算法复习提纲

一、 算法效率分析

算法效率分析主要涉及以下几个方面：

输入规模的度量：确定算法的输入参数和问题的规模之间的关系，例如排序算法的输入规模就是待排序的元素个数。

运行时间的度量：确定算法的基本操作和执行次数之间的关系，例如顺序查找算法的基本操作就是比较元素的值，执行次数取决于元素在列表中的位置。

增长次数：确定算法的执行次数随着输入规模的增长而增长的趋势，例如二分查找算法的执行次数随着输入规模的增长呈对数增长，记为 O(logn)。

最优、最差和平均效率：确定算法在不同输入情况下的运行时间的上界、下界和期望值，例如顺序查找算法的最优效率是 O(1)，最差效率是 O(n)，平均效率是 O(n/2)。

算法复杂度分析，这次真懂了 - 知乎：这篇文章介绍了算法复杂度分析的基本概念和方法，以及常见的时间复杂度和空间复杂度的例子。

算法效率分析_算法的效率-CSDN 博客：这篇文章介绍了算法效率分析的目的和意义，以及如何选择合适的算法来解决问题。

【算法】算法效率分析（一） - CSDN 博客：这篇文章介绍了算法效率分析的基本步骤和原则，以及如何使用大 O 记法

来表示算法的时间复杂度和空间复杂度。

算法时间效率分析-CSDN 博客：这篇文章介绍了算法时间效率分析的重要性和方法，以及如何使用增长次数和最优、最差和平均效率来评估算法的性能。

二、 算法概念

分治法：分治法是一种将原问题分解为若干个规模较小的子问题，递归地求解子问题，然后合并子问题的解得到原问题的解的方法。分治法要求子问题相互独立，且与原问题具有相同的结构。分治法通常采用递归的方式实现，适用于问题的规模缩小后，可以容易地解决的情况。分治法的典型例子有二分查找，归并排序，快速排序等。

动态规划法：动态规划法是一种将原问题分解为若干个子问题，按照一定的顺序求解子问题，利用子问题的解构造原问题的解的方法。动态规划法要求子问题具有最优子结构性质，即子问题的最优解可以用于求解原问题的最优解。动态规划法通常采用自底向上或自顶向下的方式实现，适用于问题的最优解依赖于其子问题的最优解的情况。动态规划法的典型例子有斐波那契数列，最长公共子序列，背包问题等。

蛮力法：蛮力法是一种直接根据问题的描述和定义，枚举所有可能的解，然后选择满足条件的解或最优的解的方法。蛮力法不需要分解问题，也不需要利用子问题的解，只需要进行简单的计算或比较。蛮力法通常采用循环或递归的方式实现，适用于问题的解空间较小，且没有更好的算法的情况。蛮力法的典型例子有字符串匹配，素数判断，全排列等。

回溯法：回溯法是一种在问题的解空间树上进行深度优先搜索的方法，从根结点出发，沿着一条路径向下探索，当发现当前路径不能达到目标时，就回溯到

上一个结点，选择另一条路径继续探索，直到找到所有的解或满足条件的解为止。回溯法要求子问题不相互独立，且与原问题具有相同的约束条件。回溯法通常采用递归的方式实现，适用于问题的解空间较大，且需要剪枝的情况。回溯法的典型例子有八皇后问题，子集和问题，图的着色问题等。

分支限界法：分支限界法是一种在问题的解空间树上进行广度优先或最小消耗优先搜索的方法，从根结点出发，按照一定的规则扩展结点，利用限界函数剪去不可能达到目标的结点，直到找到一个解或满足条件的解为止。分支限界法要求子问题不相互独立，且与原问题具有相同的目标函数。分支限界法通常采用队列或优先队列的方式实现，适用于问题的解空间较大，且需要求解最优解的情况。分支限界法的典型例子有单源最短路径问题，旅行商问题，0-1 背包问题等。

算法期末复习总结 || 分治、蛮力、回溯、分支限界、贪心、动态规划算法分析和比较：这篇文章详细地比较了这些算法思想的特点和适用场景，以及给出了一些常见的问题和解法。

五大经典算法（贪婪、动态规划、分治、回溯、分支限界法）及其联系和比较：这篇文章总结了这些算法思想的定义和步骤，以及给出了一些经典的问题和解法。

分治法，动态规划及贪心算法区别：这篇文章用一些简单的例子来说明分治法，动态规划法和贪心法的区别和联系。

2.问题和背包容量问题

背包问题是指给定一组物品，每个物品有自己的重量和价值，以及一个背包，背包有一定的容量，要求从物品中选择若干个放入背包，使得背包中的物品的总价值最大化。背包问题有多种变形，例如 0-1 背包问题，完全背包问题，多重背包问题等[1]。

背包容量问题是指给定一组物品，每个物品有自己的重量和价值，以及一个背包，背包有一定的容量，要求从物品中选择若干个放入背包，使得背包中的物品的总重量最小化。背包容量问题是背包问题的一种特殊情况，可以看作是背包问题中的最小化问题[2]。

背包问题和背包容量问题的区别主要在于目标函数的不同，前者是最大化价值，后者是最小化重量。它们的解法也有所不同，一般来说，背包问题可以用动态规划或贪心算法来解决，而背包容量问题可以用分支限界或遗传算法来解决[3]。

动态规划-背包问题(01 背包、完全背包、多重背包 ...：这篇文章详细地介绍了背包问题的各种变形和动态规划算法的实现。

咱就把 0-1 背包问题讲个通透！ - 知乎专栏：这篇文章用一个具体的例子来解释 0-1 背包问题的原理和步骤。

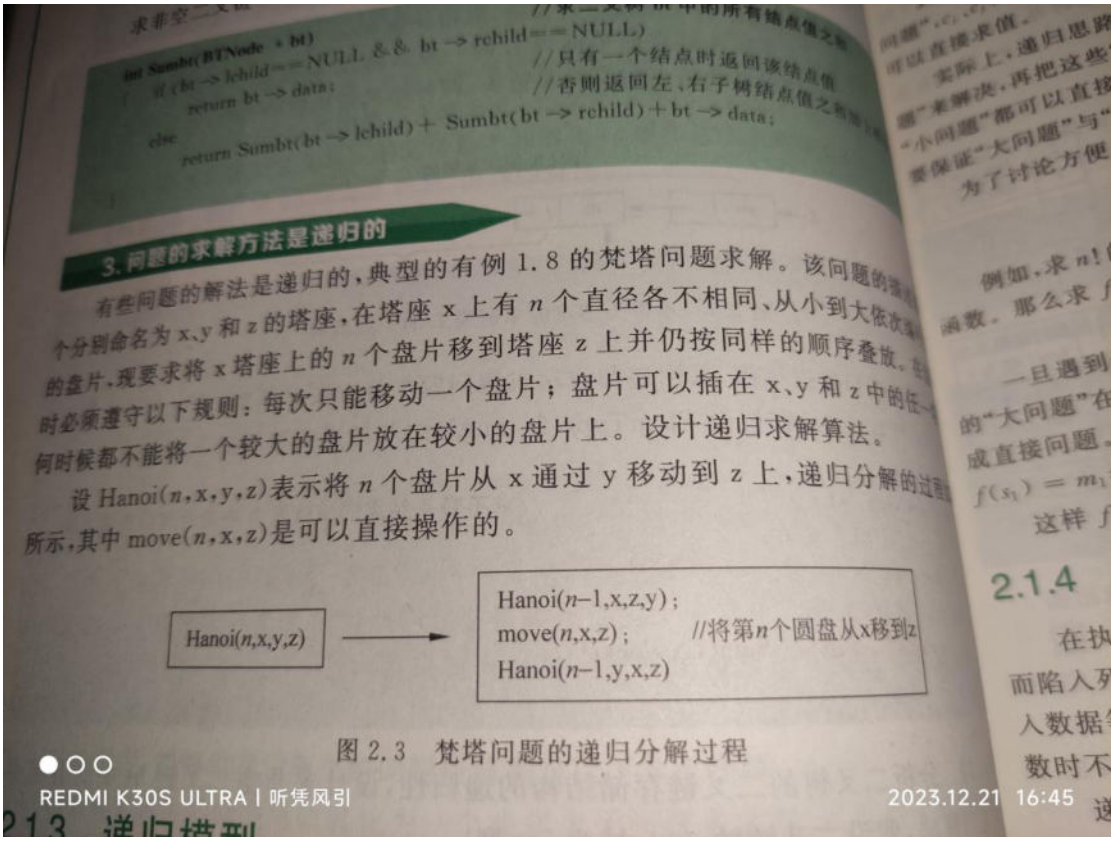
背包问题算法全解析：动态规划和贪心算法详解 - 知乎：这篇文章比较了动态规划和贪心算法在解决背包问题时的优缺点和应用场景。

三、 算法代码填空

1. 汉诺塔问题

汉诺塔问题是一个经典的递归问题，它的思路是将 n 个盘子从一个柱子借助另一个柱子移动到第三个柱子，其中要保证小盘子始终在大盘子上方。

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

//定义一个全局变量，记录移动的次数
int count = 0;

//定义一个函数，实现将一个盘子从一个柱子移动到另一个柱子
void move(char a, int n, char c) {
    count++; //每移动一次，次数加一
    cout << a << "->" << c << endl; //输出移动的过程
}

//定义一个函数，实现将 n 个盘子从一个柱子借助另一个柱子移动到第三个柱子
void hanoi(int n, char a, char b, char c) {
    if (n == 1) { //当只有一个盘子时，直接移动
        move(a, 1, c);
    }
    else { //当有多个盘子时，分三步进行
        hanoi(n - 1, a, c, b); //第一步，将 n-1 个盘子从 a 柱借助 c 柱移动到 b
柱
        move(a, n, c); //第二步，将第 n 个盘子从 a 柱移动到 c 柱
        hanoi(n - 1, b, a, c); //第三步，将 n-1 个盘子从 b 柱借助 a 柱移动到 c
柱
    }
}

int main() {
    int n; //定义一个变量，表示盘子的个数
    cin >> n; //输入盘子的个数
    hanoi(n, 'a', 'b', 'c'); //调用 hanoi 函数，实现汉诺塔问题
    cout << count << endl; //输出移动的总次数
    return 0;
    }
```



C++汉诺塔问题 解题思路及递归算法实现：这篇文章介绍了汉诺塔问题的起源和问题描述，以及用 c++实现的递归算法和运行结果。

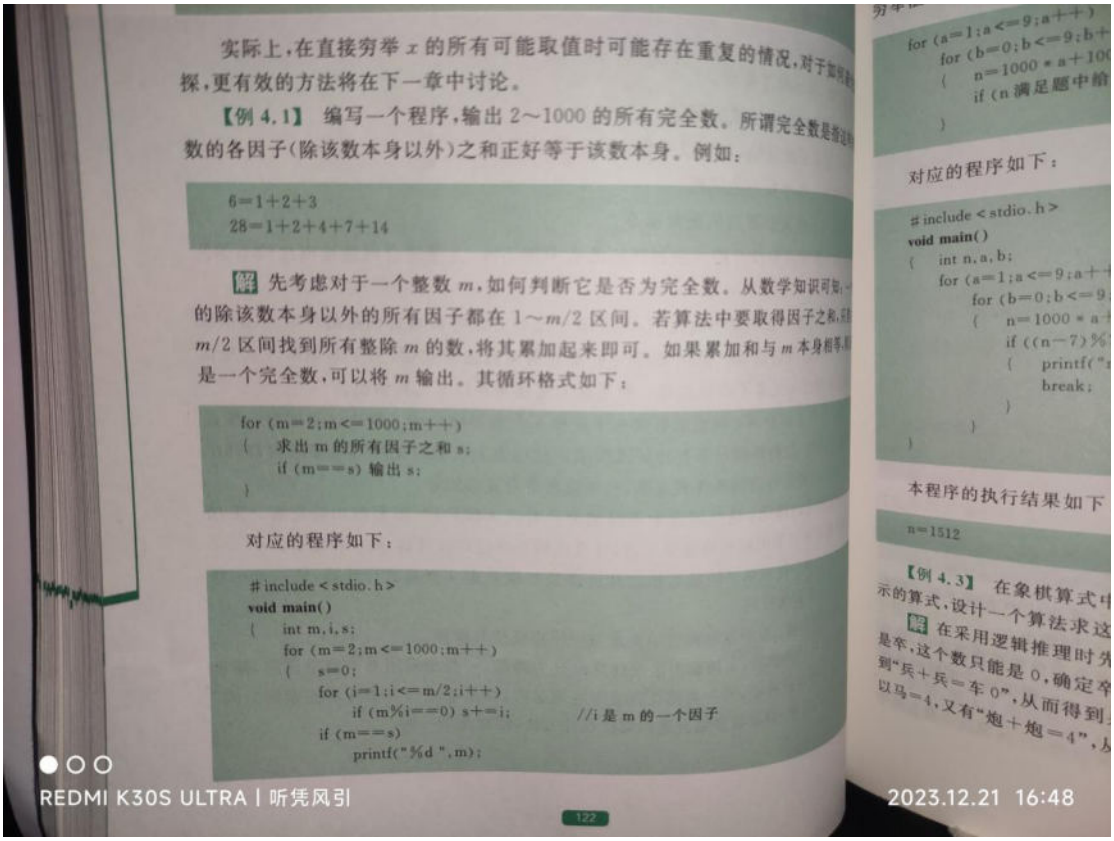
利用递归巧解汉诺塔问题（c++）：这篇文章用动画的方式讲解了汉诺塔问题的解决思路，以及用 c++实现的递归算法和效果图。

C++解决汉诺塔问题（递归实现）：这篇文章给出了用 c++实现的递归算法和完整代码，以及一些注意事项和优化建议。

汉诺塔问题（分治+源码+动画演示）：这篇文章讲解了汉诺塔问题的分治算法的原理和步骤，以及用 c/c++、java、

python 实现的源码和动画演示。

2.蛮力法基本应用



实际上，在直接穷举 x 的所有可能取值时可能存在重复的情况，对于此探，更有效的方法将在下一章中讨论。

【例 4.1】 编写一个程序，输出 2～1000 的所有完全数。所谓完全数是指这数的各因子（除该数本身以外）之和正好等于该数本身。例如：

```
6=1+2+3
28=1+2+4+7+14
```

解 先考虑对于一个整数 m，如何判断它是否为完全数。从数学知识可知，的除该数本身以外的所有因子都在 1～$m/2$ 区间。若算法中要取得因子之和，$m/2$ 区间找到所有整除 m 的数，将其累加起来即可。如果累加和与 m 本身相等，是一个完全数，可以将 m 输出。其循环格式如下：

```
for (m=2;m<=1000;m++)
{   求出 m 的所有因子之和 s;
    if (m==s) 输出 s;
}
```

对应的程序如下：

```
#include <stdio.h>
void main()
{   int m,i,s;
    for (m=2;m<=1000;m++)
    {   s=0;
        for (i=1;i<=m/2;i++)
            if (m%i==0) s+=i;          //i 是 m 的一个因子
        if (m==s)
            printf("%d ",m);
```

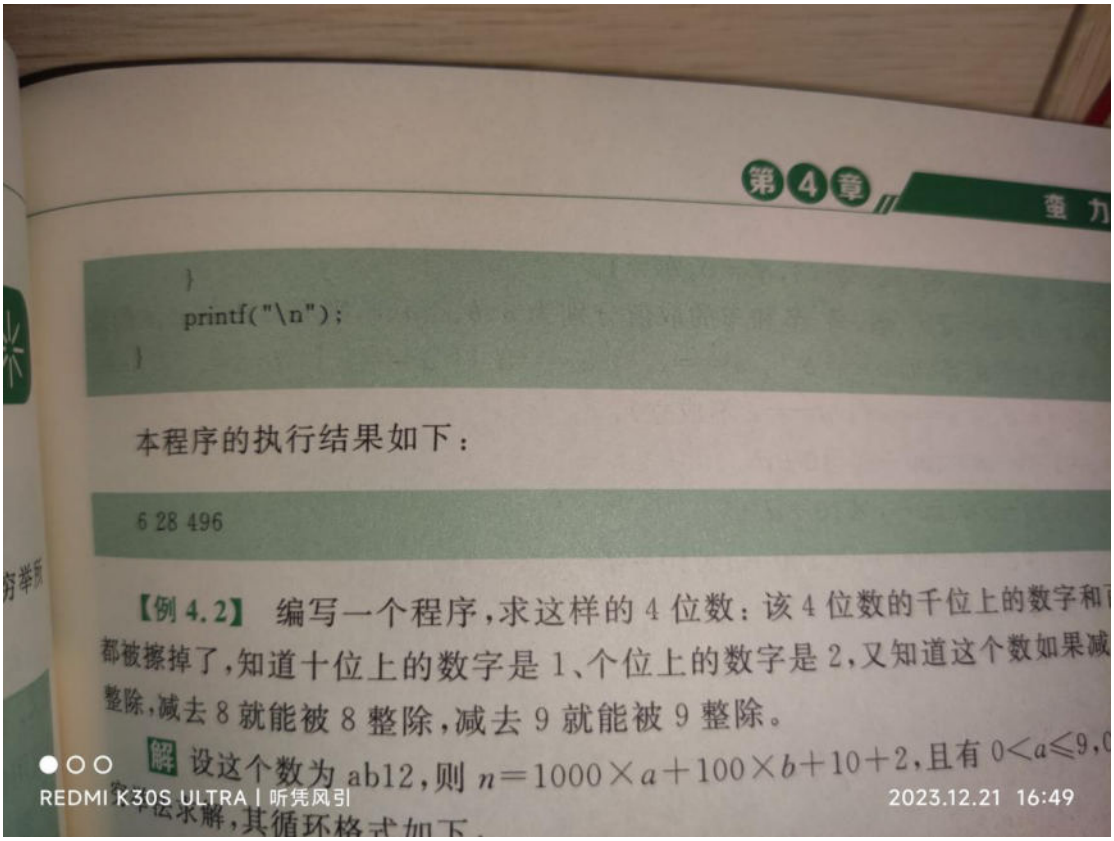
对应的程序如下：

```
#include <stdio.h>
void main()
{   int n,a,b;
    for (a=1;a<=9;a++)
        for (b=0;b<=9;b
        {   n=1000*a+
            if ((n-7)%
            {   printf("
                break;
            }
        }
}
```

本程序的执行结果如下

```
n=1512
```

【例 4.3】 在象棋算式中
示的算式，设计一个算法求这
解 在采用逻辑推理时先
是卒，这个数只能是 0，确定卒
到“兵+兵=车 0”，从而得到
以马=4，又有“炮+炮=4”，从

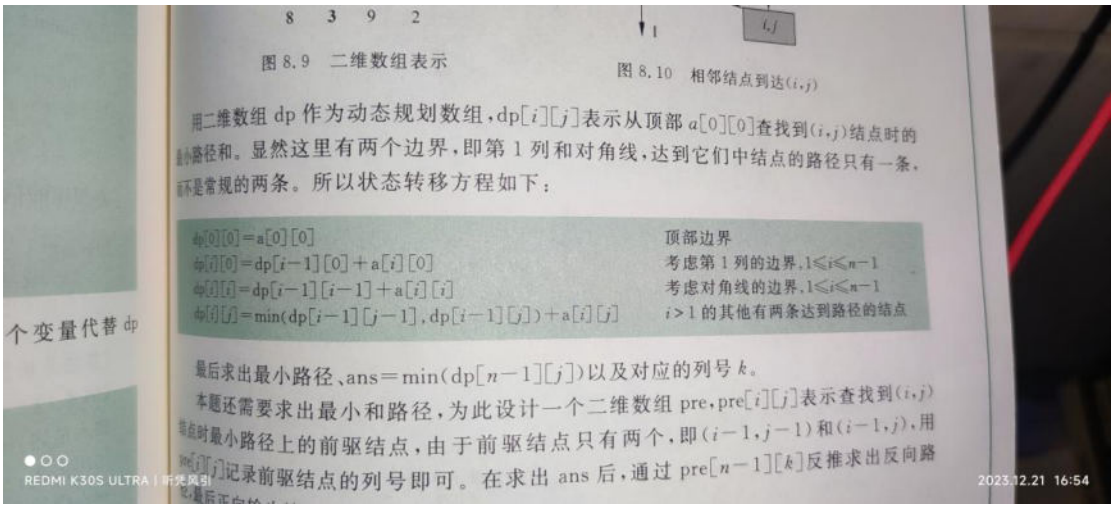


3.求解三角形最小路径问题

三角形最小路径问题是一个动态规划问题，它的思路是从三角形的底部向上递推，每个位置的最小路径和等于它下面两个相邻位置的最小路径和中较小的一个加上它自己的值

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

//定义一个函数，求解三角形最小路径和
int minPathSum(vector<vector<int>>& triangle) {
    int n = triangle.size(); //获取三角形的行数
    vector<int> dp(n); //定义一个一维数组，存储每一行的最小路径和
    //从三角形的最后一行开始，初始化 dp 数组
    for (int i = 0; i < n; i++) {
        dp[i] = triangle[n - 1][i];
```

```
    }
    //从倒数第二行开始，向上递推，更新 dp 数组
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            //每个位置的最小路径和等于它下面两个相邻位置的最小路径和中较小的一个加上它自己的值
            dp[j] = min(dp[j], dp[j + 1]) + triangle[i][j];
        }
    }
    //返回三角形顶部的最小路径和
    return dp[0];
}

int main() {
    //定义一个三角形，可以根据需要修改
    vector<vector<int>> triangle = {
        {2},
        {3, 4},
        {6, 5, 7},
        {4, 1, 8, 3}
    };
    //调用函数，求解三角形最小路径和
    int ans = minPathSum(triangle);
    //输出结果
    cout << "The minimum path sum is: " << ans << endl;
    return 0;
    }
```



8 3 9 2

图 8.9 二维数组表示

图 8.10 相邻结点到达(i,j)

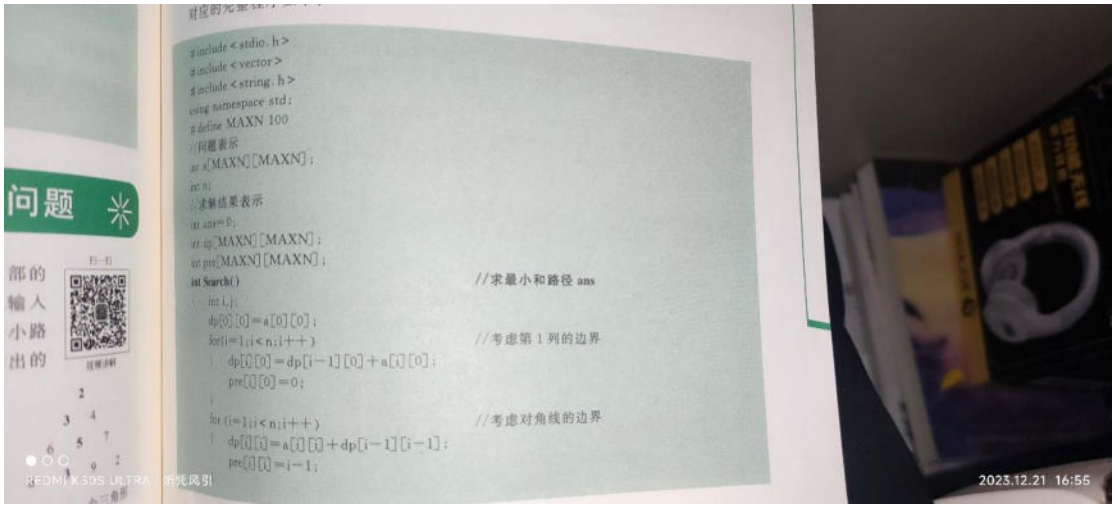
用二维数组 dp 作为动态规划数组，$dp[i][j]$表示从顶部 $a[0][0]$查找到(i,j)结点时的最小路径和。显然这里有两个边界，即第 1 列和对角线，达到它们中结点的路径只有一条，而不是常规的两条。所以状态转移方程如下：

$dp[0][0]=a[0][0]$	顶部边界
$dp[i][0]=dp[i-1][0]+a[i][0]$	考虑第 1 列的边界，$1\leqslant i\leqslant n-1$
$dp[i][i]=dp[i-1][i-1]+a[i][i]$	考虑对角线的边界，$1\leqslant i\leqslant n-1$
$dp[i][j]=\min(dp[i-1][j-1],dp[i-1][j])+a[i][j]$	$i>1$ 的其他有两条达到路径的结点

最后求出最小路径，$ans=\min(dp[n-1][j])$以及对应的列号 k。

本题还需要求出最小和路径，为此设计一个二维数组 pre，$pre[i][j]$表示查找到(i,j)结点时最小路径上的前驱结点，由于前驱结点只有两个，即$(i-1,j-1)$和$(i-1,j)$，用$pre[i][j]$记录前驱结点的列号即可。在求出 ans 后，通过 $pre[n-1][k]$反推求出反向路

个变量代替 dp


问题
部的
输入
小路
出的
#include <stdio.h>
#include <vector>
#include <string.h>
using namespace std;
#define MAXN 100
//问题表示
int a[MAXN][MAXN];
int n;
//求解结果表示
int ans=0;
int dp[MAXN][MAXN];
int pre[MAXN][MAXN];
int Search()
//求最小和路径 ans
int i,j;
dp[0][0]=a[0][0];
for(i=1;i<n;i++)
//考虑第 1 列的边界
dp[i][0]=dp[i-1][0]+a[i][0];
pre[i][0]=0;
for(i=1;i<n;i++)
//考虑对角线的边界
dp[i][i]=a[i][i]+dp[i-1][i-1];
pre[i][i]=i-1;
REDMI K30S ULTRA
2023.12.21 16:55

```
    for(i=2;i<n;i++)                                    //考虑其他有两条达到路径的结点
    {   for(j=1;j<i;j++)
        {   if(dp[i-1][j-1]<dp[i-1][j])
            {   pre[i][j]=j-1;
                dp[i][j]=a[i][j]+dp[i-1][j-1];
            }
            else
            {   pre[i][j]=j;
                dp[i][j]=a[i][j]+dp[i-1][j];
            }
        }
    }
    ans=dp[n-1][0];
    int k=0;
    for (j=1;j<n;j++)                                   //求出最小 ans 和对应的列号 k
    {   if (ans>dp[n-1][j])
        {   ans=dp[n-1][j];
            k=j;
        }
    }
    return k;
}
void Disppath(int k)                                    //输出最小和路径
{   int i=n-1;
    vector<int> path;                                   //存放逆路径向量 path
    while (i>=0)                                        //从(n-1,k)结点反推求出反向路径
    {   path.push_back(a[i][k]);
        k=pre[i][k];                                    //最小路径在前一行中的列号
        i--;                                            //在前一行中查找
    }
    vector<int>::reverse_iterator it;                   //定义反向迭代器
    for (it=path.rbegin();it!=path.rend();++it)
        printf("%d ",*it);                              //反向输出构成正向路径
    printf("\n");
}
int main()
{   int k;
    memset(pre,0,sizeof(pre));
    memset(dp,0,sizeof(dp));
    scanf("%d",&n);                                     //输入三角形的高度
    for (int i=0;i<n;i++)                               //输入三角形
        for (int j=0;j<=i;j++)
            scanf("%d",&a[i][j]);
    k=Search();                                         //求最小路径和
    Disppath(k);                                        //输出正向路径
    printf("%d\n",ans);                                 //输出最小路径和
    return 0;
}
```

三角形最小路径和(动态规划)：这篇文章详细地介绍了三角形最小路径和的动态规划算法的实现和优化，以及给出了两种版本的 c++代码。

120. 三角形最小路径和(C++)—动态规划解题：这篇文章用一个具体的例子来解释三角形最小路径和的动态规划算法的原理和步骤，以及给出了自顶向下和自底向上两种版本的 c++代码。

【动态规划/路径问题】进阶「最小路径和」问题：这篇文章讲解了三角形最小路径和问题的数学模型和状态转移方程，以及给出了 java 代码和运行结果。

【Leetcode 每日打卡】三角形最小路径和：这篇文章总结了三角形最小路径和问题的动态规划算法的思路和技巧，以及给出了 java 代码和注释。

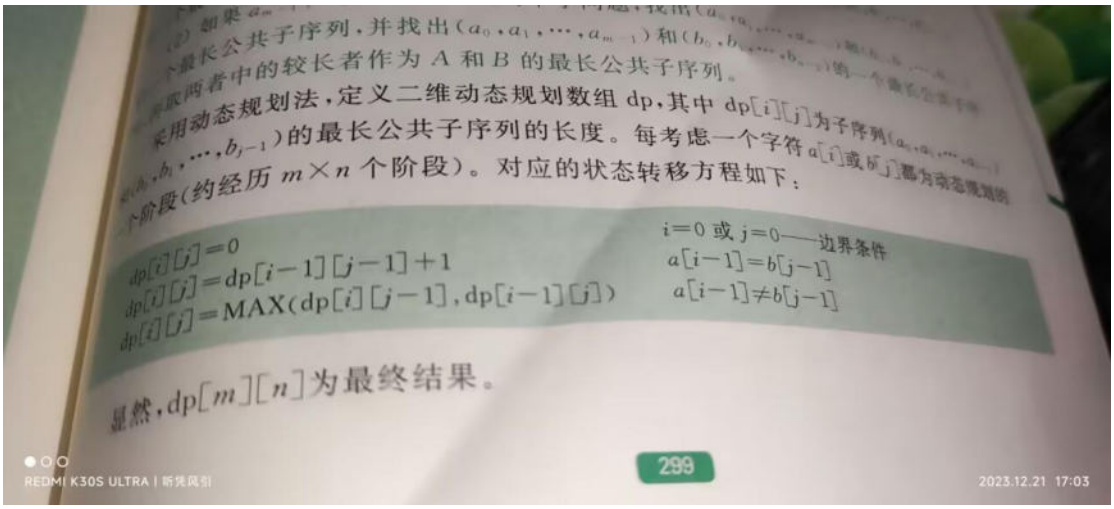
4.求解最长公共子序列问题

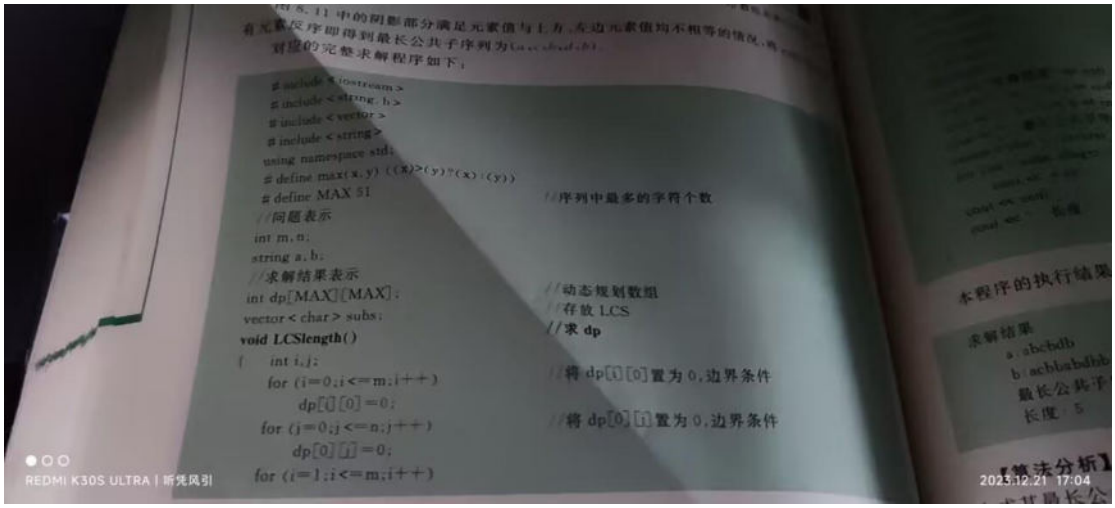
最长公共子序列问题是一个动态规划问题，它的思路是用一个二维数组记录两个字符串的最长公共子序列的长度，然后根据数组的值回溯找出最长公共子序列的内容。

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

```cpp
//定义一个函数，求解两个字符串的最长公共子序列的长度和内容
string lcs(string str1, string str2) {
    int m = str1.size(); //获取第一个字符串的长度
    int n = str2.size(); //获取第二个字符串的长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0)); //定义一个二维数组，存储最长公共子序列的长度
    //从左上角开始，填充二维数组
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (str1[i - 1] == str2[j - 1]) { //如果两个字符相等，则当前位置的值等于左上角的值加一
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            else { //如果两个字符不等，则当前位置的值等于上方或左方的值中较大的一个
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    //从右下角开始，回溯找出最长公共子序列的内容
    string res = ""; //定义一个字符串，存储最长公共子序列的内容
    int i = m, j = n; //定义两个变量，表示当前位置的坐标
    while (i > 0 && j > 0) { //当坐标没有越界时，循环执行
        if (str1[i - 1] == str2[j - 1]) { //如果两个字符相等，则将该字符加入到结果字符串的前面，并将坐标向左上方移动一格
            res = str1[i - 1] + res;
            i--;
            j--;
        }
        else { //如果两个字符不等，则将坐标向上方或左方移动一格，取决于哪个方向的值较大
            if (dp[i - 1][j] > dp[i][j - 1]) {
                i--;
            }
            else {
                j--;
            }
        }
    }
    return res; //返回结果字符串
}

int main() {
    string str1, str2; //定义两个字符串，存储用户的输入
```

```
    cout << "Please enter the first string: ";
    cin >> str1; //输入第一个字符串
    cout << "Please enter the second string: ";
    cin >> str2; //输入第二个字符串
    string ans = lcs(str1, str2); //调用 lcs 函数，求解最长公共子序列的内容
    cout << "The longest common subsequence is: " << ans << endl; //输出
结果
    return 0;
}
```





```
    for (j=1;j<=n;j++)                          //两重 for 循环处理 a,b 的所有字符
        if (a[i-1]==b[j-1])                     //情况(1)
            dp[i][j]=dp[i-1][j-1]+1;
        else                                    //情况(2)
            dp[i][j]=max(dp[i][j-1],dp[i-1][j]);
    }
}
void Buildsubs()                                //由 dp 构造 subs
{   int k=dp[m][n];                             //k 为 a 和 b 的最长公共子序列的长度
    int i=m;
    int j=n;
    while (k>0)                                 //在 subs 中放入最长公共子序列(反向)
    {   if (dp[i][j]==dp[i-1][j])
            i--;
        else if (dp[i][j]==dp[i][j-1])
            j--;
        else                                    //与上方、左边元素的值均不相等
        {   subs.push_back(a[i-1]);             //在 subs 中添加 a[i-1]
            i--;j--;k--;
        }
    }
}
void main()
{   a="abcbdb";
    b="acbbabdbb";
    m=a.length();                               //m 为 a 的长度
    n=b.length();                               //n 为 b 的长度
    LCSlength();                                //求出 dp
    Buildsubs();                                //求出 LCS
    cout<<"求解结果"<<endl;
    cout<<"  a:"<<a<<endl;
    cout<<"  b:"<<b<<endl;
    cout<<"  最长公共子序列:";
    vector<char>::reverse_iterator rit;
    for (rit=subs.rbegin();rit!=subs.rend();++rit)
        cout<<*rit;
    cout<<endl;
    cout<<"  长度:"<<dp[m][n]<<endl;
}
```

本程序的执行结果如下：

```
求解结果
  a:abcbdb
  b:acbbabdbb
  最长公共子序列:acbdb
  长度:5
```

【算法分析】 在 LCSlength 算法中使用了两重循环，所以对于长度分别为 m 和 n 的两个字符串 a 和 b，求其最长公共子序列的时间复杂度为 $O(mn)$、空间复杂度为 $O(mn)$。

最长公共子序列（LCS）的C++实现：这篇文章详细地介绍了最长公共子序列的动态规划算法的实现和优化，以及给出了两种版本的c++代码。

最长公共子序列问题 C++ （超详细）：这篇文章用一个具体的例子来解释最长公共子序列的动态规划算法的原理和步骤，以及给出了自顶向下和自底向上两种版本的c++代码。

c++解决最长公共子序列问题：这篇文章总结了最长公共子序列问题的动态规划算法的思路和技巧，以及给出了c++代码和注释。

[动态规划]最长公共子序列：这篇文章讲解了最长公共子序列问题的数学模型和状态转移方程，以及给出了c++代码和运行结果。

四、 算法设计与分析

1. 水仙花数问题

水仙花数是指一个三位数，它的每个位上的数字的立方和等于它本身，例如153是水仙花数，因为：153 = 1^3 + 5^3 + 3^3。要求解水仙花数问题，可以用

以下的思路：

遍历所有的三位数，从 100 到 999；

对每个三位数，分别取出它的个位、十位和百位数字；

计算这三个数字的立方和，判断是否等于原数；

如果相等，就输出该数，否则继续遍历。

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

int main() {
  // 定义三个变量，分别存储个位、十位和百位数字
  int a, b, c;
  // 用一个循环遍历所有的三位数
  for (int i = 100; i < 1000; i++) {
    // 用取余和除法运算分别得到个位、十位和百位数字
    a = i % 10;
    b = i / 10 % 10;
    c = i / 100;
    // 计算三个数字的立方和
    int sum = a * a * a + b * b * b + c * c * c;
    // 判断是否等于原数
    if (sum == i) {
      // 如果相等，就输出该数
      cout << i << endl;
    }
  }
  return 0;
}
```

水仙花数的总结概括以及各种代码实现（最详细）

C 语言水仙花数（详解版）

C++练习：打印水仙花数

C++ 水仙花数：多种实现方法对比

C++ 求水仙花数

---

2. 斐波那契数列问题

斐波那契数列是一个由 0 和 1 开始，后面每一项都是前两项之和的数列，例如：0, 1, 1, 2, 3, 5, 8, 13, 21, …。要求解斐波那契数列问题，可以用以下的思路：

递归：直接根据斐波那契数列的定义，用递归函数来求解第 n 项，但是这种方法效率很低，时间复杂度是指数级的，而且容易造成栈溢出。

迭代：用两个变量来保存前两项的值，然后用循环来求解第 n 项，这种方法效率较高，时间复杂度是线性的，空间复杂度是常数的。

数组：用一个数组来保存前 n 项的值，然后用循环来求解第 n 项，这种方法适合求解斐波那契数列的前 n 项，时间复杂度是线性的，空间复杂度是线性的。

通项公式：利用斐波那契数列的通项公式，直接计算第 n 项的值，这种方法效率最高，时间复杂度是常数的，但是需要用到浮点数的运算，可能会有精度的误差。

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

// 递归实现
int fib1(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib1(n - 1) + fib1(n - 2);
}
```

```cpp
// 迭代实现
int fib2(int n) {
  int x = 0, y = 1;
  for (int i = 0; i < n; i++) {
    y = x + y;
    x = y - x;
  }
  return x;
}

// 数组实现
int fib3(int n) {
  int* fib = new int[n + 1];
  fib[0] = 0;
  fib[1] = 1;
  for (int i = 2; i <= n; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }
  int result = fib[n];
  delete[] fib;
  return result;
}

// 通项公式实现
int fib4(int n) {
  const double sqrt5 = sqrt(5);
  double result = 1 / sqrt5 * (pow((1 + sqrt5) / 2, n) - pow((1 - sqrt5)
/ 2, n));
  return (int)result;
}

int main() {
  int n;
  cout << "请输入一个整数：" << endl;
  cin >> n;
  cout << "递归实现：" << fib1(n) << endl;
  cout << "迭代实现：" << fib2(n) << endl;
  cout << "数组实现：" << fib3(n) << endl;
  cout << "通项公式实现：" << fib4(n) << endl;
  return 0;
}
```

斐波那契数列（递归及非递归）C/C++实现

斐波那契数列（Fibonacci sequence）【思路及实现】

斐波那契数列的四种解法（头递归、尾递归、迭代与通项公式）

C++用递归实现斐波那契数列

3. 查找最大和次大元素

查找最大和次大元素的问题可以用以下的思路：

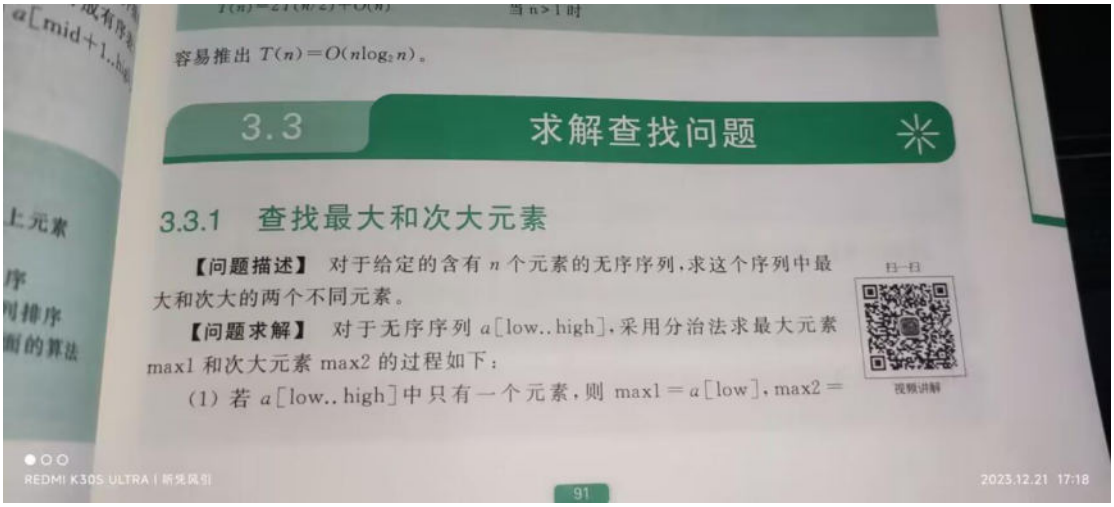
遍历数组，用两个变量分别记录最大和次大的元素，初始值设为最小的整数；

对每个元素，如果它大于最大的元素，就更新最大和次大的元素；如果它大于次大的元素，就更新次大的元素；

遍历结束后，返回最大和次大的元素。

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

// 查找最大和次大元素
void find_max_and_second_max(int a[], int n, int &max1, int &max2) {
  // 初始化最大和次大元素为最小的整数
  max1 = max2 = INT_MIN;
  // 遍历数组
  for (int i = 0; i < n; i++) {
    // 如果当前元素大于最大元素，更新最大和次大元素
    if (a[i] > max1) {
      max2 = max1;
      max1 = a[i];
    }
    // 如果当前元素大于次大元素，更新次大元素
    else if (a[i] > max2) {
      max2 = a[i];
```

```
    }
  }
}

int main() {
  // 定义一个数组
  int a[] = {3, 5, 2, 7, 4, 9, 6};
  // 数组的长度
  int n = sizeof(a) / sizeof(a[0]);
  // 定义两个变量，用来存储最大和次大元素
  int max1, max2;
  // 调用函数，查找最大和次大元素
  find_max_and_second_max(a, n, max1, max2);
  // 输出结果
  cout << "最大元素是：" << max1 << endl;
  cout << "次大元素是：" << max2 << endl;
  return 0;
}
```



$-INF(-\infty)$。

(2) 若 $a[low..high]$中只有两个元素，则 $max1=max\{a[low],a[high]\}$，$max2=min\{a[low],a[high]\}$。

(3) 若 $a[low..high]$中有两个以上元素，按中间位置 $mid=(low+high)/2$ 划分为 $a[low..mid]$和 $a[mid+1..high]$两个区间(注意左区间包含 $a[mid]$元素)。求出左区间的最大元素 lmax1 和次大元素 lmax2，求出右区间的最大元素 rmax1 和次大元素 rmax2。

若 lmax1>rmax1，则 max1=lmax1，max2=max{lmax2，rmax1}；否则 max1=rmax1，max2=max{lmax1，rmax2}。

例如，对于 $a[0..4]=\{5,2,1,4,3\}$，$mid=(0+4)/2=2$，划分为左区间 $a[0..2]=\{5,2,1\}$，右区间 $a[3..4]=\{4,3\}$。在左区间中求出 lmax1=5，lmax2=2，在右区间中求出 rmax1=4，rmax2=3。所以 max1=max{lmax1，rmax1}=5，max2=max{lmax2，rmax1}=4。

对应的算法如下：

```
void solve(int a[],int low,int high,int &max1,int &max2)
{   if (low==high)                          //区间中只有一个元素
    {   max1=a[low];
        max2=-INF;
    }
    else if (low==high-1)                   //区间中只有两个元素
    {   max1=max(a[low],a[high]);
        max2=min(a[low],a[high]);
    }
    else                                    //区间中有两个以上元素
    {   int mid=(low+high)/2;
        int lmax1,lmax2;
        solve(a,low,mid,lmax1,lmax2);       //左区间求 lmax1 和 lmax2
        int rmax1,rmax2;
        solve(a,mid+1,high,rmax1,rmax2);    //右区间求 rmax1 和 rmax2
        if (lmax1>rmax1)
        {   max1=lmax1;
            max2=max(lmax2,rmax1);          //lmax2,rmax1 中求次大元素
        }
        else
        {   max1=rmax1;
            max2=max(lmax1,rmax2);          //lmax1,rmax2 中求次大元素
        }
    }
}
```

【算法分析】 对于 solve(a,0,$n-1$,max1,max2)调用，其比较次数的递推式如下：

$T(1)=T(2)=1$

$T(n)=2T(n/2)+1$ //合并的时间为 $O(1)$

可以推导出 $T(n)=O(n)$。

算法 || 分治法【查找最大元素和次大元素)】 #01

找出数组中最大值次大值的一次遍历方法(C++)

分治法求 n 个数中第二个最大元素 C 语言

经典算法详解（11）递归查找数组中的最大值

分治法查找最大和次大元素

---

4. 折半查找

折半查找，也叫二分查找，是一种在有序数组中查找某一特定元素的搜索算法。要求解折半查找问题，可以用以下的思路：

定义两个变量，分别表示数组的最小下标和最大下标，初始值分别为 0 和数组长度减 1；

计算数组的中间下标，即最小下标和最大下标的平均值，向下取整；

比较中间下标对应的元素和目标元素，如果相等，就返回中间下标；

如果目标元素小于中间下标对应的元素，就把最大下标更新为中间下标减 1，表示缩小搜索范围到数组的前半部分；

如果目标元素大于中间下标对应的元素，就把最小下标更新为中间下标加 1，表示缩小搜索范围到数组的后半部分；

重复上述步骤，直到找到目标元素或者最小下标大于最大下标，表示数组中

不存在目标元素。

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

// 折半查找函数，参数为数组，数组长度，目标元素，返回值为目标元素的下标，如果不存在，返回-1
int binary_search(int arr[], int len, int target) {
  // 定义最小下标和最大下标
  int low = 0;
  int high = len - 1;
  // 定义中间下标
  int mid;
  // 循环查找，直到最小下标大于最大下标
  while (low <= high) {
    // 计算中间下标
    mid = (low + high) / 2;
    // 比较中间下标对应的元素和目标元素
    if (target == arr[mid]) {
      // 如果相等，返回中间下标
      return mid;
    } else if (target < arr[mid]) {
      // 如果目标元素小于中间下标对应的元素，更新最大下标为中间下标减 1
      high = mid - 1;
    } else {
      // 如果目标元素大于中间下标对应的元素，更新最小下标为中间下标加 1
      low = mid + 1;
    }
  }
  // 如果循环结束，还没有找到目标元素，返回-1
  return -1;
}

int main() {
  // 定义一个有序数组
  int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
  // 数组的长度
  int len = sizeof(arr) / sizeof(arr[0]);
  // 定义一个目标元素
  int target = 13;
  // 调用折半查找函数，得到目标元素的下标
  int index = binary_search(arr, len, target);
  // 输出结果
  if (index == -1) {
    cout << "数组中不存在" << target << endl;
```

```
  } else {
    cout << target << "在数组中的下标是：" << index << endl;
  }
  return 0;
}
```

3.3.2 折半查找

扫一扫

视频讲解

折半查找又称二分查找，它是一种效率较高的查找方法。但是折半查找要求查找序列中的元素是有序的，为了简单，假设是递增有序的。

折半查找的基本思路：设 $a[low..high]$ 是当前的查找区间，首先确定该区间的中点位置 $mid=\lfloor(low+high)/2\rfloor$；然后将待查的 k 值与 $a[mid].key$ 比较。

(1) 若 $k==a[mid].key$，则查找成功并返回该元素的物理下标。

(2) 若 $k<a[mid]$，则由表的有序性可知 $a[mid..high]$ 均大于 k，因此若表中存在关键字等于 k 的元素，则该元素必定位于左子表 $a[low..mid-1]$ 中，故新的查找区间是左子表 $a[low..mid-1]$。

(3) 若 $k>a[mid]$，则要查找的 k 必定位于右子表 $a[mid+1..high]$ 中，即新的查找区间是右子表 $a[mid+1..high]$。

下一次查找是针对新的查找区间进行的。

因此可以从初始的查找区间 $a[0..n-1]$ 开始，每经过一次与当前查找区间的中点位置上的关键字比较就可确定查找是否成功，不成功则当前的查找区间缩小一半。重复这一过程，直到找到关键字为 k 的元素，或者直到当前的查找区间为空(即查找失败)时为止。

折半查找对应的完整程序如下：

```
#include <stdio.h>
int BinSearch(int a[],int low,int high,int k)        //折半查找算法
{   int mid;
    if (low<=high)                                   //当前区间存在元素时
    {   mid=(low+high)/2;                            //求查找区间的中间位置
        if (a[mid]==k)                               //找到后返回其物理下标 mid
            return mid;
        if (a[mid]>k)                                //当 a[mid]>k 时在 a[low..mid-1]中递归查找
            return BinSearch(a,low,mid-1,k);
        else                                         //当 a[mid]<k 时在 a[mid+1..high]中递归查找
            return BinSearch(a,mid+1,high,k);
    }
    else return -1;                                  //当前查找区间没有元素时返回-1
}
void main()
{   int n=10,i;
    int k=6;
    int a[]={1,2,3,4,5,6,7,8,9,10};
    i=BinSearch(a,0,n-1,k);
    if (i>=0)printf("a[%d]=%d\n",i,k);
    else printf("未找到%d 元素\n",k);
}
```

可以将折半查找递归算法等价地转换成以下非递归算法：

```
int BinSearch1(int a[],int n,int k)                  //非递归折半查找算法
{   int low=0,high=n-1,mid;
```

```
    while (low<=high)                    //当前区间存在元素时循环
    {   mid=(low+high)/2;                //求查找区间的中间位置
        if (a[mid]==k)                   //找到后返回其物理下标 mid
            return mid;
        if (a[mid]>k)                    //继续在 a[low..mid-1]中查找
            high=mid-1;
        else                             //a[mid]<k
            low=mid+1;                   //继续在 a[mid+1..high]中查找
    }
    return -1;                           //当前查找区间没有元素时返回-1
}
```

【算法分析】 折半查找算法的主要时间花费在元素的比较上，对于含有 n 个元素的有序表，采用折半查找时最坏情况下的元素比较次数为 $C(n)$，则有：

$$\begin{cases} C(n)=1 & \text{当 } n=1 \text{ 时} \\ C(n)\leqslant 1+C(\lfloor n/2 \rfloor) & \text{当 } n\geqslant 2 \text{ 时} \end{cases}$$

设对某个整数 $k\geqslant 2$，满足 $2^{k-1}\leqslant n<2^k$。展开上述递推式，可得到：

$$\begin{aligned} C(n) &\leqslant 1+C(\lfloor n/2 \rfloor) \\ &\leqslant 2+C(\lfloor n/4 \rfloor) \\ &\cdots \\ &\leqslant (k-1)+C(\lfloor n/2^{k-1} \rfloor) \\ &=(k-1)+1 \\ &=k \end{aligned}$$

而 $2^{k-1}\leqslant n<2^k$，即 $k\leqslant \log_2 n+1<k+1$，$k=\lfloor \log_2 n \rfloor+1$。

由此得到 $C(n)\leqslant \lfloor \log_2 n \rfloor+1$。

也就是说，在含有 n 个元素的有序序列中采用折半查找算法查找指定的元素所需的元素比较次数不超过 $\lfloor \log_2 n \rfloor+1$（或者 $\lceil \log_2(n+1) \rceil$）。实际上，$n$ 个元素的折半查找对应判定树的高度恰好是 $\lfloor \log_2 n \rfloor+1$。折半查找的主要时间花在元素的比较上，所以算法的时间复杂度为 $O(\log_2 n)$。

折半查找的思路很容易推广到三分查找，显然三分查找对应判断树的高度恰好是 $\lfloor \log_3 n \rfloor+1$，推出查找时间复杂度为 $O(\log_3 n)$，由于 $\log_3 n=\log_2 n/\log_2 3$，所以三分查找和二分查找的时间是同一个数量级的。

【例 3.1】 求解假币问题。有 100 个硬币，其中有一个假币（与真币一模一样，只是比真币的重量轻），采用天平称重方法找出这个假币，最少用天平称重多少次保证找出假币。

解 已知假币比真币的重量轻，可以将 100 个硬币分为两组，每组 50 个硬币，称重一次可以确定假币所在的组，即二分法。更好的方法是采用三分法，将 100 个硬币分为 33、33、34 三组，用天平一次称重可以找出假币所在的组，依次进行，对应一棵三分判定树，树高度恰好是称重次数，结果为 $\lceil \log_3(100+1) \rceil=5$。

3.3.3 寻找一个序列中第 k 小的元素

【问题描述】 对于给定的含有 n 个元素的无序序列，求这个序列中第 $k(1\leqslant k\leqslant n)$ 小的元素。

C++折半查找的实现

折半查找 (C++实现)

二分查找（折半查找）实例讲解

C 语言简单实现折半查找法

---

5. 求解 0/1 背包问题

0/1 背包问题是一种经典的组合优化问题，它的目标是在给定一组物品的重量和价值以及一个背包的容量的情况下，选择一些物品放入背包，使得背包中物品的总价值最大，同时不超过背包的容量。要求解 0/1 背包问题，可以用以下的思路：

动态规划：用一个二维数组 f [i] [j]表示前 i 个物品放入容量为 j 的背包的最大价值，根据第 i 个物品是否放入背包，分为两种情况，取其中的较大值作为 f [i] [j]的值，最后返回 f [n] [m]作为最优解。这种方法的时间复杂度和空间复杂度都是 O (n*m)。

贪心算法：按照物品的单位价值（价值/重量）从高到低排序，依次选择单位价值最高的物品放入背包，直到背包容量不足。这种方法的时间复杂度是 O (n*logn)，空间复杂度是 O (n)，但是不一定能得到最优解。

```cpp
// 动态规划实现
#include <iostream>
using namespace std;

// 定义物品的结构体，包含重量和价值
struct Item {
  int weight;
  int value;
};

// 定义一个函数，返回两个数中的较大值
int max(int a, int b) {
  return a > b ? a : b;
}

// 定义一个函数，求解 0/1 背包问题的最大价值
// 参数为物品数组，物品个数，背包容量
// 返回值为最大价值
int knapsack(Item items[], int n, int m) {
  // 定义一个二维数组，表示前 i 个物品放入容量为 j 的背包的最大价值
  int f[n + 1][m + 1];
  // 初始化边界条件，当 i=0 或 j=0 时，f [i] [j]=0
  for (int i = 0; i <= n; i++) {
    f[i][0] = 0;
  }
  for (int j = 0; j <= m; j++) {
    f[0][j] = 0;
  }
  // 逐行填充数组，从第一行开始
  for (int i = 1; i <= n; i++) {
    // 逐列填充数组，从第一列开始
    for (int j = 1; j <= m; j++) {
      // 如果第 i 个物品的重量大于背包的容量，那么不能放入，f [i] [j]等于 f
[i-1] [j]
      if (items[i - 1].weight > j) {
        f[i][j] = f[i - 1][j];
      } else {
        // 如果第 i 个物品的重量小于等于背包的容量，那么有两种选择，放入或不放
入
        // 如果不放入，f [i] [j]等于 f [i-1] [j]
        // 如果放入，f [i] [j]等于 f [i-1] [j-items [i-1].weight]+items
[i-1].value
```

```
        // 取两种选择中的较大值作为f [i] [j]的值
        f[i][j] = max(f[i - 1][j], f[i - 1][j - items[i - 1].weight] +
items[i - 1].value);
      }
    }
  }
  // 返回f [n] [m]作为最优解
  return f[n][m];
}

// 测试
int main() {
  // 定义一个物品数组，包含4个物品
  Item items[4] = {{2, 3}, {3, 4}, {4, 5}, {5, 6}};
  // 定义一个背包的容量，为8
  int m = 8;
  // 调用函数，求解0/1背包问题的最大价值
  int result = knapsack(items, 4, 8);
  // 输出结果
  cout << "The maximum value is: " << result << endl;
  return 0;
}

// 贪心算法实现
#include <iostream>
#include <algorithm>
using namespace std;

// 定义物品的结构体，包含重量，价值和单位价值
struct Item {
  int weight;
  int value;
  double unit_value;
};

// 定义一个比较函数，按照单位价值从高到低排序
bool compare(Item a, Item b) {
  return a.unit_value > b.unit_value;
}

// 定义一个函数，求解0/1背包问题的最大价值
// 参数为物品数组，物品个数，背包容量
// 返回值为最大价值
int knapsack(Item items[], int n, int m) {
```

```
  // 定义一个变量，表示当前背包的剩余容量
  int remain = m;
  // 定义一个变量，表示当前背包的总价值
  int result = 0;
  // 对物品数组按照单位价值从高到低排序
  sort(items, items + n, compare);
  // 遍历物品数组，依次选择单位价值最高的物品
  for (int i = 0; i < n; i++) {
    // 如果当前物品的重量小于等于背包的剩余容量，那么放入背包
    if (items[i].weight <= remain) {
      // 更新背包的剩余容量和总价值
      remain -= items[i].weight;
      result += items[i].value;
    } else {
      // 如果当前物品的重量大于背包的剩余容量，那么结束循环
      break;
    }
  }
  // 返回背包的总价值作为最优解
  return result;
}

// 测试
int main() {
  // 定义一个物品数组，包含 4 个物品
  Item items[4] = {{2, 3, 1.5}, {3, 4, 1.333}, {4, 5, 1.25}, {5, 6,
1.2}};
  // 定义一个背包的容量，为 8
  int m = 8;
  // 调用函数，求解 0/1 背包问题的最大价值
  int result = knapsack(items, 4, 8);
  // 输出结果
  cout << "The maximum value is: " << result << endl;
  return 0;
}
```

类推，第 n 层有 $m_0m_1\cdots m_{n-1}$ 个满足约束条件的结点，则采用回溯法求所有解的算法的时间为 $T(n)=m_0+m_0m_1+m_0m_1m_2+\cdots+m_0m_1m_2\cdots m_{n-1}$。

这是一种最基本的时间分析方法，在实际中并不一定如此，如第 1 层有 m_0 个满足约束条件的结点，但每个结点满足约束条件的结点并非都是 m_1 个。为了使估算更精确，可以选取若干条不同的随机路径，分别对各随机路径估算结点总数，然后再取这些结点总数的平均值。在通常情况下，回溯法的效率会高于蛮力法。

通常，解空间树为子集树时对应算法的时间复杂度为 $O(2^n)$，解空间树为排列树时对应算法的时间复杂度为 $O(n!)$。

5.2 求解 0/1 背包问题

0/1 背包问题的描述见第 4 章的 4.2.6 小节，在给定 w、v、W 的条件下求最大的装入物品价值和方案。第 4 章中采用蛮力法求解，这里采用回溯法求解，分为两种情况讨论。

1. 装入背包中物品重量和恰好为 W

扫一扫

视频讲解

设 n 个物品的编号为 1～n，重量用数组 $w[1..n]$ 存放，价值用数组 $v[1..n]$ 存放，限制重量用 W 表示。用 $x[1..n]$ 数组存放最优解，$x[i]=1$ 表示第 i 个物品放入背包中，$x[i]=0$ 表示第 i 个物品不放入背包中。

由于每个物品要么装入，要么不装入，其解空间是一棵子集树，树中每个结点表示背包的一种选择状态，记录当前放入背包的物品总重量和总价值，每个分枝结点下面有两条边表示对某物品是否放入背包的两种可能的选择。

对第 i 层上的某个分枝结点，对应的状态为 dfs(i,tw,tv,op)，其中 tw 表示装入背包中的物品总重量，tv 表示背包中物品总价值，op 记录一个解向量。该状态的两种扩展如下：

(1) 选择第 i 个物品放入背包：op[i]=1，tw=tw+$w[i]$，tv=tv+$v[i]$，转向下一个状态 dfs(i+1,tw,tv,op)。该决策对应左分枝。

(2) 不选择第 i 个物品放入背包：op[i]=0，tw 不变，tv 不变，转向下一个状态 dfs(i+1,tw,tv,op)。该决策对应右分枝。

解空间树中叶子结点表示已经对 n 个物品做了决策，对所有叶子结点进行比较求出满足 tw=W 的最大价值 maxv，对应的最优解 op 存放到 x 中。

对于表 4.2 所示的 4 个物品，在限制背包总重量 $W=6$ 时，描述问题求解过程的解空间树如图 5.10 所示，每个结点中两个数值为(tw,tv)。

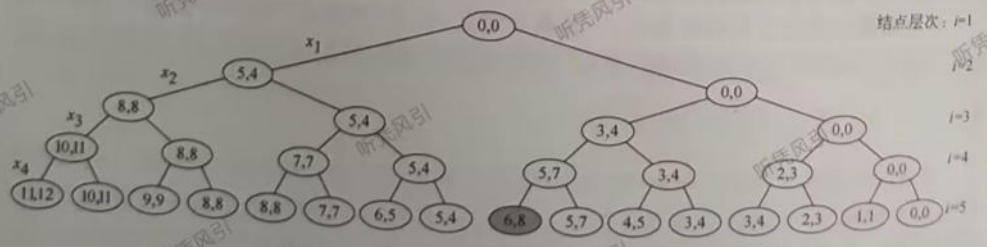


图 5.10 0/1 背包问题求解过程的解空间树

对于层次为1的根结点为(0,0)，考虑物品1，选择它：op[1]=1，tw=0+5=5，tv=0+4=4，产生新结点(5,4)作为根的左孩子；不选择它：op[1]=0，tw=0，tv=0，产生新结点(0,0)作为根的右孩子。以此类推可以构造出整棵子集树(共有$2^5-1=31$个结点)。采用递归框架设计的求解程序(其中dispasolution()函数用于输出最优解)如下：

```
#include <stdio.h>
#define MAXN 20                          //最多物品数
//问题表示
int n=4;                                 //4种物品
int W=6;                                 //限制重量为6
int w[]={0,5,3,2,1};                     //存放4个物品重量,不用下标0元素
int v[]={0,4,4,3,1};                     //存放4个物品价值,不用下标0元素
//求解结果表示
int x[MAXN];                             //存放最终解
int maxv=0;                              //存放最优解的总价值
void dfs(int i,int tw,int tv,int op[])   //求解0/1背包问题
{   if (i>n)                             //找到一个叶子结点
    {   if (tw==W && tv>maxv)            //找到一个满足条件的更优解,保存它
        {   maxv=tv;
            for (int j=1;j<=n;j++)
                x[j]=op[j];
        }
    }
    else                                 //尚未找完所有物品
    {   op[i]=1;                         //选取第i个物品
        dfs(i+1,tw+w[i],tv+v[i],op);
        op[i]=0;                         //不选取第i个物品,回溯
        dfs(i+1,tw,tv,op);
    }
}
void main()
{   int op[MAXN];                        //存放临时解
    dfs(1,0,0,op);                       //i从1开始
    dispasolution();
}
```

执行上述程序求出一种最佳装入方案是选取第2、3、4个物品，对应物品总重量为6，总价值为8。

从图5.10中看到，对于第i层的有些结点，tw+$w[i]$已超过了W，显然再选择$w[i]$是不合适的。如第2层的(5,4)结点，tw=5，$w[2]=3$，而tw+$w[2]>W$，选择物品2的扩展是不必要的，可以增加一个限界条件进行剪枝，如果选择物品i会导致超重，即tw+$w[i]>W$，就不再扩展该结点，也就是仅仅扩展tw+$w[i]\leqslant W$的左孩子结点。剪枝后的解空间树如图5.11所示(共21个结点，不计虚结点)。对应的带左孩子剪枝的dfs算法如下：

```
void dfs(int i,int tw,int tv,int op[])
{   if (i>n)
    {   if (tw==W && tv>maxv)
        {   maxv=tv;
            for (int j=1;j<=n;j++)
                x[j]=op[j];
        }
    }
    else
    {   if (tw+w[i]<=W)
        {   op[i]=1;
            dfs(i+1,tw+w[i],tv+v[i],op);
        }
        op[i]=0;
        dfs(i+1,tw,tv,op);
    }
}
```

//求解 0/1 背包问题
//找到一个叶子结点
//找到一个满足条件的更优解，保存它
//尚未找完所有物品
//左孩子结点剪枝
//选取第 i 个物品
//不选取第 i 个物品，回溯

从图 5.11 看到，只对左子树进行了剪枝，没有对右子树进行剪枝，下面考虑对右子树剪枝。用 rw 表示考虑第 i 个物品时剩余物品的重量，即 $rw=w[i]+\cdots+w[n]$（其中包含 $w[i]$），初始时 rw 是所有物品的重量和。对于第 i 层上的某个分枝结点（对应物品 i 的选择和不选择），其状态改为 dfs(i,tw,tv,rw,op)，对应的两种扩展如下：

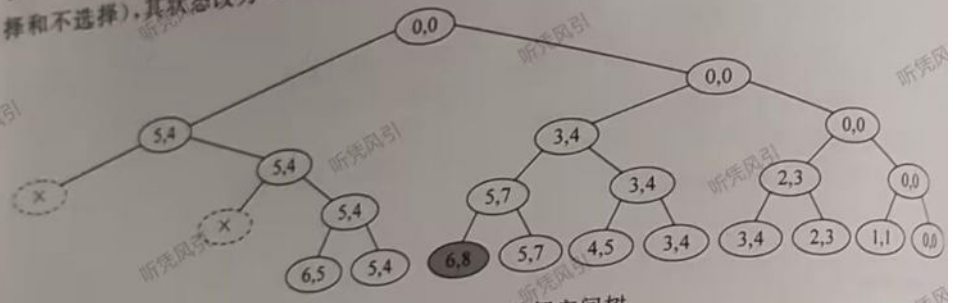


图 5.11　左剪枝后的解空间树

(1) 选择第 i 个物品放入背包（不超重，超重时左孩子剪枝）：op[i]=1，tw=tw+$w[i]$，tv=tv+$v[i]$，rw=rw−$w[i]$，转向下一个状态 dfs(i+1,tw,tv,rw,op)。该决策对应左分枝。

(2) 不选择第 i 个物品放入背包：op[i]=0，tw 不变，tv 不变，rw=rw−$w[i]$，转向下一个状态 dfs(i+1,tw,tv,rw,op)。该决策对应右分枝。

显然，当不选择物品 i 时，tw+rw−$w[i]$=tw+$w[i+1]+\cdots+w[n]$，若该值小于 W 时，也就是说即使选择后面的所有物品，背包中所有物品的重量也不会达到 W（该问题要求所有背包中物品的重量后恰好为 W），因此不必要再考虑扩展这样的右结点，也就是说，对于右分枝仅仅扩展 tw+rw−$w[i]\geqslant W$ 的结点。从而产生进一步剪枝后的解空间树如图 5.12 所示（共 9 个结点，不计虚结点），图中结点内的 3 个数字分别表示 tw、tv 和 rw。该算法仍能产生最优解，但比图 5.10 的结点减少一半以上，因此效率得到提高。

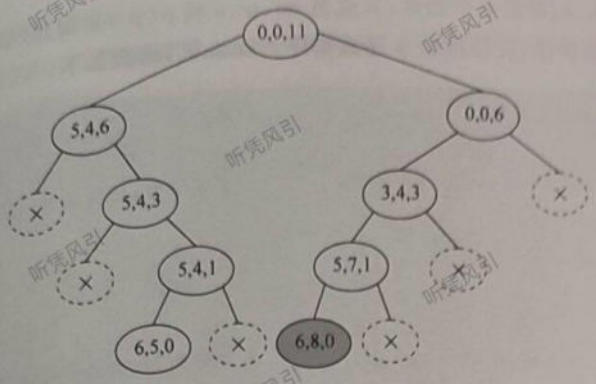


图 5.12 左右进一步剪枝后的解空间树

对应的算法(初始调用时 rw 为所有物品重量和)如下:

```
void dfs(int i,int tw,int tv,int rw,int op[])      //求解 0/1 背包问题
{   if (i>n)                                       //找到一个叶子结点
    {   if (tw==W && tv>maxv)                      //找到一个满足条件的更优解,保存它
        {   maxv=tv;
            for (int j=1;j<=n;j++)                 //复制最优解
                x[j]=op[j];
        }
    }
    else                                           //尚未找完所有物品
    {   if (tw+w[i]<=W)                            //左孩子结点剪枝
        {   op[i]=1;                               //选取第 i 个物品
            dfs(i+1,tw+w[i],tv+v[i],rw-w[i],op);
        }
        if (tw+rw-w[i]>=W)                         //右孩子结点剪枝
        {   op[i]=0;                               //不选取第 i 个物品,回溯
            dfs(i+1,tw,tv,rw-w[i],op);
        }
    }
}
```

从本问题求解过程看到,为了提高算法的效率,选择合理的限界条件是剪枝的关键,在第6章将进一步介绍这种剪枝技术。

2. 装入背包中物品重量和不超过 W

由于问题变为求背包中物品重量和不超过 W 的最大价值的装入方案,前面左剪枝方式不变,但右剪枝方式不再有效,改为采用上界函数进行右剪枝。

对于第 i 层上的某个分枝结点,其状态为 dfs(i,tw,tv,op),若不选择物品 i,设置对应的上界函数 bound(i)=tv+r(表示沿着该方向选择得到物品的价值上限),其中 r 表示剩余物品的总价值。假设当前求出最大价值 maxv,若 bound(i)≤maxv,则右剪枝,否则继续扩展。显然 r 越小,bound(i)也越小,剪枝越多,为了构造更小的 r,将所有物品以单位重量价值递减排列。

扫一扫

视频讲解

采用数组 A[1..n]存放 n 个物品，其成员 no、w、v 和 p（$p=v/w$）分别表示物品编号、重量、价值和单位重量价值，先按成员 p 递减排序。bound()函数如下：

```
int bound(int i,int tw,int tv)                    //求上界
{   i++;                                          //从 i+1 开始
    while (i<=n && tw+A[i].w<=W)                  //若序号为 i 的物品可以整个放入
    {   tw+=A[i].w;
        tv+=A[i].v;
        i++;
    }
    if (i<=n) return tv+(W-tw)*A[i].p;            //序号为 i 的物品不能整个放入
    else return tv;
}
```

采用这样的剪枝后，一旦找到一个解后，后面找到的其他解(tv,op)只能越来越优，将存放在最优解(maxv,x)中。对应的算法如下：

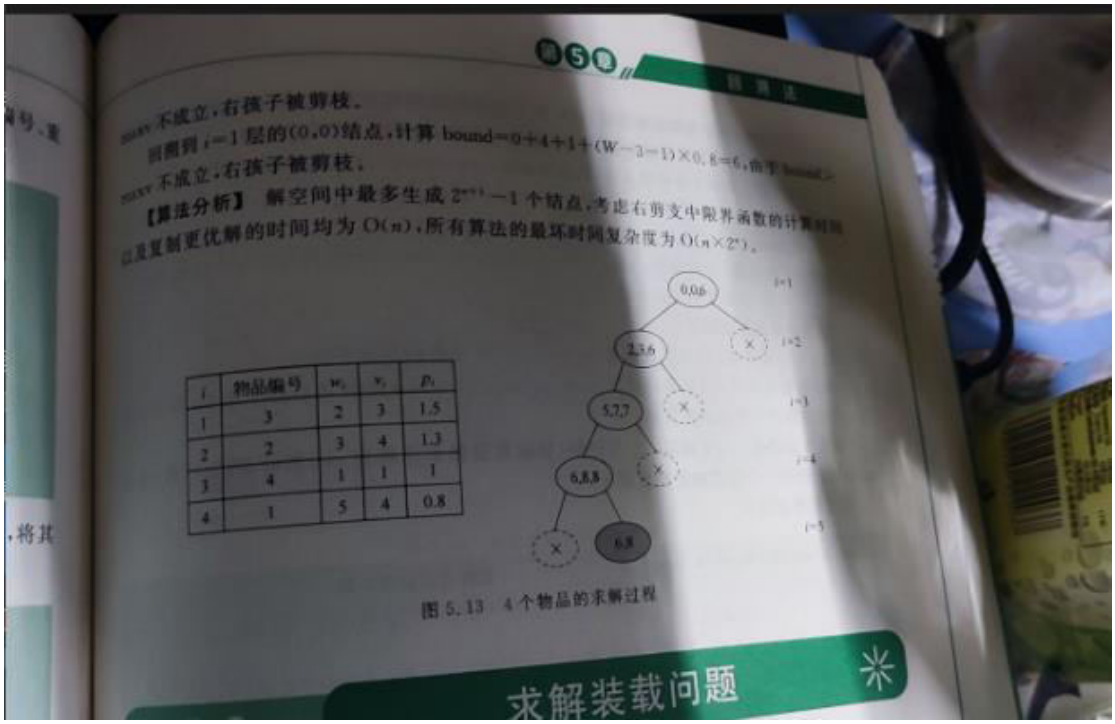
```
void dfs(int i,int tw,int tv,int op[])            //求解 0/1 背包问题
{   if (i>n)                                      //找到一个叶子结点
    {   maxv=tv;                                  //存放更优解
        for (int j=1;j<=n;j++)
            x[j]=op[j];
    }
    else                                          //尚未找完所有物品
    {   if (tw+A[i].w<=W)                         //左孩子结点剪枝
        {   op[i]=1;                              //选取序号为 i 的物品
            dfs(i+1,tw+A[i].w,tv+A[i].v,op);
        }
        if (bound(i,tw,tv)>maxv)                  //右孩子结点剪枝
        {   op[i]=0;                              //不选取序号为 i 的物品，回溯
            dfs(i+1,tw,tv,op);
        }
    }
}
```

如图 5.13 所示，对表 4.2 的 4 个物品按 p_i 递减排序，结点中的 3 个数字分别表示 tw、tv 和 bound(bound 仅仅在选择右分枝时计算出来)。

初始时 maxv=0，从根结点开始，沿着左分枝依次选择物品 3、2 和 4，到达 i=4 层的(6,8)结点，左孩子被剪枝，计算 bound=tv+0=8，bound>maxv 成立，不选择序号为 4 的物品(物品 1)，到达叶子结点得到一个解 maxv=8。

回溯到 i=3 层的(5,7)结点，计算 bound=7+(W−5)×0.8=7，由于 bound>maxv 成立，右孩子被剪枝。

回溯到 i=2 层的(2,3)结点，计算 bound=3+1+(W−2−1)×0.8=6，由于 bound



【0-1 背包问题 】详细解析+图解+详细代码

01 背包各种算法代码实现总结（穷举，贪心，动态，递归，回溯，分支限界）

动态规划——0/1 背包问题(全网最细+图文解析)「建议收藏」

c++—0/1 背包问题-贪心算法（详解）

6.采用优先队列式分支限界法求解

优先队列式分支限界法是一种在问题的解空间树上搜索最优解的算法，它的基本思想是：

用一个优先队列来存储活结点，即可能包含最优解的子树的根结点；

用一个函数来计算每个活结点的上界，即该结点所在子树的最优解的可能值；

每次从优先队列中选取一个上界最小（或最大）的结点作为当前扩展结点，生成其所有子结点，并计算它们的上界；

如果某个子结点的上界小于（或大于）当前最优解的值，就舍弃该子结点，否则将其加入优先队列；

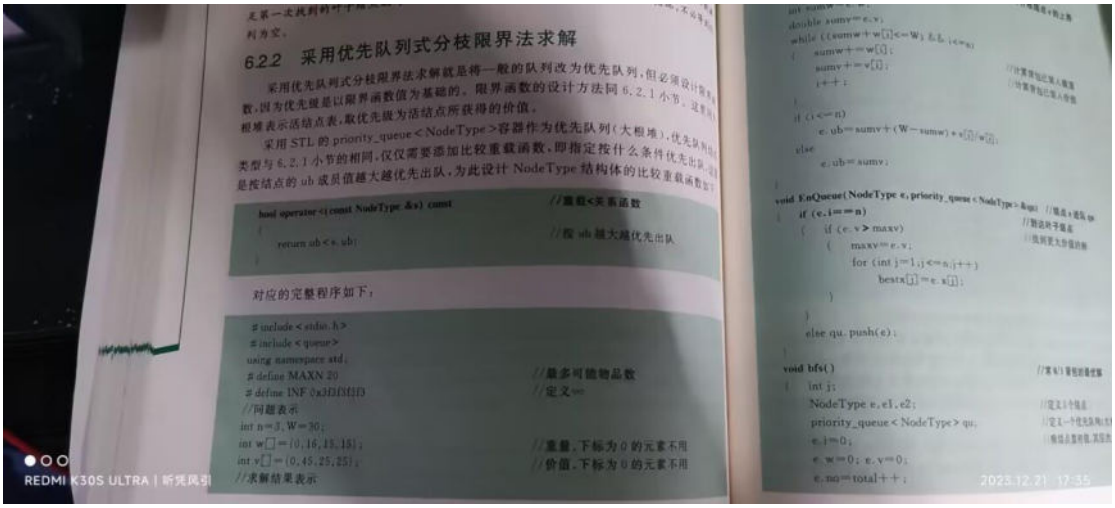
重复上述过程，直到找到最优解或优先队列为空。

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

// 定义物品的结构体，包含重量和价值
struct Item {
  int weight;
  int value;
};

// 定义结点的结构体，包含层次，重量，价值，上界和左右子结点
struct Node {
  int level;
  int weight;
  int value;
  double bound;
  Node* left;
  Node* right;
};

// 定义一个比较函数，按照上界从小到大排序
struct Compare {
  bool operator() (Node* a, Node* b) {
    return a->bound > b->bound;
  }
};
```

```
// 定义一个函数，计算结点的上界
double bound(Node* node, int n, int c, Item items[]) {
  // 如果结点的重量已经超过背包的容量，返回 0
  if (node->weight >= c) return 0;
  // 否则，初始化上界为结点的价值
  double result = node->value;
  // 初始化剩余容量为背包的容量减去结点的重量
  int remain = c - node->weight;
  // 初始化当前层次为结点的层次
  int level = node->level;
  // 从当前层次开始，尽可能地装入剩余物品
  while (level < n && items[level].weight <= remain) {
    remain -= items[level].weight;
    result += items[level].value;
    level++;
  }
  // 如果还有剩余物品，按照单位价值比例装入部分物品
  if (level < n) {
    result += items[level].value / (double)items[level].weight * remain;
  }
  // 返回上界
  return result;
}

// 定义一个函数，求解 0/1 背包问题的最大价值
// 参数为物品数组，物品个数，背包容量
// 返回值为最大价值
int knapsack(Item items[], int n, int c) {
  // 定义一个优先队列，存储活结点
  priority_queue<Node*, vector<Node*>, Compare> pq;
  // 定义一个变量，存储当前最大价值
  int max_value = 0;
  // 创建一个根结点，层次为-1，重量和价值为 0，上界为背包容量
  Node* root = new Node();
  root->level = -1;
  root->weight = 0;
  root->value = 0;
  root->bound = bound(root, n, c, items);
  // 将根结点加入优先队列
  pq.push(root);
  // 当优先队列不为空时，循环执行
  while (!pq.empty()) {
    // 取出优先队列的首元素，作为当前扩展结点
```

```
    Node* current = pq.top();
    pq.pop();
    // 如果当前结点的上界大于当前最大价值，继续搜索
    if (current->bound > max_value) {
      // 创建一个左子结点，表示选择当前层次的物品
      Node* left = new Node();
      // 左子结点的层次为当前结点的层次加一
      left->level = current->level + 1;
      // 左子结点的重量为当前结点的重量加上当前层次物品的重量
      left->weight = current->weight + items[left->level].weight;
      // 左子结点的价值为当前结点的价值加上当前层次物品的价值
      left->value = current->value + items[left->level].value;
      // 如果左子结点的重量小于等于背包的容量，且左子结点的价值大于当前最大价值，更新当前最大价值
      if (left->weight <= c && left->value > max_value) {
        max_value = left->value;
      }
      // 计算左子结点的上界
      left->bound = bound(left, n, c, items);
      // 如果左子结点的上界大于当前最大价值，将左子结点加入优先队列
      if (left->bound > max_value) {
        pq.push(left);
      }
      // 创建一个右子结点，表示不选择当前层次的物品
      Node* right = new Node();
      // 右子结点的层次，重量，价值和当前结点相同
      right->level = current->level + 1;
      right->weight = current->weight;
      right->value = current->value;
      // 计算右子结点的上界
      right->bound = bound(right, n, c, items);
      // 如果右子结点的上界大于当前最大价值，将右子结点加入优先队列
      if (right->bound > max_value) {
        pq.push(right);
      }
    }
  }
  // 返回当前最大价值
  return max_value;
}

// 测试
int main() {
  // 定义一个物品数组，包含 4 个物品
```

```
    Item items[4] = {{2, 3}, {3, 4}, {4, 5}, {5, 6}};
    // 定义一个背包的容量，为 8
    int c = 8;
    // 调用函数，求解 0/1 背包问题的最大价值
    int result = knapsack(items, 4, 8);
    // 输出结果
    cout << "The maximum value is: " << result << endl;
    return 0;
}
```



```
int maxv=-9999;                          //存放最大价值,全局变量
int bestx[MAXN];                         //存放最优解,全局变量
int total=1;                             //解空间中的结点数累计,全局变量
struct NodeType                          //队列中的结点类型
{   int no;                              //结点编号
    int i;                               //当前结点在搜索空间中的层次
    int w;                               //当前结点的总重量
    int v;                               //当前结点的总价值
    int x[MAXN];                         //当前结点包含的解向量
    double ub;                           //上界
    bool operator<(const NodeType &s) const   //重载<关系函数
    {
        return ub<s.ub;                  //ub 越大越优先出队
    }
};
void bound(NodeType &e)                  //计算分枝结点 e 的上界
{   int i=e.i+1;
    int sumw=e.w;
    double sumv=e.v;
    while ((sumw+w[i]<=W) && i<=n)
    {   sumw+=w[i];                      //计算背包已装入载重
        sumv+=v[i];                      //计算背包已装入价值
        i++;
    }
    if (i<=n)
        e.ub=sumv+(W-sumw)*v[i]/w[i];
    else
        e.ub=sumv;
}
void EnQueue(NodeType e,priority_queue<NodeType> &qu)   //结点 e 进队 qu
{   if (e.i==n)                          //到达叶子结点
    {   if (e.v>maxv)                    //找到更大价值的解
        {   maxv=e.v;
            for (int j=1;j<=n;j++)
                bestx[j]=e.x[j];
        }
    }
    else qu.push(e);
}
void bfs()                               //求 0/1 背包的最优解
{   int j;
    NodeType e,e1,e2;                    //定义 3 个结点
    priority_queue<NodeType> qu;         //定义一个优先队列(大根堆)
    e.i=0;                               //根结点置初值,其层次计为 0
    e.w=0; e.v=0;
    e.no=total++;
```

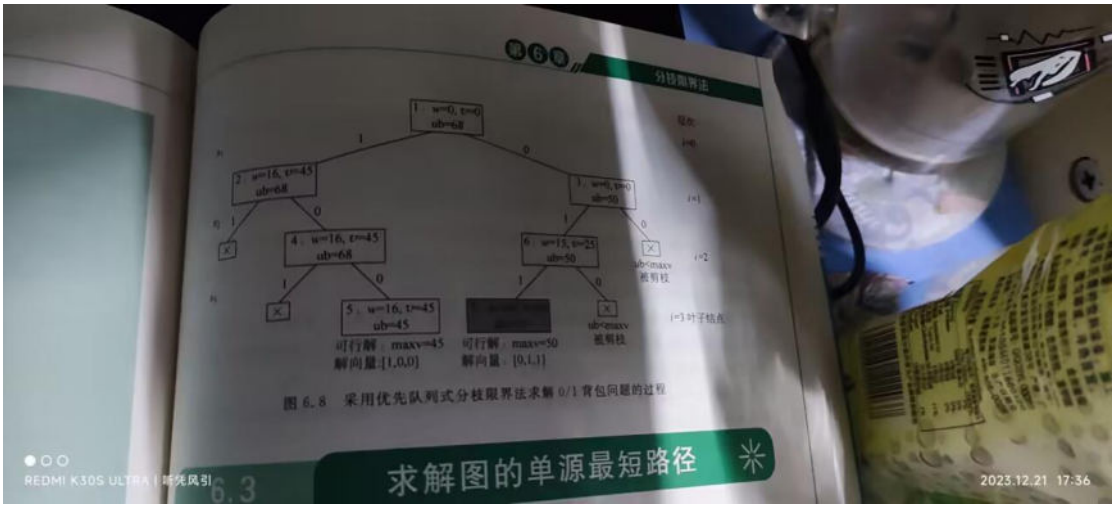
```
    for (j=1;j<=n;j++)
        e.x[j]=0;
    bound(e);
    qu.push(e);
    while (!qu.empty())
    {   e=qu.top(); qu.pop();
        if (e.w+w[e.i+1]<=W)
        {   e1.no=total++;
            e1.i=e.i+1;
            e1.w=e.w+w[e1.i];
            e1.v=e.v+v[e1.i];
            for (j=1;j<=n;j++)
                e1.x[j]=e.x[j];
            e1.x[e1.i]=1;
            bound(e1);
            EnQueue(e1,qu);
        }
        e2.no=total++;
        e2.i=e.i+1;
        e2.w=e.w; e2.v=e.v;
        for (j=1;j<=n;j++)
            e2.x[j]=e.x[j];
        e2.x[e2.i]=0;
        bound(e2);
        if (e2.ub>maxv)
            EnQueue(e2,qu);
    }
}
void main()
{   bfs();
    printf("分枝限界法求解 0/1 背包问题:\n X=[");
    for(int i=1;i<=n;i++)
        printf("%2d",bestx[i]);
    printf("],装入总价值为%d\n",maxv);
}
```

//求根结点的上界
//根结点进队
//队不空时循环
//出队结点 e
//剪枝：检查左孩子结点
//建立左孩子结点
//复制解向量
//求左孩子结点的上界
//左孩子结点进队
//建立右孩子结点
//复制解向量
//求右孩子结点的上界
//若右孩子结点可行，则进队，否则被剪枝
//调用队列式分枝限界法求 0/1 背包问题
//输出最优解
//输出所求 X[n]数组

该程序的执行结果与采用队列式分枝限界法求解程序的结果相同。

采用优先队列式分枝限界法求解上述 0/1 背包问题的搜索过程如图 6.8 所示，图中方"×"的结点表示死结点，带阴影的结点是最优解结点，结点的编号为搜索顺序。从中看到由于采用优先队列，结点的扩展不再是一层一层顺序展开的，而是按限界函数值的大小跳跃式选取扩展结点的。该求解过程实际搜索的结点个数比队列式求解要少，当物品数较多时这种效率的提高会更为明显。

【算法分析】 无论是采用队列式分枝限界法还是采用优先队列式分枝限界法求解 0/1 背包问题，在最坏情况下都要搜索整个解空间树，所以最坏时间和空间复杂度均为 $O(2^n)$，其中 n 为物品个数。



0-1 背包问题-分支限界法(优先队列分支限界法)

经典算法思想 4——分支限界（branch-and-bound）

秒懂算法 | 排列树模型——旅行商问题的分支限界法

优先队列式分支限界法求解单源最短路径

四、不知道题型是什么

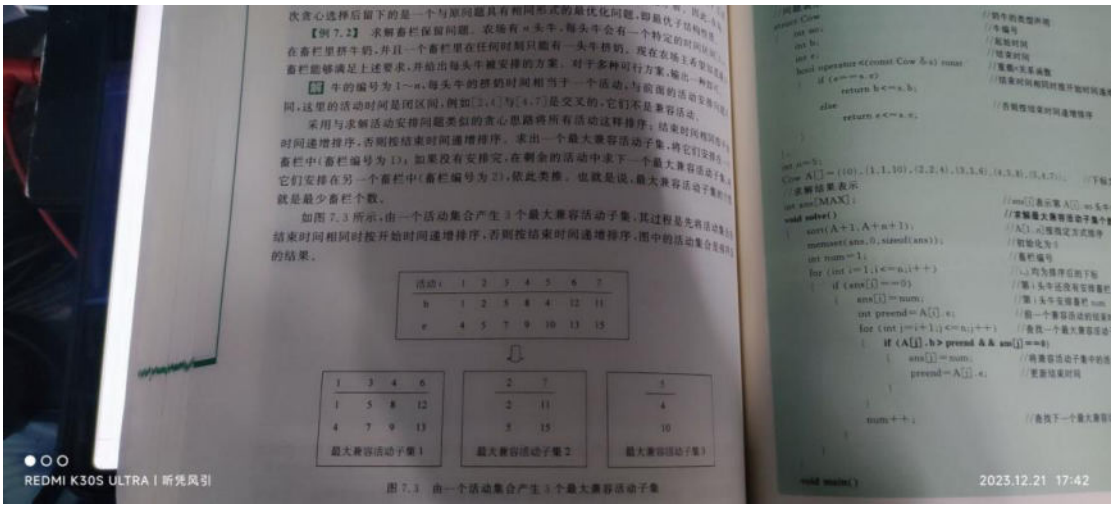
1.求解蓄栏保留问题

蓄栏保留问题是一种在给定一组牛的挤奶时间区间和一个畜栏的数量的情况下，选择一些牛放入畜栏，使得畜栏中牛的总数最大，同时满足每个畜栏中同时只能有一头牛在挤奶的约束。要求解蓄栏保留问题，可以用以下的思路：

贪心算法：按照牛的挤奶开始时间从小到大排序，用一个优先队列维护已有畜栏的结束时间，每次选择结束时间最早的畜栏来放入新的牛，如果没有合适的畜栏，就新建一个畜栏。这种方法的时间复杂度是 O(n*logn)，空间复杂度是 O(n)。

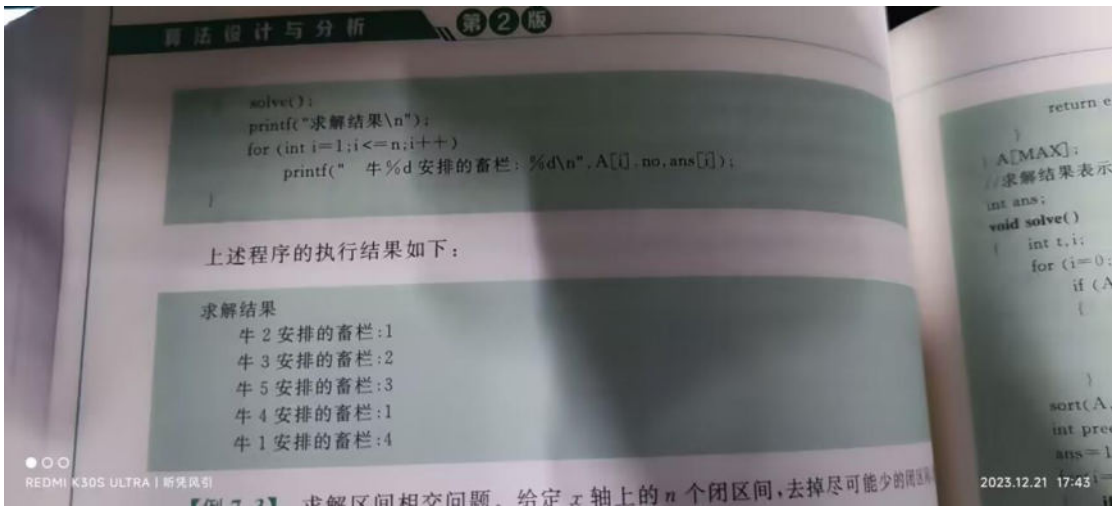
```
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;

// 定义牛的结构体，包含开始时间，结束时间和编号
struct Cow {
  int start;
  int end;
  int id;
};

// 定义一个比较函数，按照开始时间从小到大排序
bool compare(Cow a, Cow b) {
  return a.start < b.start;
}

// 定义一个函数，求解畜栏保留问题的最大牛数
// 参数为牛的数组，牛的个数，畜栏的个数
// 返回值为最大牛数
int barn(Cow cows[], int n, int m) {
  // 定义一个优先队列，存储畜栏的结束时间，按照从小到大排序
  priority_queue<int, vector<int>, greater<int>> pq;
  // 定义一个变量，存储当前的牛数
  int count = 0;
  // 对牛的数组按照开始时间从小到大排序
  sort(cows, cows + n, compare);
  // 遍历牛的数组，依次选择开始时间最早的牛
  for (int i = 0; i < n; i++) {
    // 如果优先队列为空，或者队首的结束时间大于当前牛的开始时间，表示没有合适的畜栏
    if (pq.empty() || pq.top() > cows[i].start) {
      // 如果畜栏的个数还没有达到上限，就新建一个畜栏，放入当前牛
      if (pq.size() < m) {
        pq.push(cows[i].end);
```

```
        count++;
      }
    } else {
      // 如果队首的结束时间小于等于当前牛的开始时间，表示有合适的畜栏
      // 将队首的畜栏弹出，更新其结束时间为当前牛的结束时间，再放回优先队列
      pq.pop();
      pq.push(cows[i].end);
    }
  }
  // 返回当前的牛数
  return count;
}

// 测试
int main() {
  // 定义一个牛的数组，包含 5 头牛
  Cow cows[5] = {{1, 10, 1}, {2, 4, 2}, {3, 6, 3}, {5, 8, 4}, {4, 7,
5}};
  // 定义一个畜栏的个数，为 4
  int m = 4;
  // 调用函数，求解蓄栏保留问题的最大牛数
  int result = barn(cows, 5, 4);
  // 输出结果
  cout << "The maximum number of cows is: " << result << endl;
  return 0;
}
```



建立一个活动标记数组 ans，ans[i]表示编号为 A[i].no 的牛安排挤奶的畜栏编号，ans[i]为 0 表示该牛尚未安排畜栏，将所有元素设置为 0，置当前选取的畜栏编号 num=1；从第一个活动开始寻找最大兼容活动子集 1，将其中所有活动编号 i 对应的 ans[i]设置为 num(1)；num=2，在所有 ans[i]=0 的活动集合中寻找最大兼容活动子集 2，将其中所有活动编号 i 对应的 ans[i]设置为 num(2)；依此类推，最后找出最大兼容活动子集 2，将其中所有活动编号 i 对应的 ans[i]设置为 num(2)；依此类推，最后找出最大兼容活动子集个数为 3。用数组 A 存放所有活动，用 ans 数组表示活动对应的畜栏编号(从 1 开始)。对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define MAX 51
//问题表示
struct Cow                                  //奶牛的类型声明
{   int no;                                 //牛编号
    int b;                                  //起始时间
    int e;                                  //结束时间
    bool operator<(const Cow &s) const      //重载<关系函数
    {   if (e==s.e)                         //结束时间相同时按开始时间递增排序
            return b<=s.b;
        else                                //否则按结束时间递增排序
            return e<=s.e;
    }
};
int n=5;
Cow A[]={{0},{1,1,10},{2,2,4},{3,3,6},{4,5,8},{5,4,7}};     //下标为 0 的元素不用
//求解结果表示
int ans[MAX];                               //ans[i]表示第 A[i].no 头牛的畜栏编号
void solve()                                //求解最大兼容活动子集个数
{   sort(A+1,A+n+1);                        //A[1..n]按指定方式排序
    memset(ans,0,sizeof(ans));              //初始化为 0
    int num=1;                              //畜栏编号
    for (int i=1;i<=n;i++)                  //i,j 均为排序后的下标
    {   if (ans[i]==0)                      //第 i 头牛还没有安排畜栏
        {   ans[i]=num;                     //第 i 头牛安排畜栏 num
            int preend=A[i].e;              //前一个兼容活动的结束时间
            for (int j=i+1;j<=n;j++)        //查找一个最大兼容活动子集
            {   if (A[j].b>preend && ans[j]==0)
                {   ans[j]=num;             //将兼容活动子集中的活动安排在 num 畜栏中
                    preend=A[j].e;          //更新结束时间
                }
            }
            num++;                          //查找下一个最大兼容活动子集，num 增 1
        }
    }
}
void main()
```



4144.畜栏保留问题【贪心+优先队列】

4144:畜栏保留问题

贪心法求解蓄栏保留问题

2.求解田忌赛马问题

田忌赛马问题是一个经典的贪心算法问题，它的目标是在给定田忌和齐王的每匹马的速度和每场比赛的赌注的情况下，选择一些马与齐王的马进行比赛，使得田忌赢得的总赌注最大，同时满足每场比赛只能有一匹马参加的约束。要求解田忌赛马问题，可以用以下的思路：

将田忌和齐王的马按照速度从快到慢排序，用两个指针分别指向田忌和齐王的当前最快的马；

对于每一场比赛，比较田忌和齐王的当前最快的马的速度，有三种情况：

如果田忌的马比齐王的马快，那么就让这两匹马比赛，田忌赢得赌注，然后移动两个指针到下一匹马；

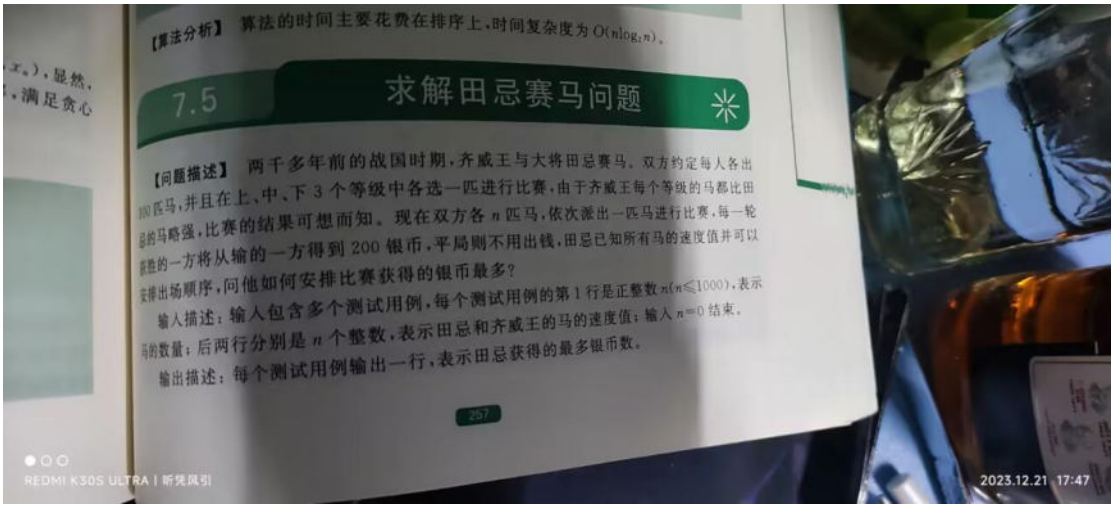
如果田忌的马比齐王的马慢，那么就让田忌的当前最慢的马与齐王的当前最快的马比赛，田忌输掉赌注，然后移动两个指针到下一匹马；

如果田忌的马和齐王的马一样快，那么就让田忌的当前最慢的马与齐王的当前最快的马比赛，如果田忌的马快，就赢得赌注，否则就输掉赌注，然后移动两个指针到下一匹马；

重复上述过程，直到所有的马都比完，计算田忌赢得的总赌注。

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

// 定义一个比较函数，按照速度从大到小排序
bool compare(int a, int b) {
  return a > b;
}

// 定义一个函数，求解田忌赛马问题的最大赌注
// 参数为田忌的马的速度数组，齐王的马的速度数组，马的数量，每场比赛的赌注
// 返回值为最大赌注
int race(int tianji[], int qiwang[], int n, int bet) {
  // 定义一个变量，表示田忌赢得的总赌注，初始为 0
  int result = 0;
  // 将田忌和齐王的马按照速度从大到小排序
  sort(tianji, tianji + n, compare);
  sort(qiwang, qiwang + n, compare);
  // 定义两个指针，分别指向田忌和齐王的当前最快的马，初始为 0
  int ti = 0;
  int qi = 0;
  // 遍历所有的马，进行比赛
  for (int i = 0; i < n; i++) {
    // 比较田忌和齐王的当前最快的马的速度
    if (tianji[ti] > qiwang[qi]) {
      // 如果田忌的马快，就赢得赌注，移动两个指针到下一匹马
      result += bet;
      ti++;
      qi++;
    } else if (tianji[ti] < qiwang[qi]) {
```

```
      // 如果田忌的马慢，就输掉赌注，移动两个指针到下一匹马
      result -= bet;
      ti++;
      qi++;
    } else {
      // 如果田忌和齐王的马一样快，就让田忌的当前最慢的马与齐王的当前最快的马比赛
      if (tianji[n - 1 - i] > qiwang[qi]) {
        // 如果田忌的马快，就赢得赌注，移动两个指针到下一匹马
        result += bet;
        ti++;
        qi++;
      } else {
        // 如果田忌的马慢或一样快，就输掉赌注，移动两个指针到下一匹马
        result -= bet;
        ti++;
        qi++;
      }
    }
  }
  // 返回田忌赢得的总赌注
  return result;
}

// 测试
int main() {
  // 定义一个田忌的马的速度数组，包含 4 匹马
  int tianji[4] = {92, 83, 71, 68};
  // 定义一个齐王的马的速度数组，包含 4 匹马
  int qiwang[4] = {95, 87, 74, 69};
  // 定义每场比赛的赌注，为 200 两
  int bet = 200;
  // 调用函数，求解田忌赛马问题的最大赌注
  int result = race(tianji, qiwang, 4, bet);
  // 输出结果
  cout << "The maximum bet is: " << result << endl;
  return 0;
}
```


【算法分析】 算法的时间主要花费在排序上，时间复杂度为 $O(n\log_2 n)$。
7.5 求解田忌赛马问题
【问题描述】 两千多年前的战国时期，齐威王与大将田忌赛马。双方约定每人各出
200匹马，并且在上、中、下3个等级中各选一匹进行比赛，由于齐威王每个等级的马都比田
忌的马略强，比赛的结果可想而知。现在双方各 n 匹马，依次派出一匹马进行比赛，每一轮
获胜的一方将从输的一方得到200银币，平局则不用出钱，田忌已知所有马的速度值并可以
安排出场顺序，问他如何安排比赛获得的银币最多？
输入描述：输入包含多个测试用例，每个测试用例的第1行是正整数 $n(n\leqslant 1000)$，表示
马的数量；后两行分别是 n 个整数，表示田忌和齐威王的马的速度值；输入 $n=0$ 结束。
输出描述：每个测试用例输出一行，表示田忌获得的最多银币数。
257
REDMI K30S ULTRA | 听凭风引
2023.12.21 17:47

输入样例：

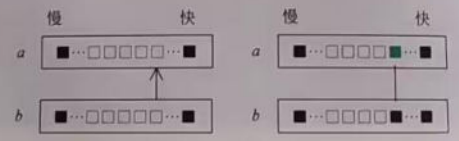
```
3
92 83 71
95 87 74
2
20 20
20 20
2
20 19
22 18
0
```

样例输出：

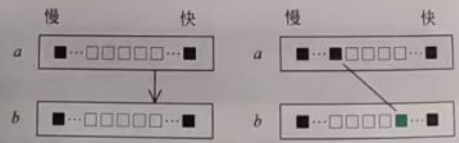
```
200
0
0
```

【问题求解】 田忌的马的速度用数组 a 表示，齐威王的马的速度用数组 b 表示，将 a、b 数组递增排序。采用常识性的贪心思路，分以下几种情况：

(1) 田忌最快的马比齐威王最快的马快，即 $a[\text{righta}]>b[\text{rightb}]$，则两者比赛（两个最快的马比赛），田忌赢。因为此时田忌最快的马一定赢，而选择与齐威王最快的马比赛对于田忌来说是最优的，如图 7.6(a)所示，图中“■”代表已经比赛的马，“□”代表尚未比赛的马，箭头指向的马速度更快。



(a) $a[\text{righta}]>b[\text{rightb}]$：ans+=200



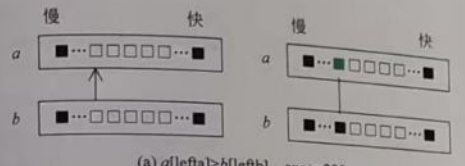
(b) $a[\text{righta}]<b[\text{rightb}]$：ans-=200

图 7.6 两者最快的马的速度不相同

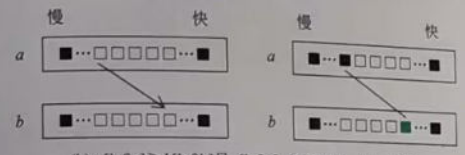
(2) 田忌最快的马比齐威王最快的马慢，即 $a[\text{righta}]<b[\text{rightb}]$，则选择田忌最慢的马与齐威王最快的马比赛，田忌输。因为齐威王最快的马一定赢，而选择与田忌最慢的马比赛对于田忌来说是最优的，如图 7.6(b)所示。

(3) 若田忌最快的马与齐威王最快的马的速度相同，即 $a[\text{righta}]=b[\text{rightb}]$，又分以下 3 种情况。

① 田忌最慢的马比齐威王最慢的马快，即 $a[lefta]>b[leftb]$，则两者比赛(两个最慢的马比赛)，田忌赢。因为此时齐威王最慢的马一定输，而选择与田忌最慢的马比赛对于田忌来说是最优的，如图 7.7(a)所示。



(a) $a[lefta]>b[leftb]$：ans+=200



(b) $a[lefta]\geqslant b[leftb]$且$a[lefta]<b[rightb]$：ans-=200

图 7.7　两者最快的马的速度相同

② 田忌最慢的马比齐威王最慢的马慢，并且田忌最慢的马比齐威王最快的马慢，即 $a[lefta]\leqslant b[leftb]$且 $a[lefta]<b[rightb]$，则选择田忌最慢的马与齐威王最快的马比赛，田忌输。因为此时田忌最慢的马一定输，而选择与齐威王最快的马比赛对于田忌来说是最优的，如图 7.7(b)所示。

③ 其他情况，即 $a[righta]=b[rightb]$且 $a[lefta]\leqslant b[leftb]$且 $a[lefta]\geqslant b[rightb]$，则 $a[lefta]\geqslant b[rightb]=a[righta]$，即 $a[lefta]=a[righta]$，$b[leftb]\geqslant a[lefta]=b[rightb]$，即 $b[leftb]=b[rightb]$，说明比赛区间的所有马的速度全部相同，任何两匹马比赛都没有输赢。

从上述过程看出每种情况对于田忌来说都是最优的，因此最终获得的比赛方案也一定是最优的。对应的完整程序如下：

```
#include <stdio.h>
#include <algorithm>
using namespace std;
#define MAX 1001
//问题表示
int n;
int a[MAX];
int b[MAX];
//求解结果表示
int ans;                                    //田忌获得的最多银币数
void solve()                                //求解算法
    sort(a,a+n);                            //对 a 递增排序
    sort(b,b+n);                            //对 b 递增排序
    ans=0;
    int lefta=0,leftb=0;
    int righta=n-1,rightb=n-1;
    while (lefta<=righta)                   //比赛直到结束
        if (a[righta]>b[rightb])            //田忌最快的马比齐威王最快的马快，两者比赛
```

```
        {   ans+=200;
            righta--;
            rightb--;
        }
        else if (a[righta]< b[rightb])          //田忌最快的马比齐威王最快的马慢
        {   ans-=200;                           //选择田忌最慢的马与齐威王最快的马比赛
            lefta++;
            rightb--;
        }
        else                                    //田忌最快的马与齐威王最快的马的速度相同
        {   if (a[lefta]> b[leftb])             //田忌最慢的马比齐威王最慢的马快,两者比赛
            {   ans+=200;
                lefta++;
                leftb++;
            }
            else
            {   if (a[lefta]< b[rightb])        //否则用田忌最慢的马与齐威王最快的马比赛
                    ans-=200;
                lefta++;
                rightb--;
            }
        }
    }
}
int main()
{   while (true)
    {   scanf("%d",&n);
        if (n==0) break;
        for (int i=0;i<n;i++)
            scanf("%d",&a[i]);
        for (int j=0;j<n;j++)
            scanf("%d",&b[j]);
        solve();
        printf("%d\n",ans);
    }
    return 0;
}
```

【算法分析】 算法的时间主要花费在排序上,时间复杂度为 $O(n\log_2 n)$。

7.6 求解多机调度问题

【问题描述】 设有 n 个独立的作业 $\{1,2,\cdots,n\}$,由 m 台相同的机器 $\{1,2,\cdots,m\}$ 进行加工处理,作业 i 所需的处理时间为 $t_i(1\leqslant i\leqslant n)$,每个作业均可在任何一台机器上加工处理,但未完工前不允许中断,任何作业也不能拆分成更小的子作业。多机调度问题要求给出一种作业调度方案,使所给的 n 个作业在[illegible]时间内由 m 台机器加工处理完成。

3.动态规划最短路径

动态规划最短路径是一种在给定一个有向图和一个起点的情况下，求出从起点到其他所有点的最短路径的算法。要求解动态规划最短路径，可以用以下的思路：

定义一个一维数组 d，表示从起点到每个点的最短距离，初始值为无穷大，除了起点的距离为 0；

定义一个一维数组 p，表示每个点的前驱节点，初始值为-1，表示没有前驱；

定义一个集合 s，表示已经确定最短距离的点，初始为空；

重复以下步骤，直到所有点都加入 s：

从 d 中选择一个不在 s 中的最小值，记为 d[u]，将 u 加入 s；

遍历所有从 u 出发的边，对于每个边的终点 v，如果 d[u]+w(u,v)<d[v]，就更新 d[v]为 d[u]+w(u,v)，并更新 p[v]为 u，其中 w(u,v)表示边(u,v)的权重；

返回 d 和 p 作为最短路径的结果。

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

// 定义一个常量，表示无穷大
const int INF = INT_MAX;

// 定义一个函数，求解动态规划最短路径
// 参数为有向图的邻接矩阵，顶点个数，起点
// 返回值为最短距离数组和前驱节点数组
```

```
pair<vector<int>, vector<int>> shortest_path(vector<vector<int>> graph,
int n, int start) {
  // 定义一个一维数组 d，表示从起点到每个点的最短距离，初始值为无穷大，除了起点
的距离为 0
  vector<int> d(n, INF);
  d[start] = 0;
  // 定义一个一维数组 p，表示每个点的前驱节点，初始值为-1，表示没有前驱
  vector<int> p(n, -1);
  // 定义一个集合 s，表示已经确定最短距离的点，初始为空
  vector<bool> s(n, false);
  // 重复 n 次以下步骤
  for (int i = 0; i < n; i++) {
    // 定义一个变量 u，表示当前要加入 s 的点，初始值为-1
    int u = -1;
    // 定义一个变量 min，表示当前最小的距离，初始值为无穷大
    int min = INF;
    // 遍历 d 中的每个元素
    for (int j = 0; j < n; j++) {
      // 如果该元素不在 s 中，且小于 min，就更新 u 和 min
      if (!s[j] && d[j] < min) {
        u = j;
        min = d[j];
      }
    }
    // 如果 u 为-1，表示没有找到合适的点，结束循环
    if (u == -1) break;
    // 否则，将 u 加入 s
    s[u] = true;
    // 遍历所有从 u 出发的边
    for (int v = 0; v < n; v++) {
      // 如果该边存在，且 d[u]+w(u,v)<d[v]，就更新 d[v]和 p[v]
      if (graph[u][v] != INF && d[u] + graph[u][v] < d[v]) {
        d[v] = d[u] + graph[u][v];
        p[v] = u;
      }
    }
  }
  // 返回 d 和 p 作为最短路径的结果
  return make_pair(d, p);
}

// 测试
int main() {
  // 定义一个有向图的邻接矩阵，用 INF 表示没有边
```

```
  vector<vector<int>> graph = {
    {0, 4, INF, 2, INF},
    {4, 0, 4, 1, INF},
    {INF, 4, 0, 1, 3},
    {2, 1, 1, 0, 7},
    {INF, INF, 3, 7, 0}
  };
  // 定义顶点个数
  int n = 5;
  // 定义起点
  int start = 0;
  // 调用函数，求解动态规划最短路径
  pair<vector<int>, vector<int>> result = shortest_path(graph, n,
start);
  // 输出结果
  cout << "The shortest distances from " << start << " are: " << endl;
  for (int i = 0; i < n; i++) {
    cout << i << ": " << result.first[i] << endl;
  }
  cout << "The predecessors of each vertex are: " << endl;
  for (int i = 0; i < n; i++) {
    cout << i << ": " << result.second[i] << endl;
  }
  return 0;
}
```

动态规划(Dynamic Programming,DP)是将多阶段决策问题进行公式化的一种技术，由 R. Bellman 于 1957 年提出，被成功应用于许多领域，也是算法设计方法之一。本章介绍动态规划求解问题的一般方法，并讨论一些用动态规划求解的经典示例。

8.1 动态规划概述

动态规划并非是“动态编程”或者“动态查询设计”。动态规划法通常基于一个递推公式及一个或多个初始状态，当前子问题的解将由上一次子问题的解推出。许多看起来复杂的问题采用动态规划法求解十分方便，而且只需要多项式时间复杂度，比回溯法，暴力法等效率高，但并非任何问题都适合采用动态规划法求解，本节介绍其相关概念。

8.1.1 从求解斐波那契数列看动态规划法

在第 2 章中讨论过求解斐波那契数列的递归算法，这里以求 Fib(5)为例可以看出如下几点：

(1) 递归调用 Fib(5)采用自顶向下的执行过程，从调用 Fib(5)开始到计算出 Fib(5)结束。

(2) 计算过程中存在大量的重复计算，例如求 Fib(5)的过程如图 8.1 所示，存在两次计算 Fib(3)的值的情况。

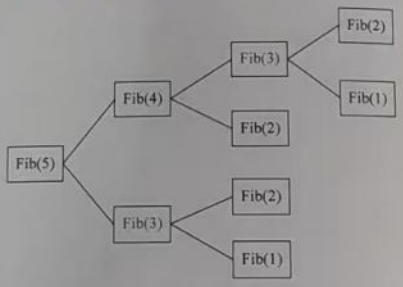


图 8.1 求 Fib(5)的过程

为了避免重复设计，设计一个 dp 数组，dp[i]存放 Fib(i)的值，首先设置 dp[1]和 dp[2]均为 1，再让 i 从 3 到 n 循环以计算 dp[3]到 dp[n]的值，最后返回 dp[n]，即 Fib(n)。对应的算法 1 如下：

```
int count=1;                                    //累计求解步骤
int Fib1(int n)                                 //求斐波那契数列的算法 1
{   dp[1]=dp[2]=1;
    printf("(%d)计算出 Fib1(1)=1\n",count++);
    printf("(%d)计算出 Fib1(2)=1\n",count++);
    for (int i=3;i<=n;i++)
```

```
        dp[i]=dp[i-1]+dp[i-2];
        printf("(%d)计算出 Fib1(%d)=%d\n",count++,i,dp[i]);
    }
    return dp[n];
```

执行 Fib1(5)时的输出结果如下：

```
(1) 计算出 Fib1(1)=1
(2) 计算出 Fib1(2)=1
(3) 计算出 Fib1(3)=2
(4) 计算出 Fib1(4)=3
(5) 计算出 Fib1(5)=5
```

显然这种方法的执行效率得到提高，执行过程改变为自底向上，即先求出子问题的解，将计算结果存放在一张表中，而且相同的子问题只计算一次，在后面需要时只要简单查一下，以避免大量的重复计算。求 Fib1(5)的过程如图 8.2 所示(图中带阴影框的结果是直接查表得到的)。

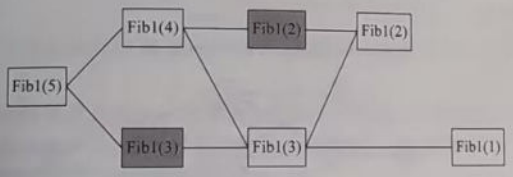


图 8.2 求 Fib1(5)的过程

上述求斐波那契数列的算法 1 属于动态规划法，其中数组 dp(表)称为动态规划数组。动态规划法也称为记录结果再利用的方法，其基本求解过程如图 8.3 所示。

图 8.3 动态规划法的求解过程

8.1.2 动态规划的原理

动态规划是一种解决多阶段决策问题的优化方法，把多阶段过程转化为一系列单阶段

【算法】用动态规划求解最短路径问题

动态规划法解多段图最短路径问题

【算法】动态规划之图的最短路径（C++源码）

c++解决动态规划最短路径

【动态规划/路径问题】进阶「最小路径和」问题
