T1 简单序列

我们将左括号看作 1，右括号看作 -1，则一个合法的括号序列需要满足：

- 所有括号的总和为 0。
- 每个前缀和均不小于 0。

我们先统计出串 s 的总和 a 以及最小的前缀和 b，然后枚举串 p 的长度 i 以及总和 j，考虑到需要满足第二个条件，那么 j 需要满足 $j+b\geq 0$。记 $f_{i,j}$ 为长度为 i，总和为 j 的括号序列数量，此时 p 的方案数为 $f_{i,j}$，q 的方案数为 $f_{n-m-i,j+a}$，将它们的乘积计入答案即可。

f 可以通过一个简单的 DP 求出，时空复杂度均为 $O((n-m)^2)$。

核心代码

```
    f[0][0] = 1;
    for (int i = 1; i <= n - m; ++i)
        for (int j = 0; j <= i; ++j)
        {
            f[i][j] = 0;
            if (j > 0)
                f[i][j] = add(f[i][j], f[i - 1][j - 1]);
            if (j < i - 1)
                f[i][j] = add(f[i][j], f[i - 1][j + 1]);
        }
```

T2 筹备计划

假设不存在 $type=3$ 和 $type=4$ 的操作，也就是说所有地方都可以举行校庆，这样最优的位置一定是中位数的位置。

如果有了 $type=3$ 和 $type=4$ 的操作，也就是某些地方是不可以举行校庆的，最优的位置就不一定是中位数的位置，因为这个位置可能不能举行校庆，那么只能另外寻找位置来举行校庆。

通过感性的理解可以发现，最优的位置一定是左边距离中位数或者右边距离中位数位置最近的那两个位置中更优的一个。既然知道了这个，就考虑用线段树维护一些什么。

1. 维护这个区间里面有多少个位置可以举办校庆。
2. 维护区间里面人数的总和。
3. 为了方便统计在某个位置举行校庆的距离和，还需要维护 $a_i\cdot i$ 和 $a_i\cdot(n-i+1)$ 的前缀和，这个可以用树状数组来维护。

每次寻找最优位置，先在线段树上二分，得到中位数的位置 pos，注意这里的中位数位置可能是一个区间。 如果可以在这个区间里面举行校庆，这就一定是最优的。如果不行，就在线段树里面找到距离左端点最近且可以举办校庆的位置 $pos1$，以及距离右端点最近且可以举办校庆的位置 $pos2$，通过树状数组维护的前缀和，分别计算出它们的距离和，然后看看哪一个更优。

核心代码

```
struct Part {
    LL a,b;
    LL sum,space,flag;
} tree[maxn*5];
struct Segment_tree {
    inline void Pushup(LL v) {
```

```
        tree[v].sum=tree[v<<1].sum+tree[v<<1|1].sum;
        tree[v].space=tree[v<<1].space+tree[v<<1|1].space;
    }
    inline void Pushdown(LL v) {
        if(tree[v].flag==1) {
            tree[v<<1].flag=tree[v<<1|1].flag=1;
            tree[v].flag=0;
            LL L=tree[v].a,R=tree[v].b;
            LL mid=(L+R)>>1;
            tree[v<<1].space=mid-L+1;
            tree[v<<1|1].space=R-mid;
        }
        if(tree[v].flag==2) {
            tree[v<<1].flag=tree[v<<1|1].flag=2;
            tree[v].flag=0;
            tree[v<<1].space=tree[v<<1|1].space=0;
        }
    }
    void Build(LL v,LL L,LL R) {
        tree[v].a=L;
        tree[v].b=R;
        tree[v].sum=tree[v].space=tree[v].flag=0;
        if(L==R) {
            tree[v].sum=a[L];
            return;
        }
        LL mid=(L+R)>>1;
        Build(v<<1,L,mid);
        Build(v<<1|1,mid+1,R);
        Pushup(v);
    }
    void Open(LL v,LL L,LL R) {
        if(tree[v].a>R||tree[v].b<L)return;
        if(tree[v].a>=L&&tree[v].b<=R) {
            tree[v].space=tree[v].b-tree[v].a+1;
            tree[v].flag=1;
            return;
        }
        Pushdown(v);
        Open(v<<1,L,R);
        Open(v<<1|1,L,R);
        Pushup(v);
    }
    void Close(LL v,LL L,LL R) {
        if(tree[v].a>R||tree[v].b<L)return;
        if(tree[v].a>=L&&tree[v].b<=R) {
            tree[v].space=0;
            tree[v].flag=2;
            return;
        }
        Pushdown(v);
        Close(v<<1,L,R);
        Close(v<<1|1,L,R);
        Pushup(v);
    }
    void Add(LL v,LL x,LL d) {
        if(tree[v].a>x||tree[v].b<x)return;
        if(tree[v].a==x&&tree[v].b==x) {
```

```
            tree[v].sum+=d;
            return;
        }
        Pushdown(v);
        Add(v<<1,x,d);
        Add(v<<1|1,x,d);
        Pushup(v);
    }
    void Find(LL v,LL L,LL R,LL s) {
        if(L==R) {
            pos2=L;
            return;
        }
        LL mid=(L+R)>>1;
        if(tree[v<<1].sum>=s)Find(v<<1,L,mid,s);
        else Find(v<<1|1,mid+1,R,s-tree[v<<1].sum);
    }
    void Get1(LL v,LL L,LL R) {
        Pushdown(v);
        if(L==R) {
            if(tree[v].space>0)pos=min(pos,L);
            return;
        }
        LL mid=(L+R)>>1;
        if(tree[v<<1].space>0)Get1(v<<1,L,mid);
        else Get1(v<<1|1,mid+1,R);
    }
    void Get2(LL v,LL L,LL R) {
        Pushdown(v);
        if(L==R) {
            if(tree[v].space>0)pos=max(pos,L);
            return;
        }
        LL mid=(L+R)>>1;
        if(tree[v<<1|1].space>0)Get2(v<<1|1,mid+1,R);
        else Get2(v<<1,L,mid);
    }
    void Right(LL v,LL L,LL R) {
        if(tree[v].a>R||tree[v].b<L)return;
        if(tree[v].a>=L&&tree[v].b<=R) {
            Get1(v,tree[v].a,tree[v].b);
            return;
        }
        Pushdown(v);
        Right(v<<1,L,R);
        Right(v<<1|1,L,R);
        Pushup(v);
    }
    void Left(LL v,LL L,LL R) {
        if(tree[v].a>R||tree[v].b<L)return;
        if(tree[v].a>=L&&tree[v].b<=R) {
            Get2(v,tree[v].a,tree[v].b);
            return;
        }
        Pushdown(v);
        Left(v<<1,L,R);
        Left(v<<1|1,L,R);
        Pushup(v);
```

```
    }
    void Ask(LL v,LL L,LL R) {
        if(tree[v].a>R||tree[v].b<L)return;
        if(tree[v].a>=L&&tree[v].b<=R) {
            pos+=tree[v].sum;
            return;
        }
        Pushdown(v);
        Ask(v<<1,L,R);
        Ask(v<<1|1,L,R);
        Pushup(v);
    }
} Line;
```

T3 简单区间

$$f(L,R)=\sum_{i=L}^{R}a_i-\max_{i=L}^{R}\{a_i\}$$

上述式子的 $\max\limits_{i=L}^{R}\{a_i\}$ 并不好处理，因此我们考虑以 a_i 为键值，从大到小建立一棵笛卡尔树。此时我们设对应笛卡尔树的一个节点所代表的一个区间为 $[L,R]$，设这个节点所代表的 a_i 最大的节点为 K，那么明显所有的为 $[L,R]$ 子区间且跨过点 K 的区间的 $\max\limits_{i=L}^{R}\{a_i\}$ 必然为 a_K。

因此我们在笛卡尔树上的每个节点，都只要处理横跨笛卡尔树该节点对应位置并且是该节点所代表区间的子区间的区间所贡献的答案即可。

此处也可以用分治来理解。考虑当前分治区间 $[L,R]$，设该区间的 a_i 最大的位置为 K，那么我们可以把答案进行转化：$solve(L,R)=solve(L,K-1)+solve(K+1,R)+calc(L,K,R)$，其中 $calc(L,K,R)$ 表示横跨点 K 且包含在区间 $[L,R]$ 内的区间对答案的贡献。明显，若考虑区间不横跨 K，那么明显在 $solve(L,K-1)$ 或 $solve(K+1,R)$ 中会被计算答案。这里与上面的笛卡尔树实际上是完全等同的。

因此我们接下来要考虑如何计算被包含在区间 $[L,R]$ 中且横跨 K 的区间的贡献。

算法 1

明显，这样的一个区间 $[l,r]$ 必然满足 $sum_r-sum_{l-1}\equiv a_K\pmod k$。此时我们可以考虑用线段树合并来维护 sum_r，然后枚举每一个 sum_{l-1}，并在线段树中查找。

显然直接这样做会当场超时，因此考虑启发式合并。我们同时维护 a_i 数组的前缀和 pre_i 与后缀和 suf_i，此时一个区间 $[l,r]$ 想产生贡献，必然满足 $pre_r-pre_{l-1}\equiv a_K\pmod k\iff suf_l-suf_{r+1}\equiv a_K\pmod k$。因此，我们考虑当前节点的两个儿子的子树大小，若左儿子子树更小，则利用 $pre_r-pre_{l-1}\equiv a_K\pmod k$ 判定；若右儿子子树更小，则利用 $suf_l-suf_{r+1}\equiv a_K\pmod k$ 判定。该做法时间复杂度为 $O(n\log^2 n)$。

核心代码

```
ll ans=0;
void getl(int u,int val,int F)
{
    if(!u)return;
    ans+=Pre.Query(F,(val+pre[u-1])%Mod);
    getl(lc[u],val,F);getl(rc[u],val,F);
}
void getr(int u,int val,int F)
{
```

```
    if(!u)return;
    ans+=Suf.Query(F,(val+suf[u+1])%Mod);
    getr(lc[u],val,F);getr(rc[u],val,F);
}
void dfs2(int u)
{
    if(!u)return;
    dfs2(lc[u]);dfs2(rc[u]);
    ans+=Pre.Query(rc[u],pre[u]);
    ans+=Suf.Query(lc[u],suf[u]);
    if(Size[lc[u]]>Size[rc[u]])getr(rc[u],a[u]%Mod,lc[u]);
    else getl(lc[u],a[u]%Mod,rc[u]);
    if(lc[u])Pre.Merge(u,lc[u]),Suf.Merge(u,lc[u]);
    if(rc[u])Pre.Merge(u,rc[u]),Suf.Merge(u,rc[u]);
}
```

算法 2

但事实上我们有更加优秀的做法：每个节点计算贡献时相当于询问 $[l,r]$ 范围中，某个数 x 在 pre_i 或 suf_i 中的出现次数。我们可以把这些询问拆分成两个询问并记录下来离线处理。根据启发式合并的思想，询问的个数不超过 $n\log n$，因此时间复杂度为 $O(n\log n)$。

核心代码

```
ll ans=0;
void insertP(int l,int r,int val)
{
    Pre[r].push_back(make(1,val));
    Pre[l-1].push_back(make(-1,val));
}
void insertS(int l,int r,int val)
{
    Suf[r].push_back(make(1,val));
    Suf[l-1].push_back(make(-1,val));
}
void getl(int u,int val,int l,int r)
{
    if(!u)return;
    insertP(l,r,(val+pre[u-1])%Mod);
    getl(lc[u],val,l,r);
    getl(rc[u],val,l,r);
}
void getr(int u,int val,int l,int r)
{
    if(!u)return;
    insertS(l,r,(val+suf[u+1])%Mod);
    getr(lc[u],val,l,r);
    getr(rc[u],val,l,r);
}
void dfs2(int u)
{
    if(!u)return;
    dfs2(lc[u]);dfs2(rc[u]);
    if(rc[u])insertP(L[rc[u]],R[rc[u]],pre[u]);
    if(lc[u])insertS(L[lc[u]],R[lc[u]],suf[u]);
    if(Size[lc[u]]>Size[rc[u]])getr(rc[u],a[u]%Mod,L[lc[u]],R[lc[u]]);
    else getl(lc[u],a[u]%Mod,L[rc[u]],R[rc[u]]);
```

```
}
```

T4 景区旅行

算法一

我们发现钱的范围 $q_i \leq n^2$，而路程的范围 d_i 很大，并且同时记录当前油量与钱量显然做不到。但是每次加油以后，油量的状态和之前无关，所以考虑以每次加油的位置为状态进行 DP。

设 $f(i,q)$ 表示当前位于景点 i，下次在 i 位置加油，剩余钱数为 q 时，之后的最大路程。

当 $q < p_i$ 时，$f(i,q) = 0$，否则，转移时枚举第二次加油的位置，得到

$$f(i,q) = \max\{f(j,q-p_i) + w(i,j,\min\{c_i, C\})\}$$

其中 $w(i,j,c)$ 为从 i 到 j 经过不超过 c 条道路的最大路程。这可以用 DP 预处理，枚举 i 出发的第一条路 e，得

$$w(i,j,c) = \max_{a_e=i}\{w(b_e,j,c-1) + l_e\}$$

边界是 $w(i,i,0) = 0$，$w(i,j,0) = -\infty (i \neq j)$。

询问时，只需枚举最小的 $1 \leq q_i$ 使得 $f(s_i,q) \geq d_i$ 即可。

时间复杂度 $O(n^4 + nmC + Tn^2)$，期望得分 75 分。

算法二

现在的瓶颈在于预处理 $w(i,j,c)$，我们发现每次 c 只会减少 1，于是可以用倍增预处理。

设 $g(i,j,k)$ 为从 i 到 j 经过不超过 2^k 条道路的最大路程，则当 $k > 0$ 时：

$$g(i,j,k) = \max_{x}\{g(i,x,k-1) + g(x,j,k-1)\}$$

注意到我们只要对所有 i,j 处理出 $w(i,j,\min(x_i,C))$，令 $c = \min(x_i,C)$，那么每次提取 c 的一个二进制位 2^k，可得

$$w(i,j,c) = \max(\max_{x}\{w(i,x,c-2^k) + g(x,j,k)\}, w(i,j,c-2^k))$$

只需倍增 $\log C$ 轮即可。

时间复杂度 $O(n^4 + n^3 \log C + Tn^2)$，期望得分 95 分。

算法三

只要把上述算法中，询问时枚举最小的 q 使得 $f(s_i,q) \geq d_i$ 这一步的枚举换成二分，就能把最后一步的 $O(Tn^2)$ 优化到 $O(T \log n^2)$。

时间复杂度 $O(n^4 + n^3 \log C + T \log n^2)$，期望得分 100 分。

核心代码

```
void init()
{
    memset(g,-1,sizeof(g));
    memset(w,-1,sizeof(w));
    for(int u=1;u<=n;u++)
    for(int i=G.head[u];i;i=G.nxt[i])
    {
        int v=G.to[i];
        g[u][v][0]=max(g[u][v][0],G.dis[i]);
```

```
    }
    for(int j=1;j<=logN;j++)
    for(int u=1;u<=n;u++)
    for(int v=1;v<=n;v++)
    {
        g[u][v][j]=g[u][v][j-1];
        for(int k=1;k<=n;k++)
        if(g[u][k][j-1]!=-1&&g[k][v][j-1]!=-1)
        g[u][v][j]=max(g[u][v][j],g[u][k][j-1]+g[k][v][j-1]);
    }

    for(int now=1;now<=n;now++)
    {
        int nowx=X[now];w[now][now]=0;
        for(int j=logN;j>=0;j--)
        if(nowx>=(1<<j))
        {
            nowx-=(1<<j);
            for(int u=1;u<=n;u++)ow[now][u]=w[now][u];
            for(int u=1;u<=n;u++)
            for(int v=1;v<=n;v++)
            if(ow[now][u]!=-1&&g[u][v][j]!=-1)
            w[now][v]=max(w[now][v],ow[now][u]+g[u][v][j]);
        }
    }

    memset(f,128,sizeof(f));
    for(int i=0;i<=n*n;i++)
    for(int u=1;u<=n;u++)
    {
        if(V[u]>i){f[u][i]=0;continue;}
        for(int v=1;v<=n;v++)
        if(w[u][v]!=-1)f[u][i]=max(f[u][i],f[v][i-V[u]]+w[u][v]);
    }
    for(int u=1;u<=n;u++)
    for(int i=1;i<=n*n;i++)
    Max[u][i]=max(f[u][i],Max[u][i-1]);
}
```