第 6 章　斜率优化动态规划

解题思路

1.太空飞船

【算法分析】

考虑先对问题进行转化。

可得平均数 $s_{avg}=\frac{\sum_{i=1}^{m}s_i}{K}$，方差 $v=\sum_{i=1}^{m}(s_i-s_{avg})^2$ 。

$$
\begin{aligned}
&K^2\times v\\
&=K^2\times\sum_{i=1}^{K}(s_i-s_{avg})^2\\
&=K^2\times\sum_{i=1}^{K}(s_i^2-2s_is_{avg}+s_{avg}^2)\\
&=K^2\sum_{i=1}^{K}s_i{}^2-K^2\times2\sum_{i=1}^{K}s_i\frac{\sum_{i=1}^{K}s_i}{K}+K^3\times\frac{(\sum_{i=1}^{K}s_i)^2}{K^2}\\
&=K^2\sum_{i=1}^{K}s_i{}^2-2K(\sum_{i=1}^{K}s_i)^2+K(\sum_{i=1}^{K}s_i)^2\\
&=K(K\sum_{i=1}^{K}s_i{}^2-(\sum_{i=1}^{K}s_i)^2)
\end{aligned}
$$

其中 $\sum_{i=1}^{K}s_i$ 就是 n 个舱室的总长度，是已知量。那么要使方差最小，即使得 $\sum_{i=1}^{K}s_i{}^2$ 最小。

由数据规模可知，要对 K 的取值进行分情况讨论：

记 $sum[i]$ 表示前 i 个舱室的长度之和，$ave=\frac{sum[n]}{K}$。

（1）当 $K=2$ 时：

为了使 K 段的平方和最小，我们显然要使每一段的和尽可能接近 ave。考虑将题目给定的序列倍长，然后枚举一个 i，每次找一个**最大**的 j，使得 $sum[i]-sum[j]$ 大于（或等于）ave，让 $j+1$ 到 i 分为一段，剩下的数分为另一段（因为是环），更新最优值。

当然，不一定存在能**正好**平均分的情况，这时要让这两段的其中一段多分一个数，判断让哪一段多分一个数更优，即边界情况。

时间复杂度为 $O(N)$。

（2）当 $K=3$ 时：

与 $K=2$ 同理，对于每个 i，找到另外两个分界点 j,k，考虑让 $j+1$ 到 i 分为一段，$i+1$ 到 k 分为一段，剩下的分为一段。依旧要考虑边界情况，尝试把 i,j,k 分别向左或向右移，取最优的结果。

时间复杂度为 $O(NK)$。

（3）当 $K>3$ 时：

考虑我们已经很难暴力使 K 段尽量均分，因为边界调整可能会使复杂度乘上一个很大的数字，注意到此时 N 很小，我们能否用 DP 来解决这个问题呢？

对于环形的情况，我们可以枚举把环断成链的位置，并把这个位置作为第 1 个舱室，最后取最优的结果。

设 DP 状态 $f[i][j]$ 表示处理到第 i 个部分第 j 个舱室时最小的平方和，则转移方程为：

$$f[i][j]=\min\{f[i-1][k]+(sum[j]-sum[k])^2\}$$

表示枚举第 i 个部分的长度之和，枚举状态复杂度为 $O(NK)$，转移复杂度为 $O(N)$，总的时间复杂度为 $O(N^3K)$，显然难以通过。

考虑怎样优化转移，这是一个斜率优化的经典模型。

先忽略状态的第一维和 $\min$，把转移方程展开：

$$f[j]=f[k]+sum[j]^2-2sum[j]\times sum[k]+sum[k]^2$$

把只含 k 的项移到右边，其余移到左边：

$$2sum[j]\times sum[k]+f[j]-sum[j]^2=f[k]+sum[k]^2$$

因此对于每个转移 k，可以看做平面直角坐标系中的点 $(sum[k],f[k]+sum[k]^2)$，使一条斜率为 $2sum[j]$ 的直线经过这些点，截距最小的点即为最优转移。

考虑对这些点维护一个下凸壳（斜率递增），则最优转移的点只可能在下凸壳上，并且对于一条特定的直线，最优的转移点为满足与凸壳中下一个点相连的斜率大于该直线斜率的最前方的点。

又因为这些点随着 k 的增大，横坐标递增，直线的斜率也随着 j 递增，我们可以用一个单调队列维护这个下凸壳，每次弹出与凸壳中下一个点相连的斜率小于等于 $2sum[j]$ 的队首元素，把点插入队尾时维护斜率递增的性质。

转移复杂度为 $O(1)$，总的时间复杂度为 $O(N^2K)$。

【参考程序】

```
#include <bits/stdc++.h>
using namespace std;
```

```
template <class T>
inline void read(T &res)
{
    char ch; bool flag = false; res = 0;
    while (ch = getchar(), !isdigit(ch) && ch != '-');
    ch == '-' ? flag = true : res = ch ^ 48;
    while (ch = getchar(), isdigit(ch))
        res = res * 10 + ch - 48;
    flag ? res = -res : 0;
}

typedef long long ll;
const ll Maxn = 1ll << 62;
const int N = 3e5 + 5;
int a[N], n, K; ll s[N], ans = Maxn;

template <class T>
inline void CkMin(T &x, T y) {if (x > y) x = y;}
template <class T>
inline T Max(T x, T y) {return x > y ? x : y;}
template <class T>
inline T Min(T x, T y) {return x < y ? x : y;}
template <class T>
inline T Sqr(T x) {return x * x;}

inline void work1()
{
    for (int i = 1; i <= n; ++i)
        a[i + n] = a[i];
    for (int i = 1, im = n << 1; i <= im; ++i)
        s[i] = s[i - 1] + a[i];
    int j = 1; ll tmp = s[n] >> 1;
    while (s[j + 1] - s[1] <= tmp) ++j;
    CkMin(ans, Sqr(s[j] - s[1]) + Sqr(s[n] - s[j] + s[1]));
    for (int i = 2; i <= n; ++i)
    {   //考虑分成 j+1 到 i，i+1 到 j 这 2 段
        while (s[j + 1] - s[i] <= tmp) ++j;
        CkMin(ans, Sqr(s[j] - s[i]) + Sqr(s[n] - s[j] + s[i]));
        CkMin(ans, Sqr(s[j - 1] - s[i]) + Sqr(s[n] - s[j - 1] + s[i]));
        //边界判断
    }
    cout << 1ll * K * K * ans - 1ll * K * s[n] * s[n] << endl;
}
```

```
ll f[25][805]; int h[805];

inline bool Slope1(int i, int j, int k, ll si)
{
    return (f[i][j] + s[j]*s[j] - f[i][k] - s[k]*s[k]) >= 2ll*si*(s[j]-s[k]);
}

inline bool Slope2(int i, int j, int k, int l) //维护凸壳斜率递增
{   //比较斜率时把除法转成乘法，避免精度误差
    return (f[i][j] + s[j] * s[j] - f[i][k] - s[k] * s[k]) * (s[k] - s[l])
         > (f[i][k] + s[k] * s[k] - f[i][l] - s[l] * s[l]) * (s[j] - s[k]);
}

inline void work3()
{
    ll ans = Maxn;
    for (int i = 1; i <= n; ++i)
        a[i + n] = a[i];
    for (int i = 1, im = n << 1; i <= im; ++i)
        s[i] = s[i - 1] + a[i];
    for (int st = 1; st <= n; ++st)
    {
        for (int i = 0; i <= K; ++i)
            for (int j = 0, jm = n << 1; j <= jm; ++j)
                f[i][j] = Maxn;
        f[0][st - 1] = 0;
        for (int i = 1; i <= K; ++i) //斜率优化 DP
        {
            int t = 1, w = 0;
            if (f[i - 1][st + i - 2] != Maxn)
                h[++w] = st + i - 2;
            for (int j = i - 1; j < n; ++j) //注意要保证转移合法
            {
                while (t < w && Slope1(i - 1, h[t], h[t + 1], s[st + j])) ++t;
                if (t <= w)
                    f[i][st + j] = f[i - 1][h[t]]
                          + (s[st + j] - s[h[t]]) * (s[st + j] - s[h[t]]);
                if (f[i - 1][st + j] != Maxn)
                {
                    while (t < w && Slope2(i-1, h[w-1], h[w], st+j)) --w;
                    h[++w] = st + j;
                }
            }
```

```
        }
        CkMin(ans, f[K][st + n - 1]);
    }
    cout << 1ll * K * K * ans - 1ll * K * s[n] * s[n] << endl;
}

inline ll calc(int j, int i, int k)
{   //计算分成 i+1 到 j，j+1 到 k，k+1 到 i 这 3 段的结果
    ll tx = s[j] + s[n] - s[k],
       ty = s[i] - s[j],
       tz = s[k] - s[i];
    return tx * tx + ty * ty + tz * tz;
}

inline void work2()
{
    int j = 0, k = 2;
    for (int i = 1; i <= n; ++i)
        s[i] = s[i - 1] + a[i];
    ll tmp = s[n] / 3;
    for (int i = 1; i <= n; ++i)
    {
        while (j < i && s[i] - s[j] > tmp) ++j;
        while (k <= n && s[k] - s[i] <= tmp) ++k;
        for (int a = Max(j - 1, 0), am = Min(j + 1, i); a <= am; ++a)
            for (int b = Max(k - 1, i), bm = Min(k + 1, n); b <= bm; ++b)
                CkMin(ans, calc(a, i, b));  //边界判断
    }
    cout << 1ll * K * K * ans - 1ll * K * s[n] * s[n] << endl;
}

int main()
{
    read(n); read(K);
    for (int i = 1; i <= n; ++i) read(a[i]);

    if (K == 2)
        work1();
    else if (n <= 400)
        work3();
    else if (K == 3)
        work2();
    return 0;
}
```

3.鏖战字符串

【算法分析】

采用 DP，定义状态 $f[i]$ 表示删除前 i 个字符所需的最少时间。

对于两种删除方式，有两种状态的转移方式。

令 $s_i=\sum_{j=1}^{i} dif_j$ ， $getmax(l,r)$ 表示区间 $[l,r]$ 出现次数最多的字符的出现次数，$[minlim,maxlim]$ 表示可使用第二种方式的最大出现次数区间，对应**题目中**的 $[l,r]$。

对两种转移分开讨论。

转移一：

$$f[i]=\min_{0\le j<i}\{f[j]+a(s_i-s_j)^2+b\}$$

将式子展开得到：

$$f[i]=\min_{0\le j<i}\{f[j]+as_i^2+as_j^2-2as_is_j+b\}$$

采用斜率优化，整理得：

$$f[j]+as_j^2=2as_is_j+f[i]-as_i^2-b$$

以 s_j 为横坐标， $f[j]+as_j^2$ 为纵坐标建立平面直角坐标系，那么上面的式子就表示一条以 $2as_i$ 为斜率， $f[i]-as_i^2-b$ 为截距的直线。

也就是说，决策候选集合可以表示为坐标系上的点集 $\{(s_j,f[j]+as_j^2)\mid 0\le j<i\}$。那么每个待求解的状态 $f[i]$ 都对应着一条直线的截距，而这条直线的斜率是定值 $2as_i$，截距未知。令直线过每个决策点 $(s_j,f[j]+as_j^2)$，就能求出直线的截距，当截距最小化时，就能得到对应 $f[i]$ 的最小值。

维护一个斜率递增的下凸壳，对于每个待求解的状态 $f[i]$，找到凸壳上的一个点，使得凸壳上该点左侧线段的斜率比 $2as_i$ 小，该点右侧线段的斜率比 $2as_i$ 大，这样该直线与凸壳相切于该点，直线截距最小化，该点就是最优的决策点。

我们只要维护一个斜率递增的单调队列来维护下凸壳。

由于每个决策点的横坐标 s_j 不降，每次插入决策点只要在队尾插入，并维护斜率单调递增的性质即可。同时注意，因为删除难度可能为 0，相邻两点的横坐标 s_j 可能相同，那

么在这种情况下，我们应当保留纵坐标 $f[j]+as_j^2$ 最小的决策点。

由于 $2as_i$ 是递增的，不断将右侧线段斜率小于 $2as_i$ 的队首决策点弹出（因为斜率递增，该决策点必然不会在后面的转移中成为最优决策）。

每次转移取队头元素即可，这样每个决策点最多出队入队 1 次，总时间复杂度为 $O(n)$。

转移二：

$$f[i]=\min_{minlim\le getmax(j+1,i)\le maxlim}\{f[j]+c(s_i-s_j)+d\}$$

我们可以发现以下几个性质：

性质 1：对于每个 i，满足 $minlim\le getmax(j+1,i)\le maxlim$ 的决策 j 构成了一个区间。

性质 2：我们假设合法的决策 j 组成区间 $[l,r]$，那么随着 i 的增加，l 和 r 都是单调不降的。

性质 3：在 i 较小时，可能没有合法的决策。但是当出现合法决策后，随着 i 的增大，合法决策始终存在。

这三个性质的证明比较显然，都是因为加入元素不会使最大出现次数减小，删除元素不会使最大出现次数增大。

根据这三个性质，我们可以维护两个指针 l 和 r 来表示合法区间，将转移方程中的括号拆开后使用单调队列优化转移，并且每次动态维护时，利用前缀和，暴力枚举每个字符，判断区间的最大出现次数。因为指针最多移动 n 次，所以时间复杂度就是 $O(26n)$。

注意两种转移需要同时进行，总时间复杂度为 $O(26n)$。

【参考程序】

```
#include <bits/stdc++.h>
typedef long long s64;

template <class T>
inline void read(T &x)
{
    static char ch;
    static bool opt;
    while (!isdigit(ch = getchar()) && ch != '-');
    x = (opt = ch == '-') ? 0 : ch - '0';
    while (isdigit(ch = getchar()))
        x = x * 10 + ch - '0';
    if (opt) x = ~x + 1;
}

inline void puts64(s64 x)
```

```
{
    static char buf[25];
    int t = 0;
    if (x == 0)
        putchar('0'), putchar('\n');
    else
    {
        for (; x; x /= 10)
            buf[++t] = x % 10 + '0';
        for (; t; --t)
            putchar(buf[t]);
        putchar('\n');
    }
}

const int MaxN = 1e5 + 5;
const int M = 26;

int n, A, B, C, D, minlim, maxlim, dif[MaxN];
//minlim 和 maxlim 对应题目中的 l 和 r，表示能使用第二种操作的合法区间
[minlim,maxlim]
int q1[MaxN], q2[MaxN], head1, head2, tail1, tail2;
int pre_ch[MaxN][M], l, r;
//pre_ch[i][c]表示[1,i]中 c 字符的出现次数
char str[MaxN];
s64 f[MaxN], s[MaxN], X[MaxN], Y[MaxN];
//X[i]和 Y[i]分别表示决策点 i 的横纵坐标

template<class T>
inline void relax(T &x, const T &y)
{
    if (x < y)
        x = y;
}

template<class T>
inline void tense(T &x, const T &y)
{
    if (x > y)
        x = y;
}

inline int getmax(const int &x, const int &y)
{   //获取区间[x,y]中出现次数最多的字符的次数
```

```
    int res = 0;
    for (int i = 0; i < M; ++i)
        relax(res, pre_ch[y][i] - pre_ch[x - 1][i]);
    return res;
}

int main()
{
    memset(f, 0x3f, sizeof(f));
    read(n),read(A),read(B),read(C),read(D),read(minlim),read(maxlim);
    scanf("%s", str + 1);
    for (int i = 1; i <= n; ++i)
    {
        read(dif[i]);
        s[i] = s[i - 1] + dif[i];   //维护删除难度的前缀和
        for (int j = 0; j < M; ++j)
            pre_ch[i][j] = pre_ch[i - 1][j];
        ++pre_ch[i][str[i] - 'a'];  //维护字符前缀和
    }
    f[0] = 0;
    q1[head1 = tail1 = 1] = 0;
    head2 = 1, tail2 = 0;
    l = 0, r = -1;
    for (int i = 1; i <= n; ++i)
    {
        while (head1 < tail1 && Y[q1[head1 + 1]] - Y[q1[head1]] < 2 * A *
s[i] * (X[q1[head1 + 1]] - X[q1[head1]]))
            ++head1;
        f[i] = f[q1[head1]] + A * s[i] * s[i] + A * s[q1[head1]] * s[q1[head1]]
- 2 * A * s[i] * s[q1[head1]] + B;

        if (minlim <= maxlim)  //如果存在可使用第二种操作的情况
        {
            while (l < i && getmax(l + 1, i) > maxlim) ++l;
            if (r < l) r = l;
            while (r < i && getmax(r + 1, i) >= minlim)
            {
                while (head2 <= tail2 && f[q2[tail2]] - C * s[q2[tail2]] >=
f[r] - C * s[r])
                    --tail2;
                q2[++tail2] = r;
                //在 q2 中维护第二种操作决策的单调队列
                ++r;
            }
```

```
            --r;
            //动态更新满足第二种操作条件的决策区间[l,r]
            while (head2 <= tail2 && q2[head2] < l)
                ++head2;
            if (head2 <= tail2)
                tense(f[i], f[q2[head2]] - C * s[q2[head2]] + C * s[i] + D);
            //如果存在合法的第二种操作决策，则用其更新答案
        }

        X[i] = s[i];
        Y[i] = f[i] + A * s[i] * s[i];
        if (head1 <= tail1 && X[q1[tail1]] == X[i])
        {   //x 坐标相等保留 y 坐标最小的决策点
            if (Y[q1[tail1]] < Y[i])
                continue;
            else
                --tail1;
        }
        while (head1 < tail1 && (Y[q1[tail1 - 1]] - Y[q1[tail1]]) *
(X[q1[tail1]] - X[i])
                        > (Y[q1[tail1]] - Y[i]) * (X[q1[tail1 - 1]] -
X[q1[tail1]]))
            --tail1;
        q1[++tail1] = i;
        //在 q1 中维护操作一的决策点形成的下凸壳
    }
    for (int i = 1; i <= n; ++i)
        puts64(f[i]);
    return 0;
}
```

4.分数

【算法分析】

先考虑没有一次询问时如何求获得分数的最大值。

容易想到设 f[i]表示前 i 个数中获得分数的最大值。

转移（s 为 T 的前缀和）：

$$f[i]=\max(f[i-1],\max_{j=0}^{i-1}\{f[j]+\frac{(i-j)(i-j+1)}{2}-s[j]+s[i]\})$$

为避免实数运算，将等式两边乘上 2，最后计算完再除以 2。

$$f[i]=\max(f[i-1],\max_{j=0}^{i-1}\{f[j]+(i-j)(i-j+1)-s[j]+s[i]\}$$

展开后移项得到：

$$f[i]=\max(f[i-1],\max_{j=0}^{i-1}\{f[j]+j^2-j-s[j]-2ij\}+i^2+i+s[i]$$

显然可以斜率优化。决策 i 为点(i,f[i]+i²-i-s[i])。

但实现时要注意：所有点的 x 坐标递增且需要维护平面点集的上凸壳斜率递减，每次求 f[i]要求一条斜率为 i 且与上凸壳有交点的直线使其纵截距最大，i 是递增的，故每次求 f[i]时要从**凸壳的右边而不是左边**删掉点来找最优决策，即用**单调栈**维护凸壳。

有询问时又该怎么做呢？

如果第 P_i 个数没有被选出，那么就比较好处理。记录 g[i]为第 i 个到第 n 个数中获得分数的最大值（同理用 DP 求出），第 P_i 个数没有被选出的分数最大值为 $f[P_i-1]+g[P_i+1]$。

否则可以找到一种标记方案，使得第 P_i 个数一定被选出，把这种方案的收益减去原来第 P_i 个数再加上 X_i 即可。

于是我们要求出 h[i]表示第 i 个数一定被选出的最大收益。

考虑**分治**。

分治区间为[l,r]时，考虑区间中点 mid，求对于任何一个[l,r]内的 i，i 在区间[l,r]内且 i 所在的**极长标记区间**左端点在[l,mid]且右端点在[mid+1,r]时选出的最大收益。

i 所在极长标记区间的定义：满足[l,r]包含 i，[l,r]内数都被标记且 r-l 最大（区间最长）的区间[l,r]。

我们考虑求出 R[i]（i 在区间[mid+1,r]内）表示 i 所在极长标记区间左端点在[l,mid]内，右端点为 i 时选出的最大收益。

容易得出：

$$R[i]=\max_{j=l-1}^{mid-1}\{f[j]+(i-j)(i-j+1)-s[j]+s[i]\}$$

发现了什么？和原转移方程大致一样！

将[l,mid]内的决策点构成凸壳后在[mid+1,r]内和 f 一样求解即可得到 R。

同样地，将[mid+1,r]内的决策点构成凸壳后在[l,mid]求解可以得到 L[i]（i 在区间[l,mid]内）表示 i 所在极长标记区间右端点在[mid+1,r]内，左端点为 i 时选出的最大收益。

将 L 求一遍前缀最大值，R 求一遍后缀最大值，就能用这两个数组更新 h。

时间复杂度为 O(NlogN)。

【参考程序】

```
#include <bits/stdc++.h>
#define For(i, a, b) for (i = a; i <= b; i++)
#define Rof(i, a, b) for (i = a; i >= b; i--)
using namespace std;

inline int read() {
    int res = 0; bool bo = 0; char c;
    while (((c = getchar()) < '0' || c > '9') && c != '-');
    if (c == '-') bo = 1; else res = c - 48;
    while ((c = getchar()) >= '0' && c <= '9')
        res = (res << 3) + (res << 1) + (c - 48);
    return bo ? ~res + 1 : res;
}
```

```
typedef long long ll;
const int N = 3e5 + 5;
const ll INF = 1ll << 62;
int n, t[N], m, Q[N], top;
ll sum[N], f[N], g[N], X[N], Y[N], h[N], L[N], R[N];

bool check(int i, int j, int k) {
    return (X[j]-X[i]) * (Y[k]-Y[i]) - (X[k]-X[i]) * (Y[j]-Y[i]) >= 0;
}

void slopedp(ll *f) {  //DP
    int i; f[0] = 0;
    X[0] = 0; Y[0] = 0;
    Q[top = 1] = 0;
    For (i, 1, n) {
        while (top > 1 && Y[Q[top]] - X[Q[top]] * i * 2 <=
            Y[Q[top - 1]] - X[Q[top - 1]] * i * 2) top--;
        f[i] = max(f[i - 1], Y[Q[top]] - X[Q[top]] * i * 2
            - sum[i] + 1ll * i * i + i);
        X[i] = i; Y[i] = f[i] + 1ll * i * i - i + sum[i];
        while (top > 1 && check(Q[top - 1], Q[top], i)) top--;
        Q[++top] = i;
    }
}
void solve(int l, int r) {  //分治
    if (l == r) return (void)
        (h[l] = max(h[l], f[l - 1] + g[l + 1] + 2 - t[l]));
    int i, mid = l + r >> 1;
    X[l-1] = l - 1;
    Y[l-1] = f[l-1] + 1ll * (l-1) * (l-1) - (l-1) + sum[l-1];
    Q[top = 1] = l - 1;
    For (i, l, mid - 1) {    //求 R
        X[i] = i; Y[i] = f[i] + 1ll * i * i - i + sum[i];
        while (top > 1 && check(Q[top - 1], Q[top], i)) top--;
        Q[++top] = i;
    }
    For (i, mid + 1, r) {
        while (top > 1 && Y[Q[top]] - X[Q[top]] * i * 2 <=
            Y[Q[top - 1]] - X[Q[top - 1]] * i * 2) top--;
        R[i] = Y[Q[top]] - X[Q[top]] * i * 2 - sum[i] + 1ll * i * i + i;
    }
    For (i, l, mid) if (i != l + r - i) swap(t[i], t[l + r - i]);
    if (!(r - l & 1)) mid--;
    For (i, l, r) sum[i] = sum[i - 1] + t[i];
```

```
    X[l - 1] = l - 1; Y[l - 1] = g[r + 1] + 1ll * (l - 1) * (l - 1)
        - (l - 1) + sum[l - 1];
    Q[top = 1] = l - 1;
    For (i, l, mid - 1) { //求 L
        X[i] = i; Y[i] = g[l + r - i] + 1ll * i * i - i + sum[i];
        while (top > 1 && check(Q[top - 1], Q[top], i)) top--;
        Q[++top] = i;
    }
    For (i, mid + 1, r) {
        while (top > 1 && Y[Q[top]] - X[Q[top]] * i * 2 <=
            Y[Q[top - 1]] - X[Q[top - 1]] * i * 2) top--;
        L[l+r-i] = Y[Q[top]] - X[Q[top]]*i*2 - sum[i] + 1ll*i*i + i;
    }
    For (i, l, mid) if (i != l + r - i) swap(t[i], t[l + r - i]);
    mid = l + r >> 1;
    For (i, l, mid) L[i] += f[i - 1];
    For (i, mid + 1, r) R[i] += g[i + 1];
    For (i, l, r) sum[i] = sum[i - 1] + t[i];
    For (i, l + 1, mid) L[i] = max(L[i], L[i - 1]);
    Rof (i, r - 1, mid + 1) R[i] = max(R[i], R[i + 1]); //求前&后缀最大值
    For (i, l, mid) h[i] = max(h[i], L[i]);
    For (i, mid + 1, r) h[i] = max(h[i], R[i]);
    solve(l, mid); solve(mid + 1, r);
}

int main() {
    int i, p, x;
    n = read();
    For (i, 1, n) t[i] = read() << 1;
    For (i, 1, n) sum[i] = sum[i - 1] + t[i];
    slopedp(f);
    For (i, 1, n >> 1) swap(t[i], t[n - i + 1]);
    For (i, 1, n) sum[i] = sum[i - 1] + t[i];
    slopedp(g);
    For (i, 1, n >> 1) swap(t[i], t[n - i + 1]);
    For (i, 1, n) sum[i] = sum[i - 1] + t[i];
    For (i, 1, n >> 1) swap(g[i], g[n - i + 1]);
    For (i, 1, n) h[i] = -INF;
    m = read();
    solve(1, n);
    while (m--) {
        p = read(); x = read() << 1;
        printf("%lld\n", max(f[p - 1] + g[p + 1],
            h[p] + t[p] - x) >> 1);
```

```
    }
    return 0;
}
```