2019-2020学年第一学期算法设计与分析

上机实验报告

实验名称：棋盘覆盖问题

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

棋盘覆盖问题：设 $n=2^k$（$k\geq 0$）。在一个 $n\times n$ 个方格组成的棋盘中，恰有 1 个方格与其他方格不同，称该方格为特殊方格。

现给定 $k>1$，$n=2^k$ 设计一个算法实现棋盘的一种覆盖。

2、实验目的

- 完成棋盘覆盖问题
- 理解分治策略的基本思路
- 学会计算分治算法的时间复杂度
- 尝试优化棋盘覆盖代码

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到带一个特殊方格的 $n=2$ 问题一定有解，考虑将 $n=2^k$ 的问题转化为 4 个 $n=2^{k-1}$ 分治求解。采用递归的方法解决问题，子问题大小为 2 时退出，否则设法将 4 个子区域都带上一个特殊方格。

问题的关键是将子问题划归为“带一个特殊方格”的区域。考虑到 4 个子区域中一定有一个已经包含特殊方格，而剩下 3 个必然包含一个“L”形的角落，必然可以构造出“4 个子区域都带上一个特殊方格”的条件，算法可行性由此得到保证。

2、算法设计与求解步骤

cur：当前已使用骨牌个数

ans：记录每一个位置对应骨牌编号

L：子问题左上角绝对横坐标

U：子问题左上角绝对纵坐标

n：子问题规模（半径）

x：特殊方格绝对横坐标

y：特殊方格绝对纵坐标

- 递归开始时，L=U=1,n 为最大规模，x 和 y 为输入特殊方格坐标。
- 每次递归采用上述思路将一个 n 的问题化为 4 个 n/2 的子问题。
- 每个子问题独立判断特殊方格是否在范围内，如在则直接递归，如不在则取靠近中心一角为特殊方格并赋值。
- 当 n=2 时结束递归，通过 cur 赋值 3 个方块，并将 cur 自增。题干保证初始条件 k>1。
- 最后输出方案。

3、算法分析

考虑到这是一个**分治问题**：

$$T(n)=\begin{cases}O(1) & n\le 2\\ 4T\left(\frac{n}{2}\right)+O(1) & n>2\end{cases}$$

Assume that $n=2^m$, then

$$\frac{T\left(2^m\right)}{4^m}=\frac{T\left(2^{m-1}\right)}{4^{m-1}}+\left(\frac{1}{4}\right)^m$$

$$\Rightarrow \frac{T\left(2^m\right)}{4^m}=\sum_{i=2}^{m}\left(\frac{1}{4}\right)^i\simeq 1$$

$$\Rightarrow T\left(2^m\right)\simeq 4^m$$

$$\Rightarrow T(n)=O\left(n^2\right)$$

考虑到**每一个方格只会被访问和赋值一次**：

$$T(n)=O\left(n^2\right)$$

4、算法程序实现

篇幅原因，见附录或附件。

若老师有心，不妨留意结果分析中提出的优化问题。

三、调试与运行

输入：

```
2 2 3
3 5 3
4 8 10
```

输出：

```
2 2 3
Case 1: n=4
   2   2   4   4
   2   1   #   4
   3   1   1   5
   3   3   5   5

3 5 3
Case 2: n=8
   3   3   5   5  13  13  15  15
   3   2   2   5  13  12  12  15
   4   2   6   6  14  14  12  16
   4   4   6   1   1  14  16  16
   8   8   #  10   1  18  20  20
   8   7  10  10  18  18  17  20
   9   7   7  11  19  17  17  21
   9   9  11  11  19  19  21  21

4 8 10
Case 3: n=16
   4   4   6   6  14  14  16  16  46  46  48  48  56  56  58  58
   4   3   3   6  14  13  13  16  46  45  45  48  56  55  55  58
   5   3   7   7  15  15  13  17  47  45  49  49  57  57  55  59
   5   5   7   2   2  15  17  17  47  47  49  44  44  57  59  59
   9   9  11   2  19  19  21  21  51  51  53  53  44  61  63  63
   9   8  11  11  19  18  18  21  51  50  50  53  61  61  60  63
  10   8   8  12  20  18  22  22  52  52  50  54  62  60  60  64
  10  10  12  12  20  20  22   1  52   #  54  54  62  62  64  64
  25  25  27  27  35  35  37   1   1  67  69  69  77  77  79  79
  25  24  24  27  35  34  37  37  67  67  66  69  77  76  76  79
  26  24  28  28  36  34  34  38  68  66  66  70  78  78  76  80
  26  26  28  23  36  36  38  38  68  68  70  70  65  78  80  80
  30  30  32  23  23  40  42  42  72  72  74  65  65  82  84  84
  30  29  32  32  40  40  39  42  72  71  74  74  82  82  81  84
  31  29  29  33  41  39  39  43  73  71  71  75  83  81  81  85
  31  31  33  33  41  41  43  43  73  73  75  75  83  83  85  85
```

四、结果分析

理论分析和测试结果表面了实验的正确性。

值得注意的是，此处赋值顺序和书上代码是不一致的，这主要是由于本文实现时采用先赋值特殊方块后递归的方法，而书上代码恰好相反。两者都是可行的，且它们是本质相同的。

考虑到分治时 4 个子问题中**至少有 3 个是旋转后本质相同的**，可以**采用同一策略减少递归总数**，时间复杂度变为：

$$T(n)=\begin{cases}O(1) & n\le 2\\ 2T\left(\frac{n}{2}\right)+2\left(\frac{n}{2}\right)^2 & n>2\end{cases}$$

Assume that $n=2^m$, then

$$\frac{T(2^m)}{2^m}\simeq\frac{T(2^{m-1})}{2^{m-1}}+2^m$$

$$\Rightarrow\frac{T(2^m)}{2^m}=\sum_{i=2}^{m}2^i\simeq 2^m$$

$$\Rightarrow T(2^m)\simeq(2^m)^2$$

$$\Rightarrow T(n)=O(n^2)$$

总时间复杂度保持不变。但考虑到递归的调用过程，实战中可能出现常数级差距。由于代码会相对难写一些，且**主要时间花费在 IO 上**，该优化对此题意义不大。但对于其他分治问题，这种思路可能会带来可观的**时间复杂度优化**，如书中矩阵分治乘等。

五、本次实验的收获、心得体会

本次实验考察对分治思想的基本理解。代码不算特别复杂，但写完此题可以对分治思想有基本的理解。

然而不论如何，棋盘覆盖问题使用此思路求解只能算是奇技淫巧。只有当 n 为 2 的幂次时，此方法才能奏效。当棋盘大小为任意规模时，是否还有解？或是说，就拿目前的问题，问有多少本质不同的摆放方式？这些都不是使用此算法就可以解决的问题。

附录：代码实现（也可见附件）

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int cur;
int ans[1025][1025];

// left, upper, scale, the position of the missing block
void dfs(int L,int U,int n,int x,int y){
    int tmp=++cur;
    if (n==2){ // sub problem is in a solvable scale
        for (int i=0;i<2;++i){
            for (int j=0;j<2;++j){
                if (L+i!=x||U+j!=y) ans[L+i][U+j]=tmp;
            }
        }
        return;
    }
    n/=2; // the scale of the sub problem is n/2
    if (x<L+n&&y<U+n)    //top left
        dfs(L,U,n,x,y);
    else{
        dfs(L,U,n,L+n-1,U+n-1);
        ans[L+n-1][U+n-1]=tmp;
    }
    if (x>=L+n&&y<U+n)  //top right
        dfs(L+n,U,n,x,y);
    else{
        dfs(L+n,U,n,L+n,U+n-1);
        ans[L+n][U+n-1]=tmp;
    }
    if (x<L+n&&y>=U+n)  // bottom left
        dfs(L,U+n,n,x,y);
    else{
        dfs(L,U+n,n,L+n-1,U+n);
        ans[L+n-1][U+n]=tmp;
    }
    if (x>=L+n&&y>=U+n) //bottom right
        dfs(L+n,U+n,n,x,y);
    else{
        dfs(L+n,U+n,n,L+n,U+n);
        ans[L+n][U+n]=tmp;
    }
```

```
}

int main(){
    int n,x,y,kase=0;
    while (cin>>n>>x>>y){
        n=(1<<n);
        cout<<"Case "<<++kase<<": n="<<n<<endl;
        ans[x][y]=0;
        cur=0;
        dfs(1,1,n,x,y);
        for (int i=1;i<=n;++i){
            for (int j=1;j<=n;++j){
                if (ans[i][j]==0){
                    cout<<"   #";
                }
                else{
                    cout<<setw(4)<<ans[i][j];
                }
            }
            cout<<endl;
        }
    }
    return 0;
}
```

2019-2020学年第一学期算法设计与分析

上机实验报告

实验名称：矩阵连乘问题

学 号	秦敏浩	姓 名	17122490	评 分	
专 业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

矩阵连乘问题：给定 n 个矩阵 A1，A2，…，An，其中，Ai 与 Aj+1 是可乘的，i=1，2，…，n-l。

你的任务是要确定矩阵连乘的运算次序，使计算这 n 个矩阵的连乘积 A1A2…An 时总的元素乘法次数达到最少。输出值和方案。

2、实验目的

- 完成矩阵连乘问题
- 理解动态规划的基本思想
- 写出此问题的动态方程
- 优化书上代码中计算最优次序的方法

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

采用动态规划的思想求解该问题。已知所有长度为 2 的矩阵区间的计算开销为 p×q×r，且为最优。设长度为 2 到 n 的最优问题已求解完毕（n≥2），则长度为 n+1 的最优问题可以由子最优问题计算得出，从而自底向上求解直至解决整个问题。

考虑到子问题会被约 n 规模的重用，应该开辟空间存储重叠子问题的答案。

在更新当前最优问题时记录其由何处更新而来，即可在求解完成后回溯最优方案，后采用一定技巧即可输出题干要求的输出格式。

2、算法设计与求解步骤

a[x]：存储第 x 个矩阵的行和列的大小

lfa[x][y]：记录区间[x,y]的矩阵最优左乘矩阵的区间

rfa[x][y]：记录区间[x,y]的矩阵最优右乘矩阵的区间

L[x]：记录第 x 个矩阵左侧右括号个数

R[x]：记录第 x 个矩阵右侧左括号个数

dp[x][y]:记录区间[x,y]的矩阵合并最优计算量

- 初始化所有长度为 1 的区间最优计算量为 0，长度为 2 的区间最优计算量为 p×q×r
- 逐渐扩大区间长度，计算最优转移，更新时记录来源
- 输出总区间最优计算量，根据记录回溯最优方案，统计各位置括号数
- 去除冗余括号，输出方案

3、算法分析

给出一个与书上代码略有不同的转移方程，有利于简化代码实现：

$$dp[j,j+i]=\begin{cases}0 & ,\ i=0\\ a[j].x\cdot a[j].y\cdot a[j+i].y & ,\ i=1\\ \min\limits_{0\le k<i}\{dp[j,j+k]+dp[j+k+1,j+i]+a[j].x\cdot a[j+k].y\cdot a[j+i].y\} & ,\ i>1\end{cases}$$

考虑到程序计算量取决于程序中三重循环的循环次数，算法时间上界限为$O(n^3)$，算法空间复杂度为$O(n^2)$。

很容易注意到，在回溯过程中可能产生两种冗余的括号形式：单矩阵添加括号（即“(A1)((A2)(A3))”）和整体冗余括号（即“(A1(A2A3))”）。单矩阵括号可以通过回溯时提前退出解决，而观察到整体冗余括号一定有且仅有一对，可以在初始化时将 A1 的左括号数和 A4 的右括号数记为-1，然后正常更新即可。

考虑到每两个矩阵之间所有左括号出现在右括号前，在更新完 L，R 数组后，

输出方案已经确定。更具体的实现和更详细的代码解释可以参考附件或附录。

4、算法程序实现

篇幅原因，见附录或附件。代码附有详细注释。

三、调试与运行

输入：

```
3

10 100 5 50

4

50 10 40 30 5

6
10 30 20 50 8 42 70
```

输出：

```
3
10 100 5 50
Case 1
7500 (A1A2)A3
4
50 10 40 30 5
Case 2
10500 A1(A2(A3A4))
6
10 30 20 50 8 42 70
Case 3
44320 (A1(A2(A3A4)))(A5A6)
```

代码通过学校 oj 测试。

算法题1 矩阵连乘问题

发布时间：2017年6月19日 00:23　最后更新：2018年9月26日 19:04　时间限制：1000ms　内存限制：128M

运行结果：Accepted

时间：1ms 语言 ：C++

提交时间：2019年10月5日 21:54

四、结果分析

代码在正确时间空间复杂度下通过学校 oj 测试。

考虑现有代码可以有的一些优化：

- lfa，rfa 数组可以优化为同一个数组，因为最优左矩阵左边界一定是 x，最优右矩阵右边界限一定是 y，而最优左矩阵有边界和最优右矩阵左边界一定相邻。
- 同理，a 数组也可以优化至一半存储空间。

五、本次实验的收获、心得体会

相较于棋盘覆盖问题，矩阵连乘问题码量稍大。但在仔细理清转移方程后，就可以比较轻松的一遍写对，毕竟代码核心就一行。

在本次实验中，我尝试用不同于书上的转移方程解决问题，个人认为自己的转移方程可以更加清晰的展现问题的本质，第三重循环变量的意义发生了改变。

此外，我还进一步修改了书上回溯部分代码，使其能够适应题干要求的输出。为了解决冗余括号问题，使用到了一些技巧。

附录：代码实现（也可见附件）

```
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
#define x first
#define y second
```

```cpp
#define INF 10000000 // big enough for this problem

pair<int,int> a[30]; // represents the size of each matrix
pair<int,int> lfa[30][30]; // {a,b}=lfa[x][y] means the optimized left parent of matrix[x to y] is matrix[a to b]
pair<int,int> rfa[30][30]; // {c,d}=rfa[x][y] means the optimized right parent of matrix[x to y] is matrix[c to d]
// apparently a==x && d==y
int L[30],R[30]; // the amount of "["s and "]"s before each matrix
int dp[30][30]; // dp[x][y] equals to the optimized cost to merge matrix[x to y]
int n;

void init(){
    for (int i=0;i<30;++i)
        for (int j=0;j<30;++j)
            dp[i][j]=INF;
    for (int i=0;i<30;++i)
        dp[i][i]=0; // the cost of a single matrix is 0
    for (int i=0;i<n-1;++i)
        dp[i][i+1]=a[i].x*a[i].y*a[i+1].y; // the cost of two connect matrix
    for (int i=0;i<30;++i)
        for (int j=0;j<30;++j)
            lfa[i][j]=rfa[i][j]=make_pair(i,j); // update where the answer is from
    memset(L,0,sizeof(L));
    memset(R,0,sizeof(R));
    --L[0];--R[n-1]; // remove the abundant outer "[]"
}

void dfs(pair<int,int> a){ // dfs to get the answer
    int l=a.x,r=a.y;
    if (l!=r){
        ++L[l];++R[r];
    }
    if (lfa[l][r]!=a) dfs(lfa[l][r]); // a is a single matrix == ( lfa[a.x][a.y]==a )
    if (rfa[l][r]!=a) dfs(rfa[l][r]);
//    printf("(%d,%d)\n",l,r);
}

int main(){
    int T,tmp,kase=0;
    while (~scanf("%d",&n)){ // scanf returns -1 if it reaches EOF, ~(-1)==0==false
        memset(a,0,sizeof(a));
        scanf("%d",&a[0].x);
        for (int i=0;i<n;++i){
            scanf("%d",&a[i].y);
            a[i+1].x=a[i].y;
        }
        a[n].x=0;
        init();
        for (int i=2;i<=n;++i){
            for (int j=0;j+i<n;++j){
                for (int k=0;k<=i-1;++k){
//                    dp[j to j+i] =====> min (dp[j to j+k] + dp[j+k+1 to j+i] + cost) for each k
                    tmp=dp[j][j+i];
                    dp[j][j+i]=min(dp[j][j+i],dp[j][j+k]+dp[j+k+1][j+i]+(a[j].x)*(a[j+k].y)*(a[j+i].y));
                    if (dp[j][j+i]!=tmp){ // update if the cost changes
                        lfa[j][j+i]=make_pair(j,j+k);
                        rfa[j][j+i]=make_pair(j+k+1,j+i);
                    }
                }
            }
        }
        dfs(make_pair(0,n-1)); // get the final answer
        printf("Case %d\n%d ",++kase,dp[0][n-1]);
        for (int i=0;i<n;++i){
            for (int j=0;j<L[i];++j) putchar('(');
            printf("A%d",i+1);
            for (int j=0;j<R[i];++j) putchar(')');
        }
        putchar('\n');
//        for (int j=0;j<n;++j){
//            for (int k=0;k<n;++k){
//                if (dp[j][k]==INF) printf("     .");
//                else printf("%6d",dp[j][k]);
//            }
//            putchar('\n');
//        }
    }
    return 0;
}
```

2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：最长公共子序列

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

最长公共子序列：给定 2 个序列 X、Y，求 X 和 Y 的最长公共子序列 Z。如有潜能，写程序找出所有的最长公共子序列。

2、实验目的

- 完成最长公共子序列问题
- 理解动态规划的要点和性质，写出动态方程
- 找出全部最长公共子序列，比较两种建图方法的优劣
- 改进书中代码

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到任何一个字符串和空串的 LCS 长度为 0。设现已知道 X-1 串和 Y-1 串的 LCS、X 串和 Y-1 串的 LCS、X-1 串和 Y 串的 LCS，则此时 X 串和 Y 串的 LCS 就可以通过以上三者得到——若 X 串和 Y 串末尾字母一致，则为 LCS(X-1,Y-1)+1，否则在 LCS(X,Y-1)和 LCS(X-1,Y)中取大者即可。

考虑在每一次转移时记录路径，正向建边或反向建边形成 DAG 图，即可通过回溯求出所有最长公共子序列。两者在效率上会有不同，这在之后会进行讨论。此外，在记录路径时可采用二进制压位的方法，可以简化书中代码。

2、算法设计与求解步骤

a,b：输入的字符串

from[x][y]：记录 LCS(X,Y)的更新来源

dp[x][y]：记录 LCS(X,Y)

- 空串和任何串的 LCS 大小为 0，记录 LCS[0,y]=LCS[x,0]=0
- 建立两重循环，外层扩展 X 串长度 x，内层扩展 Y 串长度 y
- 根据转移方程完成转移，记录更新来源
- 根据更新来源，回溯所有最长公共子序列，并输出

3、算法分析

动态规划部分：由于每个数组单元计算耗费为 O(1)，因此总复杂度为 O(mn)，其中 m、n 为两个串的长度。动态方程为：

$$dp[i,j]=\begin{cases}0 & i=0 \text{ or } j=0\\ dp[i-1,j-1]+1 & i,j>0;X_i=Y_j\\ \max\left(dp[i,j-1],dp[i-1,j]\right) & i,j>0;X_i\neq Y_j\end{cases}$$

寻找一个 LCS 串：如果从[0,0]开始寻找，则复杂度为 O(mn)；如果从[m,n]开始回溯，则复杂度为 O(m+n)

寻找全部 LCS 串：欧拉拆分函数：约 $O\left(e^{\sqrt{\frac{20}{3}(m+n-2)}}\right)$

简单说明：考虑到最坏情况下 LCS 长度为 0，m*n 的矩阵除边界外所有节点都有向上或向左两个转移方向。此时所有递归路径边界与 m*n 的 ferres 图像定义一致。原问题等价于求 m 个小于等于 n 的自然数的组合方案数。即等价于将整数 m+n 拆分成 n 个正整数无序排列方案数。如果从[0,0]开始寻找，则复杂度更高。

4、算法程序实现

篇幅原因，见附录或附件。

三、调试与运行

输入：

```
2

7 6

A B C B D A B

B D C A B A

8 9

b a a b a b a b

a b a b b a b b a
```

输出：

```
2
7 6
A B C B D A B
B D C A B A

Case 1
LCS(X, Y) = 4

B C B A
B C A B
B D A B
B D A B

  0  0  0  0  0  0  0
  0  0  0  0  1  1  1
  0  1  1  1  1  2  2
  0  1  1  2  2  2  2
  0  1  1  2  2  3  3
  0  1  2  2  2  3  3
  0  1  2  2  3  3  4
  0  1  2  2  3  4  4

000 010 010 010 010 010 010
001 011 011 011 100 010 110
001 100 010 010 011 100 010
001 001 011 100 010 011 011
001 101 011 001 011 100 010
001 001 100 011 011 001 011
001 001 001 011 100 011 100
001 101 001 011 001 100 011
```

```
8 9
b a a b a b a b
a b a b b a b b a

Case 2
LCS(X, Y) = 6

b a b a b a
b a b a b a
a a b a b a
b a b a b a
b a b a b a
b a b a b a
a a b a b a
b a a b b a
b a b a b a
b a b a b a
a a b a b a
b a b a b a
b a b a b a
b a b a b a
a a b a b a
a b a b a b
a b a b a b
b a b b a b
b a b b a b
a a b b a b
a b a b a b
a b a b a b
b a b a b b
b a b a b b
a a b a b b
b a b a b b
b a b a b b
b a b a b b
a a b a b b
a b a b a b
a b a b a b
b a b b a b
b a b b a b
a a b b a b
a b a b a b
a b a b a b

  0  0  0  0  0  0  0  0  0  0
  0  0  1  1  1  1  1  1  1  1
  0  1  1  2  2  2  2  2  2  2
  0  1  1  2  2  2  3  3  3  3
  0  1  2  2  3  3  3  4  4  4
  0  1  2  3  3  3  4  4  4  5
  0  1  2  3  4  4  4  5  5  5
  0  1  2  3  4  4  5  5  5  6
  0  1  2  3  4  5  5  6  6  6

000 010 010 010 010 010 010 010 010 010
001 011 100 010 110 110 010 110 110 010
001 100 011 100 010 010 110 010 010 110
001 101 011 101 011 011 100 010 010 110
001 001 100 011 100 110 011 100 110 010
001 101 001 100 011 011 100 011 011 100
001 001 101 001 100 110 011 100 110 011
001 101 001 101 001 011 100 011 011 100
001 001 101 001 101 100 011 100 110 011
```

四、结果分析

理论分析和测试结果表明了实验的正确性。以下分析对书上代码的优化问题：

回溯时使用二进制优化：书中记录 LCS 更新来源时采用 1,2,3 来记录。这样是无法输出全部最长公共子序列的。一般需要使用 4 种状态进行记录：dp[i,j]由 dp[i-1,j-1]得到；dp[i,j]仅由 dp[i,j-1]得到；dp[i,j]仅由 dp[i-1,j]得到；dp[i,j]仅由 dp[i,j-1]，dp[i-1,j]共同得到。可采用三位二进制来表示这 4 种状态，最高位代表 dp[i-1,j-1]，以此类推。这样在回溯时可以简化代码逻辑。

回溯方向问题：考虑到记录更新来源的本质属于创建一种 DAG 图，包含正向建图和反向建图两种方法。由于反向建图后从[m,n]开始回溯的每条路径都最终到达[0,0]，而正向建图则不然。因此采用反向建图回溯效率会更高。

其他优化：理论上可行的其他优化：寻找全部 LCS 串时进行记忆化操作可以将复杂度减少至 $O\left(m\cdot n\cdot \min\left(m,n\right)\right)$ 即多项式级别；目前代码实现输出**全部最长公共子序列**，采用带状态回溯可以不重复不缺少的找到**本质方案不同的全部最长公共子序列**，采用排序后去重可以找到全部**字符意义上不同的全部最长公共子序列**。

五、本次实验的收获、心得体会

首先，最长公共子序列是一道非常基本而典型的动态规划问题。使用贪心思想决策每一个 dp[i,j]，如果思考清楚很容易推出转移方程。

其次，这是一个开放问题。正如结果分析中所述，题干中对“全部最长公共子序列”的理解有多种不同的方式，由此会产生不同的做法。对于不同的做法，又有着不同的优化方式和写法。

最后，这是一个很有意思的问题。对于不同的需求，不仅代码也会截然不同，甚至还会影响代码时间或空间复杂度。对于寻找全部最长公共子序列问题，可以采用简单回溯问题，此时时间复杂度为指数级，空间复杂度为 O(mn)。若采用带状态的记忆化搜索，则时间复杂度为多项式级，空间复杂度上升为 $O\left(m\cdot n\cdot \min\left(m,n\right)\right)$。此外，对于这些问题的时间复杂度问题也是一个有趣的问题。如之前将问题转化为整数拆分方案数问题，转化过程就比较复杂，而这个问题本身也可以通过动态规划思想进行求解。

附录：代码实现（也可见附件）

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

string ans;
int from[55][55]; // created reversed to get more speed.
char a[55],b[55];         // since there all at most three kinds of froms, use integer to save them
int dp[55][55];

void dfs(int x,int y){ // to find all the LCSs
    if (x==0&&y==0){ // print and return if it get the answer
        for (int i=ans.length()-1;i>=0;--i){ // print in the reversed order
            cout<<ans[i]<<' ';
        }
        cout<<endl;
        return;
    }
    if (from[x][y]&1){            // however not all the froms in the original graph end to [m,n]
        dfs(x-1,y);
    }
    if (from[x][y]&2){
        dfs(x,y-1);
    }
    if (from[x][y]&4){
        ans.push_back(a[x]);
        dfs(x-1,y-1);
        ans.erase(--ans.end()); // to save time
    }
}

int main(){
    int T,kase=0,n,m;
    cin>>T;
    while (T--){
        cin>>m>>n;
        for (int i=0;i<=m;++i){ // initialization
            for (int j=0;j<=n;++j){
                dp[i][j]=0;
            }
        }
        for (int i=1;i<=m;++i){
            from[i][0]=1; // required for tracing back
            cin>>a[i];
        }
        for (int i=1;i<=n;++i){
            from[0][i]=2; // required for tracing back
            cin>>b[i];
        }
        for (int i=1;i<=m;++i){
            for (int j=1;j<=n;++j){
                dp[i][j]=a[i]==b[j]?dp[i-1][j-1]+1:max(dp[i][j-1],dp[i-1][j]);
                from[i][j]=0;
                if (dp[i][j]==dp[i-1][j]){
                    from[i][j]|=1;  // from[i][j]==1: [i,j]->[i-1,j]
                }
                if (dp[i][j]==dp[i][j-1]){
                    from[i][j]|=2; // from[i][j]==-1: [i,j]->[i,j-1]
                }
                if (dp[i][j]==dp[i-1][j-1]+1&&a[i]==b[j]){
                    from[i][j]|=4;  // from[i][j]==0: [i,j]->[i-1,j-1]
                }
            }
        }
        cout<<endl<<"Case "<<++kase<<endl;
        cout<<"LCS(X, Y) = "<<dp[m][n]<<endl<<endl;;
        ans="";
```

```
        dfs(m,n);
        cout<<endl;
        for (int i=0;i<=m;++i){
            for (int j=0;j<=n;++j){
                cout<<setw(3)<<dp[i][j];
            }
            cout<<endl;
        }
        cout<<endl;
        for (int i=0;i<=m;++i){
            for (int j=0;j<=n;++j){
                cout<<setw(4)<<((bitset<3>)from[i][j]);
            }
            cout<<endl;
        }
        cout<<endl;
    }
    return 0;
}
```

2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：哈弗曼编码

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

哈夫曼编码：对给定的 n 个字母（或字）在文档中出现的频率序列，

$$X=\langle x1, x2, \cdots, xn\rangle$$

根据特定的规则，求它们的哈弗曼编码。

2、实验目的

- 完成哈夫曼编码问题
- 理解哈夫曼树创建中的贪心策略
- **改进书上的代码，使得实现更为简单**
- **可视化哈夫曼树产生的过程**

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14，Python3.6（可视化）

二、算法设计与分析

1、算法基本思想

按题意模拟即可。考虑初始状态时，每一个节点都是一棵哈弗曼树。根据题目要求，采用堆排管理优先级次序。此后动态维护这个哈夫曼森林，取出两个优先级最高的进行合并操作直至最后形成一棵完整的哈弗曼树，递归输出哈夫曼编码即可。

考虑到哈弗曼树的编码在每一次合并时可以提前得到，可以进一步**压缩每一棵子树结构**，而**不维护树形**，简化书上代码实现难度。

2、算法设计与求解步骤

方案一：维护哈弗曼树

que：存储当前哈夫曼森林中所有哈夫曼树指针的最小堆

num：每个字符的权值

ans：每个字符对应的哈夫曼编码

- 完整构造出哈弗曼树节点类（权值，生成时间序，左右孩子指针）
- 构造哈弗曼树类（根节点，深度，生成时间序）
- 封装哈夫曼树合并方法，重载哈夫曼树类型指针小于运算符
- 使用优先队列（堆）维护哈弗曼树，并合并至只剩一棵
- 深度优先遍历哈弗曼树，记录哈夫曼编码

方案二：压缩哈弗曼树，不维护树形

num：每个字符的权值

sta：**以栈的形式储存**每个字符对应的哈弗曼编码

- 构造一个类型存放哈夫曼树的所有节点和权值（std::vector）
- 重载小于号，使用优先队列（堆）维护哈弗曼树，并合并至只剩一棵
- **对于每一次合并，遍历两棵子树的节点，记录至 sta 中（左 0 右 1）**
- 输出每一个字符对应的哈弗曼编码

3、算法分析

考虑到无论使用哪一种方案，每一次合并时间复杂度都是 $O(1)$ ，维护优先队列（堆）的时间复杂度为 $O(\log(n))$ ，供需进行 $n-1$ 此更新操作得到完整的哈弗曼树，输出需要时间复杂度 $O(n)$。故总体复杂度为 $O(n\log(n))$ 。空间复杂度有常数级差距。由于方案二不需要维护树形，可以节省约一倍空间，空间复杂度 $O(n)$ 。

4、算法程序实现

篇幅原因，见附录或附件。

三、调试与运行

输入：

2

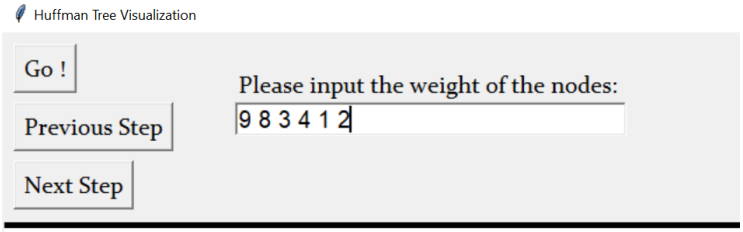
6

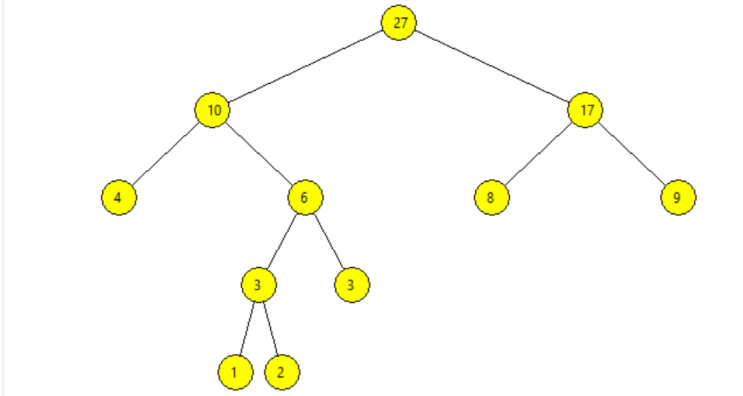
9 8 3 4 1 2

8

60 20 5 5 3 3 3 1

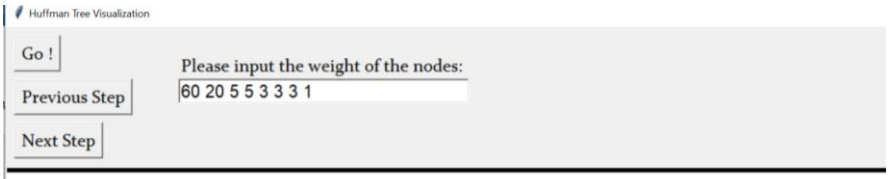
输出：

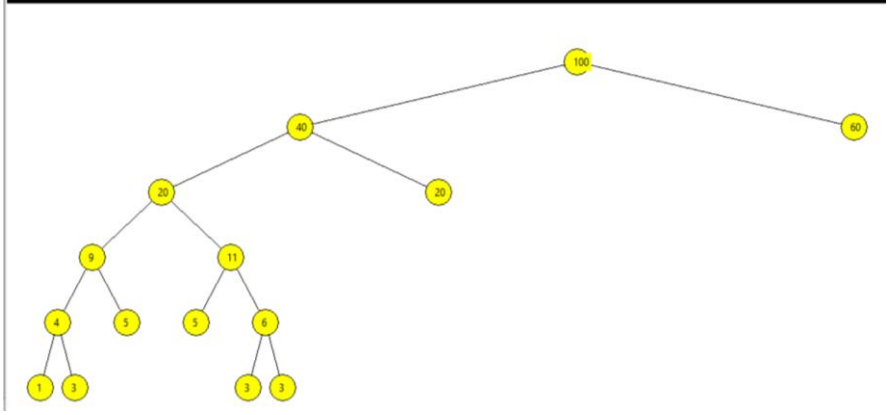





```
2
6
9 8 3 4 1 2
Case 1
9 00
8 01
3 100
4 11
1 1011
2 1010

8
60 20 5 5 3 3 3 1
Case 2
60 0
20 10
5 1101
5 1110
3 11000
3 11001
3 11110
1 11111
```

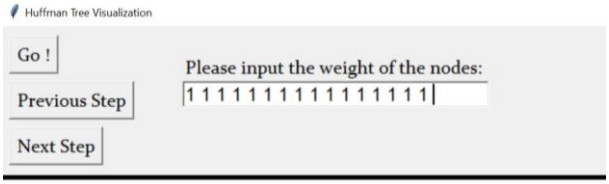


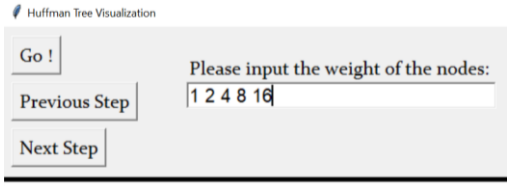


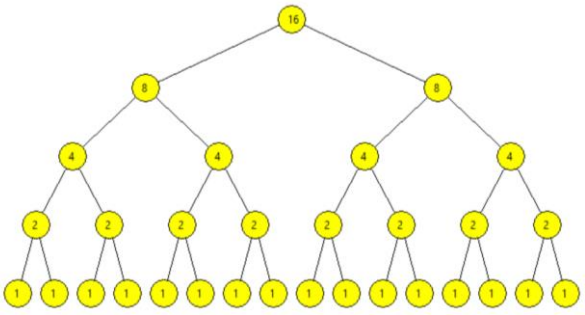
四、结果分析

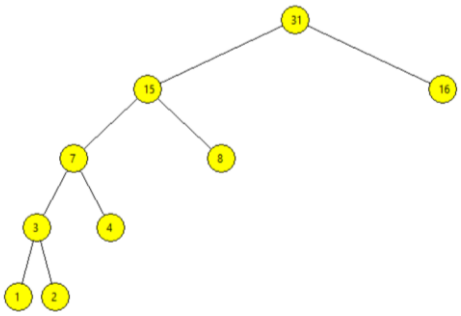
理论分析和测试结果表明了实验的正确性。以下分析哈弗曼树节点绘制时坐标的布局问题：

哈弗曼树的本质属于一种二叉树：考虑一棵满二叉树，最密集的节点分布在其叶子节点一层。每一层节点个数成二的幂次递增。为了树形保证在任何情况下不会出现节点冲突，只需保证满二叉树最下层节点间距即可。纵坐标直接使用简单的深度×单位长度，而横坐标通过维护哈弗曼树双亲节点的坐标和每一层的步长即可得到。如哈夫曼树深度为 n，则根节点横坐标为 2^{n-1} ，第一层步长为 2^n 。每层步长减少一倍，直至叶子节点步长为 1。









五、本次实验的收获、心得体会

对于这个哈夫曼树问题，我还是花了一定的功夫去实现各类版本的代码的。对比附录中各类代码的长度，不难发现哈夫曼树在仅要求输出编码而非输出树形的情况下是有极大代码上的优化空间的（时间复杂度不变）。

后续为了完成哈夫曼树动态可视化问题，我用 C++写了一个哈夫曼树类，然后再根据自己之前想到的 idea 去递归输出节点坐标。事实上，由于涉及指针操作和可视化界面制作，我偷懒地将后端（直至计算完坐标的部分）全部丢给了 C++，而剩下的部分由 Python 完成。两者之间的交互是通过文件进行的。这样的操作是跟着一个知网上看到的研究生大哥学来的。特别对于我这种不是很喜欢泛型语言的人来说有很大的便利。然而考虑到文件 IO 的速度，更应该尝试使用进程间的通信解决这个问题。我想 Python 一定有做这方面的兼容，感兴趣的话我未来可能学一下这样的技术。

对于已经学过哈夫曼树的学生来说，整道题的思想还是比较朴素的。

附录：代码实现（也可见附件）

方案一：构造哈夫曼树

```
/******method 1*****/

#include <iostream>
#include <queue>
#include <cassert>
using namespace std;

class Huffman_Tree_Node{
public:
    int value,id; // only leaves use id to represent the input order
    Huffman_Tree_Node *left_child,*right_child;
    Huffman_Tree_Node(int v=0,Huffman_Tree_Node *l=NULL,Huffman_Tree_Node *r=NULL):value(v),left_child(l),right_child(r){}
    inline void set_id(int x){
        id=x;
    }
};

class Huffman_Tree{
protected:
    int depth,id; // id represents the time order of the tree
    Huffman_Tree_Node *root; // the root of the tree

    void clear(Huffman_Tree_Node *p){ // release memory
        if (p==NULL) return;
        clear(p->left_child);
        clear(p->right_child);
        delete p;p=NULL;
    }

public:
    ~Huffman_Tree(){clear();}

    Huffman_Tree(int value,int c){ // create a tree with an only root
        root=new Huffman_Tree_Node(value);
        root->set_id(c);
        depth=1;id=c;
    }

    Huffman_Tree(Huffman_Tree *l,Huffman_Tree *r,int c){ // create a tree with two subtrees
        assert(l!=NULL);assert(r!=NULL);
        root=new Huffman_Tree_Node(l->get_value()+r->get_value(),l->root,r->root);
        depth=max(l->depth,r->depth)+1;id=c;
    }

    inline int get_depth()const{
        return depth;
    }

    inline int get_value() const{
        assert(root!=NULL);
        return root->value;
    }

    inline Huffman_Tree_Node* get_root() const{
        assert(root!=NULL);
        return root;
    }

    void clear(){
        clear(root);
        root=NULL;
        depth=-1;
    }

    bool operator<(const Huffman_Tree& qmh) const{ // for use of priority queue
        if (get_value()==qmh.get_value()) return id<qmh.id;
        return get_value()>qmh.get_value(); // value of the whole tree is of the first priority
    }
};

struct cmp{
    bool operator()(const Huffman_Tree* a,const Huffman_Tree* b) const{
        return (*a)<(*b);
    }
};

priority_queue<Huffman_Tree*,vector<Huffman_Tree*>,cmp> que;

Huffman_Tree* merge(int &cnt){
    Huffman_Tree *l,*r,*p;
    while (true){
        l=que.top();que.pop(); // left subtree
        if (que.empty()) return l; // return if the task is done
        r=que.top();que.pop(); // right subtree
        p=new Huffman_Tree(l,r,cnt++); // push a new merged tree
        que.push(p);
    }
}

int num[100];
string ans[100];

```

```
void dfs(Huffman_Tree_Node* p,string& cur){ // dfs to get ans, uses & to save time
    if (p->left_child==NULL){
        assert(p->right_child==NULL); // the property of a Huffman tree
        ans[p->id]=cur;
    }
    else{
        assert(p->right_child!=NULL);
        cur.push_back('1'); // right subtree push one
        dfs(p->left_child,cur);
        cur.erase(--cur.end()); // erase it
        cur.push_back('0');  // left subtree push zero
        dfs(p->right_child,cur);
        cur.erase(--cur.end()); // erase it
    }
}

void get_ans(Huffman_Tree* t,int kase,int n){
    cout<<"Case "<<kase<<endl;
    string cur="";
    dfs(t->get_root(),cur); // dfs to get the answers
    for (int i=0;i<n;++i){ // print the answers
        cout<<num[i]<<' '<<ans[i]<<endl;
    }
    cout<<endl;
}

int main(){
    int T,n,x,kase=0;
    cin>>T;
    while (T--){
        int cnt=0;
        cin>>n;
        for (int i=0;i<n;++i){
            cin>>num[i];
            Huffman_Tree* p=new Huffman_Tree(num[i],cnt++); // create a new tree
            que.push(p);
        }
        Huffman_Tree* result=merge(cnt); // merge to get final results
        get_ans(result,++kase,n);
        result->clear(); // release memory
    }
    return 0;
}
```

方案二：压缩哈夫曼树：

```
/***method 2***/
#include <cstdio>
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int cnt;
int num[100];
stack<int> sta[100];  // the answer of each input

struct node{
    int v,id; //v: the value of the tree; id: the time order of the tree
    vector<int> content; // collapse its subtree to a vector
    void init(){
        content.clear();
    }
    bool operator <(const node& qmh) const{ // for priority queue
        if (v==qmh.v) return id<qmh.id;
        return v>qmh.v; // value is for the first priority
    }
};

priority_queue<node> que; // the current forest in ascending order

node merge(node a,node b){// merge two trees into a new one
    node ret;
    ret.v=a.v+b.v; // value accumulates
    ret.id=cnt++; // time order
    for (auto& i:a.content){
```

```
        ret.content.push_back(i);
        sta[i].push(1); // left child push 1
    }
    for (auto& i:b.content){
        ret.content.push_back(i);
        sta[i].push(0); // right child push 0
    }
    return ret;
}

int main(){
    int T,n;
    scanf("%d",&T);
    node tmp,t1,t2;
    for (int kase=1;kase<=T;++kase){
        cnt=0;
        scanf("%d",&n);
        while (!que.empty()) que.pop();
        for (int i=0;i<n;++i){
            tmp.init();
            while (!sta[i].empty()) sta[i].pop();
            scanf("%d",&num[i]);
            tmp.v=num[i];
            tmp.id=cnt++;
            tmp.content.push_back(i); // a new tree
            que.push(tmp);
        }
        while (true){
            t1=que.top();que.pop();
            if (que.empty()) break; // mere tree remain, mission done
            t2=que.top();que.pop();
            que.push(merge(t1,t2)); // more than one tree, merge and push
        }
        printf("Case %d\n",kase);
        for (int i=0;i<n;++i){ // output each answer
            printf("%d ",num[i]);
            while (!sta[i].empty()){
                printf("%d",sta[i].top());
                sta[i].pop();
            }
            putchar('\n');
        }
        putchar('\n');
    }
    return 0;
}
```

哈夫曼树节点坐标计算：

```
/***support***/

#include <iostream>
#include <string>
#include <queue>
#include <cassert>
using namespace std;

struct Huffman_Tree_Node{
    int value,pos[2],id;
    Huffman_Tree_Node *left_child,*right_child;
    Huffman_Tree_Node(int v=0,Huffman_Tree_Node *l=NULL,Huffman_Tree_Node *r=NULL):value(v),left_child(l),right_child(r){}
    inline void set_position(int x,int y) {pos[0]=x;pos[1]=y;}
    inline void set_id(int x) {id=x;}
};
```

```
16
17  class Huffman_Tree{
18  protected:
19      int depth,id;
20      Huffman_Tree_Node *root;
21
22      void clear(Huffman_Tree_Node *p){
23          if (p==NULL) return;
24          clear(p->left_child);
25          clear(p->right_child);
26          delete p;p=NULL;
27      }
28      void set_position(Huffman_Tree_Node *p,int x,int y,int dx){
29          if (p==NULL) return;
30          p->set_position(x,y);
31          set_position(p->left_child,x-dx/2,y+1,dx/2);
32          set_position(p->right_child,x+dx/2,y+1,dx/2);
33      }
34
35  public:
36      ~Huffman_Tree(){clear();}
37      Huffman_Tree(int value,int c){
38          root=new Huffman_Tree_Node(value);
39          root->set_id(c);
40          depth=1;id=c;
41      }
42      Huffman_Tree(Huffman_Tree *l,Huffman_Tree *r,int c){
43          assert(l!=NULL);assert(r!=NULL);
44          root=new Huffman_Tree_Node(l->get_value()+r->get_value(),l->root,r->root);
45          depth=max(l->depth,r->depth)+1;id=c;
46      }
47      inline int get_depth() const{return depth;}
48      inline int get_value() const{
49          assert(root!=NULL);
50          return root->value;
51      }
52      inline Huffman_Tree_Node* get_root() const{
53          assert(root!=NULL);
54          return root;
55      }
56      void clear(){
57          clear(root);
58          root=NULL;
59          depth=-1;
60      }
61      void set_position(int base){
62          assert(root!=NULL);
63          set_position(root,base+(1<<(depth-1)),1,1<<(depth-1));
64      }
65      bool operator<(const Huffman_Tree& qmh) const{
66          if (get_value()==qmh.get_value()) return id<qmh.id;
67          return get_value()>qmh.get_value();
68      }
69  };
70
71  struct cmp{
72      bool operator()(const Huffman_Tree* a,const Huffman_Tree* b) const{
73          return (*a)<(*b);
74      }
75  };
76
77  typedef priority_queue<Huffman_Tree*,vector<Huffman_Tree*>,cmp> Huffman_Tree_Priority_Queue;
78
79
80
81  /*****************************************************************************************************/
82
83
84  const bool logging=true;
85  const bool output=false;
86  Huffman_Tree_Priority_Queue que;
87
88  void show_position(Huffman_Tree_Node *cur,Huffman_Tree_Node *parent=NULL){
89  //    cout<<"Node: ["<<cur->pos[0]<<", "<<cur->pos[1]<<"] value="<<cur->value<<endl;
90      cout<<"1 "<<cur->pos[0]*20<<' '<<cur->pos[1]*75<<' '<<cur->value<<endl;
91      if (parent!=NULL){
92  //        cout<<"Line: ["<<parent->pos[0]<<", "<<parent->pos[1]<<"]<->["<<cur->pos[0]<<", "<<cur->pos[1]<<"]"<<endl;
93          cout<<"2 "<<parent->pos[0]*20<<' '<<parent->pos[1]*75<<' '<<cur->pos[0]*20<<' '<<cur->pos[1]*75<<endl;
94      }
95      if (cur->left_child!=NULL){
96          show_position(cur->left_child,cur);
97          show_position(cur->right_child,cur);
98      }
99  }
100
101 void display(){
102     if (!logging) return;
103     Huffman_Tree_Priority_Queue q=que;
104     int x=1;
105 //    cout<<"round"<<endl;
106     cout<<"-1"<<endl;
107     while (!q.empty()){
108         Huffman_Tree *p=q.top();q.pop();
109         p->set_position(x);
110         show_position(p->get_root());
111         x+=(1<<p->get_depth())+1;
```

```
    }
}

Huffman_Tree* merge(int &cnt){
    Huffman_Tree *l,*r,*p;
    while (true){
        display();
        l=que.top();que.pop();
        if (que.empty()) return l;
        r=que.top();que.pop();
        p=new Huffman_Tree(l,r,cnt++);
        que.push(p);
    }
}

int num[100];
string ans[100];

void dfs(Huffman_Tree_Node* p,string& cur){
    if (p->left_child==NULL){
        assert(p->right_child==NULL);
        ans[p->id]=cur;
    }
    else{
        assert(p->right_child!=NULL);
        cur.push_back('1');
        dfs(p->left_child,cur);
        cur.erase(--cur.end());
        cur.push_back('0');
        dfs(p->right_child,cur);
        cur.erase(--cur.end());
    }
}

void get_ans(Huffman_Tree* t,int kase,int n){
    if (!output) return;
    cout<<"Case "<<kase<<endl;
    string cur="";
    dfs(t->get_root(),cur);
    for (int i=0;i<n;++i){
        cout<<num[i]<<' '<<ans[i]<<endl;
    }
    cout<<endl;
}

int main(){
    freopen("data.in","r",stdin);
    freopen("data.out","w",stdout);
    int T,n,kase=0;
    cin>>T;
    while (T--){
        int cnt=0;
        cin>>n;
        for (int i=0;i<n;++i){
            cin>>num[i];
            Huffman_Tree* p=new Huffman_Tree(num[i],cnt++);
            que.push(p);
        }
        Huffman_Tree* result=merge(cnt);
        get_ans(result,++kase,n);
        result->clear();
    }
    cout<<"-2"<<endl;
    return 0;
}
```

可视化（Python 实现）：

```
# visualize

import tkinter as tk
import os

window = tk.Tk()
window.title("Huffman Tree Visualization")
window.geometry('1920x1080')
canvas = tk.Canvas(window, bg="white", height=980, width=1920)
input_entry = tk.Entry(window, bd=2, font="Constaintia", width=30)

step = -1
nodes = []
lines = []
numbers = []
tag_list = []


def print_screen():
    global tag_list
    canvas.delete('all')
    canvas.create_line(0, 5, 1920, 5, width=5)
    if step != -1:
        for line in lines[step]:
```

```
            canvas.create_line(line[0], line[1], line[2], line[3])
        for node in nodes[step]:
            canvas.create_oval(node[0] - 15, node[1] - 15, node[0] + 15, node[1] + 15, fill="yellow")
        for elem in tag_list:
            elem.destroy()
        tag_list = []
        for num in numbers[step]:
            tag_list.append(tk.Label(window, text=str(num[2]), bg="yellow"))
            tag_list[len(tag_list) - 1].place(x=num[0] - 7, y=num[1] + 150)


def previous_step():
    global step
    step = max(0, step - 1)
    print_screen()


def next_step():
    global step, nodes
    step = min(len(nodes) - 1, step + 1)
    print_screen()


def read():
    global nodes, lines, step, numbers
    nodes = []
    lines = []
    step = 0
    numbers = []
    content = input_entry.get()
    try:
        out = list(map(int, content.split()))
        assert len(out) != 0, "empty input"
        with open("data.in", 'w') as FILE:
            FILE.write('1\n' + str(len(out)) + '\n')
            for elem in out:
                FILE.write(str(elem) + ' ')
            FILE.write('\n')
        os.system("back_up.exe")
        with open("data.out", 'r') as FILE:
            content = FILE.read()
        for line in content.split('\n'):
            cur = list(map(int, line.split()))
            if cur[0] == -2:
                break
            elif cur[0] == -1:
                lines.append([])
                nodes.append([])
                numbers.append([])
            elif cur[0] == 1:
                nodes[len(nodes) - 1].append([cur[1], cur[2]])
                numbers[len(numbers) - 1].append([cur[1], cur[2], cur[3]])
            else:
                lines[len(lines) - 1].append([cur[1], cur[2], cur[3], cur[4]])
    except Exception as err:
        nodes = []
        lines = []
        numbers = []
        step = -1
        print(str(err))
    print_screen()


def main():
    global step
    tk.Button(window, text='Go !', font="constantia", command=read).place(x=10, y=10)
    tk.Button(window, text='Previous Step', font="constantia", command=previous_step).place(x=10, y=60)
    tk.Button(window, text='Next Step', font="constantia", command=next_step).place(x=10, y=110)
    tk.Label(window, text="Please input the weight of the nodes: ", font="constantia").place(x=200, y=30)
    input_entry.place(x=200, y=60)
    print_screen()
    canvas.place(x=0, y=160)
    window.update()
    window.mainloop()


if __name__ == '__main__':
    main()
```

2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：Djikstra 算法

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.13	实验学时	2		

一、实验问题描述

1、实验内容

Dijkstra 算法：对给定的一个（有向）图 G，及 G 中的两点 s、t，确定一条从 s 到 t 的最短路径。

附加要求：如果有多条最短路径，输出路段最少者，如果还有多条，输出字典序最小者。

2、实验目的

- 使用 Dijkstra 算法完成问题
- 理解分支界限法的基本思想，完成附加要求中的内容
- 探讨对 Dijkstra 算法的优化问题

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到在单源点最短路问题中，源点到自身的最短路一定为零。在所有与源点直接相连的点中，路径长度最短的边就一定属于该图的最短路树。因此，对于每一次加入的新的边，贪心地更新其对最短路的贡献，即可得到此连通块中单源点到其他所有目标点中的最短路径长度。

考虑在每一次松弛操作时记录更新来源，即可得到最短路径。**反向建图可贪心使字典序最小**。是否进行松弛操作逻辑需要根据题目要求进行调整。

2、算法设计与求解步骤

vis：当前该节点是否已经在最短路树上（或者说是否其最短路长度是否已经被确定）

G：原图的邻接矩阵，使用-1 表示无边

path：记录当前到此节点的最短路长度，路段个数和更新来源

- 重载路径节点类的小于运算符，用于更新时比较
- **反向建图**，寻找从终点向起点的最短路
- 初始化源点（终点）至自己的距离为 0，且访问过。其余节点均未访问。
- 在每一步中挑出最短路径长度最短的未被 vis 的节点，将其访问
- 考虑 vis 节点的所有松弛，实用小于运算符判断是否松弛
- 如果进行松弛，则记录更新来源，并更新最短路
- 重复以上操作直至所有节点都被 vis
- 从汇点（起点）回溯寻找符合要求的最短路

3、算法分析

考虑到 Dijkstra 算法的本质属于一种分支限界算法。对于一个 n 个节点的图，最多进行 n-1 次循环，每次循环确定一个活动节点。如果使用邻接矩阵，每次循环需要对所有 n 个点进行松弛检查。每次松弛操作时间复杂度为$O(1)$，故使用邻接矩阵的普通 Dijkstra 算法时间复杂度为$O(n^2)$，空间复杂度同样为$O(n^2)$。

回溯时最大深度为 n，每次回溯分支数为 1，时间成本为$O(1)$，故输出路径的时间复杂度为$O(n)$。

4、算法程序实现

篇幅原因，见附录或附件。

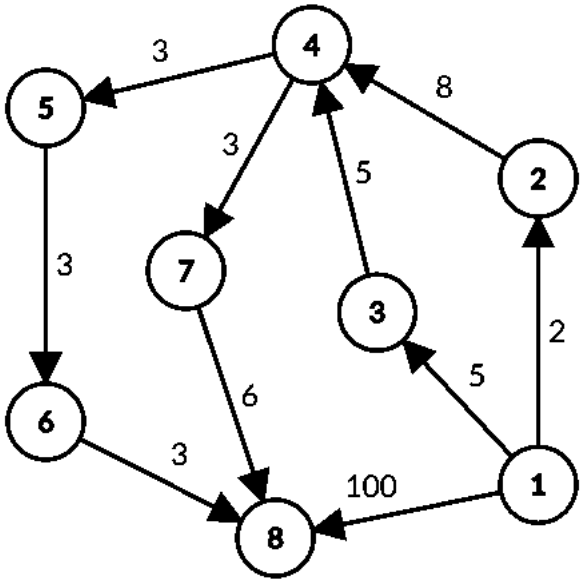
三、调试与运行

输入：

```
5
-1 10  -1 30 100
-1 -1 50 -1  -1
-1 -1 -1 -1 10
-1 -1 20 -1 60
-1 -1 -1 -1 -1
1 5

6
-1 1 12 -1 -1 -1
-1 -1 9 3 -1 -1
-1 -1 -1 -1 5 -1
-1 -1 4 -1 13 13
-1 -1 -1 -1 -1 4
-1 -1 -1 -1 -1 -1
1 6

8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
```



输出：

```
5
-1 10  -1 30 100
-1 -1 50 -1  -1
-1 -1 -1 -1 10
-1 -1 20 -1 60
-1 -1 -1 -1 -1
1 5
Case 1
The least distance from 1->5 is 60
the path is 1->4->3->5

6
-1 1 12 -1 -1 -1
-1 -1 9 3 -1 -1
-1 -1 -1 -1 5 -1
-1 -1 4 -1 13 13
-1 -1 -1 -1 -1 4
-1 -1 -1 -1 -1 -1
1 6
Case 2
The least distance from 1->6 is 17
the path is 1->2->4->6

8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
Case 3
The least distance from 1->8 is 19
the path is 1->2->4->7->8
```

四、结果分析

首先，原题干中**描述有误**（无向图却给了非对称矩阵作为邻接矩阵）。报告中已做修正。

反向建图的正确性保证：考虑到有向图中任何一个由 s 到 t 的路径，在其反向图中，都有一条同样的 t 到 s 的反向路径。由最短路的定义可知，这两个问题时等价的。

反向建图的原因：在路径长度和路段数相同的条件下，为了保证输出满足字典序最小，可以采用**类似基数排序**的贪心算法——从最低位开始每次都尽可能取来源节点编号最小的那一条边。不同于基数排序的是，该贪心算法只需要保留最小的那个编号即可，而不需要维护所有的编号。为了实现“从最低位开始贪心”的操作，采取了反向建图的操作。

事实上，本题采用的是没有堆优化的 Dijkstra 算法。如果进一步使用堆对本题进行优化，可以将最短路部分时间复杂度缩减到 $O(n\log n)$ 。但考虑到邻接矩阵输入方式的 IO 就已经到了 $O(n^2)$ 的规模，总时间复杂度保持不变。

五、本次实验的收获、心得体会

本次实验考察对分支限界算法的理解，同时单源最短路问题也是一个必须要掌握的图论基本算法。事实上，Dijkstra 算法早在数据结构和离散数学课堂上就已经有提到。但堆优化 Dijkstra 问题是第一次出现的。

对于本实验中的附加部分，如果能够清晰地理解 Dijkstra 算法的本质，是很好实现的。在实现过程前或过程中，需要想清楚字典序最小与原图中边来源的对应关系，才不至于在记录路径的时候提升时间复杂度和空间复杂度。

附录：代码实现（也可见附件）

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

const int maxn=55; // maximum of n
const int inf=0x3f3f3f3f;   // infinity

struct node{
    int dis,len,pre; // distance, length, previous node number
    node(){}
    node(int d,int l,int p):dis(d),len(l),pre(p){}
    void init(){
        dis=len=pre=inf;
```

```
    }
    bool operator<(const node &qmh) const{
        if (dis!=qmh.dis) return dis<qmh.dis;    // distance is of the first priority
        if (len!=qmh.len) return len<qmh.len;    // then the length of the route
        return pre<qmh.pre;                      // then the previous node number
    }
}path[maxn];

bool vis[maxn];  // whether it is visited
int G[maxn][maxn]; // the graph

void dijkstra(int n,int s){ // node count, start point
    for (int i=1;i<=n;++i){
        vis[i]=false;
        path[i].init();
    }
    vis[s]=true; // start point has been visited
    path[s]={0,0,-1}; // update the path of the start point
    for (int i=1;i<=n;++i){ // the conjuncted nodes of the start point
        if (G[s][i]!=-1){
            path[i]={G[s][i],1,s};
        }
    }
    while (true){
        int next=-1;
        for (int i=1;i<=n;++i){
            if (!vis[i]&&(next==-1||path[i].dis<path[next].dis)){
                next=i; // find the node of the first priority
            }
        }
        if (next==-1) break; // break if all the connected nodes are visited
        vis[next]=true;
        for (int i=1;i<=n;++i){
            if (vis[i]||G[next][i]==-1) continue;
            node update(path[next].dis+G[next][i],path[next].len+1,next);
            if (update<path[i]){
                path[i]=update;
            }
        }
    }
}

void get_path(int p){
    if (path[p].pre==-1){ // already found the end of the route
        cout<<p<<endl;
        return;
    }
    cout<<p<<"->";
    get_path(path[p].pre);
}

int main(){
    int n,t1,t2,kase=0;
    while (cin>>n){
        for (int i=1;i<=n;++i){
            for (int j=1;j<=n;++j){
                cin>>G[j][i];
            }
        }
```

```
        cin>>t1>>t2;
        dijkstra(n,t2);
        cout<<"Case "<<++kase<<endl;
        cout<<"The least distance from "<<t1<<"->"<<t2<<" is "<<path[t1].dis<<endl;
        cout<<"the path is ";
        get_path(t1);
    }
    return 0;
}

/*
8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
*/
```

2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：跳马问题

学 号	秦敏浩	姓 名	17122490	评 分	
专 业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.17	实验学时	2		

一、实验问题描述

1、实验内容

跳马问题：给定 8*8 方格棋盘，求棋盘上一只马从一个位置到达另一位置的最短路径长。

2、实验目的

- 分析问题，说明原理和方法
- 正确实现跳马问题
- 探讨使用 DFS 与 BFS 求解此题的效率和可能的问题

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到每一个位置最多有 8 个可能的转移。从起点出发，遍历每一种转移，若出现已经被走过的情况，则不进行尝试；否则，继续尝试下一步，直至走到目的地为止。

该题本质是一个 64 节点的无向图求任一两点之间的最短路问题，可以使用弗洛伊德算法一次性解决，或使用 Dijkstra 算法。同理，也可以使用深度优先搜索或广度优先搜索的方法解决。无论表现形式如何，它们的本质都是在寻找图上最短路。

2、算法设计与求解步骤

vis：当前该节点被走到过的最短时间，若未访问为-1

d：常量，记录马可以走的八个转移方向

que：目前仍然可以更新的节点组成的队列

- 将起点放入队列，标记起点 vis 为 0
- 每次取出队列中第一个元素，如果是目标点，输出 vis，结束
- 遍历该节点的八个转移，如有合法未访问的转移，将其加入队列
- 重复上述过程直至队列为空或走到目标点为止

3、算法分析

考虑到每一个节点最多会进入队列一次，最多被其他节点访问八次，对于一个 n*n 的棋盘，采用 BFS 算法求解时间复杂度为每次询问 $O(n^2)$ 。由于采用 DFS 算法可能会导致一个节点被多次访问，只能在当前访问时间大于 vis 时进行剪枝，代码效率更低，时间复杂度可能是指数级的。

4、算法程序实现

篇幅原因，见附录或附件。

三、调试与运行

输入：

```
a1 a2
a1 a3
a1 h8
g2 b8
```

输出：

```
a1 a2
a0==>a1: 3 moves
a1 a3
a0==>a2: 2 moves
a1 h8
a0==>h7: 6 moves
g2 b8
g1==>b7: 5 moves
```

四、结果分析

本题做法多样，再此分析各算法的优点和缺点。

弗洛伊德：复杂度$O\left(n^6+q\right)$，可以直接求出任意两点之间需要最少移动的步数，和询问次数是并行关系。

Dijkstra：复杂度$O\left(q\cdot n^2\log n\right)$，考虑到**原图稀疏**，边数约为 4 倍点数，故 Dijkstra 是一种可以考虑的解法。常数较大。

BFS：复杂度$O\left(q\cdot n^2\right)$，证明如上。

DFS：复杂度为指数级，原因如上。

五、本次实验的收获、心得体会

本次实验是一道我也不知道为什么会出现在这里的题目。解法的思路十分多样，对于输入规模的数据大小，无论采用何种算法应该都是可以容忍的（DFS 除外）。

由于接触 BFS 算法较早已经习以为常，加之时间紧张，没有给出有关 BFS 算法合理性的证明。BFS 算法的本质是需要保证队列里的 vis 值是单调不减的，不然就会产生答案错误的问题。当然，这也是为什么 Dijkstra 不能跑负环的根本原因。

此外，此题可能还有复杂度更加优秀的做法。例如当棋盘规模达到 10000*10000 时，即使使用 BFS 可能也不可容忍。可以采用一定的贪心地策略去保证答案正确的情况下将问题划归到一个很小的规模（如目的地周围 80*80 的大小）去求解，可以得到更好的结果。

附录：代码实现（也可见附件）

```cpp
#include <cstring>
#include <cstdio>
#include <queue>
using namespace std;

int vis[10][10]; // minimal time to get there (-1: unvisited)
int d[8][2]={{2,1},{2,-1},{-2,1},{-2,-1},{1,2},{1,-2},{-1,2},{-1,-2}}; // 8 directions

inline bool can_vis(int x,int y){ // check if the chess is out of the board
    if (x<0||x>=8) return false;
    if (y<0||y>=8) return false;
    return true;
```

```
}

int main(){
    int x1,x2,y1,y2,curx,cury,nxtx,nxty;
    char in[5];
    while (fgets(in,10,stdin)){ // scan until it reaches '\n'
        x1=in[0]-'a'; // transfer to number
        x2=in[1]-'1';
        y1=in[3]-'a';
        y2=in[4]-'1';
        memset(vis,-1,sizeof(vis)); // set all the places unvisited
        queue<pair<int,int>> que; // a queue for BFS
        que.emplace(x1,x2);
        vis[x1][x2]=0;
        while (!que.empty()){
            curx=que.front().first; // current position x
            cury=que.front().second; // current position y
            que.pop();
            if (curx==y1&&cury==y2){ // already at the destination
                printf("%c%d==>%c%d: %d moves\n",'a'+x1,x2,'a'+y1,y2,vis[curx][cury]);
                break;
            }
            for (int i=0;i<8;++i){ // try to get a next step
                nxtx=curx+d[i][0]; // next position x
                nxty=cury+d[i][1]; //next position y
                if (!can_vis(nxtx,nxty)) continue; // out of the board
                if (vis[nxtx][nxty]!=-1) continue; // already visited
                vis[nxtx][nxty]=vis[curx][cury]+1;
                que.emplace(nxtx,nxty);
            }
        }
    }
    return 0;
}
```

2019-2020学年第一学期算法设计与分析

上机实验报告

实验名称：装载问题

学 号	秦敏浩	姓 名	17122490	评 分	
专 业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.17	实验学时	2		

一、实验问题描述

1、实验内容

装载问题：有 n 个集装箱要装上 2 艘载重量分别为 c_1 和 c_2 的轮船，其中第 i 个集装箱的重量为 w_i，要求确定是否有一个合理的装载方案可将这个集装箱装上这 2 艘轮船。如果有，找出一种装载方案。

2、实验目的

- 分析问题，说明原理和方法
- 使用 BFS 或 DFS 求解该问题
- 将时间复杂度限制在 $O(2^n)$

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

Brute force。考虑 n 个物品最多有 2^n 种状态可以枚举。如果采用 DFS 的方法进行，每一层决策当前物品分给 1 号船还是 2 号船，通过参数维护当前船的重量，即可在总体时间复杂度为 $O(2^n)$ 下通过此题。

特别的，当物品总重量超过船的总承载能力时，答案一定是不存在的。当递归到某一层物品既无法放入 1 号船又无法放入 2 号船时，可以进行**可行性剪枝**。

2、算法设计与求解步骤

ans，bestans：用 01 串的形式记录当前决策和最佳决策

a：每个物品的重量

w1，w2：当前船 1 和船 2 的剩余载重能力

- 输入 w1，w2 和 a
- 从第一个物品开始 dfs，直至深度为 n
- 对于每一次 dfs，尽可能将物品往 w1 船放，其次尝试往往 w2 船放
- 当能成功放置完最后一个物品时输出答案，并更新最佳决策

3、算法分析

考虑到这是一个深度为 n 的递归，而每一层分支数为 2 个，处理复杂度为 $O(1)$，故总时间复杂度为$O(2^n)$。

4、算法程序实现

篇幅原因，见附录或附件。

三、调试与运行

输入：

```
3
10 40 40
50 50
3
20 40 40
50 50
```

输出：

```
3
10 40 40
50 50
Case 1
answer: 50 110
answer: 50 101
answer: 40 010
answer: 40 001
Best answer: 50 110
3
20 40 40
50 50
Case 2
No
```

四、结果分析

本题属于最基本的 Brute Force 做法。按照最朴素的想法求解问题即可。

由于题目输入没有保证每一个物品重量之间的关系，又规定了当有多种方案时输出字典序最小的方案，代码优化空间不大。

但如果要求输出任意一种方案时，可以先 $O(n\log n)$ 将每个物品按重量从大到小排序，这样剪枝的深度就会更浅一些。

五、本次实验的收获、心得体会

本次实验可以说是所有实验中最简单的一个，使用最朴素的思路和解法。

实际上本次实验的代码的本质 一个二进制枚举。若将 n 个物品的选择映射到 0 至 2 的 n 次方减一的话，那么每一个 n 位二进制数恰好对应一种枚举方式。但由于如此做的检查合法复杂度为 $O(n)$ ，影响到总体时间复杂度为 $O(n\cdot 2^n)$ 。考虑到可以边枚举边维护，就自然的产生了 DFS 的做法。

附录：代码实现（也可见附件）

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

int n,w1,w2,bestw; // w1,w2: the weight the boats can take currently
int a[25];
string ans,bestans; // string that takes the answer

void dfs(int p,int sum){
    if (p==n){
        if (sum>bestw){ // if sum==bestw then the lexicographical order must be lower
            bestw=sum;
            bestans=ans;
        }
        cout<<"answer: "<<sum<<' '<<ans<<endl; // a feasible answer (not necessarily the best)
        return;
    }
    if (a[p]<=w1){ // boat 1 takes
        w1-=a[p];
        ans.push_back('1'); // add 1
        dfs(p+1,sum+a[p]);
        ans.erase(--ans.end()); // delete the 1
        w1+=a[p];
    }
    if (a[p]<=w2){ // boat 2 takes
        w2-=a[p];
        ans.push_back('0');
        dfs(p+1,sum);
        ans.erase(--ans.end());
        w2+=a[p];
    }
}

```

```cpp
int main(){
    int kase=0;
    while (cin>>n){ // do until EOF
        int sum=0;
        for (int i=0;i<n;++i){
            cin>>a[i];
            sum+=a[i];
        }
        cin>>w1>>w2;
        cout<<"Case "<<++kase<<endl;
        if (sum>w1+w2){ // obviously "No"
            cout<<"No"<<endl;
            continue;
        }
        bestw=-1;
        ans=bestans="";
        dfs(0,0); // O(2^n) to find all the solutions
        if (bestw==-1){
            cout<<"No"<<endl; // still possible that there is no solution existing
        }
        else{
            cout<<"Best answer: "<<bestw<<' '<<bestans<<endl; // else output the best solution
        }
    }
    return 0;
}
```