目录

搜索剪枝

1. 概述

搜索的算法时间复杂度大多是指数级的。即使是简单的不加优化的搜索，其时间效率也低得让人无法忍受，难以满足我们竞赛时对程序的运行时间的要求。所以建立算法结构之后，有一种对程序进行优化的基本方法——剪枝。

1.1. 剪枝优化三原则

1）正确性 正如上文所述，枝条不是爱剪就能剪的. 如果随便剪枝，把带有最优解的那一分支也剪掉了的话，剪枝也就失去了意义. 所以，剪枝的前提是一定要保证不丢失正确的结果。

2）准确性 在保证了正确性的基础上，我们应该根据具体问题具体分析，采用合适的判断手段，使不包含最优解的枝条尽可能多的被剪去，以达到程序“最优化”的目的。可以说，剪枝的准确性，是衡量一个优化算法好坏的标准。

3）高效性 设计优化程序的根本目的，是要减少搜索的次数，使程序运行的时间减少. 但为了使搜索次数尽可能的减少，我们又必须花工夫设计出一个准确性较高的优化算法，而当算法的准确性升高，其判断的次数必定增多，从而又导致耗时的增多，这便引出了矛盾. 因此，如何在优化与效率之间寻找一个平衡点，使得程序的时间复杂度尽可能降低，同样是非常重要的. 倘若一个剪枝的判断效果非常好，但是它却需要耗费大量的时间来判断、比较，结果整个程序运行起来也跟没有优化过的没什么区别，这样就太得不偿失了。

1.2. 剪枝技巧

1) 优化搜索顺序：在一些搜索问题中，搜索树的各个层次，各个分支之间的顺序是不固定的。不同的搜索顺序会产生不同的搜索树形态，其规模大小也相差甚远。

2) 排除等效冗余：在搜索过程中，如果我们能够判定从搜索树的当前节点上沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支执行搜索。

3) 可行性剪枝（上下界剪枝）：该方法判断继续搜索能否得出答案，如果不能直接回溯。 在搜索过程中，及时对当前状态进行检查，如果发现分支已经无法到达递归边界，就执行回溯。

4) 最优性剪枝：最优性剪枝，是一种重要的搜索剪枝策略。它记录当前得到的最优值，如果当前结点已经无法产生比之前最优解更优的解时，可以提前回溯。

5) 记忆化：可以记录每个状态的搜索结果，再重复遍历一个状态时直接检索并返回。这好比我们对图进行深度优先遍历时，标记一个节点是否已经被访问过。

2. 基础题目

2.1. 数的划分（可行性剪枝，上下界剪枝）

【题目描述】

将整数 n 分成 k 份，且每份不能为空，任意两种划分方案不能相同(不考虑顺序)。

例如：n=7，k=3，下面三种划分方案被认为是相同的。

1 1 5

1 5 1

5 1 1

问有多少种不同的分法。

【输入格式】

n，k（6<n<=200，2<=k<=6）。

【输出格式】

一个整数，即不同的分法

【样例输入】

```
7 3
```

【样例输出】

```
4
```

【输出格式】

本题就是要求把数字 n 无序划分为 k 份的方案数。 也就是 x1+x2+……xk=n(x1<=x2……<=xk)的解数。搜索的方法是依次枚举 x1,x2……xk 的值，然后判断。

约束条件：

1）由于分解数不考虑顺序，因此设定分解数非递降，所以扩展结点时的“下界”应是不小于前一个扩展结点的值，即 a[i－1]≤a[i]。

2）假设我们将 n 已经分解成了 a[1]+a[2]+...+a[i－1]，则 a[i]的最大值为将 i~k 这 k-i+1 份平均划分，即设 m=n－(a[1]+a[2]+...+a[i－1])，则 a[i]≤m/(k－i+1)，所以扩展结点的“上界”是 m/(k－i+1)。

【参考代码】

```
#include <iostream>
using namespace std;
int n,m,a[8],s=0;
void pr()
{
    for(int i=1;i<=m;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
}
void dfs(int k) //分第 k 份
{
    if(n==0)
    {
        return;
    }
    if(k==m)
    {
        if(n>=a[k-1])
        {
            s++;
        }
        a[k]=n;
        pr();
        return;
    }
    for(int i=a[k-1];i<=n/(m-k+1);i++) //第 K 份的上下界
    {
        a[k]=i;
        n=n-i;
```

```
            dfs(k+1);
            n=n+i;
        }
    }
int main()
{
    cin>>n>>m;
    a[0]=1;
    dfs(1);
    cout<<s<<endl;
    return 0;
}
```

2.2. 小木棍(最优性剪枝，可行性剪枝)

【问题描述】

乔治有一些同样长的小木棍， 他把这些木棍随意砍成几段，已知每段的长都不超过 50。 现在他想把小木棍拼接成原来的样子，但是却忘记了自己开始时有多少根木棍和它们的长度。给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

【输入格式】

第一行为一个整数 N，表示砍过以后的小木棍的总数，其中 N≤60。

第二行为 N 个用空格隔开的正整数，表示 N 根小木棍的长度。

【输出格式】

输出文件仅一行，表示要求的原始最短木棍的可能长度。

【样例输入】

```
9
5 2 1 5 2 1 5 2 1
```

【样例输出】

```
6
```

【算法分析】

最优性剪枝：

1、设所有木棍的长度和为 sum，原长度一定能被 sum 整除；

2、木棍原始长度一定不小于所有木棍中最长的那个；

可行性剪枝：

3、长木棍肯定比几根短木棍拼成同样长度的用处小，短木棍更灵活，可以对木棍从大到小排序；

4、当用木棍 i 拼凑原始木棍时，可以从第 i+1 后的木棍开始搜，因为根据优化 1，i 前面的木棍已经用过了；

5、用当前最长长度的木棍开始搜，如果拼不出当前设定的圆木棍长度 len，则直接返回，换一个原始木棍长度 len；

6、相同长度的木棍不要搜索多次。用当前长度的木棍得不出结果是，用同一支长度相同的还是得不到结果，可以提前返回；

7、判断收到的几根木棍组成的长度是否大于原始长度 len，如果大于，可以提前返回；

8、判断当前剩的木棍根数是否够拼成木棍，如果不够，直接返回；

9、找到结果后，在能返回的地方法返回到上一层的递归处。

【参考代码】

```
#include <iostream>
#include <algorithm>
#include <cstring>
```

```
using namespace std;
int a[105],used[105],n,len,m,mmin=0,sum=0,bj;
bool cmp(const int &x,const int &y)
{
    return x>y;
}
void dfs(int k,int last,int rest)//第 k 跟木棍，last 为上一节木棍编号，rest 为第 k 根木棍还需要的长度
{
    int i,j;
    if(k==m)// 剪枝 9
    {
        bj=1;
        return;
    }
    if(rest==0)
    {
        for(i=1;i<=n;i++)//剪枝 4
        {
            if(!used[i])
            {
                used[i]=1;break;
            }
        }
        dfs(k+1,i,len-a[i]);
    }
    for(i=last+1;i<=n;i++)//剪枝 5 和 7
    {
        if(!used[i]&& rest>=a[i])
        {
            used[i]=1; //当前木棍已使用
            dfs(k,i,rest-a[i]);
            used[i]=0;
            j=i;
            while(i<n && a[i]==a[j])i++;//剪枝 6
            if(i==n)
            {
                return;
            }
        }
    }
}
void solve()
{
    int i,j;
    for(i=mmin;i<=sum;i++)//剪枝 2
    {
        if(sum%i==0)//剪枝 1
        {
            memset(used,0,sizeof(used));
            len=i;
            used[1]=1;
```

```
            bj=0;
            m=sum/i;//木棍的根数
            dfs(1,1,len-a[1]);
            if(bj)
            {
                cout<<len<<endl;
                break;
            }
        }
    }
}
int main()
{
    cin>>n;
    for(int i=1;i<=n;i++)
    {
        cin>>a[i];
        mmin=max(mmin,a[i]);
        sum=sum+a[i];
    }
    sort(a+1,a+1+n,cmp);
    solve();
    return 0;
}
```

2.3. weight（搜索对象的选择）

【问题描述】

已知原数列 a1 ，a2 ，…，an 中的前 1 项，前 2 项，前 3 项，… ，前 n 项的和，以及后 1 项，后 2 项，后 3 项，…，后 n 项的和，但是所有的数都被打乱了顺序。此外，我们还知道数列中的数存在于集合 S 中。试求原数列。当存在多组可能的数列时，求左边的数最小的数列。

【输入格式】

第 1 行，一个整数 n。

第 2 行，2n 个正整数，注意：数据已被打乱。

第 3 行，一个整数 m ，表示 S 集合的大小。

第 4 行， m 个整数，表示 S 集合中的元素。

【输出格式】

输出满足条件的最小序列。

【样例输入】

```
5
1 2 5 7 7 9 12 13 14 14
4
1 2 4 5
```

【样例输出】

```
1 1 5 2 5
```

【样例解释】

```
1=1                    5=5
2=1+1                  7=2+5
7=1+1+5                12=5+2+5
```

```
9=1+1+5+2            13=1+5+2+5
14=1+1+5+2+5           14=1+1+5+2+5
```

【数据规模】

```
n≤1000,s∈{1,2…500}
```

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;
const int size=100;
int n,m,total;
int a[2*size+1],s[size+1],ans[size+1];
//left right 需要安置的新数组的左边界 有边界
// now 需要将排序后的第 now 个元素进行安置
// pre 上次放在左侧的可行的元素值（元素值属于数组）
// post 上次放在右侧的可行的元素值（元素值属于数组）
bool dfs(int left,int right,int now,int pre,int post)
{
    int x;
    if(left==right)
    {
        x=total-pre-post;
        if(x>=1 && x<=500 && s[x])
        {
            ans[left]=x;
            return 1;
        }
        return 0;
    }
    x=a[now]-pre;
    if(x>=1 && x<=500 && s[x])
    {
        ans[left]=x;
        if(dfs(left+1,right,now+1,a[now],post))
        {
            return 1;
        }
    }
    x=a[now]-post;
    if(x>=1 && x<=500 && s[x])
    {
        ans[right]=x;
        if(dfs(left,right-1,now+1,pre,a[now]))
        {
            return 1;
        }
    }
}
int main()
{
    cin>>n;
    for(int i=1;i<=2*n;i++)
    {
```

```
        cin>>a[i];
    }
    cin>>m;
    int t;
    for(int i=1;i<=m;i++)
    {
        cin>>t;
        s[t]=1;
    }
    sort(a+1,a+1+2*n);
    total=a[2*n];
    dfs(1,n,1,0,0);
    for(int i=1;i<=n;i++)
    {
        cout<<ans[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

2.4. 生日蛋糕

标签：上下界剪枝，优化搜索顺序，可行性剪枝，最优性剪枝

【问题描述】

小明要制作一个体积为Nπ的M层生日蛋糕，每层都是一个圆柱体。设从下往上数第i(1≤i≤M)层蛋糕是半径为Ri，高度为Hi的圆柱。当i＜M时，要求Ri＞Ri+1且Hi＞Hi+1。由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积Q最小。

令Q = Sπ，请编程对给出的N和M，找出蛋糕的制作方案（适当的Ri和Hi的值），使S最小。（除Q外，以上所有数据皆为正整数）

【输入格式】

第一行为N（N≤10000），表示待制作的蛋糕的体积为Nπ

第二行为M（M≤20），表示蛋糕的层数为M。

【输出格式】

输出仅一行，是一个整数S（若无解则S=0）。

【样例输入】

```
100
2
```

【样例输出】

```
68
```

附：圆柱公式：体积V=π*R^2*H；侧面积A′=2*π*R*H；底面积A=π*R^2

【算法分析】

解题思路：要求出最少的表面积的和，肯定是一直枚举半径和高度，然后算出最小的表面积，这样的话肯定是深搜，这个很容易想到。但是题意中体积最大是10000， 要枚举的话肯定会用时会特别多。应该加上一点剪枝。这个题必须用好几种剪枝才能过。

剪枝一：如果在搜索的过程中体积小于0 或者当前半径或高度小于剩余要制作的层数（因为每层必须是相差1的，当前半径或高度小于要制作的层数，肯定不能制作完）就要剪枝。

剪枝二：当前剩下m层需要制作，可以预测制作这m层蛋糕最少需要多少体积，如果这个最少的体积还大于当前剩余的体

积，那个这个不必继续搜索，所以要剪枝。

剪枝三：因为要求最小表面积，如果当前枚举到的面积已经大于要求得的最优表面积，就要剪枝，但是这个剪枝还可以更优化，根据剪枝二的启发，如果当前枚举到的面积加上制作剩余 m 层蛋糕所要花费的最小表面积大于已经求得的最优解，那么剪枝。

剪枝四：因为体积必须为 N，根据剪枝二，如果剩下 m 层蛋糕制作都按照最大的来，依然不能够使用完剩余的体积那么也要减掉。根据上面四个剪枝，基本就能过这个题目。

【参考代码】

```
#include <iostream>
#include <cmath>
using namespace std;
int n,m,minv[21],mins[21];
//V=n*pi  m 层数 自顶向下 1.2.3...m
//minv[i]表示 i 层到 0 层加起来的最小总体积 minvs 最小表面积
const int inf=1000000000; //inf 足够大就可以 int(32)  -2^31~2^31-1=2147483647
int best=inf;                //best 最小表面积
//深度优先搜索 自底 m 向上搜索 r h 表示当前层得半径和高度
//sumv 已经用的总体积 sums 已经生成的总表面积
void dfs(int depth,int sumv,int sums,int r,int h)
{
    if(0==depth)
    {
        //搜索完成 更新最小表面积 best
        if(sumv==n&&sums<best)
        {
            best=sums;
        }
        return;
    }
    // 三个剪枝条件：
    //1、已经搜索过的体积加上还未搜索过的最小体积不能比总体积 n 大
    //2、已经搜索过的表面积加上还未搜索过的最小表面积不能比之前的最小总表面积 best 大
    //3、n-sumv 既所剩体积记作 dv 还需要的表面积为 s
    //s=2*ri*hi+2*r(i-1)*h(i-1)+..>=2*ri*hi*ri/r+2*r(i-1)*h(i-1)*r(i-1)/r+..
    //                             =2*dv/r（i 从 depth-1 取，r 为当前半径 ri/r<1）
    // 所以得到还需要的最小表面积 s=2*(n-sumv)/r，如果最小的 s 和已经搜索过的表面积 sums 依然比 best 大 就不用继续搜索了
    //剪枝如上所述
    if(sumv+minv[depth-1]>n||sums+mins[depth-1]>best||sums+2*(n-sumv)/r>=best)
        return;
    //递减顺序枚举 depth 层半径的每一个可能值，这里第 depth 层的半径最小值为 depth
    for(int i=r-1;i>=depth;i--)
    {
        if(depth==m)
            sums=i*i; //俯视蛋糕底面积作为外表面积的初始值(总的上表面积，以后只需计算侧面积)
        int maxh=min((n-sumv-minv[depth-1])/(i*i),h-1);
        //maxh 最大高度，即 depth 层蛋糕高度的上限，(n-sumv-minv[dep-1])表示第 depth 层最大的体积
        //同理，第 depth 层的最小高度值为 depth
        for(int j=maxh;j>=depth;j--)
        {
```

```
            //递归搜索子状态
            dfs(depth-1,sumv+i*i*j,sums+2*i*j,i,j);
        }
    }
}
int main()
{
    cin>>n;
    cin>>m;
    //rmax 初始半径 底层半径 最大值为 sqrt(n)
    int rmax=(int)sqrt((double)n);
    //hmax 初始高度 高度最大为 n
    int hmax=n;
    minv[0]=mins[0]=0;
    //初始化 minv 和 mins 数组
    //从顶层(即第 1 层)到第 i 层的最小体积 minv[i]成立时第 j 层的半径和高度都为 j
    for(int i=1;i<=m;i++)
    {
        minv[i]=minv[i-1]+i*i*i;
        mins[i]=mins[i-1]+2*i*i;
    }
    dfs(m,0,0,rmax,hmax);
    //dfs(m,0,0,n+1,n+1);
    if(best==inf)
        best=0;       //无解
    if(best==0)
        cout<<"No solution"<<endl;
    else
        cout<<"The min Q ="<<best<<"*pi"<<endl;
    return 0;
}
```