# 数学基础

## 取模运算

取模运算在C++中使用的符号是%。a mod p = b表示a除以p的余数是b。

根据余数的定义，任意一个整数a，对于一个确定的模数p，可以转换成：

$$a = kp + b$$
$$0 < b < p$$
$$k \in Z$$

k等于a除p向下取整

取模运算具有部分和四则运算类似的性质（除法例外）

较为常用的如下

$$(a \pm b)\%p = (a\%p \pm b\%p)\%p$$
$$(a * b)\%p = (a\%p * b\%p)\%p$$

注意取模运算对除法不具有特别的性质

$$\frac{a}{b}\%p \neq \frac{a\%p}{b\%p}\%p$$

为了表示分数，根据费马小定理有

$$\frac{1}{a} \equiv a^{p-2}$$

通过费马小定理，可以在模p的意义下，使用整数表示分数。部分题目会要求使用该方法来避免精度误差。

取模运算更多的用途是避免结果溢出。

涉及取模运算的题目中，往往中间过程的结果会超出int或long long的范围。需要使用取模来避免出现错误

## 位运算

常用的位运算有按位与、按位或、取反、按位异或、右移和左移。

位运算一般用于节省时间和简化状态的改变。

```
//位运算结果示例
#include<bits/stdc++.h>

using namespace std;

int main()  {
    clock_t s, t;
    s = clock();
    int a = 100 , b, c;
    for (int i = 1;i <= 1000000000; i++)  {
        c = a / 2;
    }
```

```
    t = clock();
    cout << t - s << endl;
    s = clock();
    for (int i = 1;i <= 1000000000; i++)  {
        c = a >> 1;
    }
    t = clock();
    cout << t - s << endl;

    a = 77;
    //a的二进制是1001101
    b = 6;
    //b的二进制为110
    cout << "a<<1 = " << (a<<1) << endl;
    cout << "a>>1 = " << (a >> 1) << endl;
    cout << "a & b = " << (a & b) << endl;
    cout << "a ^ b = " << (a ^ b) << endl;
    cout << "a | b = " << (a | b) << endl;
    return 0;
}
```

## 快速幂

快速幂用于快速计算幂次的结果。

直接使用循环计算幂次，需要的时间与幂次相关。当求的幂次非常大时会导致TLE。

可以发现直接使用循环计算时，会进行许多不必要的计算。

例如求$a^{1000}$,可以知道$a^{1000} = a^{500} * a^{500}$。如果已经使用循环计算出$a^{500}$的结果，则后一个$a^{500}$可以直接使用之前的结果。但在朴素的方法中，会再次计算$a^{500}$。

因此，可以考试使用分治的思想优化幂运算。

对于$a^b$。当b是偶数时，可以化成$(a^{\frac{b}{2}})^2$。当b时奇数时，可以提取出一个a，化成$a(a^{\lfloor\frac{b}{2}\rfloor})^2$

每次b都会除2，时间复杂度是O(logb)

```
ll Pow1(ll v, ll u)  {
    if (!u) return 1;
    if (u % 2) return v * Pow1(v, u - 1) % mod;
    return Pow1(v * v % mod, u / 2);
}

ll Pow2(ll v, ll u)  {
    int res = 1;
    while (u)  {
        if (u % 2) res = res * v % mod;
        v = v * v % mod;
        u >>= 1;
    }
    return res;
}
```

快速幂可以使用在矩阵上。

但是在矩阵上使用快速幂时，若使用递归的写法，由于每次递归都会定义一个新的矩阵，可能会出现栈溢出的情况，导致RE。推荐使用非递归版本。

# 线性结构

## 前缀和与差分

前缀和是一种预处理操作，能大大降低查询的时间复杂度

当涉及快速求取某一区域和时，可考虑使用前缀和进行快速计算

常用的前缀和分为一维前缀和和二维前缀和

### 一维前缀和

对于一维数组a，其前缀和数组sum定义为$sum_i$是数组a中第一个元素到第i个元素的和。

根据定义，$sum_i = sum_{i-1} + a_i$

那么，可以在O（n）的时间内得到sum数组。

使用sum数组可以查询a数组的区间和。

a数组中L到R的区间和可以表示成$sum_R - sum_{L-1}$

### 二维前缀和

对于二维数组a，前缀和数组sum定义为$sum_{i,j}$，表示数组a中以(1,1)为左上角、(i, j)为右下角的矩阵中元素的和

同一维前缀和，需要预处理出sum数组。

使用sum数组可以查询a数组的区域和

虽然前缀和可以优化查询速度，但是使用前缀和需要进行预处理。若题目包含修改操作，那么每次修改后都需要重新进行预处理。

因此前缀和只适用于没有或只有几次修改的情况。

### 差分

差分是一种与前缀和相关的构造方法。

对于原数组A，数组范围是1~n。要构造一个差分数组B

$$B[1] = A[1]$$
$$B[i] = A[i] - A[i-1]$$

A中的元素可以用B的前缀和表示。

数组A的第i个元素，等于数组B中前i个元素的和。

对于数组B的单点修改，相当于对数组A进行区间加

通俗的解释是，数组A的第i个元素是数组B前i个元素的和。数组B前i个元素中，只要有一个元素加了C，那么数组A的第i个元素就会加上C。所以对数组B的第i个元素加C，会导致数组A中第i、i+1、i+2..n个元素都加上C。

## 例1

该题是对差分数组最基础的应用

题意：。。。

思路：。。。

值得注意的是，差分数组虽然能对原数组快速的进行区间修改，但是想要从差分数组得到新数组需要进行O(n)的处理。所以差分数组只适用多次修改，只有一次或几次查询的情况。

并且，由于在差分数组构造时，用到了前缀和。差分数组只能处理区间加或者区间减的情况。

# 尺取法

尺取法又称双指针法。是一种数组上的常见操作。

### 经典例题：

给出一个长度为n的正整数数组，求长度最小的连续区间，使得所选区间和大于S。

具体流程：

维护双指针L，R，分别表示所选区间的左右端点，初始情况下L=1,R=1,sum=a[1]

考虑每个以L为做短点的区间是否满足要求。

若不满足要求，说明需要增大区间来增加区间和，将R加一。

若满足要求，用当前区间长度更新答案。此时，如果保持L不变，R增加的话，区间和将会增加，但是区间长度也会增加，并且肯定大于当前答案。因此R应该保持不变，考虑将L增加的情况。

### 代码实现

尺取法的过程中，左端点L和右端点R都会从1增加至n，可以考虑枚举其中一个端点从1增加至n的过程，并在枚举中改变另一个端点的值来实现。

个人习惯枚举右端点，考虑左端点的变化。

为了方便写代码，假设初始情况R=0，L=1，sum=0

时间复杂度：L和R都最多只会从1增加至n，因此时间复杂度是O(n)

```
//http://poj.org/problem?id=3061
//尺取法
include<iostream>

using namespace std;

long long a[100010], n;
long long Sum, s;

void work()  {
    cin >> n >> s;
    for (int i = 1;i <= n; i++) cin >> a[i];
    int Ans = n + 1, l = 1;
    Sum = 0;
    for (int i = 1;i <= n; i++)  {
        Sum += a[i];
        while (Sum >= s)  {
            Ans = min(Ans, i - l + 1);
```

```
            Sum -= a[l];
            l++;
        }
    }
    if (Ans == n + 1) cout << "0\n";
    else cout << Ans << '\n';
}

int main()  {
    ios::sync_with_stdio(0);
    int T;
    cin >> T;
    for (;T--;)
        work();
    return 0;
}
```

### 简要证明

根据算法流程，左端点L会从1增加至n，每个左端点都会找到一个R，使得区间[L,R]刚好满足条件，或者R=n依旧不满足条件。

因此答案区间一定会在上述尺取过程中出现。

### 总结

尺取法的过程是使用两个指针向同个方向进行扫描

使用尺取法的条件：求解答案是一个连续区间。对于一个左端点L， [L,R] 的区间是否满足条件是单调的。单调指的是，L不变，在R慢慢增加的过程中，当R较小时，不满足条件，当R为某个值时恰好满足条件，并且在此之后，R一直满足条件

## 例2

题意：。。。

思路：。。。

如何判断[L,R]满足要求：对单种字符进行考虑，在平衡字符串中，每种字符数量是n/4，在选择区间中，每种字符的数量可以是任意的，则只需要判断不可变化的字符中，每种字符数量是否满足要求。只要每种字符不变的数量小于或等于n/4即可。

代码实现：

```
//https://vjudge.net/problem/Gym-270737B
//平衡字符串
#include<iostream>

using namespace std;
string s;
int n, a[100100], Sum[4], Cnt[4];

int main()  {
    ios::sync_with_stdio(0);
    cin >> s;
    n = s.length();
    for (int i = 0;i < n; i++)  {
        if (s[i] == 'Q') Sum[0]++, a[i] = 0;
```

```
        if (s[i] == 'W') Sum[1]++, a[i] = 1;
        if (s[i] == 'E') Sum[2]++, a[i] = 2;
        if (s[i] == 'R') Sum[3]++, a[i] = 3;
    }
    int Ans = n, flag, l = 0;
    for (int i = 0;i < n; i++)  {
        Cnt[a[i]]++;
        while (l <= i + 1)  {
            flag = 1;
            for (int j = 0;j < 4; j++) if (Sum[j] - Cnt[j] > n/ 4) flag = 0;
            if (!flag) break;
            Ans = min(Ans, i - l + 1);
            Cnt[a[l]]--;
            l++;
        }
    }
    cout << Ans;

    return 0;
}
```

# 单调栈

单调栈=单调+栈

单调性:

单调递增：数组中每个元素严格大于上一个元素

单调递减：数组中每个元素严格小于上一个元素

单调非减：数组中每个元素大于或等于上一个元素

单调非增：数组中每个元素小于或等于上一个元素

栈：一种线性结构，支持入栈和出栈操作。满足先进后出操作

单调栈：栈内元素自栈顶到栈底满足单调性

## 单调递增栈

当想要把一个元素加入单调递增栈中，为了满足栈中元素递增的规则，需要使得栈顶大于当前元素。所以需要将小于等于当前元素的栈顶元素出栈，直到栈顶元素满足要求或栈为空。否则会破坏栈中元素单调性。

## 单调栈的作用

线性复杂度：考虑数列中的每一个数，都会被加入栈中。所以最多有n个数会被弹出栈。时间复杂度是O（n）

单调递增栈可以找到原数组中，每个元素往左/往右第一个比当前元素大的元素

简要证明：从左往右将数组的元素加入栈中，为了满足单调性，会将左边小于等于当前元素、并且在栈顶的元素弹出，直到遇见一个比它大的栈顶元素或者栈为空。所以，在当前元素加入栈前，新的栈顶元素比当前元素大。是当前元素左边第一个大于它的元素。

单调递减栈可以找到原数组中，每个元素往左/往右第一个比当前元素小的元素

## 例3

题意：给一个直方图，求直方图中最大的矩形面积。

思路：

最直观的思路是枚举矩形的左右端点，依照题意，矩形的高不能超过左右端点的最小值。因此，左右端点确定的条件下，最大矩形的高是左右端点内的最小值。

但是上述方法需要枚举左端点和右端点，时间复杂度是O(n^2)。

转换下枚举的内容，如果以一个点的高作为备选矩阵的高，那么要得到当前情况下最大的矩形，需要左端点尽量靠左，右端点尽量靠右。

因此左端点可以确定为往左数第一个小于此高度点右边的点，右端点可以确定为往右数第一个小于此高度点左边的点。

使用单调递减栈查询以每个点为高的左右端点

```

```

# 单调队列

单调队列要求队列里的元素满足单调性

队列：线性数据结构，可以从队尾入队、出队，从队头出队。满足先进先出。

### 单调递增队列

当要把一个元素加入队列中时，为了队列中元素满足单调性，需要使得队尾元素小于当前元素。在加入队列前，要把所有大于等于当前元素的队尾元素弹出队列。直到满足单调性或者队列为空。

### 单调队列的特点

与单调栈类似，需要队列（栈）中满足单调性。

区别在于，单调栈只维护一端(栈顶), 而单调队列可以维护两端(队首和队尾)

单调栈通常维护 **全局** 的单调性, 而单调队列通常维护 **局部** 的单调性

单调栈满足先进后出，先进入的元素可能一直在栈中。维护区间的左端点不会变化。

单调队列满足先进先出。从队头出队的元素可以看成维护区间左端点右移。

## 例4

题意：给出一个长度为n的数组，有一个长度为k的窗口从左向右滑动。想知道每次窗口中数的最小值和最大值。

```

```