+=s[i].v;

            cout**<<**"物品"**<<**i+1**<<**"取重量："**<<**s[i].w**<<**"，价值为："**<<**s[i].v**<<**"元"**<<endl**;

    }

    else{

            sum+=m\*s[i].p;

            cout**<<**"物品"**<<**i+1**<<**"取重量："**<<**m**<<**"，价值为："**<<**m\*s[i].p**<<**"元"**<<endl**;

      break;

    }

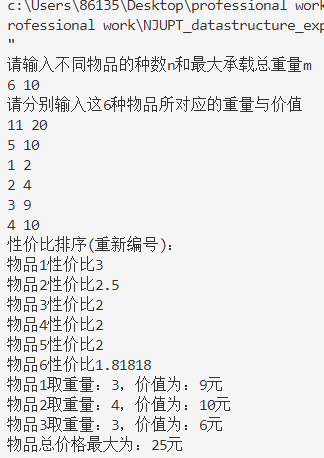
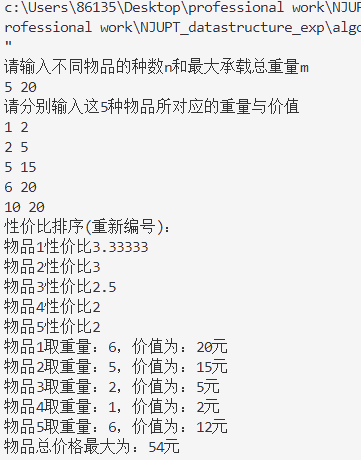
  }

  cout**<<**"物品总价格最大为："**<<**sum**<<**"元"**<<endl**;

  return 0;

}

1. **系统测试**



1. **评价与结论**
2. 一般背包的思路简单明了，是一般的以性价比来确定优先级的贪心算法。
3. 关键在取到即将装载的最后一个物品时，对物品切割以填满背包。
4. 其中自定义比较函数cmp，作为sort函数的第三个值以实现降序的用法比较巧妙。

**0/1背包问题**

**一、概论**

（1）题目简述：

要求：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。

（2）分析：

0/1背包和一般背包最大的不同在于物品不可拆分，所以会在基础上，引入动态规划的思想。

**二、设计与实现**

1. 程序的整体架构

在递归公式的过程中取的最大的价值。由二维dp数组优化为一维。

1. 主要功能的流程

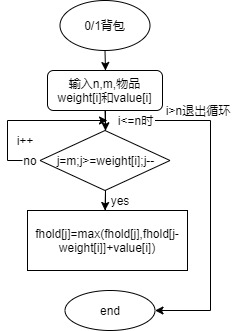


图3 0/1背包问题主函数流程图

1. 主要数据结构的设计

一维的滚动数组，Dp问题分析出状态表示之后, 最重要的就是推出状态计算,即状态转移方程.。

它一定是这些可行方案中的一种, 故我们需要进行划分子集的操作, 即来将这堆方案进行分类, 也叫划分子集，那么我们划分的标准是什么呢? 由于我们的此时的集合的方案都是从 1 ~ i 种物品种选, 故我们将集合分为两个子集, (1)不选择第i个物品, (2)选择第i个物品. 划分子集的原则是不重不漏, 划分依据为第i个物品是否被选取, 显然是一种不重不漏的方案。

1. 数据的设计

先使用二维数组分析： 不选择第 i 种物品, 由于我们是要求最大价值, 此时所容纳的体积为j, 但不选第i个物品问题转变为了从1 ~ (i - 1)个物品中选, 且所选总体积不超过j的方案的最大价值,这就是fhold[i - 1][j], 故fhold[i][j]的一个子集可以由 fhold[i - 1][j] 转移过来。

选择第 i 种物品, 此时由于选择了第i个物品, 故此时背包的体积为 j - w[i], 即减去这件物品的体积, 要想得到最大的价值, 由于选择了第i件物品, 故只剩下的物品只能在1 ~ (i - 1)中选, 且此时背包的体积为 j - w[i]. 那么问题就变为了在 1 ~ (i - 1)件物品中选, 且体积不超过 j - v[i]的方案的最大价值, 这个就是fhold[i - 1][j - w[i]]故fhold[i][j]的另一个子集表示fhold [i - 1][j - w[i]] + v[i],由于选择了第i个物品,故总价值加上 v[i]。

一维数组：

由于 j - w[i] < j 故fhold[j - w[i]] 先于fhold[j] 被计算出来, 故其实它是原本本层, 并不是上一层的。故这里的话我们要作第二层循环做一个变化,即我们原本是从小到大枚举体积, 我们这里变为从大到小枚举体积即可。因为这样的话fhold[j] 就会比fhold[j - w[i]] 先计算, 这样fhold[j - w[i]] 就是上一层没被计算的数值的了. 故第二层for改为:for(int j = m; j >= w[i]; --j)

这里的终止条件改为 j >= w[i], 因为根据if(j >= w[i])知道, 若j < v[i]就不会进行状态转移了。

1. 部分关键的代码

#include <iostream>

using namespace std;

const int N=105;

int weight[N],value[N],fhold[N];

int n,m;

int **main**()

{

    cout**<<**"请输入不同物品的种数n和最大承载总重量m"**<<endl**;

**scanf**("%d%d",&n,&m);

    cout**<<**"请分别输入这"**<<**n**<<**"种物品所对应的重量与价值"**<<endl**;

    for(int i=1;i<=n;i++) **scanf**("%d%d",&weight[i],&value[i]);

    for(int i=1;i<=n;i++)

      for(int j=m;j>=weight[i];j--)

      {

        fhold[j]=**max**(fhold[j],fhold[j-weight[i]]+value[i]);

**printf**("i=%d,fhold[%d]=max(fhold[%d],fhold[%d-weight[%d]]+value[%d])=%d \n",i,j,j,j,i,i,fhold[j]);

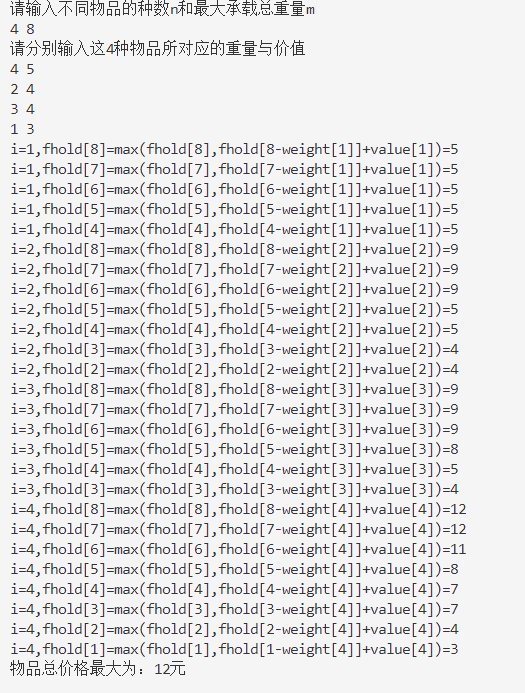
      }

    cout**<<**"物品总价格最大为："**<<**fhold[m]**<<**"元"**<<endl**;

    return 0;

}

1. **系统测试**



1. **评价与结论**
2. 0/1背包的思路：每一件物品其实只有两个状态，取或者不取，所以可以使用回溯法搜索出所有的情况，那么时间复杂度就是O(2^n)，这里的n表示物品数量。所以暴力的解法是指数级别的时间复杂度。进而才需要动态规划的解法来进行优化。
3. 在二维情况下，状态fhold[i][j]是由上一轮i - 1的状态得来的，fhold[i][j]与fhold[i - 1][j]是独立的。而优化到一维后，如果我们还是正序，则有fhold[较小体积]更新到fhold[较大体积]，则有可能本应该用第i-1轮的状态却用的是第i轮的状态。  **KMP算法**
4. **概论**

（1）题目简述：

简单字符串匹配->KMP算法

要求：字符串A中查找是否有子串B。

（2）分析：

若j位置与i位置匹配失败，j跳转到最大公共前后缀的后一位。类似于状态机思想，回到上一次匹配的状态，以减少次数。

1. **设计与实现**
2. 程序的整体架构

re函数计算0-i最大公共前后缀,next[i]进行改进，是前i+1个字符的最大公共前后缀，注意：和传统next数组有细微差别。

Kmp函数实现子串和主串的匹配，输出每个匹配的位置（头为i-j）

1. 主要功能的流程

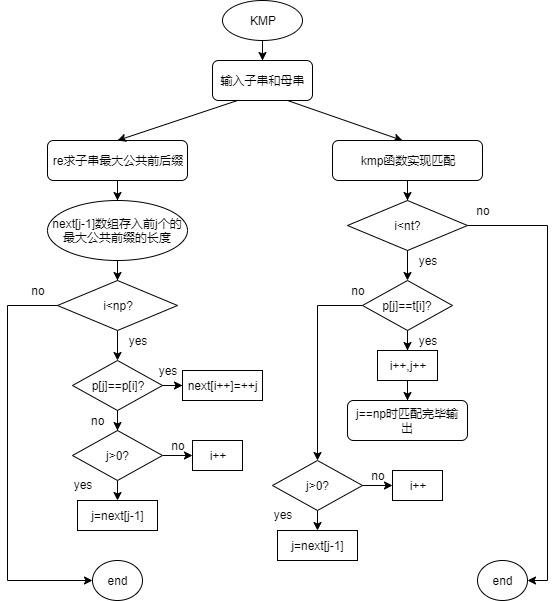


图4 KMP算法流程图

1. 主要数据结构的设计

使用next数组存储最大公共前后缀，若j位置与i位置匹配失败，j跳转到最大公共前后缀的后一位来减少匹配次数。

1. 数据的设计

计算子串a的最大公共前后缀，在子串a与主串b匹配的过程中，若j位置与i位置匹配失败，j跳转到最大公共前后缀的后一位，即next[j-1](区分于传统的next[j])，若j跳转到头仍然不匹配，则i++。

1. 部分关键的代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    求next[i]:p[0]~p[i]共计i+1个字符的最大公共前后缀

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void **re**(int next[], const char \*p)

{

    int i = 1, j = 0, np = **strlen**(p);

**memset**(next,0, sizeof(int)\*np);

    while(i < np)

    {

        if(p[j] == p[i])

            next[i++] = ++j;

        else

        {

            if(j > 0)   j = next[j-1];

            else        i++;

        }

    }

}

void **kmp**(const char \*t, const char \*p)//主串和子串

{

**printf**("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

**printf**("%s\n",t);

    int next[**SIZE**];

**re**(next, p);

    int i = 0, j = 0, nt = **strlen**(t), np = **strlen**(p);

    while(i < nt)

    {

        if(p[j] == t[i])

        {

            i++,j++;

            if(j == np)

            {

**result**(p, i-j);

                j = next[np-1];

            }

        }

        else

        {

            if(j > 0)   j = next[j-1];

            else        i++;

        }

    }

}

int **main**()

{

**string** a, b;

    cout**<<**"输入子串和主串："**<<endl**;

    while (cin **>>** a **>>** b)

    {

**kmp**(b.**c\_str**(), a.**c\_str**());

    }

**kmp**("ABABABABC", "ABAB");

**kmp**("ABABCABAB", "ABAB");

**kmp**("AAAAAAA", "AAA");

**kmp**("ABABABC", "ABABC");

**kmp**("XYXZdeOXZZKWXYZ", "WXYZ");

**kmp**("GCAATGCCTATGTGACCTATGTG", "TATGTG");

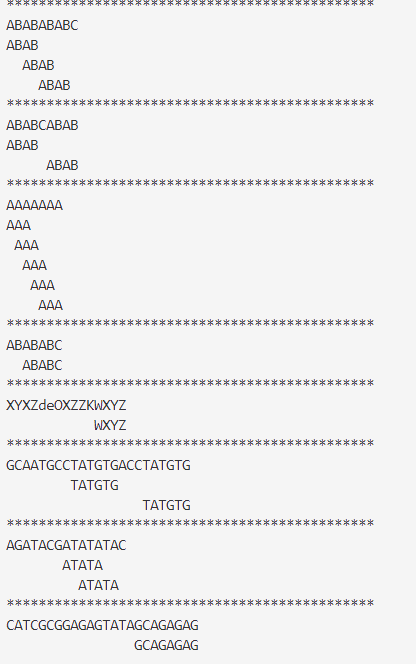
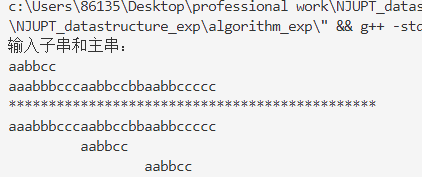
**kmp**("AGATACGATATATAC", "ATATA");

**kmp**("CATCGCGGAGAGTATAGCAGAGAG", "GCAGAGAG");

    return 0;

}

1. **系统测试**



1. **评价与结论**

（1）KMP的核心思想是最大公共前后缀，以及ij不匹配时j如何跳转，j匹配到头的时候i如何跳转。

（2）算法比较与暴力匹配的优化在于使用了next数组的“记忆”，使得已经匹配相同的前缀不需要重新从头匹配，故减少了时间

（3）以空间换时间，KMP算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。具体实现就是通过一个next()函数实现，函数本身包含了模式串的局部匹配信息。KMP算法的时间复杂度O(m+n)。（4）查找A串中是否有与B串相等的子串，我们可以用KMP算法，KMP算法原理是A中这一次匹配并不是从上一次匹配的下一个位置开始，也不是从头匹配B串，而是从A中不匹配的字符开始，同时B串中是从不匹配字符前面串的最大前后缀和位置开始。之所以能这样做是因为A中不匹配字符的前面部分串与B中不匹配字符前面部分串相同，而B中不匹配字符前面部分串又与B中开头的部分串相同，最长的相同的串就是最长公共前后缀。

（5） 为什么不相同的时候可以通过这样比较来判断前缀值呢，说起来有点复杂，既然前一个元素对应的最长后缀加上当前元素不行，那就去找第二长的后缀加上当前元素，第二长的后缀在哪里呢，就是当前比较元素的前一个元素对应的前缀值。为什么，当前比较元素前一个前缀值代表以当前比较元素结尾的串最长公共前后缀长度，即该前缀值对应的前面部分的串与后面部分的串相同，而后面部分的串又与以当前元素结尾的部分串相同，所以我们直接比较当前元素与该前缀值对应位置的元素就能求出当前元素的前缀值。

**图像压缩-动态规划法**

1. **概论**

（1）题目简述：

要求：计算机的图像是用灰度值序列来表示的｛P1,P2......Pn｝,其中Pi表示像素点i的灰度值。而通常灰度值的范围是0~255，因此需要8位二进制数来表示一个像素。可否用更少的位数来表示灰度值。

（2）分析：

简单来说就是动态规划进行分组，每组使用更少的位数来一定程度上消除冗余，选择最优的分组方式，最后实现一定程度上的图像压缩。

1. **设计与实现**
2. 程序的整体架构

使用Compress函数实现求最优空间值，使用Traceback函数实现构造最优分段解的功能。

1. 主要功能的流程

PS：b[i]存像素点所需位数，s[i]存压缩后空间，l[i]存i-j+1~i最优长度,c[j]和w[j]来跟踪l[n]和b[n]。

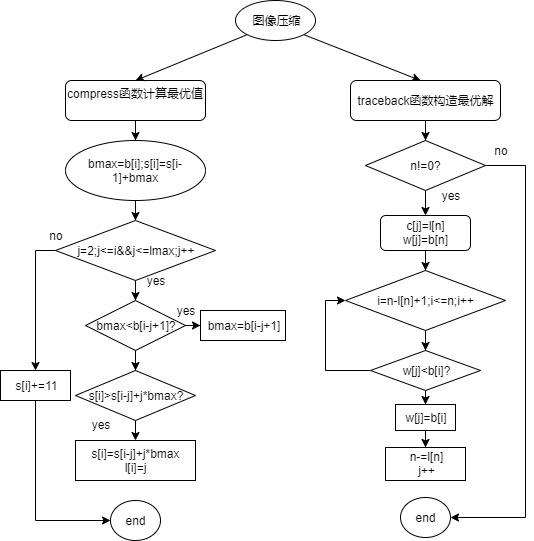


图5 图像压缩流程图

1. 主要数据结构的设计

s[i]不断与s[i-j]+j\*bmax比较来得出更优的压缩空间

c[j]和w[j]从后面开始跟踪最优分段长度l[i]和每一段的最大位数b[i]，n更新为n-l[n]即倒数第j段之前的序列长度。

1. 数据的设计

后一个分段的最优和前面分段的最优解一定是整体的最优组合，compress中从后向前寻找最小的最终值，并求出了每次的分组与最优长度，traceback中c和w数组追踪最优的长度和每段位大小。

1. 部分关键的代码

#define N 100

int **length**( int val ) {

    int count = 1;

    val /= 2;

    while( val > 0 ) {

        count++;

        val /= 2;

    }

    return count;

}

//计算最优值

// l[i]存的的从1到i的序列中某个位置i-j+1隔开到最后i这一段的最优长度l[i]

// b[i]应该要存相应段的最大像素位数，但实际存的是每个像素的所占位数

//变量含义  n：序列p中元素个数  p：灰度值序列  s：存从1到i最少存储位数

void **Compress**( int n, int \*p, int \*s, int \*l, int \*b ) {

    int bb[**N**];

    int Lmax = 255;  //像素序列最大长度

    int header = 11; //每一段的头部信息3（元素最多用8位二进制数表示）+8（段最大长度255）=11

    s[0] = 0;

    //子问题的后边界i

    for( int i = 1; i <= n; i++ ) {

        //计算像素点p需要的存储位数

        b[i] = **length**( p[i] );

        int bMax = b[i];            //后面段中元素所占位数最大值

        s[i] = s[i - 1] + 1 \* bMax; //只有一个元素，所以\*1

        l[i] = 1;

        // j为最后一段中元素个数，j取i和255中的最小值

        //可看作从i往前逐渐扩大i到i-1,i再到i-2,i-1,i...

        for( int j = 2; j <= i && j <= Lmax; j++ ) {

            //元素逐渐向前更新，可能会出现比bMax更大的，需要更新

            if( bMax < b[i - j + 1] ) {

                bMax = b[i - j + 1];

            }

            //找到更好的分段  j为段中元素个数

            if( s[i] > s[i - j] + j \* bMax ) {

                s[i] = s[i - j] + j \* bMax;

                l[i] = j;

            }

        }

        //加上每一段的头部信息3（元素最多用8位二进制数表示）+8（段最大长度255）

        s[i] += header;

    }

    cout **<<** "图像压缩后的最小空间：" **<<** s[n] **<<** **endl**;

}

// 构造最优解

void **TraceBack**( int n, int \*l, int \*b ) { // n，l,b与上方同义

    int j = 1; //正在追踪的段数，从后往前，最后面是第一段

    int c[100];  //存划分段的的长度

    int w[100];  //存划分段元素所需最大位数

    while( n ) {

        c[j] = l[n];  //从后往前追踪的第j段长度

        w[j] = b[n];  //从后往前追踪的第j段元素最大位数

        //求位数的方法

        //遍历每一个段所有元素，取最大值

        for( int i = n - l[n] + 1; i <= n; i++ ) {

            if( w[j] < b[i] ) {

                w[j] = b[i];

            }

        }

        n = n - l[n]; // n更新为第j段之前序列长度

        j++;

    }

    cout **<<** "将原序列划分成 " **<<** j - 1 **<<** " 段\n";

    for( int i = j - 1; i >= 1; i-- ) {

        cout **<<** "段长度:" **<<** c[i] **<<** ",所需存储位数：" **<<** w[i] **<<** **endl**;

    }

}

int **main**() {

    int p[**N**] = {0} ; //图像灰度数组{p1,p2,p3,p4,p5,p6...pn} 下标从1开始计数,p0是0

    int s[**N**] = {0}, l[**N**] = {0}, b[**N**] = {0}; //s存最优值，l存最优解

    int n;

    cout **<<** "请输入灰度值序列元素个数(最多99):";

    cin **>>** n;

    cout **<<** "请输入灰度值序列元素:";

    for( int i = 1; i <= n; i++ ) {

        cin **>>** p[i];

    }

    cout **<<** "图像的灰度序列为：" ;

    for( int i = 1; i < n + 1; i++ ) {

        cout **<<** p[i] **<<** " ";

    }

    cout **<<** **endl**;

**Compress**( n, p, s, l, b );

**TraceBack**( n, l, b );

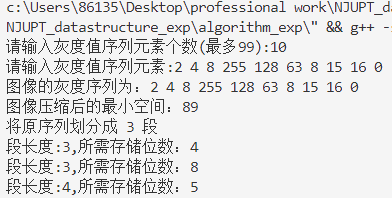
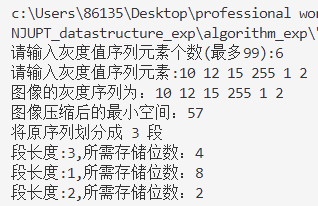
    //Output(s, l, b, n);

**system**( "pause" );

    return 0;

}

1. **系统测试**



1. **评价与结论**

（1）本题使用到的有header=11（每一段的头部信息3（元素最多用8位二进制数表示）+8（段最大长度255）=11）

（2）比较所有子解从中选择最优解，其核心在于s[i]>s[i-j]+j\*bmax，从后面一段选择使得s[i]最小的最优解，记录l[i]=j为最优长度。

（3）动态规划异常巧妙，从后向前遍历可能取的段长度，n=n-l[n]更新为第j段之前的序列长度，最后实现图像压缩。

**残缺棋盘-分治法**

1. **概论**

（1）题目简述：

要求：残缺棋盘是一个2^k\*2^个方格的棋盘，其中恰有1个方格残缺。图中给出，其中残缺部分用阴影表示。

Shape, square

Description automatically generated

这样的棋盘称为"三格板"，残缺棋盘问题就是用这四种三格板覆盖更大的残缺棋盘。再次覆盖中要求:

(1)两个三格板不能重复。

(2)三格板不能覆盖残缺棋盘方格，但必须覆盖到其他所有的方格。

（2）分析：将整个棋盘分为左上、右上、左下、右下四个区域，使用同步判断，同时进行递归来填充棋盘，相当于解决每块的log2 n个子问题，直到边长size=1，整个棋盘便填满了。

1. **设计与实现**
2. 程序的整体架构

采用分四个区域，每个区域进行递归解决本区域的子问题，同时在空区域形成的L棋盘上标注相同的三个数字，按一般的左上、右上、左下、右下的顺序来进行判定。

1. 主要功能的流程

PS：divide(ox,oy,row,column,s)，参数分别为原点(ox,oy)，残缺格(row,column),边长s。t表示要填入的相同数字，即三个数字组成的L小棋盘。

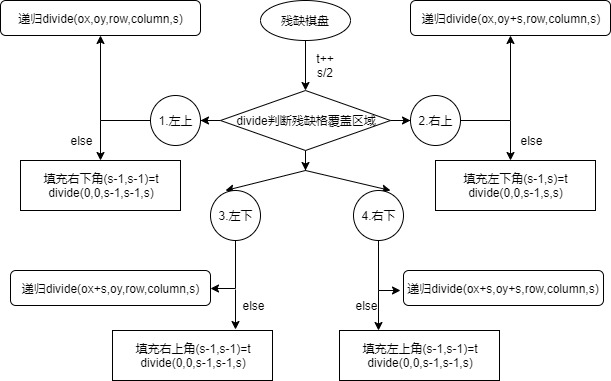


图6 残缺棋盘流程图

1. 主要数据结构的设计

规定了原点（0，0）（也可以更改），输入第一个特殊残缺格的坐标以及棋盘总边长，根据这个坐标进行判断与递归，将数字填满。

1. 数据的设计

要填入的数字每次+1，处理每个子区域时s=size/2,左上区域没有特殊格则填充其右下角，右上对应左下，左下对应右上，右下对应左上（即“十”字交界处以L小棋盘填充）。同步的判断，必定是在某个区域，其余三个区域以数字填充其交界处，这样填满整个棋盘就会不重叠，不遗漏。

1. 部分关键的代码

static int tile = 1;

static int board[32][32];

void divide(int ox,int oy,int row,int column,int size){

    if (size == 1) return;

    int  t = tile++;

    int  s = size/2;

    // 覆盖左上角子棋盘

    if (row<ox+s && column<oy+s) {

        // 特殊方格在此棋盘

        divide(ox,oy,row,column,s);

    } else {

        // 此棋盘中无特殊方格，用t号L型骨牌覆盖右下角\*

        board[ox+s-1][oy+s-1] = t;

        // 覆盖其余方格

        divide(ox,oy,ox+s-1,oy+s-1,s);

    }

    // 覆盖右上角子棋盘

    if (row<ox+s && column>=oy+s) {

        //        特殊方格在此棋盘

        divide(ox,oy+s,row,column,s);

    } else {

        // 此棋盘中无特殊方格，用t号L型骨牌覆盖右上角

        board[ox+s-1][oy+s] = t;

        // 覆盖其余方格

        divide(ox,oy+s,ox+s-1,oy+s,s);

    }

    // 覆盖左下角子棋盘

    if (row>=ox+s && column<oy+s) {

        //  特殊方格在此棋盘

        divide(ox+s,oy,row,column,s);

    } else {

        //  此棋盘中无特殊方格，用t号L型骨牌覆盖左下角

        board[ox+s][oy+s-1] = t;

        //  覆盖其余方格

        divide(ox+s,oy,ox+s,oy+s-1,s);

    }

    // 覆盖右下角子棋盘

    if (row>=ox+s && column>=oy+s) {

        //  特殊方格在此棋盘

        divide(ox+s,oy + s,row,column,s);

    } else {

        // 此棋盘中无特殊方格，用t号L型骨牌覆盖右下角

        board[ox+s][oy+s] = t;

        // 覆盖其余方格

        divide(ox+s,oy+s,ox+s,oy+s,s);

    }

}

int main() {

int length,x,y;

cout << "\*\*原点为(0,0) 且边长为2^n\*\*" << endl;

cout<<"依次输入:特殊方格的x y 总边长"<<endl;

cin>>x>>y>>length;

divide(0,0,x,y,length);

for(int i = 0;i< length;i++) {

for (int j = 0; j<length;j++) {

cout<<setiosflags(ios::left)<<setw(3)<<board[i][j] << " ";

}

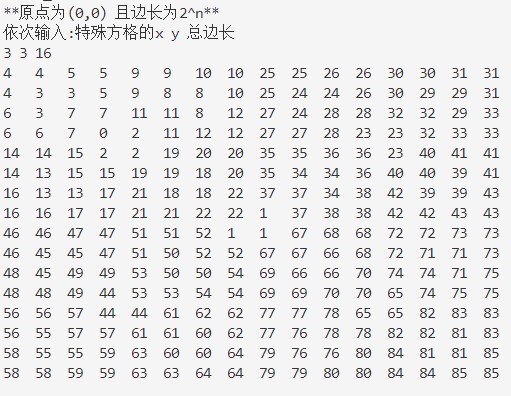
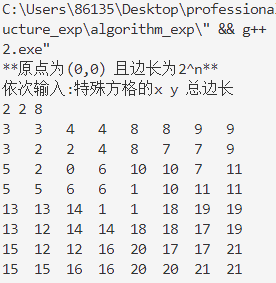
cout << endl;

}

return 0;

}

1. **系统测试**



1. **评价与结论**

（1）递归解决子问题是分治法的核心思想，想要把棋盘按要求填满，就要以一个相同的法则，递归完成子棋盘，子棋盘的子棋盘……

（2）此算法的时间复杂度为4O(k-1)+O(1) k>0

（3）分治每一个区域、以及子区域，并且每一步都是四个区域同步判断，同步递归，最终非常快速地实现最终棋盘。

（4）得益于博客、开源平台的帮助，我了解到许多新思路、新算法，各种奇妙的题目和对应的解虽然刚开始理解起来有些困难，但是只要有耐心，多做题，手感和思路自然会跟上来。以后的编程路还很漫长，要砥砺前行。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程目标** | | **评价准则** | | | **计分** |
| 课程目标1：要求学生掌握软件开发技术，能针对要解决的问题进行建模、对模型的正确性进行分析和论证，以及模型求解。能够使用科学的方法对程序设计理论和技术进行研究，并通过开发过程的信息综合得到合理有效的设计结果。（40分） | | 1、能够掌握软件开发的有关技术和相关知识，能针对要解决的问题收集资料进行合理的分析与设计（10分） | | |  |
| 2、能够给出数据结构和算法的设计描述，给出关键算法的流程图或伪代码（10分） | | |
| 3、能够选择合适的程序设计语言与软件开发平台，对求解的问题进行编程实现（20分） | | |
| 课程目标2：在掌握软件工程领域的相关知识基础上，比较各种软件开发解决方案的优缺点，提高解决复杂工程问题的实践创新能力，优化设计方案。（30分） | | 4、掌握调试方法与工具，对软件开发过程中出现的问题进行分析与调试，并能够进行充分测试（15分） | | |  |
| 5、能够根据测试的情况，合理地对算法进行优化改进（15分） | | |
| 课程目标3：分组完成一次软件项目设计与开发的全过程，组内成员通过讨论和交流解决实践中的难题。在实验报告中准确阐述课程设计的内容，清晰陈述观点和回答问题并能回应来自社会公众的质疑。（30分） | | 6、具备一定的语言表达能力与文字处理能力，报告内容和实验数据详实，格式规范（15分） | | |  |
| 7、能够正确、完整地回答指导教师关于课题的问询，反映其对软件设计技术及相关理论知识具有较好的理解和掌握，能够回应来自公众的质疑（15分） | | |
| 软件基础实践 能力测评总分 | | | | |  |
| **指导教师： 年 月 日** | | | | | |
| **平时成绩（20%）** | **能力测评总分（80%）** | | **总成绩** | **等级** | |
|  |  | |  |  | |