算法学习：AcWing

Algorithm Learning: AcWing

吴小强 编著

深圳 • SHENZHEN

前言

目 录

PART I

第一部分

算法基础

第一章

算法基础

1.1 快速排序

1.1.1 AcWing 785. 快速排序

AcWing 785. 快速排序

给定你一个长度为 n 的整数数列。请你使用快速排序对这个数列按照从小到大进行排序。并将排好序的数列按顺序输出。

输入格式：

输入共两行，第一行包含整数 n。第二行包含 n 个整数（所有整数均在 $1 \sim 10^9$ 范围内），表示整个数列。

输出格式：

输出共一行，包含 n 个整数，表示排好序的数列。

数据范围：$1 \leq n \leq 100000$

输入样例：

```
5
3 1 2 4 5
```

输出样例：

```
1 2 3 4 5
```

快速排序算法基于**分治**算法，以一个数来作为分治的节点。

随机选取数组中的某个元素 x 作为分界点，操作数组中的元素使得数组被分割为两个部分，左边一侧的元素小于等于 x，右边一侧则大于等于 x。接下来递归的对左右两侧数组进行操作，直到最小数组只有一个元素则完成排序。

主要步骤如下：

1. 确定分界点 x，可取值：q[l], q[r], q[(l + r) >> 1], random value
2. 调整数组，使得左边小于等于 x，右边大于等于 x

3. 递归处理左右两段

quick sort

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int *q, int a, int b) {
    int tmp = q[a];
    q[a] = q[b];
    q[b] = tmp;
}

void quick_sort(int *q, int l, int r)
{
    if (l >= r) {
        return;
    }
    int x = q[(l + r) >> 1];
    int i = l - 1;
    int j = r + 1;
    while (i < j) {
        do i++; while (q[i] < x);
        do j--; while (q[j] > x);
        if (i < j) {
            swap(q, i, j);
        }
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}

int main() {
    int n;
    scanf("%d", &n);
    int *q = (int *)calloc(sizeof(int), n);
    if (q == NULL) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", q + i);
    }

    quick_sort(q, 0, n - 1);

    for (int i = 0; i < n; i++) {
        printf("%d ", q[i]);
    }
    free(q);
    return 0;
}
```

从上述代码段中可以清晰看到递归处理的过程，每次选取分界点，之后将左右两侧的元素进行调整，此处采用双指针算法。

这里有两个问题：

1. 在选择 x 时选择q[l]则在递归是不能选用i，会出现边界问题 | i $-$ 1, i
2. 在选择 x 时选择q[r]则在递归是不能选用j，会出现边界问题 | j, j $+$ 1

边界用例可使用1, 2这个例子，会有递归不结束的问题

该算法**不稳定**，因为q[i]和q[j]相等的时候会发生交换。
这里调整数组的部分是难点，怎么优雅的调整数组？暴力做法可以开辟两个辅助数组来存储。
双指针做法优雅简洁。

时间复杂度分析：

1.1.2 AcWing 786. 第 k 个数

快速选择算法可选出有序数组中的第 k 个数，与快排中逻辑相同，左侧的元素都小于 x 右侧元素都大于 x。如果左侧元素的数量大于等于 k 则表示第 k 个元素在左侧数组中，反之则在右侧数组中寻找 $k-$ left length 的元素。

AcWing 786. 第 k 个数

给定一个长度为 n 的整数数列，以及一个整数 k，请用快速选择算法求出数列从小到大排序后的第 k 个数。

输入格式：

第一行包含两个整数 n 和 k。第二行包含 n 个整数（所有整数均在 $1 \sim 10^9$ 范围内），表示整数数列。

输出格式：

输出一个整数，表示数列的第 k 小数。

数据范围：

$1 \leq n \leq 100000, 1 \leq k \leq n$

输入样例：

5 3

3 1 2 4 5

输出样例：

3

find kth smallest number

```c
#include <stdio.h>
#include <stdlib.h>

int q_select(int *q, int l,
             int r, int k) {
    if (r <= l) {
        return q[l];
    }

    int x = q[(l + r) >> 1];
    int i = l - 1;
    int j = r + 1;

    while (i < j) {
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);

        if (i < j) {
            int tmp = q[i];
            q[i] = q[j];
            q[j] = tmp;
        }
    }

    int length = j - l + 1;
    if (length < k) {
        return q_select(q, j + 1, r,
            k - length);
    } else {
        return q_select(q, l, j, k);
    }
}

int main()
{
    int n;
    int k;
    scanf("%d %d", &n, &k);
    int *q = (int *)calloc(sizeof(int), n);
    if (q == NULL) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", q + i);
    }
    int ret = q_select(q, 0, n - 1, k);
    printf("%d", ret);
    return 0;
}
```

1.2 归并排序

归并排序同样是基于**分治**算法，不过是以整个数组的中间位置来分。

将数组分割成两个已经分别排序好的有序数组，再将其二者合并即可。此方法需要有单独的空间来存放合并的临时结果，再将临时结果写入到原始区域中。

主要步骤如下：

1. 确定分界点，mid = (l + r) >> 1

2. 递归排序左右两边

3. 归并,将两个有序的子数组合二为一

1.2.1 AcWing 787. 归并排序

AcWing 787. 归并排序

给定你一个长度为 n 的整数数列。请你使用归并排序对这个数列按照从小到大进行排序。并将排好序的数列按顺序输出。

输入格式:

输入共两行,第一行包含整数 n。第二行包含 n 个整数(所有整数均在 $1 \sim 10^9$ 范围内),表示整个数列。

输出格式:

输出共一行,包含 n 个整数,表示排好序的数列。

数据范围:$1 \leq n \leq 100000$

输入样例:

```
5
3 1 2 4 5
```

输出样例:

```
1 2 3 4 5
```

merge sort

```c
#include <stdio.h>
#include <stdlib.h>

#define N 100010

int backup[N];

void merge_sort(int *q, int l, int r)
{
    if (r <= l) {
        return;
    }
    int mid = (l + r) >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);
    int k = 0;
    int i = l;
    int j = mid + 1;
    while (i <= mid && j <= r) {
        if (q[i] <= q[j]) {
            backup[k++] = q[i++];
        }
        if (q[j] < q[i]) {
            backup[k++] = q[j++];
        }
    }
    while (i <= mid) {
        backup[k++] = q[i++];
    }
    while (j <= r) {
        backup[k++] = q[j++];
    }

    for (i = l, j = 0; j < k; i++, j++) {
        q[i] = backup[j];
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    int *q = (int *)calloc(sizeof(int), n);
    if (q == NULL) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", q + i);
    }
    merge_sort(q, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", q[i]);
    }
    return 0;
}
```

双指针算法做归并

这里归并两个子数组之后要写回去，backup数组只是临时存储使用。

1.2.2 AcWing 788. 逆序对的数量

AcWing 788. 逆序对的数量

给定一个长度为 n 的整数数列，请你计算数列中的逆序对的数量。逆序对的定义如下：对于数列的第 i 个和第 j 个元素，如果满足 $i < j$ 且 $a[i] > a[j]$，则其为一个逆序对；否则不是。

输入格式：

第一行包含整数 n，表示数列的长度。第二行包含 n 个整数，表示整个数列。

输出格式：

输出一个整数，表示逆序对的个数。

数据范围

$1 \leq n \leq 100000$，数列中的元素的取值范围 $[1, 10^9]$。

输入样例：	输出样例：
2 3 4 5 6 1	5

分治思路，将整个区间一分为二。考虑到归并排序的时候需要将两个有序数组合并，此时恰好可以做逆序对的统计。假设有一种算法，可以将数组排序的过程中统计该数组中的逆序对数量，则问题变为怎么统计两个有序数组中合起来的逆序对。

归并排序计算逆序对数量

```c
#include <stdio.h>
#include <stdlib.h>

long long merge_sort(int *q, int *tmp,
                     int l, int r) {
    if (l >= r) {
        return 0;
    }
    int mid = (l + r) >> 1;
    // 左侧的数组已统计逆序对且已经排序，右侧同样
    long long res = merge_sort(q, tmp, l, mid) + merge_sort(q, tmp, mid + 1, r);
    // 统计两个有序数组合起来的逆序对数量
    int i = l;
    int j = mid + 1;
    int k = 0;
    while (i <= mid && j <= r) {
        if (q[i] > q[j]) {
            res += mid - i + 1;
            tmp[k++] = q[j++];
        } else {
            tmp[k++] = q[i++];
        }
    }

    while (i <= mid) {
        tmp[k++] = q[i++];
    }
    while (j <= r) {
        tmp[k++] = q[j++];
    }
    for (i = l, j = 0; i <= r;
         i++, j++) {
        q[i] = tmp[j];
    }

    return res;
}

int main()
{
    int n;
    scanf("%d", &n);
    int *q = (int *)calloc(n, sizeof(int));
    if (q == NULL) {
        return -1;
    }
    int *tmp = (int *)calloc(n, sizeof(int));
    if (tmp == NULL) {
        free(q);
        return -1;
    }
    for(int i = 0; i < n; i++) {
        scanf("%d", &q[i]);
    }

    printf("%ld", merge_sort(q, tmp, 0, n - 1));

    return 0;
}
```

1.3 二分

整数二分和浮点数二分，二分即查找一个边界值，在左侧满足某种性质，右侧不满足。

二分用模版如下：

二分模版

```c
// 区间[l, r]被划分为[l, mid] 和
// [mid + 1, r]时使用，往左找
int bsearch_1(int l, int r)
{
    while (l < r) {
        mid = (l + r) / 2;
        if (Check(mid)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
}
```

```c
// 区间[l, r]被划分为[l, mid - 1] 和
// [mid, r]时使用，往右找
int bsearch_2(int l, int r)
{
    while (l < r) {
        mid = (l + r + 1) / 2;
        if (Check(mid)) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
}
```

每次要保证答案在区间中。

第二个模版加一的原因在于，如果某次循环结束后，l = r − 1，如果不加 1，此时因为向下取整的缘故mid = l，check 成功后l被再次赋值为mid即l，则此时进入死循环。

1.3.1 AcWing 789. 数的范围

AcWing 789. 数的范围

给定一个按照升序排列的长度为 n 的整数数组，以及 q 个查询。对于每个查询，返回一个元素 k 的起始位置和终止位置（位置从 0 开始计数）。如果数组中不存在该元素，则返回−1 −1。

输入格式：

第一行包含整数 n 和 q，表示数组长度和询问个数。第二行包含 n 个整数（均在 $1 \sim 10000$ 范围内），表示完整数组。接下来 q 行，每行包含一个整数 k，表示一个询问元素。

输出格式：

共 q 行，每行包含两个整数，表示所求元素的起始位置和终止位置。

数据范围 $1 \leq n \leq 100000$，$1 \leq q \leq 10000$，$1 \leq k \leq 10000$

输入样例：	输出样例：
6 3	3 4
1 2 2 3 3 4	5 5
3	-1 -1
4	
5	

binary search

```c
#include <stdio.h>
#include <stdlib.h>

int b_search_l(int *q, int l,
               int r, int t) {
    while (l < r) {
        int mid = (l + r) >> 1;
        if (q[mid] >= t) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
    if (q[l] != t) {
        return -1;
    }
    return l;
}

int b_search_r(int *q, int l,
               int r, int t) {
    while (l < r) {
        int mid = (l + r) / 2 + 1;
        if (q[mid] <= t) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
    if (q[l] != t) {
        return -1;
    }
    return l;
}

int main()
{
    int n;
    int q;
    scanf("%d %d", &n, &q);
    int *q = (int *)calloc(sizeof(int), n);
    if (q == NULL) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", q + i);
    }
    while(q--) {
        int t;
        scanf("%d", &t);
        printf("%d %d\n",
        b_search_l(q, 0, n - 1, t),
        b_search_r(q, 0, n - 1, t));
    }
    return 0;
}
```

这里不能使用bsearch函数来完成左端点的搜索，因为该函数在面对重复值时返回值不确定，是未定义行为。

1.3.2 AcWing 790. 数的三次方根

AcWing 790. 数的三次方根

给定一个浮点数 n，求它的三次方根。

输入格式：

共一行，包含一个浮点数 n。

输出格式：

共一行，包含一个浮点数，表示问题的解。

注意，结果保留 6 位小数。

数据范围：

$-10000 \leq n \leq 10000$

输入样例：	输出样例：
1000.00	10.000000

数的三次方根

```c
#include <stdio.h>

#define N 10000

int main()
{
    double n;
    scanf("%lf", &n);
    double l = 0 - N;
    double r = N;

    while (r - l > 1e-8) {
        double mid = (l + r) / 2;
        if (mid * mid * mid < n) {
            l = mid;
        } else {
            r = mid;
        }
    }
    printf("%.6f", l);
    return 0;
}
```

1.4 高精度

1.5 前缀和与差分

1. 前缀和可方便地求取数组中某个区间的元素和，或者矩阵的子矩阵的和 O(1)

2. 差分和方便的将数组某个区间的所有元素加上一个数，O(1) 时间复杂度

1.5.1 AcWing 795. 前缀和

AcWing 795. 前缀和

输入一个长度为 n 的整数序列。接下来再输入 m 个询问，每个询问输入一对 l, r。对于每个询问，输出原序列中从第 l 个数到第 r 个数的和。

输入格式：

第一行包含两个整数 n 和 m。第二行包含 n 个整数，表示整数数列。接下来 m 行，每行包含两个整数 l 和 r，表示一个询问的区间范围。

输出格式：

共 m 行，每行输出一个询问的结果。

数据范围：

$1 \leq l \leq r \leq n$, $1 \leq n, m \leq 100000$, $-1000 \leq$ 数列中元素的值 ≤ 1000

输入样例：

```
5 3
2 1 3 6 4
1 2
1 3
2 4
```

输出样例：

```
3
6
10
```

前缀和

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    int *q = (int *)calloc(n + 1, sizeof(int));
    int *preSum = (int *)calloc(n + 1, sizeof(int));
    for (int i = 1; i <= n; i++) {
        scanf("%d", q + i);
        preSum[i] = preSum[i - 1] + q[i];
    }
    while (m--) {
        int l;
        int r;
        scanf("%d %d", &l, &r);
        printf("%d\n", preSum[r] - preSum[l - 1]);
    }
    return 0;
}
```

1.5.2 AcWing 796. 子矩阵的和

子矩阵

AcWing 796. 子矩阵的和

子矩阵的和 输入一个 n 行 m 列的整数矩阵，再输入 q 个询问，每个询问包含四个整数 x_1, y_1, x_2, y_2，表示一个子矩阵的左上角坐标和右下角坐标。对于每个询问输出子矩阵中所有数的和。

输入格式：

第一行包含三个整数 n, m, q。接下来 n 行，每行包含 m 个整数，表示整数矩阵。接下来 q 行，每行包含四个整数 x_1, y_1, x_2, y_2，表示一组询问。

输出格式：

共 q 行，每行输出一个询问的结果。

数据范围：

$1 \le n, m \le 1000$, $1 \le q \le 200000$, $1 \le x_1 \le x_2 \le n$, $1 \le y_1 \le y_2 \le m$, $-1000 \le \ \le 1000$

输入样例：

```
3 4 3
1 7 2 4
3 6 2 8
2 1 2 3
1 1 2 2
2 1 3 4
1 3 3 4
```

输出样例：

```
17
27
21
```

子矩阵的和

```cpp
#include<stdio.h>

#define N 1010

int mat[N][N];
int preMat[N][N];

int main()
{
    int n, m, q;
    scanf("%d %d %d", &n, &m, &q);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            scanf("%d", &mat[i][j]);
            preMat[i][j] = preMat[i - 1][j] + preMat[i][j - 1] - preMat[i - 1][j - 1] + mat[i][j];
        }
    }
    while (q--) {
        int x1, y1, x2, y2;
        scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
        printf("%d\n", preMat[x2][y2] - preMat[x2][y1 - 1] - preMat[x1 - 1][y2] + preMat[x1 - 1][y1 - 1]);
    }
    return 0;
}
```

1.5.3 AcWing 797. 差分

AcWing 797. 差分

输入一个长度为 n 的整数序列。接下来输入 m 个操作，每个操作包含三个整数 l, r, c，表示将序列中 $[l, r]$ 之间的每个数加上 c。请你输出进行完所有操作后的序列。

输入格式：

第一行包含两个整数 n 和 m。第二行包含 n 个整数，表示整数序列。接下来 m 行，每行包含三个整数 lrc，表示一个操作。

输出格式：

共一行，包含 n 个整数，表示最终序列。

数据范围：

$1 \leq n, m \leq 100000$, $1 \leq l \leq r \leq n$, $-1000 \leq c \leq 1000$, $-1000 \leq$ 整数序列中元素的值 ≤ 1000

输入样例：

```
6 3
1 2 2 1 2 1
1 3 1
3 5 1
1 6 1
```

输出样例：

```
3 4 5 3 4 2
```

差分

```c
#include <stdio.h>
#include <stdlib.h>

void Insert(int *diff, int l, int r, int c)
{
    diff[l] += c;
    diff[r + 1] -= c;
}

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    int *q = (int *)calloc(n + 1, sizeof(int));
    int *diff = (int *)calloc(n + 1, sizeof(int));
    for (int i = 1; i <= n; i++) {
        scanf("%d", q + i);
        Insert(diff, i, i, q[i]);
    }
    while (m--) {
        int l;
        int r;
        int c;
        scanf("%d %d %d", &l, &r, &c);
        Insert(diff, l, r, c);
    }
    for (int i = 1; i <= n; i++) {
        diff[i] += diff[i - 1];
        printf("%d ", diff[i]);
    }
    free(diff);
    free(q);
    return 0;
}
```

1.5.4 AcWing 798. 差分矩阵

AcWing 798. 差分矩阵

输入一个 n 行 m 列的整数矩阵，再输入 q 个操作，每个操作包含五个整数 x_1, y_1, x_2, y_2, c，其中 (x_1, y_1) 和 (x_2, y_2) 表示一个子矩阵的左上角坐标和右下角坐标。每个操作都要将选中的子矩阵中的每个元素的值加上 c。请你将进行完所有操作后的矩阵输出。

输入格式：

第一行包含整数 n, m, q。接下来 n 行，每行包含 m 个整数，表示整数矩阵。接下来 q 行，每行包含 5 个整数 x_1, y_1, x_2, y_2, c，表示一个操作。

输出格式：

共 n 行，每行 m 个整数，表示所有操作进行完毕后的最终矩阵。

数据范围：

$1 \le n, m \le 1000$, $1 \le q \le 100000$, $1 \le x_1 \le x_2 \le n$, $1 \le y_1 \le y_2 \le m$, $-1000 \le c \le 1000$, $-1000 \le \ \le 1000$

输入样例：

```
3 4 3
1 2 2 1
3 2 2 1
1 1 1 1
1 1 2 2 1
1 3 2 3 2
3 1 3 4 1
```

输出样例：

```
4 3 4 1
2 2 2 2
```

差分矩阵

```cpp
#include <stdio.h>
#include <stdlib.h>

#define N 1010

int arr[N][N];
int diff[N][N];

void Insert(int x1, int y1, int x2, int y2, int c)
{
    diff[x1][y1] += c;
    diff[x1][y2 + 1] -= c;
    diff[x2 + 1][y1] -= c;
    diff[x2 + 1][y2 + 1] += c;
}

int main()
{
    int n;
    int m;
    int q;
    scanf("%d %d %d", &n, &m, &q);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            scanf("%d", &arr[i][j]);
            Insert(i, j, i, j, arr[i][j]);
        }
    }
    while (q--) {
        int x1, y1, x2, y2, c;
        scanf("%d %d %d %d %d", &x1, &y1, &x2, &y2, &c);
        Insert(x1, y1, x2, y2, c);
    }

    // construct origin matrix
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            diff[i][j] += diff[i - 1][j] + diff[i][j - 1] - diff[i - 1][j - 1];
            printf("%d ", diff[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

1.6 双指针算法

归并排序中的双指针，分别指向两个序列；指向同一个序列的不同位置，同向移动或者相向而行

1.6.1 AcWing 799. 最长连续不重复子序列

AcWing 799. 最长连续不重复子序列

给定一个长度为 n 的整数序列，请找出最长的不包含重复的数的连续区间，输出它的长度。

输入格式：

第一行包含整数 n。第二行包含 n 个整数（均在 $0 \sim 10^5$ 范围内），表示整数序列。

输出格式：

共一行，包含一个整数，表示最长的不包含重复的数的连续区间的长度。

数据范围：

$1 \leq n \leq 10^5$

输入样例：
5
1 2 2 3 5

输出样例：
3

滑动窗口，需要有个 visited 数组来统计每个元素的出现数量

最长连续不重复子序列

```c
#include <stdio.h>
#include <stdlib.h>

#define N 100010
int visited[N];

int main()
{
    int n;
    scanf("%d", &n);
    int *q = (int *)calloc(n, sizeof(int));
    if (q == NULL) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", &q[i]);
    }
    int start = 0;
    int end = 0;
    int max = 0;
    for (end = 0; end < n; end++) {
        // 标记当前元素
        visited[q[end]] += 1;
        while (visited[q[end]] > 1) {
            // 如果当前元素重复出现，缩小窗口
            visited[q[start]]--;
            start++;
        }
        max = max > (end - start + 1) ? max : (end - start + 1);
    }
    printf("%d", max);
    free(arr);
    return 0;
}
```

1.6.2 AcWing 800. 数组元素的目标和

AcWing 800. 数组元素的目标和

给定两个升序排序的有序数组 A 和 B，以及一个目标值 x。数组下标从 0 开始。请你求出满足 $A[i]+B[j]=x$ 的数对 (i,j)。数据保证有唯一解。

输入格式：

第一行包含三个整数 n,m,x，分别表示 A 的长度，B 的长度以及目标值 x。第二行包含 n 个整数，表示数组 A。第三行包含 m 个整数，表示数组 B。

输出格式：

共一行，包含两个整数 i 和 j。

数据范围：

数组长度不超过 10^5。同一数组内元素各不相同。$1 \leq$ 数组元素 $\leq 10^9$

输入样例：

```
4 5 6
1 2 4 7
3 4 6 8 9
```

输出样例：

```
1 1
```

数组元素的目标和

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    int m;
    int x;
    scanf("%d %d %d", &n, &m, &x);
    int *a = (int *)calloc(n, sizeof(int));
    int *b = (int *)calloc(m, sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", a + i);
    }
    for (int i = 0; i < m; i++) {
        scanf("%d", b + i);
    }

    int i = 0;
    int j = m - 1;
    while (i < n && j >= 0) {
        int sum = a[i] + b[j];
        if (sum == x) {
            printf("%d %d", i, j);
            return 0;
        } else if (sum > x) {
            j--;
        } else {
            i++;
        }
    }
    free(a);
    free(b);
    return 0;
}
```

1.6.3 AcWing 2816. 判断子序列

AcWing 2816. 判断子序列

给定一个长度为 n 的整数序列 $a_1, a_2, \ldots;, a_n$ 以及一个长度为 m 的整数序列 $b_1, b_2, \ldots;, b_m$。</p>
请你判断 a 序列是否为 b 序列的子序列。子序列指序列的一部分项按**原有次序排列**而得的序列，例如序列 $\{a_1, a_3, a_5\}$ 是序列 $\{a_1, a_2, a_3, a_4, a_5\}$ 的一个子序列。

输入格式：

第一行包含两个整数 n, m。第二行包含 n 个整数，表示 $a_1, a_2, \ldots;, a_n$。第三行包含 m 个整数，表示 $b_1, b_2, \ldots;, b_m$。

输出格式：

如果 a 序列是 b 序列的子序列，输出一行 Yes。否则，输出 No。

数据范围：

$1 \le n \le m \le 10^5$, $-10^9 \le a_i, b_i \le 10^9$

输入样例：

```
3 5
1 3 5
1 2 3 4 5
```

输出样例：

```
Yes
```

判断子序列

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool IsSubSeq(int *a, int n, int *b, int m)
{
    int i = 0;
    int j = 0;
    while (i < n && j < m) {
        if (a[i] == b[j]) {
            i++;
            j++;
        } else {
            j++;
        }
    }
    return i == n;
}

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    int *a = (int *)calloc(n, sizeof(int));
    int *b = (int *)calloc(m, sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", a + i);
    }
    for (int i = 0; i < m; i++) {
        scanf("%d", b + i);
    }
    printf("%s",
      IsSubSeq(a, n, b, m) ? "Yes" : "No");
    free(a);
    free(b);
    return 0;
}
```

1.7 位运算

1.7.1 AcWing 801. 二进制中 1 的个数

AcWing 801. 二进制中 1 的个数

给定一个长度为 n 的数列，请你求出数列中每个数的二进制表示中 1 的个数。

输入格式：

第一行包含整数 n。第二行包含 n 个整数，表示整个数列。

输出格式：共一行，包含 n 个整数，其中的第 i 个数表示数列中的第 i 个数的二进制表示中 1 的个数。

数据范围：

$1 \leq n \leq 100000$, $0 \leq$ 数列中元素的值 $\leq 10^9$

输入样例：
5
1 2 3 4 5

输出样例：
1 1 2 1 2

LowBit 运算

```cpp
#include <stdio.h>
#include <stdlib.h>

int LowBit(int x)
{
    return x & (-x);
}

int Count(int x)
{
    int cnt = 0;
    while (x != 0) {
        cnt++;
        x -= LowBit(x);
    }
    return cnt;
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--) {
        int x;
        scanf("%d", &x);
        printf("%d ", Count(x));
    }
    return 0;
}
```

1.8 离散化

1.8.1 AcWing 802. 区间和

AcWing 802. 区间和

假定有一个无限长的数轴，数轴上每个坐标上的数都是 0。现在，我们首先进行 n 次操作，每次操作将某一位置 x 上的数加 c。接下来，进行 m 次询问，每个询问包含两个整数 l 和 r，你需要求出在区间 $[l,r]$ 之间的所有数的和。

输入格式：

第一行包含两个整数 n 和 m。接下来 n 行，每行包含两个整数 x 和 c。再接下来 m 行，每行包含两个整数 l 和 r。

输出格式：

共 m 行，每行输出一个询问中所求的区间内数字和。

数据范围：

$-10^9 \le x \le 10^9$, $1 \le n,m \le 10^5$, $-10^9 \le l \le r \le 10^9$,$-10000 \le c \le 10000$

输入样例：

```
3 3
1 2
3 6
7 5
1 3
4 6
7 8
```

输出样例：

```
8
0
5
```

1.9 区间合并

1.9.1 AcWing 803. 区间合并

AcWing 803. 区间合并

给定 n 个区间 $[l_i, r_i]$，要求合并所有有交集的区间。注意如果在端点处相交，也算有交集。输出合并完成后的区间个数。例如：$[1, 3]$ 和 $[2, 6]$ 可以合并为一个区间 $[1, 6]$。

输入格式：

第一行包含整数 n。接下来 n 行，每行包含两个整数 l 和 r。

输出格式：

共一行，包含一个整数，表示合并区间完成后的区间个数。

数据范围：

$1 \leq n \leq 100000$, $-10^9 \leq l_i \leq r_i \leq 10^9$

输入样例：

```
5
1 2
2 4
5 6
7 8
7 9
```

输出样例：

```
3
```

区间合并

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int l;
    int r;
} Seg;

int Max(int a, int b)
{
    return a > b ? a : b;
}

int CmpFunc(const void *a, const void *b)
{
    return ((Seg *)a)->l - ((Seg *)b)->l;
}

int main()
{
    int n;
    scanf("%d", &n);
    Seg *segs = (Seg *)calloc(n, sizeof(Seg));
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &(segs[i].l), &(segs[i].r));
    }
    qsort(segs, n, sizeof(Seg), CmpFunc);
    Seg init = segs[0];
    int ans = 1;
    for (int i = 1; i < n; i++) {
        if (segs[i].l > init.r) {
            ans++;
            init = segs[i];
        } else {
            init.r = Max(segs[i].r, init.r);
        }
    }
    printf("%d", ans);
    return 0;
}
```

第二章

数据结构

2.1 单链表

2.1.1 AcWing 826. 单链表

AcWing 826. 单链表

实现一个单链表，链表初始为空，支持三种操作：

1. 向链表头插入一个数；
2. 删除第 k 个插入的数后面的数；
3. 在第 k 个插入的数后插入一个数。

现在要对该链表进行 M 次操作，进行完所有操作后，从头到尾输出整个链表。**注意**: 题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，... 第 n 个插入的数。

输入格式：

第一行包含整数 M，表示操作次数。接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1. H x，表示向链表头插入一个数 x。
2. D k，表示删除第 k 个插入的数后面的数（当 k 为 0 时，表示删除头结点）。
3. I k x，表示在第 k 个插入的数后面插入一个数 x（此操作中 k 均大于 0）。

输出格式：

共一行，将整个链表从头到尾输出。

数据范围：

$1 \leq M \leq 100000$ 所有操作保证合法。

输入样例：
```
10
H 9
I 1 1
D 1
D 0
H 6
I 3 6
I 4 5
I 4 5
I 3 4
D 6
```

输出样例：
```
6 4 6 5
```

linked list

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define N 100010
int idx; // 标识当前用到了哪个节点
int head; // 头节点指向的元素
int e[N]; // 存储节点的值
int ne[N]; // 存储节点的next指针

// 初始化
void Init()
{
    idx = -1;
    head = -1;
}

// 在头节点后插入元素
void AddHead(int x) {
    e[++idx] = x;
    ne[idx] = head;
    head = idx;
}

// 在k节点之后插入元素
void Add(int k, int x)
{
    e[++idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
}

// 删除k之后的那个元素
void Remove(int k) {
    ne[k] = ne[ne[k]];
}

int main()
{
    Init();
    int n;
    scanf("%d", &n);
    while (n--) {
        int x;
        int k;
        char op;
        scanf(" %c", &op);
        if (op == 'H') {
            scanf("%d", &x);
            AddHead(x);
        }
        if (op == 'I') {
            scanf("%d %d", &k, &x);
            Add(k - 1, x);
        }
        if (op == 'D') {
            scanf("%d", &k);
            if (k == 0) {
                head = ne[head];
            }
            Remove(k - 1);
        }
    }

    int tmp = head;
    while (tmp != -1) {
        printf("%d ", e[tmp]);
        tmp = ne[tmp];
    }
    return 0;
}
```

2.2 双链表

2.2.1 AcWing 827. 双链表

AcWing 827. 双链表

实现一个双链表，双链表初始为空，支持 5 种操作：

1. 在最左侧插入一个数；
2. 在最右侧插入一个数；
3. 将第 k 个插入的数删除；
4. 在第 k 个插入的数左侧插入一个数；
5. 在第 k 个插入的数右侧插入一个数。

现在要对该链表进行 M 次操作，进行完所有操作后，从左到右输出整个链表。**注意**: 题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，…第 n 个插入的数。

输入格式：

第一行包含整数 M，表示操作次数。接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1. L x，表示在链表的最左端插入数 x。
2. R x，表示在链表的最右端插入数 x。
3. D k，表示将第 k 个插入的数删除。
4. IL k x，表示在第 k 个插入的数左侧插入一个数。
5. IR k x，表示在第 k 个插入的数右侧插入一个数。

输出格式：一行，将整个链表从左到右输出

数据范围：

$1 \leq M \leq 100000$ 所有操作保证合法。

输入样例：

```
10
R 7
D 1
L 3
IL 2 10
D 3
IL 2 7
L 8
R 9
IL 4 7
IR 2 2
```

输出样例：

```
8 7 7 3 2 9
```

双链表

```c
#include <stdio.h>
#include <stdlib.h>

#define N 100010

int e[N];
int l[N];
int r[N];
int idx = 1;

void Init()
{
    r[0] = 1;
    l[1] = 0;
    idx = 1;
}

void AddRight(int k, int x)
{
    e[++idx] = x;
    r[idx] = r[k];
    l[r[k]] = idx;
    l[idx] = k;
    r[k] = idx;
}

void Remove(int k)
{
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}

int main()
{
    Init();
    int n;
    scanf("%d", &n);
    while (n--) {
        char op[3];
        scanf("%s", op);
        int k;
        int x;
        if (op[0] == 'R') {
            scanf("%d", &x);
            AddRight(l[1], x);
        }
        if (op[0] == 'D') {
            scanf("%d", &k);
            Remove(k + 1);
        }
        if (op[0] == 'L') {
            scanf("%d", &x);
            AddRight(0, x);
        }
        if (op[0] == 'I') {
            scanf("%d %d", &k, &x);
            if (op[1] == 'L') {
                AddRight(l[k + 1], x);
            } else {
                AddRight(k + 1, x);
            }
        }
    }
    for (int i = r[0]; i != 1; i = r[i]) {
        printf("%d ", e[i]);
    }
    return 0;
}
```

2.3 栈

2.3.1 AcWing 828. 模拟栈

AcWing 828. 模拟栈

实现一个栈，栈初始为空，支持四种操作：

1. push x - 向栈顶插入一个数 x；
2. pop - 从栈顶弹出一个数；
3. empty - 判断栈是否为空；
4. query - 查询栈顶元素

现在要对栈进行 M 个操作，其中的每个操作 3 和操作 4 都要输出相应的结果。

输入格式：

第一行包含整数 M，表示操作次数。接下来 M 行，每行包含一个操作命令，操作命令为 push x，pop，empty，query 中的一种。

输出格式：

对于每个 empty 和 query 操作都要输出一个查询结果，每个结果占一行。其中，empty 操作的查询结果为 YES 或 NO，query 操作的查询结果为一个整数，表示栈顶元素的值。

数据范围：

$1 \le M \le 100000$, $1 \le x \le 10^9$ 所有操作保证合法。

输入样例：

```
10
push 5
query
push 6
pop
query
pop
empty
push 4
query
empty
```

输出样例：

```
5
5
YES
4
NO
```

模拟栈

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    scanf("%d", &n);
    int *stack = (int *)calloc(n, sizeof(int));
    int top = 0;
    while (n--) {
        char op[10];
        scanf("%s", op);
        int x;
        if (strcmp(op, "push") == 0) {
            scanf("%d", &x);
            stack[top++] = x;
        }
        if (strcmp(op, "query") == 0) {
            printf("%d\n", stack[top - 1]);
        }
        if (strcmp(op, "pop") == 0) {
            top--;
        }
        if (strcmp(op, "empty") == 0) {
            printf("%s\n", top == 0 ? "YES" : "NO");
        }
    }
    free(stack);
    return 0;
}
```

2.3.2 AcWing 3302. 表达式求值

2.4 队列

2.4.1 AcWing 829. 模拟队列

AcWing 829. 模拟队列

实现一个队列，队列初始为空，支持四种操作：

1. push x - 向队尾插入一个数 x；
2. pop - 从队头弹出一个数；
3. empty - 判断队列是否为空；
4. query - 查询队头元素

现在要对队列进行 M 个操作，其中的每个操作 3 和操作 4 都要输出相应的结果。

输入格式：

第一行包含整数 M，表示操作次数。接下来 M 行，每行包含一个操作命令，操作命令为 push x，pop，empty，query 中的一种。

输出格式：

对于每个 empty 和 query 操作都要输出一个查询结果，每个结果占一行。其中，empty 操作的查询结果为 YES 或 NO，query 操作的查询结果为一个整数，表示栈顶元素的值。

数据范围：

$1 \le M \le 100000$, $1 \le x \le 10^9$ 所有操作保证合法。

输入样例：

```
10
push 6
empty
query
pop
empty
push 3
push 4
pop
query
push 6
```

输出样例：

```
NO
YES
6
4
```

模拟队列

```c
#include <stdio.h>
#include <stdbool.h>

#define N 100010
int queue[N];
int tail;
int head;

void push(int x)
{
    queue[++tail] = x;
}

bool isEmpty()
{
    return head == tail;
}

void pop()
{
    head++;
}

int query()
{
    return queue[head + 1];
}

int main()
{
    int n;
    scanf("%d", &n);
    char op[10];
    while (n--) {
        scanf("%s", op);
        if (strcmp(op, "push") == 0) {
            int x;
            scanf("%d", &x);
            push(x);
        }
        if (strcmp(op, "pop") == 0) {
            pop();
        }
        if (strcmp(op, "empty") == 0) {
            printf("%s\n", isEmpty() ? "YES" : "NO");
        }
        if (strcmp(op, "query") == 0) {
            printf("%d\n", query());
        }
    }
    return 0;
}
```

2.5 单调栈

2.5.1 AcWing 830. 单调栈

AcWing 830. 单调栈

给定一个长度为 N 的整数数列，输出每个数左边第一个比它小的数，如果不存在则输出 -1。

输入格式：

第一行包含整数 N，表示数列长度。第二行包含 N 个整数，表示整数数列。

输出格式：

共一行，包含 N 个整数，其中第 i 个数表示第 i 个数的左边第一个比它小的数，如果不存在则输出 -1。

数据范围：

$1 \leq N \leq 10^5$，$1 \leq$ 数列中元素 $\leq 10^9$

输入样例：
5
3 4 2 7 5

输出样例：
-1 3 -1 2 2

单调栈

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    scanf("%d", &n);
    int *stack = (int *)calloc(sizeof(int), n);
    int top = 0;
    while (n--) {
        int x;
        scanf("%d", &x);
        while (top != 0 && stack[top] >= x)
        {
            top--;
        }
        if (top == 0) {
            printf("%d ", -1);
        } else {
            printf("%d ", stack[top]);
        }
        stack[++top] = x;
    }
    free(stack);
    return 0;
}
```

2.6 单调队列

2.6.1 AcWing 154. 滑动窗口

AcWing 154. 滑动窗口

给定一个大小为 $n \leq 10^6$ 的数组。有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边。你只能在窗口中看到 k 个数字。每次滑动窗口向右移动一个位置。

以下是一个例子：

该数组为 $[1\ \ 3\ \ -1\ \ -3\ 5\ 3\ 6\ 7]$，$k$ 为 3。

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

你的任务是确定滑动窗口位于每个位置时，窗口中的最大值和最小值。

输入格式：输入包含两行。第一行包含两个整数 n 和 k，分别代表数组长度和滑动窗口的长度。第二行有 n 个整数，代表数组的具体数值。同行数据之间用空格隔开。

输出格式：输出包含两个。第一行输出，从左至右，每个位置滑动窗口中的最小值。第二行输出，从左至右，每个位置滑动窗口中的最大值。

输入样例：
8 3
1 3 -1 -3 5 3 6 7

输出样例：
-1 -3 -3 -3 3 3
3 3 5 5 6 7

滑动窗口:单调队列

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    int k;
    scanf("%d %d", &n, &k);
    int *a = (int *)calloc(sizeof(int), n);
    for (int i = 0; i < n; i++) {
        scanf("%d", a + i);
    }
    int *queue = (int *)calloc(sizeof(int), n);
    int head = 0;
    int tail = -1;
    for (int i = 0; i < n; i++) {
        if (head <= tail && i - k + 1 > queue[head]) {
            head++;
        }
        while (head <= tail && a[queue[tail]] >= a[i]) {
            tail--;
        }
        queue[++tail] = i;
        if (i >= k - 1) {
            printf("%d ", a[queue[head]]);
        }
    }
    puts("");
    head = 0;
    tail = -1;
    for (int i = 0; i < n; i++) {
        if (head <= tail && i - k + 1 > queue[head]) {
            head++;
        }
        while (head <= tail && a[queue[tail]] <= a[i]) {
            tail--;
        }
        queue[++tail] = i;
        if (i >= k - 1) {
            printf("%d ", a[queue[head]]);
        }
    }
    free(queue);
    free(a);
    return 0;
}
```

2.7 KMP

2.7.1 AcWing 831. KMP 字符串

AcWing 831. KMP 字符串

给定一个模式串 S,以及一个模板串 P,所有字符串中只包含大小写英文字母以及阿拉伯数字。模板串 P 在模式串 S 中多次作为子串出现。求出模板串 P 在模式串 S 中所有出现的位置的起始下标。

输入格式:

第一行输入整数 N,表示字符串 P 的长度。第二行输入字符串 P。第三行输入整数 M,表示字符串 S 的长度。第四行输入字符串 S。

输出格式:

共一行,输出所有出现位置的起始下标(下标从 0 开始计数),整数之间用空格隔开。

数据范围:

$1 \le N \le 10^5$,$1 \le M \le 10^6$

输入样例:

```
3
aba
5
ababa
```

输出样例:

```
0 2
```

KMP

2.8 Trie

高效地存储和查找字符串集合的数据结构，可以用传统结构体实现，亦可用数组来模拟指针。

2.8.1 AcWing 835. Trie 字符串统计

AcWing 835. Trie 字符串统计

维护一个字符串集合，支持两种操作：

I x 向集合中插入一个字符串 x；

Q x 询问一个字符串在集合中出现了多少次。

共有 N 个操作，输入的字符串总长度不超过 10^5，字符串仅包含小写英文字母。

输入格式：

第一行包含整数 N，表示操作数。接下来 N 行，每行包含一个操作指令，指令为 I x 或 Q x 中的一种。

输出格式：

对于每个询问指令 Q x，都要输出一个整数作为结果，表示 x 在集合中出现的次数。每个结果占一行。

数据范围：

$1 \leq N \leq 2 * 10^4$

输入样例：	输出样例：
5	1
I abc	0
Q abc	1
Q ab	
I ab	
Q ab	

这里有两种写法：链表形式和数组模拟的方式。

链表形式的 Trie

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define ALPH_NUM 26
#define N 100

typedef struct Node_ {
    struct Node_ *nodes[ALPH_NUM];
    int cnt;
    bool isEnd;
} Node;

void Insert(Node *head, char *str)
{
    int len = strlen(str);
    Node *tmp = head;
    for (int i = 0; i < len; i++) {
        int idx = str[i] - 'a';
        if (tmp->nodes[idx] == NULL) {
            tmp->nodes[idx] = (Node *)calloc(sizeof(Node), 1);
        }
        tmp = tmp->nodes[idx];
    }
    tmp->cnt += 1;
    tmp->isEnd = true;
}

int Query(Node *head, char *str)
{
    int len = strlen(str);
    Node *tmp = head;
    for (int i = 0; i < len; i++) {
        int idx = str[i] - 'a';
        if (tmp->nodes[idx] == NULL) {
            return 0;
        }
        tmp = tmp->nodes[idx];
    }
    return tmp->isEnd ? tmp->cnt : 0;
}

void Free(Node *head) {
    Node *tmp = head;
    for (int i = 0; i < ALPH_NUM; i++) {
        if (tmp->nodes[i] == NULL) {
            continue;
        }
        Free(tmp->nodes[i]);
    }
    free(tmp);
}

int main()
{
    int n;
    scanf("%d", &n);
    char op[2];
    char str[N];
    Node *head = (Node *)calloc(sizeof(Node), 1);
    while (n--) {
        scanf("%s %s", op, str);
        if (op[0] == 'I') {
            Insert(head, str);
        }
        if (op[0] == 'Q') {
            printf("%d\n", Query(head, str));
        }
    }
    Free(head);
    return 0;
}
```

数组模拟 Trie

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// define a macro for alphabets number
#define ALPHABETS 26
#define N 20010
#define MAX_BUF 100

int trie[N][ALPHABETS];
int idx;
int cnt[N];

// insert str to trie
void Insert(char *str) {
    int i;
    int len = strlen(str);
    int cur = 0;
    for (i = 0; i < len; i++) {
        int ch = str[i] - 'a';
        if (trie[cur][ch] == 0) {
            trie[cur][ch] = ++idx;
        }
        cur = trie[cur][ch];
    }
    cnt[cur]++;
}

// query str from trie
int Query(char *str) {
    int i;
    int len = strlen(str);
    int cur = 0;
    for (i = 0; i < len; i++) {
        int ch = str[i] - 'a';
        if (trie[cur][ch] == 0) {
            return 0;
        }
        cur = trie[cur][ch];
    }
    return cnt[cur];
}

int main()
{
    int n;
    scanf("%d", &n);
    char op[2];
    char str[MAX_BUF];
    while (n--) {
        scanf("%s %s", op, str);
        if (op[0] == 'I') {
            Insert(str);
        }
        if (op[0] == 'Q') {
            printf("%d\n", Query(str));
        }
    }

    return 0;
}
```

根据上述两种方法，其实数组模拟 Trie 和链表形式的 Trie 在本质上也只是一个使用数组模拟的链表，所以对应到之前数组模拟的单链表可知：数组模拟无非就是将链表节点展平后每个数组元素存储下一个链表节点对应的数组下标。

2.8.2 AcWing 143. 最大异或对

143. 最大异或对

在给定的 N 个整数 $A_1, A_2, \ldots A_N$ 中选出两个进行 xor（异或）运算，得到的结果最大是多少？

输入格式：

第一行输入一个整数 N。第二行输入 N 个整数 $A_1, \ldots A_N$。

输出格式：

输出一个整数表示答案。

数据范围：

$1 \leq N \leq 10^5$, $0 \leq A_i < 2^{31}$

输入样例：
3
1 2 3

输出样例：
3

同样的,这里采用两种写法：

链表形式的 Trie

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    struct Node *nexts[2];
} Trie;

// Insert int value into trie
void Insert(Trie *trie, int value) {
    Trie *tmp = trie;
    for (int i = 30; i >= 0; i--) {
        int bit = (value >> i) & 1;
        if (tmp->nexts[bit] == NULL) {
            tmp->nexts[bit] = calloc(sizeof(Trie), 1);
        }
        tmp = tmp->nexts[bit];
    }
}

// Query the trie to get the value which xor to the value get the max
int Query(Trie *trie, int value) {
    Trie *tmp = trie;
    int max = 0;
    for (int i = 30; i >= 0; i--) {
        int bit = (value >> i) & 1;
        if (tmp->nexts[!bit] != NULL) {
            max += (1 << i);
            tmp = tmp->nexts[!bit];
        } else {
            tmp = tmp->nexts[bit];
        }
    }
    return max;
}

// get max value from two int value
int Max(int a, int b) {
    return a > b ? a : b;
}

// free entire trie
void FreeTrie(Trie *trie) {
    for (int i = 0; i < 2; i++) {
        if (trie->nexts[i] != NULL) {
            FreeTrie(trie->nexts[i]);
        }
    }
    free(trie);
}

int main()
{
    int n;
    scanf("%d", &n);
    int *arr = (int *) calloc(n, sizeof(int));
    Trie *trie = (Trie *)calloc(1, sizeof(Trie));
    for (int i = 0; i < n; i++) {
        int x;
        scanf("%d", &x);
        arr[i] = x;
        Insert(trie, x);
    }
    int res = 0;
    for (int i = 0; i < n; i++) {
        res = Max(res, Query(trie, arr[i]));
    }
    printf("%d", res);
    FreeTrie(trie);
    free(arr);
    return 0;
}
```

2.9 并查集

两个操作：

1. 将两个集合合并
2. 询问两个元素是否在一个集合中

基本原理：每个集合用一棵树来表示。树根的编号就是整个集合的编号，每个节点存储它的父节点，p[x]标识 x的父节点

问题：

1. 如何判断树根：if (p[x] == x)

2. 如何求 x 的集合编号：while(p[x] != x) x = p[x]

3. 如何合并两个集合：p[p[x]] = p[y]

2.9.1 AcWing 836. 合并集合

AcWing 836. 合并集合

一共有 n 个数，编号是 $1 \sim n$，最开始每个数各自在一个集合中。现在要进行 m 个操作，操作共有两种：

M a b，将编号为 a 和 b 的两个数所在的集合合并，如果两个数已经在同一个集合中，则忽略这个操作；

Q a b，询问编号为 a 和 b 的两个数是否在同一个集合中；

输入格式：

第一行输入整数 n 和 m。接下来 m 行，每行包含一个操作指令，指令为 M a b 或 Q a b 中的一种。

输出格式：

对于每个询问指令 Q a b，都要输出一个结果，如果 a 和 b 在同一集合内，则输出 Yes，否则输出 No。每个结果占一行。

数据范围：

$1 \leq n, m \leq 10^5$

输入样例：

```
4 5
M 1 2
M 3 4
Q 1 2
Q 1 3
Q 3 4
```

输出样例：

```
Yes
No
Yes
```

合并集合

```c
#include <stdlib.h>
#include <stdio.h>

int Find(int *parent, int a)
{
    if (parent[a] != a) {
        parent[a] = Find(parent, parent[a]);
    }
    return parent[a];
}

void Union(int *parent, int a, int b)
{
    int aP = Find(parent, a);
    int bP = Find(parent, b);
    parent[aP] = bP;
}

int main()
{
    int m, n;
    scanf("%d %d", &m, &n);
    int *parent = (int *) calloc(n + 1,
    sizeof(int));
    if (parent == NULL) {
        return -1;
    }
    for (int i = 0; i < n + 1; i++) {
        parent[i] = i;
    }
    while (n--) {
        char op[2];
        int a, b;
        scanf("%s %d %d", op, &a, &b);
        if (op[0] == 'M') {
            Union(parent, a, b);
        }
        if (op[0] == 'Q') {
            printf("%s\n", Find(parent, a)
    == Find(parent, b) ? "Yes" : "No");
        }
    }
    free(parent);
    return 0;
}
```

2.9.2 AcWing 837. 连通块中点的数量

AcWing 837. 连通块中点的数量

给定一个包含 n 个点（编号为 $1 \sim n$）的无向图，初始时图中没有边。现在要进行 m 个操作，操作共有三种：

C a b，在点 a 和点 b 之间连一条边，a 和 b 可能相等；

Q1 a b，询问点 a 和点 b 是否在同一个连通块中，a 和 b 可能相等；

Q2 a，询问点 a 所在连通块中点的数量；

输入格式：

第一行输入整数 n 和 m。接下来 m 行，每行包含一个操作指令，指令为 C a b，Q1 a b 或 Q2 a 中的一种。

输出格式：

对于每个询问指令 Q1 a b，如果 a 和 b 在同一个连通块中，则输出 Yes ，否则输出 No 。对于每个询问指令 Q2 a ，输出一个整数表示点 a 所在连通块中点的数量，每个结果占一行。

数据范围：

$1 \leq n, m \leq 10^5$

输入样例：

```
5 5
C 1 2
Q1 1 2
Q2 1
C 2 5
Q2 5
```

输出样例：

```
Yes
2
3
```

```c
#include <stdio.h>

int Find(int *p, int a)
{
    if (p[a] != a) {
        p[a] = Find(p, p[a]);
    }
    return p[a];
}

void Union(int *p, int *cnt, int a, int b)
{
    int ap = Find(p, a);
    int bp = Find(p, b);
    if (ap != bp) {
        p[ap] = bp;
        cnt[bp] += cnt[ap];
    }
}

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    int *p = (int *)calloc(sizeof(int), n + 1);
    int *cnt = (int *)calloc(sizeof(int), n + 1);
    for (int i = 0; i <= n; i++) {
        p[i] = i;
        cnt[i] = 1;
    }
    char op[3];
    int a;
    int b;
    while (m--) {
        scanf("%s", op);
        if (op[0] == 'C') {
            scanf("%d %d", &a, &b);
            Union(p, cnt, a, b);
        }
        if (op[0] == 'Q') {
            if (op[1] == '1') {
                scanf("%d %d", &a, &b);
                printf("%s\n", Find(p, a) == Find(p, b) ? "Yes" : "No");
            } else {
                scanf("%d", &a);
                printf("%d\n", cnt[Find(p, a)]);
            }
        }
    }
    free(cnt);
    free(p);
    return 0;
}
```

此处维护了一个 cnt 数组来标记集合中的元素个数，仅保证根节点的 cnt 是有意义的。

2.9.3 AcWing 240. 食物链

2.10 堆

用一维数组来模拟一个堆，起始坐标从 1 开始，因为 x 节点的左儿子是 2x 右儿子是 2x + 1 ，不能用 0 开始。几种操作的实现如下：

1. 插入一个数：heap[++size] = x; up(size);
2. 求集合中的最小值：heap[1];
3. 删除最小值：heap[1] = heap[size]; size−−; down(1);
4. 删除任意一个元素：heap[k] = heap[size]; size−−; down(k); up(k);
5. 修改任意一个元素：heap[k] = x; down(k); up(k);

2.10.1 AcWing 838. 堆排序

此时只涉及到 2 和 3 两个操作即可，故仅需使用 down 操作，无需 up 操作。

AcWing 838. 堆排序

输入一个长度为 n 的整数数列，从小到大输出前 m 小的数。

输入格式：

第一行包含整数 n 和 m。第二行包含 n 个整数，表示整数数列。

输出格式：共一行，包含 m 个整数，表示整数数列中前 m 小的数。

数据范围：

$1 \leq m \leq n \leq 10^5$，$1 \leq \ \leq 10^9$

输入样例：
5 3
4 5 1 3 2

输出样例：
1 2 3

堆排序

```c
#include <stdio.h>
#include <stdlib.h>

#define N 100010
int h[N];
int size;

void down(int k)
{
    int t = k;
    if (2 * k <= size && h[2 * k] < h[t]) {
        t = 2 * k;
    }
    if (2 * k + 1 <= size && h[2 * k + 1] < h[t]) {
        t = 2 * k + 1;
    }
    if (t != k) {
        int tmp = h[t];
        h[t] = h[k];
        h[k] = tmp;
        down(t);
    }
}

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    for (int i = 1; i < n + 1; i++) {
        scanf("%d", &h[i]);
    }
    size = n;
    for (int i = n / 2; i > 0; i--) {
        down(i);
    }

    while (m--) {
        printf("%d ", h[1]);
        h[1] = h[size];
        size--;
        down(1);
    }
    return 0;
}
```

2.10.2 AcWing 839. 模拟堆

此处为了支持删除和修改任意一个元素，用了两个数组 hp 和 ph 来保存从堆（h，heap）到插入的操作数（p，pointer）之间的映射关系。

AcWing 839. 模拟堆

维护一个集合，初始时集合为空，支持如下几种操作：

1. I x，插入一个数 x；
2. PM，输出当前集合中的最小值；
3. DM，删除当前集合中的最小值（数据保证此时的最小值唯一）；
4. D k，删除第 k 个插入的数；
5. C k x，修改第 k 个插入的数，将其变为 x；

现在要进行 N 次操作，对于所有第 2 个操作，输出当前集合的最小值。

输入格式：

第一行包含整数 N。接下来 N 行，每行包含一个操作指令，操作指令为 I x，PM，DM，D k 或 C k x 中的一种。

输出格式：

对于每个输出指令 PM，输出一个结果，表示当前集合中的最小值。每个结果占一行。

数据范围

$1 \leq N \leq 10^5, 10^9 \leq x \leq 10^9$

数据保证合法。

输入样例：

```
8
I -10
PM
I -10
D 1
C 2 8
I 6
PM
DM
```

输出样例：

```
-10
6
```

可修改任意元素的堆

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 100010
int heap[N];
int hp[N];
int ph[N];
int size;

void Swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void HeapSwap(int k, int t)
{
    Swap(&ph[hp[k]], &ph[hp[t]]);
    Swap(&hp[k], &hp[t]);
    Swap(&heap[k], &heap[t]);
}

void Up(int k)
{
    while (k / 2 > 0 &&
        heap[k / 2] > heap[k]) {
        HeapSwap(k, k / 2);
        k >>= 1;
    }
}

void Down(int k)
{
    int t = k;
    if (2 * k <= size &&
        heap[2 * k] < heap[t]) {
        t = 2 * k;
    }
    if (2 * k + 1 <= size &&
        heap[2 * k + 1] < heap[t]) {
        t = 2 * k + 1;
    }
    if (k != t) {
        HeapSwap(k, t);
        Down(t);
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    int idx = 0;
    while (n--) {
        char op[3];
        scanf("%s", op);
        int x;
        if (strcmp(op, "I") == 0) {
            scanf("%d", &x);
            heap[++size] = x;
            hp[size] = ++idx;
            ph[idx] = size;
            Up(size);
        }
        if (strcmp(op, "PM") == 0) {
            printf("%d\n", heap[1]);
        }
        if (strcmp(op, "DM") == 0) {
            HeapSwap(1, size);
            size--;
            Down(1);
        }
        if (strcmp(op, "D") == 0) {
            int k;
            scanf("%d", &k);
            k = ph[k];
            HeapSwap(k, size--);
            Down(k);
            Up(k);
        }
        if (strcmp(op, "C") == 0) {
            int k;
            scanf("%d %d", &k, &x);
            k = ph[k];
            heap[k] = x;
            Down(k);
            Up(k);
        }
    }
    return 0;
}
```

2.11 哈希表

2.11.1 AcWing 840. 模拟散列表

AcWing 840. 模拟散列表

维护一个集合，支持如下几种操作：

I x，插入一个数 x；

Q x，询问数 x 是否在集合中出现过；

现在要进行 N 次操作，对于每个询问操作输出对应的结果。

输入格式：

第一行包含整数 N，表示操作数量。接下来 N 行，每行包含一个操作指令，操作指令为 I x，Q x 中的一种。

输出格式：

对于每个询问指令 Q x，输出一个询问结果，如果 x 在集合中出现过，则输出 Yes，否则输出 No。每个结果占一行。

数据范围：

$1 \leq N \leq 10^5, -10^9 \leq x \leq 10^9$

输入样例：

```
5
I 1
I 2
I 3
Q 2
Q 5
```

输出样例：

```
Yes
No
```

两种写法：

1. 开放寻址法

此法将从哈希后的那个位置开始查找，如果已经有了不是自己的人占用该位置则继续向后寻找，直到找到一个空位置，返回该位置。此法中 Find 函数将返回 x应该存放的位置。此法仅需开一个数组即可，通常来讲要开数据规模的两倍到三倍，以防 buffer overflow。

开放寻址法

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 200003

int h[N];
const int null = 0x3f3f3f3f;

int Find(int x)
{
    int k = (x % N + N) % N;
    while (h[k] != null && h[k] != x) {
        k++;
        if (k == N) {
            k = 0;
        }
    }
    return k;
}

int main()
{
    int n;
    scanf("%d", &n);
    memset(h, 0x3f, sizeof(h));
    int x;
    int k;
    while (n--) {
        char op[2];
        scanf("%s %d", op, &x);
        k = Find(x);
        if (op[0] == 'I') {
            h[k] = x;
        } else {
            if (h[k] == x) {
                puts("Yes");
            } else {
                puts("No")
            }
        }
    }
    return 0;
}
```

2. 拉链法

此法将在哈希数组的每一个位置 h[t] 处放置一个单链表，来存储所有哈希值为 t 的数字。该单链表可以仅有一个数组表示，每个 h[t] 中存储这一个数组中的不同下标表示单链表的头节点位置。

拉链法

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define N 100003

int h[N];
int e[N];
int ne[N];
int idx;

void Insert(int x)
{
    int t = (x % N + N) % N;
    e[++idx] = x;
    ne[idx] = h[t];
    h[t] = idx;
}

bool Find(int x)
{
    int k = (x % N + N) % N;
    for (int i = h[k]; i != -1; i = ne[i]) {
        if (e[i] == x) {
            return true;
        }
    }
    return false;
}

int main()
{
    int n;
    scanf("%d", &n);
    memset(h, -1, sizeof(h));
    int x;
    int k;
    while (n--) {
        char op[2];
        scanf("%s %d", op, &x);
        if (op[0] == 'I') {
            Insert(x);
        } else {
            if (Find(x)) {
                puts("Yes");
            } else {
                puts("No");
            }
        }
    }
    return 0;
}
```

2.11.2 AcWing 841. 字符串哈希

字符串哈希使用的方法叫做:字符串前缀哈希法。若有一字符串 str = "abcedacbd",则哈希表中的元素表示从左边起到第 i 个位置上的子字符串的哈希值。即,h[1] -> "a"; h[2] -> "ab"; h[3] -> "abc",以此类推。

1. 将字符串存储在下标为 1 的 char 数组中;

2. 计算字符串的前缀数字, 将他们哈希化, 将字符串看作是一个 P 进制数, 这里 P = 131 或者 P = 13331 按照经验,这两个值对字符串哈希会产生非常小的冲突可能。

3. 将这个 P 进制数转换为 10 进制,通常转换出来的数字会很大,则将其对 Q 取模。这样任意一个字符串都可以转换成 $0 \cdots Q-1$ 之间的数字:$(1234)_P => (1 * P^3 + 2 * P^2 + 3 * P^1 + 4 * P^0) \mod Q$

4. 对于 Hash 函数,将计算字符串中从 l 到 r 的哈希值,通过前缀哈希表以及下公式计算,注意左侧为低位,右侧为高位。

   h[r] - h[l - 1] * p[r - l + 1] ,其中 p 数组存放了从 $P^0 - P^N$ 的所有数字,方便查询。

注意:一般来讲不能将字符串映射成 0 ,因为任意进制的 0 都将映射成十进制的 0 。此时有:'A' $\mapsto$ 0;'AA' $\mapsto$ 0,这样不同的字符串映射成了相同的值。另外,这种方法假定了冲突不存在,不需要解决冲突。按照经验一般取 $P = 131 or 13331, Q = 2^{64}$ 时冲突的概率几乎为 0 。

AcWing 841. 字符串哈希

给定一个长度为 n 的字符串,再给定 m 个询问,每个询问包含四个整数 l_1, r_1, l_2, r_2,请你判断 $[l_1, r_1]$ 和 $[l_2, r_2]$ 这两个区间所包含的字符串子串是否完全相同。字符串中只包含大小写英文字母和数字。

输入格式:

第一行包含整数 n 和 m,表示字符串长度和询问次数。第二行包含一个长度为 n 的字符串,字符串中只包含大小写英文字母和数字。接下来 m 行,每行包含四个整数 l_1, r_1, l_2, r_2,表示一次询问所涉及的两个区间。注意,字符串的位置从 1 开始编号。

输出格式:

对于每个询问输出一个结果,如果两个字符串子串完全相同则输出 Yes ,否则输出 No。每个结果占一行。

数据范围:

$1 \le n, m \le 10^5$

输入样例:

```
8 3
aabbaabb
1 3 5 7
1 3 6 8
1 2 1 2
```

输出样例:

```
Yes
No
Yes
```

字符串前缀哈希法

```
#include <stdio.h>
#include <stdbool.h>

#define N 100010
char str[N];
unsigned long long h[N];
unsigned long long p[N];
const int P = 131;

int Hash(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}

int main()
{
    int n;
    int m;
    scanf("%d %d", &n, &m);
    scanf("%s", str + 1);

    p[0] = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = p[i - 1] * P;
        h[i] = str[i] + h[i - 1] * P;
    }

    while (m--) {
        int l1, r1, l2, r2;
        scanf("%d %d", &l1, &r1);
        scanf("%d %d", &l2, &r2);
        if (Hash(l1, r1) == Hash(l2, r2)) {
            puts("Yes");
        } else {
            puts("No");
        }
    }
    return 0;
}
```

注意：此处将字符串存储在下标为 1 的地方开始。是因为这里将字符串看作是 P 进制数，如果从 0 开始会将同一个字符构成的字符串都算做相同的值，会有问题。

另外，这里没有显式的对 Q = 2^64 取模，是因为用了 unsinged long long 溢出后就相当于对 Q 取模。

第三章

搜索和图论

	数据结构	空间	
DFS	stack	O(h)	不具有最短性
BFS	queue	O(2^h)	第一次扩展到最近点,最短路性质(边的权重为 1 时)

稠密图:邻接矩阵 g[N][N], a 和 b 之间是否有边,且边的权重是多少

稀疏图:邻接表 h[N], e[N], ne[N], idx, 和拉链法的哈希是一样的数据结构

3.1 DFS

从搜索树的角度来看。

回溯:

剪枝:

3.1.1 AcWing 842. 排列数字

AcWing 842. 排列数字

给定一个整数 n，将数字 $1 \sim n$ 排成一排，将会有很多种排列方法。现在，请你按照字典序将所有的排列方法输出。

输入格式：

共一行，包含一个整数 n。

输出格式：

按字典序输出所有排列方案，每个方案占一行。

数据范围：

$1 \leq n \leq 7$

输入样例：
3

输出样例：
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

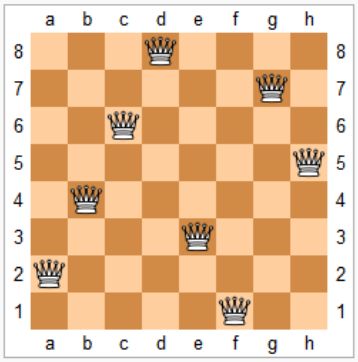
排列数字

```c
#include <stdio.h>
#include <stdbool.h>

#define N 10

int path[N];
bool st[N];

void dfs(int u, int n)
{
    if (u == n) {
        for (int i = 0; i < n; i++) {
            printf("%d ", path[i]);
        }
        printf("\n");
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (!st[i]) {
            path[u] = i;
            st[i] = true;
            dfs(u + 1, n);
            st[i] = false;
        }
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    dfs(0, n);
    return 0;
}
```

3.1.2 AcWing 843. n-皇后问题

AcWing 843. n-皇后问题

$n-$ 皇后问题是指将 n 个皇后放在 $n \times n$ 的国际象棋棋盘上，使得皇后不能相互攻击到，即任意两个皇后都不能处于同一行、同一列或同一斜线上。



One solution to the eight queens puzzle

现在给定整数 n，请你输出所有的满足条件的棋子摆法。

输入格式：

共一行，包含整数 n。

输出格式：

每个解决方案占 n 行，每行输出一个长度为 n 的字符串，用来表示完整的棋盘状态。其中 . 表示某一个位置的方格状态为空，Q 表示某一个位置的方格上摆着皇后。每个方案输出完成后，输出一个空行。

注意：行末不能有多余空格。输出方案的顺序任意，只要不重复且没有遗漏即可。

数据范围：

$1 \leq n \leq 9$

输入样例：

```
4
```

输出样例：

```
.Q..
...Q
Q...
..Q.

..Q.
Q...
...Q
.Q..
```

有两种解法，第一种：因发现每一行每一列有且仅有一个皇后，可以枚举每一行，来看看可以将皇后放置在第几列。

n-皇后问题：解法一

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define N 20

char g[N][N];
bool col[N];
bool dg[N];
bool udg[N];

void dfs(int u, int n)
{
    if (u == n) {
        for (int i = 0; i < n; i++) {
            puts(g[i]);
        }
        printf("\n");
        return;
    }
    for (int i = 0; i < n; i++) {
        if (col[i] || dg[u + i] || udg[u - i + n]) {
            continue;
        }
        g[u][i] = 'Q';
        col[i] = dg[u + i] = udg[u - i + n] = true;
        dfs(u + 1, n);
        g[u][i] = '.';
        col[i] = dg[u + i] = udg[u - i + n] = false;
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            g[i][j] = '.';
        }
    }
    dfs(0,  n);
    return 0;
}
```

解法二：从左上角开始枚举每一个格子，每个格子有两种状态（放或者不放），在枚举过程中检查每一个格子是否可以继续放置。若皇后数量已经达到最大数量则找到了一个方案

n-皇后问题：解法二

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define N 20

char g[N][N];
bool row[N];
bool col[N];
bool dg[N];
bool udg[N];

void dfs(int x, int y, int s, int n)
{
    if (y == n) {
        y = 0;
        x++;
    }
    if (x == n) {
        if (s == n) {
            for (int i = 0; i < n; i++) {
                puts(g[i]);
            }
            puts("");
        }
        return;
    }
    // do not put queen here
    dfs(x, y + 1, s, n);

    // put queen here
    if (!row[x] && !col[y] && !dg[x + y] && !udg[x - y + n]) {
        g[x][y] = 'Q';
        row[x] = col[y] = dg[x + y] = udg[x - y + n] = true;
        dfs(x, y + 1, s + 1, n);
        g[x][y] = '.';
        row[x] = col[y] = dg[x + y] = udg[x - y + n] = false;
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            g[i][j] = '.';
        }
    }
    dfs(0, 0, 0, n);
    return 0;
}
```

3.2 BFS

3.2.1 AcWing 844. 走迷宫

AcWing 844. 走迷宫

给定一个 $n \times m$ 的二维整数数组，用来表示一个迷宫，数组中只包含 0 或 1，其中 0 表示可以走的路，1 表示不可通过的墙壁。最初，有一个人位于左上角 $(1,1)$ 处，已知该人每次可以向上、下、左、右任意一个方向移动一个位置。请问，该人从左上角移动至右下角 (n,m) 处，至少需要移动多少次。数据保证 $(1,1)$ 处和 (n,m) 处的数字为 0，且一定至少存在一条通路。

输入格式：

第一行包含两个整数 n 和 m。接下来 n 行，每行包含 m 个整数（0 或 1），表示完整的二维数组迷宫。

输出格式：

输出一个整数，表示从左上角移动至右下角的最少移动次数。

数据范围：

$1 \leq n, m \leq 100$

输入样例：

```
5 5
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

输出样例：

```
8
```

3.2.2 AcWing 845. 八数码

AcWing 845. 八数码

在一个 3×3 的网格中，$1 \sim 8$ 这 8 个数字和一个 x 恰好不重不漏地分布在这 3×3 的网格中。

例如：

1 2 3

x 4 6

7 5 8

在游戏过程中，可以把 x 与其上、下、左、右四个方向之一的数字交换（如果存在）。我们的目的是通过交换，使得网格变为如下排列（称为正确排列）：

1 2 3

4 5 6

7 8 x

例如，示例中图形就可以通过让 x 先后与右、下、右三个方向的数字交换成功得到正确排列。交换过程如下：

1 2 3　1 2 3　1 2 3　1 2 3

x 4 6　4 x 6　4 5 6　4 5 6

7 5 8　7 5 8　7 x 8　7 8 x

现在，给你一个初始网格，请你求出得到正确排列至少需要进行多少次交换。

输入格式：

输入占一行，将 3×3 的初始网格描绘出来。例如，如果初始网格如下所示：

1 2 3

x 4 6

7 5 8

则输入为：1 2 3 x 4 6 7 5 8

输出格式：

输出占一行，包含一个整数，表示最少交换次数。如果不存在解决方案，则输出 -1。

输入样例：	输出样例：
2 3 4 1 5 x 7 6 8	19

3.3 树与图的深度优先遍历

3.3.1 AcWing 846. 树的重心

AcWing 846. 树的重心

给定一棵树，树中包含 n 个结点（编号 $1 \sim n$）和 $n-1$ 条无向边。请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。

重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。

输入格式：

第一行包含整数 n，表示树的结点数。接下来 $n-1$ 行，每行包含两个整数 a 和 b，表示点 a 和点 b 之间存在一条边。

输出格式：

输出一个整数 m，表示将重心删除后，剩余各个连通块中点数的最大值。

数据范围：

$1 \leq n \leq 10^5$

输入样例：

```
9
1 2
1 7
1 4
2 8
2 5
4 3
3 9
4 6
```

输出样例：

```
4
```

3.4 树与图的广度优先遍历

3.4.1 AcWing 847. 图中点的层次

AcWing 847. 图中点的层次

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环。所有边的长度都是 1，点的编号为 $1 \sim n$。请你求出 1 号点到 n 号点的最短距离，如果从 1 号点无法走到 n 号点，输出 -1。

输入格式：

第一行包含两个整数 n 和 m。接下来 m 行，每行包含两个整数 a 和 b，表示存在一条从 a 走到 b 的长度为 1 的边。

输出格式：

输出一个整数，表示 1 号点到 n 号点的最短距离。

数据范围：

$1 \leq n, m \leq 10^5$

输入样例：

```
4 5
1 2
2 3
3 4
1 3
1 4
```

输出样例：

```
1
```

3.5 拓扑排序

3.5.1 AcWing 848. 有向图的拓扑序列

AcWing 848. 有向图的拓扑序列

给定一个 n 个点 m 条边的有向图，点的编号是 1 到 n，图中可能存在重边和自环。请输出任意一个该有向图的拓扑序列，如果拓扑序列不存在，则输出 -1。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 是该图的一个拓扑序列。

输入格式：

第一行包含两个整数 n 和 m。接下来 m 行，每行包含两个整数 x 和 y，表示存在一条从点 x 到点 y 的有向边 (x, y)。

输出格式：

共一行，如果存在拓扑序列，则输出任意一个合法的拓扑序列即可。否则输出 -1。

数据范围：

$1 \leq n, m \leq 10^5$

输入样例：

```
3 3
1 2
2 3
1 3
```

输出样例：

```
1 2 3
```

3.6 有向图的最短路问题

这一节将关注在图的最短路的求解上，最短路问题及其常见方法如下：

最短路问题：n 表示点数，m 表示边数

1. 单源最短路：求一个点到其他所有点的最短路
   - 所有边权都是正数：1. 朴素 dijkstra 算法 $\mathcal{O}(n^2)$，与边数无关，适合稠密图 2. 堆优化版的 dijkstra 算法 $\mathcal{O}(m \log n)$ 稀疏图：m 与 n 量级相同
   - 存在负权边：1. Bellman-Ford 算法 $\mathcal{O}(nm)$ 2. SPFA $\mathcal{O}(m)$ 最坏 $\mathcal{O}(nm)$
2. 多源汇最短路：起点和终点不确定，问 a 到 b 的最短路
   - Floyd 算法 $\mathcal{O}(n^3)$

3.6.1 AcWing 849. Dijkstra 求最短路 I

朴素版的 Dijkstra 算法：

1. 用邻接矩阵存储图；g[N][N]

2. 用一个数组存储每一个点到起始点的距离d[N]；

3. st[N]，存储每一个点是否已经找到了最短路

算法逻辑：

1. 将距离数组初始化为d[i] $= +\infty$; d[k] $= 0$; k 表示起始点

2. 从第 0 个点到第 $n-1$ 个点，找到距离当前点最近的下一个点 (遍历 $1-n$) t; st[t] = true;

3. 用 t 更新其他点的距离 (遍历 $1-n$)

AcWing 849. Dijkstra 求最短路 I

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为正值。请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出 -1。

输入格式：

第一行包含整数 n 和 m。接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式：

输出一个整数，表示 1 号点到 n 号点的最短距离。如果路径不存在，则输出 -1。

数据范围：

$1 \le n \le 500$, $1 \le m \le 10^5$，图中涉及边长均不超过 10000。

输入样例：

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例：

```
3
```

3.6.2 AcWing 850. Dijkstra 求最短路 II

注意到在朴素版的 Dijkstra 算法中需要找到当前节点之后的所有节点中距离最小的那个点的复杂度是 $\mathcal{O}(n)$，而堆中寻找最小值的复杂度则为 $\mathcal{O}(1)$。所以可以用堆来优化这个过程。后面用到了修改堆中元素，为了简单起见，直接插入即可。

如果稠密图中 n 的量为 10^5 时 $\mathcal{O}(n^2)$ 将会超时，所以选择使用堆对其进行优化，将其降低到 $\mathcal{O}(m \log n)$

AcWing 850. Dijkstra 求最短路 II

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为正值。请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出 -1。

输入格式：

第一行包含整数 n 和 m。接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式：

输出一个整数，表示 1 号点到 n 号点的最短距离。如果路径不存在，则输出 -1。

数据范围：

$1 \leq n, m \leq 1.5 \times 10^5$，图中涉及边长均不小于 0，且不超过 10000。

输入样例：

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例：

```
3
```

3.6.3 bellman-ford

处理有负权边的图，如果有负权回路的话最短路不一定存在。所以一般不能有负权回路。

算法逻辑：

1. 循环 n 个点

2. 循环所有边 a, b, w: dist [b] = min(dist[b], dist [a] + w) 更新所有距离

因为算法特点存储图时不必使用邻接矩阵或者邻接表，开一个结构体数组：

边集

```
struct {
    int a;
    int b;
    int w;
} Edges[N]
```

只要能够遍历到所有边即可。可以证明，该算法完成后一定有三**角不等式**：dist [b] <= dist[a] + w成立更新距离数组的过程叫做**松弛操作**。

第一个循环中，如果当前迭代了 k 次，此时的 dist 数组表示的是从 1 号点经过不超过 k 条边走到每个点的距离，所以可以用这个原理来找负环，迭代到第 n 次仍能更新则表示有 $n+1$ 个点，但实际只有 n 个点，所以一定存在负环。但一般不用该算法找负环。

3.6.4 AcWing 853. 有边数限制的最短路

3.6.5 spfa

spfa 算法是对 bellman-ford 算法的优化，在松弛操作中，不一定每一次都会对该点的距离减小有所贡献，只有与之相连的前驱节点距离减小了，此时才有可能有所贡献。

用一个 bfs 来做优化，队列中存储所有变小了距离的节点

3.6.6 AcWing 851. spfa 求最短路

3.6.7 AcWing 852. spfa 判断负环

3.6.8 Floyd

求多源汇最短路，用邻接矩阵来存储 d[i, j]

Floyd 算法

```
01 for (int k = 1; k <= n; ++k) {
02     for (int i = 1; i <= n; ++i) {
03         for (int j = 1; j <= n; ++j) {
04             d[i, j] = min(d[i, j], d[i, k] + d[k, j]);
05         }
06     }
07 }
```

初始时，d[i, j] 就是邻接矩阵，结束之后 d[i, j] 是最短路长度

3.6.9 AcWing 854. Floyd 求最短路

3.7 无向图的最小生成树问题

1. 普里姆算法（Prim 算法）1. 朴素版 Prim（稠密图）$\mathcal{O}(n^2)$ 2. 堆优化版 Prim（稀疏图）$\mathcal{O}(m \log n)$，一般用不到

2. 克鲁斯卡尔算法（Kruskal 算法）$\mathcal{O}(m \log m)$ 稀疏图

3.7.1 AcWing 858. Prim 算法求最小生成树

Prim 算法，与 dijkstra 算法类似：

Prim 算法：

1. 用邻接矩阵存储图；g[N][N]

2. 用一个数组存储每一个点到集合的距离 d[N], 即存储该点所连接边的最小权重；

3. 集合 st[N]，存储当前已经在连通块中的点

算法逻辑：

1. 将距离数组初始化为 $\mathrm{d[i]} = +\infty$; $\mathrm{d[0]} = 1$;
2. 从第 0 个点到第 $n-1$ 个点，找到集合外距离最近的点 t; $\mathrm{st[t]} = \mathrm{true}$;
3. 用 t 更新其他点到集合的距离

3.7.2 AcWing 859. Kruskal 算法求最小生成树

1. 将所有边按照权重从小到大排序；$\mathcal{O}(m \log m)$
2. 枚举每条边 a, b, 权重 c，如果 a 和 b 不连通，就把这条边加入到集合中。并查集

3.8 二分图

定义：一个图的所有节点可以划分到两个集合中使得图中的边都只存在于集合之间的图称其为可二分的图。

性质：一个图是二分图，当且仅当一个图可以被二染色，当且仅当图中没有奇数环（环中边个数为奇数）

1. 染色法 $\mathcal{O}(n+m)$
2. 匈牙利算法最坏 $\mathcal{O}(mn)$, 实际运行时间一般远小于 $\mathcal{O}(mn)$

3.8.1 AcWing 860. 染色法判定二分图

3.8.2 匈牙利算法

3.8.3 AcWing 861. 二分图的最大匹配

第四章

数学知识

第五章

动态规划

第六章

贪心算法

PART II

第二部分

算法提高

PART III

第三部分

算法进阶

PART IV

第四部分

LeetCode 究极班

第一章

C 语言常用函数及技巧

A.1 输入输出

int scanf(const char *format, ...) 当需要读入一个字符时需要加一个空格scanf(" %c", &op);,要有个空格用来跳过回车,不然会出现读入数量不够的情况。