A

在我的世界中甘蔗有个奇怪的设定，那就是必须生长在水边的陆地上。在一个$n*m$的网格中，点(x,y)可以种甘蔗当且仅当其是陆地且其上下左右与其相邻的点至少有一个是水源，每个点只能种1个甘蔗。现在给你一个仅由'.'和'$*$'构成的矩形网格，
'.'表示陆地，'$*$'表示水源，请你回答该网格最多可以种多少甘蔗。

> 输入:
>
> 第一行输入一个整数T代表测试用例个数，接下来一行输入两个整数n,m(分别表示网格的行数和列数)，接下来n行，每行输入长度为m仅包含'.'和'$*$'的字符串代表网格的每一行。
> $1<=T<=100,1<=n,m<=100$

> 输出:
>
> 输出T行，每行表示每个测试用例最多能种多少甘蔗。

例如:

> 输入:
>
> 1
> 4 5
> ....*
> .. * ..
> .. * **
> * * * * *

> 输出:
>
> 7

题解:

直接遍历每个陆地即可。时间复杂度$O(n*m)$

```cpp
#include <iostream>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
using i64=long long;
using pii=pair<int,int>;
using str=string;
const int N=100;
int n,m,k,ans;
char g[N+10][N+10];
pii xy[4]={{1,0},{-1,0},{0,1},{0,-1}};
void solve()
{
    ans=0;
    cin>>n>>m;
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            cin>>g[i][j];
        }
    }
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            if(g[i][j]=='*') continue;
            for(auto [dx,dy]:xy) {
                int x=i+dx,y=j+dy;
                if(x<1||x>n||y<1||y>m) continue;
                if(g[x][y]=='*') {
                    ans++;
                    break;
                }
            }
        }
    }
    cout<<ans<<endl;
}
signed main()
{
    IOS;
    int _=1;
    cin>>_;
```

```
    while(_--){
        solve();
    }
}
```

B

在我的世界中，玩家会使用折半的方法来寻找要塞。在起点为1长度为n的直线上存在一座要塞，其坐标为k。steve的指针在每个点都会指向要塞的方向。
可用长度为n且只包含字符'L'和'R'的字符串s表示其在每个点的指向('R'为右，'L'为左，在要塞点一定为'L'),
他仍以为指针是正常的，于是会按照以下形式寻找要塞:

首先初始化指针l为0，r为n+1，然后执行下述循环：

1. 如果$l+1=r$，循环结束。
2. $m=\lfloor (r+l)/2 \rfloor$。
3. 如果该点指向左，赋值r=m，否则l=m。

最后r即是要塞坐标。

现在steve的指针坏了(他仍会按照上述流程寻找要塞)，用字符串s表示指针坏掉之后在每个点的指向。

!!! 定义直线上每个点的状态为该点指针所指方向('R'或'L')以及要塞的存在情况(有或无)。

为了保证steve一定能找到要塞，你可以任意选择两个点，并交换此两点状态。

> 输入:
>
> 第一行输入一个整数T代表测试用例个数，第二行输入两个整数n,k(分别代表直线长度和要塞坐标)，接下来一行输入n个字符(代表每个点指针的方向)。
> $1<=T<=100, 3<=n<=1e5, 1<=k<=n$

> 输出:
>
> 输出T行，每行代表要交换两点的下标(按从小到大顺序输出)，如果不需要交换，则直接输

> 出0。

例如:

> 输入:
>
> 1
> 9 5
> RLRLLLRRL

> 输出:
>
> 4 5

题解:

由于指针在要塞处一定为'L'，所以按照上述步骤steve肯定会停留在一个指向'L'的地方，那么交换要塞坐标和steve的停留点的状态，注意到此时并没有改变每个点二分的状态(即指针在每个点的状态)，交换之后steve还是会停留在之前的点，此点要塞的状态已经为'有'，steve成功找到要塞。时间复杂度$O(t * log_2 n)$;

```cpp
#include <iostream>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
using i64=long long;
using str=string;
int n,k;
str s;
int check() {
    int l=0,r=n+1;
    while(l+1!=r) {
        int m=l+r>>1;
        if(s[m]=='L') r=m;
        else l=m;
    }
    return r;
}
void solve()
{
    cin>>n>>k>>s;
    s=' '+s;
    int t=check();
    if(k==t) cout<<0<<endl;
    else cout<<t<<' '<<k<<endl;
}
signed main()
{
    IOS;
    int _=1;
    cin>>_;
    while(_--){
        solve();
    }
}
```

C

在二维我的世界中水只会向左右扩散。一条长度为n的直线由n个编号为$1, 2, 3, \ldots, n$的方块组

成，在编号为i的方块上倒一桶水，
水会向左右各扩散7格，即区间$[max(1,i-7),min(n,i+7)]$内的方块将变成有水方块。已知有m个方块被倒上了水，
有q次询问，每次询问区间$[l,r]$内共有多少有水方块。

> 输入:
>
> 第一行输入3个整数n,m,q(分别表示直线长度，倒水方块个数，询问次数);
> 接下来1行输入m个整数w_i(表示倒水方块的编号);
> 接下来q行每行输入两个整数l,r(表示询问的区间)。
> $1<=n<=2e6,1<=m,w_i<=n,1<=q<=1e4,1<=l<r<=n$

> 输出:
>
> 输出q行，每行一个整数表示区间内的有水方块个数。

例如:

> 输入:
>
> 100 5 5
> 9 18 34 68 81
> 1 100
> 3 59
> 1 18
> 32 89
> 77 93

> 输出:
>
> 67
> 38
> 17
> 38
> 12

题解:

维护长度为n的数组$a[n]$，其中$a[i]=0$表示无水，$a[i]=1$表示有水，将有水区间全部初始化为1，
考虑差分，直接将区间两端+1和-1，再考虑前缀和，将$a[i]$表示为前i个点有水方块个数。时间复杂度$O(n+m+q)$

```
#include <iostream>
#include <vector>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
int n,m,q;
void solve()
{
    cin>>n>>m>>q;
    vector w(n+10,0);
    for(int i=1;i<=m;i++) {
        int t;cin>>t;
        w[max(1,t-7)]++;
        w[min(n,t+7)+1]--;
    }
    for(int i=1;i<=n;i++) w[i]+=w[i-1];
    for(int i=1;i<=n;i++) w[i]=w[i]>0?1:0;
    for(int i=1;i<=n;i++) w[i]+=w[i-1];

    while(q--) {
        int l,r;cin>>l>>r;
        cout<<w[r]-w[l-1]<<endl;
    }
}
signed main()
{
    IOS;
    int _=1;
    //cin>>_;
    while(_--){
        solve();
    }
}
```

D

steve是我的世界种菜高手。在一张只包含整点的二维笛卡尔坐标系中，有n个点包含水源。
如果某个正矩形的四个顶点都是水源，则steve认为这是一个种菜圣地。

容易知道：当选择两个横纵坐标都不相同的点时，
这两点一定可以确定一个正矩形，例如$(1,2)$和$(3,5)$所确定的正矩形四个顶点为
$(1,5),(3,5),(3,2),(1,2)$。

如果steve任选两个横纵坐标都不相同的水源所确定的矩形都是一个种菜圣地且至少存在一处种菜圣地时，steve是高兴的，否则steve会很伤心。

- 正矩形定义：每条边至少和一个坐标轴平行的矩形。

> 输入:
>
> 第一行输入一个整数T表示有T组测试用例，接下来T组用例每组第一行输入一个n表示有n个水源，接下来n行每行输入两个数x,y表示水源的坐标。
> $1 <= T <= 100, 2 <= n <= 1e4, 1 <= x, y <= 1e9$

> 输出:
>
> 输出T行，steve高兴则输出"happy"，否则输出"nohappy"。

例如:

> 输入:
>
> 3
> 6
> 1 1
> 1 2
> 1 3
> 2 1
> 2 2
> 3 1
> 2
> 1 4

> 1 9
> 4
> 1 1
> 1 2
> 2 1
> 2 2

> 输出:
>
> nohappy
> nohappy
> happy

提示:

> 在第一个测试用例中，水源$(1,2)$和$(3,1)$所确定的正矩形顶点为
> $(1,2),(3,2),(3,1),(1,1)$,其中$(3,2)$不是水源，故不是种菜圣地。
> 在第二个测试用例中，steve无论如何也找不出一个种菜圣地。

题解:

当且仅当这些点的组成是一个矩阵任意的划分时符合题意，时间复杂度O($T*n*log_2n$)。

```cpp
#include <bits/stdc++.h>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
void solve() {
    int n;cin>>n;
    map<int,bool> a,b;
    for(int i=1;i<=n;i++) {
        int x,y;cin>>x>>y;
        a[x]=true,b[y]=true;
    }
    if(a.size()*b.size()==n&&a.size()>1&&b.size()>1) cout<<"happy"<<endl;
    else cout<<"nohappy"<<endl;
}
signed main() {
    IOS;
    int _=1;
    cin>>_;
    while(_--) {
        solve();
    }
}
```

E

在我的世界中，沙块只能被放在其它方块之上(不能悬浮，沙块也属于方块)，当其下方块消失时，沙块会一直掉落直至其下方有方块为止。现在有一个高度为n的沙
块堆，其中的沙块有4种不同颜色：白-绿-黑-红(分别用a, b, c, d表示), 其之间有克制关系(a被b克，c被d克)，当有克制关系的沙块相邻且被克制的沙
块在下方时，这两个沙块会迅速消失。

给你一串长度为n且仅由a, b, c, d组成的字符串表示地面上自下往上堆成的高度为n的沙块堆，请你回答这个沙堆的最终高度是多少。

> 输入:
>
> 第一行输入一个T表示T组数据，接下来T行每行输入一个字符串代表沙块堆。
> $1 <= T <= 100, 1 <= n <= 1e4$

> 输出:
>
> 输出T行，每行一个整数表示沙堆最终高度。

例如:

> 输入:
>
> 1
> abaacdb

> 输出:
>
> 1

提示:

> 第一个测试用例中，容易观察到：第1个沙块被第2个沙块克制，所以会消失，接着第5个沙块被第6个沙块克制，会直接消失，然后第7个沙块
> 掉落到第4个沙块上，第4个沙块会被第7个沙块克制，然后消失，最后仅剩下第3个沙块(这里所描述的下标都是相对原字符串)。

题解:

从下往上处理字符串，当某两个沙块消失时，上面的沙块会掉下来组成新的结构，属于后进后考虑，使用栈维护沙堆情况。第i个沙块是否会消失需要考虑
其是否被第i+1个沙块克制，如果被克制，则说明第i和i+1个沙块会消失，直接出栈即可；否则将第i+1个沙块压入栈顶。
时间复杂度$O(T*n)$

```cpp
#include <iostream>
#include <stack>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
using str=string;
char col['z'+10];
void init() {
    col['a']='b';
    col['c']='d';
}
void solve()
{
    stack<char> sand;
    str s;cin>>s;
    for(auto i:s) {
        if(sand.empty()) {
            sand.emplace(i);
            continue;
        }
        if(col[sand.top()]==i) sand.pop();
        else sand.emplace(i);
    }
    cout<<sand.size()<<endl;
}
signed main()
{
    IOS;
    init();
    int _=1;
    cin>>_;
    while(_--){
        solve();
    }
}
```

F

在我的世界起床战争中有n座编号为$1, 2, 3, ..., n$的岛屿以及m条路，第i条路连接着岛屿u_i和v_i，长度为w_i。
你的速度为s(其初始化为0)，通过每条路的时间为$\lfloor w_i/(s+1) \rfloor$，幸运的是地图上共有$k$瓶药水且每座岛屿都至多存在一瓶药水，
若你此时所在的岛屿存在药水，你可以选择喝下该药水，每瓶药水喝完就会立即消失。这些药水有2个等级(I级或II级)，假设喝之前你的速度为i,
当你喝下等级为j的药水后：

- 若$i > j$,你的速度将不会变化。
- 若$i = j$,你的速度将迅速+1。
- 若$i <$
  j,你的速度将迅速变为j。

你开始在1号岛屿上想要用最短时间去n号岛屿。

> 输入:
>
> 第一行输入3个整数n, m, k(分别表示岛屿数，边数，药水数);
> 接下来k行每行输入2个整数a, b(a表示有药水的顶点，b表示该药水等级);
> 随后m行分别输入3个整数u_i， v_i， w_i(表示第岛屿u_i和v_i之间有一条路，路长为w_i)。
> 保证不会出现自环和多条边。
> $2 <= n <= 100$，$1 <= m <= 4000$，$0 <= k <= 10$，$0 <= a <= n$，$1 <= b <= 2$，$1 <= u_i, v_i <= n$，$10 <= w_i <= 1e5$。

> 输出:
>
> 输出一个整数代表最少花费的时间，如果无法到达，请输出-1。

例如:

> 输入:
>
> 5 5 2
> 2 2
> 4 2
> 1 2 100

> 2 3 3000
> 1 4 100
> 3 4 3000
> 3 5 100

> 输出:
>
> 941

提示:

> 你的路线为$1->2$(此时速度变为2)$->1->4$(此时速度变为3)$->3->5$.
> 可以验证这是最优解。

题解:

这题还需要题解?

```cpp
#include <iostream>
#include <queue>
#include <map>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
using i64=long long;
using i128=__int128;
using t4i=tuple<int,int,int,i128>;
using t3i=tuple<int,int,i128>;
using t2i=tuple<int,int>;
constexpr int N=100;
constexpr int inf=0x3f3f3f3f;
int n,m,k,ans=inf;

map<int,int> eff;
priority_queue<t4i> pq;
vector<t2i> g[N+10];

void bit(i128 &x,int y) {//将y点药水清空
    i128 tmp=1;
    tmp<<=y;
    x|=tmp;
}

bool check(i128 x,int y) {//检测y点是否有药水
    i128 tmp=1;
    tmp<<=y;
    return x&tmp;
}

void bfs() {
    map<t3i,int> w;
    map<t3i,bool> v;
    pq.emplace(0,0,0,0);//时间，点，速度，药水状态
    w[{0,0,0}]=0;//点，速度，药水状态

    while (!pq.empty()) {
        auto [t,id,e,s]=pq.top();//时间，点 速度，药水状态
        pq.pop();
```

```
        t=-t;

        if(v[{id,e,s}]) continue;
        v[{id,e,s}]=true;//对该状态进行标记

        for(auto [a,b]:g[id]) {
            i128 s_t=s;
            int e_t=e;
            int tmp=b/(e_t+1)+t-inf;//不喝药水
            if(tmp<w[{a,e_t,s_t}]) {
                w[{a,e_t,s_t}]=tmp;
                pq.emplace(-(w[{a,e_t,s_t}]+inf),a,e_t,s_t);
                if(a==n) ans=min(ans,tmp+inf);//更新状态
            }

            if(!check(s_t,id)&&eff[id]>0) {//检查该点是否有药水
                if(eff[id]>e_t) e_t=eff[id];
                else if(eff[id]==e_t) e_t++;
                if(eff[id]>=e_t) bit(s_t,id);//标记该点药水已消失

                tmp=b/(e_t+1)+t-inf;//整数偏移，将0作为无穷大

                if(tmp<w[{a,e_t,s_t}]) {
                    w[{a,e_t,s_t}]=tmp;
                    pq.emplace(-(w[{a,e_t,s_t}]+inf),a,e_t,s_t);
                    if(a==n) ans=min(ans,tmp+inf);//更新状态
                }
            }
        }
    }
}

void solve()
{
    cin>>n>>m>>k;
    for(int i=1;i<=k;i++) {
        int a,b;cin>>a>>b;
        eff[a]=b;
    }
```

```
    g[0].emplace_back(1,0);
    for(int i=1;i<=m;i++) {
        int u,v,w;cin>>u>>v>>w;
        g[u].emplace_back(v,w);
        g[v].emplace_back(u,w);
    }
    bfs();
    if(ans>=inf) ans=-1;
    cout<<ans<<endl;
}
signed main()
{
    IOS;
    int _=1;
    //cin>>_;
    while(_--){
        solve();
    }
}
```

G

你是我的世界高手，想要创造一台能计算斐波那契数列无穷项的红石计算机。你的同学是数学高手，现在他给你一些资料：

1. 斐波那契数列递推公式为$F_n = F_{n-1} + F_{n-2}$;
2. 2×2矩阵与2×1矩阵乘法法则为
   $$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{12} \\ a_{21} * b_{11} + a_{22} * b_{12} \end{bmatrix},$$
   例如
   $$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_{n-1} + F_{n-2} \end{bmatrix};$$
3. 2×2矩阵与2×2矩阵乘法法则为
   $$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix},$$
   例如

$$\begin{bmatrix}0 & 1\\1 & 1\end{bmatrix}^2 = \begin{bmatrix}1 & 1\\1 & 2\end{bmatrix};$$

4. 矩阵乘法满足结合律;

给定斐波那契数列前两项(第1项和第2项)，请你计算出第n项的值，由于结果很大，请你将结果对$1e9+7$取模;

输入:

> 第一行输入一个整数T表示有T组测试数据，接下来每组测试数据输入三个整数a,b,n分别表示斐波那契数列前两项和项数.
> $1<=T<=1e4, 1<=a,b<=100, 1<=n<=1e18$

输出:

> 输出T行，每行一个整数代表答案.

例如:

> 输入:
>
> 1
> 3 4 17
>
> 输出:
>
> 5778

题解:

注意到第2条资料，$\begin{bmatrix}F_{n-1}\\F_{n-2}\end{bmatrix}$就相当于斐波那契数列第$i$和$i-1$项，$\begin{bmatrix}F_{n-1}\\F_{n-1}+F_{n-2}\end{bmatrix}$就是斐波那契数列第$i$和$i+1$项，所以当拿到数列第i-1和i项时，我们可以将其类比为一个矩阵，将其与$\begin{bmatrix}0 & 1\\1 & 1\end{bmatrix}$相乘n次就能得到第i+n项，由于矩阵乘法满足结合律，所以直接将n个$\begin{bmatrix}0 & 1\\1 & 1\end{bmatrix}$拿出来做快速幂即可，时间复杂度O($T*log_2 n$)。

```cpp
#include <iostream>
#include <vector>
#include <valarray>
#include <iomanip>
#define IOS ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)
using namespace std;
using i64=long long;
using str=string;
using f64=double;
constexpr int md=1e9+7;
i64 a,b,n;

template <typename T>
T mul(T& x,T& y) {
    int l=x.size(),r=y[0].size(),s=y.size();
    T ret;
    ret.resize(l,vector(r,0ll));
    for(int i=1;i<=l-1;i++) {
        for(int j=1;j<=r-1;j++) {
            i64 t=0;
            for(int k=1;k<=s-1;k++) t=(t+x[i][k]*y[k][j]%md)%md;
            ret[i][j]=t;
        }
    }
    return ret;
}

template <typename T>
T qmi(T& x,i64 y) {
    T ret;
    i64 len=x.size();
    ret.resize(len,vector(len,0ll));
    for(int i=1;i<=len-1;i++) {
        for(int j=1;j<=len-1;j++) {
            if(i==j) ret[i][j]=1;
        }
    }
    while(y>0){
        if(y&1) ret=mul(ret,x);
```

```
        y>>=1;
        x=mul(x,x);
    }
    return ret;
}

i64 fab(i64 _) {
    vector<vector<i64>> x = {{0,0,0},
                             {0,0,1},
                             {0,1,1}};
    vector<vector<i64>> y = {{0,0},
                             {0,a},
                             {0,b}};
    auto r = qmi(x, _);
    r=mul(r,y);
    return r[1][1];
}
void solve()
{
    cin>>a>>b>>n;
    cout<<fab(n-1)<<endl;
}
signed main()
{
    IOS;
    int _=1;
    cin>>_;
    while(_--){
        solve();
    }
}
```

H

下面哪位学长是奶龙:

xmy, clh, ghf, cmy, hwj, lyx, wdw, wpc, wyd, xng, zqt, chx, hjx, lz, zjh

> 输入:
>
> 无

> 输出:
>
> 输出"xxx"(不含引号)表示xxx是奶龙

题解:

任意输出一位学长名字都可。

```
xmy
```

I

数字王国里在进行24点的速算pk（给定三个数字(可以任意交换位置)，只用加减乘除四个运算符号计算出24)

> 输入:
>
> 第一行输入一个T，代表有T组数据，接下来T行，每行输入3个整数a, b, c。
> $1 <= t <= 100, -1e4 <= a, b, c <= 1e4; a, b, c$都不为$0$

> 输出:
>
> 输出T行，假如这3个数能通过运算得到24，那么输出"YES"，如果不能则输出"NO"(注意运算符号的减号不能使用在第一个数字之前)。

例如:

> 输入:
>
> 2
> 6 2 2
> 1 2 3

> 输出:

```
YES
NO
```

题解:

枚举即可。

```cpp
#include <bits/stdc++.h>
using namespace std;
using i64 = long long;
typedef long long ll;

void solve()
{
    vector<double> n(5, 0);
    cin >> n[0] >> n[1] >> n[2];
    for (i64 i = 0; i < 3; i++)
    {
        for (i64 j = 0; j < 3; j++)
        {
            for (i64 k = 0; k < 3; k++)
            {
                if (i != j && i != k && k != j)
                {
                    if (n[i] + n[j] + n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] + n[j] - n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] + n[j] * n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] + n[j] / n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] - n[j] + n[k] == 24)
                    {
```

```
                cout << "YES" << endl;
                return;
            }
            if (n[i] - n[j] - n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] - n[j] * n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] - n[j] / n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] * n[j] + n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] * n[j] - n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] * n[j] * n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] * n[j] / n[k] == 24)
            {
                cout << "YES" << endl;
                return;
            }
            if (n[i] / n[j] + n[k] == 24)
```

```
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] / n[j] - n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] / n[j] * n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                    if (n[i] / n[j] / n[k] == 24)
                    {
                        cout << "YES" << endl;
                        return;
                    }
                }
            }
        }
    }
    cout << "NO" << endl;
}

signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    int _ = 1;
    cin >> _;
    while (_--)
        solve();

    return 0;
}
```

J

奶茶店正在进行一个奇怪的促销活动， 现在有n瓶奶茶排成一排摆在你面前，价格都为x，但是活动要求你只能购买一次，并且只能购买连续的一段的奶茶，
你知道每一瓶奶茶的原价，现在你想知道购买任意瓶奶茶最多能省多少钱。

> 输入:
>
> 第一行输入两个正整数n, x,第二行输入n个正整数a_i，表示每一瓶奶茶的原价。
> $1 <= n <= 1e3, 1 <= x <= 1e9, 1 <= ai <= 1e9$

> 输出:
>
> 输出一行整数代表答案。

例如:

> 输入:
>
> 6 8
> 12 5 7 19 20 2

> 输出:
>
> 23

题解:

方法1.

维护一个变量p表示上一个点所在连续段能省钱的最大值，当遍历到下一个点时有两种选择：接着上一个点继续买或者从该点起重新开始买，如果接着买，那么p=p+该点能省的钱,
如果不接着买，那么p应该抛弃之前节省的钱重新开始计算(p=该点省的钱),于是就有状态转移方程p=max(p+该点省钱数,该点省钱数),同时再维护一个ans表示所有连续段中最省钱的段吗，时间复杂度O(n)。

方法2.

暴力遍历每个区间，得出最大值即可，时间复杂度$O(n^2)$。

```cpp
//方法一代码
#include <iostream>
using namespace std;
int n,k,x,ans,p;
signed main(){
    cin>>n>>x;
    for(int i=1;i<=n;i++) {
        cin>>k;
        ans=max(p=max(p+k-x,k-x),ans);
    }
    cout<<ans<<endl;
}
```