

Blogggggg

模板总集



just continue.

# 图论

1、最短路问题

**SPFA（链式前向星版）**

//复杂度：最坏为O(|E|\*|V|), 但一般为O(k\*|E|) 不会到达最坏的上界。

//无论边权是否非负，图是否有负圈，使用SPFA都有很不错的效果。

//MAX\_V代表点数的最大值,MAX\_E代表边数的最大值

typedef pair<int, int> P;

int edgecount; //记录总边数，初始化为0

int head[MAX\_V]; //head[i]记录从 i 结点出发的第一条边，初始化为-1

//在面对多组数据的时候切记edgecount和head[]要在开始加边之前初始化

int d[MAX\_V]; //顶点s出发的最短距离

bool inq[MAX\_V]; //记录顶点是否在队列中

int visitcount[MAX\_V]; //记录顶点访问次数

queue<P> q; //second记录结点号，first记录目前已知的到达该结点的最短距离

struct edge {

int from, to, val, next; //记录从 from 结点出发的下一条边

edge() {}

edge(int \_f, int \_t, int \_v, int \_n) {

from = \_f, to = \_t, val = \_v, next = \_n;

}

}es[MAX\_E];

void addedge(int from, int to, int val) {

es[edgecount] = edge(from, to, val, head[from]);

head[from] = edgecount++;

}

bool spfa(int s) {

for (int i = 0; i < MAX\_V; i++) { //点的下标从0开始

d[i] = (i == s) ? 0 : inf;

inq[i] = (i == s);

visitcount[i] = 0;

}

while (!q.empty()) q.pop(); //特别是在面对多组数据的时候，这一步万万不可省略

q.push(make\_pair(d[s], s));

while (!q.empty()) {

int dist = q.front().first, u = q.front().second;

q.pop();

inq[u] = false;

if (visitcount[u]++ > MAX\_V) return true;

for (int e = head[u]; e != -1; e = es[e].next) {

int t = es[e].to, value = es[e].val;

if (d[t] > d[u] + value) {

d[t] = d[u] + value;

if (!inq[t]) {

inq[t] = true;

q.push(make\_pair(d[t], t));

}

}

}

}

return false; //不存在负圈

}

**Dijkstra算法**

注意：边的权值必须都是非负的！

邻接矩阵实现

复杂度：O(|V|2)

int cost[MAX\_V][MAX\_V]; //cost[u][v]表示边e=(u,v)的权值（不存在这条边时设为INF）

int d[MAX\_V]; //顶点s出发的最短距离

bool used[MAX\_V]; //已经使用过的点

int V; //顶点数

void dijkstra(int s) {

fill(d, d + V, INF); //fill()头文件为<iostream>

fill(used, used + V, false);

d[s] = 0;

while (1) {

int v = -1;

//从尚未使用过的顶点中选择一个距离最小的顶点

for (int u = 0; u < V; u++) {

if (!used[u] && (v == -1 || d[u] < d[v])) v = u;

}

if (v == -1) break;

used[v] = true;

for (int u = 0; u < V; u++) {

d[u] = min(d[u], d[v] + cost[v][u]);

}

}

}

1. priority\_queue实现

复杂度：O(|E|\*log|V|)

struct edge { int to, cost; };

typedef pair<int, int> P; //first是最短距离，second是顶点编号

int V;

vector<edge> G[MAX\_V];

int d[MAX\_V];

void dijkstra(int s) {

//通过指定greater<P>参数，堆按照first从小到大的顺序取出值

priority\_queue<P, vector<P>, greater<P> > que; //greater的头文件为<xfunctional>

// 如果要按照从大到小的顺序取出值，那么就用less<P>.less则不需添加头文件<xfunctional>

//e.s. priority\_queue<P, vector<P>, less<P> > que;

fill(d, d + V, INF);

d[s] = 0;

que.push(P(0, s));

while (!que.empty()) {

P p = que.top(); que.pop();

int v = p.second;

if (d[v] < p.first) continue;

for (int i = 0; i < G[v].size(); i++) {

edge e = G[v][i];

if (d[e.to] > d[v] + e.cost) { //这里可能要把d[v]改成p.first

d[e.to] = d[v] + e.cost;

que.push(P(d[e.to], e.to));

}

}

}

}

//dijkstra(链式前向星版）

const int maxm = 1000000 + 5; //边数

const int maxn = 10000 + 5; //点数

const int inf = 0x3f3f3f3f;

map<string, int> ind;

struct Edge {

int to, nxt, cost;

}edge[maxm];

int min\_cost[maxn];

int head[maxm], tot;

void init() {

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

tot = 0;

}

void addedge(int u, int v, int cost) {//u -> v

edge[tot].to = v;

edge[tot].nxt = head[u];

edge[tot].cost = cost;

head[u] = tot++;

}

void dijkstra(int s) {

queue<int> que;

que.push(s);

min\_cost[s] = 0;

while (!que.empty()) {

int v = que.front();

que.pop();

for (int i = head[v]; i != -1; i = edge[i].nxt) {

int to = edge[i].to;

if (min\_cost[to]>min\_cost[v] + edge[i].cost) {

min\_cost[to] = min\_cost[v] + edge[i].cost;

que.push(to);

}

}

}

}

**Bellman-Ford算法**

复杂度：O(|E|\*|V|)

//判断是否存在负圈

bool find\_negative\_loop() {

memset(d, 0, sizeof(d));

for (int i = 0; i < V; i++) {

for (int j = 0; j < E; j++) {

edge e = es[j];

if (d[e.to] > d[e.from] + e.cost) {

d[e.to] = d[e.from] + e.cost;

//如果第n次仍然更新，则存在负圈

if (i == V - 1) return true;

}

}

}

return false;

}

//计算单源最短路

void bellman\_ford(int s) { //s为源点

for (int i = 0; i < V; i++) d[i] = INF;

d[s] = 0;

while (1) {

bool update = false;

for (int i = 0; i < E; i++) {

edge e = es[i];

if (d[e.from] != INF&&d[e.to]>d[e.from] + e.cost) {

d[e.to] = d[e.from] + e.cost;

update = true;

}

}

if (!update) break;

}

}

**路径还原**

int prev[MAX\_V]; //最短路上的前趋顶点

//求从起点s出发到各个顶点的最短距离

void dijkstra(int s) {

fill(d, d + V, INF);

fill(used, used + V, false);

fill(prev, prev + V, -1);

d[s] = 0;

while (1) {

int v = -1;

for (int u = 0; u < V; u++) {

if (!used[u] && (v == -1 || d[u] < d[v])) v = u;

}

if (v == -1) break;

used[v] = true;

for (int u = 0; u < V; u++) {

if (d[u] > d[v] + cost[v][u]) {

d[u] = d[v] + cost[v][u];

prev[u] = v;

}

}

}

}

//到顶点t的最短路径

vector<int> get\_path(int t) {

vector<int> path;

for (; t != -1; t = prev[t]) path.push\_back(t);//不断沿着prev[t]走直到t=s

//这样得到的路径是反序的，所以翻转之

reverse(path.begin(), path.end());

return path;

}

**最小生成树**

const int maxn = 110; //最大点数

const int inf = 0x7fffffff;

int cost[maxn][maxn]; //cost[u][v] 表示边 e=(u,v) 的权值（不存在的情况下设为inf)

int mincost[maxn]; //从集合 X 出发的边到每个顶点的最小权值

bool used[maxn]; //顶点 i 是否包含在集合 X 中

int N; //顶点数

int prim() {

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

mincost[i] = inf, used[i] = false;

}

mincost[1] = 0;

int ret = 0;

while (1) {

int v = -1;

for (int u = 1; u <= N; u++) {

if (!used[u] && (v == -1 || mincost[u]<mincost[v])) v = u;

}

if (v == -1) break;

used[v] = true; //顶点 v 加入 X

ret += mincost[v];

for (int u = 1; u <= N; u++) {

mincost[u] = min(mincost[u], cost[v][u]);

}

}

return ret;

}

**有向无环图（DAG）的最小路径覆盖**

**定义**：在一个DAG中，找出最少的路径，使得这些路径经过了所有的点。

最小路径覆盖分为**最小不相交路径覆盖**和**最小可相交路径覆盖**。

**最小不相交路径覆盖**：各条路径经过的顶点各不相同。

**最小可相交路径覆盖**：各条路径经过的顶点可以相同。

特别的，每个点自己也可以称为是路径覆盖，只不过路径的长度是0。

**DAG的最小不相交路径覆盖**

**算法：把原图的每个点V拆成Vx和Vy两个点，如果有一条有向边A->B，那么就加边**

**Ax−>By。这样就得到了一个二分图。那么最小路径覆盖=原图的结点数-新图的最大匹配数。**

**证明：一开始每个点都是独立的为一条路径，总共有n条不相交路径。我们每次在二分图里找一条匹配边就相当于把两条路径合成了一条路径，也就相当于路径数减少了1。所以找到了几条匹配边，路径数就减少了多少。所以有最小路径覆盖=原图的结点数-新图的最大匹配数。**

**因为路径之间不能有公共点，所以加的边之间也不能有公共点，这就是匹配的定义。**

**DAG的最小可相交路径覆盖**

**算法：先用floyd求出原图的传递闭包，即如果a到b有路径，那么就加边a->b。然后就转化成了最小不相交路径覆盖问题。**

**证明：为了连通两个点，某条路径可能经过其它路径的中间点。比如1->3->4，2->4->5。但是如果两个点a和b是连通的，只不过中间需要经过其它的点，那么可以在这两个点之间加边，那么a就可以直达b，不必经过中点的，那么就转化成了最小不相交路径覆盖。**

//POJ2594 最小可相交路径覆盖

#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

const int maxn = 520;

bool G[maxn][maxn], used[maxn];

int to[maxn];

bool finds(int x, int n) {

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

if (G[x][j] && !used[j]) {

used[j] = true;

if (to[j] == 0 || finds(to[j], n)) {

to[j] = x;

return true;

}

}

}

return false;

}

int main()

{

int A, B;

int N, M;

while (scanf("%d%d", &N, &M) == 2) {

if (!N&&!M) break;

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

for (int j = 1; j <= N; j++) G[i][j] = false;

to[i] = 0;

}

while (M--) {

scanf("%d%d", &A, &B);

G[A][B] = true;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

if (G[j][i]) {

for (int k = 1; k <= N; k++) {

if (!G[j][k]) G[j][k] = G[i][k];

}

}

}

}//floyd求原图的传递闭包

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int ans = 0;

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

memset(used, false, sizeof(used));

if (finds(i, N)) ans++;

}

printf("%d\n", N - ans);

}

return 0;

}

**带花树开花算法**

//带花树开花算法（解决一般匹配问题）

#include <cstdio>

#include <cstring>

const int maxn = 250;

int N; //点数，编号从1到N

bool Graph[maxn][maxn];

int Match[maxn];

bool InQueue[maxn], InPath[maxn], InBlossom[maxn];

int Head, Tail;

int Queue[maxn];

int Start, Finish;

int NewBase;

int Fa[maxn], Base[maxn];

void Push(int u) {

Queue[Tail] = u;

Tail++;

InQueue[u] = true;

}

int Pop() {

int res = Queue[Head];

Head++;

return res;

}

int FindCommonAncestor(int u, int v) {

memset(InPath, false, sizeof(InPath));

while (1) {

u = Base[u];

InPath[u] = true;

if (u == Start) break;

u = Fa[Match[u]];

}

while (1) {

v = Base[v];

if (InPath[v]) break;

v = Fa[Match[v]];

}

return v;

}

void ResetTrace(int u) {

int v;

while (Base[u] != NewBase) {

v = Match[u];

InBlossom[Base[u]] = InBlossom[Base[v]] = true;

u = Fa[v];

if (Base[u] != NewBase) Fa[u] = v;

}

}

void BloosomContract(int u, int v) {

NewBase = FindCommonAncestor(u, v);

memset(InBlossom, false, sizeof(InBlossom));

ResetTrace(u); ResetTrace(v);

if (Base[u] != NewBase) Fa[u] = v;

if (Base[v] != NewBase) Fa[v] = u;

for (int tu = 1; tu <= N; tu++) {

if (InBlossom[Base[tu]]) {

Base[tu] = NewBase;

if (!InQueue[tu]) Push(tu);

}

}

}

void FindAugmentingPath() {

memset(InQueue, false, sizeof(InQueue));

memset(Fa, false, sizeof(Fa));

for (int i = 1; i <= N; i++) Base[i] = i;

Head = Tail = 1;

Push(Start);

Finish = 0;

while (Head < Tail) {

int u = Pop();

for (int v = 1; v <= N; v++) {

if (Graph[u][v] && (Base[u] != Base[v]) && (Match[u] != v)) {

if ((v == Start) || ((Match[v]>0) && Fa[Match[v]] > 0))

BloosomContract(u, v);

else if (Fa[v] == 0) {

Fa[v] = u;

if (Match[v] > 0) Push(Match[v]);

else {

Finish = v;

return;

}

}

}

}

}

}

void AugmentPath() {

int u, v, w;

u = Finish;

while (u > 0) {

v = Fa[u];

w = Match[v];

Match[v] = u;

Match[u] = v;

u = w;

}

}

void Edmonds() {

memset(Match, 0, sizeof(Match));

for (int u = 1; u <= N; u++) {

if (Match[u] == 0) {

Start = u;

FindAugmentingPath();

if (Finish > 0) AugmentPath();

}

}

}

void solve() {

Edmonds(); //进行匹配

int Count = 0; //匹配数，匹配对数是Count/2;

for (int u = 1; u <= N; u++) {

if (Match[u] > 0) Count++;

}

printf("%d\n", Count);

for (int u = 1; u <= N; u++) {

if (u < Match[u]) printf("%d %d\n", u, Match[u]);

}

}

int main() {

int u, v;

scanf("%d", &N);

while (scanf("%d%d", &u, &v) == 2)

Graph[u][v] = Graph[v][u] = true;

solve();

return 0;

}

**差分约束**

1、介绍

如若一个系统由n个变量和m个不等式组成，并且这m个不等式对应的系数矩阵中每一行有且仅有一个1和-1，其它的都为0，这样的系统称为**差分约束( difference constraints )**系统。

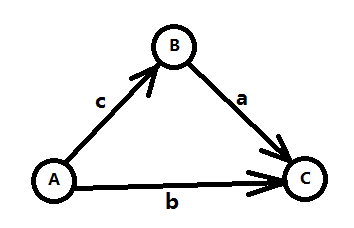
我们先来介绍一种最简单的情况，如下三个不等式：

B - A <= c      (1)

C - B <= a      (2)

C - A <= b      (3)

我们想要知道C - A的最大值，通过(1) + (2)，可以得到 C - A <= a + c，所以这个问题其实就是求min{b, a+c}。可以将A、B、C看成点来建立如下的图象：

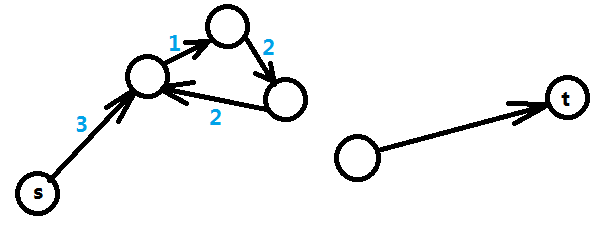


我们发现min{b, a+c}正好对应了A到C的最短路，而这三个不等式就是著名的**三角不等式**。将三个不等式推广到m个，变量推广到n个，就变成了n个点m条边的最短路问题了。

2、解的存在性

差分约束系统的解有三种情况：1、有解；2、无解；3、无限多解。

有解对应到最短路图象中就是图象存在最短路，而当路径中出现负圈时，最短路不存在，解也就不存在了。而当从起点s无法到达终点t时，表明X[t]和X[s]之间并没有约束关系，这种情况下X[t] - X[s]的最大值是无限大， X[t]和X[s]的取值有无限多种。其图象如下：



3、变种

将原来的不等式变成如下：

B - A >= c      (1)

C - B >= a      (2)

C - A >= b      (3)

求C-A的最小值，类比之前的方法，其实就是求max{b, c+a}，对应的是求C到A的最长路。

4、不等式标准化

如果给出的不等式有"<="也有">="，又该如何解决呢？很明显，首先需要关注最后的问题是什么，如果需要求的是两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路；相反，如果需要求的是两个变量差的最小值，那么需要将所有不等式转化成">="，建图后求最长路。

      如果有形如：A - B = c 这样的等式呢？我们可以将它转化成以下两个不等式：

A - B >= c      (1)

A - B <= c      (2)

       再通过上面的方法将其中一种不等号反向，建图即可。

       最后，如果这些变量都是整数域上的，那么遇到A - B < c这样的不带等号的不等式，我们需要将它转化成"<="或者">="的形式，即 A - B <= c - 1。

# 数学

1、组合数学

**打表求组合数**

//利用公式 C(i,j)=C(i-1,j-1)+C(i-1,j)

int C[maxn][maxn]; //C[i][j]代表从i个中取出j个的组合数

void init() {

C[0][0] = C[1][0] = C[1][1] = 1;

for (int i = 2; i<maxn; i++) {

C[i][0] = C[i][i] = 1;

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

C[i][j] = C[i - 1][j - 1] + C[i - 1][j];

}

}

2、代数

**（矩阵）快速幂**

const int maxn = 1000;

struct Matrix {

int mat[maxn][maxn];

};

Matrix Multiply(Matrix x, Matrix y, int n) {

Matrix temp;

memset(temp.mat, 0, sizeof(temp.mat));

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

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

temp.mat[i][j] += (x.mat[i][k] \* y.mat[k][j]);

}

}

}

return temp;

}

Matrix Fast\_Power(Matrix a, int m, int n) {

Matrix res;

memset(res.mat, 0, sizeof(res.mat));

for (int i = 0; i < n; i++) res.mat[i][i] = 1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//快速幂模板

while (m) {

if (m & 1) res = Multiply(res, a, n);

m >>= 1;

a = Multiply(a, a, n);

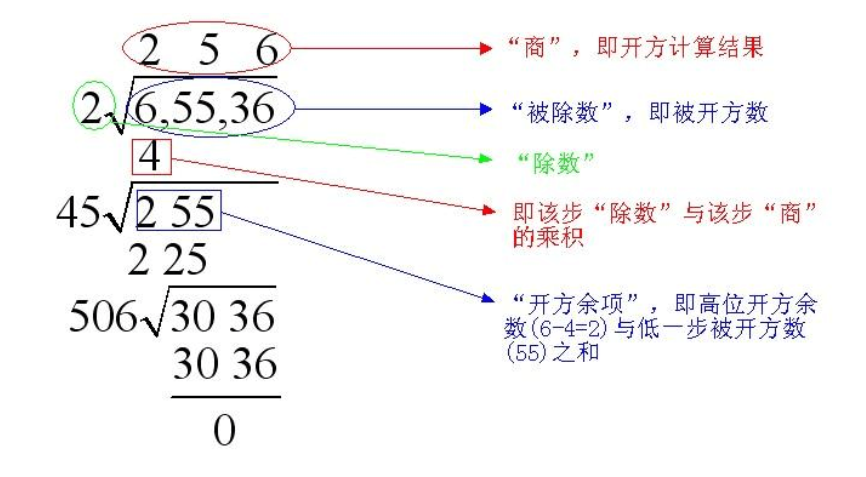
}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

return res;

}

**手算开平方**



Step1：将被开方数(为了形象，表述成“被除数”，此例中即为65536)从个位往高位每两位一断写成6,55,35的形式，为了方便表述，以下每一个“,”称为一步

Step2：从高位开始计算开方。例如第一步为6，由于22=4<6<9=32，因此只能商2(这就是和除法不同的地方，“除数”和“商”的计算位必须相同)。于是将2写在根号上方，计算开方余项。即高位余项加一步低位，此例中，即为高位余项2和低位一步55，余项即为255。

Step3：将Step2得到的第一步开方得数2乘以20(原理在后面证明)作为第二步除数的高位。即本步除数是4x(四十几)。按照要求，本步的商必须是x。因为45×5=225<255<46×6=276，所以本步商5。

Step4：按照类似方法，继续计算以后的各步。其中，每一步的除数高位都是20×已求出的部分商。例如第三步的除数高位就是25×20=500，所以第三步除数为50x。本例中，506×6=3036恰好能整除，所以256就是最终计算结果。

3、数论

**欧拉函数**

在[数论](https://baike.baidu.com/item/%E6%95%B0%E8%AE%BA)，对[正整数](https://baike.baidu.com/item/%E6%AD%A3%E6%95%B4%E6%95%B0)n，[欧拉](https://baike.baidu.com/item/%E6%AC%A7%E6%8B%89)函数是小于n的正整数中与n[互质](https://baike.baidu.com/item/%E4%BA%92%E8%B4%A8)的数的数目（φ(1)=1）。此函数以其首名研究者欧拉命名(Euler'so totient function)，它又称为Euler's totient function、[φ函数](https://baike.baidu.com/item/%CF%86%E5%87%BD%E6%95%B0)、欧拉[商数](https://baike.baidu.com/item/%E5%95%86%E6%95%B0)等。 例如φ(8)=4，因为1,3,5,7均和8互质。

//直接求法

int euler\_phi(int n) {

int m = (int)sqrt(n + 0.5); //计算上界

int ans = n;

for (int i = 2; i <= m; i++) {

if (n % i == 0) { //如果当前这个数是n的质因子

ans = ans / i \* (i - 1);

while (n % i == 0) n /= i; //除去n的质因子

}

}

if (n > 1) ans = ans / n \* (n - 1); //还剩下最后一个质因子

return ans;

}

//打表求法

void phi\_table(int n) {

for (int i = 2; i <= n; i++) { //初始化

phi[i] = 0;

}

phi[1] = 1;

for (int i = 2; i <= n; i++)

if (!phi[i]) //如果phi[i] == 0则表明i是素数

for (int j = i; j <= n; j += i) { //将所以i的倍数赋值

if (!phi[j]) phi[j] = j; //如果j没访问过则phi[j] = j

phi[j] = phi[j] / i \* (i - 1); //j的phi值 = j \* (质因子i - 1) / (质因子)

}

}

4、笔记

**二分矩阵公式**

设矩阵A.

A + A2 + A3 + A4 + A5 + A6 = (A + A2 + A3) + A3(A + A2 + A3)

**循环矩阵**

定义  具有以下形式的n阶方阵A称为关于a0, a1, …, an-1的循环矩阵。

主要性质：

同阶循环矩阵的和矩阵为循环矩阵，乘积也为循环矩阵。利用这个性质可以把矩阵乘法的时间复杂度降为O(N2)。

**算术基本定理**

任何一个大于1的自然数 N,如果N不为[质数](https://baike.baidu.com/item/%E8%B4%A8%E6%95%B0)，那么N可以唯一分解成有限个质数的乘积***N***=P1a1P2a2P3a3 … Pnan，这里P1 < P2 < P3 … < Pn均为[质数](https://baike.baidu.com/item/%E8%B4%A8%E6%95%B0)，其中指数ai是正整数。

应用：

1. 一个大于1的正整数N，如果它的标准分解式为：***N***=P1a1P2a2P3a3 … Pnan，那么它的正因数个数为σ0(N) = (1+a1)(1+a2) … (1+an).
2. 它的全体[正因数](https://baike.baidu.com/item/%E6%AD%A3%E5%9B%A0%E6%95%B0)之和为σ1(N) = (1+p1+p12+ … +p1a1)(1+p2+p22+ … +p2a2) … (1+pn+pn2+ … +pnan). 当σ1(N) = 2N时就称N为完全数。是否存在奇完全数，是一个至今未解决之猜想。

**调和级数**

**调和级数（英语：Harmonic series）**是一个发散的无穷级数，表达式为：

调和级数的第n个*部分和*为：

,

也叫第n个调和数。第n个调和数与n的自然对数的差值（即）收敛于欧拉-马歇罗尼常数

（）。两个不同的调和数之间的差值永远不是整数。除了n=1时以外，没有任何一个调和数是整数。

求第n个调和级数的方法：

当n < 1e6（具体大小可以自己判断）时直接打表解决；

当n较大时调用公式。

**枚举全排列**

直接使用next\_permutation() 函数（头文件：#include <algorithm>）

**e.s.**

while (next\_permutation(str, str + lenth\_of\_str)) {

...

}

**勾股数**

指能够构成直角三角形三条边的三个正整数。对于勾股数(a, b, c)，有a2+b2=cc, a,b,c都是自然数。

已知一个数a，构造勾股数的方法：

若a为大于1的奇数2n+1，b=2n2+2n, c=2n2+2n+1；

若a为大于4的偶数2n，b=n2-1, c=n2+1.

# 数据结构

1、字符串

**后缀数组**

DA算法（sort排序）

const int maxn = 1e4;

int len, tk;

int Rank[maxn], tmp[maxn];

char S[maxn];

int sa[maxn], lcp[maxn];

//sa[i]: rank 为 i 的后缀的第一个字母的位置；lcp[i]: 后缀S[sa[i]...]与后缀S[sa[i+1]...]的最长公共前缀的长度

bool compare\_sa(int i, int j) {// 比较 (Rank[i], Rank[i+k]) 和 (Rank[j], Rank[j+k])

if (Rank[i] != Rank[j]) return Rank[i] < Rank[j];

else {

int ri = i + tk <= len ? Rank[i + tk] : -1;

int rj = j + tk <= len ? Rank[j + tk] : -1;

return ri < rj;

}

}

void construct\_sa() {//计算字符串 S 的后缀数组

len = strlen(S);

//初始长度为 1，Rank[] 直接取字符的编码

for (int i = 0; i <= len; i++) {

sa[i] = i;

Rank[i] = i < len ? S[i] : -1;

}

//利用对长度为 k 的排序的结果对长度为 2k 的排序

for (tk = 1; tk <= len; tk \*= 2) {

sort(sa, sa + len + 1, compare\_sa);

//先在 tmp[] 中临时存储新计算的 Rank，再转存回 Rank[] 中

tmp[sa[0]] = 0;

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

tmp[sa[i]] = tmp[sa[i - 1]] + (compare\_sa(sa[i - 1], sa[i]) ? 1 : 0);

}

for (int i = 0; i <= len; i++) {

Rank[i] = tmp[i];

}

}

}

void construct\_lcp() {

int h = 0;

lcp[0] = 0;

for (int i = 0; i < len; i++) {

//计算字符串中从位置 i 开始的后缀及其在后缀数组中的前一个后缀的 lcp

int j = sa[Rank[i] - 1];

//将 h 先减去首字母的 1 长度，在保持前缀相同的前提下不断增加

if (h > 0) h--;

for (; j + h < len && i + h < len; h++) {

if (S[j + h] != S[i + h]) break;

}

lcp[Rank[i] - 1] = h;

}

}

DA算法（基数排序）

//求重复次数最多的连续重复子串，SPOJ - REPEATS

#include <iostream>

#include <cstdio>

#include <algorithm>

using namespace std;

const int maxn = 50000 + 10, inf = 0x7fffffff;

int wa[maxn], wb[maxn], wv[maxn], wss[maxn];

int cmp(int \*r, int a, int b, int l){

return r[a] == r[b] && r[a + l] == r[b + l];

}

void da(int \*r, int \*sa, int n, int m)

{

int i, j, p, \*x = wa, \*y = wb, \*t;

for (i = 0; i<m; i++) wss[i] = 0;

for (i = 0; i<n; i++) wss[x[i] = r[i]]++;

for (i = 1; i<m; i++) wss[i] += wss[i - 1];

for (i = n - 1; i >= 0; i--) sa[--wss[x[i]]] = i;

for (j = 1, p = 1; p<n; j \*= 2, m = p){

for (p = 0, i = n - j; i<n; i++) y[p++] = i;

for (i = 0; i<n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;

for (i = 0; i<n; i++) wv[i] = x[y[i]];

for (i = 0; i<m; i++) wss[i] = 0;

for (i = 0; i<n; i++) wss[wv[i]]++;

for (i = 1; i<m; i++) wss[i] += wss[i - 1];

for (i = n - 1; i >= 0; i--) sa[--wss[wv[i]]] = y[i];

for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i<n; i++)

x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;

}

return;

}

int Rank[maxn], height[maxn];

void calheight(int \*r, int \*sa, int n)

{

int i, j, k = 0;

for (i = 1; i <= n; i++) Rank[sa[i]] = i;

for (i = 0; i<n; height[Rank[i++]] = k)

for (k ? k-- : 0, j = sa[Rank[i] - 1]; r[i + k] == r[j + k]; k++);

return;

}

int RMQ[maxn];

int mm[maxn], best[20][maxn];

void initRMQ(int n) {

mm[0] = -1;

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

mm[i] = ((i&(i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

for (int i = 1; i <= n; i++) best[0][i] = i;

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

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

int a = best[i-1][j];

int b = best[i - 1][j + (1 << (i - 1))];

if (RMQ[a] < RMQ[b]) best[i][j] = a;

else

{

best[i][j] = b;

}

}

}

}

int askRMQ(int a, int b) {

int t;

t = mm[b - a + 1];

b -= (1 << t) - 1;

a = best[t][a]; b = best[t][b];

return RMQ[a] < RMQ[b] ? a : b;

}

int lcp(int a, int b) {

a = Rank[a]; b = Rank[b];

if (a > b) swap(a, b);

return height[askRMQ(a + 1, b)];

}

int sa[maxn];

int S[maxn];

int main()

{

char inp[5];

int H, len;

scanf("%d", &H);

while (H--) {

scanf("%d", &len);

for (int i = 0; i<len; i++) {

scanf("%s", inp);

if (inp[0] == 'a') S[i] = 1;

else S[i] = 2;

}

S[len] = 0;

da(S, sa, len + 1, 3);

calheight(S, sa, len);

for (int i = 1; i <= len; i++) RMQ[i] = height[i];

initRMQ(len+1);

int ans = 1;

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

int ret = 0;

for (int j = 0; j + i<len; j+=i) {

int L = lcp(j, j + i);

int temp = L / i + 1;

int k = j - (i - L%i);

if (k >= 0 && lcp(k, k + i) / i + 1>temp)

temp++;

ret = max(ret, temp);

}

ans = max(ans, ret);

}

printf("%d\n", ans);

}

return 0;

}

**字典树**

//HDU1671

//题目大意：给出n个号码，看其中是否有一个号码是另一个号码的前缀，例如“123”就是“12345”//的前缀，有这种情况时输出NO，反之输出YES

#include <cstring>

#include <cstdio>

#include <cstdlib>

#define MAX 12

struct Trie {

int next[MAX];

int v; //根据需要变化

void init() {

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

v = 1;

}

};

int cnt;

Trie Heap[100000];

void createTrie(char \*str) {

int len = strlen(str);

int now = 0;

for (int i = 0; i<len; i++) {

int id = str[i] - '0';

if (Heap[now].next[id] == -1) {

Heap[++cnt].init();

Heap[now].next[id] = cnt;

now = cnt;

}

else {

int net = Heap[now].next[id];

Heap[net].v++;

now = net;

}

}

Heap[now].v = -1; //若为结尾，则将v改成-1表示

}

int findTrie(char \*str) {

int len = strlen(str);

int now = 0;

for (int i = 0; i<len; i++) {

int id = str[i] - '0';

int net = Heap[now].next[id];

now = net;

if (now == -1) //若为空集，表示不存在以此为前缀的串

return 0;

if (Heap[now].v == -1) //字符集中已有串是此串的前缀

return -1;

}

return 1; //此串是字符集中某串的前缀

}

int main() {

int t, n;

char temp[15];

scanf("%d", &t);

while (t--) {

Heap[0].init();

cnt = 0;

bool yes = true;

scanf("%d", &n);

while (n--) {

scanf("%s", temp);

if (!yes) continue;

if (findTrie(temp) != 0) {

yes = false;

}

createTrie(temp);

}

if (yes) printf("YES\n");

else printf("NO\n");

}

return 0;

}

**字典树（儿子-兄弟表示法）**

//UVa11732(《算法竞赛入门经典——训练指南》P210）

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const int maxn = 4000004;

char ch[maxn]; //保存结点字符

int son[maxn], bro[maxn]; //son是链表头, bro[u]表示u的下一个兄弟结点

int nds; //nds是下一个可以分配的结点

int cnt[maxn]; //cnt[x]表示经过结点x的字符串数

ll add(char \*str) {

int len = strlen(str);

str[len] = '$', str[len + 1] = '\0';

ll ret = 0;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//儿子-兄弟表示法（就是结合了链表的字典树，有点类似链式前向星，目的是节省空间）

int now = 0;

int last = cnt[now];

for (int i = 0; i <= len; i++) {

int v;

for (v = son[now]; v; v = bro[v]) {

if (ch[v] == str[i]) break;

}

if (!v) {

bro[nds] = son[now];

ch[nds] = str[i];

cnt[nds] = son[nds] = 0;

v = son[now] = nds++;

}

now = v;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int news = cnt[now];

ret += news \* 2;

ret += (last - news);

last = cnt[now];

cnt[now]++;

}

return ret;

}

char str[4005];

int main() {

int N, kase = 1;

while (scanf("%d", &N) == 1 && N) {

nds = 1, cnt[0] = son[0] = bro[0] = 0; //初始化

ll ans = 0;

while (N--) {

scanf("%s", str);

ans += add(str);

cnt[0]++;

}

printf("Case %d: %lld\n", kase++, ans);

}

return 0;

}

**哈希公式**

* **hash[i]=(hash[i-1]\*p+idx(s[i]))%mod**

表示第 i 个前缀的哈希值，是一个哈希的前缀和

P 和 mod 取素数，p 取 6 到 8 位的素数，mod 可取 1e9+7 或 1e9+9.

* **hash[l..r]=(hash[r]-hash[l-1]\*(p^(r-l+1)))%mod**

表示 S[l…r]这个子串的hash值.

2、其他

**并查集**

int par[maxn]; //父亲

int Rank[maxn]; //树的高度

//初始化n个元素

void init(int n) {

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

par[i] = i;

Rank[i] = 0;

}

}

//查询树的根

int finds(int x) {

if (par[x] == x)

return x;

return par[x] = finds(par[x]);

}

//合并x和y所属集合

void unite(int x, int y) {

x = finds(x);

y = finds(y);

if (x == y) return;

if (Rank[x] < Rank[y]) {

par[x] = y;

}

else {

par[y] = x;

if (Rank[x] == Rank[y]) Rank[x]++;

}

}

//判断x和y是否属于同一个集合

bool same(int x, int y) {

return finds(x) == finds(y);

}

**线段树**

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

#define root 1 , N , 1

typedef long long ll;

const int maxn = 111111;

ll add[maxn << 2], sum[maxn << 2];

void PushUp(int rt) {

sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];

}

void PushDown(int rt, int m) {

if (add[rt]) {

add[rt << 1] += add[rt];

add[rt << 1 | 1] += add[rt];

sum[rt << 1] += add[rt] \* (m - (m >> 1));

sum[rt << 1 | 1] += add[rt] \* (m >> 1);

add[rt] = 0;

}

}

void build(int l, int r, int rt) {

add[rt] = 0;

if (l == r) {

scanf("%lld", &sum[rt]);

return;

}

int m = (l + r) >> 1;

build(lson);

build(rson);

PushUp(rt);

}

void update(int L, int R, int c, int l, int r, int rt) {

if (L <= l && r <= R) {

add[rt] += c;

sum[rt] += (ll)c \* (r - l + 1);

return;

}

PushDown(rt, r - l + 1);

int m = (l + r) >> 1;

if (L <= m) update(L, R, c, lson);

if (m < R) update(L, R, c, rson);

PushUp(rt);

}

ll query(int L, int R, int l, int r, int rt) {

if (L <= l && r <= R) {

return sum[rt];

}

PushDown(rt, r - l + 1);

int m = (l + r) >> 1;

ll ret = 0;

if (L <= m) ret += query(L, R, lson);

if (m < R) ret += query(L, R, rson);

return ret;

}

int main() {

int N, Q, a, b, c;

char op[2];

scanf("%d%d", &N, &Q);

build(root);

while (Q--) {

scanf("%s", op);

if (op[0] == 'Q') {

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

printf("%lld\n", query(a, b, root));

}

else {

scanf("%d%d%d", &a, &b, &c);

update(a, b, c, root);

}

}return 0;

}

**线段树之区间合并**

**例题：POJ3667 Hotel**

题意:1 a:询问是不是有连续长度为a的空房间,有的话住进最左边  
2 a b:将[a,a+b-1]的房间清空

思路:记录区间中最长的空房间

线段树操作:update:区间替换 query:询问满足条件的最左断点

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

using namespace std;

#define lson l , m , rt << 1

#define rson m + 1 , r , rt << 1 | 1

const int maxn = 55555;

int lsum[maxn << 2], rsum[maxn << 2], msum[maxn << 2];

int cover[maxn << 2];

void PushDown(int rt, int m) {

if (cover[rt] != -1) {

cover[rt << 1] = cover[rt << 1 | 1] = cover[rt];

msum[rt << 1] = lsum[rt << 1] = rsum[rt << 1] = cover[rt] ? 0 : m - (m >> 1);

msum[rt << 1 | 1] = lsum[rt << 1 | 1] = rsum[rt << 1 | 1] = cover[rt] ? 0 :(m>>1);

cover[rt] = -1;

}

}

void PushUp(int rt, int m) {

lsum[rt] = lsum[rt << 1];

rsum[rt] = rsum[rt << 1 | 1];

if (lsum[rt] == m - (m >> 1)) lsum[rt] += lsum[rt << 1 | 1];

if (rsum[rt] == (m >> 1)) rsum[rt] += rsum[rt << 1];

msum[rt] = max(lsum[rt << 1 | 1] + rsum[rt << 1], max(msum[rt << 1], msum[rt<<1|1]));

}

void build(int l, int r, int rt) {

msum[rt] = lsum[rt] = rsum[rt] = r - l + 1;

cover[rt] = -1;

if (l == r) return;

int m = (l + r) >> 1;

build(lson);

build(rson);

}

void update(int L, int R, int c, int l, int r, int rt) {

if (L <= l && r <= R) {

msum[rt] = lsum[rt] = rsum[rt] = c ? 0 : r - l + 1;

cover[rt] = c;

return;

}

PushDown(rt, r - l + 1);

int m = (l + r) >> 1;

if (L <= m) update(L, R, c, lson);

if (m < R) update(L, R, c, rson);

PushUp(rt, r - l + 1);

}

int query(int w, int l, int r, int rt) {

if (l == r) return l;

PushDown(rt, r - l + 1);

int m = (l + r) >> 1;

if (msum[rt << 1] >= w) return query(w, lson);

else if (rsum[rt << 1] + lsum[rt << 1 | 1] >= w) return m - rsum[rt << 1] + 1;

return query(w, rson);

}

int main() {

int n, m;

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

build(1, n, 1);

while (m--) {

int op, a, b;

scanf("%d", &op);

if (op == 1) {

scanf("%d", &a);

if (msum[1] < a) puts("0");

else {

int p = query(a, 1, n, 1);

printf("%d\n", p);

update(p, p + a - 1, 1, 1, n, 1);

}

}

else {

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

update(a, a + b - 1, 0, 1, n, 1);

}

}

return 0;

}

**可持久化线段树**

//可持久化线段树求区间第K小的数（POJ2104)

#include <cstdio>

#include <algorithm>

using namespace std;

const int maxn = 2000000; //数组开多大需要自己估算一下，大概是(4\*n+m\*log n),其中 m 是建新树的次数

int num[maxn], san[maxn];

int lson[maxn], rson[maxn]; //左、右儿子

int sum[maxn]; //线段树里保存的值

int T[maxn]; //每个结点在线段树里的标号

int tot = 0;

void build(int l, int r, int &rt) {

rt = ++tot; //不再使用堆式存储，而是动态开点

sum[rt] = 0;

if (l == r) return;

int m = (l + r) >> 1;

build(l, m, lson[rt]); build(m + 1, r, rson[rt]);

}

void update(int last, int pos, int l, int r, int &rt) { //pos点加一,last是旧点的标号

rt = ++tot;

lson[rt] = lson[last];

rson[rt] = rson[last];

sum[rt] = sum[last] + 1; //继承之前的线段树并加一

if (l == r) return;

int m = (l + r) >> 1;

if (pos <= m) update(lson[last], pos, l, m, lson[rt]);

else update(rson[last], pos, m + 1, r, rson[rt]);

}

int query(int s, int t, int k, int l, int r) { //查询s到t区间第k大的数

if (l == r) return l;

int m = (l + r) >> 1;

int cnt = sum[lson[t]] - sum[lson[s]]; //cnt为左子树新树减旧树

if (k <= cnt) return query(lson[s], lson[t], k, l, m);

else return query(rson[s], rson[t], k - cnt, m + 1, r);

}

int main() {

int N, M; //N个值，M次询问

scanf("%d%d", &N, &M);

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

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

san[i] = num[i];

}

sort(san + 1, san + N + 1); //离散化后的数组

int cnt = unique(san + 1, san + N + 1) - san - 1;

build(1, cnt, T[0]);

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

num[i] = lower\_bound(san + 1, san + cnt + 1, num[i]) - san;

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

update(T[i - 1], num[i], 1, cnt, T[i]);

int s, t, k;

while (M--) {

scanf("%d%d%d", &s, &t, &k); //求[s,t]内第k小的数

printf("%d\n", san[query(T[s - 1], T[t], k, 1, cnt)]);

}

return 0;

}

//可持久化线段树（区间更新，区间查询）

//SPOJ TTM

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const int maxn = 1e5 + 5;

int lson[40 \* maxn], rson[40 \* maxn];

ll sum[40 \* maxn], add[40 \* maxn]; //使用 add[] 数组取代 pushdown()

int T[maxn];

int tot = 0;

void pushup(int now, int l, int r) {

sum[now] = sum[lson[now]] + sum[rson[now]] + add[now] \* (r - l + 1);

}

void build(int l, int r, int &rt) {

rt = ++tot;

add[rt] = 0;

if (l == r) {

scanf("%lld", &sum[rt]);

return;

}

int m = (l + r) >> 1;

build(l, m, lson[rt]);

build(m + 1, r, rson[rt]);

pushup(rt, l, r);

}

void update(int last, int L, int R, ll ad, int l, int r, int &rt) {

rt = ++tot;

lson[rt] = lson[last]; rson[rt] = rson[last];

sum[rt] = sum[last], add[rt] = add[last];

if (L <= l&&r <= R) {

sum[rt] += (r - l + 1)\*ad;

add[rt] += ad;

return;

}

int m = (l + r) >> 1;

if (L <= m) update(lson[last], L, R, ad, l, m, lson[rt]);

if (m<R) update(rson[last], L, R, ad, m + 1, r, rson[rt]);

pushup(rt, l, r);

}

ll query(int L, int R, ll ad, int l, int r, int rt) {

if (L <= l&&r <= R)

return ad\*(r - l + 1) + sum[rt];

ad += add[rt];

int m = (l + r) >> 1;

ll ret = 0;

if (L <= m) ret += query(L, R, ad, l, m, lson[rt]);

if (R>m) ret += query(L, R, ad, m + 1, r, rson[rt]);

return ret;

}

int main() {

int n, m;

int now = 0;

char ord[5];

int l, r, t;

ll d;

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

build(1, n, T[0]);

while (m--) {

scanf("%s", ord);

if (ord[0] == 'C') { //[l,r] 中的每一个数 +d

scanf("%d%d%lld", &l, &r, &d);

update(T[now], l, r, d, 1, n, T[now + 1]);

now++; //时间戳 +1

}

else if (ord[0] == 'Q') { //查询目前 [l,r] 的区间和

scanf("%d%d", &l, &r);

printf("%lld\n", query(l, r, 0, 1, n, T[now]));

}

else if (ord[0] == 'H') { //查询在时刻 t [l,r] 的区间和

scanf("%d%d%d", &l, &r, &t);

printf("%lld\n", query(l, r, 0, 1, n, T[t]));

}

else { //使时间倒回到时刻 t

scanf("%d", &t);

now = t;

}

}

return 0;

}

**康托展开**

int factor[] = { 1,1,2,6,24,120,720,5040,40320,362880 };//阶乘

int cantor(char x[]){//康托展开

int sum = 0, s;

for (int i = 0; i < 9; i++){

s = 0;

for (int j = i + 1; j < 9; j++)

if (x[j] < x[i])

s++;

sum += s \* factor[9 - i - 1];

}

return sum;

}

void trans(int x) {//康托展开逆运算

int visit[10] = { 0 };

int t;

for (int i = 0; i < 9; i++){

t = x / factor[9 - i - 1];

x %= factor[9 - i - 1];

for (int j = 0; j <= t; j++)

if (visit[j] == 1)

t++;

visit[t] = 1;

st[i] = t;

}

}

# 动态规划

**数位DP**

ll dp[20][state];//不同题目状态不同

int a[maxn];

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说明这个数枚举完了

if (pos == -1) return 1;/\*这里一般返回1，表示枚举的这个数是合法的，那么这里就需要在枚举时每一位都满足题目条件，

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 \*/

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

int up = limit ? a[pos] : 9;//根据limit判断枚举的上界up;

ll ans = 0;

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

if () ...

else if ()...

ans += dfs(pos - 1,/\*状态转移\*/, lead && i == 0, limit && i == a[pos])

/\*大概就是说，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\*/

}

if (!limit && !lead) dp[pos][state] = ans;//记录状态

/\*此处应与上面的记忆化保证一致性\*/

return ans;

}

ll solve(ll x){

int pos = 0;

while (x){//把数位分解出来

a[pos++] = x % 10;

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

}

int main(){

ll le, ri;

//在此处初始化dp数组为-1，在某些情况下万不得已需要将初始化放到 while 循环内

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

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

}

return 0;

}

# 其他

**三分法**

//CF808E

//题意：

//有n个物品，背包容量为m。(1 ≤ n ≤ 100000, 1 ≤ m ≤ 300000)

//每个物品占有空间w，价值为cost[i]。(1 ≤ w ≤ 3, 1 ≤ cost ≤ 109)

//现在问如何可以使拿到的物品的价值最大。

#include <cstdio>

#include <algorithm>

#include <functional>

using namespace std;

const int MAXN = 100000 + 5;

int cost[5][MAXN];

int t[5];

int n, m;

long long sum[5][MAXN];

int main()

{

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

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

int k;

scanf("%d", &k);

scanf("%d", &cost[k][++t[k]]);

}

for (int i = 1; i <= 3; i++) sort(cost[i] + 1, cost[i] + t[i] + 1, greater<int>());

for (int i = 1; i <= 3; i++)

for (int j = 1; j <= t[i]; j++)

sum[i][j] = sum[i][j - 1] + cost[i][j];

long long ans = 0;

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

int l = 0, r = min(t[2], i / 2);

while (r - l>30) {

int mid = (l + r) / 2, mmid = (mid + r) / 2;

if (sum[2][mid] + sum[1][i - mid \* 2] >= sum[2][mmid] + sum[1][i - mmid \* 2])

r = mmid;

else

l = mid;

}

int t1 = min(t[1], i - 2 \* l);

long long maxc = sum[2][l] + sum[1][t1];

int maxn = l, maxm = t1 + l \* 2;

for (int j = l + 1; j <= r; j++) {

int tt1 = min(t[1], i - 2 \* j);

if (sum[2][j] + sum[1][tt1]>maxc) {

maxn = j;

maxc = sum[2][j] + sum[1][tt1];

maxm = j \* 2 + tt1;

}

}

long long temp = maxc;

int t3 = min(t[3], (m - maxm) / 3);

temp += sum[3][t3];

ans = max(ans, temp);

}

printf("%I64d\n", ans);

return 0;

}

**遍历set**

set<int> s;

for (set<int>::iterator it = s.begin(); it != s.end(); it++)

printf("%d\n", \*it);//从小到大输出

**最优调度问题及Johnson法则**

**问题描述：**  
    n个作业{1,2,3....,n}，要由两台机器M1, M2 组成的流水线上完成加工，每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1，M2 加工作业i所需时间为ai和bi 。问如何调度使得完成所有作业的总用时最少。

**Johnson法则**

（1）令N1 = {i|ai<bi},N2 = {i|ai>=bi}。  
（2）将N1中作业依照ai增序排列，N2中作业依bi减序排列。  
（3）N1中作业接N2中作业构成满足Johnson法则的最优调度。