动态规划

by 202208040412 邹林壮

基本要素：最优子结构性质；**重叠子问题性质**

基本思想：保存已解决的子问题的解，避免重复计算

动态规划案例复习

案例一最大字段和

问题定义

输入：数组长度n，数组元素a_i

输出：最大字段和v，最大字段范围

约束条件：字段连续

最优子结构证明

假设子段{$a_s, a_{s+1}, ... a_i$}是以a_i为结尾的最大子段，定义b[i]=max{$a_s + a_{s+1} + ... + a_i$}

那必有{$a_s, a_{s+1}, ... a_{i-1}$}是以$a_{i-1}$为结尾的最大子段，即b[i-1]=max{$a_s + a_{s+1} + ... + a_{i-1}$}

反证法证明b[i-1]是问题的最优子结构，假设b[i-1]不是问题的最优子结构，则存在b'[i-1]>b[i-1],假设取到a[i],那么b'[i]=b'[i-1]+a[i],b[i]=b[i-1]+a[i],则b'[i]>b[i],与假设前提b[i]是以i结尾的最大子段和相矛盾

递推式

$b[i] = max\{b[i-1] + a[i], a[i]\}$,$0 < i \leq n$

案例二最大公共子序列

问题定义

输入：两个序列a[],b[]

输出：两个序列的最大公共子序列的长度

约束条件：最大

最优子结构证明

假设X={x1,x2,..xm},Y={y1,y2,..yn},Z={z1,z2,...,zk}

(1)若xm=yn,则zk=xm=yn,且Zk-1是Xm-1和Yn-1的最长公共子序列

(2)若xm≠yn且zk≠xm，则Z是Xm-1和Y的最长公共子序列。

(3)若xm≠yn且zk≠yn，则Z是X和Yn-1的最长公共子序列。

递推式

$$c[i][j]=\begin{cases}0 & i=0,y=0\\ maxc[i-1][j],c[i][j-1] & i>0,j>0,x[i]\neq y[j]\\ c[i-1][j-1]+1 & i>0,j>0,x[i]=y[j]\end{cases}$$

案例三01背包

问题定义

输入：背包的容量c，物品数量n，各物品的质量wi,价值vi

输出:背包存储的最大价值maxv

约束条件：物品必须完全取出，最大价值

最优子结构证明

假设$b[i][j]$，即(x1,x2,..,xi),xi∈{0，1}，是背包容量为j，选择前i件物品的最大价值，那么$b[i-1][j-w[i]]$，即(x1,x2,..,x_{i-1}),xi∈{0，1}是背包容量为j-w[i],选择前i-1件物品的最大价值，即$b[i-1][j-w[i]]$是问题的最优子结构，如果这不是问题的最优解，即存在$b'[i-1][j-w[i]]$>$b[i-1][j-w[i]]$,那么$b'[i-1][j-w[i]]+w[i]$>$b[i-1][j-w[i]]$+w[i],与问题的假设，$b[i][j]$是问题的最优解矛盾，因此$b[i-1][j-w[i]]$是问题的最优子结构

递推式

$$b[i][j]=\begin{cases}max\{b[i-1][j-w[i]]+v[i],b[i-1][j]\} & j\geq w_i\\ b[i-1][j] & 0\leq j<w_i\end{cases}$$

代码

```
#include<iostream>
#include<cmath>
using namespace std;
#define N 10001

int c,n;//c:capacity n:number
int v[N],w[N];//value weight
int m[N][N];//maxvalue
int x[N];//choice

void _01bag(int c,int *v,int *w,int n,int m[N][N]){
    for(int j=0;j<=c;j++)m[0][j]=0;
    for(int i=1;i<=n;i++)
        for(int j=0;j<=c;j++){

            if(j>=w[i])m[i][j]=max(m[i-1][j],m[i-1][j-w[i]]+v[i]);
            else m[i][j]=m[i-1][j];
        }
}

void _01bagchoice(int c,int *w,int n,int m[N][N],int* x){
    for(int i=n;i>0;i--){
        if(m[i][c]==m[i-1][c])x[i]=0;
        else{
            c-=w[i];
```

```
            x[i]=1;
        }
    }
}

//void _01bag(int c, int *v, int *w, int n) {
//    for (int j = 0; j <= c; j++) m[j] = 0;
//    for (int i = 1; i <= n; i++) {
//        for (int j = c; j >= w[i]; j--) {
//            m[j] = max(m[j], m[j - w[i]] + v[i]);
//        }
//    }
//}
//
//void _01bagchoice(int c, int *w, int n) {
//    int current_c = c;
//    for (int i = n; i > 0; i--) {
//        if (m[current_c] == m[current_c - w[i]]) x[i] = 0;
//        else {
//            x[i] = 1;
//            current_c -= w[i];
//        }
//    }
//}

int main(){
    cin>>n;
    cin>>c;
    for(int i=1;i<=n;i++)cin>>w[i]>>v[i];
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)m[i][j]=0;
    _01bag(c,v,w,n,m);
    cout<<m[n][c];
    _01bagchoice(c,w,n,m,x);
    for(int i=1;i<=n;i++)cout<<"第"<<i<<"件物品选用："<<x[i]<<endl;
}
```

案例四矩阵连乘

问题定义

矩阵乘法的代价/复杂性: 乘法的次数

若*A*是*p*×*q*矩阵，*B*是*q*×*r*矩阵，则*A*×*B*的代价是*O(pqr)*

输入：矩阵数量n，$< A_1, A_2, .., A_n >$

输出：$A_1 * A_2 * ... * A_n$的最小代价方法

约束条件：A_i与A_{i+1}是可乘的

最优子结构证明

假设问题的最优解为$A_{1-n}=A_{1-k} \times A_{k+1-n}$,那么$A_{1-k}$和$A_{k+1-n}$必须是其问题的最优解

若子问题A_{1-k}的解a不是问题的最优解，那么存在更有的解a',那么a'×A_{k+1-n}>a*A_{k+1-n},与假设相矛盾，因此A_{1-k}是问题的最优子结构

递推式

$$m[i][j]=\begin{cases}0, & \text{if } i=j, \\ \max\limits_{i \le k<j}(m[i][k]+m[k+1][j])+p_{i-1} \cdot p_k \cdot p_j, & \text{if } i<j.\end{cases}$$

案例五凸多边形最优三角剖分

与案例四同构

凸多边形：用多边形顶点的逆时针序列表示凸多边形，即P={v_0，v_1，...，v_{n-1}}表示具有n条边的凸多边形。

弦：若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦。弦将多边形分割成2个多边形{v_i，v_{i+1}，...，v_j}和{v_j，v_{j+1}，...，v_i}

凸多边形最优三角剖分：给定凸多边形P，以及定义在由多边形的边和弦组成的三角形上的权函数w。确定该凸多边形的三角剖分，使得该三角剖分中诸三角形上权之和为最小。

递归结构

定义$t[i][j]$，1≤i＜j≤n为凸子多边形{v_{i-1}，v_i，...，v_j}的最优三角剖分所对应的权函数值，即其最优值。

$t[i][j]=t[i][k]+t[k+1][j]+$(△$v_{i-1}v_kv_j$权值)

递归定义

时间复杂性：$O(n^3)$ 空间复杂性：$O(n^2)$

针对求具有单调性子序列问题

1.求n个元素组成的序列的最长单调递增子序列

$O(n^2)$

容易得到时间复杂度为$O(n^2)$的方法

设$b[i]$是以$a[i]$为结尾元素的最长单调递增子序列的长度，则序列a的最长单调递增子序列的长度为

$$\max_{0<=k<=i a[k]<=a[i]} b[k]+1$$

```
ll lengthOfLIS(vector<ll> a){
    int n=a.size();
    vector<ll> b(n);
    ll max=0;
    b[0]=1;
    for(ll i=0;i<n;i++){
        ll k=0;
        for(ll j=0;j<i;j++){
            if(b[j]>k&&a[i]>a[j])k=b[j];
        }
        b[i]=k+1;
        if(b[i]>max)max=b[i];
    }

    return max;
}
```

使用两层循环进行状态的转移，得到每一个状态的dp，并且记录最大的dp作为最初长单调递增子序列的值

$O(nlogn)$

通过记录最小结尾元素值来优化

对该算法进行优化，容易看出$i-1$到i的循环中，$a[i]$的值起关键作用。如果$a[i]$能够扩展到序列$a[0:i-1]$的最长递增子序列的长度，则k=k+1,否则k不变。设$a[0:i-1]$中长度为**k**的最长可以扩展递增子序列的结尾元素是$a[j](0<=j<i-1)$，则当$a[i]>=a[j]$时可以扩展，否则不可以扩展。

如果存在多个长度为k的递增子序列,只需要递增子序列中结尾元素的最小值$b[k]$，因此将$b[k]$作为序列$a[0:i-1]$中所有长度为k的递增子序列中的最小结尾元素值。

增强假设后，在i-1到i的循环中，当$a[i]>=b[k]$时，$k=k+1,b[k]=a[i]$,否则k值不变；当$a[i]<b[k]$时，如果$a[i]<b[1]$,则应该将$b[1]$的值更新为$a[i]$,如果$b[1]<=a[i]<=b[k]$,则二分搜索查找下标j,使得$b[j-1]<=a[i]<b[j]$,此时$b[1:j-1]$和$b[j+1:k]$的值不变，$b[j]$的值更改为$a[i]$

```
ll lengthOfLIS(int a[]){
    b[1]=a[0];
    ll k=1;
    for(ll i=1;i<n;i++){
        if(a[i]>b[k]){
            b[++k]=a[i];
        }else{
            b[std::lower_bound(b,b+k,a[i])-b]=a[i];
/*          //二分查找算法，用于在已排序的范围内查找第一个不小于给定值的元素
            ForwardIterator lower_bound(ForwardIterator first,
                          ForwardIterator last,
                          const T& value);
            返回的是一个地址，-b得到该元素所在位置
*/
        }
    }
    return k;
}
```

通过耐心排序进行优化

使用耐心排序，并证明最小堆数=LIS长度

耐心排序可以找到耐心游戏的最小堆数，证明**耐心排序的最小堆数 = LIS 的长度**

引理 1：最小堆数 >= LIS 长度

证明：假设我们有一个最长递增子序列：$c1 < c2 < ... < cn$。 如果我们知道牌 c(i) 的位置，那么牌 c(i+1) 在哪里？ 首先，c(i+1) 不可能和 c(i) 放在同一堆中，因为 c(i) 下面所有的牌都必须小于 c(i) 本身（游戏规则的设定）。 其次，c(i+1) 不能放在 c(i) 左边的一堆上，否则 c(i) 会放在那堆的上面（耐心排序原理）。 因此，我们知道牌 c(i+1) 一定位于 c(i) 右侧的某个堆中，这意味着LIS的长度至多就是耐心排序的最小堆数。

引理 2：最小堆数 <= LIS 的长度 证明：再次假设我们有一个 LIS：$c1 < c2 < ... < cn$。 首先我们考虑一张牌，c(i)。这张牌必须大于 c(i) 左侧牌堆上的顶牌 c(i-1)，否则 c(i) 将被放入那堆的上面。然后我们来考虑 c(i-1)，这张牌必须大于 c(i-1) 左侧的牌堆顶牌。因此我们可以看到，每堆中必定有一张牌可以连起来并形成递增顺序（但不一定是最长的，这就是为什么最小堆数最多是 LIS 的长度)。

因此：最小堆数 = LIS 的长度 通过以上两个引理，我们知道最小桩数必须等于 LIS 的长度才能同时满足两个引理。因此，最小堆数 = LIS 的长度。

```
int lengthOfLIS(vector<ll> a){
    int size=0;
    vector<ll> q(a.size()+1);
    int i,j;
    for(int x:a){
        i=0,j=size;
        while(i<j){
            int m=i+(j-i)/2;
            if(q[m]<x)i=m+1;
            else j=m;
        }
        q[i]=x;
        size=max(i+1,size);
    }
    return size;
}
```

2.

引理：Dilworth 定理

> 狄尔沃斯定理亦称偏序集分解定理，该定理断言：对于任意有限偏序集，其最大反链中元素的数目必等于最小链划分中链的数目。此定理的对偶形式亦真，它断言：对于任意有限偏序集，其最长链中元素的数目必等于其最小反链划分中反链的数目。

该定理在该问题上可以理解成：把**序列分成不上升子序列的最少个数**，等于序列的最长上升子序列长度。把序列分成不降子序列的最少个数，等于序列的最长下降子序列长度。