1: 双指针算法：（首先用暴力做法写出来，然后看单调性把算法的时间复杂度降低一维）

核心思想：

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

将这个朴素算法优化到 O(n)

模板：

```
for(int i = 0, j = 0; i < n; i++){
    while(j <= i && check(j, i)){
        j++;
        //每道题目的逻辑
    }
}
```

例题：

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

输入样例：

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

输出样例：

```
3
```

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] a = new int[100010];
        int[] s = new int[100010];
```

```
        for(int i = 0; i < n; i++){
            a[i] = sc.nextInt();
        }
        int res = 0;
        for(int i = 0, j = 0; i < n; i++){
            s[a[i]] ++;
            while(j <= i && s[a[i]] > 1){
                s[a[j]] --;
                j++;
            }
            res = Math.max(res, i - j + 1);
        }
        System.out.println(res);
    }
}
```

2： 位运算：

n 的二进制表示中第 k 位是几（k 的下标：个位是第 0 位，十位是第 1 位，，，，，依次类推）

基本思路：

1：先把第 k 位移到最后一位， n >> k;

2：看个位是几，x & 1

3：合并一起， n >> k & 1

4：lowbit（x）：返回 x 的最后一位 1（返回的是一个二进制的数）

x & - x

例题：

给定一个长度为 n 的数列， 请你求出数列中每个数的二进制表示中1 的个数

输入格式：

第一行包含整数 n

第二行包含 n 个整数，表示整个数列

输出格式：

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

```
import java.util.Scanner;
public class Main{
    private static int lowbit(int x){
```

```
        return x & -x;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        while(n -- > 0){
            int x = sc.nextInt();
            int res = 0;
            while(x != 0){
                x -= lowbit(x);
                res++;
            }
            System.out.printf("%d ", res);
        }
    }
}
```

3：离散化：

假设有一个保序数组 a [i]，数值很大，个数很少，需要把数组里的数值映射到以 0， 1，2， 3， 4为结尾的另一个数组 b [i]

1：数组中可能有重复元素，需要去重

2：如何算出 x 离散化之后的值， 二分法

例题：

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

输入格式：

第一行包含两个整数 m 和 n

接下来 n 行， 每行包含两个整数 x 和 c

再接下来 m 行， 每行包含两个整数 l 和 r

输出格式：

共 m 行， 每行输出一个询问中所求的区间内数字和

```
import java.util.*;
public class Main{
    public static void main(String[] args){
```

```java
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        int[] a = new int[300010];
        int[] s = new int[300010];
        List<Integer> alls = new ArrayList<>();
        List<Pair> add = new ArrayList<>();
        List<Pair> query = new ArrayList<>();
        for(int i = 0; i < n; i++){
            int x = sc.nextInt();
            int c = sc.nextInt();
            add.add(new Pair(x, c));
            alls.add(x);
        }
        for(int i = 0; i < m; i++){
            int l = sc.nextInt();
            int r = sc.nextInt();
            query.add(new Pair(l, r));
            alls.add(l);
            alls.add(r);
        }
        Collections.sort(alls);
        int unique = unique(alls);
        alls = alls.subList(0, unique);
        for(Pair item : add){
            int index = find(item.first, alls);
            a[index] += item.second;
        }
        for(int i = 1; i <= alls.size();i++){
            s[i] = s[i - 1] + a[i];
        }
        for(Pair item : query){
            int l = find(item.first, alls);
            int r = find(item.second, alls);
            System.out.println(s[r] - s[l - 1]);
        }
    }
    public static int unique(List<Integer> list){
        int j = 0;
        for(int i = 0; i < list.size(); i++){
```

```
            if(i == 0 || list.get(i) != list.get(i - 1)){
                list.set(j, list.get(i));
                j++;
            }
        }
        return j;
    }
    public static int find(int x, List<Integer> list){
        int l = 0;
        int r = list.size() - 1;
        while(l < r){
            int mid = (l + r) / 2;
            if(list.get(mid) >= x){
                r = mid;
            }else{
                l = mid + 1;
            }
        }
        return l + 1;
    }
}
        class Pair{
            int first;
            int second;
            public Pair(int x , int c){
                this.first = x;
                this.second = c;
            }
        }
```

4：链表：

单链表：邻接表（用来存储图和树）

双链表：优化某些问题

head : 一般指的是头结点的下标（为 - 1）

e [ ] : 表示当前结点的值

ne [ ] : 表示当前结点的 next 指针是多少

idx : 存储当前已经用到了哪个点

具体操作：

1：在链表的头结点中插入一个结点 x：

```
public static void insertHead(int x){
      e[idx] = x;
      ne[idx] = head;
      head = idx;
      idx++;
}
```

2：将新结点 x 插入到下标是 k 的点的后面：

```
public static void insert(int k, int x){
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
    idx ++;
}
```

3：将下标是 k 的点后面的点删掉：

```
public static void delete(int k){
    ne[k] = ne[ne[k]];
}
```

4：遍历链表：

```
for(int i = head; i != -1; i = ne[i]){
    System.out.printf("%d ", e[i]);
}
```

例题：

实现一个单链表， 链表初始为空， 支持三种操作：

1：向链表头插入一个数

2：删除第 k 个插入的数后面的数

3：在第 k 个插入的数后插入一个数

现在要对该链表进行 m 次操作，进行完所有操作后，从头到尾输出链表

输入格式：

第一行包含整数 m， 表示操作次数

接下来 m 行， 每行包含一个操作命令，操作命令可能为以下几种：

1：H x，表示向链表头插入一个数 x

2：D k，表示删除第 k 个插入的数后面的数（当 k 为 0 时，表示删除头结点）

3：I k x，表示在第 k 个插入的数后面插入一个数 x

输出格式：

共一行，将整个链表从头到尾输出

```java
import java.util.Scanner;
public class Main{
    public static int[] e = new int[100010];
    public static int[] ne = new int[100010];
    public static int idx, head;
    public static void init(){
        idx = 0;
        head = -1;
    }
    public static void insertHead(int x){
        e[idx] = x;
        ne[idx] = head;
        head = idx;
        idx ++;
    }
    public static void insert(int k, int x){
        e[idx] = x;
        ne[idx] = ne[k];
        ne[k] = idx;
        idx ++;
    }
    public static void delete(int k){
        ne[k] = ne[ne[k]];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        init();
        while(m -- > 0){
```

```
            String str = sc.next();
            char op = str.charAt(0);
            if(op == 'H'){
                int x = sc.nextInt();
                insertHead(x);
            }else if(op == 'D'){
                int k = sc.nextInt();
                if(k == 0){
                    head = ne[head];
                }else{
                    delete(k - 1);
                }
            }else{
                int k = sc.nextInt();
                int x = sc.nextInt();
                insert(k - 1, x);
            }
        }
        for(int i = head; i != -1; i = ne[i]){
            System.out.printf("%d ", e[i]);
        }
    }
}
```

双链表：

首先用 r [ ]， l [ ] 两个数组表示双链表的头结点和尾结点，初始化为 r [0] = 1； l [1] = 0; r 代表右指针， l 代表左指针

具体操作：

在下标为 k 的结点右边插入一个数 x：

```
public static void add(int k, int x){
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx;
}
```

删除结点 k：

```
public static void remove(int k){
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

遍历双链表：

```
for(int i = r[0]; i != 1; i = r[i]){
    System.out.printf("%d ", e[i]);
}
```

例题：

实现一个双链表，双链表初始为空，支持5种操作：

1：在最左侧插入一个数

2：在最右侧插入一个数

3：将第 k 个插入的数删除

4：在第 k 个插入的数左侧插入一个数

5：在第 k 个插入的数右侧插入一个数

现在要对该链表进行 M 次操作，进行完所有操作后，从左到右输出整个链表

输入格式：

第一行包含整数 M ， 表示操作次数

接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1：L，x ，表示在链表的最左端插入数 x

2：R，x，表示在链表的最右端插入数 x

3：D，k，表示将插入的第 k 个数删除

4：IL，k，x，表示在第 k 个插入的数左侧插入一个数

5：IR，k，x，表示在第 k 个插入的数右侧插入一个数

输出格式：

共一行，将整个链表从左到右输出

```
import java.util.Scanner;
public class Main{
```

```java
    public static int idx;
    public static int[] e = new int[100010];
    public static int[] l = new int[100010];
    public static int[] r = new int[100010];
    public static void init(){
        idx = 2;
        r[0] = 1;
        l[1] = 0;
    }
    public static void add(int k, int x){
        e[idx] = x;
        r[idx] = r[k];
        l[idx] = k;
        l[r[k]] = idx;
        r[k] = idx;
        idx ++;
    }
    public static void delete(int k){
        r[l[k]] = r[k];
        l[r[k]] = l[k];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        init();
        while(m -- > 0){
            String str = sc.next();
            char op = str.charAt(0);
            if(op == 'L'){
                int x = sc.nextInt();
                add(0, x);
            }else if(op == 'R'){
                int x = sc.nextInt();
                add(l[1], x);
            }else if(op == 'D'){
                int k = sc.nextInt();
                delete(k + 1);
            }else if(str.equals("IR")){
                int k = sc.nextInt();
                int x = sc.nextInt();
```

```
                add(k + 1, x);
            }else{
                int k = sc.nextInt();
                int x = sc.nextInt();
                add(l[k + 1], x);
            }
        }
        for(int i = r[0]; i != 1; i = r[i]){
            System.out.printf("%d ", e[i]);
        }
    }
}
```

5：栈：（先进后出）

栈是一种 "先进先出" 的数据结构，可以用数组stk[ ]来模拟栈，tt为初始的栈顶指针

具体操作：

向栈顶插入一个元素 x：

```
stk[++ tt] = x;
```

从栈顶弹出一个元素：

```
tt--;
```

判断栈顶元素是否为空：

```
if(tt > 0){
    not empty;
}else{
    empty;
}
```

查询栈顶元素：

```
stk[tt];
```

例题：

实现一个栈，栈初始为空，支持四种操作：

1：push x ----向栈顶插入一个数 x

2：pop --------从栈顶弹出一个数

3：empty -----判断栈是否为空

4：query ------查询栈顶元素

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

输入格式：

第一行包含整数 M，表示操作次数

接下来 M 行，每行包含一个操作命令，操作命令为 push x, pop, empty, query 中的一种

输出格式：

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

输入样例：

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

输出样例：

```
5
5
YES
4
NO
```

```
import java.util.Scanner;
public class Main{
```

```
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        int tt = 0;
        int[] stk = new int[100010];
        while(m -- > 0){
            String str = sc.next();
            if(str.equals("push")){
                int x = sc.nextInt();
                stk[++ tt] = x;
            }else if(str.equals("pop")){
                tt --;
            }else if(str.equals("empty")){
                if(tt > 0){
                    System.out.println("NO");
                }else{
                    System.out.println("YES");
                }
            }else{
                System.out.println(stk[tt]);
            }
        }
    }
}
```

6：队列：（先进先出）：

队列是一种先进先出的数据结构，也可以用数组q [ ]来模拟队列，hh是队列的头部指针，tt是队列的尾部指针，默认初始化为 -1

具体操作：

1：在队尾插入元素，在队头弹出元素

```
q[++ tt] = x;
hh++;
```

2：判断队列是否为空：

```
if(hh <= tt){
    not empty;
```

```
}else{
    empty;
}
```

3：取出队头元素：

```
q[hh];
```

例题：

实现一个队列，队列初始为空，支持四种操作

1：push x ------向队尾插入一个数 x

2：pop ----------从队头弹出一个数

3：empty--------判断队列是否为空

4：query---------查询队头元素

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

输入格式：

第一行包含整数 M ，表示操作次数

接下来 M 行，每行包含一个操作命令，操作命令为 push x ，pop，empty，query中的一种

输出格式：

对于每个empty 和 query 操作都要输出一个查询结果，每个结果占一行，其中 empty 操作的查询结果为

YES或 NO，query 操作的查询结果为一个整数，表示队头元素的值

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int[] q = new int[100010];
        int hh = 0;
        int tt = -1;
        int m = sc.nextInt();
        while(m -- > 0){
            String str = sc.next();
            if(str.equals("push")){
                int x = sc.nextInt();
```

```
                q[++ tt] = x;
            }else if(str.equals("pop")){
                hh++;
            }else if(str.equals("empty")){
                if(hh <= tt){
                    System.out.println("NO");
                }else{
                    System.out.println("YES");
                }
            }else{
                System.out.println(q[hh]);
            }
        }
    }
}
```

7：单调栈：

常用题型：找出每个数左边离它最近的比它小 / 大的数

以 3 ，4，2， 7， 9为例，过程如下：

例题：

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

输入格式：

第一行包含整数 n , 表示数列长度

第二行包含 n 个整数，表示整数数列

输出格式：

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

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int tt = 0;
        int[] stk = new int[100010];
        for(int i = 0; i < n; i++){
            int x = sc.nextInt();
```

```
            while(tt != 0 && stk[tt] >= x){
                tt--;
            }
            if(tt != 0){
                System.out.print(stk[tt] + " ");
            }else{
                System.out.print(-1 + " ");
            }
            stk[++ tt] = x;
        }
    }
}
```

8：单调队列：

常见类型：找出滑动窗口中的最大值 / 最小值

思路：

最小值和最大值分开来做，两个 for 循环完全类似，都做以下四步：

1：解决队首已经出窗口的问题

2：解决队尾与当前元素a[i]不满足单调性的问题

3：将当前元素下标加入队尾

4：如果满足条件则输出结果

需要注意的细节：

1：上面四个步骤中一定要先3后4，因为有可能输出的正是新加入的那个元素

2：队列中存的是原数组的下标，取值时要再套一层，a[q[ ]]

3：算最大值前注意将hh和tt重置

4：hh从0开始，数组下标也要从0开始

例题：

给定一个大小为 n <= 1000000的数组，有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边，只能在窗口中看到 k 个数字

每次滑动窗口向右移动一个位置

你的任务是确定滑动窗口位于每个位置时，窗口中的最小值和最大值

输入格式：

输入包含两行

第一行包含两个整数 n 和 k ，分别代表数组长度和滑动窗口长度，第二行有 n 个整数，代表数组的具体数值

输出格式：

输出包含两个

第一行输出，从左至右，每个位置滑动窗口中的最小值

第二行输出，从左至右，每个位置滑动窗口中的最大值

```java
import java.io.*;
public class Main{
    public static int n, k;
    public static int hh, tt;
    public static int[] a = new int[1000010];
    public static int[] q = new int[1000010];
    public static void main(String[] args) throws IOException{
        BufferedReader bw = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out));
        String[] st = bw.readLine().split(" ");
        int n = Integer.parseInt(st[0]);
        int k = Integer.parseInt(st[1]);
        String[] str = bw.readLine().split(" ");
        for(int i = 0; i < n; i++){
            a[i] = Integer.parseInt(str[i]);
        }
        hh = 0;
        tt = -1;
        for(int i = 0; i < n; i++){
            if(hh <= tt && q[hh] < i - k + 1){
                hh ++;
            }
            while(hh <= tt && a[i] <= a[q[tt]]){
                tt --;
            }
            q[++ tt] = i;
            if(i >= k - 1){
                pw.print(a[q[hh]] + " ");
            }
        }
        pw.println();
        hh = 0;
        tt = -1;
```

```
        for(int i = 0; i < n; i++){
            if(hh <= tt && q[hh] < i - k + 1){
                hh ++;
            }
            while(hh <= tt && a[i] >= a[q[tt]]){
                tt --;
            }
            q[++ tt] = i;
            if(i >= k - 1){
                pw.print(a[q[hh]] + " ");
            }
        }
        pw.flush();
    }
}
```

9：Kmp 算法：

Kmp 全称为Knuth Morris Pratt算法，三个单词分别是三个作者的名字。Kmp 是一种高效的字符串匹配算法，用来在主字符串中查找模式字符串的位置(比如在 “ hello,world ” 主串中查找 “ world ” 模式串的位置)

Kmp 算法的高效性：

高效性是通过和其他字符串搜索算法对比得到的，在这里拿暴力朴素算法做一下对比。主要的思想是在主串的[0, n-m]区间内依次截取长度为m的子串，看子串是否和模式串一样(n是主串的长度，m是子串的长度)。举个例子如下：给定文本串S， “aaaababaaba” ，和模式串P， “ababa” ，现在要拿模式串P去跟文本串S匹配，整个过程如下所示：

暴力算法的时间复杂度是O(N*N)，存在很大优化空间。当模式串和主串匹配时，遇到模式串中某个字符不能匹配的情况，对于模式串中已经匹配过的那些字符，如果我们能找到一些规律，将模式串多往后移动几位，而不是像暴力算法算法一样，每次把模式串移动一位，就可以提高算法的效率。kmp算法给我们提供的思路是：对于模式串，将每一个字符在匹配失败时可以向后移动的最大距离保存在一个prefix数组中，有的也叫next数组。这样当匹配失败时就可以按照prefix数组中保存的数字向后多移动几位，从而提高算法的效率。

前缀数组 prefix ：

它记录着字符串匹配过程中失配情况下可以向后多跳几个字符，其实也是子串的前缀和后缀相同的最长长度

算法思路：

就是在暴力的算法的基础上，在匹配失败的时候往后多跳几位，而跳几位保存在前缀数组中。举个例子，下图中已经写好了总串s和模式串p,模式串的前缀数组为[0,0,1,2,3]，且所以下标都是从1开始。看图中当i=8，j= 4时s[i] != p[j + 1]，即将要匹配失败了，图中红色圈住的是子串的后缀。黄圈圈住的是前缀。蓝色圈圈住的是已经和后缀匹配过的部分，那么下一次将模式串后移prefix[j]=2位时，原来的前缀正好对着蓝色圈圈部分，因为前缀=后缀=蓝色圈圈部分，所以移动后的橙色部分就不用再判断了

再用上一个双指针算法思路。i遍历总串s，j遍历模式串p，判断s[i] 和 p[j + 1]是否匹配。不匹配就将 j 重置为前缀数组中prefix[j]的值。匹配的话 j 往后移动一位。当匹配了n个字符后即代表完全匹配。此时答案即为 i- n，如果要继续搜索，要将j再置为prefix[j]
为了方便写代码所有数组的下标都从 1 开始

```
// kmp匹配
for (int i = 1, j = 0; i <= m; i++) {
    while (j > 0 && s[i] != p[j + 1]) {
        j = prefix[j]; // s[i] != p[j + 1]即不匹配，则往后移动
    }
    if (s[i] == p[j + 1])
        j++; // 匹配时将j++进行下一个字符得匹配
    if (j == n) { // 匹配了n字符了即代表完全匹配了
        System.out.print(i - n + " ");
        j = prefix[j]; // 完全匹配后继续搜索
    }
}
```

如何求前缀数组：
这里使用的方式是和上面的匹配过程类似的方法，也就是将前缀看作模式串，在p中匹配他。也就是字符串p自己找自己的匹配串
例题：
给定一个字符串 S，以及一个模式串 P， 所有字符串中只包含大小写英文字母以及阿拉伯数字
模式串 P 在字符串 S 中多次作为子串出现
求出模式串 P 在字符串 S 中所有出现的位置的起始下标
输入格式：
第一行输入整数 N ， 表示字符串 P 的长度
第二行输入字符串 P

第三行输入整数 M，表示字符串 S 的长度

第四行输入字符串 S

输出格式：

共一行，输出所有出现位置的起始下标（下标从0开始计数），整数之间用空格隔开

```java
import java.io.*;
public class Main{
    public static void main(String[] args) throws IOException{
        BufferedReader bw = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter pw = new BufferedWriter(new OutputStreamWriter(System.out));
        int n = Integer.parseInt(bw.readLine());
        String P = bw.readLine();
        char[] p = new char[100010];
        for(int i = 1; i <= n; i++){
            p[i] = P.charAt(i - 1);
        }
        int m = Integer.parseInt(bw.readLine());
        String S = bw.readLine();
        char[] s = new char[1000010];
        for(int i = 1; i <= m; i++){
            s[i] = S.charAt(i - 1);
        }
        int[] next = new int[100010];
        for(int i = 2, j = 0; i <= n; i++){
            while(j > 0 && p[i] != p[j + 1]){
                j = next[j];
            }
            if(p[i] == p[j + 1]){
                j++;
            }
            next[i] = j;
        }
        for(int i = 1, j = 0; i <= m; i++){
            while(j > 0 && s[i] != p[j + 1]){
                j = next[j];
            }
            if(s[i] == p[j + 1]){
                j++;
```

```
            }
            if(j == n){
                pw.write((i - n) + " ");
                j = next[j];
            }
        }
        pw.flush();
        pw.close();
        bw.close();
    }
}
```