哈希

```
//字符串哈希模板

##### typedef unsigned long long ull;
ull h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64
const int P=13331;
初始化
p[0]=1;
for(int i=1;i<=n;i++)
{
    h[i]=h[i-1]*P+str[i];
    p[i]=p[i-1]*P;
}

// 计算子串str[l~r]的哈希值,l和r范围为1~n***
ull get(int l, int r)//画图理解  进制运算  得到子串哈希值
{
    return h[r]-h[l-1]*p[r-l+1];
}
```

```
#include<iostream>
using namespace std;
const int MAX=100005;            //数组的最大长度（即图中点个数的最大值）
int m,n;                         //当前图的长宽规格
int pre[MAX];                    //用于存放每个点的根节点
void init(int n) {           //初始化函数
    for(int i=1; i<=n; i++)
        pre[i]=i;
}
int find(int x) {//递归
    if (x != pre[x]) pre[x] = find_r(pre[x]);//
    return pre[x];
}
//循环
int _find(int x) {
    while(x != pre[x]) { //如果x元素的父亲指向的不是自己，说明x并不是集合中的根元素，还需要一直向上查找和路径压缩
        //在find查询中嵌入一个路径压缩操作
        pre[x]=pre[pre[x]];//区别
        //x元素不再选择原来的父亲节点，而是直接选择父亲节点的父亲节点来做为自己新的一个父亲节点
        //这样的操作使得树的层数被压缩了
        x=pre[x];//x压缩完毕后且x并不是根节点，x变成x新的父节点继续进行查找和压缩的同时操作
    }
    return x;//经过while循环后，x=pre[x],一定是一个根节点，且不能够再进行压缩了,我们返回即可
}
void merge(int x,int y) {       //合并函数
    int rx=find_r(x);
    int ry=find_r(y);
```

```
    if(rx!=ry) pre[rx]=ry;
}
```

字典树

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;
int t;
//统计不同单词数
//flag =1 结点数
const int MAX =2e6+5;//如果是64MB可以开到2e6+5，尽量开大
int tree[MAX][30];//tree[i][j]表示节点i的第j个儿子的节点编号
int flag[MAX];//表示以该节点结尾是一个单词
int sum[MAX];
int tot=0;//总节点数
int ans=0;
void insert_(string s)
{
   int  len=s.size();
   int root=0;
   for(int i=0;i<len;i++)
   {
       int id=s[i]-'0';
       if(!tree[root][id]) tree[root][id]=++tot;//新子树 编号tot+1
       sum[tree[root][id]]++;//
       root=tree[root][id];
   }
   if(!flag[root]){flag[root]=1;ans++;}//未出现
}
int  find_(string s)//查询操作，按具体要求改动
{
    int len=s.size();
    int root=0;
    for(int i=0;i<len;i++)
    {
        int id=s[i]-'0';
        if(!tree[root][id]) return 0;
        root=tree[root][id];
    }
    return sum[root];////返回当前字符串结尾节点的访问次数，也就是作为前缀的出现次数
}
void init()//最后清空，节省时间
{
    ans=0;
    for(int i=0;i<=tot+5;i++)//?
    {
       flag[i]=false;
       sum[i]=0;
       for(int j=0;j<35;j++)
```

```
            tree[i][j]=0;

     }
    tot=0;
}

int main(){
std::ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);

string s;
while(getline(cin,s)){
    if(s=="#"){
    break;
    }
    string s1="";
    for(int i=0;i<s.size();i++){//注意输入
        if(i>0&&s[i]==' '&&s[i-1]!=' '){insert_(s1);s1="";}//
        else if(s[i]!=' '){
            s1+=s[i];
            if(i==s.size()-1&&s1!=" ")insert_(s1);//最后也要
             }
    }
    cout<<ans<<endl;
    init();
}
    return 0;
}
```

单调队列

滑动窗口

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e6+ 5;
int T, n, k,a[N];
int q[N]; //单调队列  存的是下标  也可以多开一个数组存下标
void solve(){
    cin>>n>>k;
    //最小值;单调递增
    for(int i=1;i<=n;i++)cin>>a[i];
    int head=1,tail=0;  //   -1和0; 0,0也可
    for(int i=1;i<=n;i++){// 每次移动一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]>=a[i])tail--; //pop 掉队尾大于a[i]的 (在前面且小于)
```

```
        q[++tail]=i ; //新元素入队
        if(i>=k)cout<<a[q[head]]<<" "; //输出队首元素max
    }
    cout<<endl;
    //最大值同理
    head=1,tail=0;
    for(int i=1;i<=n;i++){// 每次一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]<=a[i])tail--; //pop 掉队尾小于a[i]的 (在前面且小于)
        q[++tail]=i ; //新元素入队
        if(i>=k) //窗口已进入
            cout<<a[q[head]]<<" "; // 输出队首元素max
    }
}
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
      solve();
  return 0;
}
```

树状数组

1. 单点修改区间查

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;//模板题 单点修改 + 区间查询
//动态维护树状数组 c[i]代表a[i]及之前共lowbit[i]个元素
int c[50005],n,t;
int lowbit(int x){
    return x& -x;
}
void update(int i,int val){//单点更新:每次加自身lowbit的元素改变
    while(i<=n){
        c[i]+=val;
        i+=lowbit(i);
    }
}//更新(从小到大)时查询(从大到小)的逆过程
int sum(int i){//求前缀和/
    int ret =0;
    while(i>0){
        ret+=c[i];
        i-=lowbit(i);
    }
    return ret;
}
int main(){
cin>>t;
```

```
int Case=0;
while(t--){
    memset(c,0,sizeof c);
    Case++;
printf("Case %d:\n",Case);
scanf("%d",&n);
for(int i=1;i<=n;i++){
    int val;
    scanf("%d",&val);
    update(i,val);
}
string s;
while(cin>>s){
    if(s=="End")break;
    int a,b;
    scanf("%d %d",&a,&b);
    int ans=0;
    if(s=="Query"){
        ans=sum(b)-sum(a-1);
       printf("%d\n",ans);
    }
    else if(s=="Sub"){
        update(a,-b);
    }
    else if(s=="Add"){
        update(a,b);
    }
}
}
    return 0;
}
```

2.区间修改，单点查

差分

```
#include <bits/stdc++.h>
#define LOCAL//由于本题是先修改最后按顺序查询 所以直接用普通数组差分实现区间修改之后按序维护输出前缀和更快(n>nlogn)  ,但如果边改变查或随机查询效率显然不如本方法
using namespace std;//nlog
//树状数组实现 单点查询 区间修改
//   用差分树状数组 d[]实现
//用差分数组将区间修改转化为单点修改    单点查询转化为求前缀和
//区间修改只需改端点的差分值
//  单点查询只需求d[i]的前缀和
//  d[0]=0,d[1]=a[1];d[i]==a[i]-a[i-1],
int n,d[100005];
int lowbit(int i){
    return i& -i;
}
void update(int i,int val){//树状数组单点修改，每次对差值更新
    while(i<=n)d[i]+=val,i+=lowbit(i);
```

```
}
int sum(int i){//求树状数组前缀和***//求sum d[i]实现单点查询
    int ret =0;
    while(i>0){
    ret+=d[i];
    i-=lowbit(i);
    }
    return ret;
}
void  range_update(int l,int r,int x){//差分 实现区间修改
    update(l,x);update(r+1,-x);//
}

int main(){
#ifdef LOCAL
    freopen("data.in","r",stdin);
    freopen("data.out","w",stdout);
#endif
while(cin>>n,n){
    memset(d,0,sizeof d);
    for(int i=1;i<=n;i++){
        int a,b;
        cin>>a>>b;
        range_update(a,b,1);//区间修改
    }
    for(int i=1;i<=n;i++){
        cout<<sum(i);
        if(i!=n)cout<<" ";
    }
    cout<<endl;
}
    return 0;
}
```

线段树

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;
// l,r 是大（总）区间 ， L，R是操作/查询区间
const int N =1e5+7;//元素总个数
int a[N];//原数组 可不用
struct Seg{
    struct segtreenode{
        int val;
      int lazy ;//懒惰标记
         //其他元素
    }segtree[N<<2];//定义原数组大小四倍的线段数组
    void pushup(int rt){
        segtree[rt].val=segtree[rt<<1].val+segtree[rt<<1|1].val;
    }//将左右字数的总值加到根节点
```

```
    void build(int l,int r,int rt){// 递归构造线段树
      segtree[rt].lazy=0;//      初始化
      if(l==r){//出口   左右相等  为叶子节点则停止向下递归
          segtree[rt].val=a[l];//叶子节点的l,r即位置下标
          return;
      }
      int mid=(l+r)/2;
      build(l,mid, rt*2);    //递归构造左子树  根序号为2*rt ,2*tr+1
      build(mid+1,r,rt*2+1);   //递归构造右子树
      pushup(rt);            //** 回溯,当左右子树都构造完后向上加到根节点
    }

    //单点更新，假设 a[t]+=c    类似二进制，每层只需更新一个   从上到下
    void updateNode(int t,int c,int l,int r,int rt){//l,r 表示当前节点区间，rt表示当前根
节点编号
        if(l==r){
        segtree[rt].val+=c;//     叶子节点   直接修改
        return;
    }
    int mid=(l+r)/2;
      if(t<=mid)  updateNode(t,c,l,mid,rt<<1);  //更新左子树
      else    updateNode(t,c,mid+1,r,rt<<1|1);
      pushup(rt);                          //回溯向上更新，相加
    }

    //区间查询(区间a[L....R]的和)   [L,R]为操作区间,[l,r]为当前区间，rt为当前节点编号
    int query(int L,int R,int l,int r,int rt){
        if(L<=l&&r<=R)
        return  segtree[rt].val;//当前区间被包含  则直接返回(整个被加)
      // 并且不再向下递归
      if(L>r||R<l)    //当前区间全部不重和，则返回0，而且其子区间也不会包含
        return 0;
    //否则部分包含   向下递归
      int mid=(l+r)/2;
      return  query(L,R,l,mid,rt<<1)+query(L,R,mid+1,r,rt<<1|1);
    }

    //ln,rn 分别为左右子区间大小
    void pushdown(int rt,int ln,int rn){
      if(segtree[rt].lazy){//有懒惰标记
        segtree[rt<<1].lazy+=segtree[rt].lazy;//更新左右子区间的值和懒惰标记
        segtree[rt<<1|1].lazy+=segtree[rt].lazy;
        segtree[rt<<1].val+=segtree[rt].lazy*ln;
        segtree[rt<<1|1].val+=segtree[rt].lazy*rn;

        segtree[rt].lazy=0;//** 清除标记
    }
}
    //区间更新-->延迟操作 ***(eg. a[L,R]+=c)  [L,R]为操作区间,[l,r]为当前区间，rt为节点编号
    // 结果：将完全包含于[L,R]的子区间更新并存lazy，其余等查询后再下推
void updateRange(int L,int R,int l,int r,int c,int rt){
```

```
        if(L<=l&&r<=R){//只有当前区间被完全包含才更新自己及子区间  ,部分包含先不更新，减少操作次数
            segtree[rt].val+=c*(r-l+1);     //更新区间总和
            segtree[rt].lazy+=c;      //根据不同操作更新懒惰标记
            return ;
        }
        int mid=(l+r)/2;
        //只做了lg级的pushdowm，其余用懒惰记录，查询时修改
        pushdown(rt,mid-l+1,r-mid);// 一次下推操作，才能准确 更新左右子节点 (非必要，不下退)
        if(L<=mid) updateRange(L,R,l,mid,c,rt<<1);//更新左子区间
        if(R>mid) updateRange(L,R,mid+1,r,c,rt<<1|1);   //更新右子区间
        pushup(rt);
    }

    //区间更新的区间（单点）查询
    int queryRange(int L,int R,int l,int r,int rt){
        if(L<=l&&r<=R)
            return  segtree[rt].val;

        if(L>r||R<l)
            return 0;

        int mid=(l+r)/2;
      pushdown(rt,mid-l+1,r-mid);//唯一不同,也是精华所在 ，查询到，必要时下推
      return  queryRange(L,R,l,mid,rt<<1)+queryRange(L,R,mid+1,r,rt<<1|1);
    }
}seg;
```

st表

```
#include <bits/stdc++.h>
#define LOCAL
//#define int long long
using namespace std;
const int N = 2e6 + 5;
int T, n,m;
int stmax[N][22];  //区间[ i , i + 2 ^j - 1 ] 的最大值
int stmin[N][22];
int logn[N];  // 存向下取整的log
int a[N];
void init() {  //预处理log2  初始化st[i][0]
  logn[0] = -1;  //这样 log[1] 为0
  for (int i = 1; i <= n; i++) {
    logn[i] = logn[i / 2] + 1;  // 也可以这样mn[i] = ((i & (i - 1)) == 0) ? mn[i- 1] + 1 : mn[i - 1]
    stmax[i][0] = stmin[i][0] = a[i];
  }


  //  nlogn 预处理st
```

```
  for (int j = 1; j <= logn[n]; j++) {  // 范围不用超过n  长度从小到大更新
    for (int i = 1; i + (1 << j) - 1 <= n; i++) {// 长度  2^j
      stmax[i][j] =    max(stmax[i][j - 1], stmax[i + (1 << j-1)][j - 1]);  //左右更
新
      stmin[i][j] = min(stmin[i][j - 1], stmin[i + (1 << (j - 1))][j - 1]);
    }
  }
}

int rmq_max(int L,int R){  // 查询
  int k=logn[R-L+1]; // >=长度的log
  return max(stmax[L][k],stmax[R-(1<<k)+1][k]);
  //*** 可重复更新
}
int rmq_min(int L,int R){  // 查询  相同
  int k=logn[R-L+1]; // >=长度的log
  return min(stmin[L][k],stmin[R-(1<<k)+1][k]);
  //*** 可重复更新
}

signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
#ifdef LOCAL
  freopen("data.in", "r", stdin);
  freopen("data.out", "w", stdout);
#endif
cin>>n>>m;

for(int i=1;i<=n;i++)cin>>a[i];
init();// 输入后初始化
for(int i=1;i<=m;i++){
int l,r;
cin>>l>>r;
cout<< rmq_max(l,r)<<endl;
}

  return 0;
}
```

单调栈

```
#include <bits/stdc++.h>
#define LOCAL
#define int long long
//单调栈求每个数之后第一个大于他的位置
// 倒序遍历  即找之前第一个大于他的位置
// 单调栈 每次 栈顶比a[i]小就弹出 此时栈顶为答案，将a[i]入栈
using namespace std;
```

```
const int N=3e6+5;
int T,n,a[N];
int f[N];
stack<int>S;  //栈中存下标
signed main(){
    std::ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);

#ifdef LOCAL
    freopen("data.in","r",stdin);
    freopen("data.out","w",stdout);
#endif
cin>>n;
for(int i=1;i<=n;i++)cin>>a[i];
for(int i=n;i>=1;i--){
        while(!S.empty()&&a[S.top()]<=a[i]){
            S.pop();
        }
    if(S.empty())f[i]=0;
    else f[i]=S.top();
    S.push(i);
}
for(int i=1;i<=n;i++)cout<<f[i]<<" ";
    return 0;
}
```

单调队列

单调队列--dp优化

定义：

什么是「单调队列」？顾名思义，「单调队列」就是队列内元素满足单调性的队列结构。

操作维护：（递增）

「单调队列」中「队尾」的操作与「单调栈」中「栈顶」的操作一致，即假设当前元素为 x，若队尾元素 <= x，则将 x 入队，否则不断弹出队首元素（单调栈是栈顶），直至队尾元素 <= x。

性质：求定长区间最值：队首/尾

有一个长为 n的序列 a，*以及一个大小为* k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

单调性：定长区间最值

维护长度为<=k的单调队列 数组模拟 stl不易实现 队尾删

最大值：单调递减队列，队首为max，每次队尾pop比a[i]小的

一些细节：

一般head tail可初始化为 0，0；-1，0 ；1,0;但当涉及某些前缀问题，(开始会调用q[head] q[0]的值，例2，4)只能0，0 //第一个元素初始化q[0]=0; i-1 下标从0开始

(无脑0，0)

例题：

例1：模板题：

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e6+ 5;
int T, n, k,a[N];
int q[N]; //单调队列  存的是下标  也可以多开一个数组存下标
void solve(){
    cin>>n>>k;
    //最小值;单调递增
    for(int i=1;i<=n;i++)cin>>a[i];
    int head=1,tail=0;  //   -1和0; 0,0也可
    for(int i=1;i<=n;i++){// 每次移动一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]>=a[i])tail--; //pop 掉队尾大于a[i]的 (在前面且小于)
        q[++tail]=i ; //新元素入队
        if(i>=k)cout<<a[q[head]]<<" "; //输出队首元素max
    }
    cout<<endl;
    //最大值同理
    head=1,tail=0;
    for(int i=1;i<=n;i++){// 每次一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]<=a[i])tail--; //pop 掉队尾小于a[i]的 (在前面且小于)
        q[++tail]=i ; //新元素入队
        if(i>=k) //窗口已进入
            cout<<a[q[head]]<<" "; // 输出队首元素max
    }


}
signed main() {
  std::ios::sync_with_stdio(false);
```

```
  cin.tie(0);
  cout.tie(0);

      solve();

  return 0;
}
```

例2：acw135. 最大子序和

输入一个长度为 n 的整数序列，从中找出一段长度不超过 m 的连续子序列，使得子序列中所有数的和最大。

思路：

滑动窗口变形：定长区间（<=len）最值问题

分析： 和最大 设a[i] 为 以i结尾的和最大的序列 则a[i] =sum[i] - sum[j] (i-k+1<j<i) 问题转化为 定长区间内sum[j] 的最小值minn ans=max(ans,sum[i]-minn)

做前缀和 用滑动窗口做法即可

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e6+ 5;
int T, n, k,a[N],sum[N];
int q[N]; //单调队列  存的是下标  也可以多开一个数组存下标
void solve(){
    cin>>n>>k;
    //最小值;单调递增
    memset(sum,0,sizeof sum);
    for(int i=1;i<=n;i++)cin>>a[i],sum[i]=sum[i-1]+a[i];
    int head=1,tail=0;   // head初始为0*因为这里是0~i-1的最小值,相当于sum下标从0开始
    // q[0]=0;
    int ans=-0x3f3f3f3f;
    for(int i=1;i<=n;i++){
                //长度限制
       if(head<=tail&&(i)-k+1>q[head])head++;  //前i-1个中最小值
       ans=max(ans,sum[i]-sum[q[head]]); //找i之前的最小值
       while(head<=tail&&sum[q[tail]]>=sum[i])tail--;   //加入i
       q[++tail]=i ;
    }
cout<<ans<<endl;
}
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
```

```
  cout.tie(0);
      solve();
  return 0;
}
```

例3.acw1088. 旅行问题

与例2类似，环形， 破换成链 2倍（2*n） 判断前缀和所有长度为n的区间最小值是否小于0

做法同例2

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

using namespace std;

const int N=2e6+10;

long long s[N*2];//前缀和
int q[N*2],o[N],d[N],mark[N];
//o[i]表示到i地点所需要的油,d[i]表示i到i+1消耗的油,mark[i]等于1时表示能环球旅行,0时不能

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

    for(int i=1;i<=n;i++) scanf("%d%d",&o[i],&d[i]);

    //计算前缀和
    for(int i=1;i<=n;i++) s[i]=s[i+n]=o[i]-d[i];//表示i地点加的油和到下一地点消耗的油的差
    for(int i=1;i<=2*n;i++) s[i]+=s[i-1];

    int hh=1,tt=0;
    for(int i=2*n;i>=1;i--)
    {
        //长度限制
        if(hh<=tt&&q[hh]>i+n-1) hh++;//窗口范围为n

        while(hh<=tt&&s[q[tt]]>=s[i]) tt--;//保持单调递增,q中大于s[i]的都出队
        q[++tt]=i;//s[i]入队

        if(i<=n&&s[q[hh]]>=s[i-1]) mark[i]=1;//最小值大于,那么可以环球旅行
    }

    //逆时针顺序
    hh=0,tt=-1;
    d[0]=d[n];//s[1]计算的时候需要
    for(int i=1;i<=n;i++) s[i]=s[i+n]=o[i]-d[i-1];
    for(int i=2*n;i>=0;i--) s[i]+=s[i+1];//因为为逆时针,所以s数组从后往前看
```

```
    for(int i=1;i<=2*n;i++)
    {
        if(hh<=tt&&q[hh]<i-n+1) hh++;//范围在n之内

        while(hh<=tt&&s[q[tt]]>=s[i]) tt--;//保持单调递增,q中大于s[i]的都出队
        q[++tt]=i;//s[i]进队
        if(i>n&&s[q[hh]]>=s[i+1]) mark[i-n]=1;//因为单调递减性,n范围内s[q[hh]]为最小值.
    }

    for(int i=1;i<=n;i++)
    if(mark[i]) printf("TAK\n");
    else printf("NIE\n");

    return 0;
}
```

####例4. hdu3530:Subsequence

题意:

题意: 给n个数，求一个最长连续子序列，在这个子序列中，最大值与最小值之差要在区间[m,k]内，输出这个子序列的长度。

思路:

用两个单调队列，一个递增，一个递减，然后枚举区间尾，不断维护两个队列，那么队首就是最大/小值，需要注意的是，当队首元素之差小于m时，不需要更新队列（因为如果后面有更大的元素进来，差可能就会大于等于m），而当队首元素之差大于k时，将两个队列中较小的队首出队，并用last标记，表示这是最新的被淘汰的下标，即所求区间的 前一个元素下标，于是答案ans=max（ans,i-last），所求区间为[last+1,i]。

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e5 + 5;
int T, n,m,k, a[N];
int q1[N],q2[N];
void solve(){
    for(int i=1;i<=n;i++)cin>>a[i];
    int head1=1,tail1=0,head2=1,tail2=0;
    memset(q1,0,sizeof q1);
    memset(q2,0,sizeof q2);
    int ans=0;
    int pos=1; //每次加入后满足的首元素
```

```
    for(int i=1;i<=n;i++){
        //无长度限制
        while(head1<=tail1&&a[q1[tail1]]>=a[i])tail1--;//递增  min
        q1[++tail1]=i;//入队
        while(head2<=tail2&&a[q2[tail2]]<=a[i])tail2--;//递减 max
        q2[++tail2]=i;
        while(a[q2[head2]]-a[q1[head1]]>k){ //本题关键操作 差值需要小于k  大于则后移head
            pos=q1[head1]<q2[head2]?q1[head1++]+1:q2[head2++]+1;//找更大的 后移
        }//满足的是跳出前后一位
        if(a[q2[head2]]-a[q1[head1]]>=m) //
            ans=max(ans,i-pos+1);   //
    }
cout<<ans<<endl;




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

  while(cin>>n>>m>>k)
      solve();
  return 0;
}
```

例5.1089. 烽火传递 （优化dp入门）

题目描述

给定一个长度为 n 的数组 w，以及一个正整数 m
其中 wi 表示第 i 个 元素 的 价值

求一种选择元素的 方案：

使得选择的 相邻元素 之间相差 不超过 m−1 个 不选 的元素
选择的元素总贡献 最小

状态表示：***

dp[i] 表示已 i 为右端点且选择i的合法方案代价最小值

状态计算

dp[i] =w[i] +min{dp[j] } (i-m<=j<i-1)

定长区间最小值问题 维护单调同时进行状态转移即可

优化:

O(n^2)->O(n)

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e5 + 5;
int T, n,m, w[N];
int dp[N];
int q[N];
void solve(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)  cin>>w[i];
    int head=0,tail=0; //这里只能0，0  i-1  下标从0开始 q[0]=0;
    for(int i=1;i<=n;i++){
        if(head<=tail&&(i-1)-q[head]+1>m)head++;  // i-1到q[head] 前m个
        dp[i]=dp[q[head]]+w[i]; //前面的最小值 转移
        while(head<=tail&&dp[q[tail]]>=dp[i])tail--;//维护单增
        q[++tail]=i;
    }
    //接下来找dp[i] 的最小值 可以枚举  但由于队首就是最小值 i=n时对应i-1及前m个
    //所以再滑动一位输出队首即可(i==n+1)
    if(n+1-q[head]>m)head++;
    cout<<dp[q[head]]<<endl;
}
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);


      solve();

  return 0;
}
```

例6.acw1090绿色通道（二分+单调队列优化）

就是加二分的上一题

题目描述

给定一个 正整数 m，以及一个长度为 n 的正整数 数组 w，其中 wi 为第 i 个元素的 价值

求一个选择元素的 方案，使得元素的 价值总和 不超过 m 且 相邻元素 的 间距最小

输出该 最小间距

分析

直接做不是很好做，不妨把问题转化为我们熟悉的模型来求解

显然，答案是存在 单调性 的：

任意比 答案 小的 间距 的选择方案，其 元素总和 必然超过 m
任意比 答案 大或相等的 间距，必然存在一个方案，使得 元素总和 小于等于 m
对于 22是显然的，我们可以在原合法方案上，删去一些数，从而实现 间距 变大的操作

对于 1，我们可以用 反证法：若小于 答案 的 间距 存在符合条件的选元素方案

则我们的 答案 应该是该 间距，这与原答案 矛盾

找出该 单调性，我们就可以上 二分 了

现问题就转化成了：在确定 最小间距 情况下，能否找出选择 元素总和 小于等于 mm 的方案

该问题 等价于： 在确定 最小间距 情况下，选择 元素总和 最小的方案 价值 是否 小于等于 m

状态表示：

dp[i] 表示已i为右端点且选i的方案 的最小代价

状态转移：

dp[i] =min{dp[j]} +w[i]; （ i-mid<j<i-1 ）

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e5 + 5;
int T, n,m, w[N];
int dp[N];
int q[N];
bool check(int mid){ //上一题的dp  check间距
     int head=0,tail=0; //这里只能0，0  i-1  下标从0开始 q[0]=0;
    for(int i=1;i<=n;i++){
        if(head<=tail&&(i-1)-q[head]+1>mid+1)head++;  //多加1  不超过
        dp[i]=dp[q[head]]+w[i]; //前面的最小值 转移
        while(head<=tail&&dp[q[tail]]>=dp[i])tail--;//维护单增
        q[++tail]=i;
    }
    //接下来找dp[i] 的最小值 可以枚举  但由于队首就是最小值 i=n时对应i-1及前m个
    //所以再滑动一位输出队首即可(i==n+1)
    if(n+1-q[head]>mid+1)head++;
    return dp[q[head]]<=m; //
}

void solve(){
```

```
    cin>>n>>m;
    for(int i=1;i<=n;i++)  cin>>w[i];
   int l=0,r=n+1;
    while(l<r){//二分答案
        int mid=(l+r)>>1;
        if(check(mid))r=mid;
        else l=mid+1;
    }
    cout<<l<<endl;
}
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);


      solve();

  return 0;
}
```

例7.修剪草坪

题目描述

给定一个长度为 n 的数组 w，其中 wi是第 i个元素的 贡献

我们可以选择的 数组 中的一些 元素，这些元素的 贡献总和 表示我们该种 方案 的 价值

但是，如果方案中出现选择了 连续相邻 且超过 m 个元素，则这些 连续相邻 的元素 贡献 归零

求解一种 方案，使得选择的 元素贡献总和 最大

分析

考虑用 动态规划 来求解本问题

由于 连续选择 超过 m 个元素时，这些元素的 贡献 为 0 （相当于没选）

而本题，所有的元素值都是 正整数，故我们的方案中，连续选择的元素数量 一定是 不超过 m 的

可以用 反证法 证明，如果方案中有超过 m 个连续元素，则我们不选中间的一个，使他断开，必然不会使方案变差
于是，我们就可以通过 ***最后一次没有选择的元素***，对 集合进行划分

闫氏DP分析法

状态表示

dp[i]：以 i为右端点的 前缀数组 的选择方案

状态计算：

fi=max{f[j-1]+si−sj}0≤i−j≤m (不选j) ---> fi=si+max{f[j-1]−s[j]} 0≤i−j≤m

由于 j是有范围的：， 于是问题就转化为 滑动窗口求极值 的问题了;我们用一个记录的 单调递减 的 f[j-1]−s[j] 队列 在 队头 维护一个 最大值 即可

边界：

dp[-1]=0

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

using namespace std;

typedef long long LL;

const int N = 1e5 + 10;

int n, m;
LL s[N], f[N];
int que[N];

LL g(int i)//即维护的  f[j-1]-s[j]
{
    return f[max(0, i - 1)] - s[i];  //f[-1]=f[0]
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) scanf("%lld", &s[i]), s[i] += s[i - 1]; //前缀和

    int hh = 0, tt = 0;   //  0 <= i - j <= m
    for (int i = 1; i <= n; i ++ )
    {
        if (hh <= tt && i - que[hh] > m) hh ++ ;
        f[i] = max(f[i-1],s[i] + g(que[hh]));       //正数的话不用max
        while (hh <= tt && g(i) >= g(que[tt])) tt -- ;
        que[ ++ tt] = i;  // i - j >= 0 故先入队

    }
    printf("%lld\n", f[n]);
    return 0;
}
```

例8.hdu3401：（二维dp）待补

题意：

股票在t天内每天买或卖或不作为，知道每一天每一支股票的买卖价格api,bpi和限购或卖的量asi,bsi，以及每天最多持有的股票数maxp，还有每次交易必须隔至少w天的限制，求最大的收益。