# 2023暑期CSP-S/NOIP模拟赛2 题解

## T1 湮灭反应

给定一个数列，问从其中取出一段区间能获得的和的绝对值最小是多少，以及和的绝对值最小情况下的方案数。

### 40%做法：

枚举区间左右端点，使用前缀和直接计算每一段区间的和与长度。

时间复杂度：$O(n^2)$

### 100%做法：

考虑数列的前缀和 $sum_i = \sum_{j=1}^{i} a_j$ ，去一段区间求和的过程其实就是从前缀和中选择两个数 $sum_l$ 和 $sum_r$ ，然后考虑他们的差值。

选取两个数，怎么去使得这两个数的差值最小？每次去寻找离当前 $sum_i$ 最近的两个值，去判断两者的差值。

一个很直观的想法是二分，给每个前缀和保存一个pair的信息$< sum_i, i >$，表示前缀和和位置，在sort完以后的序列中去二分得到最近的两个位置。

当然，不使用二分也是可以的。可以发现，对前缀和排完序以后，相邻两个的差就是第一个询问的答案。

对于第二个询问，这里有一个细节需要注意到，如果存在一段 $sum_i$ 相同的情况，需要将这一段 $sum_i$ 作为一个整体考虑到，考虑最左位置的sum和最右位置的sum。我们可以对前缀和数组进行unique，并记录当前 $sum_i$ 最左的位置 L[sum[i]]和最右的位置R[sum[i]]。（如果这里不考虑最左和最右，会有部分情况考虑不到）

每次对比相邻两个的权值差和组合起来的位置差就可以了。

由于权值范围较大，这里的L[]和R[]可以使用map来实现。

```
#include <bits/stdc++.h>
#define mk make_pair
using namespace std;
typedef long long LL;
const int maxn = 1e5 + 10;
const LL inf = 1e18;
int n;
LL a[maxn], sum[maxn];
map<LL, int> L, R;

void add(int i) {
  if(!L.count(sum[i])) L[sum[i]] = i;
  else L[sum[i]] = min(L[sum[i]], i);
  if(!R.count(sum[i])) R[sum[i]] = i;
  else R[sum[i]] = max(R[sum[i]], i);
}

int main() {
  ios::sync_with_stdio(false);
```

```
  cin.tie(NULL);
  cin >> n;
  LL ans = inf;
  int len = 0;
  for(int i = 1; i <= n; ++i) {
    cin >> a[i];
    sum[i] = sum[i - 1] + a[i];
    add(i);
  }
  int m = n;
  sum[++m] = 0;
  add(0);
  sort(sum + 1, sum + 1 + m);
  m = unique(sum + 1, sum + 1 + n) - sum - 1;
  for(int i = 1; i < m; ++i) {
    LL x = sum[i], y = sum[i + 1];
    LL z = abs(x - y);
    int l1 = L[x], r1 = R[x];
    int l2 = L[y], r2 = R[y];
    int l = max(max(abs(l1 - l2), abs(l1 - r2)), max(abs(r1 - l2), abs(r1 - r2)));
    if(z < ans) ans = z, len = l;
    else if(z == ans) len = max(len, l);
  }
  int res=-1;
  for(int i = 1; i <= m; ++i) {
    int l1 = L[sum[i]], r1 = R[sum[i]];
    if(l1 != r1) ans = 0, res = max(res, abs(l1 - r1));
  }
  if(res != -1) len = res;
  cout << ans << '\n' << len << '\n';
  return 0;
}
```

# T2 树的计算

给定树的形成规则，求编号为n的树的结构。

### 30%做法：

枚举节点个数，按照定义将左子树节点个数从小到大枚举，右子树节点个数从大到小枚举，依次得到每个编号的树的样子。

时间复杂度 $O(n)$

### 100%做法：

我们只关注编号为n的树的形状，对于其他形状的树，我们考虑有没有方法直接统计。

这样我们得到一个子问题：对于具有x个节点的二叉树，不同形状的结构有多少种?

这是一个经典问题，假设f[n]表示n个节点的二叉树的个数，那么$f[n] = \sum_{i=0}^{n-1} f[i] * f[n-1-i]$，可以预处理一下f[n]。只需要预处理到f[n]的前缀和覆盖编号大小即可。

接下来，我们考虑如何去快速的确定树的形状：首先确定编号为n的数具有多少个节点，再递归去确认左儿子有多少个节点，右儿子有多少个节点，具体的大小可以通过枚举左儿子和右儿子的个数来确定。对左儿子和右儿子的具体形态，用类似的方法递归处理，直到只有一个节点。

```
#include <bits/stdc++.h>
#define mk make_pair
using namespace std;
typedef long long LL;
LL n;
LL C[30];

void dfs(int Size, LL Ord) {
  // Size: 节点数, Ord: 编号为第几个
  if(!Size) return;
  if(Size == 1) {
    putchar('X');
    return ;
  }
  // 确认id所在的LSize和RSize
  LL tot = Ord;
  int LSize = 0, RSize = Size - 1;
  while(tot > 0) {
    tot -= C[LSize] * C[RSize];
    ++LSize, --RSize;
  }
  --LSize, ++RSize;
  tot += C[LSize] * C[RSize];
  if(LSize) {
    putchar('(');
    dfs(LSize, (tot - 1) / C[RSize] + 1);
    putchar(')');
  }
    putchar('X');
  if(RSize) {
    putchar('(');
    dfs(RSize, (tot - 1) % C[RSize] + 1);
    putchar(')');
  }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  C[0] = C[1] = 1;
  for(int i = 2; i < 21; ++i)
    for(int j = 0; j < i; ++j)
```

```
      C[i] += C[j] * C[i -j - 1];
  cin >> n;
  int Size = 0;
  LL tot = n + 1;
  while(tot > 0) {
    tot -= C[Size];
    ++Size;
  }
  --Size;
  tot += C[Size];
  dfs(Size, tot);
  puts("");
  return 0;
}
```

# T3 我没有说谎

编号为i的人说，有$a_i$个人分数比我高，$b_i$个人分数比我低，求问想要不发生冲突，最多多少人在说谎?

### 20%做法：

$2^n$ 枚举每个人是否有说谎，对结果进行判断。

那么要如何判断呢?

$a_i$个人分数比我高，$b_i$个人分数比我低 <=> 有且仅有 $n-a_i-b_i$ 个人与我分数相同，也即$[b_i+1, n-a_i]$这一段的分数相同。

如果出现两段区间[l1,r1]和[l2,r2]，两个区间出现相交，那么区间[l1,r1]的人的分数要和[l2,r2]的人的分数相同，这与上面有且仅有区间内的人分数相同冲突，这两个人的话不能同时为真。

时间复杂度：$O(n*2^n)$

### 40%做法：

想到了上面的那种判断，就可以考虑将问题进行一个转化：

从n个区间中，求问最多能选取多少个不相交的区间?

可以考虑使用dp求解，令f[i]表示以i作为区间结尾，最多选取多少个不相交的区间。

特别的，对于两个相同的区间[l,r]，他们说的话可以同时为真，但是这里有一个细节需要注意，**区间[l,r]的有效个数只有r-l+1个，多出来的人仍然是在说谎。**

具体做法：枚举区间，对区间[l,r], $f[r]=max(f[r], f[i]+c)$, $i \in [1, l-1]$，其中c为[l,r]出现的次数 。最后的答案枚举所有区间$[l_i, r_i]$，取最大的$f[r_i]$。

时间复杂度 $O(n^2)$

### 100%做法：

考虑优化上面的dp过程，dp数组更改一下定义，令f[i]表示到i为止，最多选取多少个不相交的区间。

在递推的过程i不一定必须得是区间的结尾。

这样可以给上面的过程优化一下: 从1枚举到n，$f[i] = f[i-1]$。当i为区间[l,r]的右端点时，考虑$f[i] = max(f[i], f[l-1] + c)$，其中c为[l,r]出现的次数。

在枚举的时候，使用一个vector来记录当前点作为右端点时，有哪些左端点，优化枚举过程。

时间复杂度$O(n)$

```cpp
#include <bits/stdc++.h>
#define mk make_pair
using namespace std;
const int maxn = 1e5 + 10;
int n;
vector<int> p[maxn];
map<pair<int, int>, int> cnt;
int f[maxn];

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
  for(int i = 1; i <= n; ++i){
    int x, y;
    cin >> x >> y;
    int r = n - x, l = y + 1;
    if(l > r) continue;
    p[r].push_back(l);
    if(!cnt.count(mk(l, r))) cnt[mk(l, r)] = 1;
    else {
      if(cnt[mk(l, r)] < r - l + 1) cnt[mk(l, r)] = cnt[mk(l, r)] + 1;
    }
  }
  for(int i = 1; i <= n; ++i) {
    f[i] = f[i - 1];
    for(auto l: p[i]) f[i] = max(f[i], f[l - 1] + cnt[mk(l, i)]);
  }
  cout << n - f[n] << '\n';
  return 0;
}
```

# T4 美食家

按桶顺序取物品，每次取后桶内物品会变深，问最多可以取多少次?

### 10%做法：

模拟吃的过程，对每一躺枚举一遍桶。

时间复杂度$O(nm)$

### 40%做法：

需要观察到一个性质：当一个桶不能再被吃到以后，这个桶将不会再也不会被选中。

考虑到答案大小不超过5e7，可以从答案的范围入手。

可以考虑使用链表维护一个每一趟能吃到的桶的集合，当这个桶不能再被吃到以后，就从链表中删除，依旧使用模拟的方式来做整个过程。

时间复杂度$O(ans)$

### 100%做法：

对每个桶考虑，可以发现，每个桶能够被吃的次数与其他桶无关，只与自己相关。

我们预处理每个桶能够被吃的次数，考虑如何处理这个过程。

由于一个次数少的桶在被吃完以后，不会对后续次数多的桶产生影响，我们将桶按照可以被吃的次数进行排序，使用一个双指针来进行模拟整个过程。

一个指针表示当前要被使用完的桶，一个指针表示当前的轮数。同时使用树状数组记录桶的使用情况（对于原始下标），可以用就是 1，不能用就是 0，利用树状数组可以计算前缀和。

对当前的桶，不断移动轮数的指针，直到当前的桶的次数将在这一轮中被使用完。对这一轮走不完的情况，我们计算得到他最远的扩展范围，也就是它完全用完的那一时刻的所在位置。这个可以对树状数组进行二分得到。具体实现是利用树状数组的每个数组包含一段区间和的特性。

交替移动这两个指针，直到某个指针达到边界。

时间复杂度 $O(nlog_2n)$

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e5 + 10;

int n, m, x, i, j, k, L, l, aim, now, pos, tmp, ans;
int sum[maxn], w[maxn], t[maxn], id[maxn];

inline bool cmp(const int &a, const int &b) {
    if (t[a] != t[b]) return t[a] < t[b];
    return a < b;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> m >> x;
    ++x;
```

```
    for(L = 1; L * 2 <= n; L *= 2);
    for(i = 1; i <= n; ++i) cin >> w[i];
    for(i = 1; i <= n; ++i) {
    cin >> k;
        id[i] = i;
        if(w[i] < x) t[i] = x - w[i], t[i] = t[i] / k + (t[i] % k != 0); //预处理次数
        if(t[i]) for(j = i; j <= n; j += j & -j) ++sum[j];
    }
    sort(id + 1, id + n + 1, cmp);
    now = 1; //轮数
    pos = 0; //上一个桶完全结束的位置
    ans = 0; //当前累计的次数
    for(i = 1; i <= n; ++i) if (t[id[i]]) break; //寻找最初能用的桶
    for(; i <= n; ++i) {
        while(now <= m) { //轮数
            tmp = n - i + 1; //当前这一轮还剩的桶的个数（能够产生的次数）
            for(j = pos; j; j -= j & -j) tmp -= sum[j]; //对于pos!=0的情况，这一轮在pos之前的数据，已经在上一个桶中计算过贡献了
            if(ans + tmp < t[id[i]]) ans += tmp, ++now, pos = 0; //当前桶在这一轮还能存在
            else break; //当前桶这一轮不存在了
        }
        if(now > m) break;
        aim = t[id[i]] - ans; //不足一轮，但是多出来的次数
        for(j = pos; j; j -= j & -j) aim += sum[j]; //考虑同一轮中，有多个桶失效的情况
        for(pos = tmp = 0, l = L; l; l /= 2) { //在树桩数组中二分
            pos += l;
            if(pos >= n || tmp + sum[pos] >= aim) pos -= l;
            else tmp += sum[pos];
        }
        ++pos;
        ans = t[id[i]];
        for(; t[id[i]] == t[id[i + 1]]; ++i) //对于相同次数的桶，在当前桶失效后一定会失效，排除失效的桶
            for(j = id[i]; j <= n; j += j & -j)
                --sum[j];
        for(j = id[i]; j <= n; j += j & -j) --sum[j];
    }
  cout << ans << '\n';
}
```