# DP模板

## 1.背包

## 2.LIS

## 3.LCS

## 4.区间DP

## 5.数位DP

## 6.状态压缩DP

7.概率DP

8.DP优化

## 背包

1. 01背包

每种物品有且每种物品有且只有一个，并且有权值和体积两个属性只有一个，并且有权值和体积两个属性。

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

for(int j=V ; j>=w[i] ; j--)

dp[j]=max(dp[j-w[i]]+val[i],dp[j]);

1. 完全背包

有N种物品和一个容量为V的背包，每种物品都有无限件可用。

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

for(int j>=w[i] ; j<=V ; j++)

dp[j]=max(dp[j-w[i]]+val[i],dp[j]);

1. 多重背包

有N种物品和一个容量为V的背包。第i种物品最多有num[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

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

for(i = 0 ; i < m ; i++)

{

scanf("%d%d%d",&pri[i],&w[i],&num[i]);

}

for(i = 0 ; i < m ; i++)

{

for(k = 0 ; k < num[i] ; k++)

{

for(j = n ; j >= pri[i]; j--)

{

f[j] = max(f[j],f[j-pri[i]]+w[i]);

}

}

}

printf f[n]

# 二维费用的背包问题

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2，第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值（两种背包容量）分别为V和U。

LL KnapSack(int n,int V,int M) {

for(int i=0; i<n; i++)

for(int j=w[i]; j<V; j++)

for(int k=m[i]; k<=M; k++)

dp[j][k]=max(dp[j][k],dp[j-v[i]][k-m[i]]+w[i]);

return dp[V][M];

最多只能取M件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为1，可以付出的最大件数费用为M。换句话说，设f[v][m]表示付出费用v、最多选m件时可得到的最大价值

5.分组背包问题

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

LL KnapSack(int n,int W,int K)

for(int k=1; k<=K; k++) //K组

for(int j=W; j>=0; j--) //01背包

for (int i=n/K\*(k-1)+1; i<n/K\*k+1; i++)

dp[j]=max( dp[j], dp[ j-v[i] ]+w[i] );

return dp[W];

**背包记录路径**;

for (int i = N;i >= 1;--i)

{

for (int j = V;j >= c[i];--j)

{

if (dp[j] < dp[j-c[i]]+w[i])

{

dp[j] = dp[j-c[i]]+w[i];

path[i][j] = 1;

}

}

}

for (int i = 1,j = V;i <= N&&j > 0;i++)

{

if (path[i][j])

{

printf("%d ",i);

j -= c[i];

}

}

LIS

最长上升子序列(nlogn)

int Longest\_NonDecreasing\_Subsequence(int a[],int n){

int b[n+1];

memset(b,0,sizeof(b));

int maxlen=0;

for (int i=1; i<=n; i++) //读入

if (a[i]>b[maxlen]) b[++maxlen]=a[i];

else {

int pos=lower\_bound(b+1,b+maxlen+1,a[i])-b;

b[pos]=a[i];

}

return maxlen;

LCS

两个有序序列a和b，求他们公共子序列的最大长度

O(n\*m)

if(s1[i]==s2[j])

d[i][j]=d[i-1][j-1]+1

else d[i][j]=max(d[i-1][j],d[i][j-1]).

O(nlogn)

先顺序扫描a串，取其在b串的所有位置。将a串每个字母用其反序列替代，则最终的最长严格递增序列的长度即为解。

数位DP

1. 如果题目中出现求满足区间[l,r]的符合......性质的数的个数，考虑使用数位dp.
2. 2.思考一下：如果我们只能从前往后一位位枚举当前的数位，要做出这道题，我们需要知道哪些量？利用这些来补充到dfs的调用参数中.

  3.套用模板.

控制上界枚举，从最高位开始往下枚举，例如：ri=213，那么我们从百位开始枚举：百位可能的情况有0,1,2(觉得这里枚举0有问题的继续看)

然后每一位枚举都不能让枚举的这个数超过上界213（下界就是0或者1，这个次要），当百位枚举了1，那么十位枚举就是从0到9，因为百位1已经比上界2小了，后面数位枚举什么都不可能超过上界。所以问题就在于：当高位枚举刚好达到上界是，那么紧接着的一位枚举就有上界限制了。具体的这里如果百位枚举了2，那么十位的枚举情况就是0到1，如果前两位枚举了21，最后一位之是0到3(这一点正好对于代码模板里的一个变量limit 专门用来判断枚举范围)。最后一个问题：最高位枚举0：百位枚举0，相当于此时我枚举的这个数最多是两位数，如果十位继续枚举0，那么我枚举的就是以为数咯，因为我们要枚举的是小于等于ri的所以数，当然不能少了位数比ri小的咯！(这样枚举是为了无遗漏的枚举，不过可能会带来一个问题，就是前导零的问题，模板里用lead变量表示，不过这个不是每个题目都是会有影响的，可能前导零不会影响我们计数，具体要看题目)

由于这种新的枚举只控制了上界所以我们的Main函数总是这样：

int main()

{

long long le,ri;

while(~scanf("%lld%lld",&le,&ri))

printf("%lld\n",solve(ri)-solve(le-1));

}

统计[1,ri]数量和[1,le-1]，然后相减就是区间[le,ri]的数量了，这里我写的下界是1，其实0也行，反正相减后就没了，注意题目中le的范围都是大于等于1的(不然le=0,再减一就G\_G了)

在讲例题之前先讲个基本的动态模板(先看后面的例题也行)：dp思想，枚举到当前位置pos，状态为state(这个就是根据题目来的，可能很多，毕竟dp千变万化)的数量(既然是计数,dp值显然是保存满足条件数的个数)

模板：

typedef long long ll;

int a[20];

ll dp[20][state];//不同题目状态不同

ll dfs(int pos,/\*state变量\*/,bool lead/\*前导零\*/,bool limit/\*数位上界变量\*/)//不是每个题都要判断前导零

{

//递归边界，既然是按位枚举，最低位是0，那么pos==-1说明这个数我枚举完了

if(pos==-1) return 1;/\*这里一般返回1，表示你枚举的这个数是合法的，那么这里就需要你在枚举时必须每一位都要满足题目条件，也就是说当前枚举到pos位，一定要保证前面已经枚举的数位是合法的。不过具体题目不同或者写法不同的话不一定要返回1 \*/

//第二个就是记忆化(在此前可能不同题目还能有一些剪枝)

if(!limit && !lead && dp[pos][state]!=-1) return dp[pos][state];

/\*常规写法都是在没有限制的条件记忆化，这里与下面记录状态是对应，具体为什么是有条件的记忆化后面会讲\*/

int up=limit?a[pos]:9;//根据limit判断枚举的上界up;这个的例子前面用213讲过了

ll ans=0;

//开始计数

for(int i=0;i<=up;i++)//枚举，然后把不同情况的个数加到ans就可以了

{

if() ...

else if()...

ans+=dfs(pos-1,/\*状态转移\*/,lead && i==0,limit && i==a[pos]) //最后两个变量传参都是这样写的

/\*这里还算比较灵活，不过做几个题就觉得这里也是套路了

大概就是说，我当前数位枚举的数是i，然后根据题目的约束条件分类讨论

去计算不同情况下的个数，还有要根据state变量来保证i的合法性，比如题目

要求数位上不能有62连续出现,那么就是state就是要保存前一位pre,然后分类，

前一位如果是6那么这意味就不能是2，这里一定要保存枚举的这个数是合法\*/

}

//计算完，记录状态

if(!limit && !lead) dp[pos][state]=ans;

/\*这里对应上面的记忆化，在一定条件下时记录，保证一致性，当然如果约束条件不需要考虑lead，这里就是lead就完全不用考虑了\*/

return ans;

}

ll solve(ll x)

{

int pos=0;

while(x)//把数位都分解出来

{

a[pos++]=x%10;//个人老是喜欢编号为[0,pos),看不惯的就按自己习惯来，反正注意数位边界就行

x/=10;

}

return dfs(pos-1/\*从最高位开始枚举\*/,/\*一系列状态 \*/,true,true);//刚开始最高位都是有限制并且有前导零的，显然比最高位还要高的一位视为0嘛

}

int main()

{

ll le,ri;

while(~scanf("%lld%lld",&le,&ri))

{

//初始化dp数组为-1,这里还有更加优美的优化,后面讲

printf("%lld\n",solve(ri)-solve(le-1));

}

例题：数位上不能有4也不能有连续的62，没有4的话在枚举的时候判断一下，不枚举4就可以保证状态合法了，所以这个约束没有记忆化的必要，而对于62的话，涉及到两位，当前一位是6或者不是6这两种不同情况我计数是不相同的，所以要用状态来记录不同的方案数。

dp[pos][sta]表示当前第pos位，前一位是否是6的状态，这里sta只需要去0和1两种状态就可以了，不是6的情况可视为同种，不会影响计数。

#include<iostream>

#include<cstdio>

#include<cstring>

#include<string>

using namespace std;

typedef long long ll;

int a[20];

int dp[20][2];

int dfs(int pos,int pre,int sta,bool limit)

{

if(pos==-1) return 1;

if(!limit && dp[pos][sta]!=-1) return dp[pos][sta];

int up=limit ? a[pos] : 9;

int tmp=0;

for(int i=0;i<=up;i++)

{

if(pre==6 && i==2)continue;

if(i==4) continue;//都是保证枚举合法性

tmp+=dfs(pos-1,i,i==6,limit && i==a[pos]);

}

if(!limit) dp[pos][sta]=tmp;

return tmp;

}

int solve(int x)

{

int pos=0;

while(x)

{

a[pos++]=x%10;

x/=10;

}

return dfs(pos-1,-1,0,true);

}

int main()

{

int le,ri;

//memset(dp,-1,sizeof dp);可优化

while(~scanf("%d%d",&le,&ri) && le+ri)

{

memset(dp,-1,sizeof dp);

printf("%d\n",solve(ri)-solve(le-1));

}

return 0;

}

例题：HDU 4734

题目给了个f(x)的定义：F(x) = An \* 2n-1 + An-1 \* 2n-2 + ... + A2 \* 2 + A1 \* 1，Ai是十进制数位，然后给出a,b求区间[0,b]内满足f(i)<=f(a)的i的个数。

常规想：这个f(x)计算就和数位计算是一样的，就是加了权值，所以dp[pos][sum]，这状态是基本的。a是题目给定的，f(a)是变化的不过f(a)最大好像是4600的样子。如果要memset优化就要加一维存f(a)的不同取值，那就是dp[10][4600][4600]，这显然不合法。

这个时候就要用减法了，dp[pos][sum]，sum不是存当前枚举的数的前缀和(加权的)，而是枚举到当前pos位，后面还需要凑sum的权值和的个数，

也就是说初始的是时候sum是f(a),枚举一位就减去这一位在计算f(i)的权值，那么最后枚举完所有位 sum>=0时就是满足的，后面的位数凑足sum位就可以了。

仔细想想这个状态是与f(a)无关的(新手似乎很难理解)，一个状态只有在sum>=0时才满足，如果我们按常规的思想求f(i)的话，那么最后sum>=f(a)才是满足的条件。

#include<cstdio>

#include<cstring>

#include<iostream>

#include<string>

using namespace std;

const int N=1e4+5;

int dp[12][N];

int f(int x)

{

if(x==0) return 0;

int ans=f(x/10);

return ans\*2+(x%10);

}

int all;

int a[12];

int dfs(int pos,int sum,bool limit)

{

if(pos==-1) {return sum<=all;}

if(sum>all) return 0;

if(!limit && dp[pos][all-sum]!=-1) return dp[pos][all-sum];

int up=limit ? a[pos] : 9;

int ans=0;

for(int i=0;i<=up;i++)

{

ans+=dfs(pos-1,sum+i\*(1<<pos),limit && i==a[pos]);

}

if(!limit) dp[pos][all-sum]=ans;

return ans;

}

int solve(int x)

{

int pos=0;

while(x)

{

a[pos++]=x%10;

x/=10;

}

return dfs(pos-1,0,true);

}

int main()

{

int a,ri;

int T\_T;

int kase=1;

scanf("%d",&T\_T);

memset(dp,-1,sizeof dp);

while(T\_T--)

{

scanf("%d%d",&a,&ri);

all=f(a);

printf("Case #%d: %d\n",kase++,solve(ri));

}

return 0;

}

区间DP

区间动态规划问题一般都是考虑，对于每段区间，他们的最优值都是由几段更小区间的最优值得到，是分治思想的一种应用，将一个区间问题不断划分为更小的区间直至一个元素组成的区间，枚举他们的组合 ，求合并后的最优值。

设F[i,j]（1<=i<=j<=n）表示区间[i,j]内的数字相加的最小代价

最小区间F[i,i]=0（一个数字无法合并，∴代价为0）

每次用变量k（i<=k<=j-1）将区间分为[i,k]和[k+1,j]两段

For l:=1 to n do // l是区间长度，作为阶段。

for i:=1 to n do // i是穷举的区间的起点

begin

j:=i+l-1; // j是 区间的终点，这样所有的区间就穷举完毕

if j>n then break; // 这个if很关键。

for k:= i to j-1 do // 状态转移，去推出 f[i,j]

f[i , j]= max{f[ i,k]+ f[k+1,j]+ w[i,j] }

模板：

int main()

{

for(int i=0;i<n;i++)

dp[i][i]=........

for(int len=1;len<=n;len++)//区间长度

{

for(int i=0;i<n;i++)//区间起点

{

j=len-1+i;//区间终点

for(int k=i;k<j;k++)//k把[i,j]分为[i,k],[k+1,j]

{

dp[i][j]=........

}

}

概率DP

概率dp主要解决的是关于概率问题和期望问题的求解。难点和普通dp一样在于dp[i][j][k]的维数控制和含义，其实就是转移方程的构建。然后一般地，求概率是正推、求期望是逆推。（开始的很多状态不可能发生概率为0，最后的状态出口期望为0）

对于求概率当前点的概率是由前面能到达当前点的点乘上到达当前点的概率得到的。也就是dp[i]=Σ(dp[j]\*p[j][i]) i是当前点、j是前面的点。

对于求期望当前点的期望是由当前点所能到达的点得到的。（注意下，对应的概率是当前点到达之后点的概率，因为你要到达后面的点后面的点才能传递给你期望！）

也就是E[i]=Σ((E[j]+k)\*p[i][j]) i是当前点、j是当前点所能到达的点，k是所需的期望值。

然后就是对于期望的话，如果成环的话数据范围小的话可以用高斯消元解决，如果范围大就要推导公式了

例题： 一个软件有s个子系统，会产生n种bug

某人一天发现一个bug,这个bug属于一个子系统，属于一个分类

每个bug属于某个子系统的概率是1/s,属于某种分类的概率是1/n

问发现n种bug,每个子系统都发现bug的天数的期望。

求解：

dp[i][j]表示已经找到i种bug,j个系统的bug，达到目标状态的天数的期dp[n][s]=0;要求的答案是dp[0][0];

dp[i][j]可以转化成以下四种状态:

dp[i][j],发现一个bug属于已经有的i个分类和j个系统。概率为(i/n)\*(j/s);

dp[i][j+1],发现一个bug属于已有的分类，不属于已有的系统.概率为 (i/n)\*(1-j/s);

dp[i+1][j],发现一个bug属于已有的系统，不属于已有的分类,概率为 (1-i/n)\*(j/s)；

dp[i+1][j+1],发现一个bug不属于已有的系统，不属于已有的分类,概率为 (1-i/n)\*(1-j/s);

整理便得到转移方程

例题：一只吸血鬼，有n条路给他走，每次他随机走一条路，

每条路有个限制，如果当时这个吸血鬼的攻击力大于等于某个值，那么就会花费t天逃出去。否则，花费1天的时间，并且攻击力增加，问他逃出去的期望天数。

思路：

方程很好像 dp[i] 有i点战斗力逃出去的期望

那么如果 x>c[i]时 dp[i]=(1.0/n)\*Σt[i]

否则 dp[i]=(1.0/n)\*Σ(dp[i+c[i]]+1)

因为很多点会被重复用，所以写成记忆化搜索。

还有一个要注意的，t[i]是向下取整的

using namespace std;

#define N 10020

double dp[N];

int c[123];

int t[123];

int MAX;

int n,f;

double dfs(int x)

{

if(x>MAX) x=MAX; //小优化 超过最大值 就为最大值

if(fabs(dp[x]+1)>eps) return dp[x];

double tep=0.0;

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

{

if(x>c[i]) tep+=1.0/n\*t[i];

else tep+=1.0/n\*(dfs(x+c[i])+1);

}

return dp[x]=tep;

}

int main()

{

while(cin>>n>>f)

{

int i;

MAX=-1;

memset(dp,-1,sizeof(dp));

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

{

scanf("%d",&c[i]);

MAX=max(MAX,c[i]+1);

t[i]=(int)((1+sqrt(5.0))/2\*c[i]\*c[i]); //这里要注意了 t[i]是一个向下取整的！

}

dp[f]=dfs(f);

printf("%.3f\n",dp[f]);

}

return 0;

}

一般的DP是这样：在DP过程中，当前状态必然是由多个子状态中的最优的转移而来

所以一般的DP求的是最优的结果，而概率不需要最优，而是实际概率

所以概率DP最大的区别在于：在DP过程中，当前状态是由所有子状态的概率共同转移而来

所以概率DP只是利用了DP的动态而没有规划 （只有状态转移，而不需要进行决策）

状态压缩DP

1.’&’符号，x&y，会将两个十进制数在二进制下进行与运算，然后返回其十进制下的值。例如3(11)&2(10)=2(10)。

2.’|’符号，x|y，会将两个十进制数在二进制下进行或运算，然后返回其十进制下的值。例如3(11)|2(10)=3(11)。

3.’^’符号，x^y，会将两个十进制数在二进制下进行异或运算，然后返回其十进制下的值。例如3(11)^2(10)=1(01)。

4.’<<’符号，左移操作，x<<2，将x在二进制下的每一位向左移动两位，最右边用0填充，x<<2相当于让x乘以4。相应的，’>>’是右移操作，x>>1相当于给x/2，去掉x二进制下的最有一位。

这四种运算在状压dp中有着广泛的应用，常见的应用如下：

1.判断一个数字x二进制下第i位是不是等于1。

方法：if ( ( ( 1 << ( i - 1 ) ) & x ) > 0)

将1左移i-1位，相当于制造了一个只有第i位上是1，其他位上都是0的二进制数。然后与x做与运算，如果结果>0，说明x第i位上是1，反之则是0。

2.将一个数字x二进制下第i位更改成1。

方法：x = x | ( 1<<(i-1) )

3.把一个数字二进制下最靠右的第一个1去掉。

方法：x=x&(x-1)

【例1】有一个N\*M(N<=5,M<=1000)的棋盘，现在有1\*2及2\*1的小木块无数个，要盖满整个棋盘，有多少种方式？答案只需要mod1,000,000,007即可。

例如：对于一个2\*2的棋盘，有两种方法，一种是使用2个1\*2的，一种是使用2个2\*1的。

【算法分析】

在这道题目中,N和M的范围本应该是一样的，但实际上，N和M的范围却差别甚远，对于这种题目，首先应该想到的就是，正确算法与这两个范围有关！N的范围特别小，因此可以考虑使用状态压缩动态规划的思想

int N, M;

long long dp[1005][34];

void dfs(int i,int j,int state,int nex)

{

if (j==N)

{

dp[i+1][nex]+=dp[i][state];

return;

}

//如果这个位置已经被上一列所占用,直接跳过

if (((1<<j)&state)>0)

dfs(i,j+1,state,nex);

//如果这个位置是空的，尝试放一个1\*2的

if (((1<<j)&state)==0)

dfs(i,j+1,state,nex|(1<<j));

//如果这个位置以及下一个位置都是空的，尝试放一个2\*1的

if (j+1<N && ((1<<j)&state)==0 && ((1<<(j+1))&state)==0)

dfs(i,j+2,state,nex);

return;

}

int main()

{

while (cin>>N>>M)

{

memset(dp,0,sizeof(dp));

if (N==0 && M==0) break;

dp[1][0]=1;

for (int i=1;i<=M;i++)

{

for (int j=0;j<(1<<N);j++)

if (dp[i][j])

{

dfs(i,0,j,0);

}

}

cout<<dp[M+1][0]<<endl;

}

}

假设第一列已经填满，则第二列的摆设方式，只与第一列对第二列的影响有关。同理，第三列的摆设方式也只与第二列对它的影响有关。那么，使用一个长度为N的二进制数state来表示这个影响，例如：4(00100)就表示了图上第二列的状态。

因此，本题的状态可以这样表示：

dp[i][state]表示该填充第i列，第i-1列对它的影响是state的时候的方法数。i<=M,0<=state<2N

对于每一列，情况数也有很多，但由于N很小，所以可以采取搜索的办法去处理。对于每一列，搜索所有可能的放木块的情况，并记录它对下一列的影响，之后更新状态。状态转移方程如下：

dp[i][state]=∑dp[i-1][pre]每一个pre可以通过填放成为state

对于每一列的深度优先搜索

DOING HOMEWORK

char name[20][120];

int cost[20],dead\_time[20],dp[1<<16],time[1<<16],def[1<<16]; //time用来储存每个状态对应的当前时间，dp为最少扣分，def为对应上一门科目

void Cout(int num) { //输出最少状况下对应的科目顺序

if(num==0) return ;

int i=def[num];

Cout(num-(1<<i));

cout<<name[def[num]]<<'\n';

}

int main() {

std::ios::sync\_with\_stdio(false);

int T;

cin>>T;

while(T--) {

int n;

cin>>n;

for(int i=0; i<n; i++)

cin>>name[i]>>dead\_time[i]>>cost[i];

dp[0]=0;

for(int i=1; i<(1<<n); i++) { //每个状态

dp[i]=INF;

for(int j=n-1; j>=0; j--) { //先判断做没做这个科目，做了的话求不做它时的状态，进行对比，从后遍历

int temp=1<<j;

if((i&temp)==0) continue;

int now\_cost=time[i-temp]+cost[j]-dead\_time[j];

if(now\_cost<0) now\_cost=0; //扣分没有负的

if(dp[i]>dp[i-temp]+now\_cost) {

dp[i]=dp[i-temp]+now\_cost;

time[i]=time[i-temp]+cost[j];

def[i]=j;

}

}

}

cout<<dp[(1<<n)-1]<<'\n';

Cout((1<<n)-1);

}

return 0;

}

# 单调队列优化DP

形如 dp[i]=max/min (f[j]) + g[i] (j<i)

（ g[i]是与j无关的变量 或者 没有g[i]只有关于j的函数）

优化对象是f[j]

维护一个表示关于j的最优解的队列

1、 队首元素出队，直到队首元素在给定的范围中。

2、 此时，队首元素就是最优决策，

3、计算dp[i]，并将其插入到单调队列的尾部，同时维持队列的单调性（不断地出队，直到队列单调为止）。

单调队列的每个元素一般会存储两个值：

1.在原数列中的位置（下标）

2.该元素在动态规划中的状态值（价值）

用单调队列来优化多重背包问题

有n个物品，每个物品的价格是w，重量是c，且每个物品的数量是k，那么用这样的一些物品去填满一个容量为m的背包，使得得到的背包价值最大化

利用单调队列的优化，复杂度是O(mn)

对于第i件物品，如果已知体积为V，价值为W，数量为K，那么可以按照V的余数，将当前的体积J分成V组(0,1,....V-1)。

对于任意一组，可以得到转移方程：f[i\*V+c]=f[k\*V+c]+(i-k)\*W，其中c是V组分组中的任意一个

令f[i\*V+c]=dp[i]，那么就得到dp[i]=dp[k]+(i-k)\*W (k>=i-K)

将dp[k]-k\*W看做是优化函数，那么就可以运用单调队列来优化了

#include<iostream>

#include<string>

#include<algorithm>

using namespace std;

int c,w,num;

int a[101],b[101],f[101];

int n,m;

int q[101];

int head,tail;

int main()

{

int i,cas,k,j;

freopen("in.txt","r",stdin);

scanf("%d",&cas);

while(cas--)

{

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

memset(f,0,sizeof(f));

for(i=0;i<n;i++)

{

scanf("%d%d%d",&c,&w,&num);

if(m/c<num) //取最小值

num=m/c;

for(k=0;k<c;k++) //分成c份

{

head=tail=0;

for(j=0;j<=(m-k)/c;j++)

{

int x=j;

int y=f[j\*c+k]-j\*w; //将f[j\*c+k]-j\*w入队

while(head<tail && y>=b[tail-1])

tail--;

a[tail]=x; //记录下标

b[tail++]=y; //记录dp值

while(a[head]<j-num) //如果超出了范围

head++;

f[j\*c+k]=b[head]+j\*w;

}

}

}

printf("%d\n",f[m]);

}

return 0;

}

**斜率优化**

对于形如 dp[i]=min{dp[j]+f(i,j)} 的方程

令k < j < i，当我们更新dp[i]时，如果有dp[j] + f(i, j) 比

dp[k] + f(i, k)更优，则有dp[j] + f(i, j) - (dp[k] + f(i, k) < 0，

对于这个不等式如果能够化解成如下形式

（Y(j)−Y(k)）/（X(j)−X(k)）< f(i)

就能通过斜率优化这个dp

优化方法

若满足（Y(j)−Y(k)）/（X(i)−X(j)）< f(i)则j转移到i，比k转移到i更优，如果我们把(X(j), Y(j)), (X(k), Y(k))当成平面上的两个点Pj, Pk，这个不等式的含义即为若PjPk的斜率＜f(i)则，从j转移更优令grad(i, j)表示PiPj的斜率，现在我们假设grad(i,j) < grad(j, k)，若grad(i, j) < f(I),则i比j更优，若grad(i, j) > f(I), 则grad(j, k) > f(I),那么从k转移比从j转移更优，当grad(i, j) < grad(j, k)的时候，无论如何j转移到i都不会是最优。所以最后我们维护一个上凸集即可。

但是此时我们还是没有解决最终问题，如何才能找到转移到i点的最优的点呢。可以发现最后的点集一定是一个凸集，也就是斜率单调！这样对于k < j, grad(j,k) < f(i),时更优，从图形特点我们可以发现如果j比k优，那么j点比所有比k小的点都优，所以对于每一个f(i),我们维护一个所有比i点小的凸集，二分查找斜率比f(i)小的编号最大的点，就是最优的转移点。

当然，如果f(i)也满足单调性，还可以直接维护一个单调队列就能解决这个问题。

f[i]=min{f[j]+(s[i]-s[j])^2+M}

展开得

f[i]=min{f[j]+s[i]^2+s[j]^2-2\*s[i]\*s[j]+M}

令f[i]=B,f[j]+s[j]^2=y,2\*s[j]=x,k=s[i]

因此k\*x+B=y

k是s[i]，前缀和随着i增大而增大，因为求最小值，故维护下凸包。

代码

long long n,m,Q[500005],f[500005],s[500005];

double Slope(long long j,long long k) { //求斜率

return double((f[j]+s[j]\*s[j])-(f[k]+s[k]\*s[k]))/(2\*s[j]-2\*s[k]);

}

int main() {

while(scanf("%lld%lld",&n,&m)!=EOF) {

for(int i=1; i<=n; i++)s[i]=s[i-1]+Get\_Int();

int Left=1,Right=1;

Q[1]=0;

f[0]=0;

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

while(Left<Right&&Slope(Q[Left],Q[Left+1])<=s[i])Left++; //维护队首（删除非最优决策）

int Front=Q[Left];

f[i]=f[Front]+(s[i]-s[Front])\*(s[i]-s[Front])+m; //计算当前f

while(Left<Right&&Slope(Q[Right-1],Q[Right])>=Slope(Q[Right],i))Right--; //维护队尾（维护下凸包性质）

Q[++Right]=i; //入队

}

printf("%lld\n",f[n]);

}

return 0;

}

对于f(i)单调的这种情况，除了使用单调队列优化的斜率优化做，我们还有另外一种分治的做法，但是复杂度会变成O(nlogn) 比O(n)差。

当f(i)单调的时候，我们可以发现若a > b,则f(a) > f(b),设转移到a的最优点是c，转移到b的最优点是d，一定有c > d。也就是转移到a的最优点一定大于等于转移到b的最优点。

多重背包二进制优化

假设某个物品有10个数量，我们取数的方法是先取1个，再取2个，再取4个以此类推（i=1;i<=10;i<<=1）；

也就是说，会取到1、2、4个，然后最后还会剩下3个。观察前面3个数，它们任意相加又可以组成3、5、6、7，这些数再与最后深下的3相加又可以得到8、9、10。也就是说，1到10全有了，但是我们操作的次数却从10次降低到了4次！

for(i=0;i<m;i++) //m种物品

{

left=num[i]; //每个物品的数量

for(j=1;j<=left;j<<=1) //二进制取数

{

/\*do something by yourself\*/

left-=j;

}

if(left) //剩余的数

{

/\*do something by yourself\*/

}

}

例

cin>>n>>m;

for(i=0;i<m;i++)

{

cin>>p[i]>>w[i]>>b[i];

}

memset(f,0,(n+1)\*4);

for(i=0;i<m;i++)

{

left=b[i];

for(j=1;j<=left;j<<=1)

{

for(k=n;k>=j\*p[i];k--)

{

f[k]=max(f[k],f[k-j\*p[i]]+j\*w[i]);

}

left-=j;

}

if(left)

{

for(k=n;k>=left\*p[i];k--)

{

f[k]=max(f[k],f[k-left\*p[i]]+left\*w[i]);

}

}

}

cout<<f[n]<<endl;