算法模板-lbs

1. 基础算法

1.1 排序

1.1.1 快速排序-递归

> **思想：**
>
> **（1）确定分界点：**左边界值，中间值，有边界值，随机值均可
>
> **（2）调整范围：**将当前区间[L, R]划分为[L, M], [M + 1, R]，左右两边分别满足（1）中分界点的一些性质即可
>
> **（3）递归处理左右两边：**针对区间[L， M]和[M + 1， R]继续（2）中的操作，知道区间长度为0即完成排序

效率：其中`n`为数组的长度

- **时间复杂度：**(平均复杂度)

$$O(n \log n) \tag{1}$$

- **空间复杂度：**

$$O(1) \tag{2}$$

> **实现：**C++

```cpp
// 快排模板

#include<iostream>

using namespace std;

const int N = 1000010;

int n;
int q[N];

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

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

```

> **实现：Java**

```java
package basic.quicksort;
import java.util.Arrays;
import java.util.Scanner;
```

```java

/**
 * @author LBS59
 * @description 快速排序模板
 */
public class Main {
    static int n;
    static int[] arr;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        quickSort(0, n - 1);
        System.out.println(Arrays.toString(arr));
    }

    private static void quickSort(int l, int r) {
        if (l >= r) {
            return;
        }
        int x = arr[l], i = l - 1, j = r + 1;
        while (i < j) {
            do {
                i++;
            } while (arr[i] < x);
            do {
                j--;
            } while (arr[j] > x);
            if (i < j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        quickSort(l, j);
        quickSort(j + 1, r);
    }
}
```

实现：Python

```python
N = int(1e6 + 10)
q = [0] * N


def quick_sort(arr, left, right):
    if left >= right:
        return
    low = left
    high = right
    key = arr[low]
    while left < right:
        while left < right and arr[right] > key:
            right -= 1
        arr[left] = arr[right]
        while left < right and arr[left] <= key:
            left += 1
        arr[right] = arr[left]
    arr[right] = key
    quick_sort(arr, low, left - 1)
    quick_sort(arr, left + 1, high)


if __name__ == '__main__':
    n = int(input())
    strs = input().split()
    for i in range(n):
        q[i] = int(strs[i])
    quick_sort(q, 0, n - 1)
    for i in range(n):
        print(q[i], end=" ")
```

1.1.2 归并排序-分治

思想：

（1）确定分治分解点：对于左右边界[L， R]，取mid = (l + r) / 2

（2）先递归排序左右两边：[L， mid]， [mid + 1， R]，将整个数组划分为长度为1的子区间，一个元素天然有序

（3）归并：将所有子区间两两合并，最终合成有序的整个数组

效率：其中`n`为数组的长度

- **时间复杂度：**

$$O(n \log n) \tag{3}$$

- **空间复杂度：**

$$O(n) \tag{4}$$

> **实现：C++**

```cpp
#include<iostream>

using namespace std;

const int N = 1000010;

int n;
int q[N], tmp[N];

void merge_sort(int q[], int l, int r)
{
    if (l >= r)
    {
        return;
    }
    int mid = l + r >> 1;
    merge_sort(q, l, mid), merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
    {
        if (q[i] <= q[j])
        {
            tmp[k++] = q[i++];
        } else
        {
            tmp[k++] = q[j++];
        }
    }
    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];
    }
}

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

> **实现：Java**

```java
package basic.mergesort;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 归并排序模板
 */
public class Main {
    static int n;
    static int[] nums, tmp;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        nums = new int[n];
        tmp = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
        mergeSort(0, n - 1);
        for (int i = 0; i < n; i++) {
            System.out.printf("%d", nums[i]);
            if (i != n - 1) {
                System.out.print(" ");
            }
        }
    }

    private static void mergeSort(int l, int r) {
        if (l >= r) {
            return;
        }
        int mid = (l + r) >> 1;
        mergeSort(l, mid);
        mergeSort(mid + 1, r);

        int k = 0, i = l, j = mid + 1;
        while (i <= mid && j <= r) {
            if (nums[i] <= nums[j]) {
                tmp[k++] = nums[i++];
            } else {
                tmp[k++] = nums[j++];
            }
        }
        while (i <= mid) {
            tmp[k++] = nums[i++];
        }
        while (j <= r) {
            tmp[k++] = nums[j++];
        }
        for (int q = l, w = 0; q <= r; q++, w++) {
            nums[q] = tmp[w];
        }
    }
}

```

实现：Python

```
N = int(1e6 + 10)
q, tmp = [0] * N, [0] * N


def merge_sort(arr, left, right):
    if left >= right:
        return
    mid = left + right >> 1
    merge_sort(q, left, mid)
    merge_sort(q, mid + 1, right)
    k, i, j = 0, left, mid + 1
    while i <= mid and j <= right:
        if q[i] <= q[j]:
            tmp[k] = q[i]
            k += 1
            i += 1
        else:
            tmp[k] = q[j]
            k += 1
            j += 1
    while i <= mid:
        tmp[k] = q[i]
        k += 1
        i += 1
    while j <= right:
        tmp[k] = q[j]
        k += 1
        j += 1
    w = 0
    for e in range(left, right + 1):
        q[e] = tmp[w]
        w += 1


if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(n):
        q[i] = int(arr[i])
    merge_sort(q, 0, n - 1)
    for i in range(n):
        print(q[i], end=" ")
```

1.2 二分模板-两套模板

> **思想：有序一定可以二分，二分不一定必须有序**
>
> **（1）划分数组：**针对某一个数组[0, n]，针对求解出一个边界将子数组划分为两个子区间[0, mid]/[0, mid + 1]和[mid, n]/[mid + 1, n]
>
> **（2）条件判断：**针对某一个子区间如[mid, n]满足一种规律
>
> **（3）循环处理：**针对满足条件的子区间继续（1）和（2）的操作，直到锁定一个目标值，判定目标值是否为所需要的目标值

效率：其中 `n` 为数组的长度

- **时间复杂度：**(平均复杂度)

$$O(\log n) \tag{5}$$

- **空间复杂度：**

$$O(1)/O(n)\text{不等，根据题目要求} \tag{6}$$

> **实现：C++**

```cpp
#include<iostream>

using namespace std;

const int N = 100010

int n;
int q[N];

bool check(int x)
{
    // 判定x满足某种性质
    /*
    ......
    */
}

int bin_search1(int l, int r)
{
    while (l < r)
    {
        // 两种模板就在mid的计算有点差异，避免边界问题产生死循环
        int mid = l + r >> 1;
        if (check(mid))
        {
            r = mid;
        }
        else
        {
            l = mid + 1;
        }
    }
    return l;
}

int bin_search2(int l, int r)
{
    while (i < r)
    {
        // 两种模板就在mid的计算有点差异，避免边界问题产生死循环
        int mid = l + r + 1 >> 1;
        if (check(mid))
        {
            l = mid;
        }
        else
        {
            r = mid - 1;
        }
    }
    return l;
}


int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d",&q[i]);
    }
    int res1 = bin_search1(0, n - 1);
    int res2 = bin_search2(0, n - 1);
    printf("%d", res1);
    printf("%d", res2);
}
```

实现：Java

```java
import java.io.*;

/**
 * @author LBS59
 */
public class Main {
    private static final int N = 100010;
    static int[] arr = new int[N];

    private static boolean check(int x, int target) {
        /*
        * check方法写x满足的某种规律
        * */
        return false;
    }

    private static int bSearch1(int l, int r, int target) {
        while (l < r) {
            int mid = l + r >> 1;
            if (check(arr[mid], target)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l;
    }

    private static int bSearch2(int l, int r, int target) {
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check(arr[mid], target)) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        return l;
    }

    public static void main(String[] args) throws IOException {
        // 输入输出模板
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s1 = in.readLine().split(" ");
        int n = Integer.parseInt(s1[0]), target = Integer.parseInt(s1[1]);
        String[] input = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            arr[i] = Integer.parseInt(input[i]);
        }
        System.out.println(bSearch1(0, n - 1, target));
        System.out.println(bSearch2(0, n - 1, target));

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python

```python
N = 100010
q = [0] * N

def check(x):
    # 这里判断x满足某种规律
    pass


def b_search1(l, r):
    while l < r:
        mid = l + r >> 1
        if check(mid) > 0:
            r = mid
        else:
            l = mid + 1


def b_search2(l, r):
    while l < r:
        mid = l + r + 1 >> 1
        if check(mid) > 0:
            l = mid
```

```
        else:
            r = mid - 1


if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(n):
        q[i] = int(arr[i])
    print(b_search1(0, n - 1))
    print(b_search2(0, n - 1))
```

1.3 前缀和数组 -- 一维+二维

> **应用场景：** 前缀和数组应用场景十分单一，其思想是将一个长度为n的数组[0, n]扩充一位，将之前的n个数组放置在新数组的下标[1,n]位置，之后采取的策略是从下标为1开始，每个位置的值更新为其当前值与前一个下标位置值之和，在二维数组中也是如此公式如下：

$$arr[i] = arr[i] + arr[i-1] \quad i \in [1, n] \tag{7}$$

$$arr[i][j] = arr[i][j] + arr[i-1][j] + arr[i][j-1] - arr[i-1][j-1] \quad i \in [1, n], j \in [1, m] \tag{8}$$

> **使用：** 当生成前缀和数组之后，如何我们想要计算[L, R]这个区间的元素和，无需使用for循环遍历，二维数组也是类似的处理。直接使用如下公式即可计算

$$\sum_{i=L}^{R} arr[i] = preSum[R] - preSum[L-1] \quad \text{这里} preSum \text{表示生成的前缀和数组} \tag{9}$$

$$\sum_{i=x1}^{x2}\sum_{j=y1}^{y2} preSum[i][j] = preSum[x2][y2] - preSum[x1-1][y2] - preSum[x2][y1-1] + preSum[x1-1][y1-1] \quad \text{这里} preSum \text{为生成的}$$

1.3.1 一维前缀和

> **实现：C++：**

```cpp
// 一维前缀和数组
#include<iostream>

using namespace std;

const int N = 100010;
int n;
int q[N];

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &q[i]);
        q[i] += q[i - 1];
    }
    for (int i = 1; i <= n; i++)
    {
        printf("%d ", q[i]);
    }
}
```

> **实现：Java：**

```java
package basic.presum;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 一维前缀和模板
 */
public class OneDPreSum {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] preSum = new int[n + 1];
        for (int i = 1; i < n + 1; i++) {
            preSum[i] = preSum[i - 1] + sc.nextInt();
        }
        for (int i = 1; i <= n; i++) {
            System.out.printf("%d ", preSum[i]);
        }
    }
}
```

> **实现：Python：**

```python
# 一维前缀和
N = 100010
q = [0] * N

if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(1, n + 1):
        q[i] = int(arr[i - 1])
        q[i] += q[i - 1]
    for i in range(1, n + 1):
        print(q[i], end=" ")
```

1.3.2 二维前缀和

实现：C++：

```cpp

// 二维前缀和数组
#include<iostream>

using namespace std;
const int N = 1010;

int n, m;
int q[N][N];

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            scanf("%d", &q[i][j]);
            q[i][j] += q[i - 1][j] + q[i][j - 1] - q[i - 1][j - 1];
        }
    }
    puts("");
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            printf("%d ", q[i][j]);
        }
        puts("");
    }
    return 0;
}
```

实现：Java：

```java
package basic.presum;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 二维前缀和模板
 */
public class TwoDPreSum {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        int[][] preSum = new int[n + 1][m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                preSum[i][j] += preSum[i - 1][j] + preSum[i][j - 1] - preSum[i - 1][j - 1] +
sc.nextInt();
            }
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                System.out.printf("%d ", preSum[i][j]);
            }
            System.out.println();
        }
    }
}
```

实现：Python：

```python
# 二维前缀和
N = 1010
```

```python
q = [[0 for _ in range(N)] for j in range(N)]

if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        arr = input().split()
        for j in range(1, m + 1):
            q[i][j] = int(arr[j - 1])
            q[i][j] += q[i - 1][j] + q[i][j - 1] - q[i - 1][j - 1]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            print(q[i][j], end=" ")
        print()
```

1.4 差分数组---一维+二维

> **应用场景：**和前缀和数组一样，差分数组的概念是使用长度为n的原数组生成一个数组，该数组的特征是：对生成的差分数组求前缀和数组可以还原原数组，构造差分数组有一种固定的方法，先假定数组所有元素为0，则差分数组也全为0，然后使用一种插值的方式来构建出差分数组的全部元素。假设要在下标为i的位置插入元素x，那么在还原数组时在x之后的所有下标位置都会有个增量x，这个不符合要求，所以只要在下标i+1的位置减去元素x，则就可以抵消掉x的增量，就能正确还原原数组。

> **使用：**差分数组限于这种要求，使用到的地方也十分单一，具体应用只有一个地方，就是针对一个数组arr，下标为[0, n]，题目需要频繁在数组中一个子数组增加或减少一个值x，如果使用以往的朴素做法就是两层循环处理，如果数量级很大就会超时，但是使用差分数组可以在`o(1)`时间复杂度内完成修改子数组操作，下面给出插值方式：

$$\text{如果在下标}i\text{出插入一个值}x\text{，则差分数组的操作为 } divArr[i]+=x;divArr[i+1]-=x \tag{11}$$

1.4.1 一维差分数组

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int n;
// divArr数组存储的差分数组
int q[N], divArr[N];

void insert_val(int l, int r, int c)
{
    divArr[l] += c;
    divArr[r + 1] -= c;
}

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &q[i]);
        insert_val(i, i, q[i]);
    }
    puts("");
    printf("差分数组：");
    for (int i = 1; i <= n; i++)
    {
        printf("%d ", divArr[i]);
    }
    puts("");
    puts("");
    printf("差分数组还原原数组：");
    for (int i = 1; i <= n; i++)
    {
        divArr[i] += divArr[i - 1];
        printf("%d ", divArr[i]);
    }
    return 0;
}
```

> **实现：Java：**

```java
package basic.preDiv;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 一维差分计算 - 注意这里的数组长度需要前后均扩增一维
 */
public class OneDPreDiv {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] q = new int[n + 2];
        int[] divArr = new int[n + 2];
        for (int i = 1; i <= n; i++) {
            q[i] = sc.nextInt();
            insert(divArr, i, i, q[i]);
        }
        System.out.println();
        System.out.print("差分数组:");
        for (int i = 1; i <= n; i++) {
            System.out.printf("%d ", divArr[i]);
        }
        System.out.println();
        System.out.println("差分数组还原: ");
        for (int i = 1; i <= n; i++) {
            divArr[i] += divArr[i - 1];
            System.out.printf("%d ", divArr[i]);
        }
    }

    private static void insert(int[] temp, int l, int r, int c) {
        temp[l] += c;
        temp[r + 1] -= c;
    }
}
```

实现：Python：

```
N = 100010
q, div_arr = [0] * N, [0] * N


def insert_val(l, r, c):
    div_arr[l] += c
    div_arr[r + 1] -= c


if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(1, n + 1):
        q[i] = int(arr[i - 1])
        insert_val(i, i, q[i])
    print()
    print("差分数组: ", end=" ")
    for i in range(1, n + 1):
        print(div_arr[i], end=" ")
    print()
    print("差分数组还原: ", end=" ")
    for i in range(1, n + 1):
        div_arr[i] += div_arr[i - 1]
        print(div_arr[i], end=" ")
```

1.4.2 二维差分数组

实现：C++:

```
#include<iostream>

using namespace std;

const int N = 1010;

int n, m;
int q[N][N], tmp[N][N];

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

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            scanf("%d", &q[i][j]);
            insert_val(i, j, i, j, q[i][j]);
        }
```

```
    }
    puts("");
    printf("差分数组：\n");
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            printf("%d ", tmp[i][j]);
        }
        puts("");
    }
    puts("");
    printf("差分数组还原(二维前缀和计算公式)：\n");
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            tmp[i][j] += tmp[i - 1][j] + tmp[i][j - 1] - tmp[i - 1][j - 1];
            printf("%d ", tmp[i][j]);
        }
        puts("");
    }
    return 0;
}
```

实现：Java：

```
package basic.preDiv;

import java.io.*;

/**
 * @author LBS59
 * @description 二维差分数组模板
 */
public class TwoDPreDiv {
    private static final int N = 1010;
    static int[][] temp = new int[N][N];

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        int[][] q = new int[n + 2][m + 2];
        for (int i = 1; i <= n; i++) {
            String[] input = in.readLine().split(" ");
            for (int j = 1; j <= m; j++) {
                q[i][j] = Integer.parseInt(input[j - 1]);
                insert(i, j, i, j, q[i][j]);
            }
        }
        System.out.println();
        System.out.println("差分数组");
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                System.out.printf("%d ", temp[i][j]);
            }
            System.out.println();
        }
        System.out.println();
        System.out.println("差分数组还原");
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                temp[i][j] += temp[i - 1][j] + temp[i][j - 1] - temp[i - 1][j - 1];
                System.out.printf("%d ", temp[i][j]);
            }
            System.out.println();
        }

        out.flush();
        in.close();
        out.close();
    }

    private static void insert(int x1, int y1, int x2, int y2, int c) {
        temp[x1][y1] += c;
        temp[x1][y2 + 1] -= c;
        temp[x2 + 1][y1] -= c;
        temp[x2 + 1][y2 + 1] += c;
    }
}
```

实现：Python： \

```python
N = 1010
q, tmp = [[0 for _ in range(N)] for _ in range(N)], [[0 for _ in range(N)] for _ in range(N)]


def insert_val(x1, y1, x2, y2, c):
    tmp[x1][y1] += c
    tmp[x1][y2 + 1] -= c
    tmp[x2 + 1][y1] -= c
    tmp[x2 + 1][y2 + 1] += c


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        arr = input().split()
        for j in range(1, m + 1):
            q[i][j] = int(arr[j - 1])
            insert_val(i, j, i, j, q[i][j])
    print()
    print("差分数组: ")
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            print(tmp[i][j], end=" ")
        print()
    print()
    print("差分数组还原(二维前缀和公式): ")
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            tmp[i][j] += tmp[i - 1][j] + tmp[i][j - 1] - tmp[i - 1][j - 1]
            print(tmp[i][j], end=" ")
        print()
```

1.5 双指针算法

> **使用思想：** 双指针的思想是将一些时间复杂度的问题，通过双指针算法优化到`o(n)`的复杂度，双指针算法有一般性的模板，但在具体细节处理方面需要字节写逻辑操作，下面给出一般性的模板(伪代码):

> **时间复杂度：**

$$\text{朴素做法}O(n^2) ==> \text{双指针做法}O(2n) <=> O(n) \tag{12}$$

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int n;
int q[N];

bool check(int l, in r)
{
    // 这里写l, r下标满足的某种关系
    return true;
}

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &q[i]);
    }
    for (int i = 0, j = 0; i < n; i++)
    {
        while (j < i && check(i, j)) {
            j++;
        }
        // 题目中处理的逻辑操作
    }
}
```

> **实现：Java：**

```java
import java.util.Scanner;

/**
 * @author LBS59
 */
public class Main {
    private static final int N = 100010;
    static int[] arr = new int[N];
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        for (int i = 0, j = 0; i < n; i++) {
            while (j < i && check(i, j)) {
                j++;
            }
            // 题目逻辑操作
        }
    }

    private static boolean check(int l, int r) {
        // 这里写下标l和r满足的某种性质
        return false;
    }
}
```

实现：Python：

```
N = 100010
q = [0] * N


def check(l, r):
    # 写下标l， r满足的某种性质
    return True


if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(n):
        q[i] = int(arr[i])

    j = 0
    for i in range(n):
        while j < i and check(i, j):
            j += 1
        # 写后续的逻辑操作
```

例子：一个由单词和空格组成的字符串，在读入字符串后，将所有单词提取出来并打印，每一个单词占一行

```
#include<iostream>
#include<string.h>

using namespace std;

int main()
{
    char str[1000];
    gets(str);

    int n = strlen(str);

    for (int i = 0; i < n; i++)
    {
        int j = i;
        while (j < n && str[j] != ' ')
        {
            j++;
        }
        // 题目逻辑是打印每一个单词
        for (int k = i; k < j; k++)
        {
            cout << str[k];
        }
        cout << endl;
        i = j;
    }
    return 0;
}
```

1.6 位运算

1.6.1 求n的二进制表示中第k位的数字(n是一个整数)

思路：因为数字在计算机底层存储就是二进制位的存储方式，因此无需转换位二进制表示，求第k位二进制表示是1还是0，就是先将n右移k位，将第k位置于二进制表示的个位，然后和1做与运算即可。

$$s_k = n >> k \ \& \ 1 \quad \text{其中} s_k \text{表示} n \text{的二进制表示中第} k \text{位的值} \tag{13}$$

1.6.2 lowbit(n)，求n的最后一位1的位置

> **思路：** 举个例子--假定正整数88的二进制表示位`01011000`，最高位0表示其为正数，其相反数为-88，在计算机底层，负数的存储方式为其补码，-88的二进制表示原码为`11011000`，反码为除符号为其余为取反-`10100111`，其补码为反码+1-`10101000`，观察规律，可以发现，88和-88的二进制表示处理最低位1的位置，高于最低位1的位置均为相反数，低于最低位1的位置均相等且为0，则可以很容易想到`lowbit(n)`的公式：

$$lowbit(n) = n\ \&\ -n = n\ \&\ (\sim n + 1)\ \text{其中}\sim n\text{为}n\text{的反码的二进制表示} \tag{14}$$

> **实例：** 求出数列中每个数的二进制表示中 11 的个数。

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

int n;

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

int main()
{
    scanf("%d", &n);
    while (n --)
    {
        int x;
        scanf("%d", &x);
        int cnt = 0;
        while (x)
        {
            x -= lowbit(x);
            cnt++;
        }
        printf("%d ", cnt);
    }
}
```

> **实现：Java：**

```java
import java.io.*;

/**
 * @author LBS59
 * @description 二进制模板使用
 */
public class Main {
    /**
     * 返回整数二进制表示中最低位的1的结果
     * @param x 整数x
     * @return 表达形式为最低位的二进制结果
     */
    public static int lowBit(int x) {
        return x & -x;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
        int n = Integer.parseInt(in.readLine());
        String[] split = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            int x = Integer.parseInt(split[i]);
            int res = 0;
            while (x > 0) {
                x -= lowBit(x);
                res++;
            }
            out.write(res + " ");
        }
        out.flush();
        in.close();
        out.close();
    }
}
```

> **实现：Python：**

```python
def low_bit(x):
    return x & -x


```

```python
if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(n):
        x = int(arr[i])
        cnt = 0
        while x:
            x -= low_bit(x)
            cnt += 1
        print(cnt, end=" ")
```

1.7 离散化

> **思路：** 将一个操作数据范围很大，但是操作有限的问题；转化到一个可以计算的数据范围内。如数据范围为[负无穷： 正无穷]，但是操作的总范围只有`10^5`~`10^6`，通常使用离散化的方式进行处理，这里只给出C++的模板，别的语言实现有点困难。

> **实现：C++：**

```cpp
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

typedef pair<int, int> PII;

const int N = 300010;

int n, m;
int a[N], s[N];

vector<int> alls;
vector<PII> add, query;

int find(int x)
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x)
        {
            r = mid;
        }
        else
        {
            l = mid + 1;
        }
    }
    return r + 1;
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i++)
    {
        int x, c;
        cin >> x >> c;
        add.push_back({x, c});

        alls.push_back(x);
    }
    for (int i = 0; i < m; i++)
    {
        int l, r;
        cin >> l >> r;
        query.push_back({l, r});

        alls.push_back(l);
        alls.push_back(r);
    }

    // alls数组去重
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());

    for (auto item : add)
    {
        int x = find(item.first);
        a[x] += item.second;
    }

    // 预处理前缀和
    for (int i = 1; i <= alls.size(); i++)
    {
```

```
        s[i] = s[i - 1] + a[i];
    }

    // 处理询问
    for (auto item : query)
    {
        int l = find(item.first), r = find(item.second);
        cout << s[r] - s[l - 1] << endl;
    }

    return 0;
}
```

1.8 区间合并问题

> **思想：** 针对于一个区间内的若干个小区间进行合并，简单的思路就是使用一个数组记录每一个位置是否被覆盖过，然后遍历一遍这个数组整理所有结果，更简单的方法是仅仅使用两个变量来维护当前合并区间的左右边界即可。
>
> (1)先将所有子区间按左端点进行排序；
>
> (2)定义变量`left`和`right`存储当前正在合并的区间信息，如果下一个区间的左端点小于等于当前区间的右端点`right`，则只需要更新`right`为更大的右端点即可；当下一个区间的左端点大于当前区间的右端点`right`时，说明一个新的区间出现了，将当前区间存储后，`left`和`right`更新为新的区间信息；
>
> (3)重复这一过程，直到所有的区间都被更新到

$$left = seg[i-1][l],\ \ right = max(seg[i-1][r], seg[i][r])\ \ if\ seg[i][l] <= seg[i-1][r] \tag{15}$$

$$left = seg[i][l],\ \ right = seg[i][r]\ \ if\ seg[i][l] > seg[i-1][r] \tag{16}$$

> **实现：C++：**

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 100010, INF = -2e9;

int n;
vector<pair<int, int>> segs;

int main()
{
    scanf("%d", &n);
    while (n--)
    {
        int l, r;
        scanf("%d%d", &l, &r);
        segs.push_back({l, r});
    }
    sort(segs.begin(), segs.end());
    int l = INF, r = INF, cnt = 0;
    for (auto item :segs)
    {
        if (l == INF && r == INF)
        {
            l = item.first, r = item.second;
        }
        if (item.first <= r)
        {
            r = max(r, item.second);
        }
        else
        {
            cnt++;
            l = item.first, r = item.second;
        }
    }
    if (l != INF && r != INF)
    {
        cnt++;
    }
    printf("%d", cnt);

    return 0;
}
```

> **实现：Java：**

```
package basic.mergeseg;

import java.io.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
```

```java
import java.util.List;

/**
 * @author LBS59
 * @description 区间合并模板
 */
public class MergeSegment {
    static final int REMOTE = (int) -2e9;

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        int[][] arr = new int[n][2];
        for (int i = 0; i < n; i++) {
            String[] pair = in.readLine().split(" ");
            arr[i][0] = Integer.parseInt(pair[0]);
            arr[i][1] = Integer.parseInt(pair[1]);
        }
        Arrays.sort(arr, Comparator.comparingInt(a -> a[0]));
        List<int[]> res = new ArrayList<>();
        int s = (int) -2e9, e = (int) -2e9;
        for (int[] item : arr) {
            if (s == REMOTE && e == REMOTE) {
                s = item[0];
                e = item[1];
            } else {
                if (item[0] <= e) {
                    e = Math.max(e, item[1]);
                } else {
                    res.add(new int[] {s, e});
                    s = item[0];
                    e = item[1];
                }
            }
        }
        if (s != REMOTE && e != REMOTE) {
            res.add(new int[] {s, e});
        }
        out.write(res.size() + "");

        out.flush();
        in.close();
        out.close();
    }
}
```

> **实现：Python：**

```python
INF = - 2e9

if __name__ == '__main__':
    segments = []
    n = int(input())
    for i in range(n):
        pair = input().split()
        l, r = int(pair[0]), int(pair[1])
        segments.append((l, r))
    segments.sort(key=lambda x: x[0])
    l, r = INF, INF
    cnt = 0
    for pair in segments:
        if l == INF and r == INF:
            l, r = pair[0], pair[1]
        if pair[0] <= r:
            r = max(r, pair[1])
        else:
            cnt += 1
            l, r = pair[0], pair[1]
    if l != INF:
        cnt += 1
    print(cnt)
```

2. 数据结构

> **本章节的数据结构并不是直接调用语言自带的容器库，而是使用数组来模拟各种数据结构，加深对数据机构的印象。**

2.1 链表

> **思想：**和数组相同，链表也是存储数据的容器，数组是内存地址中一片连续的空间，每个元组占用一定的存储空间，使用一个指针作为整个数组的起始地址，通过数据类型作为偏移量，使用起始地址 + 偏移量 * 下标的方式可以快速查找到某一个元素；不同的是，链表使用一种链式连接方式，将内存空间中的碎片充分利用，针对每一个节点，存在一个值域和指针域，分别存储值和下一个节点的下标地址，通过头节点，可以依次找到每一个存储的元素。

2.1.1 单链表

需要记录状态：$e[i]$表示存储存储值的节点；$ne[i]$表示i号节点的下一个节点的下标为$ne[i]$；额外使用变量idx表示下一个节点可以插入节点的下标位置

实现：C++：

```cpp
#include <iostream>

using namespace std;

const int N = 100010;

int head, idx;
int e[N], ne[N];

void init()
{
    head = -1;
    idx = 0;
}

/**
 * 头插法
 **/
void add2Head(int x)
{
    e[idx] = x;
    ne[idx] = head;
    head = idx++;
}

/**
 * 将x插入到下标为k后面
 **/
void add(int k, int x)
{
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx++;
}

/**
 * 删除下标为k的下一个节点
 **/
void remove(int k)
{
    ne[k] = ne[ne[k]];
}
```

实现：Java：

```java
class ListNode {
    private static final int N = 100010;
    /**
     * head表示头节点的下标，e[i]表示节点i的值，ne[i]表示节点i的next指针是多少，这里指下标，idx存储当前可以用到哪个点
     */
    int head, idx;
    int[] e, ne;

    public ListNode() {
        head = -1;
        idx = 0;
        e = new int[N];
        ne = new int[N];
    }

    /**
     * 头插法
     */
    public void add2Head(int x) {
        e[idx] = x;
        ne[idx] = head;
        head = idx++;
    }

    /**
     * 将x值插入到下标为k后面
     * @param k 下标
     * @param x 值
     */
    public void add(int k, int x) {
        e[idx] = x;
        ne[idx] = ne[k];
        ne[k] = idx++;
    }

```

```
    /**
     * 删除下标为k的节点
     * @param k 下标
     */
    public void remove(int k) {
        ne[k] = ne[ne[k]];
    }
}
```

实现：Python：

```
class ListNode:
    def __init__(self):
        self.N = 100010
        self.e = [0] * self.N
        self.ne = [0] * self.N
        self.head = -1
        self.idx = 0

    # 头插法
    def add_2_head(self, x: int) -> None:
        self.e[self.idx] = x
        self.ne[self.idx] = self.head
        self.head = self.idx
        self.idx += 1

    # 将x插入下标为k的节点后面
    def add(self, k: int, x: int) -> None:
        self.e[self.idx] = x
        self.ne[self.idx] = self.ne[k]
        self.ne[k] = self.idx
        self.idx += 1

    # 删除下标为k的节点后面的节点
    def remove(self, k: int) -> None:
        self.ne[k] = self.ne[self.ne[k]]
```

```
N = 100010
head, idx = -1, 0
e, ne = [0] * N, [0] * N


# 将x插入到单链表的头，成为新的头节点
def add_2_head(x: int) -> None:
    global head, idx
    e[idx] = x
    ne[idx] = head
    head = idx
    idx += 1


# 将x插入下标为k的节点后面
def add(k: int, x: int) -> None:
    global head, idx
    e[idx] = x
    ne[idx] = ne[k]
    ne[k] = idx
    idx += 1


## 将下标为k的节点之后的节点删除
def remove(k: int) -> None:
    ne[k] = ne[ne[k]]
```

2.1.2 双链表

需要记录状态：$e[i]$表示存储值的节点；$l[i]$表示i号节点的左节点的下标；$r[i]$表示i号节点的右节点的下标；idx表示下一个可以插入节点的下标 (18)

实现：C++：

```
#include <iostream>

using namespace std;

const int N = 100010;

int idx;
int e[N], l[N], r[N];

void init()
{
    r[0] = 1;
    l[1] = 0;
    idx = 2;
}
```

```

// 在第k个节点之后插入节点x
void add(int k, int x)
{
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx;
    idx++;
}

// 删除第k个节点
void remove(int k)
{
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

实现：Java：

```
class DeListNode {
    private static final int N = 100010;

    int[] e, l, r;
    int idx;

    public DeListNode() {
        // 0表示左端点，1表示右端点
        e = new int[N];
        l = new int[N];
        r = new int[N];
        r[0] = 1;
        l[1] = 0;
        idx = 2;
    }

    /**
     * 在第k个点的右边插入节点x
     * @param k 下标
     * @param x 插入节点
     */
    public void add(int k, int x) {
        e[idx] = x;
        r[idx] = r[k];
        l[idx] = k;
        l[r[k]] = idx;
        r[k] = idx;
        idx++;
    }

    /**
     * 删除第k个节点
     * @param k 待删节点
     */
    public void remove(int k) {
        r[l[k]] = r[k];
        l[r[k]] = l[k];
    }
}
```

实现：Python：

```
class DeListNode:
    def __init__(self):
        self.N = 100010
        self.idx = 0
        self.e = [0] * self.N
        self.l = [0] * self.N
        self.r = [0] * self.N

    # 在第k个节点之后插入节点x
    def add(self, k: int, x: int) -> None:
        self.e[self.idx] = x
        self.r[idx] = self.r[k]
        self.l[idx] = k
        self.l[self.r[k]] = idx
        self.r[k] = idx
        idx += 1

    # 删除第k个节点
    def remove(self, k: int):
        self.r[self.l[k]] = self.r[k]
        self.l[self.r[k]] = self.l[k]
```

```python
N = 100010
idx = 0
e, l, r = [0] * N, [0] * N, [0] * N


# 在第k个插入节点之后插入节点x
def add(k: int, x: int) -> None:
    global idx
    e[idx] = x
    r[idx] = r[k]
    l[idx] = k
    l[r[k]] = idx
    r[k] = idx
    idx += 1


# 删除第k个节点
def remove(k: int) -> None:
    l[r[k]] = l[k]
    r[l[k]] = r[k]
```

2.2 栈

> **思想：** FILO(先进后出)

状态表示：tt表示栈顶的下标位置，初始为 -1，入栈$tt++$，出栈$tt--$,注意判空即可 (19)

> **实现：C++：** 没有判空

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int tt = -1;
int stk[N];

void push(int x)
{
    stk[++tt] = x;
}

void pop()
{
    tt--;
}

bool isEmpty()
{
    return tt < 0;
}

int peek()
{
    return stk[tt];
}
```

> **实现：Java：**

```java
class MyStack {
    private static final int N = 100010;

    int[] stk;
    int tt;

    public MyStack() {
        stk = new int[N];
        tt = -1;
    }

    public void push(int x) {
        if (tt < N) {
            stk[++tt] = x;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            tt--;
        }
    }

    public boolean isEmpty() {
        return tt < 0;
    }
```

```
    public int top() {
        if (!isEmpty()) {
            return stk[tt];
        } else {
            return Integer.MAX_VALUE;
        }
    }
}
```

实现：Python：

```python
N = 100010
tt = -1
stk = [0] * N


# 入栈操作
def push(x: int) -> None:
    global tt
    tt += 1
    stk[tt] = x


# 出栈操作
def pop() -> None:
    global tt
    tt -= 1


# 判空操作
def empty() -> bool:
    global tt
    return tt < 0


# 获取栈顶元素
def peek() -> int:
    global tt
    return stk[tt]
```

```python
class Stack:
    def __init__(self):
        self.N = 100010
        self.tt = -1
        self.stk = [0] * self.N

    def push(self, x: int) -> None:
        self.tt += 1
        self.stk[self.tt] = x

    def pop(self) -> None:
        self.tt -= 1

    def empty(self) -> bool:
        return self.tt < 0

    def peek(self) -> int:
        return self.stk[self.tt]
```

2.3 队列

思想：

状态表示：使用hh表示队头，初始化为0；tt表示队尾，初始化为 -1，注意判空即可 (20)

实现：C++：

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int hh = 0, tt = -1;
int q[N];

void push(int x)
{
    q[++tt] = x;
}

void poll()
{
    hh++;
```

```
}

bool isEmpty()
{
    return hh > tt;
}

int peekHead()
{
    return q[hh];
}

int peekTail()
{
    return q[tt];
}
```

实现：Java：

```java
class MyQueue {
    private static final int N = 100010;

    int[] q;
    int hh, tt;

    public MyQueue() {
        q = new int[N];
        hh = 0;
        tt = -1;
    }

    public void push(int x) {
        if (tt < N) {
            q[++tt] = x;
        }
    }

    public void poll() {
        if (!isEmpty()) {
            hh++;
        }
    }

    public boolean isEmpty() {
        return hh > tt;
    }

    public int peekHead() {
        if (!isEmpty()) {
            return q[hh];
        }
        return Integer.MAX_VALUE;
    }

    public int peekTail() {
        if (!isEmpty()) {
            return q[tt];
        }
        return Integer.MAX_VALUE;
    }
}
```

实现：Python：

```python
class Queue:
    def __init__(self):
        self.N = 100010
        self.hh = 0
        self.tt = -1
        self.q = [0] * self.N

    def push(self, x: int) -> None:
        self.tt += 1
        self.q[self.tt] = x

    def poll(self) -> None:
        self.hh += 1

    def empty(self) -> bool:
        return self.tt < self.hh

    def peekHead(self) -> int:
        return self.q[self.hh]

    def peekTail(self) -> int:
```

```
        return self.q[self.tt]
```

```
N = 100010
hh, tt = 0, -1
q = [0] * N


def push(x: int) -> None:
    global tt
    tt += 1
    q[tt] = x


def poll() -> None:
    global hh
    hh += 1


def empty() -> bool:
    global hh, tt
    return tt < hh


def peekHead() -> int:
    global hh
    return q[hh]


def peekTail() -> int:
    global tt
    return q[tt]
```

2.4 单调栈

> **思想：** 使用栈的结果来完成一个操作，使得存储进栈的元素呈现一种单调的性质
>
> **操作：**
>
> (1)针对一个整数序列，依次压入栈中；
>
> (2)设当前入栈元素为`x`，帧空间为`stk[]`，栈顶指针为`tt`，则栈顶元素为`stk[tt]`，如果当前栈不为空，并且`stk[tt] > x`/`stk[tt] < x`，就让栈顶元素出栈；
>
> (3)重复(2)过程，只要满足栈为空或者栈顶元素`stk[tt] <= x`/`stk[tt] >= x`
>
> (4)将`x`入栈
>
> **用途：** 通过这一系列的操作，我们可以快速求出在一个序列中每一个元素左边/右边离自己最近的大于(大于等于)/小于(小于等于)自己的数。

> **实现：C++：** 输出每一个元素左侧第一个比自己小的数，不存在输出-1

数组模拟

```
#include <iostream>

using namespace std;

const int N = 100010;

int n;
int stk[N], tt;

int main()
{
    cin >> n;
    while (n -- ) {
        int x;
        cin >> x;
        while (tt && stk[tt] >= x)
        {
            tt--;
        }
        if (tt)
        {
            cout << stk[tt] << ' ';
        }
        else
        {
            cout << -1 << ' ';
        }
        stk[++tt] = x;
    }
    return 0;
}
```

使用stl容器

```cpp
#include<iostream>
#include<stack>

using namespace std;

int main()
{
    stack<int> s;
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int x;
        scanf("%d", &x);
        while (!s.empty() && s.top() >= x)
        {
            s.pop();
        }
        if (s.empty())
        {
            printf("-1 ");
        }
        else
        {
            printf("%d ", s.top());
        }
        s.push(x);
    }
    return 0;
}
```

实现：Java： 输出每一个元素左侧第一个比自己小的数，不存在输出-1

数组模拟

```java
import java.io.*;

/**
 * @author LBS59
 * @description 单调栈模板 --- 解决问题：针对数组中每一个元素，找到其左/右边距离其最近比其大/小的元素，不存在则用-1表示
 */
public class Main {
    static final int N = 100010;
    static int[] stk;
    static int tt = 0;

    public static void main(String[] args) throws IOException {
        stk = new int[N];
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        String[] s = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            int cur = Integer.parseInt(s[i]);
            while (tt > 0 && stk[tt] >= cur) {
                tt--;
            }
            if (tt > 0) {
                out.write(stk[tt] + " ");
            } else {
                out.write(-1 + " ");
            }
            stk[++tt] = cur;
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

使用Collection容器

```java
import java.io.*;
import java.util.Stack;

/**
 * @author LBS59
 * @description 单调栈应用
 */
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
```

```
        Stack<Integer> stack = new Stack<>();
        int n = Integer.parseInt(in.readLine());
        String[] s = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            int x = Integer.parseInt(s[i]);
            while (!stack.isEmpty() && stack.peek() >= x) {
                stack.pop();
            }
            if (stack.isEmpty()) {
                out.write(-1 + " ");
            } else {
                out.write(stack.peek() + " ");
            }
            stack.push(x);
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

> **实现：Python：** 输出每一个元素左侧第一个比自己小的数，不存在输出-1

> 数组模拟

```
N = 100010
tt = 0
stk = [0] * N


if __name__ == '__main__':
    n = int(input())
    arr = input().split()
    for i in range(n):
        x = int(arr[i])
        while tt and stk[tt] >= x:
            tt -= 1
        if tt:
            print(stk[tt], end=" ")
        else:
            print(-1, end=" ")
        tt += 1
        stk[tt] = x
```

> 容器模拟

```
if __name__ == '__main__':
    stk = []
    n = int(input())
    arr = input().split()
    for i in range(n):
        x = int(arr[i])
        while stk and stk[-1] >= x:
            stk.pop()
        if stk:
            print(stk[-1], end=" ")
        else:
            print(-1, end=" ")
        stk.append(x)
```

2.5 单调队列

> **思想：** 具体的思想和单调栈类似，都是保证一种规律，保证存储进队列的元素具有单调性。

> **应用：** 单调队列可以使用的范围比较有限，主要可以优化一些滑动窗口中的最值问题。

> **实现：C++：** 确定滑动窗口位于每个位置时，窗口中的最大值和最小值。

> 数组模拟

```
#include<iostream>
using namespace std;

const int N = 1000010;
int a[N], q[N];
int n, k;

int main()
{
    scanf("%d%d", &n, &k);
    for (int i = 0; i < n; i ++) {
        scanf("%d", &a[i]);
    }
    int hh = 0, tt = -1;
```

```
    for (int i = 0; i < n; i ++) {
        if (hh <= tt && i - k + 1 > q[hh]) {
            hh++;
        }
        while (hh <= tt && a[q[tt]] >= a[i]) {
            tt--;
        }
        q[++tt] = i;
        if (i >= k - 1) {
            printf("%d ", a[q[hh]]);
        }
    }
    puts("");

    hh = 0, tt = -1;
    for (int i = 0; i < n; i ++) {
        if (hh <= tt && i - k + 1 > q[hh]) {
            hh++;
        }
        while (hh <= tt && a[q[tt]] <= a[i]) {
            tt--;
        }
        q[++tt] = i;
        if (i >= k - 1) {
            printf("%d ", a[q[hh]]);
        }
    }
    return 0;
}
```

> 使用stl容器

```
#include<iostream>
#include<deque>
using namespace std;

const int N = 1000010;

int n, k;
int a[N];

int main()
{
    scanf("%d%d", &n, &k);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    deque<int> mi, ma;
    for (int i = 0; i < n; i++)
    {
        if (!mi.empty() && i - k + 1 > mi.front())
        {
            mi.pop_front();
        }
        while (!mi.empty() && a[mi.back()] >= a[i])
        {
            mi.pop_back();
        }
        mi.push_back(i);
        if (i >= k - 1)
        {
            printf("%d ", a[mi.front()]);
        }
    }
    puts("");
    for (int i = 0; i < n; i++)
    {
        if (!ma.empty() && i - k + 1 > ma.front())
        {
            ma.pop_front();
        }
        while (!ma.empty() && a[ma.back()] <= a[i])
        {
            ma.pop_back();
        }
        ma.push_back(i);
        if (i >= k - 1)
        {
            printf("%d ", a[ma.front()]);
        }
    }
}
```

> **实现：Java：**确定滑动窗口位于每个位置时，窗口中的最大值和最小值。

数组模拟

```java
import java.io.*;

/**
 * @author LBS59
 * @description 单调队列模板--- 解决问题：等长滑动窗口中的最值问题
 */
public class Main {
    static final int N = 1000010;
    static int[] a = new int[N], q = new int[N];

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s1 = in.readLine().split(" ");
        int n = Integer.parseInt(s1[0]), k = Integer.parseInt(s1[1]);
        String[] s2 = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            a[i] = Integer.parseInt(s2[i]);
        }
        int hh = 0, tt = -1;
        for (int i = 0; i < n; i++) {
            // 判断队头是否已经滑出窗口
            if (hh <= tt && i - k + 1 > q[hh]) {
                hh++;
            }
            while (hh <= tt && a[q[tt]] >= a[i]) {
                tt--;
            }
            q[++tt] = i;
            if (i >= k - 1) {
                System.out.printf("%d ", a[q[hh]]);
            }
        }
        System.out.println();

        hh = 0;
        tt = -1;
        for (int i = 0; i < n; i++) {
            // 判断队头是否已经滑出窗口
            if (hh <= tt && i - k + 1 > q[hh]) {
                hh++;
            }
            while (hh <= tt && a[q[tt]] <= a[i]) {
                tt--;
            }
            q[++tt] = i;
            if (i >= k - 1) {
                System.out.printf("%d ", a[q[hh]]);
            }
        }
        System.out.println();

        out.flush();
        in.close();
        out.close();
    }
}
```

使用容器

```java
import java.io.*;
import java.util.ArrayDeque;
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author LBS59
 * @description 单调队列滑动窗口问题
 */
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), k = Integer.parseInt(s[1]);
        int[] a = new int[n];
        String[] input = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            a[i] = Integer.parseInt(input[i]);
        }
        LinkedList<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
```

```
            if (!queue.isEmpty() && i - k + 1 > queue.peekFirst()) {
                queue.pollFirst();
            }
            while (!queue.isEmpty() && a[queue.peekLast()] >= a[i]) {
                queue.pollLast();
            }
            queue.addLast(i);
            if (i >=  k - 1) {
                out.write(a[queue.peekFirst()] + " ");
            }
        }
        out.write("\n");

        queue.clear();
        for (int i = 0; i < n; i++) {
            if (!queue.isEmpty() && i - k + 1 > queue.peekFirst()) {
                queue.pollFirst();
            }
            while (!queue.isEmpty() && a[queue.peekLast()] <= a[i]) {
                queue.pollLast();
            }
            queue.addLast(i);
            if (i >= k - 1) {
                out.write(a[queue.peekFirst()] + " ");
            }
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

数组模拟

```
N = 1000010
a, q = [0] * N, [0] * N

if __name__ == '__main__':
    s = input().split()
    n, k = int(s[0]), int(s[1])
    arr = input().split()
    for i in range(n):
        a[i] = int(arr[i])
    hh, tt = 0, -1
    for i in range(n):
        # 当前队列不为空，并且队头下标超过了滑动窗口的宽度
        if hh <= tt and i - k + 1 > q[hh]:
            # 弹出队头节点下标
            hh += 1
        # 队列不为空，并且当前队尾的元素大于待插入值
        while hh <= tt and a[q[tt]] >= a[i]:
            # 弹出队尾节点下标
            tt -= 1
        tt += 1
        q[tt] = i
        # 如果窗口容量等于k之后，开始打印结果
        if i >= k - 1:
            print(a[q[hh]], end=" ")
    print()

    hh, tt = 0, -1
    for i in range(n):
        # 当前队列不为空，并且队头下标超过了滑动窗口的宽度
        if hh <= tt and i - k + 1 > q[hh]:
            # 弹出队头节点下标
            hh += 1
        # 队列不为空，并且当前队尾的元素小于等于待插入值
        while hh <= tt and a[q[tt]] <= a[i]:
            # 弹出队尾节点下标
            tt -= 1
        tt += 1
        q[tt] = i
        # 如果窗口容量等于k之后，开始打印结果
        if i >= k - 1:
            print(a[q[hh]], end=" ")
```

使用容器

```
from collections import deque

if __name__ == '__main__':
    s = input().split()
    n, k = int(s[0]), int(s[1])
```

```
    a = []
    arr = input().split()
    for i in range(n):
        a.append(int(arr[i]))
    queue = deque()
    for i in range(n):
        if queue and i - k + 1 > queue[0]:
            queue.popleft()
        while queue and a[queue[-1]] >= a[i]:
            queue.pop()
        queue.append(i)
        if i >= k - 1:
            print(a[queue[0]], end=" ")
    print()

    queue.clear()
    for i in range(n):
        if queue and i - k + 1 > queue[0]:
            queue.popleft()
        while queue and a[queue[-1]] <= a[i]:
            queue.pop()
        queue.append(i)
        if i >= k - 1:
            print(a[queue[0]], end=" ")
```

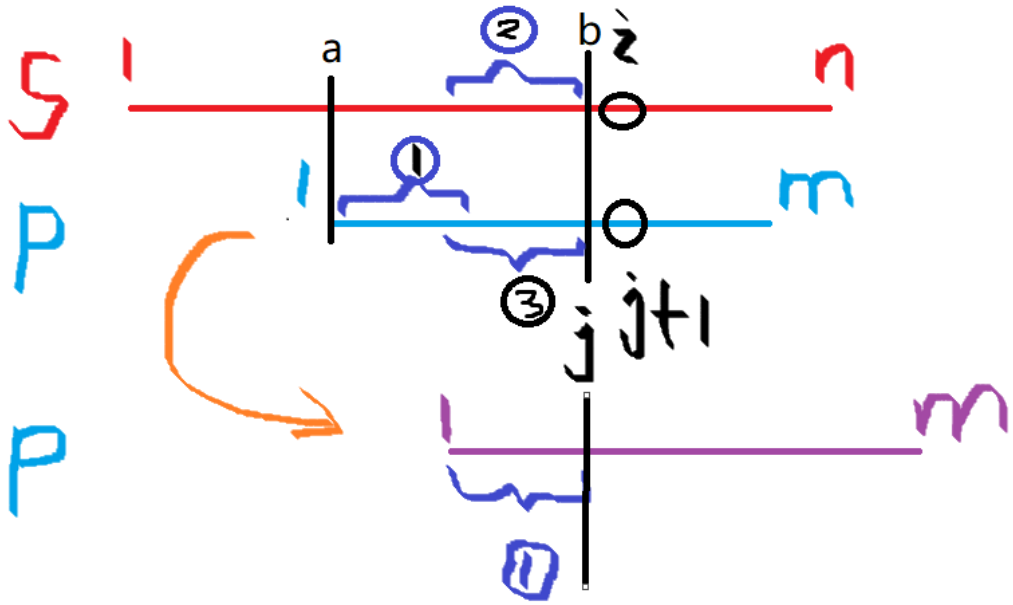
2.6 KMP算法(字符串匹配算法)

> **基础概念：**
>
> 1.s[ ]是模式串，即比较长的字符串。
>
> 2.p[ ]是模板串，即比较短的字符串。（这样可能不严谨。。。）
>
> 3."非平凡前缀"：指除了最后一个字符以外，一个字符串的全部头部组合。
>
> 4."非平凡后缀"：指除了第一个字符以外，一个字符串的全部尾部组合。（后面会有例子，均简称为前/后缀）
>
> 5."部分匹配值"：前缀和后缀的最长共有元素的长度。
>
> 6.next[ ]是"部分匹配值表"，即next数组，它存储的是每一个下标对应的"部分匹配值"，是KMP算法的核心。（后面作详细讲解）。

> **核心思想：**在每次失配时，不是把p串往后移一位，而是把p串往后移动至下一次可以和前面部分匹配的位置，这样就可以跳过大多数的失配步骤。而每次p串移动的步数就是通过查找next[ ]数组确定的。
>
> **next数组含义：**
>
> 对next[j]，是p[1, j]串中前缀和后缀相同的最大长度（部分匹配值），即 p[1, next[j]] = p[j - next[j] + 1, j]。
>
> **举个例子：**
>
> 针对p串：a b c a b
>
> 下标：　1 2 3 4 5
>
> 手动模拟求next数组：

p	a	b	c	a	b
下标	1	2	3	4	5
next[i]	0	0	0	1	2

> **匹配思路：**s[a , b] = p[1, j] && s[i] != p[j + 1] 此时要移动p串（不是移动1格，而是直接移动到下次能匹配的位置）
>
> 其中1串为[1, next[j]]，3串为[j - next[j] + 1 , j]。由匹配可知 1串等于3串，3串等于2串。所以直接移动p串使1到3的位置即可。这个操作可由j = next[j]直接完成。 如此往复下去，当 j == m时匹配成功。



实现：C++：

```cpp
#include <iostream>

using namespace std;

const int N = 100010, M = 10010; //N为模式串长度，M匹配串长度

int n, m;
int ne[M]; //next[]数组，避免和头文件next冲突
char s[N], p[M];  //s为模式串，  p为匹配串

int main()
{
    cin >> n >> s+1 >> m >> p+1;  //下标从1开始

    //求next[]数组
    for(int i = 2, j = 0; i <= m; i++)
    {
        while(j && p[i] != p[j+1]) j = ne[j];
        if(p[i] == p[j+1]) j++;
        ne[i] = j;
    }
    //匹配操作
    for(int i = 1, j = 0; i <= n; i++)
    {
        while(j && s[i] != p[j+1]) j = ne[j];
        if(s[i] == p[j+1]) j++;
        if(j == m)  //满足匹配条件，打印开头下标，从0开始
        {
            //匹配完成后的具体操作
            j = ne[j]; // 再次继续匹配
        }
    }

    return 0;
}
```

实现：Java：

```java
import java.io.*;

/**
 * @author LBS59
 * @description KMP字符串匹配算法模板 --- 这里下标从1开始
 */
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        // p为匹配串-一般较短
        int n = Integer.parseInt(in.readLine());
        String p = in.readLine();
        char[] pcs = new char[n + 1];
        for (int i = 1; i <= n; i++) {
            pcs[i] = p.charAt(i - 1);
        }
```

```java
        // m为模式串-一般较长
        int m = Integer.parseInt(in.readLine());
        String s = in.readLine();
        char[] scs = new char[m + 1];
        for (int i = 1; i <= m; i++) {
            scs[i] = s.charAt(i - 1);
        }

        // 求next数组
        int[] next = new int[n + 1];
        for (int i = 2, j = 0; i <= n; i++) {
            while (j > 0 && pcs[i] != pcs[j + 1]) {
                j = next[j];
            }
            if (pcs[i] == pcs[j + 1]) {
                j++;
            }
            next[i] = j;
        }

        // KMP匹配过程
        for (int i = 1, j = 0; i <= m; i++) {
            while (j > 0 && scs[i] != pcs[j + 1]) {
                j = next[j];
            }
            if (scs[i] == pcs[j + 1]) {
                j++;
            }
            if (j == n) {
                out.write(i - n + " ");
                // 完全匹配后继续搜索
                j = next[j];
            }
        }


        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```python
if __name__ == '__main__':
    n = int(input())
    pcs = input()
    p = [""] * (n + 1)
    for i in range(1, n + 1):
        p[i] = pcs[i - 1]

    m = int(input())
    scs = input()
    s = [""] * (m + 1)
    for i in range(1, m + 1):
        s[i] = scs[i - 1]

    # 求next数组
    ne = [0] * (n + 1)
    j = 0
    for i in range(2, n + 1):
        while j and p[i] != p[j + 1]:
            j = ne[j]
        if p[i] == p[j + 1]:
            j += 1
        ne[i] = j

    # kmp匹配过程
    j = 0
    for i in range(1, m + 1):
        while j and s[i] != p[j + 1]:
            j = ne[j]
        if s[i] == p[j + 1]:
            j += 1
        if j == n:
            print(i - n, end=" ")
            j = ne[j]
```

2.7 Trie树

> **一种快速存储和查找字符串集合的数据结构。**

思想：

> **构建Trie树：**构建的方式采用多叉树的形式，根据题意将所有出现过的字符串按每一个字符作为多叉树某一条路径上的节点，如果其中某个路径上不存在单词中的某个字符，直接创建这个字符节点后继续向下构建，直到所有单词构建完成，在每一次构建一个单词之后，都会在单词末尾节点打标记，表示从根节点到此节点形成的单词出现的次数，实现快速查询。

- 这里程序只给出Trie的关键两个操作，插入字符串和查询字符串的个数：

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int son[N][26], cnt[N], idx; // 下标是0的点，既是根节点，又是空节点
char str[N];

void insert_str(char str[])
{
    int p = 0;
    for (int i = 0; str[i]; i++)
    {
        int u = str[i] - 'a';
        if (!son[p][u])
        {
            son[p][u] = ++idx;
        }
        p = son[p][u];
    }
    cnt[p]++;
}

int query(char str[])
{
    int p = 0;
    for (int i = 0; str[i]; i++)
    {
        int u = str[i] - 'a';
        if (!son[p][u])
        {
            return 0;
        }
        p = son[p][u];
    }
    return cnt[p];
}
```

> **实现：Java：**

```java
class Trie {
    /**
     * son[i][j]记录第i层中j这个路径是否存在，大于0表示有路径
     */
    private final int[][] son;
    /**
     * cnt[i]表示以当前字符结尾的单词的个数
     */
    private final int[] cnt;
    /**
     * 当前所处的层
     */
    private int idx;

    public Trie(int n) {
        son = new int[n][26];
        cnt = new int[n];
        // 初始化根节点为第0层
        idx = 0;
    }

    /**
     * 向字典树中添加字符串s
     **/
    public void insert(String s) {
        // 从根节点开始查询
        int p = 0;
        char[] chs = s.toCharArray();
        for (char c : chs) {
            // 找到对应位置
            int u = c - 'a';
            if (son[p][u] == 0) {
                // 如果当前层没有对象字符，则创建并指向对应字符的路径
                son[p][u] = ++idx;
            }
            p = son[p][u];
        }
```

```
        // 将此位置单词数+1
        cnt[p]++;
    }

    /**
     * 查询字典树中有几个字符串s
     **/
    public int query(String s) {
        // 从根节点开始查询
        int p = 0;
        char[] chs = s.toCharArray();
        for (char c : chs) {
            // 获取字符对应下标
            int u = c - 'a';
            if (son[p][u] == 0) {
                return 0;
            }
            p = son[p][u];
        }
        return cnt[p];
    }
}
```

实现：Python：

```
N = 100010
son = [[0 for _ in range(26)] for _ in range(N)]
cnt = [0] * N
idx = 0


def insert_str(s: str) -> None:
    global idx
    p = 0
    for i in range(len(s)):
        u = ord(s[i]) - ord("a")
        if not son[p][u]:
            idx += 1
            son[p][u] = idx
        p = son[p][u]
    cnt[p] += 1


def query_str(s: str) -> int:
    p = 0
    for i in range(len(s)):
        u = ord(s[i]) - ord("a")
        if not son[p][u]:
            return 0
        p = son[p][u]
    return cnt[p]
```

2.8 并查集

> 一种数据结构，支持两种操作：
>
> - 快速合并两个集合；
> - 询问两个元素是否处于同一个集合中。

> 来个简单的例子，对于一个数列，我们存在一个数组`s[i]`表示i这个元素属于某一个集合；
>
> 当我们判断两个元素`x`和`y`是否属于用一个集合时，直接判断s[x] 和 s[y]相等与否即可；但是当我们要将两个集合合并，比如将集合`a`和集合`b`的元素合并到一个集合中，则至少需要的操作为$max(len(a),len(b))$，这是十分耗时的，但是并查集就可以使用近乎`O(1)`的时间复杂度解决这一问题

> **思想：**每一个集合使用一棵树来表示，树根的编号就是整个结合的编号。每个节点存储它的父节点，`p[x]`表示`x`的父节点
>
> **小问题1：**如何求一个节点是否为父节点: $if(p[x]==x)$，使用表达式判断，因为只有父节点的编号等于它自己
>
> **小问题2：**如何找到一个节点的编号：$while(p[x]!=x)\{x=p[x]\}$，根据树的结构依次向上找父节点
>
> **小问题3：**如何合并两个结合：`p[x]`是`x`的编号，`p[y]`是`y`的编号，直接将x的整棵树作为y所在树的子树即可，$p[x]=y$
>
> **并查集路径压缩：**在查询某一个点`x`的根节点时，当找到`x`的根节点后，将从`x`到所在树根节点上的所有节点直接指向根节点。$if(p[x]!=x)p=find(p[x])$

实现：C++：只实现核心函数

```
#include<iostream>

using namespace std;

const int N = 100010;

int p[N], s[N];

void init()
```

```
{
    for (int i = 1; i <= N; i++)
    {
        p[i] = i;
        s[i] = 1;
    }
}

// 返回x所在子树的编号
int find(int x)
{
    return p[x] == x ? p[x] : find(p[x]);
}

void union(int a, int b)
{
    if (find(a) != find(b)) {
        size[find(b)] += size[find(a)];
        p[find(a)] = find(b);
    }
}
```

实现：Java：

```
/**
 * 并查集模板 --- 路径压缩+连通分量中节点个数统计
 */
class UF {
    /**
     * 记录每一个节点的父亲节点
     */
    int[] parent, size;

    public UF(int n) {
        parent = new int[n];
        size = new int[n];
        for (int i = 1; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /**
     * 获取x的祖宗节点 + 路径压缩
     * @param x 查询节点
     * @return 子树编号
     */
    public int find(int x) {
        return parent[x] == x ? parent[x] : find(parent[x]);
    }

    /**
     * 合并两个子树
     */
    public void union(int a, int b) {
        if (find(a) != find(b)) {
            size[find(b)] += size[find(a)];
            parent[find(a)] = find(b);
        }
    }
}
```

实现：Python：

```
N = 100010
parent, size = [0] * N, [0] * N


def init():
    for i in range(1, N):
        parent[i] = i
        size[i] = 1


def find(x: int) -> int:
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]


def union(a: int, b: int) -> None:
    if parent[a] != parent[b]:
        size[find(b)] += size[find(a)]
        parent[find(a)] = find(b)
```

2.9 堆

> **堆是一种存储数据的数据容器(形如完全二叉树)，其支持三种基本操作：**
>
> - 向堆中插入一个元素；
> - 求堆中的最小/最大值
> - 删除队中的最小/最大值
> - 删除任意一个元素(**非基本操作**)
> - 修改任意一个元素(**非基本操作**)

> **核心思想：**
>
> **结构**：完全二叉树的结构，完全二叉树是二叉树中的一种，除了叶子节点外，所有的层都是满的，叶子节点层的节点都处于二叉树的左侧；
>
> **特点**：二叉树是一种迭代的结构，针对每一个节点，这里以小根堆为例，每一个节点都是当前子树中的最小值，所以根节点是整个堆中的最小值；
>
> **存储**：堆可以直接使用一维数组来存储，我们让下标从`1`开始，并且下标`1`为根节点的坐标，则有一种规律：如果下标为`i`的节点存在左右儿子节点，则左儿子的下标为`2 * i`，右儿子的下标为`2 * i + 1`，通过这个规律可以快速找到所有节点。
>
> **操作**：针对堆中的所有操作，无外乎可以使用两种操作来整合，及向下调整和向上调整方法，下面给出的模板也仅限这两种操作。

> **实现：C++**

```cpp
#include<iostream>
#include<algorithm>
#include<string.h>

using namespace std;

const int N = 100010;

int h[N], ph[N], hp[N], cnt;

void heap_swap(int a, int b)
{
    swap(ph[hp[a], ph[hp[b]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    // 使用t存储u及左右儿子中节点值最小的下标
    int t = u;
    // 如果u存在左儿子并且左儿子小于当前的t
    if (u * 2 <= cnt && h[u * 2] < h[t])
    {
        t = u * 2;
    }
    // 如果u存在右儿子并且右儿子小于当前的t
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] <h[t])
    {
        t = u * 2 + 1;
    }
    // 如果u对应节点不是最小值
    if (u != t)
    {
        // 交换u和最小值的值
        heap_swap(h[u], h[t]);
        // 因为t现在存储的是以前的u，继续调整t
        down(t);
    }
}

void up(int u)
{
    while (u / 2 && h[u / 2] > h[u])
    {
        heap_swap(h[u / 2], h[u]);
        u /= 2;
    }
}
```

> **实现：Java：**

```java
package datastruction.heap;

/**
 * @author LBS59
 * @description 带有映射关系的堆
 */
public class MyHeap {
    /**
     * h存储队中节点的值，ph表示从指针到堆的映射，hp表示从堆到指针的映射
     */
```

```java
    int[] h, ph, hp;
    /**
     * size表示当前队中存储节点的个数
     */
    int size;

    public MyHeap(int n) {
        h = new int[n];
        ph = new int[n];
        hp = new int[n];
        size = 0;
    }

    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public void heapSwap(int a, int b) {
        swap(ph, hp[a], hp[b]);
        swap(hp, a, b);
        swap(h, a, b);
    }

    /**
     * 堆向下调整操作
     * @param u 待调整位置
     */
    public void down(int u) {
        // 使用t来记录当前位置和其左/右儿子中最小值的位置
        int t = u;
        // 如果存在左儿子并且左儿子更小，则较小位置更新
        if (u * 2 <= size && h[u * 2] < h[t]) {
            t = u * 2;
        }
        // 如果存在右儿子并且右儿子更小，则较小位置更新
        if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) {
            t = u * 2 + 1;
        }
        // 如果存在儿子位置变动
        if (u != t) {
            // 则将最小值交换至根位置，确保堆的性质，交换后的t位置为原来根的位置，递归处理即可
            heapSwap(u, t);
            down(t);
        }
    }

    /**
     * 堆向上调整操作
     * @param u 待调整位置
     */
    public void up(int u) {
        // 如果当前位置存在父节点并且父节点的值大于当前值，则交换父子节点的值，并且继续向上调整
        while (u / 2 > 0 && h[u] < h[u / 2]) {
            heapSwap(u / 2, u);
            u /= 2;
        }
    }
}

```

实现：Python：

```python
from typing import List

N = 100010
h, ph, hp = [0] * N, [0] * N, [0] * N
size = 0


def _swap(arr: List, i: int, j: int) -> None:
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp


def heap_swap(a: int, b: int) -> None:
    _swap(ph, hp[a], hp[b])
    _swap(hp, a, b)
    _swap(h, a, b)


def down(u: int) -> None:
    t = u
    if u * 2 <= size and h[u * 2] < h[t]:
```

```
        t = u * 2
    if u * 2 + 1 <= size and h[u * 2 + 1] < h[t]:
        t = u * 2 + 1
    if u != t:
        heap_swap(u, t)
        down(t)


def up(u: int) -> None:
    while u // 2 and h[u // 2] > h[u]:
        heap_swap(u // 2, u)
        u //= 2
```

2.10 哈希表(散列表)

> **哈希表的存储结构：**
>
> - 开放寻址法；
> - 拉链法

> **拉链法C++实现：**

```cpp
// 拉链法实现哈希表
#include<iostream>
#include<cstring>

using namespace std;

const int N = 100003;

// 这里h表示散列中槽的位置，后面的e,ne和idx都是单链表的成分
int h[N], e[N], ne[N], idx;

// 头插法
void insert_val(int x)
{
    // 加N的目的是使得mod后的值为正
    int k = (x % N + N) % N;

    e[idx] = x, ne[idx] = h[k], h[k] = idx++;

}

// 遍历一个槽
bool find_val(int x)
{
    // 加N的目的是使得mod后的值为正
    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);

    while (n--)
    {
        char op[2];
        int x;
        scanf("%s%d", op, &x);

        if (*op == 'I')
        {
            insert_val(x);
        }
        else
        {
            if (find_val(x))
            {
                puts("Yes");
            }
            else
            {
                puts("No");
            }
```

```
        }
    }
    return 0;
}
```

拉链法Java实现：

```java
class MyHash1 {
    /**
     * 这里N取质数
     */
    private static final int N = (int) 1e5 + 3;
    int[] h, e, ne;
    int idx;

    public MyHash1() {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[N];
        ne = new int[N];
        idx = 0;
    }

    public void insert(int x) {
        int hash = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[hash];
        h[hash] = idx++;
    }

    public boolean find(int x) {
        int hash = (x % N + N) % N;
        for (int i = h[hash]; i != -1; i = ne[i]) {
            if (e[i] == x) {
                return true;
            }
        }
        return false;
    }
}
```

拉链法Python实现：

```python
N = int(1e5 + 3)

h, e, ne = [-1] * N, [0] * N, [0] * N
idx = 0


def insert(x: int) -> None:
    global idx
    k = (x % N + N) % N
    e[idx] = x
    ne[idx] = h[k]
    h[k] = idx
    idx += 1


def find(x: int) -> bool:
    k = (x % N + N) % N
    i = h[k]
    while i != -1:
        if e[i] == x:
            return True
        i = ne[i]
    return False
```

开放寻址法C++实现：

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 200003, null = 0x3f3f3f3f;

int h[N];

int find_val(int x)
{
    int k = (x % N + N) % N;

    while (h[k] != null && h[k] != x)
    {
        k++;
```

```
        if (k == N)
        {
            k = 0;
        }
    }
    return k;
}
```

开放寻址法Java实现：

```
class MyHash2 {
    private static final int N = (int) 2e5 + 3, NULL = 0x3f3f3f3f;
    int[] h;

    public MyHash2() {
        h = new int[N];
        Arrays.fill(h, NULL);
    }

    public 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;
    }
}
```

开放寻址法Python实现：

```
N, null = int(2e5 + 3), int(0x3f3f3f3f)

h = [0] * N


def find_val(x: int) -> int:
    k = (x % N + N) % N
    while h[k] != null and h[k] != x:
        k += 1
        if k == N:
            k = 0
    return k
```

3. 搜索与图论

3.1 深度优先搜索-DFS

老生常谈了，直接上code：

DFS没有一个能直接使用的模板，通常的做法是使用计算机底层方法栈的原理做回溯处理，在回溯的过程中进行剪枝操作，下面有两个DFS算法的小例子：

Example1:排列数字：给定一个整数n，将数字1~n拍成一排

实现：C++：

```
#include<iostream>

using namespace std;

const int N = 10;

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

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

```
            st[i] = true;
            dfs(u + 1);
            st[i] = false;
        }
    }
}

int main()
{
    scanf("%d", &n);

    dfs(0);

    return 0;
}
```

实现：Java：

```
import java.util.*;

public class Main {
    static final int N = 10;
    static int[] path = new int[N];
    static boolean[] visited = new boolean[N];

    private static void dfs(int idx, int n) {
        if (idx == n) {
            for (int i = 0; i < n; i++) {
                System.out.printf("%d ", path[i]);
            }
            System.out.println();
            return;
        }
        for (int i = 1; i <= n; i++) {
            if (!visited[i]) {
                path[idx] = i;
                visited[i] = true;
                // 走向下一层
                dfs(idx + 1, n);
                // 回溯，恢复现场
                visited[i] = false;
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        dfs(0, n);
    }
}
```

实现：Python：

```
N = 10

path = [0] * N
st = [False] * N


def dfs(u: int, n: int) -> None:
    if u == n:
        for i in range(n):
            print(path[i], end=" ")
        print()
        return
    for i in range(1, n + 1):
        if not st[i]:
            path[u] = i;
            st[i] = True
            dfs(u + 1, n)
            st[i] = False


if __name__ == "__main__":
    n = int(input())
    dfs(0, n);
```

Example2:n皇后问题： n-n-皇后问题是指将 nn 个皇后放在 n×nn×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。

实现：C++：

```
#include<iostream>

```

```cpp
using namespace std;

const int N = 20;

int n;
char g[N][N];
bool col[N], dg[N], udg[N];

void dfs(int u)
{
    if (u == n)
    {
        for (int i = 0; i < n; i++)
        {
            puts(g[i]);
        }
        puts("");
        return;
    }

    for (int i = 0; i < n; i++)
    {
        if (!col[i] && !dg[u + i] && !udg[n - u + i])
        {
            g[u][i] = 'Q';
            col[i] = dg[u + i] = udg[n - u + i] = true;
            dfs(u + 1);
            g[u][i] = '.';
            col[i] = dg[u + i] = udg[n - u + i] = false;
        }
    }
}

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

    return 0;
}
```

实现：Java：

```java
import java.io.*;

public class Main {
    private static final int N = 20;
    static char[][] g = new char[N][N];
    static boolean[] col = new boolean[N], dg = new boolean[N], udg = new boolean[N];

    private static void dfs(int idx, int n) {
        if (idx == n) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    System.out.printf("%c", g[i][j]);
                }
                System.out.println();
            }
            System.out.println();
        }
        for (int i = 0; i < n; i++) {
            if (!col[i] && !dg[idx + i] && !udg[n - idx + i]) {
                // 找到了一个放置皇后的位置
                g[idx][i] = 'Q';
                col[i] = dg[idx + i] = udg[n - idx + i] = true;
                //寻找下一行放置的位置
                dfs(idx + 1, n);
                // 回溯，现场恢复
                col[i] = dg[idx + i] = udg[n - idx + i] = false;
                g[idx][i] = '.';
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
```

```
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                g[i][j] = '.';
            }
        }
        dfs(0, n);

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
N = 20

g = [["" for _ in range(N)] for _ in range(N)]
col, dg, udg = [False] * N, [False] * N, [False] * N


def dfs(u: int, n: int) -> None:
    if u == n:
        for i in range(n):
            for j in range(n):
                print(g[i][j], end="")
            print()
        print()
    for i in range(n):
        if not col[i] and not dg[u + i] and not udg[n - u + i]:
            g[u][i] = "Q"
            col[i] = dg[u + i] = udg[n - u + i] = True
            dfs(u + 1, n)
            g[u][i] = "."
            col[i] = dg[u + i] = udg[n - u + i] = False


if __name__ == "__main__":
    n = int(input())
    for i in range(n):
        for j in range(n):
            g[i][j] = "."
    dfs(0, n)
```

3.2 广度优先搜索-BFS

> **思想：** BFS通常都有一个较为固定的套路，使用队列来作为中间数据结构，针对一个类似树的结果，采用层级遍历的方式逐个搜索到所有的节点。
>
> **引申问题：** 如果树/图中所有节点的权重为一个固定值，则BFS搜索到的路径一定为最短路径。

> **Example1：走迷宫：** 给定一个 n×mn×m 的二维整数数组，用来表示一个迷宫，数组中只包含 00 或 11，其中 00 表示可以走的路，11 表示不可通过的墙壁。请问，该人从左上角移动至右下角 (n,m)(n,m) 处，至少需要移动多少次。(每次可以向上、下、左、右任意一个方向移动一个位置)

实现：C++：

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

using namespace std;

const int N = 110;

typedef pair<int, int> PII;

int n, m;
// 存储整个迷宫
int g[N][N];
// 存储每一位置到起点的距离
int d[N][N];

int bfs()
{
    memset(d, -1, sizeof d);
    queue<PII> q;
    q.push({0, 0});
    d[0][0] = 0;
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    while (!q.empty()) {
        auto t = q.front();
        q.pop();
        for (int i = 0; i < 4; i++)
        {
```

```
            int x = t.first + dx[i], y = t.second + dy[i];
            if (x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y] == -1)
            {
                d[x][y] = d[t.first][t.second] + 1;
                q.push({x, y});
            }
        }
    }
    return d[n - 1][m - 1];
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> g[i][j];
        }
    }
    cout << bfs() << endl;
}
```

实现：Java：

```
import java.io.*;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author LBS59
 * @description BFS模板及解决问题-走迷宫问题
 */
public class Main {
    private static final int N = 110;
    static int[][] g = new int[N][N], d = new int[N][N];
    static int[] dx = {-1, 0, 1, 0}, dy = {0, 1, 0, -1};

    private static int bfs(int row, int col) {
        Queue<int[]> q = new LinkedList<>();
        q.offer(new int[] {0, 0});
        d[0][0] = 0;
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            for (int i = 0; i < 4; i++) {
                int x = cur[0] + dx[i], y = cur[1] + dy[i];
                if (x >= 0 && x < row && y >= 0 && y < col && g[x][y] == 0 && d[x][y] == -1) {
                    d[x][y] = d[cur[0]][cur[1]] + 1;
                    q.offer(new int[] {x, y});
                }
            }
        }
        return d[row - 1][col - 1];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] strs = in.readLine().split(" ");
        int n = Integer.parseInt(strs[0]), m = Integer.parseInt(strs[1]);
        for (int i = 0; i < n; i++) {
            String[] col = in.readLine().split(" ");
            for (int j = 0; j < m; j++) {
                g[i][j] = Integer.parseInt(col[j]);
            }
        }
        for (int i = 0; i < N; i++) {
            Arrays.fill(d[i], -1);
        }
        System.out.println(bfs(n, m));

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
from collections import deque

N = 110
```

```python
# g用来存储整个矩阵，d用来存储每一个位置到起点的距离
g, d = [[0 for _ in range(N)] for _ in range(N)], [[-1 for _ in range(N)] for _ in range(N)]
dx, dy = [-1, 0, 1, 0], [0, 1, 0, -1]


def bfs(n: int, m: int) -> int:
    q = deque()
    q.append((0, 0))
    d[0][0] = 0
    while q:
        t = q.popleft()
        for i in range(4):
            x, y = t[0] + dx[i], t[1] + dy[i]
            if n > x >= 0 == g[x][y] and 0 <= y < m and d[x][y] == -1:
                d[x][y] = d[t[0]][t[1]] + 1
                q.append((x, y))
    return d[n - 1][m - 1]


if __name__ == '__main__':
    nm = input().split()
    n, m = int(nm[0]), int(nm[1])
    for i in range(n):
        row = input().split()
        for j in range(m):
            g[i][j] = int(row[j])
    print(bfs(n, m))
```

3.3 树/图的存储及遍历

3.3.1 存储

> **存储：**邻接矩阵、邻接表。针对稠密图，可以使用二者任意一个；针对稀疏的图，使用邻接矩阵比较浪费空间，因此这里推荐使用邻接表来存储。

> **邻接表存储图/树：**采取散列表存储，存储方式和建立hash表完全一样，可以借鉴一下，这里直接给出程序。

> **邻接表C++：**

```cpp
#include<iostream>

using namespace std;

const int N = 100010, M = N * 2;

// 这里h表示每一个节点，后面三个变量采取和拉链法相同的存储方法
int h[N], e[M], ne[M], idx;

// 将节点b插入到节点a的拉链中
void add(int a, int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}
```

> **邻接表Java：**

```java
public class Save {
    private static final int N = 100010, M = N * 2;
    /**
     * 这里h存储每一个节点，e表示单链表中的节点的值域，ne表示单链表中的节点的next域
     */
    int[] h, e, ne;
    /**
     * idx表示当前单链表中元素的个数，也是下一个节点插入的下标
     */
    int idx;

    public Save() {
        h = new int[N];
        e = new int[M];
        ne = new int[M];
        idx = 0;
    }

    /**
     * 存储一条从a指向b的边
     * @param a 起点
     * @param b 终点
     */
    public void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }
```

```
}
```

邻接表Python：

```python
N = 100010
M = N * 2

h, e, ne = [0] * N, [0] * M, [0] * M
idx = 0


def add(a: int, b: int) -> None:
    global idx
    e[idx] = b
    ne[idx] = h[a]
    h[a] = idx
    idx += 1
```

3.3.2 遍历

DFS：使用深度优先遍历来遍历图/树，直接上code

实现：C++：

```cpp
#include<iostream>

using namespace std;

const int N = 100010, M = N * 2;

// 这里h表示每一个节点，后面三个变量采取和拉链法相同的存储方法
int h[N], e[M], ne[M], idx;
bool st[N];

// u表示当前遍历下标
void dfs(int u)
{
    st[u] = true;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            dfs(j);
        }
    }
}
```

实现：Java：

```java
class SaveAndTraverse {
    private static final int N = 100010, M = N * 2;
    /**
     * 这里h存储每一个节点，e表示单链表中的节点的值域，ne表示单链表中的节点的next域
     */
    int[] h, e, ne;
    /**
     * 在遍历时起标记作用
     */
    boolean[] st;
    /**
     * idx表示当前单链表中元素的个数，也是下一个节点插入的下标
     */
    int idx;

    public SaveAndTraverse() {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[M];
        ne = new int[M];
        st = new boolean[N];
        idx = 0;
    }

    /**
     * 存储一条从a指向b的边
     * @param a 起点
     * @param b 终点
     */
    public void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

```

```java
    /**
     * dfs遍历图和树
     * @param u 当前遍历位置
     */
    public void dfs(int u) {
        st[u] = true;
        for (int i = h[u]; i != -1; i = ne[i]) {
            int j = e[i];
            if (!st[j]) {
                dfs(j);
            }
        }
    }
}
```

实现：Python：

```python
N = 100010
M = N * 2

h, e, ne, st = [0] * N, [0] * M, [0] * M, [False] * N
idx = 0


def add(a: int, b: int) -> None:
    global idx
    e[idx] = b
    ne[idx] = h[a]
    h[a] = idx
    idx += 1


def dfs(u: int) -> None:
    st[u] = True
    i = h[u]
    while i != -1:
        j = e[i]
        if not st[j]:
            dfs(j)
        i = ne[i]
```

BFS：使用广度优先遍历来遍历图/树，直接上code

实现：C++：使用数组模拟队列

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], idx;
int d[N], q[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void bfs()
{
    int hh = 0, tt = 0;
    q[0] = 1;
    memset(d, -1, sizeof d);
    while (hh < tt)
    {
        int t = q[hh++];
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1)
            {
                d[j] = d[t] + 1;
                q[++tt] = j;
            }
        }
    }
}
```

实现：C++：使用STL容器中的队列

```cpp
#include<iostream>
#include<cstring>
```

```cpp
#include<queue>

using namespace std;

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], idx;
int d[N], q[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void bfs()
{
    queue<int> q;
    memset(d, -1, sizeof d);
    q.push(1);
    d[1] = 0;
    while (!q.empty())
    {
        int t = q.pop();
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1)
            {
                d[j] = d[t] + 1;
                q.push(j);
            }
        }
    }
}
```

实现：Java：两种方式重构

```java
package searchandgraph.treeandgragp;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

class SaveAndTraverse {
    private static final int N = 100010, M = N * 2;
    /**
     * 这里h存储每一个节点，e表示单链表中的节点的值域，ne表示单链表中的节点的next域
     */
    int[] h, e, ne;
    /**
     * 在遍历时起标记作用
     */
    boolean[] st;
    /**
     * idx表示当前单链表中元素的个数，也是下一个节点插入的下标
     */
    int idx;
    /**
     * 这里d表示任意一点到起点的距离，q模拟队列
     */
    int[] d, q;

    public SaveAndTraverse() {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[M];
        ne = new int[M];
        st = new boolean[N];
        idx = 0;
        d = new int[N];
        Arrays.fill(d, -1);
        q = new int[N];
    }

    /**
     * 存储一条从a指向b的边
     * @param a 起点
     * @param b 终点
     */
    public void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

```

```java
    public void bfsWithArrayQueue() {
        int hh = 0, tt = 0;
        q[0] = 1;
        d[1] = 0;
        while (hh <= tt) {
            int t = q[hh++];
            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                if (d[j] == -1) {
                    d[j] = d[t] + 1;
                    q[++tt] = j;
                }
            }
        }
    }

    public void bfsWithQueue() {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(1);
        d[1] = 0;
        while (!queue.isEmpty()) {
            int t = queue.poll();
            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                if (d[j] == -1) {
                    d[j] = d[t] + 1;
                    queue.offer(j);
                }
            }
        }
    }
}
```

实现：Python： 数组模拟队列

```python
N = 100010
h, e, ne, d, q = [-1] * N, [0] * N, [0] * N, [-1] * N, [0] * N
idx = 0


def bfs() -> None:
    hh, tt = 0, 0
    q[0] = 1
    d[1] = 0
    while hh <= tt:
        t = q[hh]
        hh += 1
        i = h[t]
        while i != -1:
            j = e[i]
            if d[j] == -1:
                d[j] = d[t] + 1
                tt += 1
                q[tt] = j
            i = ne[i]
```

实现：Python： 使用deque容器作为队列

```python
from collections import deque

N = 100010
h, e, ne, d = [-1] * N, [0] * N, [0] * N, [-1] * N
idx = 0


def bfs() -> None:
    q = deque()
    q.appendleft(1)
    d[1] = 0
    while q:
        t = q.pop()
        i = h[t]
        while i != -1:
            j = e[i]
            if d[j] == -1:
                d[j] = d[t] + 1
                q.appendleft(j)
            i = ne[i]
```

3.4 有向图的拓扑序列

> **什么是拓扑系列：**针对一个已知的有向图，针对图中的每一条边，从`xi`指向`yi`，并且我们可以构造出一个序列，满足序列中的每个节点的值都满足`xi`出现在`yi`的前面，我们成这个序列为有向图的拓扑序列。
>
> **有向图中的度：**有向图中的每一个节点有入度和出度两个概念，入度即有边指向自己，出度即从自己引出一条边指向别人。
>
> **小技巧：**有向图中，所有入度为0的节点都可以作为拓扑序列的起点，因为没有边指向自己，没有一个节点可以排在自己前面.
>
> **核心思想：**使用上面的小技巧，可以使用广度优先遍历构造拓扑序列，创建一个队列，先将入度为0的所有节点入队，作为起点，然后逐层遍历每一个节点的出度指向的节点，更新出度。循环此过程，直到所有节点都遍历一次结束。

> **实现：C++：**使用数组模拟队列

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], idx;
int d[N], q[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool topsort()
{
    int hh = 0, tt = -1;
    for (int i = 1; i <= n; i++)
    {
        if (!d[i])
        {
            q[++tt] = i;
        }
    }
    while (hh <= tt)
    {
        int t = q[hh++];
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            d[j]--;
            if (d[j] == 0)
            {
                q[++tt] = j;
            }
        }
    }

    return tt == n - 1;
}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        d[b]++;
    }

    if (topsort())
    {
        for (int i = 0; i < n; i++)
        {
            printf("%d ", q[i]);
        }
        puts("");
    }
    else
    {
        puts("-1");
    }

    return 0;
}
```

> **实现：C++：**使用STL库中的queue容器

```cpp
#include<iostream>
#include<cstring>
#include<queue>

using namespace std;

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], idx;
int d[N], res[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool topsort()
{
    queue<int> q;
    int cnt = 0;
    for (int i = 1; i <= n; i++) {
        if (!d[i])
        {
            q.push(i);
            res[cnt++] = i;
        }
    }

    while (!q.empty())
    {
        int t = q.front();
        q.pop();
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            d[j]--;
            if (d[j] == 0)
            {
                q.push(j);
                res[cnt++] = j;
            }
        }
    }

    return cnt == n;
}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        d[b]++;
    }

    if (topsort())
    {
        for (int i = 0; i < n; i++)
        {
            printf("%d ", res[i]);
        }
        puts("");
    }
    else
    {
        puts("-1");
    }

    return 0;
}
```

实现：Java： 两种实现队列方式放在一起

```java
package searchandgraph.treeandgragp;

import java.io.*;
import java.util.*;

class TopSort {
    private final int N = 100010;

```

```java
    /**
     * 散列结构四件套
     */
    private int[] h, e, ne;
    private int idx, n, cnt;

    /**
     * d数组存储每一个节点的入度，q用来模拟队列实现
     */
    int[] d, q, res;

    public TopSort(int n) {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[N];
        ne = new int[N];
        idx = 0;
        this.n = n;
        cnt = 0;
        d = new int[N];
        q = new int[N];
        res = new int[N];
    }

    /**
     * 存在一条边: a >> b
     * @param a 起始节点
     * @param b 终止节点
     */
    public void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    public boolean topSortWithArrayQueue() {
        int hh = 0, tt = -1;
        for (int i = 1; i <= n; i++) {
            if (d[i] == 0) {
                q[++tt] = i;
            }
        }
        while (hh <= tt) {
            int t = q[hh++];
            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                d[j]--;
                if (d[j] == 0) {
                    q[++tt] = j;
                }
            }
        }

        return tt == n - 1;
    }

    public boolean topSortWithQueue() {
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 1; i <= n; i++) {
            if (d[i] == 0) {
                queue.offer(i);
                res[cnt++] = i;
            }
        }
        while (!queue.isEmpty()) {
            int t = queue.poll();
            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                d[j]--;
                if (d[j] == 0) {
                    queue.offer(j);
                    res[cnt++] = j;
                }
            }
        }
        return cnt == n;
    }
}

/**
 * @author LBS59
 * @description 树和图的谱图排序实现
 */
public class TopSortImpl {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
```

```
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        TopSort topSort = new TopSort(n);
        while (m-- > 0) {
            String[] r = in.readLine().split(" ");
            int a = Integer.parseInt(r[0]), b = Integer.parseInt(r[1]);
            topSort.add(a, b);
            topSort.d[b]++;
        }
        if (topSort.topSortWithArrayQueue()) {
            for (int i = 0; i < n; i++) {
                out.write(topSort.q[i] + " ");
            }
            out.write("\n");
        } else {
            out.write(-1 + "\n");
        }

        if (topSort.topSortWithQueue()) {
            for (int i = 0; i < n; i++) {
                out.write(topSort.res[i] + " ");
            }
            out.write("\n");
        } else {
            out.write(-1 + "\n");
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python： 使用数组模拟队列

```
N = 100010
h, e, ne = [-1] * N, [0] * N, [0] * N
idx = 0
d, q = [0] * N, [0] * N


def add(a: int, b: int) -> None:
    global idx
    e[idx] = b
    ne[idx] = h[a]
    h[a] = idx
    idx += 1


def top_sort(n: int) -> bool:
    hh, tt = 0, -1
    for i in range(1, n + 1):
        if not d[i]:
            tt += 1
            q[tt] = i
    while hh <= tt:
        t = q[hh]
        hh += 1
        temp = h[t]
        while temp != -1:
            j = e[temp]
            d[j] -= 1
            if not d[j]:
                tt += 1
                q[tt] = j
            temp = ne[temp]
    return tt == n - 1


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), s[1]
    for i in range(m):
        row = input().split()
        a, b = int(row[0]), int(row[1]);
        add(a, b)
        d[b] += 1
    if top_sort(n):
        for i in range(n):
            print(q[i], end=" ")
        print()
    else:
        print(-1)
```

实现：Python： 使用deque容器作为队列

```python
import collections

N = 100010
h, e, ne = [-1] * N, [0] * N, [0] * N
idx, cnt = 0, 0
d, res = [0] * N, [0] * N


def add(a: int, b: int) -> None:
    global idx
    e[idx] = b
    ne[idx] = h[a]
    h[a] = idx
    idx += 1


def top_sort(n: int) -> bool:
    global cnt
    queue = collections.deque()
    for i in range(1, n + 1):
        if not d[i]:
            queue.appendleft(i)
            res[cnt] = i
            cnt += 1
    while queue:
        t = queue.pop()
        temp = h[t]
        while temp != -1:
            j = e[temp]
            d[j] -= 1
            if not d[j]:
                queue.appendleft(j)
                res[cnt] = j
                cnt += 1
            temp = ne[temp]
    return cnt == n


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), s[1]
    for i in range(m):
        row = input().split()
        a, b = int(row[0]), int(row[1]);
        add(a, b)
        d[b] += 1
    if top_sort(n):
        for i in range(n):
            print(res[i], end=" ")
        print()
    else:
        print(-1)
```

3.5 最短路问题

3.5.1 朴素版Dijkstra算法

> **核心思想：**贪心：
>
> - 首先设置所有节点到达起点的距离，起点节点距离为0，其余节点到达起点的距离为∞，并且使用一个结合s来存储所有已确定距离的所有节点
> - 从1~n遍历所有节点，从中找出一个确定距离的，距离起点最近的点t，使用t来更新其他点的距离，简单理解就是，如果存在一个节点s，从t到s经过权重未w的边可以达到，则可以从起点到s，起点到t，t到s中的权重和选较小者作为节点s的新的距离长，即$d[s]=min(d[s],d[t]+w_{t->s})$，并且每次更新的距离就是当前节点的所有出边指向的所有节点；
> - 重复上述过程，直到确定所有节点的最短路径。

> **应用：**针对于稠密图的最短路单源最短路算法，稠密图就是点少边多的图，边的数量级和点的平方为统一数量级。

> **实现：C++：**

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 510;

int n, m;
int g[N][N];
int dist[N];
bool st[N];

int dijkstra()
{
    // 将所有节点距离初始化为正无穷
    memset(dist, 0x3f, sizeof dist);
    // 起点距离设置为0
    dist[1] = 0;
    for (int i = 1; i <= n; i++)
    {
        int t = -1;
        for (int j = 1; j <= n; j++)
        {
            // 在所有未确定距离的点中找出一个距离起点最近的点
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
            {
                t = j;
            }
        }
        if (t == n) {
            break;
        }
        st[t] = true;
        // 使用t节点更新所有节点的最短距离
        for (int j = 1; j <= n; j++)
        {
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        }
    }
    if (dist[n] == 0x3f3f3f3f)
    {
        // 从起点不存在一条通往节点n的边
        return -1;
    }
    return dist[n];
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(g, 0x3f, sizeof g);

    while (m--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        g[a][b] = min(g[a][b], c);
    }
    int t = dijkstra();

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

    return 0;
}
```

> **实现：Java：**

```java
package searchandgraph.shortestroad;

import java.io.*;
import java.util.Arrays;
```

```java

/**
 * @author LBS59
 * @description 朴素版迪杰斯特拉算法---解决稠密图的最短路问题(单源最短路问题)
 */
public class Dijkstra1 {
    private static final int N = 510, INF = 0x3f3f3f3f;
    static int[][] g;
    static int[] dist;
    static boolean[] vis;

    static {
        g = new int[N][N];
        for (int i = 0; i < N; i++) {
            Arrays.fill(g[i], INF);
        }
        dist = new int[N];
        Arrays.fill(dist, INF);
        vis = new boolean[N];
    }

    private static int dijkstra(int n) {
        dist[1] = 0;
        // 遍历剩余没有遍历到的点中距离最近者
        for (int i = 0; i < n; i++) {
            int t = -1;
            for (int j = 1; j <= n; j++) {
                if (!vis[j] && (t == -1 || dist[t] > dist[j])) {
                    t = j;
                }
            }
            if (t == n) {
                break;
            }
            vis[t] = true;
            for (int j = 1; j <= n; j++) {
                dist[j] = Math.min(dist[j], dist[t] + g[t][j]);
            }
        }
        if (dist[n] == INF) {
            return -1;
        }
        return dist[n];
    }


    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        for (int i = 0; i < m; i++) {
            String[] input = in.readLine().split(" ");
            int a = Integer.parseInt(input[0]), b = Integer.parseInt(input[1]), w =
Integer.parseInt(input[2]);
            g[a][b] = Math.min(g[a][b], w);
        }
        out.write(dijkstra(n) + "\n");

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```python
N, INF = 510, int(1e9 + 7)

g = [[INF for _ in range(N)] for _ in range(N)]
dist = [INF for _ in range(N)]
st = [False for _ in range(N)]


def dijkstra(n: int) -> int:
    global INF
    dist[1] = 0
    for i in range(1, n + 1):
        t = -1
        for j in range(1, n + 1):
            if not st[j] and (t == -1 or dist[t] > dist[j]):
                t = j
        if t == n:
            break
        st[t] = True
```

```
        for j in range(1, n + 1):
            dist[j] = min(dist[j], dist[t] + g[t][j])
    if dist[n] == INF:
        return -1
    return dist[n]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        g[a][b] = min(g[a][b], c)
    t = dijkstra(n)
    print(t)
```

3.5.2 堆优化版Dijkstra算法

> **核心思想：** 核心思想同朴素版Dijkstra算法相同，因为使用朴素版计算的时间复杂度为$O\log N^2$，这里`N`为图中点的个数，针对于稀疏图可以使用，但是针对于稠密图，就是点多边少，二者处于同样的数量级，但是N为10^5数量级，这时候使用朴素版算法就会超时，需要堆算法进行改进。
>
> **堆优化方式：** 堆可以快速找到一个序列中最大/最小值的功能，这个在朴素版Dijkstra算法中使用$O(N)$时间复杂度来处理，效率不高，但是使用堆的数据结构可以实现在$O(1)$时间复杂度内求得最值，相反的，原先的朴素版更新每一个点的最短路是$O(1)$时间复杂度，但是在堆中求改元素的值需要调整，复杂度为$O(logM)$，这里`M`为边的个数，这里有一个小技巧，在更新每一个点的最短路时，可以直接将新的最短路加入到堆中，这样就不用实现堆调整的逻辑了，只需要使用一个额外的bool数组记录每一个点的最短路是否已经确定。

- 这里无需手写堆，直接使用对应的容器即可

> **实现：C++：**

```
#include<iostream>
#include<cstring>
#include<queue>

using namespace std;

typedef pair<int, int> PII;

const int N = 100010;

int n, m;
// 稀疏图使用邻接表来存储图
int h[N], e[N], w[N], ne[N], idx;
int dist[N];
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

int dijkstra()
{
    // 将所有节点距离初始化为正无穷
    memset(dist, 0x3f, sizeof dist);
    // 起点距离设置为0
    dist[1] = 0;

    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});
    while (heap.size())
    {
        auto = heap.top();
        heap.pop();

        int ver = t.second, dis = t.first;
        if (st[ver])
        {
            continue;
        }
        st[ver] = true;
        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dis + w[i])
            {
                dist[j] = dis + w[i];
                heap.push({dist[j], j})
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f)
```

```
    {
        // 从起点不存在一条通往节点n的边
        return -1;
    }
    return dist[n];
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);

    while (m--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }
    int t = dijkstra();

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

    return 0;
}
```

实现：Java：

```
package searchandgraph.shortestroad;

import java.io.*;
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;

/**
 * @author LBS59
 * @description 堆优化版的dijkstra算法，解决稀疏图中的最短路问题(单源最短路算法)
 */
public class Dijkstra2 {
    private static final int N = 150010, INF = 0x3f3f3f3f;
    /**
     * 稀疏图(边较点少的图)节点过多，不能用邻接矩阵来存，使用于邻接表，w存储每条边的权重
     */
    private static final int[] h, e, ne, w;
    private static int idx;
    /**
     * dist存储每个点到起点的距离
     */
    private static int[] dist;
    /**
     * vis过滤掉已经确定最短路径的点
     */
    private static boolean[] vis;

    static {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[N];
        ne = new int[N];
        w = new int[N];
        idx = 0;
        dist = new int[N];
        Arrays.fill(dist, INF);
        vis = new boolean[N];
    }

    public static void add(int from, int to, int weight) {
        e[idx] = to;
        w[idx] = weight;
        ne[idx] = h[from];
        h[from] = idx++;
    }

    public static int dijkstra(int n) {
        dist[1] = 0;
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(o -> o[0]));
        pq.offer(new int[] {0, 1});
        while (!pq.isEmpty()) {
            int[] cur = pq.poll();
            int ver = cur[1], dis = cur[0];
            if (vis[ver]) {
                continue;
            }
            if (ver == n) {
                break;
            }
```

```
            vis[ver] = true;
            for (int i = h[ver]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dis + w[i]) {
                    dist[j] = dis + w[i];
                    pq.offer(new int[] {dist[j], j});
                }
            }
        }
        if (dist[n] == INF) {
            return -1;
        }
        return dist[n];
    }


    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            add(a, b, c);
        }
        out.write(dijkstra(n) + "\n");

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
import heapq

N, INF = 150010, int(0x3f3f3f3f)

# 稀疏图使用邻接表来存储
h, e, ne, w, dist, vis = [-1] * N, [0] * N, [0] * N, [0] * N, [INF] * N, [False] * N
idx = 0


def add(fr: int, to: int, wei: int) -> None:
    global idx
    e[idx] = to
    w[idx] = wei
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def dijkstra(num: int) -> int:
    dist[1] = 0
    heap = []
    heapq.heappush(heap, (0, 1))
    while heap:
        cur = heapq.heappop(heap)
        ver, dis = int(cur[1]), int(cur[0])
        if vis[ver]:
            continue
        if ver == num:
            break
        vis[ver] = True
        temp = h[ver]
        while temp != -1:
            j = e[temp]
            if dist[j] > dis + w[temp]:
                dist[j] = dis + w[temp]
                heapq.heappush(heap, (dist[j], j))
            temp = ne[temp]
    if dist[num] == INF:
        return -1
    return dist[num]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        add(a, b, c)
```

```
    print(dijkstra(n))
```

3.5.3 Bellman-ford算法

> **核心思想：**暴力解决单源最短路问题，但是可能图中存在负权边。
>
> - 遍历`n - 1`次，这里n为点的个数，为什么是`n - 1`后面会说一个问题；
> - 每次使用上一次备份的所有节点的最短距离来更新图中所有边的最短距离，因为如果不使用上次的结果，在这一次更新的时候会形成串联更新的现象，导致更新的次数变多。
>
> **问题：**当一个图中全在负权回路时，使用Bellman-ford算法更新时某些节点存在于回路中，得到的结果可能就不准确，此外，使用此算法可以判断是否存在负权回路：假设存在节点数为`n`，我们遍历`n`次更新，假设在第`n`次更新时，还会存在有路径更新的问题，那么说明某个节点存在一条经过`n`个点的最短路，但是节点数只有`n`，无环图最多只有`n - 1`条边，故图中存在回路。

> **实现：C++：**

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>

using namespace std;

const int N = 510, M = 10010;

int n, m, k;
int dist[N], backup[N];

struct Edge
{
    int a, b, w;
}edges[M];

int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for (int i = 0; i < k; i++)
    {
        memcpy(backup, dist, sizeof dist);
        for (int j = 0; j < m; j++)
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            dist[b] = min(dist[b], backup[a] + w);
        }
    }
    return dist[n];
}

int main()
{
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 0; i < m; i++)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edges[i] = {a, b, c};
    }
    int t = bellman_ford();

    if (t > 0x3f3f3f3f / 2)
    {
        puts("impossible");
    }
    else
    {
        printf("%d\n", t);
    }

    return 0;
}
```

> **实现：Java：**

```java
package searchandgraph.shortestroad;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * Bellman-ford算法只需要遍历所有边，因此使用Edge类数组存储每一条边，遍历起来比较方便
 */
class Edge {
    int from;
    int to;
```

```
    int weight;

    public Edge(int from, int to, int weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
}

/**
 * @author LBS59
 * @description 解决单源最短路中存在负权边的问题
 */
public class BellmanFord {
    private static final int N = 510, M = 10010, INF = 0x3f3f3f3f;
    /**
     * dist[i]存储从起点到节点i的最短路径
     */
    private static int[] dist;
    /**
     * 防止串联更新，使用backup每次存储上次的结果
     */
    private static int[] backup;
    /**
     * 使用Edge数组存储图中的每一条边
     */
    private static Edge[] edges;

    static {
        dist = new int[N];
        Arrays.fill(dist, INF);
        backup = new int[N];
        edges = new Edge[M];
    }

    public static int bellmanFord(int n, int m, int k) {
        dist[1] = 0;
        for (int i = 0; i < k; i++) {
            backup = Arrays.copyOf(dist, dist.length);
            for (int j = 0; j < m; j++) {
                int a = edges[j].from, b = edges[j].to, w = edges[j].weight;
                dist[b] = Math.min(dist[b], backup[a] + w);
            }
        }
        return dist[n];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]), k = Integer.parseInt(s[2]);
        for (int i = 0; i < m; i++) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            edges[i] = new Edge(a, b, c);
        }
        int t = bellmanFord(n, m, k);
        if (t > INF / 2) {
            System.out.println("impossible");
        } else {
            System.out.println(t);
        }

        in.close();
    }
}
```

实现：Python：

```
N, M, INF = 510, 10010, int(0x3f3f3f3f)


class Edge:
    def __init__(self, fr: int, to: int, wei: int):
        self.fr = fr
        self.to = to
        self.wei = wei


dist, backup, edges = [INF] * N, [0] * N, [None] * M


def bellman_ford(n: int, m: int, k: int) -> int:
    dist[1] = 0
    for i in range(k):
```

```
        for j in range(N):
            backup[j] = dist[j]
        for j in range(m):
            a, b, c = edges[j].fr, edges[j].to, edges[j].wei
            dist[b] = min(dist[b], backup[a] + c)
    return dist[n]


if __name__ == '__main__':
    s = input().split()
    n, m, k = int(s[0]), int(s[1]), int(s[2])
    for i in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        edges[i] = Edge(a, b, c)
    t = bellman_ford(n, m, k)
    if t > INF // 2:
        print("impossible")
    else:
        print(t)
```

3.5.4 SPFA算法（帝中帝）

> **核心思想：** 针对于Bellman-ford算法的优化，Bellman-ford中有一行程序为`dist[b] = min(dist[b], backup[a] + c)`，这里是无条件更新，在很多情况下，如果`backup[a]`没有发生变化时，`dist[b]`就不会发生改变，就是做了很多无用功，而SPFA算法就是针对这一问题进行了优化，在上述公式中，只有当`backup[a]`变小后，整体的值`backup[a] + c`才有可能替换掉`dist[b]`，而其余情况都不会更新，在SPFA算法中，当每一次`backup[a]`在一次更新变小后，才有可能更新其所有出边为更小的最短路，如果`backup[a]`在一次更新后没有变化，在下次更新后就不会在操作此节点。

> **实现：C++：**

```
#include<iostream>
#include<cstring>
#include<queue>

using namespace std;

const int N = 100010;

int n, m;
int h[N], w[N], e[N], ne[N], idx;
int dist[N];
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    queue<int> q;
    q.push(1);
    st[1] = true;
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return dist[n];
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m--)
    {
        int a, b, c;
```

```
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }
    int t = spfa();
    if (t == 0x3f3f3f3f)
    {
        puts("impossible");
    }
    else
    {
        printf("%d\n", t);
    }

    return 0;
}
```

实现：Java：

```
package searchandgraph.shortestroad;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author LBS59
 * @description 对Bellman-ford优化的SPFA算法
 */
public class SPFA {
    private static final int N = 100010, INF = 0x3f3f3f3f;

    private static int[] h, e, ne, w;
    private static int[] dist;
    private static boolean[] vis;
    private static int idx;

    static {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[N];
        ne = new int[N];
        w = new int[N];
        dist = new int[N];
        Arrays.fill(dist, INF);
        vis = new boolean[N];
        idx = 0;
    }

    public static void add(int a, int b, int c) {
        e[idx] = b;
        w[idx] = c;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    public static int spfa(int n) {
        Queue<Integer> q = new LinkedList<>();
        dist[1] = 0;
        vis[1] = true;
        q.offer(1);
        while (!q.isEmpty()) {
            int t = q.poll();
            vis[t] = false;
            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dist[t] + w[i]) {
                    dist[j] = dist[t] + w[i];
                    if (!vis[j]) {
                        q.offer(j);
                        vis[j] = true;
                    }
                }
            }
        }
        return dist[n];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        while (m-- > 0) {
```

```
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            add(a, b, c);
        }
        int t = spfa(n);
        if (t == INF) {
            System.out.println("impossible");
        } else {
            System.out.println(t);
        }

        in.close();
    }
}
```

实现：Python：

```
from collections import deque

N, INF = 100010, int(0x3f3f3f3f)

h, e, ne, w, dist, st = [-1] * N, [0] * N, [0] * N, [0] * N, [INF] * N, [False] * N
idx = 0


def add(fr: int, to: int, wei: int) -> None:
    global idx
    e[idx] = to
    w[idx] = wei
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def spfa(num: int) -> int:
    q = deque()
    dist[1] = 0
    st[1] = True
    q.append(1)
    while q:
        x = q.popleft()
        st[x] = False
        temp = h[x]
        while temp != -1:
            j = e[temp]
            if dist[j] > dist[x] + w[temp]:
                dist[j] = dist[x] + w[temp]
                if not st[j]:
                    st[j] = True
                    q.append(j)
            temp = ne[temp]
    return dist[num]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        add(a, b, c)
    t = spfa(n)
    if t == INF:
        print("impossible")
    else:
        print(t)
```

小技巧：使用SPFA算法判断图中是否存在负权回路

思想：抽屉原理，我们每一次更新一个距离dist时，同时更新这个点到起点所要经过的边的数量，每次更新边的数量我们都判断当前经过的边数是否大于等于`n`，`n`为图中节点的个数，因为只有`n`个点，最短路按理最多经过`n-1`条边就可到达，经过`n`条边必然存在`n+1`个点，所以必然存在负权回路。

实现：C++：

```
#include<iostream>
#include<cstring>
#include<queue>

using namespace std;

const int N = 10010;

int n, m;
int h[N], e[N], ne[N], w[N], idx;
int dist[N], cnt[N];
```

```cpp
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

bool spfa()
{
    queue<int> q;
    for (int i = 1; i <= n; i++)
    {
        st[i] = true;
        q.push(i);
    }
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n)
                {
                    return true;
                }
                if (!st[j])
                {
                    st[j] = true;
                    q.push(j);
                }
            }
        }
    }
    return false;
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }
    if (spfa())
    {
        puts("Yes");
    }
    else
    {
        puts("No");
    }

    return 0;
}
```

实现：Java：

```java
package searchandgraph.shortestroad;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author LBS59
 * @description SPFA算法判断有向图中是否存在负权回路
 */
public class SPFANegRound {
    private static final int N = 10010;

    /**
     * 散列表四件套加w权重数组
     */
```

```java
    private static int[] h, w, e, ne;
    private static int idx;
    /**
     * 记录每一个点到起点的最短距离，cnt表示最短距离经过的变数
     */
    private static int[] dist, cnt;
    /**
     * 判重数组，防止重复访问
     */
    private static boolean[] vis;

    static {
        h = new int[N];
        Arrays.fill(h, -1);
        w = new int[N];
        e = new int[N];
        ne = new int[N];
        idx = 0;
        dist = new int[N];
        cnt = new int[N];
        vis = new boolean[N];
    }

    public static void add(int a, int b, int c) {
        e[idx] = b;
        w[idx] = c;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    public static boolean spfa(int n) {
        Queue<Integer> q = new LinkedList<>();
        for (int i = 1; i <= n; i++) {
            vis[i] = true;
            q.offer(i);
        }
        while (!q.isEmpty()) {
            int t = q.poll();
            vis[t] = false;

            for (int i = h[t]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dist[t] + w[i]) {
                    dist[j] = dist[t] + w[i];
                    cnt[j] = cnt[t] + 1;
                    if (cnt[j] >= n) {
                        return true;
                    }
                    if (!vis[j]) {
                        vis[j] = true;
                        q.offer(j);
                    }
                }
            }
        }
        return false;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            add(a, b, c);
        }
        if (spfa(n)) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }

        in.close();
    }
}
```

实现：Python：

```python
from collections import deque

N = 10010
h, w, e, ne, idx, dist, cnt, st = [-1] * N, [0] * N, [0] * N, [0] * N, 0, [0] * N, [0] * N, [False] * N


```

```python
def add(fr: int, to: int, wei: int) -> None:
    global idx
    e[idx] = to
    w[idx] = wei
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def spfa(num: int) -> bool:
    q = deque()
    for i in range(1, num + 1):
        st[i] = True
        q.append(i)
    while q:
        t = q.popleft()
        st[t] = False
        temp = h[t]
        while temp != -1:
            j = e[temp]
            if dist[j] > dist[t] + w[temp]:
                dist[j] = dist[t] + w[temp]
                cnt[j] = cnt[t] + 1
                if cnt[j] >= num:
                    return True
                if not st[j]:
                    st[j] = True
                    q.append(j)
            temp = ne[temp]
    return False


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        add(a, b, c)
    if spfa(n):
        print("Yes")
    else:
        print("No")
```

3.5.5 Floyd算法(多源最短路问题)

> **核心思想：** 动态规划：暴力
>
> - 使用邻接矩阵`g[i][j]`来存储有向图；
> - 循环`n`轮，`n`为图中节点个数。每一轮做的事情就是更新当前轮中每一条边的最短路径距离，即
>
> $$g[i][j] = min(g[i][j], g[i][k] + g[k][j])$$ 其中k为循环的轮数，表示当前轮更新的最短路的所有情况边的数量不超过k条 (21)
>
> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 210, INF = 0x3f3f3f3f;

int n, m, k;
int d[N][N];

void floyd()
{
    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]);
            }
        }
    }
}

int main()
{
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
        {
            if (i == j)
            {
```

```
                d[i][j] == 0;
            }
            else
            {
                d[i][j] = INF;
            }
        }
    }
    while (m--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        d[a][b] = min(d[a][b], c);
    }
    floyd();
    while (k--)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        if (d[a][b] > INF / 2)
        {
            puts("impossible");
        }
        else
        {
            printf("%d\n", d[a][b]);
        }
    }
}
```

实现：Java：

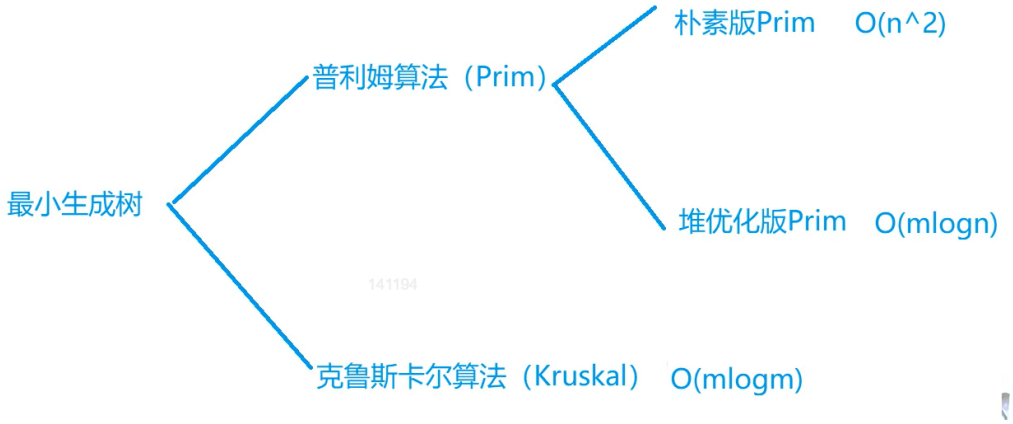
```
package searchandgraph.shortestroad;

import java.io.*;

/**
 * @author LBS59
 * @description Floyd算法：解决多源最短路问题
 */
public class Floyd {
    private static final int N = 210, INF = 0x3f3f3f3f;

    private static int[][] dist;

    static {
        dist = new int[N][N];
    }

    public static void floyd(int n) {
        for (int k = 1; k <= n; k++) {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]), k = Integer.parseInt(s[2]);

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == j) {
                    dist[i][j] = 0;
                } else {
                    dist[i][j] = INF;
                }
            }
        }

        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            dist[a][b] = Math.min(dist[a][b], c);
        }

        floyd(n);

        while (k-- > 0) {
            String[] q = in.readLine().split(" ");
```

```
            int i = Integer.parseInt(q[0]), j = Integer.parseInt(q[1]);
            if (dist[i][j] > INF / 2) {
                out.write("impossible\n");
            } else {
                out.write(dist[i][j] + "\n");
            }
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
N, INF = 210, int(0x3f3f3f3f)
dist = [[0] * N for _ in range(N)]


def floyd(cnt: int) -> None:
    for t in range(1, cnt + 1):
        for z in range(1, cnt + 1):
            for x in range(1, cnt + 1):
                dist[z][x] = min(dist[z][x], dist[z][t] + dist[t][x])


if __name__ == '__main__':
    s = input().split()
    n, m, k = int(s[0]), int(s[1]), int(s[2])
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if i == j:
                dist[i][j] = 0
            else:
                dist[i][j] = INF
    for _ in range(m):
        r = input().split()
        a, b, c = int(r[0]), int(r[1]), int(r[2])
        dist[a][b] = min(dist[a][b], c)

    floyd(n)

    for _ in range(k):
        q = input().split()
        i, j = int(q[0]), int(q[1])
        if dist[i][j] > INF // 2:
            print("impossible")
        else:
            print(dist[i][j])
```

3.6 最小生成树



3.7.1 Prim算法

核心思想：类似Dijkstra算法：

- 初始化所有的点距离到存储集合的距离为∞
- 创建一个集合，集合中保存已经确定距离的点
- 找到未在集合中的距离起点最近的一个点，使用其更新其余所有未在集合中点到集合的距离，注意这里是更新到集合的距离，而不是到某一个节点的距离。
- 重复上述过程，直到所有点的距离都确定下来。

实现：C++：

```
#include<iostream>
#include<cstring>
```

```cpp

using namespace std;

const int N = 510, INF = 0x3f3f3f3f;

int n, m;
int g[N][N];
int dist[N];
bool st[N];

int prim()
{
    // 初始化所有距离为正无穷
    memset(dist, 0x3f, sizeof dist);
    int res = 0;
    // 遍历n次
    for (int i = 0; i < n; i++)
    {
        // 找到连通块之外的距离连通块最近的节点
        int t = -1;
        for (int j = 1; j <= n; j++)
        {
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
            {
                t = j;
            }
        }
        // 如果找到的节点不连通，直接返回正无穷，最小生成树不存在
        if (i && dist[t] == INF)
        {
            return INF;
        }
        // 不是第一个节点，就将结果加入
        if (i)
        {
            res += dist[t];
        }
        // 用当前点更新其他所有点距连通块的最小距离
        for (int j = 1; j <= n; j++)
        {
            dist[j] = min(dist[j], g[t][j]);
        }
        // 将当前点存入连通块中
        st[t] = true;
    }
    return res;
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(g, 0x3f, sizeof g);
    while (m--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        // 存在自环和重边，这里只需要保留边权最小的那条边即可
        g[a][b] = g[b][a] = min(g[a][b], c);
    }
    int t = prim();

    if (t == INF)
    {
        puts("impossible");
    }
    else
    {
        printf("%d\n", t);
    }

    return 0;
}
```

实现：Java：

```java
package searchandgraph.mingeneratetree;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description Prim算法求最小生成树
 */
```

```java
public class Prim {
    private static final int N = 510, INF = 0x3f3f3f3f;

    /**
     * 稠密图使用邻接矩阵来存储
     */
    private static int[][] g;
    /**
     * dist存储距离连通块的最短距离
     */
    private static int[] dist;
    /**
     * 判重
     */
    private static boolean[] vis;

    static {
        g = new int[N][N];
        for (int i = 0; i < N; i++) {
            Arrays.fill(g[i], INF);
        }
        dist = new int[N];
        Arrays.fill(dist, INF);
        vis = new boolean[N];
    }

    public static int prim(int n) {
        int res = 0;
        for (int i = 0; i < n; i++) {
            int t = -1;
            for (int j = 1; j <= n; j++) {
                if (!vis[j] && (t == -1 || dist[t] > dist[j])) {
                    t = j;
                }
            }
            if (i > 0 && dist[t] == INF) {
                return INF;
            }
            if (i > 0) {
                res += dist[t];
            }
            for (int j = 1; j <= n; j++) {
                dist[j] = Math.min(dist[j], g[t][j]);
            }
            vis[t] = true;
        }
        return res;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), c = Integer.parseInt(row[2]);
            g[a][b] = g[b][a] = Math.min(g[a][b], c);
        }
        int t = prim(n);
        if (t == INF) {
            System.out.println("impossible");
        } else {
            System.out.println(t);
        }

        in.close();
    }
}
```

实现：Python：

```python
N, INF = 510, int(0x3f3f3f3f)

g, dist, st = [[INF] * N for _ in range(N)], [INF] * N, [False] * N


def prim(x: int) -> int:
    res = 0
    for i in range(x):
        t = -1
        for j in range(1, x + 1):
            if not st[j] and (t == -1 or dist[t] > dist[j]):
                t = j
        if i > 0 and dist[t] == INF:
            return INF
```

```
        if i > 0:
            res += dist[t]
        for j in range(1, x + 1):
            dist[j] = min(dist[j], g[t][j])
        st[t] = True
    return res


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b, c = int(row[0]), int(row[1]), int(row[2])
        g[a][b] = g[b][a] = min(g[a][b], c)
    ans = prim(n)
    if ans == INF:
        print("impossible")
    else:
        print(ans)
```

3.7.2 Kruskal算法

> **核心思想：** **并查集应用**：
>
> - 将所有边按权重由小到大排序；
> - 枚举每一条边a，b，权重c；
>   - 如果a，b不连通，就将a，b这条边加入集合中

> **实现：C++：**

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 200010;

int n, m;
int p[N];

struct Edge {
    int a, b, w;

    bool operator < (const Edge &W) const
    {
        return w < W.w;
    }
}edges[N];


int find(int x)
{
    if (p[x] != x)
    {
        p[x] = find(p[x]);
    }
    return p[x];
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; i++)
    {
        int a, b, w;
        scanf("%d%d%d", &a, &b, &w);
        edges[i] = {a, b, w};
    }
    sort(edges, edges + m);
    for (int i = 1; i <= n; i++)
    {
        p[i] = i;
    }
    int res = 0, cnt = 0;
    for (int i = 0; i < m; i++)
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);
        if (a != b)
        {
            p[a] = b;
            res += w;
            cnt++;
        }
    }
```

```
    if (cnt < n - 1)
    {
        puts("impossible");
    }
    else
    {
        printf("%d\n", res);
    }
}
```

实现：Java：

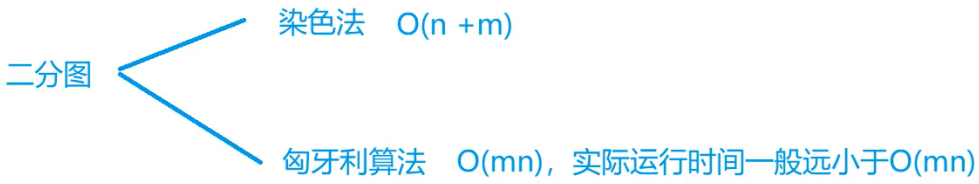
```java
package searchandgraph.mingeneratetree;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Comparator;

/**
 * @author LBS59
 * @description Kruskal算法求最小生成树
 */
public class Kruskal {

    /**
     * 边的定义
     */
    private static class Edge implements Comparable<Edge> {
        /**
         * 起点
         */
        int fr;
        /**
         * 终点
         */
        int to;
        /**
         * 边权
         */
        int weight;

        public Edge(int fr, int to, int weight) {
            this.fr = fr;
            this.to = to;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge o) {
            return Integer.compare(weight, o.weight);
        }
    }
    private static final int N = 200020;

    private static int[] p;
    private static Edge[] edges;

    static {
        p = new int[N];
        edges = new Edge[N];
    }

    public static int find(int x) {
        return p[x] == x ? p[x] : find(p[x]);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        for (int i = 0; i < m; i++) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), w = Integer.parseInt(row[2]);
            edges[i] = new Edge(a, b, w);
        }
        Arrays.sort(edges, 0, m);
        for (int i = 1; i <= n; i++) {
            p[i] = i;
        }
        int res = 0, cnt = 0;
        for (int i = 0; i < m; i++) {
            int a = edges[i].fr, b = edges[i].to, w = edges[i].weight;
            a = find(a);
```

```
            b = find(b);
            if (a != b) {
                p[a] = b;
                res += w;
                cnt++;
            }
        }
        if (cnt < n - 1) {
            System.out.println("impossible");
        } else {
            System.out.println(res);
        }

        in.close();
    }
}
```

实现：Python：

```
N, INF = 200010, 0x3f3f3f3f


class Edge:
    def __init__(self, fr: int, to: int, weight: int):
        self.fr = fr
        self.to = to
        self.weight = weight


def find(x: int) -> int:
    if p[x] != x:
        return find(p[x])
    return p[x]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    p = [0] * (n + 1)
    edges = [None] * m
    for i in range(m):
        row = input().split()
        a, b, w = int(row[0]), int(row[1]), int(row[2])
        edges[i] = Edge(a, b, w)
    edges.sort(key=lambda e: e.weight)
    for i in range(1, n + 1):
        p[i] = i
    res, cnt = 0, 0
    for i in range(m):
        a, b, w = edges[i].fr, edges[i].to, edges[i].weight
        a = find(a)
        b = find(b)
        if a != b:
            p[a] = b
            res += w
            cnt += 1
    if cnt < n - 1:
        print("impossible")
    else:
        print(res)
```

3.7 二分图



> **神马是二分图：二分图就是可以找到两个集合，使得图中的所有节点位于两个集合中，所有的边处于集合之间。**
>
> **一个图是二分图，当且仅当图中不存在奇数环**

3.7.1 染色法

> **核心思想**：**判断一个图是否为二分图：**
>
> 使用两种颜色对整张图进行染色，一条边的两边需要染成不同的颜色(符合二分图的定义)，如果在染色的过程中出现了矛盾，则证明此图不是二分图

实现：C++：

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 100010, M = 200010;

int n, m;
int h[N], e[M], ne[M], idx;
int color[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!color[j])
        {
            if (!dfs(j, 3 - c))
            {
                return false;
            }
        }
        else if (color[j] == c)
        {
            return false;
        }
    }
    return true;
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m--)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b), add(b, a);
    }
    bool flag = true;
    for (int i = 1; i <= n; i++)
    {
        if (!color[i])
        {
            if (!dfs(i, 1))
            {
                flag = false;
                break;
            }
        }
    }
    if(flag)
    {
        puts("Yes");
    }
    else
    {
        puts("No");
    }
}
```

实现：Java：

```java
package searchandgraph.binarygraph;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 染色法判断二分图
 */
public class Paint {
    private static final int N = 100010, M = 200010;

    private static int[] h, e, ne;
```

```java
    private static int idx;
    private static int[] color;

    static {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[M];
        ne = new int[M];
        idx = 0;
        color = new int[N];
    }

    public static void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    public static boolean dfs(int u, int c) {
        color[u] = c;
        for (int i = h[u]; i != -1; i = ne[i]) {
            int j = e[i];
            if (color[j] == 0) {
                if (!dfs(j, 3 - c)) {
                    return false;
                }
            } else if (color[j] == c) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]);
            add(a, b);
            add(b, a);
        }
        boolean flag = true;
        for (int i = 1; i <= n; i++) {
            if (color[i] == 0) {
                if (!dfs(i, 1)) {
                    flag = false;
                    break;
                }
            }
        }
        if (flag) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }

        in.close();
    }
}
```

> **实现：Python：** 有BUG，小数据过得去大数据直接RE。

```python
N, M = 100010, 200010
h, e, ne, idx = [-1] * N, [0] * M, [0] * M, 0
color = [0] * N


def add(fr: int, to: int) -> None:
    global idx
    e[idx] = to
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def dfs(u: int, c: int) -> bool:
    color[u] = c
    t = h[u]
    while t != -1:
        j = e[t]
        if not color[j]:
            if not dfs(j, 3 - c):
                return False
```

```
        elif color[j] == c:
            return False
        t = ne[t]
    return True


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for _ in range(m):
        row = input().split()
        a, b = int(row[0]), int(row[1])
        add(a, b)
        add(b, a)
    flag = True
    for i in range(1, n + 1):
        if not color[i]:
            if not dfs(i, 1):
                flag = False
                break
    if flag:
        print("Yes")
    else:
        print("No")
```

3.7.2 匈牙利算法：寻找二分图的最大匹配

> **核心思想：** 最大匹配指的是：在一个二分图中，我们定义左边的集合存在`n1`个点，右边的集合存在`n2`个点，所有的边存在于两个集合之间，即所给图是一个二分图，最大匹配找的是存在一系列边，保证两侧所连接节点不重复，即每一个边两边连接唯一节点，每一个节点最多只有一个入度和出度，不存在一个点有大于一个入度和出度，此时所有边的数量被称为最大匹配数量

> **实现：C++：**

```
#include<iostream>
#include<cstring>

using namespace std;

const int N = 510, M = 100010;

int n1, n2, m;
int h[N], e[M], ne[M], idx;
int match[N];
bool st[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }
    return false;
}

int main()
{
    scanf("%d%d%d", &n1, &n2, &m);
    while (m--)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b);
    }
    int res = 0;
    for (int i = 1; i <= n1; i++)
    {
        memset(st, false, sizeof st);
        if (find(i))
        {
            res++;
        }
    }
```

```
    printf("%d\n", res);

    return 0;
}
```

实现：Java：

```java
package searchandgraph.binarygraph;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 匈牙利算法寻找二分图最大匹配
 */
public class XYL {
    private static final int N = 510, M = 100010;

    private static int[] h, e, ne;
    private static int idx;
    private static int[] match;
    private static boolean[] vis;

    static {
        h = new int[N];
        Arrays.fill(h, -1);
        e = new int[M];
        ne = new int[M];
        idx = 0;
        match = new int[N];
        vis = new boolean[N];
    }

    public static void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    public static boolean find(int x) {
        for (int i = h[x]; i != -1; i = ne[i]) {
            int j = e[i];
            if (!vis[j]) {
                vis[j] = true;
                if (match[j] == 0 || find(match[j])) {
                    return true;
                }
            }
        }
        return false;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n1 = Integer.parseInt(s[0]), n2 = Integer.parseInt(s[1]), m = Integer.parseInt(s[2]);
        while (m-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]);
            add(a, b);
        }
        int res = 0;
        for (int i = 1; i <= n1; i++) {
            Arrays.fill(vis, false);
            if (find(i)) {
                res++;
            }
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python

```python
N, M = 510, 100010
h, e, ne, idx, match, st = [-1] * N, [0] * M, [0] * M, 0, [0] * N, [False] * N


def add(fr: int, to: int) -> None:
```

```
    global idx
    e[idx] = to
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def find(x: int) -> bool:
    t = h[x]
    while t != -1:
        j = e[t]
        if not st[j]:
            st[j] = True
            if not match[j] or find(match[j]):
                match[j] = x
                return True
        t = ne[t]
    return False


if __name__ == '__main__':
    s = input().split()
    n1, n2, m = int(s[0]), int(s[1]), int(s[2])
    for _ in range(m):
        row = input().split()
        a, b = int(row[0]), int(row[1])
        add(a, b)
    res = 0
    for i in range(1, n1 + 1):
        st = [False] * N
        if find(i):
            res += 1
    print(res)
```

4. 数学知识

4.1 质数

> **定义：**在大于1的整数中，如果只包含1和本身两个约数，就称其为质数，也叫素数。

4.1.1 质数的判定-试除法（$O(sqrt(n))$）

> **核心思想：**使用暴力的方式判断质数就是判断当前数`n`是否能整除`2~n - 1`之间的数，如果能，则证明`n`有除了1和本身的其他约数，就不是质数，这个时间复杂度为$O(N)$，n为要判定的数，如果n特别大的话，就会超时，而试除法的思想是：假设`x`能被`n`整除，则`n/x`也能被`n`整除，这是`n`的一对约数，所以在枚举除数时，只需要枚举到`n/x`即可。

> **实现：C++：**

```
#include<iostream>

using namespace std;

int n;

bool isPrime(int n)
{
    if (n < 2)
    {
        return false;
    }
    for (int i = 2; i <= n / i; i++)
    {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

int main()
{
    scanf("%d", &n);
    while (n--)
    {
        int x;
        scanf("%d", &x);
        if (isPrime(x))
        {
            puts("Yes");
        }
        else
        {
            puts("No");
        }
    }

```

```
    return 0;
}
```

实现：Java：

```java
package mathematic.prime.isprime;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 试除法判断质数
 */
public class CheckPrimes {
    public static boolean isPrime(int n) {
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= n / i; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            int x = sc.nextInt();
            if (isPrime(x)) {
                System.out.println("Yes");
            } else {
                System.out.println("No");
            }
        }
    }
}
```

实现：Python：

```python
import math


def isPrime(c: int) -> bool:
    if c < 2:
        return False
    sq = int(math.sqrt(c))
    for i in range(2, sq + 1):
        if c % i == 0:
            return False
    return True


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        x = int(input())
        if isPrime(x):
            print("Yes")
        else:
            print("No")
```

4.1.2 分解质因数-试除法（$O(\log n) - O(sqrt(n))$）

> **核心思想：**一个数`n`中最多只包含一个大于等于$\sqrt{n}$的质因子，我们就可以只枚举$\sqrt{n}$以内的所有质因子，剩余的一个质因子单独处理。

实现：C++：

```cpp
#include<iostream>

using namespace std;

void divide(int n)
{
    for (int i = 2; i <= n / i; i++)
    {
        // i一定是质数
        if (n % i == 0)
        {
            int s = 0;
            while (n % i == 0)
            {
```

```
                n /= i;
                s++;
            }
            printf("%d %d\n", i, s);
        }
    }
    if (n > 1)
    {
        printf("%d %d\n", n, 1);
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int x;
        scanf("%d", &x);
        divide(x);
        puts("");
    }

    return 0;
}
```

实现：Java：

```
package mathematic.prime.isprime;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 试除法分解质因子，以及质因子的指数
 */
public class DivisePrimeFactor {
    public static void divide(int n) {
        for (int i = 2; i <= n / i; i++) {
            // 当前的i一定是一个质数
            if (n % i == 0) {
                // 求质数的指数
                int s = 0;
                while (n % i == 0) {
                    n /= i;
                    s++;
                }
                System.out.println(i + " " + s);
            }
        }
        if (n > 1) {
            System.out.println(n + " " + 1);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            int x = sc.nextInt();
            divide(x);
            System.out.println();
        }
    }
}
```

实现：Python：

```
import math


def divide(c: int) -> None:
    sq = int(math.sqrt(c))
    for i in range(2, sq + 1):
        if c % i == 0:
            s = 0
            while c % i == 0:
                c //= i
                s += 1
            print(str(i) + " " + str(s))
    if c > 1:
        print(str(c) + " " + str(1))


```

```python
if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        x = int(input())
        divide(x)
        print()
```

4.1.3 筛质数

埃氏筛法：

> 给定一个正整数`n`，求出`n`以内的所有质数有多少个
>
> **核心思想：** 枚举`2~n`的所有数，拿到当前最小的质数`x`保存，同时将`n`以内所有`x`的倍数(不含`x`)全部过滤掉，每次都这样操作，只要遍历完所有的数
>
> **时间复杂度：** $O(nlog(log\,n))$

实现：C++：

```cpp
#include<iostream>

using namespace std;

const int N = 1000010;

int primes[N], cnt;
bool st[N];

void get_primes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!st[i])
        {
            primes[cnt++] = i;
            for (int j = i + i; j <= n; j += i)
            {
                st[j] = true;
            }
        }
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    get_primes(n);
    printf("%d\n", cnt);

    return 0;
}
```

实现：Java：

```java
package mathematic.prime;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 筛质法
 */
public class GetPrimes {
    private static final int N = 1000010;
    /**
     * 保存质数
     */
    private static int[] primes;
    /**
     * 保存质数个数
     */
    private static int cnt;
    /**
     * 筛掉合数
     */
    private static boolean[] vis;

    static {
        primes = new int[N];
        cnt = 0;
        vis = new boolean[N];
    }

    public static void getPrimes(int n) {
```

```
        for (int i = 2; i <= n; i++) {
            if (!vis[i]) {
                primes[cnt++] = i;
                for (int j = i + i; j <= n; j += i) {
                    vis[j] = true;
                }
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        getPrimes(n);
        System.out.println(cnt);
    }
}
```

> **实现：Python：**

```
N = 1000010
primes, cnt, st = [0] * N, 0, [False] * N


def get_primes(x: int) -> None:
    global cnt
    for i in range(2, x + 1):
        if not st[i]:
            primes[cnt] = i
            cnt += 1
            for j in range(i + i, x + 1, i):
                st[j] = True


if __name__ == '__main__':
    n = int(input())
    get_primes(n)
    print(cnt)
```

线性筛法

> **核心思想：**任何一个合数都存在一个最小质因子；每一个合数都会被其最小质因子筛掉；我们使用每一个合数的最小质因子筛掉该合数，每一个数只会被筛一次。
>
> **时间复杂度：**$O(n)$

> **实现：C++：**

```
#include<iostream>

using namespace std;

const int N = 1000010;

int primes[N], cnt;
bool st[N];

void get_primes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!st[i])
        {
            primes[cnt++] = i;
        }
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0)
            {
                break;
            }
        }
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    get_primes(n);
    printf("%d\n", cnt);

    return 0;
}
```

> **实现：Java：**

```java
package mathematic.prime.getprime;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 线性筛法
 */
public class GetPrimes2 {
    private static final int N = 1000010;

    private static int[] primes;
    private static int cnt;
    private static boolean[] st;

    static {
        primes = new int[N];
        cnt = 0;
        st = new boolean[N];
    }

    public static void getPrimes(int n) {
        for (int i = 2; i <= n; i++) {
            if (!st[i]) {
                primes[cnt++] = i;
            }
            for (int j = 0; primes[j] <= n / i; j++) {
                st[primes[j] * i] = true;
                if (i % primes[j] == 0) {
                    break;
                }
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        getPrimes(n);
        System.out.println(cnt);
    }
}

```

> **实现：Python：**

```python
N = 1000010
primes, cnt, st = [0] * N, 0, [False] * N


def get_primes(x: int) -> None:
    global cnt
    for i in range(2, x + 1):
        if not st[i]:
            primes[cnt] = i
            cnt += 1
        j = 0
        t = x // i
        while primes[j] <= t:
            st[primes[j] * i] = True
            if i % primes[j] == 0:
                break
            j += 1


if __name__ == '__main__':
    n = int(input())
    get_primes(n)
    print(cnt)
```

4.2 约数

4.2.1 试除法求一个数的所有约数

> **核心思想：**和试除法原理一样

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;
```

```cpp

vector<int> get_divisors(int n)
{
    vector<int> res;
    for (int i = 1; i <= n / i; i++)
    {
        if (n % i == 0)
        {
            res.push_back(i);
            if (i != n / i)
            {
                res.push_back(n / i);
            }
        }
    }
    sort(res.begin(), res.end());
    return res;
}

int main()
{
    int n;
    cin >> n;
    while (n--)
    {
        int x;
        cin >> x;
        auto res = get_divisors(x);
        for (auto t : res)
        {
            cout << t << ' ';
        }
        cout << endl;
    }
}
```

实现：Java：

```java
package mathematic.divisor;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 试除法求一个数的所有约数
 */
public class GetDivisors {
    public static List<Integer> getDivisors(int n) {
        List<Integer> res = new ArrayList<>();
        for (int i = 1; i <= n / i; i++) {
            if (n % i == 0) {
                res.add(i);
                if (i != n / i) {
                    res.add(n / i);
                }
            }
        }
        Collections.sort(res);
        return res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            int x = sc.nextInt();
            List<Integer> res = getDivisors(x);
            for (int t : res) {
                System.out.printf("%d ", t);
            }
            System.out.println();
        }
    }
}
```

实现：Python：

```python
import math
from typing import List


def get_divisors(c: int) -> List:
```

```
    res = []
    sq = int(math.sqrt(c))
    for j in range(1, sq + 1):
        if c % j == 0:
            res.append(j)
            if j != c // j:
                res.append(c // j)
    res.sort()
    return res


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        x = int(input())
        r = get_divisors(x)
        for i in r:
            print(i, end=' ')
        print()
```

4.2.2 约数个数

> **核心思想：算术基本定理**
>
> 任何一个正整数`n`，都可以写成下面的形式：
>
> $$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k} \tag{22}$$
>
> 则约数的个数为$(\alpha_1 + 1)(\alpha_2 + 1)(\alpha_3 + 1)\ldots(\alpha_k + 1)$

> **实现：C++：**

```
#include<iostream>
#include<unordered_map>

using namespace std;

typedef long long LL;

const int mod = 1e9 + 7;

int main()
{
    int n;
    cin >> n;
    unordered_map<int, int> primes;
    while (n--)
    {
        int x;
        cin >> x;
        for (int i = 2; i < x / i; i++)
        {
            while (x % i == 0)
            {
                x /= i;
                primes[i]++;
            }
        }
        if (x > 1)
        {
            primes[x]++;
        }
    }
    LL res = 1;
    for (auto prime : primes)
    {
        res = res * (prime.second + 1) % mod;
    }
    cout << res << endl;

    return 0;
}
```

> **实现：Java：**

```
package mathematic.divisor;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 求一个数所有约数的个数
 */
public class CountOfDivisors {
```

```java
    private static final int MOD = (int) 1e9 + 7;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Map<Integer, Integer> primes = new HashMap<>();
        while (n-- > 0) {
            int x = sc.nextInt();
            for (int i = 2; i <= x / i; i++) {
                while (x % i == 0) {
                    x /= i;
                    primes.put(i, primes.getOrDefault(i, 0) + 1);
                }
            }
            if (x > 1) {
                primes.put(x, primes.getOrDefault(x, 0) + 1);
            }
        }
        long res = 1;
        for (int p : primes.values()) {
            res = res * (p + 1) % MOD;
        }
        System.out.println(res);
    }
}
```

实现：Python：

```python
import math
from collections import defaultdict

MOD = int(1e9 + 7)

if __name__ == '__main__':
    n = int(input())
    primes = defaultdict(int)
    for _ in range(n):
        x = int(input())
        sq = int(math.sqrt(x))
        for i in range(2, sq + 1):
            while x % i == 0:
                x //= i
                primes[i] += 1
        if x > 1:
            primes[x] += 1
    res = 1
    for v in primes.values():
        res = res * (v + 1) % MOD
    print(res)
```

4.2.3 约数之和

核心思想：算术基本定理

任何一个正整数`n`，都可以写成下面的形式：

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k} \tag{23}$$

则约数的和为$(p_1^0 + p_1^1 + \ldots + p_1^{\alpha_1}) \ldots (p_k^0 + p_k^1 + \ldots + p_k^{\alpha_k})$

实现：C++：

```cpp
#include<iostream>
#include<unordered_map>

using namespace std;

typedef long long LL;

const int mod = 1e9 + 7;

int main()
{
    int n;
    cin >> n;

    unordered_map<int, int> primes;

    while (n--)
    {
        int x;
        cin >> x;
        for (int i = 2; i <= x / i; i++)
        {
            while (x % i == 0)
            {
```

```
                x /= i;
                primes[i]++;
            }
        }
        if (x > 1)
        {
            primes[x]++;
        }
    }
    LL res = 1;
    for (auto prime : primes)
    {
        int p = prime.first, a = prime.second;
        LL t = 1;
        while (a--)
        {
            t = (t * p + 1) % mod;
        }
        res = res * t % mod;
    }

    cout << res << endl;

    return 0;
}
```

实现：Java：

```
package mathematic.divisor;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 整数的约数之和
 */
public class SumOfDivisors {
    private static final int MOD = (int) 1e9 + 7;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        Map<Integer, Integer> primes = new HashMap<>();

        while (n-- > 0) {
            int x = sc.nextInt();
            for (int i = 2; i <= x / i; i++) {
                while (x % i == 0) {
                    x /= i;
                    primes.put(i, primes.getOrDefault(i, 0) + 1);
                }
            }
            if (x > 1) {
                primes.put(x, primes.getOrDefault(x, 0) + 1);
            }
        }
        long res = 1;
        for (Map.Entry<Integer, Integer> entry : primes.entrySet()) {
            int p = entry.getKey(), a = entry.getValue();
            long t = 1;
            while (a-- > 0) {
                t = (t * p + 1) % MOD;
            }
            res = res * t % MOD;
        }
        System.out.println(res);
    }
}
```

实现：Python：

```
import math
from collections import defaultdict

MOD = int(1e9 + 7)

if __name__ == '__main__':
    n = int(input())
    primes = defaultdict(int)
    for _ in range(n):
        x = int(input())
        sq = int(math.sqrt(x))
```

```
        for i in range(2, sq + 1):
            while x % i == 0:
                x //= i
                primes[i] += 1
        if x > 1:
            primes[x] += 1
    res = 1
    for p, a in primes.items():
        t = 1
        for _ in range(a):
            t = (t * p + 1) % MOD
        res = res * t % MOD
    print(res)
```

4.2.4 欧几里得算法(辗转相除法)

> **核心原理：** 假定一个数 c 可以被 a 整除，表示为$c \mid a$，并且 c 可以被 b 整除，$c \mid b$，则 c 可以被 a 和 b 的线性组合整除，$c \mid ax + by$，则有$gcd(a,b) = gcd(b, a \bmod b)$

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}

int main()
{
    int n;
    cin >> n;
    while (n--)
    {
        int a, b;
        cin >> a >> b;
        cout << gcd(a, b) << endl;
    }

    return 0;
}
```

> **实现：Java：**

```java
package mathematic.gcd;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 欧几里得算法求最大公约数
 */
public class GCD {
    public static int gcd(int a, int b) {
        return b > 0 ? gcd(b, a % b) : a;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            int a = sc.nextInt(), b = sc.nextInt();
            System.out.println(gcd(a, b));
        }
    }
}
```

> **实现：Python：**

```python
def gcd(x: int, y: int) -> int:
    return gcd(y, x % y) if y else x


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        r = input().split()
        a, b = int(r[0]), int(r[1])
        print(gcd(a, b))
```

4.3 欧拉函数

> **欧拉函数：**：$\psi(n)$表示在`1~n`之间所有和n互质的数的个数，互质简单理解就是$gcd(i,n)=1$，如$\psi(6)=2$
>
> **核心求法：**
>
> （1）假定一个数`n`分解质因子后$n=p_1^{\alpha_1}\cdot p_2^{\alpha_2}\cdot....\cdot p_k^{\alpha_k}$；
>
> （2）则n的欧拉函数$\psi(n)=n(1-\frac{1}{p_1})(1-\frac{1}{p_2})...(1-\frac{1}{p_k})$
>
> **核心思想：容斥原理：**
>
> （1）已知一个数`n`分解质因子后$n=p_1^{\alpha_1}\cdot p_2^{\alpha_2}\cdot....\cdot p_k^{\alpha_k}$；
>
> （2）从`1~n`中去掉$p_1,p_2,...p_k$的所有倍数；
>
> （3）加上所有$p_i\times p_j$的倍数；
>
> （4）减去所有$p_i\times p_j\times p_k$的倍数。
>
> ...

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

int main()
{
    int n;
    cin >> n;
    while (n--)
    {
        int a;
        cin >> a;
        int res = a;
        for (int i = 2; i <= a / i; i++)
        {
            if (a % i == 0)
            {
                res = res / i * (i - 1);
                while (a % i == 0)
                {
                    a /= i;
                }
            }
        }
        if (a > 1)
        {
            res = res / a * (a - 1);
        }
        cout << res << endl;
    }
    return 0;
}
```

> **实现：Java：**

```java
package mathematic.euler;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 容斥原理求欧拉函数
 */
public class Euler {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            int a = sc.nextInt();
            int res = a;
            for (int i = 2; i <= a / i; i++) {
                if (a % i == 0){
                    res = res / i * (i - 1);
                    while (a % i == 0) {
                        a /= i;
                    }
                }
            }
            if (a > 1) {
                res = res / a * (a - 1);
            }
            System.out.println(res);
        }
    }
```

```
}
```

实现：Python：

```python
import math


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        a = int(input())
        sq = int(math.sqrt(a))
        res = a
        for i in range(2, sq + 1):
            if a % i == 0:
                res = res // i * (i - 1)
                while a % i == 0:
                    a //= i
        if a > 1:
            res = res // a * (a - 1)
        print(res)
```

线性筛法求欧拉函数： 将时间复杂度压缩到$O(N)$

实现：C++：

```cpp
#include<iostream>
using namespace std;

typedef long long LL;

const int N = 100010;

int primes[N], cnt;
int phi[N];
bool st[N];

LL get_eulers(int n)
{
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        if (!st[i])
        {
            primes[cnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0)
            {
                phi[primes[j] * i] = phi[i] * primes[j];
                break;
            }
            phi[primes[j] * i] = phi[i] * (primes[j] - 1);
        }
    }
    LL res = 0;
    for (int i = 1; i <= n; i++)
    {
        res += phi[i];
    }
    return res;
}

int main()
{
    int n;
    cin >> n;
    cout << get_eulers(n) << endl;

    return 0;
}
```

实现：Java：

```java
package mathematic.euler;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 线性筛法求欧拉函数
 */
```

```java
public class GetEulers {
    private static final int N = 1000010;
    /**
     * 线性筛法求质数三件套
     */
    private static int[] primes;
    private static int cnt;
    private static boolean[] vis;
    /**
     * 每一个数的欧拉函数
     */
    private static int[] phi;

    static {
        primes = new int[N];
        cnt = 0;
        vis = new boolean[N];
        phi = new int[N];
    }

    public static long getEulers(int n) {
        phi[1] = 1;
        for (int i = 2; i <= n; i++) {
            if (!vis[i]) {
                primes[cnt++] = i;
                phi[i] = i - 1;
            }
            for (int j = 0; primes[j] <= n / i; j++) {
                vis[primes[j] * i] = true;
                if (i % primes[j] == 0) {
                    phi[primes[j] * i] = phi[i] * primes[j];
                    break;
                }
                phi[primes[j] * i] = phi[i] * (primes[j] - 1);
            }
        }
        long res = 0;
        for (int i = 1; i <= n; i++) {
            res += phi[i];
        }
        return res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        System.out.println(getEulers(n));
    }
}
```

实现：Python：

```python
N = 1000010
primes, cnt, st, phi = [0] * N, 0, [False] * N, [0] * N


def get_eulers(n: int) -> int:
    global cnt
    phi[1] = 1
    for i in range(2, n + 1):
        if not st[i]:
            primes[cnt] = i
            cnt += 1
            phi[i] = i - 1
        j = 0
        d = n // i
        while primes[j] <= d:
            st[primes[j] * i] = True
            if i % primes[j] == 0:
                phi[primes[j] * i] = phi[i] * primes[j]
                break
            phi[primes[j] * i] = phi[i] * (primes[j] - 1)
            j += 1
    res = 0
    for i in range(n + 1):
        res += phi[i]
    return res


if __name__ == '__main__':
    x = int(input())
    print(get_eulers(x))
```

欧拉定理：

- 若存在a与n互质，则有$a^{\psi(n)}\%\ n = 1$

> **费马定理：**
>
> - 在欧拉定理中如何`n`为质数，则有$a^{n-1}\%\ n=1$

4.4 快速幂

> **应用：**
>
> 可以快速(时间复杂度在$O(logk)$)的求出$a^k\ mod\ p$ 其中，$1\leq a,p,k\leq 10^9$的结果
>
> **核心思路：**反复平方法！
>
> - 先预处理出$logk$个值，分别为$a^{2^0}\ mod\ p,a^{2^1}\ mod\ p,\dots,a^{2^{logk}}\ mod\ p$
> - 将a^k拆分为若干项`2`的幂次之积，即$a^k=a^{2^{x_1}}\times a^{2^{x_2}}\times\dots\times a^{2^{x_k}}=a^{2^{x_1}+2^{x_2}+\dots+2^{x_k}}$
> - 这里只需要将k表示为二进制形式，找到对应的预处理结果相乘即可。

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

typedef long long LL;

int qmi(int a, int k, int p)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
        {
            res = (LL) res * a % p;
        }
        k >>= 1;
        a = (LL) a * a % p;
    }
    return res;
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int a, k, p;
        scanf("%d%d%d", &a, &k, &p);
        printf("%d\n", qmi(a, k, p));
    }

    return 0;
}
```

> **实现：C++：**

```java
import java.io.*;

/**
 * @author LBS59
 * @description 快速幂模板
 */
public class Main {
    public static int qMi(int a, int k, int p) {
        long res = 1;
        long t = a;
        while (k > 0) {
            if ((k & 1) > 0) {
                res = res * t % p;
            }
            k >>= 1;
            t = t * t % p;
        }
        return (int) res;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), k = Integer.parseInt(row[1]), p = Integer.parseInt(row[2]);
            out.write(qMi(a, k, p) + "\n");
        }

        out.flush();
```

```
        in.close();
        out.close();
    }
}
```

实现：Python：

```python
def quick_mi(a: int, k: int, p: int) -> int:
    res = 1
    while k:
        if k & 1:
            res = res * a % p
        k >>= 1
        a = a * a % p
    return res


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        row = input().split()
        a, k, p = int(row[0]), int(row[1]), int(row[2])
        print(quick_mi(a, k, p))
```

使用快速幂求逆元：什么是逆元(给定数`a`和`b`，找到一个数`x`，使得`b % (a * x) = 1`，这里`x`称为`a mod b`的逆元)

乘法逆元定义：若整数`b`，`m`互质，并且对于任意的整数`a`，如果满足$b|a$，则存在一个整数`x`，使得$a|b \equiv a \times x (mod\ m)$，则称`x`为`b`的模`m`乘法逆元，记作$b^{-1}(mod\ m)$。`b`存在乘法逆元的充要条件是`b`与模数`m`互质。当模数`m`为质数时，b_{m-2}即为`b`的乘法逆元。

实现：C++：

```cpp
#include<iostream>

using namespace std;

typedef long long LL;

int quick_mi(int a, int k, int p)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
        {
            res = (LL) res * a % p;
        }
        k >>= 1;
        a = (LL) a * a % p;
    }
    return res;
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int a, p;
        scanf("%d%d", &a, &p);
        int res = quick_mi(a, p - 2, p);
        if (a % p)
        {
            printf("%d\n", res);
        }
        else
        {
            puts("impossible");
        }
    }
    return 0;
}
```

实现：Java：

```java
package mathematic.quickmi;

import java.io.*;

/**
 * @author LBS59
 * @description 使用快速幂求逆元
 */
public class GetInverseElByQmi {
```

```java
    public static int quickMi(int a, int k, int p) {
        long res = 1;
        while (k > 0) {
            if ((k & 1) > 0) {
                res = res * (long) a % p;
            }
            k >>= 1;
            a = (int) (((long) a * a) % p);
        }
        return (int) res;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] q = in.readLine().split(" ");
            int a = Integer.parseInt(q[0]), p = Integer.parseInt(q[1]);
            int res = quickMi(a, p - 2, p);
            if (a % p > 0) {
                out.write(res + "\n");
            } else {
                out.write("impossible\n");
            }
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

> **实现：Python：**

```python
def quick_mi(a: int, k: int, p: int) -> int:
    res = 1
    while k:
        if k & 1:
            res = res * a % p
        k >>= 1
        a = a * a % p
    return res


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        row = input().split()
        a, p = int(row[0]), int(row[1])
        res = quick_mi(a, p - 2, p)
        if a % p:
            print(res)
        else:
            print("impossible")
```

4.5 扩展欧几里得算法

> **应用：**求解裴蜀定理中的`x`和`y`。
>
> **裴蜀定理：**对于任意一对正整数`a`和`b`，一定存在一组非零整数`x`，`y`，使得$ax + by = \gcd(a, b)$

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

void exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1, y = 0;
        return;
    }
    exgcd(b, a % b, y, x);
    y -= a / b * x;
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--)
    {
```

```
        int a, b, x, y;
        scanf("%d%d", &a, &b);

        exgcd(a, b, x, y);

        printf("%d %d\n", x, y);
    }

    return 0;
}
```

实现：Java：

```
package mathematic.extendgcd;

import java.io.*;

/**
 * @author LBS59
 * @description 扩展欧几里得算法求解裴蜀定理
 */
public class ExGcd {
    private static int x, y;

    public static void exGcd(int a, int b) {
        if (b == 0) {
            x = 1;
            y = 0;
            return;
        }
        exGcd(b, a % b);
        int temp = x;
        x = y;
        y = temp - a / b * y;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]);
            exGcd(a, b);
            out.write(x + " " + y + "\n");
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
x, y = 0, 0


def exgcd(a: int, b: int) -> None:
    global x, y
    if not b:
        x = 1
        y = 0
        return
    exgcd(b, a % b)
    temp = x
    x = y
    y = temp - a // b * y


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        row = input().split()
        a, b = int(row[0]), int(row[1])
        exgcd(a, b)

        print(str(x) + " " + str(y))
```

扩展欧几里得算法求解线性同余方程：

- 给定`a`，`b`，`m`，求出一个正整数`x`使得$a \times x \equiv b (mod\ m)$

实现：C++：

```cpp
#include<iostream>

using namespace std;

typedef long long LL;

int exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y , x);
    y -= a / b * x;

    return d;
}

int main()
{
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int a, b, m;
        scanf("%d%d%d", &a, &b, &m);
        int x, y;
        int d = exgcd(a, m, x, y);
        if (b % d)
        {
            puts("impossible");
        }
        else
        {
            printf("%d\n", (LL) x * (b / d) % m);
        }
    }

    return 0;
}
```

实现：Java：

```java
package mathematic.extendgcd;

import java.io.*;

/**
 * @author LBS59
 * @description 使用扩展的欧几里得算法求解线性同余方程
 */
public class GetLinCongByExGcd {
    private static int x, y;

    public static int exGcd(int a, int b) {
        if (b == 0) {
            x = 1;
            y = 0;
            return a;
        }
        int d = exGcd(b, a % b);
        int temp = x;
        x = y;
        y = temp - a / b * y;
        return d;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]), m = Integer.parseInt(row[2]);
            int d = exGcd(a, m);
            if (b % d > 0) {
                out.write("impossible\n");
            } else {
                out.write(((long) x * (b / d)) % m + "\n");
            }
        }

        out.flush();
        in.close();
```

```
        out.close();
    }
}
```

实现：Python：

```python
x, y = 0, 0


def exgcd(a: int, b: int) -> int:
    global x, y
    if not b:
        x = 1
        y = 0
        return a
    res = exgcd(b, a % b)
    temp = x
    x = y
    y = temp - a // b * y
    return res


if __name__ == '__main__':
    n = int(input())
    for _ in range(n):
        row = input().split()
        a, b, m = int(row[0]), int(row[1]), int(row[2])
        d = exgcd(a, m)
        if b % d:
            print("impossible")
        else:
            print(x * (b // d) % m)
```

4.6 中国剩余定理

核心思想：

- 给定k个两两互质的数$m_1, m_2, \ldots, m_k$，求解线性方程组的解，其中线性方程组为：

$$\begin{cases} x \equiv a_1 (mod\ m_1) \\ x \equiv a_2 (mod\ m_2) \\ \quad\quad\quad \ldots \\ x \equiv a_k (mod\ m_k) \end{cases} \tag{24}$$

- 其中使用M表示$M = m_1, m2, \ldots, m_k$，M_i表示$\frac{M}{m_i}$，M^{-1}表示M_i模m_i的逆元；
- 则`x`的通解表示为$x = a_1 \cdot M_1 \cdot M_1^{-1} + a_2 \cdot M_2 \cdot M_2^{-1} + \ldots + a_k \cdot M_k \cdot M_k^{-1}$

4.7 高斯消元法求解多元一次方程组

问题：求解方程组的解：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \quad\quad\quad \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases} \tag{25}$$

解的情况：

- 有无穷多组解；（0 = 0型 r < d）
- 有唯一解；（矩阵完美阶梯型 r = d）
- 无解。（0 = 非零型 r > d）

方法：矩阵的初等行/列变换

- 将某一行乘以一个非零的数；
- 交换某两行；
- 将某一行的若干倍加到零一行。

核心思想：

- 枚举每一列c_i；
  - 找到当前列中绝对值最大的一行；
  - 将这行换到最上方去；
  - 将当前行第一个系数变为1；
  - 将下面所有行的c_i列消为0。

实现：C++：

```cpp
#include<iostream>
#include<cmath>

using namespace std;

const int N = 110;
const double eps = 1e-6;
```

```

int n;
double a[N][N];

/*
res = 0表示有唯一解
res = 1表示有无穷多组解
res = 2表示无解
*/
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c++)
    {
        int t = r;
        // 找到当前列绝对值最大的系数
        for (int i = r; i < n; i++)
        {
            if (fabs(a[i][c]) > fabs(a[t][c]))
            {
                t = i;
            }
        }
        if (fabs(a[t][c]) < eps)
        {
            continue;
        }
        for (int i = c; i <= n; i++)
        {
            swap(a[t][i], a[r][i]);
        }
        for (int i = n; i >= c; i--)
        {
            a[r][i] /= a[r][c];
        }
        for (int i = r + 1; i < n; i++)
        {
            if (fabs(a[i][c]) > eps)
            {
                for (int j = n; j >= c; j--)
                {
                    a[i][j] -= a[r][j] * a[i][c];
                }
            }
        }
        r++;
    }
    if (r < n)
    {
        for (int i = r; i < n; i++)
        {
            if (fabs(a[i][n]) > eps)
            {
                return 2;
            }
        }
        return 1;
    }
    for (int i = n - 1; i >= 0; i--)
    {
        for (int j = i + 1; j < n; j++)
        {
            a[i][n] -= a[i][j] * a[j][n];
        }
    }
    return 0;
}

int main()
{
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n + 1; j++)
        {
            cin >> a[i][j];
        }
    }

    int t = gauss();

    if (t == 0)
    {
        for (int i = 0; i < n; i++)
        {
            if (fabs(a[i][n]) < eps)
            {
```

```
                printf("0.00");
            }
            else
            {
                printf("%.2lf\n", a[i][n]);
            }
        }
    }
    else if (t == 1)
    {
        puts("Infinite group solutions");
    }
    else
    {
        puts("No solution");
    }

    return 0;
}
```

实现：Java：

```
package mathematic.Gausseliminate;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 高斯消元法求解多元一次线性方程组
 */
public class GaussLiminate {
    private static final int N = 110;
    /**
     * 浮点数判0可能存在精度问题
     */
    private static final double EPS = 1e-6;
    /**
     * a表示方程组的系数矩阵
     */
    private static double[][] a;

    static {
        a = new double[N][N];
    }

    private static void swap(double[][] a, int x1, int y1, int x2, int y2) {
        double temp = a[x1][y1];
        a[x1][y1] = a[x2][y2];
        a[x2][y2] = temp;
    }

    public static int gauss(int n) {
        int c, r;
        for (c = 0, r = 0; c < n; c++) {
            int t = r;
            // 找到从r行开始，当前所有列中系数绝对值最大的行
            for (int i = r; i < n; i++) {
                if (Math.abs(a[i][c]) > Math.abs(a[t][c])) {
                    t = i;
                }
            }
            // 如果当前列不存在最大的系数，没有更新，则直接处理下一列
            if (Math.abs(a[t][c]) < EPS) {
                continue;
            }
            // 将当前行替换为当前的第r行
            for (int i = c; i <= n; i++) {
                swap(a, t, i, r, i);
            }
            // 将当前行的c列系数替换为1
            for (int i = n; i >= c; i--) {
                a[r][i] /= a[r][c];
            }
            // 将当前所有未处理过的c列替换为0
            for (int i = r + 1; i < n; i++) {
                if (Math.abs(a[i][c]) > EPS) {
                    for (int j = n; j >= c; j--) {
                        a[i][j] -= a[r][j] * a[i][c];
                    }
                }
            }
            r++;
        }
        if (r < n) {
            for (int i = r; i < n; i++) {
                if (Math.abs(a[i][n]) > EPS) {
```

```
                    return 2;
                }
            }
            return 1;
        }
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                a[i][n] -= a[i][j] * a[j][n];
            }
        }
        return 0;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n + 1; j++) {
                a[i][j] = sc.nextDouble();
            }
        }
        int res = gauss(n);
        if (res == 0) {
            for (int i = 0; i < n; i++) {
                if (Math.abs(a[i][n]) < EPS) {
                    System.out.println("0.00");
                } else {
                    System.out.printf("%.2f\n", a[i][n]);
                }
            }
        }
    }
}
```

实现：Python：

```
EPS = 1e-6


def gauss(n: int) -> int:
    # 初始化行数
    r = 0
    for c in range(n):
        # 从0开始遍历每一列
        t = r
        i = r
        while i < n:
            if abs(a[i][c]) > abs(a[t][c]):
                t = i
            i += 1
        # 如果当前列无需处理，直接跳过
        if abs(a[t][c]) < EPS:
            continue
        for i in range(c, n + 1):
            # 交换当前行和第r行
            a[t][i], a[r][i] = a[r][i], a[t][i]
        for i in range(n, c, -1):
            # 让首列系数变为1
            a[r][i] /= a[r][c]
        for i in range(r + 1, n):
            if abs(a[i][c]) > EPS:
                for j in range(n, c, -1):
                    a[i][j] -= a[r][j] * a[i][c]
        r += 1
    if r < n:
        for i in range(r, n):
            if abs(a[i][n]) > EPS:
                return 2
        return 1
    for i in range(n - 1, -1, -1):
        for j in range(i + 1, n):
            a[i][n] -= a[i][j] * a[j][n]
    return 0


if __name__ == '__main__':
    n = int(input())
    a = [0] * n
    for i in range(n):
        a[i] = list(map(float, (input().split())))
    res = gauss(n)
    if res == 0:
        for i in range(n):
            if abs(a[i][n]) < EPS:
                print("0.00")
            else:s
```

```
                print("%.2f" % a[i][n])
    elif res == 1:
        print("Infinite group solutions")
    else:
        print("No solution")
```

4.8 组合数

4.8.1 组合问题一

> 给定a，b，输出$C_a^b mod(10^9+7)$，其中a，b共100000组，a和b的范围为$[1,2000]$
>
> 如果查询的a，b对特别多，每一次计算a，b阶乘的时间复杂度就会非常大，容易超时
>
> **解决方案：**因为a，b的范围不是很大，可以预先处理出所有的C_a^b的值，后面的查询可以直接拿来使用$C_a^b = \frac{a!}{b!\times(a-b)!}$
>
> **核心思想：**递推表达式：
>
> $$C_a^b = C_{a-1}^b + C_{a-1}^{b-1} \tag{26}$$
>
> **时间复杂度：**$O(N^2)$

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 2010, mod = 1e9 + 7;

int c[N][N];

void init()
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            if (!j)
            {
                c[i][j] = 1;
            }
            else
            {
                c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
            }
        }
    }
}

int main()
{
    init();
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        printf("%d\n", c[a][b]);
    }

    return 0;
}
```

> **实现：Java：**

```java
package mathematic.combination.getcombination1;

import java.io.*;

/**
 * @author LBS59
 * @description 求组合数问题：给定a，b，求a对b的组合数，最后对1e9+7取模
 */
public class GetCombination {
    private static final int N = 2010, MOD = (int) 1e9 + 7;

    private static int[][] c;

    static {
        c = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j <= i; j++) {
```

```
                if (j == 0) {
                    c[i][j] = 1;
                } else {
                    c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD;
                }
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] query = in.readLine().split(" ");
            int a = Integer.parseInt(query[0]), b = Integer.parseInt(query[1]);
            out.write(c[a][b] + "\n");
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
N, MOD = 2010, int(1e9 + 7)
c = [[0] * N for _ in range(N)]


def init() -> None:
    for i in range(N):
        j = 0
        while j <= i:
            if not j:
                c[i][j] = 1
            else:
                c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD
            j += 1


if __name__ == '__main__':
    init()
    n = int(input())
    for _ in range(n):
        query = input().split()
        a, b = int(query[0]), int(query[1])
        print(c[a][b])
```

4.8.2 组合问题二

给定a，b，输出$C_a^b mod(10^9+7)$，其中a，b共10000组，a和b的范围为$[1,100000]$

核心思想：预处理。$C_a^b=\frac{a!}{(a-b)!\times b!}$，这里的除法计算取模不是很好计算，我们这里使用$fact[i]=i!\ mod\ M$表示i的阶乘取模后的值，使用$infact[i]=(i!])^{-1}\ mod\ M$表示i的阶乘的逆元，则有$C_a^b=\frac{a!}{(a-b)!\times b!}=fact[a]\times infact[b-a]\times infact[b]$。

时间复杂度：$O(NlogN)$

实现：C++：

```
#include<iostream>

using namespace std;

typedef long long LL;
const int N = 100010, mod = 1e9 + 7;

int fact[N], infact[N];

int qmi(int a, int k, int p)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
        {
            res = (LL) res * a % p;
        }
        a = (LL) a * a % p;
        k >>= 1;
    }
    return res;
}

```

```
int main()
{
    fact[0] = infact[0] = 1;
    for (int i = 1; i < N; i++)
    {
        fact[i] = (LL) fact[i - 1] * i % mod;
        infact[i] = (LL) infact[i - 1] * qmi(i, mod - 2, mod) % mod;
    }
    int n;
    scanf("%d", &n);
    while (n--)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        printf("%d\n", (LL) fact[a] * infact[b] % mod * infact[a - b] % mod);
    }

    return 0;
}
```

实现：Java：

```
package mathematic.combination.getcombination2;

import java.io.*;

/**
 * @author LBS59
 * @description 求组合数问题
 */
public class GetCombination {
    private static final int N = 100010, MOD = (int) 1e9 + 7;

    /**
     * fact数组存放阶乘结果，infact存放逆元结果
     */
    private static int[] fact;
    private static int[] infact;

    static {
        fact = new int[N];
        infact = new int[N];
        fact[0] = infact[0] = 1;
        for (int i = 1; i < N; i++) {
            fact[i] = (int) (((long) fact[i - 1] * i) % MOD);
            infact[i] = (int) (((long) infact[i - 1] * quickPower(i, MOD - 2, MOD)) % MOD);
        }
    }

    private static int quickPower(int a, int k, int p) {
        long res = 1;
        long t = a;
        while (k > 0) {
            if ((k & 1) != 0) {
                res = res * t % MOD;
            }
            t = t * t % MOD;
            k >>= 1;
        }
        return (int) res;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.parseInt(in.readLine());
        while (n-- > 0) {
            String[] query = in.readLine().split(" ");
            int a = Integer.parseInt(query[0]), b = Integer.parseInt(query[1]);
            out.write((int) ((((long) fact[a] * infact[a - b] % MOD) * infact[b]) % MOD) + "\n");
        }

        out.flush();
        in.close();
        out.close();
    }
}
```

实现：Python：

```
N, MOD = 100010, int(1e9 + 7)
fact, infact = [0] * N, [0] * N


```

```python
def init() -> None:
    fact[0] = infact[0] = 1
    for i in range(1, N):
        fact[i] = fact[i - 1] * i % MOD
        infact[i] = infact[i - 1] * quick_power(i, MOD - 2, MOD) % MOD


def quick_power(a: int, k: int, p: int) -> int:
    res = 1
    while k:
        if k & 1:
            res = res * a % p
        a = a * a % p
        k >>= 1
    return res


if __name__ == '__main__':
    init()
    n = int(input())
    for _ in range(n):
        query = input().split()
        a, b = int(query[0]), int(query[1])
        print(fact[a] * infact[a - b] * infact[b] % MOD)
```

4.8.3 组合问题三

> 给定a，b，输出$C_a^b \ mod \ p$，这里a，b的组数只有两位数级别，但是a，b的范围为$[1, 10^{18}]$，p的范围为$[1, 10^5]$。
>
> **解决方案：** 卢卡斯定理(Lucas 定理)
>
> $$C_a^b \equiv C_{a \ mod \ p}^{b \ mod \ p} \times C_{a/p}^{b/p} (mod \ p)$$
>
> **时间复杂度：** $O(plogNlogp)$

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

typedef long long LL;

int quick_power(int a, int k, int p)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
        {
            res = (LL) res * a % p;
        }
        a = (LL) a * a % p;
        k >>= 1;
    }
    return res;
}

int C(int a, int b, int p)
{
    int res = 1;
    for (int i = 1, j = a; i <= b; i++, j--)
    {
        res = (LL) res * j % p;
        res = (LL) res * quick_power(i, p - 2) % p;
    }
    return res;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p)
    {
        return C(a, b);
    }
    return (LL) C(a % p, b % p) * lucas(a / p, b / p) % p;
}

int main()
{
    int n;
    cin >> n;
    while (n--)
    {
        LL a, b;
        int p;
        cin >> a >> b >> p;
```

```
        cout << lucas(a, b, p) << endl;
    }
    return 0;
}
```

实现：Java：

```java
package mathematic.combination.getcombination3;

import java.util.Scanner;

/**
 * @author LBS59
 */
public class GetCombination {
    private static int quickPower(int a, int k, int p) {
        long res = 1;
        long t = a;
        while (k > 0) {
            if ((k & 1) != 0) {
                res = res * t % p;
            }
            t = t * t % p;
            k >>= 1;
        }
        return (int) res;
    }

    private static int combination(int a, int b, int p) {
        int res = 1;
        for (int i = 1, j = a; i <= b; i++, j--) {
            res = (int) ((long) res * j % p);
            res = (int) ((long) res * quickPower(i, p - 2, p) % p);
        }
        return res;
    }

    private static long lucas(long a, long b, int p) {
        if (a < p && b < p) {
            return combination((int) a, (int) b, p);
        }
        return combination((int) (a % p), (int) (b % p), p) * lucas(a / p, b / p, p) % p;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while (n-- > 0) {
            long a = sc.nextLong(), b = sc.nextLong();
            int p = sc.nextInt();
            System.out.println(lucas(a, b, p));
        }
    }
}
```

实现：Python：

```python
def quick_power(a: int, k: int, p: int) -> int:
    res = 1
    while k:
        if k & 1:
            res = res * a % p
        a = a * a % p
        k >>= 1
    return res


def combination(a: int, b: int, p: int) -> int:
    res = 1
    j = a
    for i in range(1, b + 1):
        res = res * j % p
        res = res * quick_power(i, p - 2, p) % p
        j -= 1
    return res


def lucas(a: int, b: int, p: int) -> int:
    if a < p and b < p:
        return combination(a, b, p)
    return combination(a % p, b % p, p) * lucas(a // p, b // p, p) % p


if __name__ == '__main__':
    n = int(input())
```

```
    for _ in range(n):
        query = input().split()
        a, b, p = int(query[0]), int(query[1]), int(query[2])
        print(lucas(a, b, p))
```

4.8.4 组合问题四

> 给定a，b，输出C_a^b，这里需要直接计算出结果，而不是对某一个数取余，其中a，b的范围为$[1, 5000]$
>
> **解决方案：仅仅对C++**对C_a^b分解质因子，表示为$C_a^b = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}$，然后实现一个高精度乘法即可;
>
> **求指数：**$C_a^b = \frac{a!}{b! \times (a-b)!}$，这里求$a!$中每一个p的不同指数采取$a! = \lfloor \frac{a}{p} \rfloor + \lfloor \frac{a}{p^2} \rfloor + \ldots + \lfloor \frac{a}{p^k} \rfloor$，这里的$p^k$是小于等于$a!$的p的最高次

> **实现：C++：**

```
#include<iostream>
#include<vector>

using namespace std;

const int N = 5010;

int primes[N], cnt, sum[N];
bool st[N];

void get_primes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!st[i])
        {
            primes[cnt++] = i;
        }
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] *i] = true;
            if (i * primes[j] == 0)
            {
                break;
            }
        }
    }
}

int get(int n, int p)
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}

vector<int> mul(vector<int> a, int b)
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i++)
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }
    while(t)
    {
        c.push_back(t % 10);
        t /= 10;
    }
    return c;
}

int main()
{
    int a, b;
    cin >> a >> b;
    get_primes(a);
    for (int i = 0; i < cnt; i++)
    {
        int p = primes[i];
        sum[i] = get(a, p) - get(b, p) - get(a - b, p);
    }
    vector<int> res;
    res.push_back(1);
    for (int i = 0; i < cnt; i++)
```

```
    {
        for (int j = 0; j < sum[i]; j++)
        {
            res = mul(res, primes[i]);
        }
    }
    for (int i = res.size() - 1; i >= 0; i--)
    {
        printf("%d", res[i]);
    }
    puts("");

    return 0;
}
```

实现：Java：

```
package mathematic.combination.getcombination4;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 求组合数
 */
public class GetCombination {
    private static final int N = 5010;

    private static int[] primes, sum;
    private static boolean[] vis;
    private static int cnt;

    static {
        primes = new int[N];
        sum = new int[N];
        cnt = 0;
        vis = new boolean[N];
    }

    /**
     * 线性筛质数
     * @param n 待筛
     */
    private static void getPrimes(int n) {
        for (int i = 2; i <= n; i++) {
            if (!vis[i]) {
                primes[cnt++] = i;
            }
            for (int j = 0; primes[j] <= n / i; j++) {
                vis[primes[j] * i] = true;
                if (i * primes[j] == 0) {
                    break;
                }
            }
        }
    }

    /**
     * 对p的各个指数次数算整数倍数
     * @param n 1
     * @param p 2
     * @return 3
     */
    private static int get(int n, int p) {
        int res = 0;
        while (n > 0) {
            res += n / p;
            n /= p;
        }
        return res;
    }

    private static List<Integer> mul(List<Integer> a, int b) {
        List<Integer> res = new ArrayList<>();
        int t = 0;
        for (int integer : a) {
            t += integer * b;
            res.add(t % 10);
            t /= 10;
        }
        while (t > 0) {
            res.add(t % 10);
            t /= 10;
        }
```

```
        return res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt(), b = sc.nextInt();
        getPrimes(a);

        for (int i = 0; i < cnt; i++) {
            int p = primes[i];
            sum[i] = get(a, p) - get(b, p) - get(a - b, p);
        }

        List<Integer> res = new ArrayList<>();
        res.add(1);

        for (int i = 0; i < cnt; i++) {
            for (int j = 0; j < sum[i]; j++) {
                res = mul(res, primes[i]);
            }
        }

        for (int i = res.size() - 1; i >= 0; i--) {
            System.out.printf("%d", res.get(i));
        }
    }
}
```

实现：Python：

```
from typing import List

N = 5010
primes, sum, cnt, st = [0] * N, [0] * N, 0, [False] * N


def get_primes(n: int) -> None:
    global cnt
    for i in range(2, n + 1):
        if not st[i]:
            primes[cnt] = i
            cnt += 1
        j = 0
        while primes[j] <= n // i:
            st[primes[j] * i] = True
            if primes[j] * i == 0:
                break
            j += 1


def get(n: int, p: int) -> int:
    ans = 0
    while n:
        ans += n // p
        n //= p
    return ans


def mul(a: List, b: int) -> List:
    ans = []
    t = 0
    for x in a:
        t += x * b
        ans.append(t % 10)
        t //= 10
    while t:
        ans.append(t % 10)
        t //= 10
    return ans


if __name__ == '__main__':
    query = input().split()
    a, b = int(query[0]), int(query[1])
    get_primes(a)

    for i in range(cnt):
        p = primes[i]
        sum[i] = get(a, p) - get(b, p) - get(a - b, p)

    res = [1]

    for i in range(cnt):
        for j in range(sum[i]):
            res = mul(res, primes[i])

```

```
    for i in range(len(res) - 1, -1, -1):
        print(res[i], end="")
```

4.9 卡特兰数

核心公式：

$$C_{2n}^{n} - C_{2n}^{n-1} = \frac{C_{2n}^{n}}{n+1} \tag{27}$$

实现：C++：

```cpp
#include<iostream>

using namespace std;

typedef long long LL;

const int mod = 1e9 + 7;

int qmi(int a, int k, int p)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
        {
            res = (LL) res * a % p;
        }
        a = (LL) a * a % p;
        k >>= 1;
    }
    return res;
}

int main()
{
    int n;
    cin >> n;

    int a = 2 * n, b = n;
    int res = 1;
    for (int i = a; i > a - b; i--)
    {
        res = (LL) res * i % mod;
    }
    for (int i = 1; i <= b; i++)
    {
        res = (LL) res * qmi(i, mod - 2, mod) % mod;
    }
    res = (LL) res * qmi(n + 1, mod - 2, mod) % mod;
    cout << res << endl;

    return 0;
}
```

实现：Java：

```java
package mathematic.cathelin;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 计算卡特兰数
 */
public class GetCathelin {
    private static final int MOD = (int) 1e9 + 7;

    private static int quickPower(int a, int k, int p) {
        long res = 1;
        long t = a;
        while (k > 0) {
            if ((k & 1) != 0) {
                res = res * t % p;
            }
            t = t * t % p;
            k >>= 1;
        }
        return (int) res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

```

```
        int res = 1;
        for (int i = n * 2; i > n; i--) {
            res = (int) ((long) res * i % MOD);
        }
        for (int i = 1; i <= n; i++) {
            res = (int) ((long) res * quickPower(i, MOD - 2, MOD) % MOD);
        }
        res = (int) ((long) res * quickPower(n + 1, MOD - 2, MOD) % MOD);
        System.out.println(res);
    }
}
```

实现：Python：

```
MOD = int(1e9 + 7)


def quick_power(a: int, k: int, p: int) -> int:
    ans = 1
    while k:
        if k & 1:
            ans = ans * a % p
        a = a * a % p
        k >>= 1
    return ans


if __name__ == '__main__':
    n = int(input())
    res = 1
    for i in range(n * 2, n, -1):
        res = res * i % MOD
    for i in range(1, n + 1):
        res = res * quick_power(i, MOD - 2, MOD) % MOD
    res = res * quick_power(n + 1, MOD - 2, MOD) % MOD
    print(res)
```

4.10 容斥原理

韦恩图引入容斥原理：存在三个两两相交的圆A，B，C，则三个圆覆盖区域的面积为

$$S_A \cup S_B \cup S_C = S_A + S_B + S_C - S_A \cap S_B - S_B \cap S_C - S_A \cap S_C + S_A \cap S_B \cap S_C$$

内容：上面的问题扩展到n个圆的情况，则有$S_{1 \cup 2 \cup \ldots \cup n} = 1 - 2 + 3 - 4 + 5 + \ldots + (-1)^{n-1} n$，其中数字表示所有数字表示数量的圆的组合，上面所有项的个数为$C_n^1 + C_n^2 + \ldots + C_n^n = 2^n - C_n^0 = 2^n - 1$

时间复杂度：$O(2^n)$

实现：C++：

```
#include<iostream>

using namespace std;

typedef long long LL;

const int N = 20;

int n, m;
int p[N];

int main()
{
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        cin >> p[i];
    }
    int res = 0;
    for (int i = 1; i < 1 << m; i++)
    {
        int t = 1, cnt = 0;
        for (int j = 0; j < m; j++)
        {
            if (i >> j & 1)
            {
                cnt++;
                if ((LL) t * p[j] > n)
                {
                    t = -1;
                    break;
                }
                t *= p[j];
            }
        }
        if (t != -1)
```

```
        {
            if (cnt % 2)
            {
                res += n / t;
            }
            else
            {
                res -= n / t;
            }
        }
    }
    cout << res << endl;

    return 0;
}
```

实现：Java：

```
package mathematic.inandexprinciple;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 容斥原理例题---计算1~n之中能被m个质数至少一个整除的个数有多少个
 */
public class InAndExPrinciple {
    private static final int N = 20;

    private static int[] p;

    static {
        p = new int[N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 0; i < m; i++) {
            p[i] = sc.nextInt();
        }
        int res = 0;
        // 二进制暴搜法
        for (int i = 1; i < 1 << m; i++) {
            // t存储当前的乘积，cnt存储当前方案中1的个数
            int t = 1, cnt = 0;
            for (int j = 0; j < m; j++) {
                // j位置被选中
                if (((i >> j) & 1) != 0) {
                    cnt++;
                    // 因为要求当前这个组合中所有质数的乘积在n中出现多少次，就是n是组合质数乘积的多少倍，如果乘到某一个质数超过了n，直接break掉
                    if ((long) t * p[j] > n) {
                        t = -1;
                        break;
                    }
                    // 否则就乘上这个位置的质数
                    t *= p[j];
                }
            }
            if (t != -1) {
                // 根据容斥原理，奇数组合做加法
                if (cnt % 2 != 0) {
                    res += n / t;
                // 偶数组合做减法
                } else {
                    res -= n / t;
                }
            }
        }
        System.out.println(res);
    }
}

```

实现：Python：

```
N = 20
p = [0] * N


if __name__ == '__main__':
    q = input().split()
    n, m = int(q[0]), int(q[1])
    row = input().split()
```

```python
    for i in range(m):
        p[i] = int(row[i])
    res = 0
    for i in range(1, 1 << m):
        t, cnt = 1, 0
        for j in range(m):
            if i >> j & 1:
                cnt += 1
                if t * p[j] > n:
                    t = -1
                    break
                t *= p[j]
        if t != -1:
            if cnt % 2:
                res += n // t
            else:
                res -= n // t
    print(res)
```

4.11 博弈论

> **公平组合游戏ICG：**
>
> **若一个游戏满足：**
>
> - 两名玩家交替行动；
> - 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
> - 不能行动的玩家判负；
>
> 则称该游戏为一个公平组合游戏。

4.11.1 Nim游戏

> 给定n堆石子，两位玩家轮流操作，每次操作可以从任意一堆石子中拿走任意数量的石子(可以拿完，但不能不拿)，最后无法进行操作的人视为失败。
>
> 两人都采用最优策略，先手是否必胜。
>
> **结论：**使用$x = a_1 \wedge a_2 \wedge a_3 \wedge \ldots \wedge a_n$，其中$a_i$表示第i堆石子的个数，$\wedge$表示按位异或操作，若`x = 0`则后手必赢，否则先手必赢。

> **实现：C++：**

```cpp
#include<iostream>

using namespace std;

int main()
{
    int n;
    scanf("%d", &n);
    int res = 0;
    while (n--)
    {
        int x;
        scanf("%d", &x);
        res ^= x;
    }
    if (res)
    {
        puts("Yes");
    }
    else
    {
        puts("No");
    }

    return 0;
}
```

> **实现：Java：**

```java
package mathematic.gametheory.nim;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * @author LBS59
 * @description Nim游戏
 */
public class Nim {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
```

```
        String[] row = in.readLine().split(" ");
        int res = 0;
        for (int i = 0; i < n; i++) {
            res ^= Integer.parseInt(row[i]);
        }
        if (res > 0) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }

        in.close();
    }
}
```

实现：Python：

```
if __name__ == '__main__':
    n = int(input())
    row = input().split()
    res = 0
    for i in range(n):
        res ^= int(row[i])
    if res:
        print("Yes")
    else:
        print("No")
```

4.11.2 SG函数

Mex运算：

设S表示一个非负整数集合，定义$mex(S)$为求出不属于集合S的最小非负整数的运算，即$mex(S) = \min\{x\}$，x属于自然数，且x不属于S。

SG函数：

在有向图游戏中，对于每一个节点x，设从x出发共有k条有向边，分别到达了节点y1，y2，...，yk，定义$SG(x)$为x的后继节点的SG函数值构成的结合再执行$mex(S)$运算的结果，即$SG(x) = mex(SG(y_1), SG(y_2), \ldots, SG(y_k))$。特别的，整个有向图邮箱G的$SG$函数值被定义为有向图游戏起点s的$SG$函数值，即$SG(G) = SG(s)$。

定理：

有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0；

有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值等于0。

实现：C++：

```
#include<iostream>
#include<cstring>
#include<unordered_set>

using namespace std;

const int N = 110, M = 10010;

int n, m;
int s[N], f[M];

int sg(int x)
{
    if (f[x] != -1)
    {
        return f[x];
    }
    unordered_set<int> S;
    for (int i = 0; i < m; i++)
    {
        int sum = s[i];
        if (x >= sum)
        {
            S.insert(sg(x - sum));
        }
    }
    for (int i = 0; ; i++)
    {
        if (!S.count(i))
        {
            return f[x] = i;
        }
    }
}

int main()
{
    cin >> m;
```

```
    for (int i = 0; i < m; i++)
    {
        cin >> s[i];
    }
    cin >> n;
    memset(f, -1, sizeof f);

    int res = 0;
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        res ^= sg(x);
    }
    if (res)
    {
        puts("Yes");
    }
    else
    {
        puts("No");
    }
}
```

实现：Java：

```
package mathematic.gametheory.mexandsg;

import java.util.Arrays;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

/**
 * @author LBS59
 * @description 集合Nim游戏
 */
public class MexAndSg {
    private static final int N = 110, M = 10010;

    private static int[] s, f;

    static {
        s = new int[N];
        f = new int[M];
        Arrays.fill(f, -1);
    }

    private static int sg(int x, int m) {
        if (f[x] != -1) {
            return f[x];
        }
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < m; i++) {
            int sum = s[i];
            if (x >= sum) {
                set.add(sg(x - sum, m));
            }
        }
        for (int i = 0; ; i++) {
            if (!set.contains(i)) {
                return f[x] = i;
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        for (int i = 0; i < m; i++) {
            s[i] = sc.nextInt();
        }
        int n = sc.nextInt();
        int res = 0;
        for (int i = 0; i < n; i++) {
            int x = sc.nextInt();
            res ^= sg(x, m);
        }
        if (res > 0) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
```

实现：Python：

```python
N, M = 110, 10010
s, f = [0] * N, [-1] * M


def sg(x: int, m: int) -> int:
    if f[x] != -1:
        return f[x]
    st = set()
    for i in range(m):
        sum = s[i]
        if x >= sum:
            st.add(sg(x - sum, m))
    i = 0
    while 1:
        if i not in st:
            f[x] = i
            return f[x]
        i += 1


if __name__ == '__main__':
    m = int(input())
    ms = input().split()
    for i in range(m):
        s[i] = int(ms[i])
    n = int(input())
    ns = input().split()
    res = 0
    for i in range(n):
        x = int(ns[i])
        res ^= sg(x, m)
    if res:
        print("Yes")
    else:
        print("No")
```

5. 动态规划

DP推导式：

- 状态表示：$f(i,j)$
  - 集合：
    - 所有选择方式：
    - 条件：
  - 属性：如最大值、最小值、数量
- 状态计算：
  - 集合划分：

5.1 背包问题

5.1.1 0-1背包问题

问题描述：有`N`个物品，每一个物品有一个体积w_i和一个价值v_i，有一个可以容纳体积为`V`的背包，可以装这`N`个物品，规定每一件物品只有一个，可选择装与不装，并且装进背包物品的总体积不超过背包的最大容纳量，问满足这样的条件，可以装进背包的物品的总价值最大为多少？

特点：每件物品最多可以装1次，也可以不装

实现：C++：二维dp

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

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N][N];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> v[i] >> w[i];
    }
    // dp[0][0] = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= m; j++)
        {
```

```
            f[i][j] = f[i - 1][j];
            if (j >= v[i])
            {
                f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
            }
        }
    }
    cout << f[n][m] << endl;

    return 0;
}
```

实现：C++：一维dp

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

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> v[i] >> w[i];
    }
    // dp[0][0] = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = m; j >= v[i]; j--)
        {
                f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << endl;

    return 0;
}
```

实现：Java：二维dp

```
package dp.bagsolution.zeroandonebag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 动态规划求解0-1背包问题
 * f[i][j]表示从前i个物品中挑选物品，并且总体积不超过j的所有方案中的最大价值
 * f[i][i] = max(f[i - 1][j], f[i - 1][j - v[i]] + w[i])
 * 注意第二项可能不满足条件，只有当前的j大于需要装的第i个物品的体积时才会取较大者
 */
public class ZeroAndOneBag {
    private static final int N = 1010;

    private static int[] v, w;
    private static int[][] f;

    static {
        v = new int[N];
        w = new int[N];
        f = new int[N][N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        // f[0][0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                f[i][j] = f[i - 1][j];
                if (j >= v[i]) {
                    f[i][j] = Math.max(f[i][j], f[i - 1][j - v[i]] + w[i]);
                }
            }
        }
```

```
        System.out.println(f[n][m]);
    }
}
```

实现：Java：一维dp

```
package dp.bagsolution.zeroandonebag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 动态规划求解0-1背包问题
 * f[i][j]表示从前i个物品中挑选物品，并且总体积不超过j的所有方案中的最大价值
 * f[i][i] = max(f[i - 1][j], f[i - 1][j - v[i]] + w[i])
 * 注意第二项可能不满足条件，只有当前的j大于需要装的第i个物品的体积时才会取较大者
 */
public class ZeroAndOneBag2 {
    private static final int N = 1010;

    private static int[] v, w;
    private static int[] f;

    static {
        v = new int[N];
        w = new int[N];
        f = new int[N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        // f[0][0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = m; j >= v[i]; j--) {
                f[j] = Math.max(f[j], f[j - v[i]] + w[i]);
            }
        }
        System.out.println(f[m]);
    }
}
```

实现：Python：二维dp

```
N = 1010
v, w, f = [0] * N, [0] * N, [[0] * N for _ in range(N)]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        row = input().split()
        v[i] = int(row[0])
        w[i] = int(row[1])
    # f[0][0] = 0
    for i in range(1, n + 1):
        for j in range(m + 1):
            f[i][j] = f[i - 1][j]
            if j >= v[i]:
                f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i])
    print(f[n][m])
```

实现：Python：一维dp

```
N = 1010
v, w, f = [0] * N, [0] * N, [0] * N


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        row = input().split()
        v[i] = int(row[0])
        w[i] = int(row[1])
    # f[0][0] = 0
    for i in range(1, n + 1):
        for j in range(m, v[i] - 1, -1):
            f[j] = max(f[j], f[j - v[i]] + w[i])
    print(f[m])
```

5.1.2 完全背包问题

> **问题描述：** 有`N`个物品，每一个物品有一个体积w_i和一个价值v_i，有一个可以容纳体积为`V`的背包，可以装这`N`个物品，规定每一件物品有无穷多个，并且每件物品可以装0个或多个，并且装进背包物品的总体积不超过背包的最大容纳量，问满足这样的条件，可以装进背包的物品的总价值最大为多少？
>
> **特点：** 每件物品可以装无穷多个，也可以不装

> **实现：C++：**

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

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> v[i] >> w[i];
    }
    // f[0] = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = v[i]; j <= m; j++)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << endl;

    return 0;
}
```

> **实现：Java：**

```java
package dp.bagsolution.completebag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 一维dp解决完全背包问题
 */
public class CompleteBag {
    private static final int N = 1010;

    private static int[] v, w;
    private static int[] f;

    static {
        v = new int[N];
        w = new int[N];
        f = new int[N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        // f[0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = v[i]; j <= m; j++) {
                f[j] = Math.max(f[j], f[j - v[i]] + w[i]);
            }
        }
        System.out.println(f[m]);
    }
}
```

> **实现：Python：**

```python
N = 1010
v, w, f = [0] * N, [0] * N, [0] * N

```

```python

if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        row = input().split()
        v[i] = int(row[0])
        w[i] = int(row[1])
    # f[0] = 0
    for i in range(1, n + 1):
        for j in range(v[i], m + 1):
            f[j] = max(f[j], f[j - v[i]] + w[i])
    print(f[m])
```

5.1.3 多重背包问题

> **问题描述：**和完全背包类似，但是规定每一件物品的个数分别有s_i个，这样就不能装无穷多个，最多只能装该物品上限个。并且装进背包物品的总体积不超过背包的最大容纳量，问满足这样的条件，可以装进背包的物品的总价值最大为多少？
>
> **特点：**在完全背包问题上，限制了每一件物品可以装的数量。

> **实现：C++：二维dp**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 110;

int n, m;
int v[N], w[N], s[N];
int f[N][N];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> v[i] >> w[i] >> s[i];
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= m; j++)
        {
            for (int k = 0; k <= s[i] && k * v[i] <= j; k++)
            {
                f[i][j] = max(f[i][j], f[i - 1][j - v[i] * k] + w[i] * k);
            }
        }
    }
    cout << f[n][m] << endl;
    return 0;
}
```

> **实现：C++：一维dp**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 12000;

int n, m;
int v[N], w[N];
int f[N];

int main()
{
    cin >> n >> m;
    int cnt = 0;
    for (int i = 1; i <= n; i++)
    {
        int a, b, s;
        cin >> a >> b >> s;
        int k = 1;
        while (k <= s)
        {
            cnt++;
            v[cnt] = a * k;
            w[cnt] = b * k;
            s -= k;
            k *= 2;
        }
        if (s > 0)
```

```
        {
            cnt++;
            v[cnt] = a * s;
            w[cnt] = b * s;
        }
    }
    n = cnt;
    for (int i = 1; i <= n; i++)
    {
        for (int j = m; j >= v[i]; j--)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

实现：Java：二维dp

```
package dp.bagsolution.multiplebag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 二维dp解决多重背包问题
 */
public class MultipleBag1 {
    private static final int N = 110;

    private static int[] v, w, s;
    private static int[][] f;

    static {
        v = new int[N];
        w = new int[N];
        s = new int[N];
        f = new int[N][N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
            s[i] = sc.nextInt();
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                for (int k = 0; k <= s[i] && k * v[i] <= j; k++) {
                    f[i][j] = Math.max(f[i][j], f[i - 1][j - v[i] * k] + w[i] * k);
                }
            }
        }
        System.out.println(f[n][m]);
    }
}
```

实现：Java：一维dp

```
package dp.bagsolution.multiplebag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 一维dp解决多重背包问题
 */
public class MultipleBag2 {
    private static final int N = 12000;

    private static int[] v, w;
    private static int[] f;

    static {
        v = new int[N];
        w = new int[N];
        f = new int[N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
```

```
        int cnt = 0;
        for (int i = 1; i <= n; i++) {
            int a = sc.nextInt(), b = sc.nextInt(), s = sc.nextInt();
            int k = 1;
            while (k <= s) {
                cnt++;
                v[cnt] = a * k;
                w[cnt] = b * k;
                s -= k;
                k <<= 1;
            }
            if (s > 0) {
                cnt++;
                v[cnt] = a * s;
                w[cnt] = b * s;
            }
        }
        n = cnt;
        for (int i = 1; i <= n; i++) {
            for (int j = m; j >= v[i]; j--) {
                f[j] = Math.max(f[j], f[j - v[i]] + w[i]);
            }
        }
        System.out.println(f[m]);
    }
}
```

实现：Python：二维dp

```
N = 1010
v, w, s, f = [0] * N, [0] * N, [0] * N, [[0] * N for _ in range(N)]


if __name__ == '__main__':
    st = input().split()
    n, m = int(st[0]), int(st[1])
    for i in range(1, n + 1):
        row = input().split()
        v[i] = int(row[0])
        w[i] = int(row[1])
        s[i] = int(row[2])
    # f[0] = 0
    for i in range(1, n + 1):
        for j in range(m + 1):
            k = 0
            while k <= s[i] and k * v[i] <= j:
                f[i][j] = max(f[i][j], f[i - 1][j - v[i] * k] + w[i] * k)
                k += 1
    print(f[n][m])
```

实现：Python：一维dp

```
N = 12000
v, w, f = [0] * N, [0] * N, [0] * N


if __name__ == '__main__':
    st = input().split()
    n, m = int(st[0]), int(st[1])
    cnt = 0
    for i in range(1, n + 1):
        row = input().split()
        a, b, s = int(row[0]), int(row[1]), int(row[2])
        k = 1
        while k <= s:
            cnt += 1
            v[cnt] = a * k
            w[cnt] = b * k
            s -= k
            k <<= 1
        if s:
            cnt += 1
            v[cnt] = a * s
            v[cnt] = b * s
    n = cnt
    for i in range(1, n + 1):
        for j in range(m, v[i] - 1, -1):
            f[j] = max(f[j], f[j - v[i]] + w[i])
    print(f[m])
```

5.1.4 分组背包问题

> **问题描述：** 在多重背包的基础上进行扩展，有`n`组多重背包，在每一组中，只能选择一个物品，并且装进背包物品的总体积不超过背包的最大容纳量，问满足这样的条件，可以装进背包的物品的总价值最大为多少？
>
> **特点：** 每一个组内部物品之间互斥，只可选择其中一组。
>
> **解决思路：** 和多重背包一样的思想

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 110;

int n, m;
int v[N][N], w[N][N], s[N];
int f[N];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> s[i];
        for (int j = 0; j < s[i]; j++)
        {
            cin >> v[i][j] >> w[i][j];
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = m; j >= 0; j--)
        {
            for (int k = 0; k < s[i]; k++)
            {
                if (v[i][k] <= j)
                {
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
                }
            }
        }
    }
    cout << f[m] << endl;

    return 0;
}
```

> **实现：Java：**

```java
package dp.bagsolution.groupbag;

import java.util.Scanner;

/**
 * @author LBS59
 * @description dp解决分组背包问题
 */
public class GroupBag {
    private static final int N = 110;

    private static int[][] v, w;
    private static int[] s;
    private static int[] f;

    static {
        v = new int[N][N];
        w = new int[N][N];
        s = new int[N];
        f = new int[N];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), m = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            s[i] = sc.nextInt();
            for (int j = 0; j < s[i]; j++) {
                v[i][j] = sc.nextInt();
                w[i][j] = sc.nextInt();
            }
        }
        for (int i = 1; i <= n; i++) {
```

```
            for (int j = m; j >= 0; j--) {
                for (int k = 0; k < s[i]; k++) {
                    if (v[i][k] <= j) {
                        f[j] = Math.max(f[j], f[j - v[i][k]] + w[i][k]);
                    }
                }
            }
        }
        System.out.println(f[m]);
    }
}
```

实现：Python：

```
N = 110
v, w, s, f = [[0] * N for _ in range(N)], [[0] * N for _ in range(N)], [0] * N, [0] * N


if __name__ == '__main__':
    st = input().split()
    n, m = int(st[0]), int(st[1])
    for i in range(1, n + 1):
        s[i] = int(input())
        for j in range(s[i]):
            row = input().split()
            v[i][j] = int(row[0])
            w[i][j] = int(row[1])
    for i in range(1, n + 1):
        for j in range(m, -1, -1):
            for k in range(s[i]):
                if v[i][k] <= j:
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k])
    print(f[m])
```

5.2 线性dp

5.2.1 数字三角形问题

问题描述：给定下图三角形，从顶点出发，在每个结点可以选择移动至其左下方的结点或移动至其右下方的结点，一直走到底层，找出一条路径，使得路径上的数组的和最大。

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

思路：有点类似于0-1背包问题的曲线救国策略

实现：C++：

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

const int N = 510, INF = 1e9;

int n;
int a[N][N];
int f[N][N];

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= i + 1; j++)
        {
            f[i][j] = -INF;
        }
    }
    f[1][1] = a[1][1];
    for (int i = 2; i <= n; i++)
    {
        for (int j = 1; j <= i; j++)
        {
```

```
            f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j]);
        }
    }
    int res = -INF;
    for (int i = 1; i <= n; i++)
    {
        res = max(res, f[n][i]);
    }
    printf("%d", res);
    return 0;
}
```

实现：Java：

```
package dp.lineardp.numtriangle;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 线性dp解决数字三角形问题
 *     7
 *    3 8
 *   8 1 0
 *  2 7 4 4
 * 4 5 2 6 5
 */
public class DigitTriangle {
    private static final int N = 510, INF = Integer.MAX_VALUE;

    private static int[][] a;
    private static int[][] f;

    static {
        a = new int[N][N];
        f = new int[N][N];
        for (int i = 0; i < N; i++) {
            Arrays.fill(f[i], -INF);
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        for (int i = 1; i <= n; i++) {
            String[] row = in.readLine().split(" ");
            for (int j = 1; j <= i; j++) {
                a[i][j] = Integer.parseInt(row[j - 1]);
            }
        }
        f[1][1] = a[1][1];
        for (int i = 2; i <= n; i++) {
            for (int j = 1; j <= i; j++) {
                f[i][j] = Math.max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j]);
            }
        }
        int res = -INF;
        for (int i = 1; i <= n; i++) {
            res = Math.max(res, f[n][i]);
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```
N, INF = 510, int(1e9)
a, f = [[0] * N for _ in range(N)], [[-INF] * N for _ in range(N)]


if __name__ == '__main__':
    n = int(input())
    for i in range(1, n + 1):
        row = input().split()
        for j in range(1, i + 1):
            a[i][j] = int(row[j - 1])
    f[1][1] = a[1][1]
    for i in range(2, n + 1):
        for j in range(1, i + 1):
```

```
            f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j])
    res = -INF
    for i in range(1, n + 1):
        res = max(res, f[n][i])
    print(res)
```

5.2.2 最长上升子序列

> **问题描述：** 给定一个长度为N的数列，求数值严格单调递增的子序列有多少个?
>
> **解决方法：** 使用`f[i]`表示以i结尾的所有上升子序列的集合，`f[i]`的值表示为这些子序列中长度最大值，曲线救国的方法
>
> $$f[i] = max(f[i], f[j] + 1)，其中j < i并且a[j] < a[i] \tag{28}$$
>
> **时间复杂度：** $O(N^2)$

实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010;

int n;
int a[N], f[N];

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
    }
    for (int i = 1; i <= n; i++)
    {
        f[i] = 1;
        for (int j = 1; j < i; j++)
        {
            if (a[j] < a[i])
            {
                f[i] = max(f[i], f[j] + 1);
            }
        }
    }
    int res = 0;
    for (int i = 1; i <= n; i++)
    {
        res = max(res, f[i]);
    }
    printf("%d", res);

    return 0;
}
```

实现：Java：

```java
package dp.lineardp.lus;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 一维dp解决最长上升子序列问题
 */
public class LongestUpSequence {
    private static final int N = 1010;

    private static int[] a, f;

    static {
        a = new int[N];
        f = new int[N];
        Arrays.fill(f, 1);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        String[] row = in.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
```

```
            a[i] = Integer.parseInt(row[i - 1]);
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j < i; j++) {
                if (a[j] < a[i]) {
                    f[i] = Math.max(f[i], f[j] + 1);
                }
            }
        }
        int res = 0;
        for (int i = 1; i <= n; i++) {
            res = Math.max(res, f[i]);
        }
        System.out.println(res);

        in.close();
    }


```

实现：Python：

```
N = 1010
a, f = [0] * N, [1] * N


if __name__ == '__main__':
    n = int(input())
    row = input().split()
    for i in range(1, n + 1):
        a[i] = int(row[i - 1])
    for i in range(1, n + 1):
        for j in range(1, i):
            if a[j] < a[i]:
                f[i] = max(f[i], f[j] + 1)
    res = 0
    for i in range(1, n + 1):
        res = max(res, f[i])
    print(res)
```

针对$O(N^2)$做贪心优化，预处理出每一个长度下的最长上升子序列末尾的最小值，然后再遍历每一个位置，找到可以匹配的末尾的最大值，更新新的长度。

实现：C++：

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010;

int n;
int a[N];
int q[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    int len = 0;
    q[0] = -2e9;
    for (int i = 0; i < n; i++)
    {
        int l = 0, r = len;
        while (l < r)
        {
            int mid = l + r + 1 >> 1;
            if (q[mid] < a[i])
            {
                l = mid;
            }
            else
            {
                r = mid - 1;
            }
        }
        len = max(len, r + 1);
        q[r + 1] = a[i];
    }
    printf("%d", len);


```

```
    return 0;
}
```

实现：Java：

```java
package dp.lineardp.lus;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * @author LBS59
 * @description 预处理优化版最长上升子序列
 */
public class LongestUpSequence2 {
    private static final int N = 1010;

    private static int[] a, q;

    static {
        a = new int[N];
        q = new int[N];
        q[0] = Integer.MIN_VALUE;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        String[] row = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            a[i] = Integer.parseInt(row[i]);
        }
        int len = 0;
        for (int i = 0; i < n; i++) {
            int l = 0, r = len;
            while (l < r) {
                int mid = l + r + 1 >> 1;
                if (q[mid] < a[i]) {
                    l = mid;
                } else {
                    r = mid - 1;
                }
            }
            len = Math.max(len, r + 1);
            q[r + 1] = a[i];
        }
        System.out.println(len);

        in.close();
    }
}
```

实现：Python：

```python
N = 1010
a, q = [0] * N, [1] * N


if __name__ == '__main__':
    n = int(input())
    row = input().split()
    for i in range(n):
        a[i] = int(row[i])
    ln = 0
    q[0] = int(-2e9)
    for i in range(n):
        l, r = 0, ln
        while l < r:
            mid = l + r + 1 >> 1
            if q[mid] < a[i]:
                l = mid
            else:
                r = mid - 1
        ln = max(ln, r + 1)
        q[r + 1] = a[i]
    print(ln)
```

5.2.3 最长公共子序列

> **问题描述：** 给定两个长度分别为 `N` 和 `M` 的字符串 `A` 和 `B`，求既是 `A` 的子序列又是 `B` 的子序列的字符串长度最长是多少？
>
> **核心思想：** 动态规划
>
> - 使用 `f[i][j]` 表示在 `A` 字符串中由前 `i` 个字符构成的所有子序列，以及在 `B` 字符串中由前 `j` 个字符构成的所有子序列中，两个子序列集合中子序列相等的最长子序列的长度。
> - 计算 `f[i][j]` 划分为四种情况：
>   - 0-0：构成 `A` 的子序列不包含第 `i` 个字符，并且构成 `B` 的子序列不包含第 `j` 个字符；（**可以不用计算，因为下面两种情况已经包含了此情况**）
>   - 0-1：构成 `A` 的子序列不包含第 `i` 个字符，并且构成 `B` 的子序列包含第 `j` 个字符；
>   - 1-0：构成 `A` 的子序列包含第 `i` 个字符，并且构成 `B` 的子序列不包含第 `j` 个字符；
>   - 1-1：构成 `A` 的子序列包含第 `i` 个字符，并且构成 `B` 的子序列包含第 `j` 个字符；

实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010;

int n, m;
char a[N], b[N];
int f[N][N];

int main()
{
    scanf("%d%d", &n, &m);
    scanf("%s%s", a + 1, b + 1);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if (a[i] == b[j])
            {
                f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
            }
        }
    }
    printf("%d", f[n][m]);

    return 0;
}
```

实现：Java：

```java
package dp.lineardp.lcs;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * @author LBS59
 * @description 动态规划解决最长公共子序列问题
 */
public class LongestCommonSubSequence {
    private static final int N = 1010;

    private static int[][] f;

    static {
        f = new int[N][N];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        String a = in.readLine();
        String b = in.readLine();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                f[i + 1][j + 1] = Math.max(f[i][j + 1], f[i + 1][j]);
                if (a.charAt(i) == b.charAt(j)) {
                    f[i + 1][j + 1] = Math.max(f[i + 1][j + 1], f[i][j] + 1);
                }
            }
        }
        System.out.println(f[n][m]);
```

```
        in.close();
    }
}
```

实现：Python：

```python
N = 1010
f = [[0] * N for _ in range(N)]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    a = input()
    b = input()
    for i in range(n):
        for j in range(m):
            f[i + 1][j + 1] = max(f[i][j + 1], f[i + 1][j])
            if a[i] == b[j]:
                f[i + 1][j + 1] = max(f[i + 1][j + 1], f[i][j] + 1)
    print(f[n][m])
```

5.3 区间dp

5.3.1 石子游戏

问题描述：设有`N`堆石子，其编号为1，2，3，...，N。每对石子有一定的质量，可以用一个整数来描述，现在要将这`N`堆石子合并称为一堆。每次之恶能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时选择的顺序不同，合并的总代价也不同。

问题是：找出一种合理的方法，使总的代价最小

核心思想：也是模拟出最后一次的操作，最后一次就是将最后的两堆石子合并，假定区间为`[i, j]`，最后的两堆石子为`[i, k]`和`[k + 1, j]`则这个`k`有若干种取法，$k \in [i, j)$，针对每一种情况，可以采取曲线救国的思想，因为所有情况最后都是要将最后两堆石子合并，所以可以得到合并最后两堆石子之前的状态为`f[i][k] + f[k + 1][j]`，并且合并最后一次的代价为`sum[i, j]`，这个区间求和可以使用前缀和提前处理好，则最终的结果就是所有情况中代价的最小值。

$$f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + sum[i, j]),\ 其中k \in [i, j) \tag{29}$$

实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 310;

int n;
int s[N];
int f[N][N];

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &s[i]);
        s[i] += s[i - 1];
    }
    for (int len = 2; len <= n; len++)
    {
        for (int i = 1; i + len - 1 <= n; i++)
        {
            int l = i, r = i + len - 1;
            f[l][r] = 1e9;
            for (int k = l; k < r; k++)
            {
                f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s[l - 1]);
            }
        }
    }
    printf("%d", f[1][n]);

    return 0;
}
```

实现：Java：

```java
package dp.sectiondp.stonemerge;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```java

/**
 * @author LBS59
 * @description 区间dp解决石子合并问题
 */
public class StoneMerge {
    private static final int N = 310;

    private static int[] s;
    private static int[][] f;

    static {
        s = new int[N];
        f = new int[N][N];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine());
        String[] row = in.readLine().split(" ");
        for (int i = 1; i <= n; i++) {
            s[i] = Integer.parseInt(row[i - 1]);
            s[i] += s[i - 1];
        }
        for (int len = 2; len <= n; len++) {
            for (int i = 1; i + len - 1 <= n; i++) {
                int r = i + len - 1;
                f[i][r] = Integer.MAX_VALUE;
                for (int k = i; k < r; k++) {
                    f[i][r] = Math.min(f[i][r], f[i][k] + f[k + 1][r] + s[r] - s[i - 1]);
                }
            }
        }
        System.out.println(f[1][n]);
    }
}
```

实现：Python：

```python
N = 310
s, f = [0] * N, [[0] * N for _ in range(N)]


if __name__ == '__main__':
    n = int(input())
    row = input().split()
    for i in range(1, n + 1):
        s[i] = int(row[i - 1])
        s[i] += s[i - 1]
    for ln in range(2, n + 1):
        i = 1
        while i + ln - 1 <= n:
            r = i + ln - 1
            f[i][r] = int(1e9)
            for k in range(i, r):
                f[i][r] = min(f[i][r], f[i][k] + f[k + 1][r] + s[r] - s[i - 1])
            i += 1
    print(f[1][n])
```

5.4 计数类dp

5.4.1 整数划分问题

问题描述：一个正整数`n`可以表示成若干个正整数之和，形如：$n=n_1+n_2+...+n_k$，其中$n_1 \geq n_2 \geq ... \geq n_k$，$k \geq 1$，我们称这样的一种表示称为正整数`n`的一种划分。现给定一个正整数`n`，求出`n`共有多少种不同的划分方法。

核心思想：

1. 使用完全背包问题来求解，因为我们在使用某个数字拼`n`时，不考虑顺序的关系，每一个数字的使用个数无上限，则可以转化为完全背包问题来求解。

- 因为此题求解的是所有可以组合成`n`的方案数
- `f[i][j]`表示使用前`i`个数字拼接成`j`的所有的方案数，针对此状态的前一个状态，就是针对`i`数字的使用个数进行讨论，则可知`i`的使用个数有$0,1,2,...s$，其中$s*i<=n$，即$f[i][j]=(f[i-1][j]+f[i-1][j-i]+...f[i-1][j-s*i])$，同理可以得出$s[i][j-i]=f[i-1][j-i]+f[i-1][j-2*i]+...+f[i-1][j-s*i]$，将$s[i][j-i$带入$f[i][j]$中，可得$f[i][j]=f[i-1][j]+f[i][j-1]$，使用完全背包问题的优化，可得$f[j]=f[j]+f[j-i]$

2. 使用`f[i][j]`表示所有的总和是`i`，并且恰好表示成`j`个数的和的方案。

- 状态计算分两大类讨论：
  - `j`个数中的最小值为1，则我们将这个最小值去掉，得到`f[i - 1][j - 1]`，这一类组合和没有去掉1之前的方案是一一对应的；
  - `j`个数中的最小值大于1，则我们将这`j`个数的每一个数都减去1，得到`f[i - j][j]`，这一类组合和没有减之前的方案是一一对应的。
  - 最后的组合的和为所有的不同数量的`j`得到`n`的组合方案之和，即$res=f[n][1]+f[n][2]+...+f[n][n]$。

完全背包方法实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010, mod = 1e9 + 7;

int n;
int f[N];

int main()
{
    cin >> n;
    f[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i; j <= n; j++)
        {
            f[j] = (f[j] + f[j - i]) % mod;
        }
    }
    cout << f[n] << endl;

    return 0;
}
```

完全背包方法实现：Java：

```java
package dp.countingdp.digitdiv;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 整数划分问题--完全背包方法求解
 */
public class IntDivide {
    private static final int N = 1010, MOD = (int) 1e9 + 7;

    private static int[] f;

    static {
        f = new int[N];
        f[0] = 1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 1; i <= n; i++) {
            for (int j = i; j <= n; j++) {
                f[j] = (f[j] + f[j - i]) % MOD;
            }
        }
        System.out.println(f[n]);
    }
}
```

完全背包方法实现：Python：

```python
N, MOD = 1010, int(1e9 + 7)
f = [0] * N


if __name__ == '__main__':
    n = int(input())
    f[0] = 1
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            f[j] = (f[j] + f[j - i]) % MOD
    print(f[n])
```

方法二实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010, mod = 1e9 + 7;

int n;
```

```cpp
int f[N][N];

int main()
{
    cin >> n;
    f[0][0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % mod;
        }
    }
    int res = 0;
    for (int i = 1; i <= n; i++)
    {
        res = (res + f[n][i]) % mod;
    }
    cout << res << endl;

    return 0;
}
```

方法二实现：Java：

```java
package dp.countingdp.digitdiv;

import java.util.Scanner;

/**
 * @author LBS59
 * @description 玄学法求解整数划分问题
 */
public class IntDivide2 {
    private static final int N = 1010, MOD = (int) 1e9 + 7;

    private static int[][] f;

    static {
        f = new int[N][N];
        f[0][0] = 1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= i; j++) {
                f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % MOD;
            }
        }
        int res = 0;
        for (int i = 1; i <= n; i++) {
            res = (res + f[n][i]) % MOD;
        }
        System.out.println(res);
    }
}
```

方法二实现：Python：

```python
N, MOD = 1010, int(1e9 + 7)
f = [[0] * N for _ in range(N)]


if __name__ == '__main__':
    n = int(input())
    f[0][0] = 1
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % MOD
    res = 0
    for i in range(1, n + 1):
        res = (res + f[n][i]) % MOD
    print(res)
```

5.5 数位统计dp

5.5.1 计数问题

> **问题描述：**给定两个整数 a 和 b，求 a 到 b 之间的所有数字中 0~9 的出现次数。
>
> **核心思想：**分情况讨论---前缀和s思想：
>
> - 设定所求区间为 [a, b]，求 a 到 b 之间 0~9 各出现了多少次
> - 设计一个函数 count(n, x) 表示 1~n 之间数字 x 出现的次数，则 [a, b] 之间 x 出现的次数可以表示为 count(b, x) - count(a - 1, x)

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int get(vector<int> num, int l, int r)
{
    int res = 0;
    for (int i = l; i >= r; i--)
    {
        res = res * 10 + num[i];
    }
    return res;
}

int power10(int x)
{
    int res = 1;
    while (x--)
    {
        res *= 10;
    }
    return res;
}

int count(int n, int x)
{
    if (!n)
    {
        return 0;
    }
    vector<int> num;
    while (n)
    {
        num.push_back(n % 10);
        n /= 10;
    }
    int res = 0;
    for (int i = n - 1 - !x; i >= 0; i--)
    {
        if (i < n - 1)
        {
            res += get(num, n - 1, i + 1) * power10(i);
            if (!x)
            {
                res -= power10(i);
            }
        }
        if (num[i] == x)
        {
            res += get(num, i - 1, 0) + 1;
        }
        else if (num[i] > x)
        {
            res += power10(i);
        }
    }
    return res;
}

int main()
{
    int a, b;
    while (cin >> a >> b, a || b)
    {
        if (a > b)
        {
            swap(a, b);
        }
        for (int i = 0; i < 10; i++)
        {
            cout << count(b, i) - count(a - 1, i) << ' ';
        }
        cout << endl;
```

```
    }

    return 0;
}
```

实现：Java：

```java
package dp.numposdp.counting;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 数位dp解决计数问题
 */
public class Count {
    private static int get(List<Integer> num, int l, int r) {
        int res = 0;
        for (int i = l; i >= r; i--) {
            res = res * 10 + num.get(i);
        }
        return res;
    }

    private static int count(int n, int x) {
        if (n == 0) {
            return 0;
        }
        List<Integer> list = new ArrayList<>();
        while (n > 0) {
            list.add(n % 10);
            n /= 10;
        }
        n = list.size();

        int res = 0;
        for (int i = n - 1 - ((x == 0) ? 1 : 0); i >= 0; i--) {
            if (i < n - 1) {
                res += get(list, n - 1, i + 1) * (int) Math.pow(10, i);
                if (x == 0) {
                    res -= (int) Math.pow(10, i);
                }
            }
            if (list.get(i) == x) {
                res += get(list, i - 1, 0) + 1;
            } else if (list.get(i) > x) {
                res += (int) Math.pow(10, i);
            }
        }
        return res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true) {
            int a = sc.nextInt(), b = sc.nextInt();
            if (a == 0 && b == 0) {
                break;
            }
            if (a > b) {
                int temp = a;
                a = b;
                b = temp;
            }
            for (int i = 0; i < 10; i++) {
                System.out.printf("%d ", count(b, i) - count(a - 1, i));
            }
            System.out.println();
        }
    }
}
```

实现：Python：

```python
from typing import List


def get(ls: List, t: int, r: int) -> int:
    res = 0
    for i in range(t, r - 1, -1):
        res = res * 10 + ls[i]
    return res

```

```python

def count(n: int, x: int) -> int:
    if not n:
        return 0
    num = []
    while n:
        num.append(n % 10)
        n //= 10
    n = len(num)

    res = 0
    for i in range(n - 1 - (not x), -1, -1):
        if i < n - 1:
            res += get(num, n - 1, i + 1) * 10 ** i
            if not x:
                res -= 10 ** i
        if num[i] == x:
            res += get(num, i - 1, 0) + 1
        elif num[i] > x:
            res += 10 ** i
    return res


if __name__ == '__main__':
    while True:
        row = input().split()
        a, b = int(row[0]), int(row[1])
        if not a and not b:
            break
        if a > b:
            a, b = b, a
        for j in range(10):
            print(count(b, j) - count(a - 1, j), end=" ")
        print()
```

5.6 状态压缩dp

5.6.1 蒙德里安的梦想

> **问题描述：** 求把$N \times M$的棋盘分割成若干个1×2的长方形，有多少种方案。
>
> **核心思想：** **先放横着的，再放竖着的。**
>
> - 因为将所有横着的方格摆放正确，竖的方格的摆放方式是固定的，这两种摆放方式是一一对应的，只需要处理其中一种，这里处理所有横着摆放的情况，定义`f[i][j]`表示前`i - 1`列已经摆好，`j`表示的是从`i - 1`列伸出来到第`i`列的所有方格的状态表示，伸出来的行表示为`1`，没有伸出来的为`0`，所有行的二进制组成的整数就是`j`的值。
> - 如何判断方案的合法性？（1）这里使用`f[i - 1][k]`表示前一列的状态，因为是前一列，所以当前列摆放的横着的方格不能与前一列在同一行，即$j \& k == 0$；（2）前一列所有剩余位置，能否使用竖着的方格摆满，按列来看，就是每一列中连续空着的小方格的数量必须为偶数个。
> - 所有`j`的可能值有2^N个，因为每个位置有伸出与不伸出两种选择，方案数就是这些所有状态下的所有方案数的和。
>
> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;

const int N = 12, M = 1 << N;

int n, m;
long long f[N][M];
bool st[M];

int main()
{
    int n, m;
    while (cin >> n >> m, n || m)
    {
        memset(f, 0, sizeof f);
        for (int i = 0; i < 1 << n; i++)
        {
            st[i] = true;
            int cnt = 0;
            for (int j = 0; j < n; j++)
            {
                if (i >> j & 1)
                {
                    if (cnt & 1)
                    {
                        st[i] = false;
                        cnt = 0;
                    }
                }
                else
```

```
                {
                    cnt++;
                }
            }
            if (cnt & 1)
            {
                st[i] = false;
            }
        }
        f[0][0] = 1;
        for (int i = 1; i <= m; i++)
        {
            for (int j = 0; j < 1 << n; j++)
            {
                for (int k = 0; k < 1 << n; k++)
                {
                    if ((j & k) == 0 && st[j | k])
                    {
                        f[i][j] += f[i - 1][k];
                    }
                }
            }
        }
        cout << f[m][0] << endl;
    }

    return 0;
}
```

实现：Java：

```
package dp.statecompressdp.dreamofmondrian;

import java.util.Arrays;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 状态压缩dp例题-蒙德里安的梦想
 */
public class DreamOfMondrian {
    private static final int N = 12, M = 1 << N;

    private static long[][] f;

    private static boolean[] st;

    static {
        f = new long[N][M];
        st = new boolean[M];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true) {
            int n = sc.nextInt(), m = sc.nextInt();
            if (n == 0 && m == 0) {
                break;
            }
            for (int i = 0; i < N; i++) {
                Arrays.fill(f[i], 0);
            }
            for (int i = 0; i < 1 << n; i++) {
                st[i] = true;
                int cnt = 0;
                for (int j = 0; j < n; j++) {
                    if ((i >> j & 1) != 0) {
                        if ((cnt & 1) != 0) {
                            st[i] = false;
                            cnt = 0;
                        }
                    } else {
                        cnt++;
                    }
                }
                if ((cnt & 1) != 0) {
                    st[i] = false;
                }
            }
            f[0][0] = 1;
            for (int i = 1; i <= m; i++) {
                for (int j = 0; j < 1 << n; j++) {
                    for (int k = 0; k < 1 << n; k++) {
                        if ((j & k) == 0 && st[j | k]) {
                            f[i][j] += f[i - 1][k];
                        }
```

```
                    }
                }
            }
            System.out.println(f[m][0]);
        }
    }
}
```

实现：Python：

```
N = 12
M = 1 << N
f = [[0] * M for _ in range(N)]
st = [False] * M


if __name__ == '__main__':
    while True:
        row = input().split()
        n, m = int(row[0]), int(row[1])
        if not (n or m):
            break
        for i in range(N):
            for j in range(M):
                f[i][j] = 0
        for i in range(1 << n):
            st[i] = True
            cnt = 0
            for j in range(n):
                if i >> j & 1:
                    if cnt & 1:
                        st[i] = False
                        cnt = 0
                else:
                    cnt += 1
            if cnt & 1:
                st[i] = False
        f[0][0] = 1
        for i in range(1, m + 1):
            for j in range(1 << n):
                for k in range(1 << n):
                    if not j & k and st[j | k]:
                        f[i][j] += f[i - 1][k]
        print(f[m][0])
```

5.6.2 最短Hamilton路径

问题描述：给定一张`n`个点的带权无向图，点从`0 ~ n - 1`编号，求起点`0`到终点`n - 1`的最短Hamilton路径。

Hamilton路径的定义是从`0`到`n - 1`不重不漏地经过每个点恰好一次。

核心思想：状态压缩+分情况讨论

- 状态表示：`f[i][j]`表示从`0`走到`j`，并且走过的所有点使用`1`表示，未走过的点用`0`表示，所有状态的二进制表示即为`i`。
- 曲线救国思想，设定我们走到`j`这个点之前走到了`k`这个点，并且`k`可以一步直接走到`j`，因为`w[j][k]`边的权重是个定值，我们只需要得到从`i`走到`k`之间的权重最小值即可，这里需要`i`可以走到`k`并且中间的路径不包含`j`。

实现：C++：

```
#include<iostream>
#include<cstring>
#include<algorithm>

using namespace std;

const int N = 20, M = 1 << N;

int n;
int w[N][N];
int f[M][N];

int main()
{
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> w[i][j];
        }
    }
    memset(f, 0x3f, sizeof f);
    f[1][0] = 0;
    for (int i = 0; i < 1 << n; i++)
    {
        for (int j = 0; j < n; j++)
```

```
        {
            if (i >> j & 1)
            {
                for (int k = 0; k < n; k++)
                {
                    if ((i - (1 << j)) >> k & 1)
                    {
                        f[i][j] = min(f[i][j], f[i - (1 << j)][k] +w[k][j]);
                    }
                }
            }
        }
    }
    cout << f[(1 << n) - 1][n - 1] << endl;

    return 0;
}
```

实现：Java：

```
package dp.statecompressdp.shp;

import java.util.Arrays;
import java.util.Scanner;

/**
 * @author LBS59
 * @description 状态压缩dp解决最短Hamilton路径问题
 */
public class ShortestHamiltonPath {
    private static final int N = 20, M = 1 << N, INF = 0x3f3f3f3f;
    /**
     * 记录边权
     */
    private static int[][] w;
    /**
     * 记录状态，f[i][j]表示，从0到达j，并且状态为i的最短Hamilton路径
     */
    private static int[][] f;

    static {
        w = new int[N][N];
        f = new int[M][N];
        // 初始化所有的状态的Hamilton路径为无穷，
        for (int i = 0; i < M; i++) {
            Arrays.fill(f[i], INF);
        }
        // f[1][0]表示从0到达0，没有经过任何边，状态为1，则Hamilton路径为0
        f[1][0] = 0;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                w[i][j] = sc.nextInt();
            }
        }
        for (int i = 0; i < 1 << n; i++) {
            for (int j = 0; j < n; j++) {
                // 如果可以从i到达j
                if ((i >> j & 1) != 0) {
                    // 遍历中间结点k，表示从i可以不经过j到达k并且从k可以一步直接到达j
                    for (int k = 0; k < n; k++) {
                        // i不经过j可以到达k
                        if (((i - (1 << j)) >> k & 1) != 0) {
                            f[i][j] = Math.min(f[i][j], f[i - (1 << j)][k] + w[k][j]);
                        }
                    }
                }
            }
        }
        // 应该输出的是从0达到n - 1并且经过所有结点的最短路径
        System.out.println(f[(1 << n) - 1][n - 1]);
    }
}
```

实现：Python：

```
N, INF = 20, int(0x3f3f3f3f)
M = 1 << N
w, f = [[0] * N for _ in range(N)], [[INF] * N for _ in range(M)]


```

```python
if __name__ == '__main__':
    n = int(input())
    for i in range(n):
        row = input().split()
        for j in range(n):
            w[i][j] = int(row[j])
    f[1][0] = 0
    for i in range(1 << n):
        for j in range(n):
            if i >> j & 1:
                for k in range(n):
                    if (i - (1 << j)) >> k & 1:
                        f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j])
    print(f[(1 << n) - 1][n - 1])
```

5.7 树形dp

5.7.1 没有上司的舞会

> **问题描述：** 学校有`N`名职员，编号为`1 ~ N`。他们的关系就像一颗以校长为根的树，父节点就是子节点的直接上司。每个职员有一个快乐指数，用整数H_i给出，其中$1 \leq i \leq N$。现要召开一场周年庆宴会，不过没有职员愿意和直接上司一起参会。在满足这个条件的前提下，主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个最大值。
>
> **核心思想：** 树形dp
>
> - 状态表示：
>   - `f[u][0]`：表示从`u`这个节点所在子树进行选择，并且不选择`u`这个节点的最大值；
>   - `f[u][1]`：表示从`u`这个节点所在子树进行选择，并且选择`u`这个节点的最大值；
> - 状态计算：曲线救国的思想，我们在计算`u`这个节点时，需要提前处理好`u`的所有儿子的最大值，假设`u`这个节点有两个儿子节点`s1`和`s1`，我们需要提前处理出`f[s1][0]`、`f[s1][1]`以及`f[s2][0]`、`f[s2][1]`，当我们不选择`u`这个节点时，$f[u][0] = \sum\{max(f(s_i,0),f(s_i,1))\}$，其中$i$表示$u$的所有儿子节点，同样地，当我们选择`u`这个节点时，$f[u][1] = \sum\{f(s_i,0)\}$，其中$i$表示$u$的所有儿子节点
>
> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;

const int N = 6010;

int n;
int happy[N];
int h[N], e[N], ne[N], idx;
int f[N][2];
bool has_father[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a]; h[a] = idx++;
}

void dfs(int u)
{
    f[u][1] = happy[u];
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        dfs(j);
        f[u][0] += max(f[j][0], f[j][1]);
        f[u][1] += f[j][0];
    }
}

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &happy[i]);
    }
    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i++)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        has_father[a] = true;
        add(b, a);
    }
    int root = 1;
    while (has_father[root])
    {
        root++;
```

```
    }
    dfs(root);
    printf("%d\n", max(f[root][0], f[root][1]));

    return 0;
}
```

实现：Java：

```java
package dp.treedp.partywithoutboss;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 树形dp解决没有上司的舞会问题
 */
public class PartyWithoutBoss {
    private static final int N = 6010;

    /**
     * 存储每一个人的快乐指数
     */
    private static int[] happy;
    /**
     * 散列表四件套
     */
    private static int[] e, ne, h;
    private static int idx;
    /**
     * 状态表示
     */
    private static int[][] f;
    /**
     * 存储当前节点是否有父节点，方便找到根节点
     */
    private static boolean[] hasFather;

    static {
        happy = new int[N];
        e = new int[N];
        ne = new int[N];
        h = new int[N];
        Arrays.fill(h, -1);
        idx = 0;
        f = new int[N][2];
        hasFather = new boolean[N];
    }

    private static void add(int a, int b) {
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx++;
    }

    private static void dfs(int u) {
        f[u][1] = happy[u];
        for (int i = h[u]; i != -1; i = ne[i]) {
            int j = e[i];
            dfs(j);
            f[u][0] += Math.max(f[j][0], f[j][1]);
            f[u][1] += f[j][0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        for (int i = 1; i <= n; i++) {
            int hp = Integer.parseInt(in.readLine());
            happy[i] = hp;
        }
        for (int i = 0; i < n - 1; i++) {
            String[] row = in.readLine().split(" ");
            int a = Integer.parseInt(row[0]), b = Integer.parseInt(row[1]);
            // 标记a已经有父节点了
            hasFather[a] = true;
            // 添加一条b->a的边
            add(b, a);
        }
        int root = 1;
        // 寻找根节点，即没有父节点的节点
```

```
        while (hasFather[root]) {
            root++;
        }
        dfs(root);

        // 两种情况取较大者
        System.out.println(Math.max(f[root][0], f[root][1]));

        in.close();
    }
}
```

实现：Python：

```
import sys
sys.setrecursionlimit(6010)

N = 6010
happy, e, ne, idx, h, f, has_father = [0] * N, [0] * N, [0] * N, 0, [-1] * N, [[0] * 2 for _ in
range(N)], [False] * N


def add(fr: int, to: int) -> None:
    global idx
    e[idx] = to
    ne[idx] = h[fr]
    h[fr] = idx
    idx += 1


def dfs(u: int) -> None:
    f[u][1] = happy[u]
    i = h[u]
    while i != -1:
        j = e[i]
        dfs(j)
        f[u][0] += max(f[j][0], f[j][1])
        f[u][1] += f[j][0]
        i = ne[i]


if __name__ == '__main__':
    n = int(input())
    for k in range(1, n + 1):
        hp = int(input())
        happy[k] = hp
    for k in range(n - 1):
        row = input().split()
        a, b = int(row[0]), int(row[1])
        has_father[a] = True
        add(b, a)
    root = 1
    while has_father[root]:
        root += 1
    dfs(root)
    print(max(f[root][0], f[root][1]))
```

5.8 记忆化搜索

5.8.1 滑雪

问题描述： 给定一个`R`行`C`列的矩阵，表示一个矩形网格滑雪场。矩阵中第`i`行第`j`列的点表示滑雪场的第`i`行第`j`列区域的高度。一个人从滑雪场中的某个区域内出发，每次可以向上下左右任意一个方向滑动一个单位距离。当然，一个人能够滑倒某相邻区域的前提是该区域的高度低于自己目前所在区域的高度。现给出二维矩阵表示滑雪场各区域的高度，请找出在该滑雪场中能够完成的最长化学轨迹并输出一个最大长度。

核心思想： **递归+记忆化搜索+分情况讨论**

- 状态表示：`f[i][j]`表示所有从点(i,j)开始滑的路径，结果就是这些所有路径中的最大值。
- 状态计算：曲线救国法！我们从点(i,j)点可以向上下左右四个方向滑，可以滑的前提是这些方向上的高低严格小于(i,j)点的高度，则我们可以先去掉滑(i,j)点，因为所有后续的状态都会经历这个状态，后续状态就有`f[i - 1][j]`、`f[i + 1][j]`、`f[i][j - 1]`、`f[i][j + 1]`，我们只需要在(i,j)的基础上加上这些方向上可以滑行路径长度的最大值，就能得到结果，然后递归的处理每一个位置。

实现：C++：

```
#include<iostream>
#include<cstring>
#include<algorithm>

using namespace std;

const int N = 310;

int n, m;
```

```
int h[N][N];
int f[N][N];
int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};

int dp(int x, int y)
{
    int &v = f[x][y];
    if (v != -1)
    {
        return v;
    }
    v = 1;
    for (int i = 0; i < 4; i++)
    {
        int a = x + dx[i], b = y + dy[i];
        if (a >= 1 && a <= n && b >= 1 && b <= m && h[a][b] < h[x][y])
        {
            v = max(v, dp(a, b) + 1);
        }
    }
    return v;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            scanf("%d", &h[i][j]);
        }
    }
    memset(f, -1, sizeof f);
    int res = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            res = max(res, dp(i, j));
        }
    }
    printf("%d", res);

    return 0;
}
```

实现：Java：

```
package dp.memorysearch.skate;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 记忆化搜索+dp解决滑雪问题
 */
public class Skate {
    private static final int N = 310;
    /**
     * 存储每一个位置的高度
     */
    private static int[][] h;
    /**
     * 存储每一个位置的状态
     */
    private static int[][] f;
    /**
     * 上下左右方向标识
     */
    private static final int[][] DIR = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

    static {
        h = new int[N][N];
        f = new int[N][N];
        //  -1表示当前位置没有处理过
        for (int i = 0; i < N; i++) {
            Arrays.fill(f[i], -1);
        }
    }

    private static int dp(int x, int y, int n, int m) {
        if (f[x][y] != -1) {
```

```
            return f[x][y];
        }
        f[x][y] = 1;
        for (int[] d : DIR) {
            int a = x + d[0], b = y + d[1];
            if (a >= 1 && a <= n && b >= 1 && b <= m && h[a][b] < h[x][y]) {
                f[x][y] = Math.max(f[x][y], dp(a, b, n, m) + 1);
            }
        }
        return f[x][y];
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]), m = Integer.parseInt(s[1]);
        for (int i = 1; i <= n; i++) {
            String[] row = in.readLine().split(" ");
            for (int j = 1; j <= m; j++) {
                h[i][j] = Integer.parseInt(row[j - 1]);
            }
        }
        int res = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                res = Math.max(res, dp(i, j, n, m));
            }
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```
N = 310
h, f = [[0] * N for _ in range(N)], [[-1] * N for _ in range(N)]
DIR = [[1, 0], [0, 1], [-1, 0], [0, -1]]


def dp(x: int, y: int, r: int, c: int) -> int:
    if f[x][y] != -1:
        return f[x][y]
    f[x][y] = 1
    for d in DIR:
        a, b = x + d[0], y + d[1]
        if 1 <= a <= r and 1 <= b <= c and h[a][b] < h[x][y]:
            f[x][y] = max(f[x][y], dp(a, b, r, c) + 1)
    return f[x][y]


if __name__ == '__main__':
    s = input().split()
    n, m = int(s[0]), int(s[1])
    for i in range(1, n + 1):
        row = input().split()
        for j in range(1, m + 1):
            h[i][j] = int(row[j - 1])
    res = 0
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            res = max(res, dp(i, j, n, m))
    print(res)
```

6. 贪心

6.2 区间问题

6.2.1 区间选点

问题描述：给定`N`个闭区间$[a_i, b_i]$，请在数轴上选择尽量少的点，使得每个区间内至少包含一个选出的点，位于区间端点上的点也算作区间内，求选择点的最小数量。

核心思想：

- 将每个区间按右端点从小到大排序
- 从前往后一次枚举每一个区间：**如果当前区间已经被所选点覆盖，直接处理下一个区间，否则就选择当前区间的右端点作为候选点**

实现：C++：

```
#include<iostream>
#include<algorithm>
```

```

using namespace std;

const int N = 100010;

int n;
struct Range
{
    int l, r;
    bool operator< (const Range &W)const
    {
        return r < W.r;
    }
}range[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        int l, r;
        scanf("%d%d", &l, &r);
        range[i] = {l, r};
    }

    sort(range, range + n);

    int res = 0, ed = -2e9;
    for (int i = 0; i < n; i++)
    {
        if (range[i].l > ed)
        {
            res++;
            ed = range[i].r;
        }
    }
    printf("%d", res);

    return 0;
}
```

实现：Java：

```
package greedy.intervals.selectpointininterval;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Comparator;

/**
 * @author LBS59
 * @description 贪心思想求解区间选点问题
 */
public class SelectPointsInIntervals {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        int[][] ranges = new int[n][2];
        for (int i = 0; i < n; i++) {
            String[] row = in.readLine().split(" ");
            int l = Integer.parseInt(row[0]), r = Integer.parseInt(row[1]);
            ranges[i] = new int[]{l, r};
        }

        // 按区间右端点由小到大排序
        Arrays.sort(ranges, Comparator.comparingInt(o -> o[1]));

        int res = 0, cur = Integer.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            if (ranges[i][0] > cur) {
                res++;
                cur = ranges[i][1];
            }
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```python
if __name__ == '__main__':
    n = int(input())
    ranges = []
    for i in range(n):
        row = input().split()
        ranges.append([int(row[0]), int(row[1])])

    # 按区间右端点从小到大排序
    ranges.sort(key=lambda x: x[1])

    res, cur = 0, int(-2e9)
    for i in range(n):
        if ranges[i][0] > cur:
            res += 1
            cur = ranges[i][1]
    print(res)
```

6.2.2 区间分组

> **问题描述：** 给定`N`个闭区间$[a_i, b_i]$，将这些区间分成若干组，使得每组内部的区间两两之间(**包括端点**)没有交集，并使得组数尽可能小。
>
> **核心思想：**
>
> - 将所有区间按左端点由小到大排序；
> - 从前往后处理每一个区间，判断当前区间能够将其放到某个现有的组中，即当前区间的左端点是否大于当前组中区间最右端点。$l[i] > Max\_r$
>   - 如果不存在这样的组，则需要开一个新的组，并将当前区间放入；
>   - 如果存在这样的组，就将其放入，并更新当前组的最右端点Max_r

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>
#include<queue>

using namespace std;

const int N = 100010;

int n;
struct Range
{
    int l, r;
    bool operator< (const Range &W)const
    {
        return l < W.l;
    }
}range[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        int l, r;
        scanf("%d%d", &l, &r);
        range[i] = {l, r};
    }

    sort(range, range + n);

    priority_queue<int, vector<int>, greater<int>> heap;
    for (int i = 0; i < n; i++)
    {
        auto r = range[i];
        if (heap.empty() || heap.top() >= r.l)
        {
            heap.push(r.r);
        }
        else
        {
            int t = heap.top();
            heap.pop();
            heap.push(r.r);
        }
    }
    printf("%d", heap.size());
    return 0;
}
```

> **实现：Java：**

```java
package greedy.intervals.groupintervals;

```

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;

/**
 * @author LBS59
 * @description 贪心思想求解区间分组问题
 */
public class GroupingIntervals {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());
        int[][] ranges = new int[n][2];
        for (int i = 0; i < n; i++) {
            String[] row = in.readLine().split(" ");
            int l = Integer.parseInt(row[0]), r = Integer.parseInt(row[1]);
            ranges[i] = new int[]{l, r};
        }

        Arrays.sort(ranges, Comparator.comparingInt(o -> o[0]));

        PriorityQueue<Integer> heap = new PriorityQueue<>();
        for (int i = 0; i < n; i++) {
            if (heap.isEmpty() || heap.peek() >= ranges[i][0]) {
                heap.offer(ranges[i][1]);
            } else {
                heap.poll();
                heap.offer(ranges[i][1]);
            }
        }
        System.out.println(heap.size());

        in.close();
    }
}
```

实现：Python：

```python
import heapq

if __name__ == '__main__':
    n = int(input())
    ranges = []
    for i in range(n):
        row = input().split()
        ranges.append([int(row[0]), int(row[1])])

    # 按区间右端点从小到大排序
    ranges.sort(key=lambda x: x[0])
    # 创建一个小根堆
    heap = []
    for i in range(n):
        if len(heap) == 0 or heap[0] >= ranges[i][0]:
            heapq.heappush(heap, ranges[i][1])
        else:
            heapq.heappop(heap)
            heapq.heappush(heap, ranges[i][1])
    print(len(heap))
```

6.2.3 区间覆盖

问题描述：给定`N`个闭区间$[a_i, b_i]$以及一个线段区间`[s, t]`，选择尽量少的区间，将指定线段区间完全覆盖，求最少区间数，无法完全覆盖默认为-1。

核心思想：

- 将所有区间按照左端点从小到大排序；
- 从前往后一次枚举每一个区间，在所有能覆盖`s`点的区间中选择一个右端点最大的区间，选择完后，将`s`点更新为这个最大的右端点

实现：C++：

```cpp
#include<iostream>
#include<algorithm>
#include<queue>

using namespace std;

const int N = 100010;

int n;
struct Range
```

```
{
    int l, r;
    bool operator< (const Range &W)const
    {
        return l < W.l;
    }
}range[N];

int main()
{
    int st, ed;
    scanf("%d%d", &st, &ed);
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        int l, r;
        scanf("%d%d", &l, &r);
        range[i] = {l, r};
    }

    sort(range, range + n);

    int res = 0;
    bool success = false;
    for (int i = 0; i < n; i++)
    {
        int j = i, r = -2e9;
        while (j < n && range[j].l <= st)
        {
            r = max(r, range[j].r);
            j++;
        }

        if (r < st)
        {
            res = -1;
            break;
        }
        res++;
        if (r >= ed)
        {
            success = true;
            break;
        }
        st = r;
        i = j - 1;
    }
    if (!success)
    {
        res = -1;
    }
    printf("%d", res);

    return 0;
}
```

实现：Java：

```
package greedy.intervals.intervalconver;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Comparator;

/**
 * @author LBS59
 * @description 贪心思想求解区间覆盖问题
 */
public class CoverIntervals {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        String[] interval = in.readLine().split(" ");
        int st = Integer.parseInt(interval[0]), ed = Integer.parseInt(interval[1]);
        int n = Integer.parseInt(in.readLine());
        int[][] ranges = new int[n][2];
        for (int i = 0; i < n; i++) {
            String[] row = in.readLine().split(" ");
            int l = Integer.parseInt(row[0]), r = Integer.parseInt(row[1]);
            ranges[i] = new int[]{l, r};
        }

        Arrays.sort(ranges, Comparator.comparingInt(o -> o[0]));

```

```
        int res = 0;
        boolean flag = false;
        // 双指针法
        for (int i = 0; i < n; i++) {
            // 起始最大右端点为负无穷
            int j = i, r = Integer.MIN_VALUE;
            while (j < n && ranges[j][0] <= st) {
                // 寻找可以覆盖住st点的最后端点
                r = Math.max(r, ranges[j][1]);
                j++;
            }
            // 找到的最后端点没有覆盖住st点
            if (r < st) {
                res = -1;
                break;
            }
            res++;
            // 区间覆盖完成
            if (r >= ed) {
                flag = true;
                break;
            }
            // 更新起始点为选择的最大右端点，继续循环
            st = r;
            i = j - 1;
        }
        if (!flag) {
            res = -1;
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```
import heapq

if __name__ == '__main__':
    interval = input().split()
    st, ed = int(interval[0]), int(interval[1])
    n = int(input())
    ranges = []
    for i in range(n):
        row = input().split()
        ranges.append([int(row[0]), int(row[1])])

    # 按区间右端点从小到大排序
    ranges.sort(key=lambda x: x[0])

    res, flag = 0, False
    for i in range(n):
        j, r = i, int(-2e9)
        while j < n and ranges[j][0] <= st:
            r = max(r, ranges[j][1])
            j += 1
        if r < st:
            res = -1
            break
        res += 1
        if r >= ed:
            flag = True
            break
        st = r
        i = j - 1
    if not flag:
        res = -1
    print(res)
```

6.3 Huffman树

6.3.1 合并果子

问题描述： 简单描述，有`n`堆果子，要求将这些堆果子合成一堆，每一次合并，可以将两堆果子合并到一起，消耗的体力等于两堆果子的重量之和，可以看出，经过`n - 1`次合并之后，就只剩下一堆，合并玩消耗的体力等于每次合并消耗的体力之和。设计一种合并次序，使得消耗的体力最少。

核心思想：

- 每一次都挑选最小的两个堆合并。

实现：C++：

```
#include<iostream>
#include<algorithm>
```

```cpp
#include<queue>

using namespace std;

int main()
{
    int n;
    scanf("%d", &n);

    priority_queue<int, vector<int>, greater<int>> heap;
    while (n--)
    {
        int x;
        scanf("%d", &x);
        heap.push(x);
    }

    int res = 0;
    while (heap.size() > 1)
    {
        int a = heap.top();
        heap.pop();
        int b = heap.top();
        heap.pop();
        res += a + b;
        heap.push(a + b);
    }
    printf("%d", res);
    return 0;
}
```

实现：Java：

```java
package greedy.huffmantree.mergefruit;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.PriorityQueue;

/**
 * @author LBS59
 * @description Huffman树贪心例题，与合并石子有点类似，但是每次合并可以合并任意两堆
 */
public class FruitMerge {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine().trim());
        String[] row = in.readLine().trim().split(" ");

        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int i = 0; i < n; i++) {
            pq.offer(Integer.parseInt(row[i]));
        }

        int res = 0;
        // 贪心思想，每一次合并最小的两堆
        while (pq.size() > 1) {
            int a = pq.poll(), b = pq.poll();
            res += a + b;
            pq.offer(a + b);
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```python
import heapq

if __name__ == '__main__':
    n = int(input())
    row = input().split()
    heap = []
    for i in range(n):
        heapq.heappush(heap, int(row[i]))
    res = 0
    # 每次合并重量最小的两堆
    while len(heap) > 1:
        a, b = heapq.heappop(heap), heapq.heappop()
        res += a + b
        heapq.heappush(heap, a + b)
```

```
    print(res)
```

6.4 排序不等式

6.4.1 排队打水

> **问题描述：** 有`n`个人排队到一个水龙头处打水，第`i`个人装满水桶所需要的时间是t_i，请问如何安排他们的打水顺序才能使所有人的等待时间之和最小。
>
> **分析：** 假设`n`个人的打水时间分别为$t_1, t_2, \ldots, t_n$，则等待的总时间为$t = t1 \times (n-1) + t_2 \times (n-2) + \ldots + t_n \times 0$，可以看出，前面的需要等待的时间乘的基数最大，将打水时间短的人放置在前面就能使得`t`最小。
>
> **方法：**
>
> 按照从小到大的顺序排队，总时间最小。

实现：C++：

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

typedef long long LL;

const int N = 100010;

int n;
int t[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &t[i]);
    }
    sort(t, t + n);
    LL res = 0;
    for (int i = 0; i < n; i++)
    {
        res += t[i] * (n - i - 1);
    }

    printf("%lld\n", res);

    return 0;
}
```

实现：Java：

```java
package greedy.sortnonequalization;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 排序不等式例题，排队打水问题
 */
public class GetWater {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine().trim());
        String[] row = in.readLine().trim().split(" ");
        int[] t = new int[n];
        for (int i = 0; i < n; i++) {
            t[i] = Integer.parseInt(row[i]);
        }
        Arrays.sort(t);
        long res = 0;
        for (int i = 0; i < n; i++) {
            res += (long) t[i] * (n - i - 1);
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```python
if __name__ == '__main__':
    n = int(input())
    row = input().split()
    t = [int(x) for x in row]
    t.sort()
    res = 0
    for i in range(n):
        res += t[i] * (n - i - 1)
    print(res)
```

6.5 绝对值不等式

6.5.1 货仓选址

> **问题描述：** 在数轴上有`N`家商店，坐标分别为$A_1 - A_N$。现需要在数轴上建立一家货仓，每一次，从货仓到每家商店都要运送商品，求把货仓建在何处，使得货仓到每家店的距离之和最小。
>
> **分析：** 假设货仓的下标为x，并且每一个商店的坐标为$x_1, x_2, \dots x_n$，设$f(x)$表示所求距离之和，则
> $f(x) = |x_1 - x| + |x_2 - x| + \dots + |x_n - x|$
>
> $$f(x) = (|x1 - x| + |x_n - x|) + (|x_2 - x| + |x_{n-1} - x|) + \dots \tag{30}$$
>
> 这样原问题就转化为一个形如$|a - x| + |b - x|$的最小值问题，可以简单得知，可得最小值为$|a - b|$，即$|a - x| + |b - x| \geq |a - b|$，当且仅当`x`取`[a, b]`之间的位置时，等号成立。
>
> $$f(x) = |x_n - x_1| + |x_{n-1} - x_2| + \dots \tag{31}$$
>
> 则，当`x`取到上面绝对值每一项之间时，可以求得最小值，即`x`的位置就是所有元素的中位数(N为奇数)，或最中间两个数之间(N为偶数)。

> **实现：C++：**

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 100010;

int n;
int a[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    sort(a, a + n);
    int res = 0;
    for (int i = 0; i < n; i++)
    {
        res += abs(a[i] - a[n / 2]);
    }
    printf("%d\n", res);

    return 0;
}
```

> **实现：Java：**

```java
package greedy.absequalization;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

/**
 * @author LBS59
 * @description 绝对值不等式贪心问题：货仓选址
 */
public class WarehouseLocation {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine().trim());
        int[] arr = new int[n];
        String[] row = in.readLine().trim().split(" ");
        for (int i = 0; i < n; i++) {
            arr[i] = Integer.parseInt(row[i]);
        }
        Arrays.sort(arr);
        int res = 0;
        for (int i = 0; i < n; i++) {
```

```
            res += Math.abs(arr[i] - arr[n / 2]);
        }
        System.out.println(res);

        in.close();
    }
}
```

实现：Python：

```
if __name__ == '__main__':
    n = int(input())
    row = input().split()
    a = [int(x) for x in row]
    a.sort()
    res = 0
    for i in range(n):
        res += abs(a[i] - a[n // 2])
    print(res)
```

6.6 公式推导

6.6.1 耍杂技的牛

问题描述： N头奶牛(编号为$1...N$)，表演叠罗汉，奶牛们站在彼此的身上，形成一个高高的垂直堆叠。奶牛们正在试图找到自己在这个堆叠中应该所处的位置顺序。每一头奶牛都有自己的重量W_i以及自己的强壮程度S_i。一头牛支撑不住的可能性取决于它头上的所有牛的总重量(不包括自己)减去它的身体强壮程度的值，现在称该数值为风险值，风险值越大，这只牛撑不住的可能性越高。求处一种牛的排序，使得所有奶牛的风险值中的最大值尽可能的小。

分析： 按照$W_i + S_i$从小到大的顺序排，最大的危险系数一定是最小的。

实现：C++：

```
#include<iostream>
#include<algorithm>

using namespace std;

typedef pair<int, int> PII;

const int N = 50010;

int n;
PII cow[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        int w, s;
        scanf("%d%d", &w, &s);
        cow[i] = {w + s, w};
    }
    sort(cow, cow + n);
    int res = -2e9, sum = 0;
    for (int i = 0; i < n; i++)
    {
        int w = cow[i].second, s = cow[i].first - w;
        res = max(res, sum - s);
        sum += w;
    }
    printf("%d\n", res);

    return 0;
}
```

实现：Java：

```
package greedy.principle.cowact;

import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

/**
 * @author LBS59
 * @description
 */
public class CowActors {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] cow = new int[n][2];
        for (int i = 0; i < n; i++) {
```

```
            int w = sc.nextInt(), s = sc.nextInt();
            cow[i] = new int[] {w, s};
        }
        Arrays.sort(cow, Comparator.comparingInt(o -> o[0] + o[1]));
        // res统计承重最大值，sum表示当前承重
        int res = Integer.MIN_VALUE, sum = 0;
        for (int i = 0; i < n; i++) {
            res = Math.max(res, sum - cow[i][1]);
            sum += cow[i][0];
        }
        System.out.println(res);
    }
}
```

实现：Python：

```
if __name__ == '__main__':
    n = int(input())
    cow = []
    for i in range(n):
        row = input().split()
        cow.append([int(row[0]), int(row[1])])
    cow.sort(key=lambda x: x[0] + x[1])
    res, sum = int(-2e9), 0
    for i in range(n):
        res = max(res, sum - cow[i][1])
        sum += cow[i][0]
    print(res)
```

7.时空复杂度分析

不同数据范围下，代码的时间复杂度和算法该如何选择：

- $n \leq 30$，指数界别，`dfs`+剪枝，状态压缩`dp`；
- $n \leq 100 => O(n^3)$，`floyd`，`dp`；
- $n \leq 1000 => O(n^2), O(n^2logn)$，`dp`，二分；
- $n \leq 10000 => O(n \times \sqrt{n})$，块状链表；
- $n \leq 100000 => O(nlogn) =>$各种`sort`，线段树，树状数组，`set/map`，`heap`，`dijkstra+heap`，`spfa`，求凸包，求半平面交，二分；
- $n \leq 1000000 => O(n)$，以及常熟较小的$O(nlogn)$算法$=>$`hash`，双指针扫描，`kmp`，`AC`自动机，常熟比较小的$O(nlogn)$的做法：`sort`，树状数组，`heap`，`dijkstra+heap`，`spfa`；
- $n \leq 10000000 => O(n)$，双指针算法，`kmp`，`AC`自动机，线性筛素数；
- $n \leq 10^9 => O(\sqrt{n})$，判断质数；
- $n \leq 10^{18} => O(logn)$，最大公约数。