算法进阶笔记

2020年四月

快读优化

```
inline int read() {
    register int x = 0, f = 0, ch;
    while(!isdigit(ch = getchar())) f |= ch == '-';
    while(isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48), ch = getchar();
    return f ? -x : x;
}
```

关于 vector

vector其实是一种可变长数组。有两个需要注意的点。

当我们直接用下标访问vector的某个位置并赋值时，必须保证vector被初始化过，访问位置不能超过初始化的长度。

当我们采用v.push_back(x)来扩充数组的时候，可以不对vector进行初始化。

递归

递归实现排列型枚举

把1~n这n(n<10)个整数排成一行后随机打乱顺序，输出所有可能的次序。

```
int n, m;
bool chosen[20];//每个数是否放到了某个位置
int order[20];//某个位置放的是哪个数
int sum;
void calcs(int k)
{
    if(k==n+1)//递归终点，输出
    {
        for (int i = 1; i <= n;i++)
        {
            printf("%d ", order[i]);
        }
        sum++;
        puts("");
        return;
    }
    for (int i = 1; i <= n;i++)
    {
        if(chosen[i])//i已经被加入了排列数组，每个数只能出现一次
            continue;
        order[k] = i;//k号位置放i
        chosen[i] = 1;//设置i已经被放置
```

```
        calcs(k + 1);
        chosen[i] = 0;//设置i未被放置,意味着k号位置还没有放置任何数
    }
}
int main()
{
    cin >> n ;
    calcs(1);
    cout << sum << endl;
    return 0;
}
```

Max2问题：找出数组中最大的两个数，要求比较的次数尽可能的少。

方法1：

```
void compare(int A[],int lo,int hi,int &x1,int &x2)
{
    //* 维护x1和x2两个指针，x1大于等于x2，先与x2比较，再与x1比较
    if(A[x1=lo]<A[x2=lo+1])
        swap(x1, x2);
    for (int i = lo + 2; i < hi; i++)
    {
        if(A[x2]<A[i])
        {
            if(A[x1]<A[x2=i])
                swap(x1, x2);
        }
    }
}
```

最好情况：比较$1+(n-2)*1$次。

最坏情况：比较$1+(n-2)*2$次。

改进算法：分治

```
void max2(int A[],int lo,int hi,int &x1,int &x2)
{
    if(lo+2==hi)
    {
        if(A[x1=lo]<A[x2=lo+1])
            swap(x1, x2);
        return;
    }
    if(lo+3==hi)
    {
        if(A[x1=lo]<A[x2=lo+1])
            swap(x1, x2);
        for (int i = lo + 2; i < hi;i++)
        {
            if(A[x2]<A[i])
            {
                if(A[x1]<A[x2=i])
                    swap(x1, x2);
```

```
            }
        }
        return;
    }
    int mid = (lo + hi) >> 1;
    int L1, R1, L2, R2;
    max2(A, lo, mid, L1, R1);
    max2(A, mid, hi, L2, R2);
    if(A[L1]>A[L2])
    {
        x1 = L1;
        if(A[L2]>A[R1])
            x2 = L2;
        else
            x2 = R1;
    }
    else
    {
        x1 = L2;
        if(A[L1]>A[R2])
            x2 = L1;
        else
            x2 = R2;
    }
}

```

移位运算

在C++中，整数/2实现为向零取整。
n>>1实现为n除以2向下取整。
例题：快速幂
求a的b次方对p取模的值，其中$1 \leq a, b, p \leq 10^9$.POJ 1995

```
int power(int a,int b,int p)
{
    int ans = 1 % p;
    for (; b;b>>=1)
    {
        if(b&1)
        {
            ans = (long long)ans * a % p;
        }
        a = (long long)a * a % p;
    }
    return ans;
}
```

如何将b用二进制表示？对b进行右移运算，取出b的低位。

```
int b = 11;
for(;b;b=(b>>1))
{
    printf("%d", b & 1);
}
```

异或运算符"^"也称XOR运算符。它的规则是若参加运算的两个二进位同号，则结果为0（假）；异号则为1（真）。即 0^0 = 0，0^1 = 1， 1^0=1，1^1 = 0。

汉诺塔

```
int Hanoi(int n)
{
    if(n==1)
        return 1;
    if(n==2)
        return 3;
    return Hanoi(n - 1) * 2 + 1;
}
int Hanoi4(int n)
{
    if(n==1)
        return 1;
    int min = 10000000;
    for (int i = 0; i < n;i++)
    {
        int tmp = 2 * Hanoi4(i) + Hanoi(n - i);
        if(tmp<min)
            min = tmp;
    }
    return min;
}
```

快速幂

求a的b次方对p取模的值，其中$1 \leq a, b, p \leq 10^9$。

```
int power(int a,int b,int p)
{
    int ans = p % 1;
    for (; b;b>>=1)
    {
        if(b&1)
            ans = (long long)ans * a % p;// 累成部分
        a = (long long)a * a % p;// 自乘部分
    }
    return ans;
}

```

矩阵快速幂，洛谷P3390

```
#include<cstdio>
#include<cstdlib>
```

```cpp
#include<cstring>
#define ll long long
using namespace std;
const int maxn=105;
const ll mod = 1e9 + 7;
ll n,k;
ll a[maxn][maxn];
ll c[maxn][maxn];
ll ans[maxn][maxn];
void mul()
{
    memset(c, 0, sizeof(c));
    for (int i = 0; i < n;i++)
        for (int j = 0; j < n;j++)
            for (int k = 0; k < n;k++)
            {
                c[i][j] = (c[i][j] + (ll)ans[i][k] * a[k][j]) % mod;
            }
    memcpy(ans, c, sizeof(c));
}
void mulself()
{
    memset(c, 0, sizeof(c));
    for (int i = 0; i < n;i++)
        for (int j = 0; j < n;j++)
            for (int k = 0; k < n;k++)
            {
                c[i][j] = (c[i][j] + (ll)a[i][k] * a[k][j]) % mod;
            }
    memcpy(a, c, sizeof(c));
}
int main()
{
    scanf("%lld%lld", &n, &k);
    for (int i = 0; i < n;i++)
        for (int j = 0; j < n;j++)
            scanf("%lld", &a[i][j]);
    for (int i = 0; i < n;i++)
        ans[i][i] = 1;
    for (; k; k >>= 1)
    {
        if (k & 1)
            mul();// 累成部分
        mulself();// 自乘部分
    }
    for (int i = 0; i < n;i++)
    {
        for (int j = 0; j < n-1; j++)
            printf("%lld ", ans[i][j]);
        printf("%lld\n", ans[i][n - 1]);
    }

        return 0;
}
```

二分

```
void erfen()
{
    //* 在闭区间[l,r]中查找大于等于x的最小值，x或x的后继
    while (l<r)
    {
        int mid = (l + r) >> 1;
        if(a[mid]>=x)
            r = mid;
        else
            l = mid + 1;
    }
    //* 在闭区间[l,r]中查找小于等于x的最大值，x或x的前驱

    while (l<r)
    {
        int mid = (l + r + 1) >> 1;
        if(a[mid]<=x)
            l = mid;
        else
            r = mid - 1;
    }
}

```

三分函数求极值

洛谷P3382 【模板】三分法

```
double f(double x)
{
    double ans = 0;
    for (int i = 0; i <= n;i++)
    {
        double tmp = 1;
        for (int j = 1; j <= i;j++)
            tmp *= x;
        ans += tmp * a[i];
    }
    return ans;
}
double three()
{
    double lmid, rmid;
    lmid = l + (r - l) / 3;//在函数域上取两个点lmid和rmid，分别取三等分点。
    rmid = l + (r - l) * 2 / 3;
    double eps = 1e-7;
    while (l+eps<r)
    {
        if(f(lmid)<f(rmid))// 极值点在lmid右侧
            l = lmid;
        else

```

```
            r = rmid;//极值点在rmid左侧
        lmid = l + (r - l) / 3;
        rmid = l + 2 * (r - l) / 3;
    }
    return l;//极值点
}

```

二分答案转化为判定

这是一个悲伤的故事

话说二分答案转判定真的应用范围极广啊啊，一定要扎实学会才行。其实也很简单的。

抽象-建模：

一个宏观的最优化问题也可以抽象为函数，其"定义域"是该问题下的可行方案，对这些可行方案进行评估得到的数指构成函数的"值域"，最优解就是评估值最优的方案。

借助二分，我们把求最优解的问题，转化为给定一个值mid，判定是否存在一个可行方案评分达到mid的问题。

一般能用二分解决的问题都有一个明显的标志，就是求最大值最小时的解或方案。

例题：

POJ2018

农场主约翰有N个牛圈，每个牛圈中有若干只奶牛。约翰想要连续的在牛圈之间造一些围栏，每个围栏至少可以包括F个牛圈。约翰想要问你如何设计围栏，可以使得在所有方案中，我们能够找到一个围栏中，牛圈里平均奶牛的数目最大。

二分答案，确定每个答案是否是可行解。

本题抽象成如下问题：

给定正整数数列A，求一个平均数最大的、长度不小于L的连续的子段。

思路：二分答案，判定 是否存在一个长度不小于L的子段，平均数不小于二分的值。

转化：我们把数列中每个数都减去二分的值，就转化为判定是否存在一个长度不小于L的子段，子段和大于等于0.

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn=1e5+100;
const int INF = 0x3f3f3f3f;
int n,m;
int F, N;

double a[maxn], b[maxn], sum[maxn];
```

```cpp
inline int read()
{
    int x = 0, f = 0, ch;
    while (!isdigit(ch=getchar()))
    {
        f |= ch == '-';
    }
    while (isdigit(ch))
    {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return f ? -x : x;
}
bool judge(double ave)
{
    for (int i = 1; i <= N;i++)
    {
        b[i] = a[i] - ave;
        sum[i] = sum[i - 1] + b[i];
    }
    double min_val = 1e6;
    double ans = -1e6;
    for (int i = F; i <= N;i++)
    {
        min_val = min(min_val, sum[i-F]);
        ans = max(ans, sum[i] - min_val);
    }
    return ans >= 0;
}
int main()
{
    scanf("%d%d", &N, &F);
    for (int i = 1; i <= N;i++)
    {
        scanf("%lf", &a[i]);
    }
    double l = 0, r = 1e6;
    double eps = 1e-5;
    while (r-l>eps)
    {
        double mid = (l + r) / 2;
        if(judge(mid))
        {
            l = mid;
        }
        else
            r = mid;
    }
    cout << int(r * 1000) << endl;
    return 0;
}
```

归并排序

在归并排序算法中，合并两个已排序数组的merge是整个算法的基础。我们要把包含n1个整数的数组L以及包含n2个整数的数组R合并到数组A中。现假设L与R中的元素都已按升序排列。我们不能直接将L和R连接起来套用普通的排序算法，而是要利用他们已经被排序的性质，借助复杂度为$O(n_1+n_2)$的合并算法进行合并。
为了简化merge的实现，我们可以在L和R的末尾分别安插一个大于所有元素的标记。在比较L和R元素的过程中，势必会遇到元素与标记相比较的情况，只要我们标记设置的足够大，且将比较次数限制在$n_1+n_2(right-left)$内，就可以既防止两个标记比较，又防止循环变量i，j分别超过n_1,n_2。
在merge处理中，由于两个待处理的局部数组都已经完成了排序，因此可以采用复杂度为$O(n_1+n_2)$的合并算法。
归并排序包含不相邻元素之间的比较，但并不会直接交换。在合并两个已排序数组时，如果遇到了相同元素只要保证前半部分数组优先于后半部分数组，相同元素的顺序就不会颠倒。因此归并排序属于稳定的排序算法。
归并排序除了数组保存数据占用空间以外，在递归调用的时候还需要占用额外的内存空间。

```
int n;
const int INF = 0x3f3f3f3f;
void merge(int A[],int left,int mid,int right)//right位置的元素是虚值
{
    int n1 = mid - left;
    int n2 = right - mid;
    int L[n1];
    int R[n2];
    for (int i = 0; i <= n1 - 1;i++)
    {
        L[i] = A[left + i];//[left,mid-1],total mid-left
    }
    for (int i = 0; i < n2;i++)
    {
        R[i] = A[mid + i];//[mid,right-1],total right-mid
    }
    L[n1] = INF;//如果不赋值为无穷大，会出错。由下面循环的比较可知，不这样赋值
    R[n2] = INF;//那么n2位置的元素也会被加入比较，并有可能小于L[i]而被排序
    int i = 0;//但是我们不需要排序n2位置的元素，因为此处的元素值未知。
    int j = 0;
    for (int k = left; k < right;k++)//right位置元素不算在内
    {
        if(L[i]<=R[j])
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
    }
}
void mergesort(int A[],int left,int right)//right位置元素不被排序
{
    if(left+1<right)
    {
        int mid = (left + right) / 2;
        mergesort(A,left, mid);
        mergesort(A, mid, right);
        merge(A, left, mid, right);//合并时A的左右两部分都已经有序
```

```
    }
}

int main()
{
    int  s[100];
    cin >> n;
    for (int i = 0; i < n;i++)
    {
        cin >> s[i];
    }
    mergesort(s, 0, n);
    for (int i = 0; i < n;i++)
    {
        cout << s[i] << " ";
    }
        return 0;
}
```

方法2

```
void Merge(int a[],int left,int mid,int right)
{
    int numele = right - left + 1;
    int lpos = left, rpos = mid+1, tpos = 1;
    int L[numele];
    while (lpos<=mid&&rpos<=right)
    {
        if(a[lpos]<a[rpos])
        {
            L[tpos++] = a[lpos++];
        }
        else
            L[tpos++] = a[rpos++];
    }
    while (lpos<=mid)
    {
        L[tpos++] = a[lpos++];
    }
    while (rpos<=right)
    {
        L[tpos++] = a[rpos++];
    }
    for (int i = 1; i <= numele;i++)
        a[i+left-1] = L[i];
}
void mergesort(int a[],int left,int right)
{
    if(left<right)
    {
        int mid = (left + right) / 2;
        mergesort(a, left, mid);
        mergesort(a, mid + 1, right);
        Merge(a, left, mid, right);
    }
}

```

快速排序

快速排序是在实践中最快的已知排序算法。该算法之所以特别快，主要是在于非常精炼而且高度优化的内部循环。

它的最坏情形的性能是$O(N^2)$。快速排序是一种分治算法。

快速排序的4步骤

- 如果S中元素个数是0或1，则返回
- 取S中任意元素v，称之为枢纽元
- 将S-{v}(S中其余元素)分成两个不相交的集合：
  $S_1 = \{x \in S - \{v\} | x \leq v\}$和$S_2 = \{x \in S - \{v\} | x \geq v\}$。
- 返回$\{quicksort(S_1)$后，继而v,继而$quicksort(S_2)\}$。

快速排序需要递归的解决两个子问题并线性的完成附加工作。与归并排序不同，两个子问题并不保证具有相等的大小。快速排序更快的原因是在第三步，第三步的分割实际上是在适当的位置进行并且非常有效，他的高效大大弥补了大小不等的递归调用的缺憾。

枢纽元的选择通常采用三数中值分割法。一般使用左端，右端和中心位置的三个元素的中值作为枢纽元。消除了预排序的输入情形。

分割策略

- 第一步是通过将枢纽元与最后的元素交换使得枢纽元离开要被分割的数据段。i从第一个元素开始而j从倒数第二个元素开始。在我们的程序中，采用三数中值分割法，第一个元素是左右两端点中最小的，一定小于枢纽元，最右端最后一个元素是一定大于枢纽元。我们把这两个元素作为警戒标志，确保i和j不会越界。所以实际上i从第二个元素也就是left+1开始，j从倒数第三个元素也就是right-2开始。
- 在分割阶段就是要把所有小元素移到数组的左边而把所有大元素移到数组的右边。小和da是相对于枢纽元而言的。
- 当i在j的左边时，我们将i右移，移过那些小于枢纽元的元素，并将j左移，移过那些大于枢纽元的元素。当i和j停止时，i指向一个大元素而j指向一个小元素。如果i在j的左边，我们就将i和j互换；这样就把一个大元素移到右边而把一个小元素移到左边。重复该过程直到i和j交错为止。
- 当i和j交错，就不再交换。分割的最后一步是将枢纽元与i指向的元素互换。这样枢纽元左边就都是小于它的元素，枢纽元右边就都是大于它的元素。
- 注意，如果元素等于枢纽元，我们也要把等于枢纽元的元素视为大于枢纽元或小于枢纽元的元素同样进行交换。

小数组

对于很小的数组，一般N=10，快速排序不如插入排序好。因此我们对于小数组不采用快速排序，而是采用插入排序。

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
```

```cpp
const int maxn=1e5+100;
int A[maxn];
int n,m;
int median3(int a[],int left,int right)
{
    int mid = (left + right) >> 1;
    if(a[left]>a[mid])
        swap(a[left], a[mid]);
    if(a[left]>a[right])
        swap(a[left], a[right]);
    if(a[mid]>a[right])
        swap(a[mid], a[right]);
    swap(a[mid], a[right - 1]);
    return a[right - 1];

}
void qsort(int a[],int left,int right)
{
    int i, j;
    int pivot;
    if(left+10<=right)
    {
        pivot = median3(a, left, right);
        i = left, j = right - 1;
        for (;;)
        {
            while (a[++i]<pivot){}
            while(a[--j]>pivot){}
            if(i<j)
            {
                swap(a[i], a[j]);
            }
            else
                break;
        }
        swap(a[i], a[right - 1]);
        qsort(a, left, i - 1);
        qsort(a, i+1, right);
    }
    else
    {
        int tmp, j;
        for (int i = left + 1; i <= right;i++)
        {
            tmp = a[i];
            for (j = i; j > left && a[j - 1] > tmp;j--)
                a[j] = a[j - 1];
            a[j] = tmp;
        }
    }
}
int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &A[i]);
    qsort(A, 1, n);
    for (int i = 1; i < n;i++)
```

```
        printf("%d ", A[i]);
    printf("%d\n", A[n]);

    return 0;
}
```

离散化

知识点总结：
离散化是把无穷大集合中的若干个元素映射为有限集合以便于统计的方法。例如在很多情况下，问题的范围虽然定义在整数集合上，但是只涉及其中M个有限数值，并且与数值的绝对大小无关(只把这些数值作为代表，或只与它们的相对顺序有关)。此时，我们就可以把整数集合Z中的这M个整数与1~M建立映射关系。离散化后该算法的时空复杂度降低为与m有关。
具体实现方法：
我们可以把a数组排序并去掉重复的数值，得到有序数组b[1]~b[m].在b数组的下标i与数值b[i]之间建立映射关系。若要查询整数i($1 \leq i \leq m$)代替的数值，只需返回b[i]；若要查询整数a[j]被哪个1~m之间的整数代替，只需在数组b中二分查找a[j]的位置即可。

```
void discrete()
{
    srot(a + 1, a + n + 1);
    for (int i = 1; i <= n;i++)
    {
        if(i==1||a[i]!=a[i-1])
            b[++m] = a[i];
    }
}
int query(int x)
{
    return lower_bound(b + 1, b + m + 1, x) - b;
}
//? 统计a数组中每个数出现的次数
void stat()
{
    for (int i = 1; i <= n;i++)
    {
        int v = a[i];
        c[query(v)]++;
    }
}
```

离散化经典例题：Cinema，洛谷CF670C Cinema

```
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;
const int maxn = 2e5 + 100;
int a[maxn];
int b[3*maxn];
int cnt;
int voice[maxn];
int word[maxn];
int c[3*maxn];
int d[3 * maxn];
```

```cpp
int t;
int main()
{
    int n, m;
    scanf("%d", &n);
    for (int i = 1; i <= n;i++)
    {
        scanf("%d", &a[i]);
        d[++t] = a[i];
    }
    scanf("%d", &m);
    for (int i = 1; i <= m;i++){
        scanf("%d", &voice[i]);
        d[++t]=voice[i];
    }
    for (int i = 1; i <= m;i++){
        scanf("%d", &word[i]);
        d[++t]=word[i];
    }
    sort(d + 1, d + t + 1);
    for (int i = 1; i <= t;i++)
    {
        if(i==1||d[i]!=d[i-1])
            b[++cnt] = d[i];
    }
    for (int i = 1; i <= n;i++)
    {
        int v = a[i];
        int p = lower_bound(b + 1, b + cnt + 1, v) - b;
        c[p]++;
    }
    int maxx = -1;
    int ans = -1;
    int tmpans = -1;
    for (int i = 1; i <= m;i++)
    {
        int v = voice[i];
        int p = lower_bound(b + 1, b + cnt + 1, v) - b;
        if(c[p]>maxx)
        {
            maxx = c[p];
            ans = i;
            int vv = word[i];
            int pp = lower_bound(b + 1, b + cnt + 1, vv) - b;
            tmpans = c[pp];
        }
        else if (c[p]==maxx)
        {
            int vv = word[i];
            int pp = lower_bound(b + 1, b + cnt + 1, vv) - b;
            if(c[pp]>tmpans)
            {
                maxx = c[p];
                ans = i;
                tmpans = c[pp];
            }
        }
    }
```

```
    printf("%d\n", ans);
    return 0;
}
```

luogu P1496 火烧赤壁

输入格式

第一行一个整数 N。

以后 N 行，每行两个数：A_i, B_i，表示连环线上着火船只的起始位置和终点。

输出格式

输出着火船只的总长度。**保证答案在 32 位带符号整数的表示范围内。**

输入输出样例

输入 #1 复制

```
3
-1 1
5 11
2 9
```

输出 #1 复制

```
11
```

AC 代码

思路：

对起点和终点两个数组分别排序。覆盖情况只有两种：一个区间完全覆盖另一个区间和两个区间有部分重合。前者可以转换为后者。因此我们可以直接对起点数组和终点数组排序，然后ans加上终点-起点的距离，再减去重合的部分。

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn=1e5+100;

int n,m;
int a[maxn], b[maxn];
int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n;i++)
    {
        scanf("%d%d", &a[i], &b[i]);
    }
    sort(a + 1, a + n + 1);
    sort(b + 1, b + n + 1);
    long long ans = 0;
    for (int i = 1; i <= n;i++)
    {
```

```
        ans += b[i] - a[i];
        if(i>=2&&(b[i-1]-a[i])>0)
            ans = ans - (b[i - 1] - a[i]);
    }
    printf("%lld\n", ans);
    return 0;
}
```

离散化方法：

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn=1e5+100;

int n,m;
int a[maxn], b[maxn], c[maxn];
int vis[maxn];
int main()
{
    scanf("%d",&n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d%d", &a[i], &b[i]);
        c[++m] = a[i];
        c[++m] = b[i];
    }
    sort(c + 1, c + m + 1);
    int cnt = unique(c + 1, c + m + 1) - (c + 1);
    for (int i = 1; i <= cnt;i++)
    {
        a[i] = lower_bound(c + 1, c + cnt + 1, a[i]) - c;
        b[i] = lower_bound(c + 1, c + cnt + 1, b[i]) - c;
        vis[a[i]]++;// 起点+1
        vis[b[i]]--;// 终点-1
    }
    int tmp = 0;
    long long ans = 0;
    for (int i = 1; i <= cnt;i++)
    {
        tmp += vis[i];
        if(tmp>0){ // c[i]可以作为起点
            ans += c[i + 1] - c[i];
        }
    }
    printf("%lld\n", ans);
    return 0;
}
```

树状数组与离散化

```cpp
#include<cstdio>
#include<algorithm>
#include<iostream>
#define lowbit(i) i&-i
#define ll long long
using namespace std;

const int maxn = 5e5 + 100;
int n, a[maxn], b[maxn], c[maxn];
inline int read()
{
    int x = 0, ch = getchar();
    while (!isdigit(ch))
    {
        ch = getchar();
    }
    while (isdigit(ch))
    {
        x = x * 10 + ch - '0', ch = getchar();
    }
    return x;
}
inline void add(int x)
{
    for (int i = x; i <= n;i+=lowbit(i))
        c[i]++;
}
inline int ask(int x)
{
    int ans = 0;
    for (; x;x-=lowbit(x))
        ans += c[x];
    return ans;
}
int main()
{
    n = read();
    for (int i = 1; i <= n;i++)
        a[i] = b[i] = -read();
    sort(b + 1, b + n + 1);
    for (int i = 1; i <= n;i++)
        a[i] = lower_bound(b + 1, b + n + 1, a[i]) - b;
    ll ans = 0;
    for (int i = 1; i <= n;i++)
    {
        add(a[i]);
        ans += ask(a[i] - 1);
    }
    printf("%lld\n", ans);
    return 0;
}
```

排序

对顶堆

堆这种数据结构可以用来动态维护某种具有单调性的东西。对顶堆是指有两个堆，一个大根堆，一个小根堆，大根堆中维护一个递降序列，小根堆中维护一个递增序列。小根堆的堆顶或者大根堆的堆顶就是我们要的答案。
以下以两个例子来说明；
1.动态维护中位数问题：
依次读入一个整数序列，每当已经读入的整数个数为奇数时，输出已读入的整数构成的序列的中位数。
2.输出第K小的数(大根堆)或第K大的数(小根堆)

1.为了动态维护中位数，我们可以建立两个二叉堆：一个小根堆，一个大根堆。在依次读入这个整数序列的过程中，设当前序列的长度为M，我们始终保持：
1.序列中从小到大排名为1~M/2的整数存储在大根堆中；
2.序列中从小到大排名M/2+1~M的整数存储在小根堆中。
任何时候，如果某一个堆中元素个数过多，打破了这个性质，就取出该堆的堆顶插入另一个堆，这样一来，序列的中位数就是小根堆的堆顶。
每次读入一个新数值X后，若X比中位数小，则插入大根堆，否则插入小根堆，再插入之后检查并维护上述性质即可。这就是对顶堆算法。

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
#include<vector>
using namespace std;
const int maxn = 1e4 + 100;
int P, M;
int a[maxn];
int ans[maxn];
priority_queue<int> qmax;
priority_queue<int, vector<int>, greater<int> > qmin;

int main()
{
    scanf("%d", &P);
    while (P--)
    {
        memset(a, 0, sizeof(a));
        memset(ans, 0, sizeof(ans));
        int num, tot;
        while (qmin.size())
        {
            qmin.pop();
        }
        while (qmax.size())
        {
            qmax.pop();
        }
        scanf("%d%d", &num, &tot);
        printf("%d %d\n", num, (tot + 1) >> 1);
        int cnt = 0;
        for (int i = 1; i <= tot;i++)
        {
```

```
            int x;
            scanf("%d", &x);
            if(i==1)
            {
                qmin.push(x);
            }
            else if(x<qmin.top())
                qmax.push(x);
            else
                qmin.push(x);
            while (qmax.size()<(i/2))
            {
                int v = qmin.top();
                qmin.pop();
                qmax.push(v);
            }
            while (qmin.size()<(i+1)/2)
            {
                int v = qmax.top();
                qmax.pop();
                qmin.push(v);
            }
            if(i&1)
            {
                ans[++cnt] = qmin.top();
            }
        }
        for (int i = 1; i <= cnt;i++)
        {
            if(i%10==0||i==cnt)
                printf("%d\n", ans[i]);
            else
                printf("%d ", ans[i]);
        }
    }
    return 0;
}
```

倍增

ST算法

上代码

```
void ST_prework()
{
    for (int i = 1; i <= n;i++)
        f[i][0] = a[i];
    int t = log(n) / log(2) + 1;
    for (int j = 1; j < t; j++)
        for (int i = 1; i <= n - (1 << j) + 1;i++)
        {
            f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
        }
}
int ST_query(int l,int r)
```

```
{
    int k = log(r - l + 1) / log(2);
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
```

堆排序

并查集

```
int fa[x];
//并查集的初始化
for(int i=1;i<=n;i++)
    fa[i]=i;
//路径压缩的查找,查找x的父节点
int get(int x)
{
    if(fa[x]==x)
        return x;
    return fa[x]=get(fa[x]);
}
//合并
void merge(int x,int y)
{
    fa[get(x)]=get(y);
}
```

单调栈

背景：求直方图中矩形面积的最大值
思路：我们维护的轮廓(栈中)是一个高度始终单调递增的矩形序列。
具体实现：
我们建立一个栈，用来保存若干个矩形。这些矩形的高度是单调递增的。我们从左到右依次扫描每个矩形:
1.如果当前矩形比栈顶矩形高，直接进栈。
2.否则不断取出栈顶，直至栈为空或者栈顶矩形高度比当前矩形小。在出栈的过程中，我们累计被弹出的矩形的宽度之和，并且每弹出一个矩形，就用它的高度乘以当前累计的宽度去更新答案。整个出栈过程结束后，我们把一个高度为当前矩形高度、宽度为累计值的新矩形入栈。
3.整个扫描结束后，我们把栈中剩余的矩形依次弹出，按照与上面相同的方法更新答案。为了简化程序实现，也可以增加一个高度为0的矩形a[n+1]，以避免在扫描结束后栈中有剩余矩形。

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

const int maxn = 1e4 + 5;
int p;//stack pointer
int a[maxn];//height
int s[maxn];//stack
int w[maxn];//width,此处默认每个矩形的宽度都是1
int ans;
int main()
```

```
{
    int n;
    cin >> n;
    a[n + 1] = 0;
    for (int i = 1; i <= n + 1;i++)
    {
        if(a[i]>s[p])
        {
            s[++p] = a[i];
            w[p] = 1;
        }
        else
        {
            int width = 0;
            while (s[p]>a[i])
            {
                width += w[p];
                ans = max(ans, (long long)width * s[p]);
                p--;
            }
            s[++p] = a[i];
            w[p] = width + 1;
        }
    }
}
```

单调队列

最大连续子序列和
给定一个长度为N的整数序列(可能有负数)，从中找出一段长度不超过M的连续子序列，使得子序列中所有数的和最大。$N, M \leq 3 * 10^5$。
计算区间和的问题，一般转化为两个前缀和相减的形式进行求解。我们先求出$S[i]$表示数组里前i项的和，则连续子序列[L,R]中数的和可以表示为$S[R]-S[L-1]$.那么原问题可以转化为：找出两个位置x，y,使$S[y]-S[x]$最大并且$y-x \leq M$。
首先我们枚举右端点i，当i固定时，问题就变为：找到一个左端点j，其中$j \in [i-m, i-1]$并且$S[j]$最小。
不妨比较一下任意两个位置j和k，如果$k<j<t$并且$S[k] \geq S[j]$，那么对于所有大于等于i的右端点,k永远不会成为最优选择。因为不但$S[k]$不小于$S[j]$，而且j离i更近，长度更不容易超过M，即j的生存能力比k更强。所以当j出现以后，k就完全是一个无用的位置。
以上比较告诉我们，可能成为最优选择的策略集合一定是一个“下标位置递增、对应的前缀和S的值也递增”的序列。我们可以用一个队列保存这个序列。随着右端点从前向后扫描，我们对每个i执行以下三个步骤：
1.判断队头决策与i的距离是否超过M的范围，若超过则出队。
2.此时队头就是右端点为i是，左端点j的最优选择。
3.不断删除队尾决策，直到队尾对应的S值小于$S[i]$.然后把i作为一个新的决策入队。

```
    int l = 1, r = 1;
    int q[maxn];
    q[1] = 0;//save choice j=0
    for (int i = 1; i <= n;i++)
    {
        while(l<=r&&q[l]<i-m)
            l++;
        ans = max(ans, sum[i] - sum[q[l]]);
        while (l<=r&&sum[i]<=sum[q[r]])
```

```
        {
            r--;
        }
        q[++r] = i;
    }
```

数组模拟链表以及邻接表的实现

```
struct Node
{
    int value;
    int prev, next;
} node[size];
int head, tail, tot;
int initialize()
{
    tot = 2;
    head = 1, tail = 2;
    node[head].next = tail;
    node[tail].prev = head;
}
int insert(int p,int val)//在p结点的位置后面插入新节点
{
    int q = ++tot;
    node[q].value = val;
    node[node[p].next].prev = q;
    node[q].next = node[p].next;
    node[p].next = q;
    node[q].prev = p;
}
void remove(int p)//插入p
{
    node[node[p].prev].next = node[p].next;
    node[node[p].next].prev = node[p].prev;
}

void clear()
{
    memset(node, 0, sizeof(node));
    head = tail = tot = 0;
}
//数组模拟链表存储一张带权有向图的邻接表结构
//加入有向边(x,y)，权值为z
void add(int x,int y,int z)
{
    ver[++tot]=y;
    edge[tot]=z;
    next[tot]=head[x];
    head[x]=tot;
}
//访问从x出发的所有边
for(int i=head[x];i;i=next[i])
{
    int y=ver[i];
    int z=edge[i];
    //找到一条有向边，权值为z
```

```
}


```

邻接表的数组模拟链表的实现
长度为n的head数组记录了从每个节点出发的第一条边在ver和edge数组中的存储位置，长度为m的边集数组ver和edge记录了每条边的终点和边权，长度为m的数组next模拟了链表指针，表示从相同节点出发的下一条边在ver和edge中的存储位置。以下用数组模拟链表的方式存储了一张带权有向图的邻接表结构。

```
//建图
void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    next[tot] = head[x];
    head[x] = tot;
}
//访问从x出发的所有有向边
for (int i = head[x]; i;i=next[i])
{
    int y = ver[i];
    z = edge[i];
}
```

对于无向图，我们把每条无向边看作两条有向边插入即可。有一个小技巧是，结合在第0x01节提到的“成对变换”的位运算性质，我们可以在程序最开始的时候，初始化变量tot=1.这样每条无向边看成的两条有向边会成对存储在ver和edge数组的下标“2和3” “4和5” “6和7” . . .的位置上。通过对下标进行xor1的运算，就可以直接定位到与当前边相反的边。

Hash表

Hash表又称散列表，一般由Hash函数与链表结构共同实现。与离散化思想类似，当我们需要若干复杂信息进行统计时，可以用Hash函数把这些复杂信息映射到一个容易维护的

有一种称为开散列的解决方案可以解决映射冲突的问题。建立一个邻接表结构，以Hash函数的值域作为表头数组head，映射后的值相同的原始信息被分到同一类，构成一个链表接到对应的表头之后，链表的结点上可以保存原始信息和一些统计数据。

Hash表的两个基本操作：

> 1.计算Hash函数的值
>
> 2.定位到对应链表中依次遍历、比较

POJ3349

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<vector>
using namespace std;
const int maxn = 1e6 + 100;
int snow[maxn][6];
int n, tot;
```

```
const int p = 999991;
int head[maxn];
int nexts[maxn];
vector<int> Ha[p];
int H(int *a)
{
    int sum = 0;
    int mul = 1;
    for (int j = 0; j < 6;j++)
    {
        sum += a[j];
        sum %= p;
        mul = (long long)mul * a[j] % p;
    }
    int ans = (sum + mul) % p;
    return ans;
}
bool equal(int *a,int *b)//判断两个环逆时针或顺时针记录结果是否相等
{
        for (int j = 0; j < 6;j++)
        {
            bool eq = 1;
            for (int k = 0; k < 6;k++)
            {
                if(a[(k)%6]!=b[(j+k)%6])//顺时针比较
                    eq = 0;
            }
            if(eq)
                return 1;
            eq = 1;
            for (int k = 0; k < 6;k++)
            if(a[(k)%6]!=b[(j-k+6)%6])//逆时针比较
                eq = 0;
            if(eq)
                return 1;
        }
    return 0;
}

bool insert(int *a,int q)
{
    int val = H(a);
    for (int i = 0; i < Ha[val].size();i++)
    {
        int u = Ha[val][i];
        if(equal(snow[u],a))
            return 1;
    }
    Ha[val].push_back(q);
    return 0;
}
int main()
{
    cin >> n;
    for (int i = 1; i <= n;i++)
    {
        for (int j = 0; j < 6;j++)
            scanf("%d", &snow[i][j]);
```

```
        if(insert(snow[i],i))
        {
            printf("Twin snowflakes found.\n");
            return 0;
        }
    }
    printf("No two snowflakes are alike.\n");
    return 0;
}
```

字符串哈希

下面介绍的字符串hash函数把一个任意长度的字符串映射成一个非负整数，并且其冲突的概率几乎为零。

取一个固定值P，把字符串看作P进制数，并分配一个大于0的数值，代表每种字符。一般来说，我们分配的数值都远小于P。例如，对于小写字母构成的字符串，可以令a=1,b=2,...,z=26。取一固定M值，求出该P进制数对M的余数，作为该字符串的hash值。

一般来说，我们取P=131或P=13331，此时hash值产生冲突的概率极低，只要hash值相同，我们就可以认为原字符串是相等的。通常我们取$M = 2^{64}$,即直接使用unsigned long long类型存储这个hash值，在计算时不处理算数溢出。

字符串匹配

KMP模式匹配

算法步骤:

贪心算法

奶牛的日光浴
每一头奶牛能够承受的阳光强度有最大值和最小值。一开始每一头奶牛都要涂防晒霜。涂完以后奶牛的承受阳光强度会固定在一个值上。防晒霜有L种，每种有一个固定阳光强度和数量。问如何给奶牛涂防晒霜，可以使得尽量多的奶牛享受阳光浴。
我们对每一头奶牛的minspf按照递减顺序排列。顺序扫描奶牛序列。对一头奶牛，它后面所有奶牛的minspf都不会大于它。每一个不低于当前奶牛minspf的防晒霜，都不会低于后面的奶牛的minspf。如果有两瓶防晒霜x和y可以被当前奶牛使用，并且$spf[x] \leq spf[y]$。那么对于后面的奶牛，有以下三种情况：可以用x和y；可以用x，不可以用y；x和y都不可以用。因此对于当前奶牛，选择尽量大的可以接受的防晒霜是一个更优的选择。

树状数组

P1972 [SDOI2009]HH的项链

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
```

```cpp
#define lowbit(i) i&-i
using namespace std;
const int maxn = 1e6 + 10;
int n,m;
int a[maxn], v[maxn], c[maxn], ans[maxn];
struct rec{
    int l, r;
    int pos;// 记录被第几个读入
} query[maxn];
inline int read()// 快读优化
{
    int x = 0, f = 0, ch;
    while (!isdigit(ch=getchar()))
    {
        f |= ch == '-';
    }
    while (isdigit(ch))
    {
        x = x * 10 + ch - '0', ch = getchar();
    }
    return f ? -x : x;
}
bool cmp(rec&a,rec &b)// 将读入的查询区间按照右端点升序排列
{
    return a.r < b.r;
}
inline void add(int x,int y)// 树状数组维护的是1~x区间内不同数的个数，
//第j个位置出现了一个新的数就是1，否则是0
{
    for (int i = x; i <= n; i += lowbit(i))
        c[i] += y;
}
inline int ask(int x)
{
    int ans = 0;
    for (int i = x; i;i-=lowbit(i))
        ans += c[i];
    return ans;
}
int main()
{
    n = read();
    for (int i = 1; i <= n;i++)
        a[i] = read();
    m = read();
    for (int i = 1; i <= m;i++)
    {
        query[i].l = read(), query[i].r = read();
        query[i].pos = i;
    }
    int next = 1;
    sort(query + 1, query + m + 1, cmp);
    for (int i = 1; i <= m;i++){
        for (int j = next; j <= query[i].r;j++)
        {
            if(v[a[j]])// 若a[j]在此之前出现过，删去之前出现过的位置，
            //维护一下树状数组
            {
```

```
                add(v[a[j]], -1);
            }
            add(j, 1);// j号位置由0 -> 1,
            //表示此处出现了一个之前没有出现过的数(有也被删掉了)
            v[a[j]] = j;//标记一下a[j]在什么位置出现了
        }
        next = query[i].r + 1;
        ans[query[i].pos] = ask(query[i].r) - ask(query[i].l - 1);
    }
    for (int i = 1; i <= m;i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

P3372 【模板】线段树 1

```
#include<cstdio>
#include<algorithm>
using namespace std;
#define ll long long
const int maxn = 1e5 + 100;
ll c1[maxn];
ll a[maxn];
ll c2[maxn];
ll n, m;
ll ask(ll x)
{
    ll ans = 0;
    ll t = x;
    for (; x;x-=x&-x){
        ans += t * c1[x] - c2[x];
    }
    return ans;
}
void add(ll x,ll y)
{
    ll t = x;
    for (; x <= n;x+=x&-x){
        c1[x] += y;
        c2[x] += y*(t - 1);
    }
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n;i++)
    {
        scanf("%lld", &a[i]);
        add(i,a[i]-a[i-1]);//维护差分数组

    }
    while (m--)
    {
        ll sym, x, y, k;
        scanf("%lld%lld%lld", &sym,&x,&y);
        if(sym==1)
        {
```

```
            scanf("%lld", &k);
            add(x, k);
            add(y + 1, -k);
        }
        else
        {
            printf("%lld\n", ask(y) - ask(x - 1));
        }
    }
    return 0;
}
```

线段树解法

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#define ll long long
using namespace std;
const int maxn=1e5+100;

struct SegmentTree{
    int l, r;
    ll sum, add;
    #define l(x) tree[x].l
    #define r(x) tree[x].r
    #define sum(x) tree[x].sum
    #define add(x) tree[x].add
} tree[maxn * 4];
int a[maxn], n, m;
void build(int p,int l,int r)
{
    l(p) = l, r(p) = r;
    if(l==r){
        sum(p) = a[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(p << 1, l, mid);
    build(p << 1 | 1, mid + 1, r);
    sum(p) = sum(p << 1) + sum(p << 1 | 1);
}
void spread(int p)
{
    if(add(p))
    {
        sum(p << 1) += add(p) * (r(2 * p) - l(2 * p) + 1);
        sum(p << 1 | 1) += add(p) * (r(p << 1 | 1) - l(p << 1 | 1) + 1);
        add(p << 1) += add(p);
        add(p << 1 | 1) += add(p);
        add(p) = 0;
    }
}
void change(int p,int l,int r,int d)
```

```
{
    if(l<=l(p)&&r>=r(p)){
        sum(p) += (ll)d * (r(p) - l(p) + 1);
        add(p) += d;
        return;
    }
    spread(p);
    int mid = (l(p) + r(p)) >> 1;
    if(l<=mid)
        change(p << 1, l, r,d);
    if(r>mid)
        change(p << 1 | 1, l, r, d);
    sum(p) = sum(p << 1) + sum(p << 1 | 1);
}
ll ask(int p,int l,int r)
{
    if(l<=l(p)&&r>=r(p))
    {
        return sum(p);
    }
    spread(p);
    int mid = (l(p) + r(p)) >> 1;
    ll val = 0;
    if(l<=mid)
    {
        val += ask(p << 1, l, r);
    }
    if(r>mid)
        val += ask(p << 1 | 1, l, r);
    return val;
}
inline int read()
{
    int x = 0, f = 0, ch;
    while (!isdigit(ch=getchar()))
    {
        f |= ch == '-';
    }
    while (isdigit(ch))
    {
        x = x * 10 + ch - '0', ch = getchar();
    }
    return f ? -x : x;
}
int main()
{
    n = read(), m = read();
    for (int i = 1; i <= n;i++)
    {
        a[i] = read();
    }
    build(1, 1, n);
    while (m--)
    {
        int opt, x, y, k;
        opt = read();
        if(opt==1)
        {
```

```
            x = read(), y = read(), k = read();
            change(1, x, y, k);
        }
        else
        {
            x = read(), y = read();
            printf("%lld\n", ask(1, x, y));
        }
    }
        return 0;
}
```

字符串

KMP模式匹配

搜索专题

树与图的深度优先遍历，树的DFS序，深度和重心。

```
void dfs(int x)
{
    vis[x] = ++n;
    for (int i = head[x]; i;i=next[i])
    {
        int y = ver[i];
        if(vis[y])
            continue;
        dfs(y);
    }
}
```

```
//树的dfs序。一般来讲，我们在对树进行深度优先遍历时，对于每个节点，在刚进入递归后以及即将回溯前各记录一次该点的编号，最后产生的长度为2N的结点序列就称为树的dfs序
void dfs(int x)
{
    a[++m] = x;//a数组存储dfs序
    v[x] = 1;  //记录点x被访问过
        for (int i = head[x]; i;i=next[i])
        {
            int y = ver[i];
            if(v[y])
                continue;
            dfs(y);
        }
    a[++m] = x;
}
```

记录树中每个结点x的大小，求出树的重心

```
int ans;
```

```cpp
int pos;
void dfs(int x)
{
    v[x] = 1;  //记录点x被访问过
    size[x] = 1;
    int max_part = 0;
    for (int i = head[x]; i; i = next[i])
    {
        int y = ver[i];
        if (v[y])
            continue;
        dfs(y);//先dfs，回溯的时候进行统计。否则子树的大小还未知
        size[x] += size[y];
        max_part = max(max_part, size[y]);
    }
    max_part = max(max_part, n - size[x]);//n为整棵树的结点数目
    if(max_part<ans)
    {
        ans = max_part;//全局变量ans记录了重心对应的max_part值
        pos = x;//全局变量pos记录了重心
    }
}
```

图的连通块的划分
cnt是图中连通块的个数，v数组标记了每个点各自属于哪个连通块

```cpp
int cnt;
void dfs(int x)
{
    v[x] = cnt;
    for (int i = head[x]; i;i=next[i])
    {
        int y = ver[i];
        if(v[y])
            continue;
        dfs(y);
    }
}
for (int i = 1; i <= n;i++)//在int main()中
{
    if(!v[i])
    {
        cnt++;
        dfs(i);
    }
}
```

对图进行广度优先遍历

```cpp
void bfs()
{
    memset(d, 0,sizeof(d));
    queue<int> q;
    q.push(1);
    d[1] = 1;// 节点的层次，从起点1走到x需要经过的最小点数
    while (q.size()>0)
```

```
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i;i=next[i])
        {
            int y = ver[i];
            if(d[y])
                continue;
            d[y] = d[x] + 1;
            q.push(y);
        }
    }
}
```

拓扑排序

若一个由图中所有点构成的序列A满足：对于图中的每条边(x,y)，x在A中都出现在y之前，则称A是该有向无环图顶点的一个拓扑序。求解序列A的过程就称为拓扑排序。
拓扑排序过程的思想：
我们只需要不断选择图中入度为0的结点x，然后把x连向的点的入度减一。结合广度优先搜索。
1.建立空的拓扑序列A。
2.预处理出所有点的入度deg[i]，起初把所有入度为0的点入队。
3.取出队头结点x，把x加入拓扑序列A的末尾。
4.对于从x出发的每条边(x,y)，把deg[y]减1.若被减为0，则把y入队。
5.重复3~4步直到队列为空，此时A即为所求。
拓扑序列可以判断有向图中是否存在环。我们可以对任意有向图执行拓扑排序，完成后检查A序列的长度。若A序列的长度小于图中点的数量，则说明某些点未被遍历，说明图中存在环。

```
#include<cstdio>
// #include<algorithm>
#include<cstring>
#include<queue>
#include<iostream>
const int maxn = 1e4 + 100;
int tot;
int ver[maxn];
int next[maxn], head[maxn];
int deg[maxn];
int a[maxn];
int cnt;
using std::cin;
using std::cout;
using std::queue;
int n, m;
void add(int x,int y)
{
    ver[++tot] = y;
    next[tot] = head[x];
    head[x] = tot;
    deg[y]++;
}

void topsort()
{
    queue<int> q;
```

```
    // ********************************************************************
    for (int i = 1; i <= n;i++)// 这个循环很重要，可以求解多连通拓扑排序问题
    {
        if(deg[i]==0)
            q.push(i);
    }
    //
********************************************************************
    while (!q.empty())
    {
        int x = q.front();
        q.pop();
        a[++cnt] = x;
        for (int i = head[x]; i;i=next[i])
        {
            int y = ver[i];
            deg[y]--;
            if(deg[y]==0)
                q.push(y);
        }
    }
}
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= m;i++)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        add(x, y);
    }
    topsort();
    for (int i = 1; i <= n;i++)
    {
        printf("%d ", a[i]);
    }
    cout << std::endl;
    return 0;
}
```

二进制状态压缩

二进制状态压缩，是指将一个长度为m的bool数组用一个m位二进制整数表示并存储的方法。

可以使用C++ STL的bitset模板库

bitset可以看作一个多位二进制数，每8位占用1个字节，相当于采用了状态压缩的二进制数组，并支持基本的位运算。在估计程序运行时间时，我们一般以32位整数的运算次数为基准，因此n位bitset执行一次位运算的复杂度可以视为$\frac{n}{32}$.

- 声明

bitset<1000> s;

表示一个1000位的二进制数，<>中填写位数。下面把位数记为n。

位运算操作符：

~s:返回对bitset s 按位取反的结果

&，|，^:返回对两个位数相同的bitset执行按位与、或、异或的运算结果

>>，<<返回把一个bitset左移、右移若干位的结果。

==，!=:比较两个bitset代表的二进制数是否相等。

- 【】操作符

s【k】表示s的第k位，可以取值或赋值。例如 s[k]=1

在10000位二进制数中，最低位为s[0]，最高位为s[9999]。

- 注意：如果想要把s的第k位赋值为1，可以写s[k]=1或s[k]=s[k]|1;.不要写s[k]|=1. 快速赋值号没有被重载。
- count：

s.count()返回有多少位为1

- any/none

若s的所有位都为0，则s.any() 返回false，s.none()返回true。

所s至少有1位为1，则s.any()返回true，s.none()返回false。

- set/reset/flip

s.set() 把s的所有位变为1.

s.set(k,v) 把s的第k位变为v，即s[k]=v.

s.reset()把s的所有位变为0；s.reset(k)把s的第k位变为0

s.flip() 把s的所有位取反，即s=~s

s.flip(k) 把s的第k位取反，即s

题目：可达性统计

给定一张N个点M条边的有向无环图，分别统计从每个点出发能够到达的点的数量。$N, M \leq 30000$.

```cpp
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
#include<bitset>
using namespace std;
const int maxn=3e4+100;

int n, m, tot;
int ver[maxn], Next[maxn], head[maxn], deg[maxn], sizes[maxn];
bitset<30010> f[maxn];
void add(int x,int y)
{
    ver[++tot] = y;
```

```
    Next[tot] = head[x];
    head[x] = tot;
    deg[y]++;
}
void topsort()
{
    queue<int> q;
    for (int i = 1; i <= n;i++)
    {
        if(deg[i]==0){
            q.push(i);
            f[i][i] = 1;
        }
    }
    while (q.size())
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i];
            deg[y]--;
            f[y] |= f[x];
            if(deg[y]==0){
                f[y][y] = 1;
                q.push(y);
            }
        }
    }
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m;i++)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        add(y, x);
    }
    topsort();
    for (int i = 1; i <= n;i++)
    {
        printf("%d\n", f[i].count());
    }
    return 0;
}
```

二叉堆

```
const int size = 1e5;
int heap[size];//堆的大小，利用数组层次式存储堆。因为堆本身是一棵完全二叉树
int n;//堆中结点的个数，堆的末尾结点的下标
void up(int p)//向上调整堆。这是一个大根堆
{
```

```
    while (p>1)
    {
        if(heap[p]>heap[p/2])//如果子节点大于父节点，交换孩子和父亲的值
        {
            swap(heap[p], heap[p / 2]);
            p /= 2;
        }
        else
        {
            break;
        }
    }
}
void insert(int val)//堆的插入，在数组末尾插入，逐渐向上调整
{
    heap[++n] = val;
    up(n);
}
int GetTop()//取出堆顶元素
{
    return heap[1];
}
void down(int p)//逐渐向下调整
{
    int s = p * 2;
    while (s<=n)
    {
        if(s<n&&heap[s]<heap[s+1])//选取左右孩子中较大的那一个，确保父节点大于两个孩子节点的值
            s++;
        if(heap[s]>heap[p])//如果孩子节点大于父节点，交换
        {
            swap(heap[s], heap[p]);
            p = s;
            s = 2 * p;
        }
        else
        {
            break;
        }
    }
}
void Extract()//把堆顶从二叉堆中移除。我们把堆顶heap[1]与存储在数组末尾的节点heap[n]交换，然后移除数组末尾的结点(令n减小1)，最后把堆顶通过交换的方式
//向下调整，直至满足堆性质。
{
    heap[1] = heap[n--];
    down(1);
}
void Remove(int k)//数组下标为k。把k位置的结点与数组末尾结点交换，n--，由于不确定应该向下调整还是向上调整，因此要都判断一下。
{
    heap[k] = heap[n--];
    up(k);
    down(k);
}
```

lowbit运算

lowbit(n)定义为非负整数n在二进制表示下“最低位的1及其后边所有的0”构成的数值。例如n=10的二进制表示为(1010)，lowbit(n)=(10)=2。下面推导lowbit的公式

关于BFS的一些重要问题

有关入队、出队顺序，是否把节点加入队列状态的判断

> 1.0-1边权的BFS，节点可以多次入队，因为可能被插入队头，也可能被插入到队尾，所以节点可以多次入队。因此vis数组的判断不能写在for循环里面。但是当节点第一次出队的时候，所得的路径长度是最短路径长度。但是for循环里面的更新类似于dijkstra等最短路算法的更新方式，只有更新后路径比以前更小才更新，也不是全更新。因为数组是全局数组。
> 2.对于边权都为1的模板BFS，有两种写法。如果开一个全局数组d[maxn][maxn]记录最短路径长度，那么每个点只能被更新一次，第一次更新的长度就是最短路。这个时候vis的判断要写在里面，或者用d来判断，d的初始值为0，为0的时候才更新，非零的时候不更新。
> 第二种写法是，可以在for循环的外面执行vis数组的判断。如果vis数组为1，那么该节点已经被访问过出队过，continue。否则记vis[node]=1。这个时候只要还未出队，那么就可以入队，因此节点可能会多次入队，但是出队一次。因此就不能设置全局数组记录d最短路了，因为每次入队都会改变全局数组的状态，除非当作最短路算法来写。这个时候，每个节点入队的同时，最短路长度也作为状态的一部分入队才可以。
> 3.对于权值可变的正常最短路问题，例如dijkstra算法可以解决的问题，都是多次入队，但是出队一次。这个时候。只有当最短路被更新后更小，才让这个被更新后的节点入队。也就是只有当节点被优化以后才可以入队，否则不可以入队。vis的判断写在for循环的外面。

0-1 BFS

题目来源：洛谷P4667 [BalticOI 2011 Day1]Switch the Lamp On
AC代码

```
#include<cstdio>
#include<queue>
#include<deque>
#include<algorithm>
#include<iostream>
using namespace std;
const int maxn = 510;
char s[maxn][maxn];
struct rec{
    int x, y;
};
int n, m;
int d[maxn][maxn];
int dx[4] = {-1, 1, -1, 1};
int dy[4] = {-1, 1, 1, -1};
deque<rec> q;
void init()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n;i++)
    {
        cin >> (s[i] + 1);
    }
    if((n&1)!=(m&1))
```

```
        printf("NO SOLUTION\n");
    for (int i = 1; i <= n+1;i++)
        for (int j = 1; j <= m+1;j++)
            d[i][j] = -1;
}
bool valid(int x,int y)
{
    return x >= 1 && y >= 1 && x <= n + 1 && y <= m + 1;
}
void bfs(int a,int b)
{
    rec st;
    st.x = a, st.y = b;
    d[a][b] = 0;
    q.push_back(st);
    while (!q.empty())
    {
        rec now = q.front();
        q.pop_front();
        rec next;
        if(d[n+1][m+1]!=-1){
            printf("%d\n",d[n+1][m+1]);
            return;
        }
        for (int i = 0; i < 4;i++)
        {
            next.x = now.x + dx[i];
            next.y = now.y + dy[i];
            // if(!valid(next.x,next.y)||d[next.x][next.y]!=-1)
            //     continue;
            if(valid(next.x,next.y)){
            int x = min(next.x, now.x);
            int y = min(next.y, now.y);
            if(i>=2)
            {
                if(s[x][y]=='/'){
                    if(d[next.x][next.y]==-1||d[next.x][next.y]>d[now.x]
[now.y])
                    {d[next.x][next.y] = d[now.x][now.y];
                    q.push_front(next);}
                }
                else
                {
                    if(d[next.x][next.y]==-1||d[next.x][next.y]>d[now.x]
[now.y]+1)
                    {d[next.x][next.y] = d[now.x][now.y]+1;
                    q.push_back(next);}
                }
            }
            else
            {
                if(s[x][y]=='\\')
                {
                    if(d[next.x][next.y]==-1||d[next.x][next.y]>d[now.x]
[now.y])
                    {d[next.x][next.y] = d[now.x][now.y];
                    q.push_front(next);}
                    // if(next.x==n+1&&next.y==m+1)
```

```
                    //     return d[next.x][next.y];
                }
                else
                {
                    if(d[next.x][next.y]==-1||d[next.x][next.y]>d[now.x][now.y]+1)
                    {d[next.x][next.y] = d[now.x][now.y]+1;
                    q.push_back(next);}
                    // if(next.x==n+1&&next.y==m+1)
                    //     return d[next.x][next.y];
                }
            }
        }
    }
}
}
int main()
{
    init();
    bfs(1, 1);
    return 0;
}
```

这道题为啥我一开始没有AC，原因如下：
在这个问题中，一个点可以入队多次。也就是可以被更新多次。后面更新可能使得该点的路径更短，所以不能让这个点只更新一次就不更新了。我写的错误代码是：

```
    if(!valid(next.x,next.y)||d[next.x][next.y]!=-1)//在for循环中
        continue;
```

这就导致了，一个点只能被更新一次，是错误的写法。
如果真的想优化代码，应该把握住BFS队列满足两段性和单调性。因此一个点虽然可以入队多次，多次被更新，但是当它第一次出队的时候，d数组中存储的就是它的最短路径。

优先队列BFS

对于更加具有普适性的情况，也就是每次扩展都有各自不同的“代价”时，求出起始状态到每个状态的最小代价，就相当于在一张带权图上求出从起点到每个节点的最短路。此时我们有两个解决方案：
1.仍然使用一般的广搜，采用一般的队列。
这时我们不再能保证每个状态第一次入队时就能得到最小代价，所以只能允许一个状态被多次更新，多次进出队列。我们不断执行搜索，直到队列为空。
整个广搜算法对搜索树进行了重复遍历和更新，直至算法收敛到最优解，其实也就是迭代的思想。最坏情况下，该算法的时间复杂度会从一般广搜的O(N)增长到$O(N^2)$.对应在最短路问题中，就是我们的SPFA算法。
2.改用优先队列进行广搜。
这里的优先队列就相当于一个二叉堆。我们可以每次从队列中取出当前代价最小的状态进行扩展，沿着每条分支把到达的新状态加入优先队列。不断执行搜索，直到队列为空。

迭代加深DFS

图论

邻接表的数组模拟链表的实现

长度为n的数组记录了从每个节点出发的第一条边在ver和edge数组中的存储位置，长度为m的边集数组ver和edge记录了每条边的终点和边权，长度为m的数组next模拟了链表指针，表示从相同节点出发的下一条边在ver和edge中的存储位置。

```
//建图
void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    next[tot] = head[x];
    head[x] = tot;
}
//访问从x出发的所有有向边
for (int i = head[x]; i;i=next[i])
{
    int y = ver[i];
    z = edge[i];
}
```

Dijkstra算法

```
const int maxn = 3010;
int a[maxn][maxn], d[maxn], n, m;
bool vis[maxn];
void dijkstra()
{
    memset(d, 0x3f, sizeof(d));
    memset(vis, 0, sizeof(vis));
    d[1] = 0;
    for (int i = 1; i < n;i++)
    {
        int x = 0;
        for (int j = 1; j <= n;j++)
        {
            if(!vis[j]&&(x==0||d[j]<d[x]))
                x = j;
        }
        vis[x] = 1;
        for (int j = 1; j <= n;j++)
            d[j] = min(d[j], d[x] + a[x][j]);
    }
}
int main()
{
    cin >> n >> m;
    memset(a, 0x3f, sizeof(a));
    for (int i = 1; i <= n;i++)
        a[i][i] = 0;
    for (int i = 1; i <= n;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        a[x][y] = min(a[x][y], z);
    }
    dijkstra();
```

```
    for (int i = 1; i <= n;i++)
        printf("%d\n", d[i]);
}
```

堆维护的dijkstra算法

```
const int N = 1e5 + 10, M = 1e6 + 10;
int head[N], Next[M], ver[M], edge[M], d[N];
bool v[N];
int n, m, tot;
//大根堆，pair的第二维为节点编号
//pair的第一维为dist的相反数
priority_queue<pair<int, int>> q;
void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    Next[tot] = head[x];
    head[x] = tot;
}
void dijkstra()
{
    memset(d, 0x3f, sizeof(d));
    memset(v, 0, sizeof(v));
    d[1] = 0;
    q.push(make_pair(0, 1));
    while(q.size())
    {
        int x = q.top().second, q.pop();
        if(v[x])
            continue;
        v[x] = 1;
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i], z = edge[i];
            if(d[y]>d[x]+z){
                d[y] = d[x] + z;
                q.push(make_pair(-d[y], y));
            }
        }
    }
}
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= m;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
    }
    dijkstra();
    for (int i = 1; i <= n;i++)
    {
        printf("%d\n", d[i]);
    }
}
```

```cpp
//SPFA算法，队列优化的bellman-ford算法
void spfa()
{
    memset(d, 0x3f, sizeof(d));
    memset(v, 0, sizeof(v));
    d[1] = 0;
    v[1] = 1;
    q.push(1);
    while (q.size())
    {
        int x = q.front(), q.pop();
        v[x] = 0;
        //扫描所有出边
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i], z = edge[i];
            if(d[y]>d[x]+z)
            {
                d[y] = d[x] + z;
                if(!v[y])
                    q.push(y), v[y] = 1;
            }
        }
    }
}
```

分层图

从最短路问题的角度去理解，图中的节点也不仅限于“整数编号”，可以扩展到二维，用二元组(x,p)代表一个节点，从(x,p)到(y,p)有长度为z的边，从(x,p)到(y,p+1)有长度为0的边。D[x,p]表示从起点(1,0)到节点(x,p)，路径上最长的边最短是多少。这是$N*K$个点，$P*K$条边的广义最短路问题，被称为分层图最短路。

```cpp
#include<cstdio>
#include<algorithm>
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;
int n, p, k, tot;
int INF = 0x3f3f3f3f;
const int N = 1e3 + 10, P = 1e4 + 10;
int head[N * N], Next[N * P], edge[N * P], ver[N * P], d[N * N];
queue<int> q;
void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    Next[tot] = head[x];
    head[x] = tot;
}
bool v[N * N];
void spfa()
{
    memset(d, INF, sizeof(d));
    memset(v, 0, sizeof(v));
    d[1] = 0;
    v[1] = 1;
```

```
    q.push(1);
    while (q.size())
    {
        int x = q.front();
        q.pop();
        v[x] = 0;
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i], z = edge[i];
            if(d[y]>max(d[x],z))
            {
                d[y] = max(d[x], z);
                if(!v[y])
                {
                    q.push(y);
                    v[y] = 1;
                }
            }
        }
    }
}
int main()
{
    cin >> n >> p >> k;
    for (int i = 1; i <= p;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        for (int j = 0; j <= k;j++)
        {
            add(x + j * n, y + j * n, z);
            add(y + j * n, x + j * n, z);
        }
        for (int j = 0; j < k;j++)
        {
            add(x + j * n, y + (j + 1) * n, 0);
            add(y + j * n, x + (j + 1) * n, 0);
        }
    }
    spfa();
    if(d[n*(k+1)]==INF)
        puts("-1");
    else
    {
        printf("%d\n", d[n * (k + 1)]);
    }
    return 0;
}
```

堆优化的分层图最短路

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

```cpp
const int N = 1e4 + 100, M = 2e5 + 100, K = 22;
const int INF = 0x3f3f3f3f;
int n, m, k, tot;
int head[N * K], edge[M * K], Next[M * K], ver[M * K], d[N * K];
bool v[N * K] = {false};
priority_queue<pair<int, int> > que;
void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    Next[tot] = head[x];
    head[x] = tot;
}
void dijkstra()
{
    for (int i = 0; i <= n*(k+1);i++)
        d[i] = INF;
    d[1] = 0;
    que.push(make_pair(0, 1));
    while (que.size())
    {
        int x = que.top().second;
        que.pop();
        if(v[x])
            continue;
        v[x] = 1;
        if(x==n*(k+1))
            return;
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i], z = edge[i];
            if(d[y]>d[x]+z)
            {
                d[y] = d[x] + z;
                que.push(make_pair(-d[y], y));
            }
        }
    }
}
int main()
{
    cin >> n >> m >> k;
    for (int i = 1; i <= m;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        for (int j = 0; j <= k;j++)
        {
            add(x + j * n, y + j * n, z);
            add(y + j * n, x + j * n, z);
        }
        for (int j = 0; j < k;j++)
        {
            add(x + j * n, y + (j + 1) * n, 0);
            add(y + j * n, x + (j + 1) * n, 0);
        }
    }
    dijkstra();
```

```
    cout << d[n * (k+1)] << endl;
    return 0;
}
```

分层图最短路的第二种思路是我们把dis数组和vis数组多开一维记录k次机会信息。
开二维数组d[i][j]，其中i代表走到第i个节点，用掉j次机会后的最短路或最小花费或其他的最值。vis[i][j]表示到达i用了j次免费机会的情况是否出现过。
仿照动态规划的思想，用D[x,p]表示从1号基站到x，途中已经制定了p条电路免费时，经过的路径上最贵的电缆花费最小是多少(也就是选择一条从1到x的路径，使得路径上第p+1大的边权尽量小)。若有一条从x到y长度为z的无向边，则应该用max(D[x,p],z)更新D[y,p]的最小值，用D[x,p]更新D[y,p+1]的最小值。前者表示不在电缆(x,y,z)上使用免费升级服务，后者表示使用。
动态规划中的无后效性其实告诉我们，动态规划对状态空间的遍历构成一张有向无环图，遍历顺序就是该有向无环图的一个拓扑序。有向无环图的节点对应问题中的状态，图中的边对应问题中的转移，转移的选取就是动态规划的中的决策。显然，我们刚才设计的状态转移是有后效性的。在有后效性时，一种解决方案就是利用迭代思想，借助spfa算法进行动态规划，直至所有状态收敛。

```
#include<cstdio>
#include<queue>
#include<algorithm>
#include<iostream>
#include<utility>
#include<cstring>
using namespace std;
const int N = 1e3 + 10, P = 2e4 + 10;
int d[N][N], head[N], edge[P], ver[P], Next[P];
bool v[N][N];
int n, p, k, tot;
queue<pair<int,int> > q;
const int INF = 0x3f3f3f3f;

void add(int x,int y,int z)
{
    ver[++tot] = y;
    edge[tot] = z;
    Next[tot] = head[x];
    head[x] = tot;
}
void spfa()
{
    memset(d, INF, sizeof(d));
    memset(v, 0, sizeof(v));
    d[1][0] = 0;
    v[1][0] = 1;
    q.push(make_pair(1,0));
    while (q.size())
    {
        pair<int, int> tmp = q.front();
        q.pop();
        int x = tmp.first;
        int pp = tmp.second;
        v[x][pp] = 0;
        for (int i = head[x]; i;i=Next[i])
        {
            int y = ver[i], z = edge[i];
            if(d[y][pp]>max(d[x][pp],z))
            {
```

```
                d[y][pp] = max(d[x][pp], z);
                if(!v[y][pp])
                    {q.push(make_pair(y, pp)), v[y][pp] = 1;}
            }
            if(pp<k&&d[y][pp+1]>d[x][pp])
            {
                d[y][pp + 1] = d[x][pp];
                if(!v[y][pp+1])
                    {q.push(make_pair(y, pp + 1)), v[y][pp + 1] = 1;}
            }
        }
    }
}
int main()
{
    cin >> n >> p >> k;
    for (int i = 1; i <= p;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
        add(y, x, z);
    }
    spfa();
    if(d[n][k]==INF)
        puts("-1");
    else
    {
        printf("%d\n", d[n][k]);
    }
}
```

Floyed 算法

设D[k,i,j]表示"经过若干个编号不超过k的节点"，从i到j的最短路。该问题可以划分为两个子问题，经过编号不超过k-1的节点从i到j，或者从i先到k再到j。于是：

$$D[k,i,j] = min(D[k-1,i,j], D[k-1,i,k] + D[k-1,k,j]) \quad (1)$$

初值为D[0,i,j]=A[i,j]，其中A为本节开头定义的邻接矩阵。
可以看到，Floyed算法的本质是动态规划。k是阶段，所以必须置于最外层循环中；i和j是附加状态，所以必须置于内层循环中。
与背包问题的状态转移方程类似，k这一维可被省略。最初，我们直接用D保存邻接矩阵，然后执行动态规划过程。当外层循环到k时，内层有状态转移：

$$D[i,j] = min(D[i,j], D[i,k] + D[k,j]) \quad (2)$$

最终，D[i,j]就保存了i到j的最短路长度。

```
int d[310][310], n, m;
int main()
{
    cin >> n >> m;
    //
    //* 把d数组初始化为邻接矩阵
    memset(d, 0x3f, sizeof(d));
```

```
    for (int i = 1; i <= n;i++)
        d[i][i] = 0;
    for (int i = 1; i <= m;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        d[x][y] = min(d[x][y], z);
    }
    //*Floyed求任意两点间的最短路径
    for (int k = 1; k <= n;k++)
        for (int i = 1; i <= n;i++)
            for (int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    // * 输出
    for (int i = 1; i <= n;i++)
    {
        for (int j = 1; j <= n;j++)
            printf("%d ", d[i][j]);
        puts("");
    }
}
```

传递闭包

在交际网络中，给定若干个元素和若干对二元关系，且关系具有传递性。“通过传递性推导出尽量多的元素之间的关系”的问题被称为传递闭包。
建立邻接矩阵d，其中d[i,j]=1表示i与j有关系，d[i,j]=0表示i与j没有关系。特别的，d[i,i]始终为1.使用floyed算法可以解决传递闭包问题。

```
bool d[310][310];
int n, m;
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n;i++)
        d[i][i] = 1;
    for (int i = 1; i <= m;i++)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        d[x][y] = d[y][x] = 1;
    }
    for (int k = 1; k <= n;k++)
        for (int i = 1; i <= n;i++)
            for (int j = 1; j <= n;j++)
                d[i][j] |= d[i][k] & d[k][j];
}
```

无向图的最小环问题

给定一张无向图，求图中一个至少包含3个点的环，环上的节点不重复，并且环上的边的长度之和最小。该问题被称为无向图的最小环问题。在本题中，你需要输出最小环的方案，若最小环不唯一，则输出任意一个均可。若无解，则输出"No solution"。图的节点数不超过100.

考虑Floyed算法的过程。当外层循环k刚开始时，d[i,j]保存着“经过编号不超过k-1的节点”从i到j的最短路长度。
于是，

$$\min_{1\le i<j<k} d[i,j]+a[j,k]+a[k,j] \tag{3}$$

就是满足一下两个条件的最小环长度：

> 1.由编号不超过k的节点构成。
> 2.经过节点k.

上式中的i,j相当于枚举了环上与k相邻的点。
$\forall k\in[1,n]$,都对上式进行计算，取最小值，即可得到整张图的最小环。
在该算法中，我们对每个k只考虑了由编号不超过k的节点构成的最小环，没有考虑编号大于k的节点。由对称性知，这样做不会影响结果。

对于有向图的最小环问题，可以枚举起点s=1~n，执行堆优化的dijkstra算法求解单源最短路问题。s一定是第一个被从堆中取出的节点，我们扫描s的所有出边，当扩展更新完成后，令$d[s]=+\inf$，然后继续求解。当s第二次被从堆中取出时，d[s]就是经过点s的最小环长度。

```cpp
#include<cstdio>
#include<iostream>
#include<algorithm>
#include<vector>
#include<cstring>
using namespace std;

const int maxn = 310;
int a[maxn][maxn], d[maxn][maxn], pos[maxn][maxn];
int n, m, ans = 0x3f3f3f3f;
vector<int> path;//?   具体方案
void get_path(int x,int y)
{
    if(pos[x][y]==0)
        return;
    get_path(x, pos[x][y]);
    path.push_back(pos[x][y]);
    get_path(pos[x][y], y);
}
int main()
{
    cin >> n >> m;
    memset(a, 0x3f, sizeof(a));
    for (int i = 1; i <= n;i++)
        a[i][i] = 0;
    for (int i = 1; i <= m;i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        a[y][x] = a[x][y] = min(a[x][y], z);//* 完整的图
    }
    memcpy(d, a, sizeof(a));
    for (int k = 1; k <= n;k++){
        for (int i = 1; i < k;i++)
            for (int j = i+1; j < k;j++)
            if((long long)d[i][j]+a[j][k]+a[k][i]<ans)
            {
```

```
                ans = d[i][j] + a[j][k] + a[k][i];
                path.clear();
                path.push_back(i);
                get_path(i, j);
                path.push_back(j);
                path.push_back(k);
            }
        for (int i = 1; i <= n;i++)
            for (int j = 1; j <= n;j++)
            {
                if(d[i][j]>d[i][k]+d[k][j])
                {
                    d[i][j] = d[i][k] + d[k][j];
                    pos[i][j] = k;
                }
            }
    }
    if(ans==0x3f3f3f3f)
    {
        puts("No solution.");
        return 0;
    }
    for (int i = 0; i < path.size();i++)
        printf("%d ", path[i]);
    puts("");
}
```

动态加边的Floyed算法

最小生成树

定义：给定一张边带权的无向图$G=(V,E),n=|V|,m=|E|$。由V中全部n个顶点和E中n-1条边构成的无向连通子图被称为G的一棵生成树。边的权值之和最小的生成树被称为无向图G的最小生成树。

定理：
任意一棵最小生成树一定包含无向图中权值最小的边。
最小生成树的三个性质：

> 1.最小生成树是树，边数等于定点数减一，且树内一定不会有环。
> 2.对于给定的图G(V,E)，最小生成树可以不唯一，但是其边权之和一定是唯一的。
> 3.由于最小生成树是在无向图上生成的，因此其根节点一定可以是这棵树上的任意一个节点。

Prim算法

基本思想是对图G(V,E)设置集合S，存放已被访问的顶点，然后每次从集合V-S中选择与集合S的最短距离最小的一个顶点(几位u)，访问并加入集合S。之后，令顶点u为中介点，

动态规划

一般的动态规划

动态规划求解的原则：

- 最优子结构
- 子问题的重叠性
- 所有节点和边的关系构成一张DAG

一个例题：

分析一下：

这个题有几个技巧值得学习：

- 题目要求输出路径，我们可以建立一个数组$pre[i]=j$,储存i节点的前置节点为j。全部更新完毕以后递归输出即可。
- 1号节点的入度为0，所以从1号节点开始，递推更新其他节点。
- 最优子结构性质：已经被更新完的节点不会因为后面节点的更新而发生变化

代码如下

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn = 210;

int n, m, id, ans;
int map[maxn][maxn], dp[maxn], a[maxn], pre[maxn];
void print(int x)
{
    if(pre[x]==0){ // 递归到最后一个位置，开始输出
        printf("%d-", x);
        return;
    }
    print(pre[x]);// 先输出前驱节点
    if(x==id)
        printf("%d\n", x);// 输出当前节点的编号
    else
        printf("%d-", x);
}
int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n;i++)
    {
        scanf("%d", &a[i]);
    }
    while (1)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        if(x==0&&y==0)
            break;
        map[x][y] = 1;
    }
    for (int i = 1; i <= n;i++)
    {
```

```
        for (int j = 1; j <= n;j++)
        {
            if(map[j][i]==1&&dp[j]>dp[i]){// i号节点可以被j号节点更新
                dp[i] = dp[j];
                pre[i] = j;// i号节点的前驱是j
            }
        }
        dp[i] += a[i];// i号节点的价值再加上自身的价值
        if(ans<dp[i]){
            ans = dp[i];// 更新最优值
            id = i;
        }
    }
    print(id);
    printf("%d\n", ans);
    return 0;
}
```

这个题也可以用DFS做，但是时间复杂度要高得多。200个地窖的情况有两个点过不去。

LIS 最长上升子序列和最长不上升子序列

问题描述1：给定一个长度为N的数列A，求数值单调递增的子序列的长度最长是多少。

问题描述2：给定长度为N的数列A，求数值不上升的子序列的长度最长是多少。

下面给出$O(nlogn)$的解法。

对于**最长上升子序列**，在$O(n^2)$的解法中，dp数组的元素$dp[i]$存储的是序列A中i号位置及以前的最长子序列长度。对每一个位置i，我们都需要与i号位置之前的每一个元素进行比对，并更新$dp[i]$。这样其实会造成不必要的时间开销，因为很多元素本可不必比对。

对于$O(nlogn)$的解法，思想如下：$dp[i]$存储的是长度为i的序列中末尾元素最小的那个元素，也就是最长子序列长度为i时候结尾的最优元素。注意dp中存储的并不是最长***子序列.

代码如下：

```
    dp[1] = a[1];
    int len = 1;
    for (int i = 1; i <= n;i++)
    {
        if(a[i]>dp[len]){// 如果a[i]大于最长上升子序列的最后一个元素，直接将其衔接上
            dp[++len] = a[i];
        }
        else
        {// 否则，查找第一个大于等于a[i]的元素的位置，将其替换掉。注意不是第一个大于，不能用upper_bound.
            int pos = lower_bound(dp + 1, dp + len + 1, a[i]) - dp;
            dp[pos] = a[i];
        }
    }
    cout << len << endl;

```

对于**最长不上升子序列**，原理类似。不过这个时候我们要维护的是一个单调不升的序列。C++ STL中的lower_bound和upper_bound 默认序列是单调递增的。因此我们需要写cmp函数或者使用greater()函数

代码如下：

```
 dp[1] = a[1];
    int len = 1;
    for (int i = 1; i <= n;i++)
    {
        if(a[i]<=dp[len]){// 如果a[i]小于等于最长上升子序列的最后一个元素，直接将其衔接上
            dp[++len] = a[i];
        }
        else
        {// 否则，查找第一个小于a[i]的元素的位置，将其替换掉。
            int pos = upper_bound(dp + 1, dp + len + 1, a[i]) - dp;
            dp[pos] = a[i];
        }
    }
    cout << len << endl;

```

编辑距离

打了这个题又再次让我感受到自己的弱小。。普及/提高- 难度

题目描述

展开

设A和B是两个字符串。我们要用最少的字符操作次数，将字符串A转换为字符串B。这里所说的字符操作共有三种：

1、删除一个字符；

2、插入一个字符；

3、将一个字符改为另一个字符；

！皆为小写字母！

输入格式

第一行为字符串A；第二行为字符串B；字符串A和B的长度均小于2000。

输出格式

只有一个正整数，为最少字符操作次数。

输入输出样例

输入 #1 复制

```
sfdqxbw
gfdgw
```

输出 #1 复制

```
4
```

曾经有个人说，90%的字符串问题都可以用动态规划解决，90%的字符串动态规划都需要二维数组。这句话的准确性是存疑的，但是可以看出，动态规划在解决字符串问题中的重要性。

步骤：

- 定义数组元素的含义
- 找出关系数组元素间的关系式

- 找出初始值

对于字符串动态规划问题，大多数情况下，$dp[i][j]$与$dp[i-1][j-1], dp[i-1][j], dp[i][j-1]$之间会存在某种关系。没有思路的时候不妨从这个角度考虑一下，but overfitting.

代码如下：

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn = 2e3 + 10;

int n,m;
char s1[maxn], s2[maxn];
int dp[maxn][maxn];

int main()
{
    scanf("%s", s1+1);
    scanf("%s", s2+1);
    int n1 = strlen(s1+1);
    int n2 = strlen(s2+1);
    for (int i = 1; i <= n2;i++)
        dp[0][i] = dp[0][i - 1] + 1;
    for (int i = 1; i <= n1;i++)
        dp[i][0] = dp[i - 1][0] + 1;
    for (int i = 1; i <= n1; i++)
    {
        for (int j = 1; j <= n2; j++)
        {
            if (s1[i] == s2[j])
            {
                dp[i][j] = dp[i - 1][j - 1];
            }
            else
                dp[i][j] = min(dp[i - 1][j - 1], min(dp[i][j - 1], dp[i - 1][j])) + 1;
        }
    }
    int ans = dp[n1][n2];
    cout << ans << endl;
    return 0;
}
```

0-1背包问题

```
void beibao()
{
    for (int i = 1; i <= n;i++)
        for (int j = W; j >= w[i];j--)
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
}

```

完全背包

```
void beibao()
{
    for (int i = 1; i <= n;i++)
        for (int j = w[i]; j <= W;j++)
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
}

```

二进制整数拆分化完全背包为01背包

```
int w_new[100], v_new[100];
int w[100], v[100];
int W;
//* 整数拆分
void intbeibao()
{
    int p = 0;
    for (int i = 1; i <= n;i++)
    {
        for (int j = 0; w[i] * (1 << j) <= W;j++)
        {
            w_new[++p] = w[i] * (1 << j);
            v_new[p] = v[i] * (1 << j);
        }
    }
    for (int i = 1; i <= p;i++)
        for (int j = W; j >= w_new[i];j--)
            dp[j] = max(dp[j], dp[j - w_new[i]] + v_new[i]);
}

```

二进制简化多重背包

```
int beibao()
{
    /*
        这道题有几个要注意的点。注意二进制拆分多重背包与完全背包的区别
        完全背包是不限数量的，只要每种物品的单个质量小于总限制W就可以
        多重背包是限制数量的。拆分出来的每种物品的数量总和要等于m[i]
        所以每次拆除一个二进制数的物品，都要在m[i]中减掉对应的数量
        由于二进制拆分拆出的是二的幂次数量。所以最后可能有剩余。内层拆分循环完毕，要在检查一下m[i]
```

```
        如果有剩余，把剩余物品作为一个新的物品加入拆分出的物品区当中
    */
    int p = 0;
    for (int i = 1; i <= n;i++)
    {
        for (int k = 0; (1 << k) <= m[i]; k++)
        {
            w_new[++p] = (1 << k) * w[i];
            v_new[p] = (1 << k) * v[i];
            m[i] -= (1 << k);
        }
        if(m[i]>0)
        {
            w_new[++p] = m[i] * w[i];
            v_new[p] = m[i] * v[i];
        }
    }
    for (int i = 1; i <= p;i++)
        for (int j = w; j >= w_new[i];j--)
            dp[j] = max(dp[j], dp[j - w_new[i]] + v_new[i]);
    return p;
}

```

多重部分和问题

```
void solve()
{
    memset(dp, -1, sizeof(dp));
    dp[0] = 0;
    for (int i = 1; i <= n;i++)
    {
        for (int i = 1; i <= n;i++)
        {
            for (int j = 0; j <=w;j++)
            {
                if(dp[j]>=0)
                {
                    dp[j] = m[i];
                }
                else if (dp[j-w[i]]<=0||w[i]>j)
                {
                    dp[j] = -1;
                }
                else
                    dp[j] = dp[j - w[i]] - 1;
            }
        }
    }
}

```

钢条切割问题

问题描述：给出不同长度的钢条对应的价格，以及要切割的钢条的长度，请你计算出如何切割才能使得收益最大。

输入：

- 正整数n，表示要切割的钢条的长度。正整数m，表示有m种长度1~m
- 接下来输入m个整数，对应长度为i的钢条价格

输出：

- 第一行是最大价值
- 第二行是切割的方法

代码如下：

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<vector>
#include<algorithm>
#include<cmath>
#include<utility>
using namespace std;
const int maxn=1e5+100;

int n,m;
// int p[20] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
int p[maxn];
int ans;
int dp[maxn], path[maxn];
void solve()
{
    for (int i = 1; i <= n;i++)
    {
        dp[i] = p[i];
        for (int j = 1; j <= i; j++)
        {
            dp[i] = max(dp[i], dp[j] + dp[i - j]);
        }
    }
}
int memoized_cut_rod_aux(int k)// 记忆化搜索方式，自顶向下
{
    int q;
    if(dp[k]){
        return dp[k];
    }
    if(k==0){
        q = 0;
    }
    else{
        q = -100;
        for (int i = 1; i <= k;i++){
            q = max(q, p[i] + memoized_cut_rod_aux(k - i));
        }
    }
    dp[k] = q;
    return q;
}
void extend_solve(int k)// 自底向下的动态规划方法，记录切割方式
```

```
{
    for (int i = 1; i <= k;i++)
    {
        dp[i] = p[i];
        path[i] = i;
        for (int j = 1; j <= i;j++)
        {
            if(dp[i]<(p[j]+dp[i-j])){
                dp[i] = p[j] + dp[i - j];
                path[i] = j;// 记录下钢条最左端被切割的长度
            }
        }
    }
}
void print(int k)
{
    cout << dp[k] << endl;// 输出钢条切割的最大价值
    while (k>0)
    {
        cout << path[k] << " ";// 循环遍历path，依次输出钢条切割的方式
        k -= path[k];
    }
}
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= m;i++)
    {
        cin >> p[i];
    }
    extend_solve(n);
    print(n);
    return 0;
}
```

数学知识

对于一个足够大的数n，不超过n的质数的个数大约是

$$\frac{n}{lnn} \tag{4}$$

质数的判定

```cpp
bool is_Prime(int n)
{
    if(n<2)
        return false;
    for (int i = 2; i <= sqrt(n);i++)
    {
        if(n%i==0)
            return false;
    }
    return true;
}
```

质数的筛选

1. 埃氏筛法
   任意整数x的倍数都不是质数。
   我们可以从2开始，由小到大扫描每个数x，把它的倍数$2x, 3x, \ldots, \lfloor N/x \rfloor * x$标记为合数。当扫描到一个数时，若它未被标记，则它不能被2~x-1之间的任何一个数整除，则它就是质数。
   实际上，小于x^2的x的倍数在扫描更小的数的时候就已经被标记过了。因此我们可以做如下优化：对于每个x，我们只需要从x^2开始，把$x^2, (x+1) * x, \ldots$,标记为合数即可。

```cpp
void primes(int n)
{
    memset(v, 0, sizeof(v));
    for (int i = 2; i <= n;i++)
    {
        if(v[i])
            continue;
        cout << i << endl;//i是质数
        for (int j = i; j <= n / i;j++)
            v[j * i] = 1;
    }
}
```

质因数分解

试除法
我们可以扫描2 ~$\lfloor N/x \rfloor$的每个数d,若d能整除N，则从N中除掉所有的因子d，同时累计除去的d的个数。
因为一个合数的因子一定在扫描到这个合数之前就被除掉了，所以在上述过程中能整除N的一定是质数。
特别的，如果N没被任何2~$\sqrt{N}$的数整除，则N是质数，无需分解。

```cpp
void divide(int n)
{
    m = 0;
    for (int i = 2; i <= sqrt(n); i++)
    {
        if(n%i==0){
            p[++m] = i, c[m] = 0;
            while (n%i==0)
            {
                n /= i, c[m]++;
            }
```

```
        }
    }
    if(n>1)
        p[++m] = n, c[m] = 1;
    for (int i = 1; i <= m;i++)
        cout << p[i] << '^' << c[m] << endl;
}
```

线性递推求模质数的逆元

```
void solve()
{
    // p is a prime number
    // inv[i]*i==1(mod p)
    inv[1] = 1;
    for (int i = 2; i < p;i++)
    {
        inv[i] = (p - p / i) * inv[p % i] % p;
    }
}
```

阶乘逆元

求模P意义下的阶乘，组合数方法：

在模p意义下，除法相当于乘以逆元。

> 阶乘逆元
>
> 我们有 $C_n^m = \frac{A_n^m}{m!} = \frac{n!/(n-m)!}{m!} = \frac{n!}{(n-m)!m!}$
>
> 所以在模 p 意义下，如果有
>
> - fac数组：fac[x]表示 $(x!) \bmod p$
> - inv数组：inv[x]表示 $x!$ 在模 p 意义下的逆元
>
> 则可以代码实现求组合数：
>
> ```
> return fac[n]*inv[m]%p*inv[n-m]%p;
> ```

```
void solve()
{
    fac[0] = 1;
    for (int i = 1; i < p;++i)
        fac[i] = fac[i - 1] * i % p;
}

```

阶乘逆元：

$$inv(n!) = inv(1) * inv(2) * \cdots * inv(n) \tag{5}$$

inv函数满足完全积性

```
void solve()
{
    // 求阶乘逆元，先线性递推求逆元，然后取前缀积。
    inv[0] = inv[1] = 1;
    for (int i = 2; i < p;++i)
        inv[i] = (p - p / i) * inv[p % i] % p;
    for (int i = 1; i < p;i++)
        inv[i] = inv[i - 1] * inv[i];
}
```

求1~n内每个数的约数，时间复杂度$O(nlogn)$

```
void solve()
{
    vector<int> factor[5000];
    for (int i = 1; i <= n;i++)
        for (int j = 1; j <= n / i;j++)
        {
            factor[i * j].push_back(i);
        }
}

```

质数

高精度

大整数的存储

利用数组存储大整数，整数的高位存储在数组的高位，整数的地位存储在数组的低位。由于在读入时高位会读入到数组的低位，因此在存入结构体的时候要翻转一下。

```
struct bign{
    int d[1000];
    int len;
    bign(){ // 构造函数初始化
        memset(d,0,sizeof(d));
        len=0;
    }
};
```

读入大整数并存入结构体，返回一个构造好的大整数

```
bign change(char str[])
{
    bign a;
    a.len = strlen(str);
    for (int i = 0; i < a.len;i++)
    {
        a.d[i] = str[a.len - i - 1] - '0';
    }
    return a;
}

```

大整数比较大小

```
int compare(bign a,bign b)
{// 比较a和b的大小，若a大返回1，若b大返回-1，若a和b相等则返回0
    if(a.len>b.len){
        return 1;
    }
    else if (a.len<b.len)
    {
        return -1;
    }
    else
    {
        for (int i = a.len - 1; i >= 0;i--)
        {
            if(a.d[i]==b.d[i])
                continue;
            if(a.d[i]>b.d[i])
                return 1;
            else if (a.d[i]<b.d[i])
            {
                return -1;
            }
        }
    }
    return 0;
}

```

高精度加法

```
bign add(bign a,bign b)
{
    // 高精度加法，a+b
    bign c;
    int carry = 0;//carry 是进位
    for (int i = 0; i < a.len || i < b.len;i++)// 以较长的为界限
    {
        int temp = a.d[i] + b.d[i] + carry;
        c.d[c.len++] = temp % 10;
        carry = tmp / 10;
    }
    if (carry)
    {
```

```
        c.d[c.len++] = carry;
    }
    return c;
}

```

写代码时请注意：

- 是否要开Long Long？数组边界处理好了么？
- 实数精度有没有处理？
- 特殊情况处理好了么？
- 做 一些总比不做好。
- 最大值和最小值问题可不可以用二分答案？
- 有没有贪心策略？否则能不能dp？