T1序列

首先可以发现，如果我们换个角度看 b_i，看作 $a_i - a_1 = cnt_i * m$

那么 cnt_i 实际上是一个固定的序列，且 $cnt_1 = 0$

很容易可以想到只要暴力枚举 a_i，即可求出对应的 m，然后再枚举一遍 $check$ 这个 m 符合多少数字即可，但是这样的暴力做法复杂度有 $O(n^2)$

那么考虑如何优化，显然这里我们可以直接用一个 map 统计对于某个 m 的答案

也就是求出每个 $m_i = (a_i - a_1)/b_i$，只要统计有多少个相同的 m_i 就是最大相似度，

但是这里要注意一些细节问题，因为这里的除法很容易出现 0

所以要进行分类讨论：

设 ans_x 表示有多少个 $m_i = x$

1. 若 $cnt[i] == 0$，则判断 $a[i] == a[1]$，如果满足则说明此时任何 m 都满足 $b[i] = a[i]$，用 $tag++$ 来记录有多少个这样的数字，否则对答案无贡献
2. 若 $cnt[i] != 0$，则可以正常计算 $k = (a[i] - a[1])/cnt[i]$，若无法整除则对答案无贡献，若能整除则统计 $num[k]++$

最后求最大的 $num[k]$ 再加上 tag 即为答案

标程

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
long long a[N], b[N];
long long n, tag = 1, ans, pos;
map<long long, long long> num;
int main(){
    ios::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);
    cin >> n;
    for(int i = 1; i <= n; i++){
        cin >> a[i];
    }
    for(int i = 2; i <= n; i++){
        if(a[i] > a[i - 1]){
            b[i] = b[i - 1] + 1;
        } else if(a[i] == a[i-1]){
            b[i] = b[i - 1];
        } else {
            b[i] = b[i - 1] - 1;
        }
        if (b[i] == 0){
            if (a[i] == a[1]){
                tag++;
            }
        }
    }
    for (int i = 2; i <= n; ++i){
        if (b[i] != 0){
            if ((a[i] - a[1]) % b[i] == 0){
                long long k = (a[i] - a[1]) / b[i];
                num[k]++;
            }
        }
    }
    for (auto it:num){
        if (it.second > ans){
            ans = it.second;
            pos = it.first;
```

```
        }
    }
    cout << ans + tag << '\n' << pos << '\n' ;
    return 0;
}
```

T2快乐

30分做法

考虑暴力搜索，每次交换相邻的一对字符，找出其中所有好的字符串，求出最少的交换次数，可以通过 $|S| \leq 12$。

另外30分

考虑有一个字符占了一半以上，假设为 A，分为两种情况。

1. $|S|$ 为偶数，此时一定无解。
2. $|S|$ 为奇数，如果 A 字符个数恰好为 $\frac{|S|+1}{2}$，那么 A 字符一定占据所有奇数位置，将它们暴力交换到各自的位置上统计交换数即可，否则一定无解。

100分做法

考虑 dp。

有一个朴素状态 $dph[i][j][k][p][0/1/2]$ 表示前 i 个位置上 A, C, K 的个数分别为 j, k, p，且第 i 位上的数字为 $A/C/K$ 的最小花费，显然，$dph[i][j][k][p]$ 是由 $dph[i-1][j-1][k][p], dph[i-1][j][k-1][p]$ 和 $dph[i-1][j][k][p-1]$ 转移过来，因为要求不能有相邻的同一种数字，所以转移时 $dph[i]$ 和 $dph[i-1]$ 的最后一维 $0/1/2$ 不能相同。

然后就是转移代价了。
首先有一个比较显然的性质，即相同数字的相对位置不会改变。
假设当前的转移是
$dph[i-1][j-1][k][p][1/2] \to dph[i][j][k][p][0]$
那么第 i 位上填的数字为 A, 该 A 显然由原序列上第 j 个 A 移动过来，我们可以计算出当前该

A 的位置 pos, 花费代价即为 $pos - i$。
k, p 的转移类似。 但是现在这个复杂度是 $O(n^4)$ 的， 需要优化。

考虑优化，因为 $j + k + p = i$, 我们可以压缩一维， 变成 $dph[i][j][k][0/1/2]$, 表示前 i 个位置 $0, 1, 2$ 的个数分别为 $j, k, i - j - k$, 第 i 位 为 $0/1/2$ 的最小代价，转移类似。

解释为什么要最后要除以 2。

因为考虑 AC 变到 CA，会发现 $i - pos_i$ 的绝对值值是 2，就是可以理解为每当一个字母往前动了一格，就会有另一个被换的字母往后动了一格

即有一个 $i > pos_i$ 的数绝对值减了 1，有一个 $i < pos_i$ 的数绝对值减了 1，你不可能换两个 $i > pos_i$ 或两个 $i < pos_i$ 的数，这是不优的，因为总能找到两个相邻的数，一个 $i > pos_i$，一个 $i < pos_i$ 把他们交换掉。

标程

```cpp
#include<bits/stdc++.h>
using namespace std;
int t[3][405], n, c[3], dp[205][205][205][3];
char s[405];
int main () {
    scanf("%s", s + 1);
    int len = strlen(s + 1);
    for(int i = 1; s[i]; ++i){
        int id = 0;
        if (s[i] == 'C'){
            id = 1;
        }
        if (s[i] == 'K'){
            id = 2;
        }
        c[id]++;
        t[id][c[id]] = i;
    }
    if(c[0] > (len + 1) / 2 || c[1] > (len + 1) / 2 || c[2] > (len + 1) / 2){
        printf("Impossible!\n");
        return 0;
    }
    memset(dp, 0x3f, sizeof dp);
    dp[0][0][0][0] = dp[0][0][0][1] = dp[0][0][0][2] = 0;
    for(int i = 0; i <= c[0]; ++i){
        for(int j = 0; j <= c[1]; ++j){
            for(int k = 0; k <= c[2]; ++k) {
                int p = i + j + k;
                if(i){
                    dp[i][j][k][0] = min(dp[i - 1][j][k][1], dp[i - 1][j][k][2])
                                    + abs(p - t[0][i]);
                }
                if(j){
                    dp[i][j][k][1] = min(dp[i][j - 1][k][0], dp[i][j - 1][k][2])
                                    + abs(p - t[1][j]);
                }
                if(k){
                    dp[i][j][k][2] = min(dp[i][j][k - 1][0], dp[i][j][k - 1][1])
                                    + abs(p - t[2][k]);
```

```
                }
            }
        }
    }
    int ans = min(dp[c[0]][c[1]][c[2]][0], dp[c[0]][c[1]][c[2]][1]);
    ans = min(ans, dp[c[0]][c[1]][c[2]][2]);
    ans /= 2;
    printf("%d", ans);
    return 0;
}
```

T3 繁花（flower）

30分

暴力枚举i_1, j_1, i_2, j_2，暴力求和，时间复杂度$O(n^5)$

50分

暴力枚举i_1, j_1, i_2, j_2，前缀和快速求和，时间复杂度$O(n^4)$

100分

不妨设$n \leq m$，那么$n \leq \sqrt{2 \times 10^5}$

枚举i_1, i_2, j_1，我们需要求出$j_2 > j_1$使得前缀和最大，这可以通过倒着枚举j_1，更新j_2实现

总时间复杂度$O(n^2 m)$

标程

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 450;
vector<int> row[N], col[N], a[N];
int f[200005], n, m;
int main() {
		scanf("%d%d", &n, &m);
		if (n < m) {
			for (int i = 0; i <= n; ++i) {
				a[i].resize(m + 1);
				row[i].resize(m + 1);
				col[i].resize(m + 1);
				for (int j = 0; j <= m; ++j)
					a[i][j] = row[i][j] = col[i][j] = 0;
			}
			for (int i = 1; i <= n; ++i)
				for (int j = 1; j <= m; ++j)
					scanf("%d", &a[i][j]);
		} else {
			for (int i = 0; i <= m; ++i) {
				a[i].resize(n + 1);
				row[i].resize(n + 1);
				col[i].resize(n + 1);
				for (int j = 0; j <= n; ++j)
					a[i][j] = row[i][j] = col[i][j] = 0;
			}
			for (int i = 1; i <= n; ++i)
				for (int j = 1; j <= m; ++j)
					scanf("%d", &a[j][i]);
			swap(n, m);
		}
		for (int i = 1; i <= n; ++i)
			for (int j = 1; j <= m; ++j) {
				row[i][j] = row[i][j - 1] + a[i][j];
				col[i][j] = col[i - 1][j] + a[i][j];
			}
		int ans = -1e9;
		for (int l = 1; l <= n; ++l)
			for (int r = l + 1; r <= n; ++r) {
```

```
                                f[m + 1] = -1e9;
                                for (int i = m; i >= 1; --i)
                                        f[i] = max(f[i + 1], row[l][i - 1] + row[r][i - 1] + col[r
                                for (int i = 1; i < m; ++i)
                                        ans = max(ans, -row[l][i] - row[r][i] + col[r][i] - col[l
                        }
                printf("%d\n", ans);
        return 0;
}
```

T4 划分（split）

20分

暴力枚举

40分

令f_i表示$1-i$的划分的最大值和最小值

$$f_i = Min f_{j-1} + Max_{k=j}^{i} a_k, \sum_{k=j}^{i} a_k \leq m$$

100分

优化dp

后面的约束条件可以通过单调队列解决

固定i

前面这部分，f_j是单调不降的，$Max_{k=j}^{i} a_k$是单调不增的

对于每一个不同的$Max_{k=j}^{i} a_k$，我们发现$f_{j-1} + Max_{k=j}^{i} a_k$都在$j$最小值处取到最小值，而$f_j$不变

我们可以再开一个单调队列，来维护$max_{k=j}^{i} a_k$

然后对于每一个队列中的元素统计他们的最小值，就可以得到f_i

我们需要求最小值、修改元素、删除元素，可以用multiset维护

时间复杂度$O(nlgn)$

标程

```cpp
#include<bits/stdc++.h>
#define ll long long
using namespace std;

const int N=1e5+5;
int n,m,a[N],b[N];
ll S[N],f[N];
deque<int>q;
multiset<ll>s;
int main() {
	scanf("%d%d",&n,&m);
	for(int i=1;i<=n;i++) {
		scanf("%d",&a[i]);
		S[i]=S[i-1]+a[i];
	}
	f[0]=0; q.push_back(0),s.insert(0);
	for(int i=1,he=0;i<=n;i++) {
		while(S[i]-S[he]>m) he++;
		while(q.size()>1&&q[1]<=he) {
			s.erase(s.find(f[q[0]]+b[q[0]]));
			q.pop_front();
		}
		if(!q.empty()&&q[0]<he){
			s.erase(s.find(f[q[0]]+b[q[0]]));
			b[he]=b[q[0]],q[0]=he;
			s.insert(f[q[0]]+b[q[0]]);
		}
		while(q.size()>1&&b[q[q.size()-2]]<=a[i]) {
			int t=q[q.size()-1];
			s.erase(s.find(f[t]+b[t]));
			q.pop_back();
		}
		if(!q.empty()&&b[q[q.size()-1]]<a[i]) {
			int t=q[q.size()-1];
			s.erase(s.find(f[t]+b[t]));
			b[t]=a[i];
			s.insert(f[t]+b[t]);
		}
		f[i]=*s.begin();
```

```
                s.insert(f[i]);
                q.push_back(i);
        }
        printf("%lld\n",f[n]);
        return 0;
}
```