2023 SCCC 봄 #5

BOJ 27931. Parity Constraint Closest Pair (Easy)

어떤 두 점의 좌표를 2로 나눈 나머지가 동일하면 두 점 사이의 거리는 짝수이고, 2로 나눈 나머지가 다르면 두 점 사이의 거리는 홀수입니다. 주어진 점들을 좌표가 짝수인 점과 홀수인 점으로 나눠서 생각해 봅시다.

좌표가 짝수인 점과 홀수인 점을 모아서 각각 정렬합시다. 두 점의 거리가 짝수인 최소 거리는 두 배열에서 각각 인접한 좌표들 간의 차이의 최솟값을 계산하면 알 수 있습니다.

홀수 거리는 짝수 점 하나와 홀수 점 하나를 선택해서 만들어야 합니다. 최소 거리를 구하는 것이 목표이기 때문에 각 짝수 점마다 자신보다 앞에 있으면서 가장 가까운 점, 자신보다 뒤에 있으면서 가장 가까운 점을 구해서 거리를 재면 됩니다.

점들을 정렬하는데 $O(N \log N)$ 시간이 걸리고, 짝수 거리 최솟값을 구하는데 $O(N)$, 홀수 거리 최솟값을 구하는 것은 이분 탐색을 사용하면 $O(N \log N)$, 투 포인터를 사용하면 $O(N)$ 만큼의 시간이 걸립니다. 전체 시간 복잡도는 $O(N \log N)$입니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll INF = 0x3f3f3f3f3f3f3f3f;

int N;
vector<ll> V[2];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=0,t; i<N; i++) cin >> t, V[t&1].push_back(t);
    for(int i=0; i<2; i++) sort(V[i].begin(), V[i].end());
    ll X = INF, Y = INF;
    for(int i=0; i<2; i++) for(int j=1; j<V[i].size(); j++) X = min(X, V[i][j] -
V[i][j-1]);
    for(int iter=0; iter<2; iter++, swap(V[0], V[1])){
        for(int i=0, j=0; i<V[0].size(); i++){
            while(j < V[1].size() && V[1][j] < V[0][i]) j++;
            for(int k=j-5; k<=j+5; k++) if(0 <= k && k < V[1].size()) Y = min(Y,
abs(V[0][i] - V[1][k]));
        }
    }
    if(X < INF) assert(X % 2 == 0);
    if(Y < INF) assert(Y % 2 == 1);
    cout << (X < INF ? X : -1) << " ";
    cout << (Y < INF ? Y : -1) << "\n";
}
```

BOJ 1807. 척 노리스

만약 N이 4의 배수라면 $A_N = N$이고, 4의 배수가 아니라면 A_N은 $10N, 10N+1, 10N+2, 10N+3$ 중 하나입니다. 따라서 A_1부터 A_N까지 이어붙인 것의 길이 $f(N)$은 $N+$ (1부터 N까지 이어붙인 것의 길이) $-\lfloor N/4 \rfloor$와 동일합니다. $f(N)$은 $O(\log_1 0N)$ 정도 시간에 계산할 수 있습니다.

$K \le 10^{15}$로 제한이 매우 크기 때문에 K번째 수를 한 단계씩 직접 찾아나가는 방식으로는 풀 수 없습니다. 더 효율적인 방법을 생각해야 합니다.

어떤 수 x가 주어졌을 때 $f(x)$를 빠르게 구할 수 있기 때문에 이분 탐색을 시도해 볼 수 있습니다. 구체적으로, K번째 글자가 속하는 A_t를 이분 탐색을 이용해 구한 다음, A_t의 $K - f(t-1)$번째 숫자를 구해서 출력하는 방식으로 해결할 수 있습니다.

이분 탐색을 이용해 $O(\log^2 K)$ 시간에 t를 찾을 수 있고, A_t의 $K - f(t-1)$번째 숫자를 구하는 것은 $O(\log t) \le O(\log K)$에 할 수 있습니다. 따라서 전체 시간 복잡도는 $O(\log^2 K)$입니다.

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

ll Sum(ll n){
    ll res = 0, s = 1, e = 10, len = 1;
    while(s <= n){
        res += (min(e-1, n) - s + 1) * len;
        s *= 10; e *= 10; len++;
    }
    return res;
}

ll Len(ll n){
    if(n == 0) return 0;
    return Sum(n) + n - n / 4;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    ll N;
    while(cin >> N && N){
        ll l = 1, r = N;
        while(l < r){
            ll m = (l + r) / 2;
            if(Len(m) >= N) r = m;
            else l = m + 1;
        }
        N -= Len(r-1);
        string s = to_string(r);
        if(r % 4 != 0) for(int i=0; i<10; i++) if((r * 10 + i) % 4 == 0) { s += char(i+'0'); break; }
        cout << s[N-1] << "\n";
    }
}
```

BOJ 11877. 홍수

높이가 $N, N-1$인 막대를 양쪽 끝에 배치해서 그릇 모양을 만들면 최대 $(N-1)(N-2)/2$ 만큼의 물을 저장할 수 있습니다. 따라서 $X > (N-1)(N-2)/2$이면 -1을 출력합시다.

$X < (N-1)(N-2)/2$이면 그릇의 부피를 줄여야 합니다. 1부터 $N-2$까지의 수를 적당히 사용해서 1부터 $(N-1)(N-2)/2$까지의 모든 수를 만들 수 있기 때문에 합이 X가 되도록 막대를 적당히 제거하면 됩니다. 제거한 기둥은 바깥쪽에 그릇이 만들어지지 않도록 정렬해서 배치하면 문제를 해결할 수 있습니다.

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

ll N, X, K;
bool Use[1010101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> X;
    K = (N - 1) * (N - 2) / 2;
    if(X > K){ cout << -1; return 0; }
    K -= X;
    memset(Use, true, sizeof Use);
    for(int i=1; i<=N-2; i++){
        if(K >= N-i-1) K -= N-i-1, Use[i] = false;
    }
    cout << N << " ";
    for(int i=1; i<=N-2; i++) if(Use[i]) cout << i << " ";
    cout << N - 1 << " ";
    for(int i=N-2; i>=1; i--) if(!Use[i]) cout << i << " ";
}
```

BOJ 5550. 헌책방

동일한 장르의 책을 매입할 때는 가격이 비싼 것부터 매입하는 것이 항상 이득입니다.

$C(i,j) :=$ i번째 장르의 책을 j권 매입할 때의 최대 가격이라고 정의합시다. 각 장르마다 책을 가격의 내림차순으로 정렬하면 $O(N \log N)$ 시간에 계산할 수 있습니다.

여기까지 왔다면 전형적인 동적 계획법 문제로 생각할 수 있습니다.

$D(t,n) := 1 \cdots t$번째 장르의 책을 총 N권 매입할 때의 최대 가격이라고 정의합시다. 상태 전이는 $D(t,n) = \max\{D(t-1,i) + C(t,n-i)\}$와 같이 나타낼 수 있고, $O(TN^2)$ 시간에 계산할 수 있습니다.

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

int N, K, A[11][2020], S[11], C[11][2020], D[11][2020];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++){
        int a, b; cin >> a >> b;
        A[b][++S[b]] = a;
```

```
    }
    for(int t=1; t<=10; t++){
        int sz = S[t];
        sort(A[t]+1, A[t]+sz+1, greater<>());
        for(int i=1, s=0; i<=sz; i++){
            s += A[t][i];
            C[t][i] = s + i * (i - 1);
        }
    }

    for(int t=1; t<=10; t++){
        for(int i=1; i<=N; i++){
            for(int j=0; j<=i; j++){
                D[t][i] = max(D[t][i], D[t-1][j] + C[t][i-j]);
            }
        }
    }
    cout << D[10][K];
}
```

BOJ 27935. 고연전/연고전 기차놀이

동적 계획법으로 풀 수 있다는 것은 쉽게 알아차릴 수 있습니다. 일단 시간 복잡도를 신경쓰지 말고 점화식을 세워봅시다.

$D(i) := 1 \cdots i$번째 사람까지 고려했을 때의 최소 기차 수라고 정의하면, 구간 $[j, i]$가 조건을 만족하도록 하는 j에 대해 $D(i) = \min\{D(j-1)+1\}$입니다. 하지만 이 점화식을 바로 계산하면 $O(N^2)$이기 때문에 더 빠른 방법을 찾아야 합니다.

K는 $+1$, Y는 -1로 취급해서 누적 합 배열 A를 만듭시다. 구간 $[j, i]$가 조건을 만족한다는 것은 $i - j + 1 \leq L$ and $|A_i - A_{j-1}| \leq 1$을 만족한다는 것과 동치입니다. 누적 합 배열의 값이 같은 인덱스끼리 모두 모으면, $D(i)$를 계산하는 것은 누적 합이 $A_i - 1, A_i, A_i + 1$인 그룹에서 $[i - L, i)$ 구간에 RMQ를 하는 것과 동일하게 생각할 수 있습니다.

쿼리를 하는 구간의 시작점과 끝점이 모두 단조증가하는 상황에서는 `std::deque`나 `std::list` 등을 사용한 슬라이딩 윈도우 기법으로 amortized $O(1)$ 시간에 구간 최솟값을 계산할 수 있습니다. 각 인덱스에서 3번의 RMQ를 수행하므로 RMQ는 총 $3N$번 수행하게 됩니다. 따라서 전체 시간 복잡도는 $O(N)$입니다.

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

int N, K, A[101010], D[101010];
vector<int> V[202020];
list<pair<int,int>> Q[202020];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K; A[0] = N + 10;
    for(int i=1; i<=N; i++){ char c; cin >> c; A[i] = c == 'K' ? 1 : -1; }
    partial_sum(A, A+N+1, A);
    for(int i=0; i<=N; i++) V[A[i]].push_back(i);
    memset(D, 0x3f, sizeof D); D[0] = 0; Q[A[0]].emplace_back(0, 0);
    for(int i=1; i<=N; i++){
        for(int j=A[i]-1; j<=A[i]+1; j++){
            while(!Q[j].empty() && Q[j].front().first < i - K) Q[j].pop_front();
            if(!Q[j].empty()) D[i] = min(D[i], Q[j].front().second + 1);
```

```
        }
        while(!Q[A[i]].empty() && Q[A[i]].back().second > D[i])
Q[A[i]].pop_back();
        Q[A[i]].emplace_back(i, D[i]);
    }
    cout << D[N];
}
```

BOJ 27933. 대회 이름 정하기

전체 구간에서의 점수를 구하는 방법을 먼저 생각해 봅시다.

$D(i, j, k) := 1 \cdots i$번째 카드만 고려했을 때 처음에 j, 마지막에 k를 선택하는 최대 횟수라고 정의하면 $O(N)$ 시간에 점화식을 계산할 수 있습니다.

구체적으로, $A_i = 0$인 경우와 $A_i = 1$인 경우로 나눠서 다음과 같이 계산할 수 있습니다.

- $A_i = 0$인 경우
  - $D(i, 0, 0) = \max\{D(i-1, 0, 0), D(i-1, 0, 1), 1\}$
  - $D(i, 0, 1) = D(i-1, 0, 1)$
  - $D(i, 1, 0) = \max\{D(i-1, 1, 0), D(i-1, 1, 1)\}$
  - $D(i, 1, 1) = D(i-1, 1, 1)$
- $A_i = 1$인 경우
  - $D(i, 0, 0) = D(i-1, 0, 0)$
  - $D(i, 0, 1) = \max\{D(i-1, 0, 1), D(i-1, 0, 0)\}$
  - $D(i, 1, 0) = D(i-1, 1, 0)$
  - $D(i, 1, 1) = \max\{D(i-1, 1, 1), D(i-1, 1, 0), 1\}$

첫 번째 값은 i번째 카드를 선택하지 않는 경우, 두 번째 값은 i번째 카드를 뒤에 붙이는 경우, 세 번째 값은 i번째 값을 처음으로 선택하는 경우에 해당합니다.

$D(i-1, *, *)$에서 $D(i, *, *)$로 전이하는 연산을 $f_{A_i}(D_i)$라고 합시다. 전체 구간에서의 점수를 구하는 것은 $D(0)$에 f_{A_1} 부터 f_{A_N} 까지를 순서대로 적용하는 것과 같습니다. 따라서 구간 $[s, e]$에서의 점수를 구하는 것은 $D(0)$에 f_{A_s} 부터 f_{A_e} 까지를 순서대로 적용하는 것과 같습니다.

이 연산은 결합 법칙이 성립하기 때문에 세그먼트 트리를 이용하면 구간 쿼리를 $O(\log N)$ 시간에 처리할 수 있습니다. 세그먼트 트리의 i번째 리프 정점에 f_{A_i} 를 저장한 다음, 두 정점을 합치는 연산을 잘 정의하면 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 19;

struct Node{
    ll a[2][2];
    Node(){ memset(a, 0xc0, sizeof a); }
    Node(char c, ll v){
        memset(a, 0xc0, sizeof a);
        if(c == 'Y') a[0][0] = v;
        else a[1][1] = v;
    }
    friend Node operator + (const Node &l, const Node &r){
        Node res;
        for(int i=0; i<2; i++){
            for(int j=0; j<2; j++){
```

```
                res.a[i][j] = max({l.a[i][j], r.a[i][j], l.a[i][0] + r.a[1][j],
l.a[i][1] + r.a[0][j]});
            }
        }
        return res;
    }
} T[SZ<<1];

int N, Q;
string S;

Node Query(int l, int r){
    Node lv, rv;
    for(l|=SZ, r|=SZ; l<=r; l>>=1, r>>=1){
        if(l & 1) lv = lv + T[l++];
        if(~r & 1) rv = T[r--] + rv;
    }
    return lv + rv;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> S;
    for(int i=1,t; i<=N; i++) cin >> t, T[i|SZ] = Node(S[i-1], t);
    for(int i=SZ-1; i; i--) T[i] = T[i<<1] + T[i<<1|1];
    cin >> Q;
    for(int i=1; i<=Q; i++){
        int l, r; cin >> l >> r;
        auto res = Query(l, r);
        ll Y = max(0LL, res.a[0][1]), K = max(0LL, res.a[1][0]);
        if(Y > K) cout << "Y " << Y << "\n";
        else if(Y < K) cout << "K " << K << "\n";
        else cout << "YK " << Y << "\n";
    }
}
```

BOJ 25054. Drone Photo

$O(N^3)$ 풀이는 간단합니다. 행 2개 i, j를 고정하면 $A(i,k) < A(j,k)$를 만족하는 k의 개수를 구해서 $\binom{x}{2}$를 구하면 $O(N^3)$에 해결할 수 있습니다. 그렇다고 자료구조 등을 이용해 이 풀이를 최적화하려고 하면 문제를 풀 수 없습니다. 다른 관점으로 바라보아야 합니다.

$a < b < c < d$인 네 수를 사각형의 꼭짓점으로 잡는다고 합시다. 네 꼭짓점을 배열하는 방법은 24가지이고, a을 왼쪽 상단 꼭짓점으로 고정하면 6가지입니다. 6가지 경우를 모두 그림으로 그려봅시다.

```
a   b  |  a   b  |  a   c  |  a   c  |  a   d  |  a   d
       |         |         |         |         |
c   d  |  d   c  |  b   d  |  d   b  |  b   c  |  c   b
```

여기에서 4번째와 6번째는 불가능한 배치입니다. 가능한 배치의 경우, b와 c는 인접한 두 수 중 하나보다 작고, 다른 하나보다 큽니다. 반대로 불가능한 배치에서는 b는 인접한 두 수 모두보다 작고, c는 두 수 모두보다 큽니다. 작은 수에서 큰 수로 가는 화살표를 그려보면 더 명확하게 알 수 있습니다.

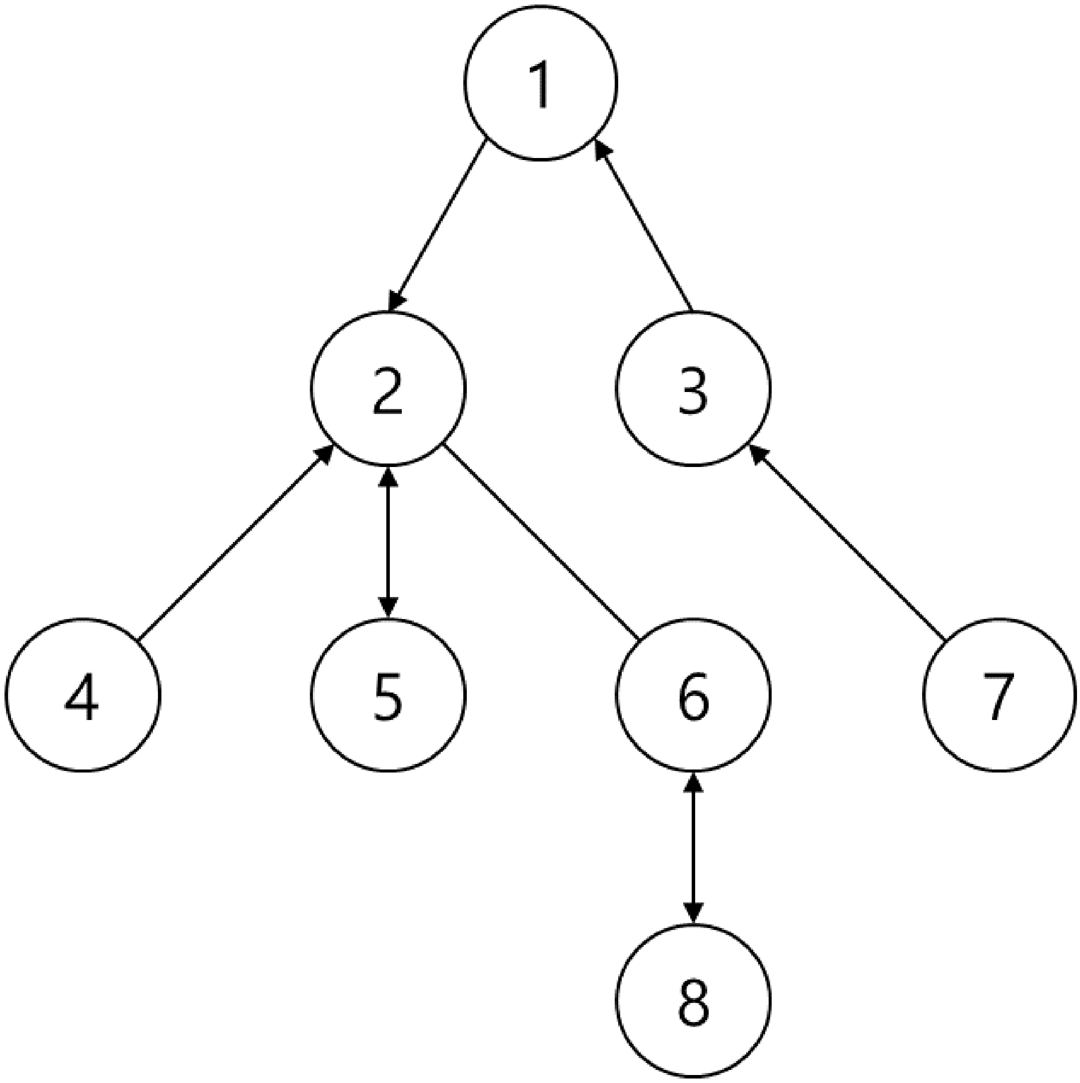
```
a → b  |  a → b  |  a → c  |  a → c  |  a → d  |  a → d
↓   ↓  |  ↓   ↓  |  ↓   ↓  |  ↓   ↑  |  ↓   ↑  |  ↓   ↑
c → d  |  d ← c  |  b → d  |  d ← b  |  b → c  |  c ← b
```

가능한 배치의 경우 각 직사각형은 in-degree와 out-degree가 모두 1인 점을 2개씩 갖고 있으며, 불가능한 배치의 경우 그러한 점이 없습니다. 따라서 더블 카운팅을 이용해 직사각형의 개수를 세는 대신, in-degree = out-degree = 1인 점의 개수를 센 다음 2로 나눠도 정답을 구할 수 있습니다.

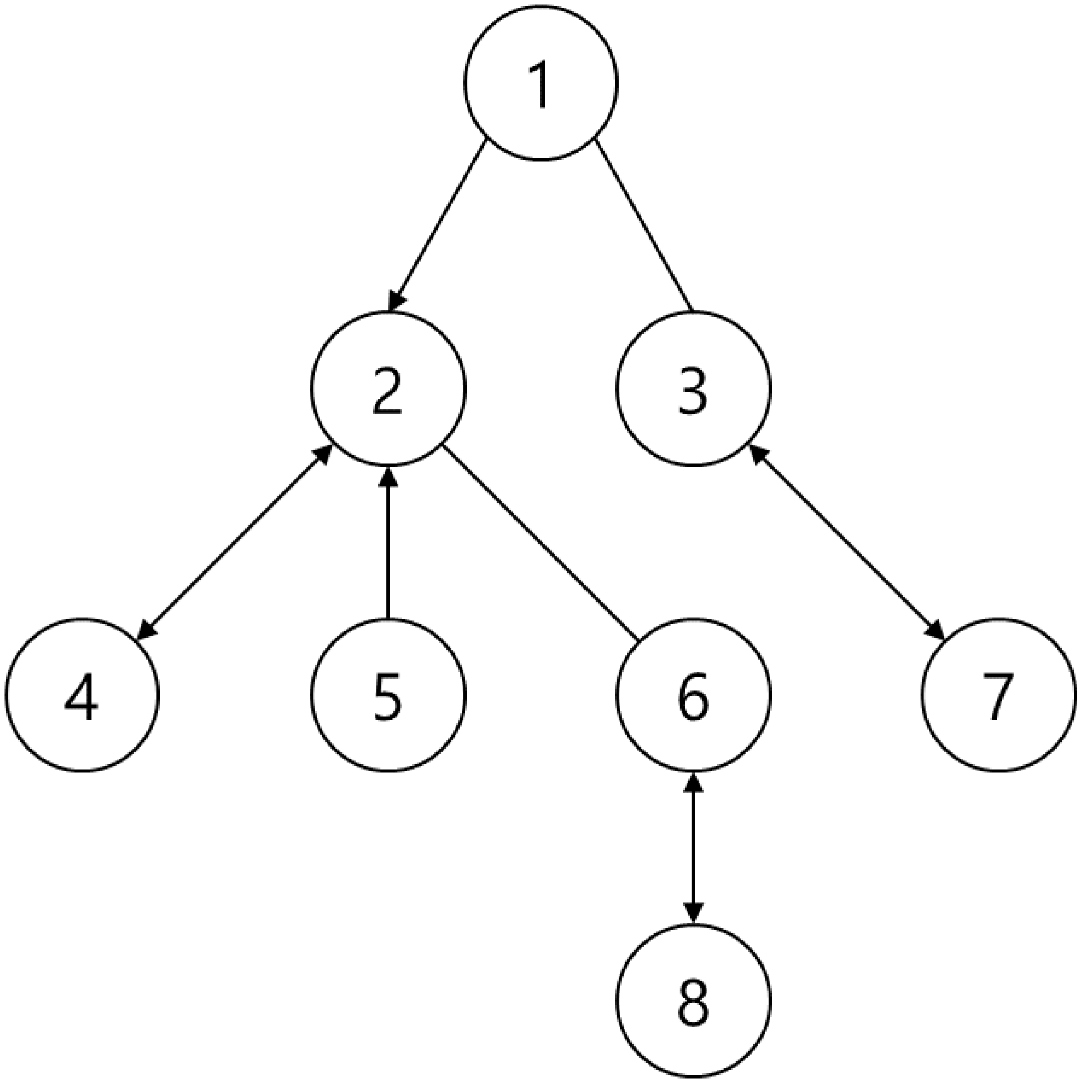
```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll N, I[2323232], J[2323232], R[1515], C[1515], A[2323232], B[2323232], X;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) for(int j=1,t; j<=N; j++) cin >> t, I[t] = i, J[t] =
j;
    for(int i=1; i<=N*N; i++) A[i] = R[I[i]]++, B[i] = C[J[i]]++;
    for(int i=1; i<=N*N; i++) X += A[i] * (N - B[i] - 1) + B[i] * (N - A[i] -
1);
    cout << X / 2;
}
```

BOJ 21825. Through Another Maze Darkly

다음과 같은 트리를 생각해 봅시다. 각 정점에서 가장 먼저 사용하는 간선을 화살표로 표현했습니다.



첫 번째 순회에서는 `1 2 5 2 6 8 6 2 1 3 1` 순서대로 방문합니다. 첫 번째 순회를 한 이후의 트리는 다음과 같습니다. 첫 번째 순회에서 방문한 정점들의 화살표가 모두 첫 번째 자식(없다면 부모) 정점을 가리키고 있다는 점을 관찰할 수 있습니다.



두 번째 순회에서는 `1 2 4 2 5 2 6 8 6 2 1 3 7 3 1` 순서대로 방문합니다. $t-1$번째 순회에서 방문한 정점은 항상 t번째 순회에서 방문한다는 점도 알 수 있습니다. 또한, 각 순회에서 정점을 방문하는 순서가 역전되지 않습니다.

따라서 자식 정점들을 적당한 순서대로 배치한 다음 오일러 투어를 구해 놓으면, 두 정수 t, k가 주어졌을 때 t번째 순회에서 k번째로 방문하는 정점을 세그먼트 트리 등을 이용해 오프라인으로 구할 수 있습니다.

입력으로 트리의 인접 리스트가 주어지면 부모 정점의 위치 p를 구합시다. p 이전에 있는 정점은 현재 정점을 처음으로 방문한 순회에서, p 이후에 있는 정점은 그 다음 순회에서 방문하게 됩니다. 따라서 DFS를 이용해 $O(N)$ 시간에 각 정점을 처음으로 방문하는 시점을 구할 수 있습니다. 각 간선을 처음 사용하게 되는 시점도 함께 구할 수 있습니다.

이후 구현의 편의를 위해 부모 정점이 가장 마지막에 오도록 인접 리스트를 cyclic shift 합시다.

이제 K번째로 방문하는 정점을 구하는 쿼리를 처리할 차례입니다. 우리의 궁극적인 목표는 K를 적당한 정수 t, k로 바꿔서, t번째 순회에서 k번째로 방문하는 정점을 구하는 문제로 변환하는 것입니다.

각 간선이 처음 사용되는 시점을 알고 있다면, 누적 합과 비슷한 방식으로 각 순회의 길이를 상수 시간에 구할 수 있습니다. 따라서 순회의 길이의 누적합 위에서 이분 탐색을 하면 $O(\log N)$ 시간에 t를 구할 수 있습니다. $N+1$번째 순회부터는 항상 모든 정점과 간선을 방문한다는 것에 유의하세요.

t를 구했다면 k를 구하는 것은 간단합니다. 단순히 모듈러 연산을 이용하면 되기 때문입니다.

t, k가 주어졌을 때 실제 정답을 찾는 것은 쿼리를 t 오름차순으로 정렬한 다음, 세그먼트 트리에서 k번째 원소를 찾는 연산을 사용하면 됩니다. 자세한 구현은 코드의 64번째 줄부터 참고하시면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 21;

int N, Q, On[808080], R[808080], T[SZ];
vector<int> G[808080];
vector<pair<int,int>> Tour;
vector<int> Idx[808080]; // Idx[t] : t에 켜지는 정점
ll TourSize[808080];

void Add(int x, int v){ for(x+=3; x<SZ; x+=x&-x) T[x] += v; }
int Get(int x){ int ret = 0; for(x+=3; x; x-=x&-x) ret += T[x]; return ret; }
int Kth(int x){
    int l = 0, r = Tour.size() - 1;
    while(l < r){
        int m = (l + r) / 2;
        if(Get(m) >= x) r = m;
        else l = m + 1;
    }
    return r;
}

void GetTime(int v, int b=-1, int t=1){
    On[v] = t;
    int pos = find(G[v].begin(), G[v].end(), b) - G[v].begin();
    for(int i=0; i<pos; i++) GetTime(G[v][i], v, t);
    for(int i=pos+1; i<G[v].size(); i++) GetTime(G[v][i], v, t+1);
    if(b != -1) rotate(G[v].begin(), G[v].begin()+pos, G[v].end());
}

inline void AddEdge(int s, int e){
    Idx[max(On[s], On[e])].push_back(Tour.size());
    Tour.emplace_back(s, e);
}

void GetTour(int v, int b=-1){
    for(auto i : G[v]) if(i != b) AddEdge(v, i), GetTour(i, v), AddEdge(i, v);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q;
    for(int i=1; i<=N; i++){
        int sz; cin >> sz; G[i].resize(sz);
        for(auto &j : G[i]) cin >> j;
        rotate(G[i].begin(), G[i].begin()+1, G[i].end());
    }
    GetTime(1);
    GetTour(1);
    for(int i=1; i<=N; i++) TourSize[i] = TourSize[i-1] + Idx[i].size();
    for(int i=1; i<=N; i++) TourSize[i] += TourSize[i-1];

    vector<tuple<ll,ll,ll>> V; // iteration, kth, idx
```

```
    for(int q=1; q<=Q; q++){
        ll K; cin >> K;
        int it = lower_bound(TourSize+1, TourSize+N+1, K) - TourSize;
        K -= TourSize[it-1];
        if(it == N+1) K = (K - 1) % Tour.size() + 1;
        V.emplace_back(it, K, q);
    }
    sort(V.begin(), V.end());

    for(int it=1, i=0; it<=N+1; it++){
        for(auto e : Idx[it]) Add(e, 1);
        while(i < V.size() && get<0>(V[i]) == it){
            R[get<2>(V[i])] = Kth(get<1>(V[i]));
            i += 1;
        }
    }
    for(int i=1; i<=Q; i++) cout << Tour[R[i]].second << " ";
}
```

BOJ 20197. 3D Histogram

2차원의 경우는 매우 유명한 문제이고, 모노톤 스택을 이용한 풀이와 분할 정복을 이용한 풀이가 잘 알려져 있습니다. 모노톤 스택 풀이를 3차원으로 확장하는 것은 어려울 것 같으니 분할 정복 풀이를 생각해봅시다.

구간 $[st, ed]$에 대한 문제를 해결할 때 구간의 중점 md를 지나는 경우를 모두 처리한 뒤, 구간 $[st, md-1]$과 $[md+1, ed]$에 대한 문제를 재귀적으로 해결할 것입니다.

두 함수 $f(i,j)$와 $g(i,j)$를 아래와 같이 정의합시다. $(e-s+1) \times f(s,e) \times g(s,e)$를 최대화 하는 것이 이 문제의 목표입니다.

$f(i,j) := \min_{i \le k \le j} A_k$

$g(i,j) := \min_{i \le k \le j} B_k$

구간의 양 끝점 s, e와 중간 지점 m에 대해, 문제의 상황을 4가지로 분류할 수 있습니다.

1. $f(s,m) \le f(m,e) \text{ and } g(s,m) \le g(m,e)$
2. $f(s,m) \le f(m,e) \text{ and } g(s,m) \ge g(m,e)$
3. $f(s,m) \ge f(m,e) \text{ and } g(s,m) \le g(m,e)$
4. $f(s,m) \ge f(m,e) \text{ and } g(s,m) \ge g(m,e)$

1, 2만 처리하면 적절히 뒤집고 swap하는 것으로 3, 4번을 똑같이 처리할 수 있습니다. 1, 2에 대한 풀이를 알아봅시다.

1번 케이스는 투포인터로 $O(ed-st+1)$ 만에 쉽게 해결할 수 있습니다.

각 시작점 s에 대해, $f(s,e) = f(s,m)$이고 $g(s,e) = g(s,m)$인 e의 최댓값을 찾아서 최대 부피를 갱신하면 됩니다. $f(i,j)$와 $g(i,j)$ 모두 j가 증가할 때 값이 단조 감소하기 때문에 투 포인터로 항상 e를 찾을 수 있습니다. 아래 코드에서 `Calc1` 함수를 참고하세요.

2번 케이스는 조금 복잡한데, $(e-s+1) \times f(s,m) \times g(m,e)$를 최대화해야 합니다.
$(e-s+1)$를 $(1-s)$와 e로 쪼갠 다음 식을 $f(s,m)$으로 묶으면
$f(s,m) \times \{g(m,e) \times (-s+1) + e \times g(m,e)\}$가 됩니다.

중괄호 안쪽의 식은 $g(m,e)$를 기울기, $e \times g(m,e)$를 y절편으로 하는 일차함수라고 생각할 수 있습니다. 여러 개의 일차함수가 있을 때 특정 x 좌표에서의 최댓값을 구하는 것은 Convex Hull Trick을 이용해 빠르게 구할 수 있으니, 이 성질을 이용해서 풀이를 찾아봅시다.

먼저 1번 케이스처럼 각 시작점 s에 대해, $f(s,m) \le f(m,e)$이고 $g(s,m) \ge g(m,e)$이 되도록 하는 e의 **구간**을 각각 전처리합시다. 투 포인터를 이용해 $O(ed-st+1)$ 시간에 구할 수 있습니다. 이 구간의 양 끝점을 각각 $l(s), r(s)$라고 합시다.

각 시작점 s에 대해, $l(s)$부터 $r(s)$번째 직선들 중에서 $-l+1$ 시점에서의 최댓값을 구해야 합니다. 이는 세그먼트 트리의 각 정점에서 CHT를 관리하는 것으로 처리할 수 있습니다. 세그먼트 트리의 각 정점에 저장되는 직선의 기울기와 쿼리로 주어지는 x 좌표인 $-l+1$ 모두 단조성이 있으니 CHT를 선형에 처리할 수 있습니다.

$T(N) = 2T(N/2) + O(N \log N)$이므로 $O(N \log^2 N)$에 문제를 풀 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 18;

struct line_t{
    ll a, b;
    line_t() : line_t(0, 0) {}
    line_t(ll a, ll b) : a(a), b(b) {}
    ll f(ll x) const { return a * x + b; }
};

ll CCW(const line_t &l1, const line_t &l2, const line_t &l3){
    return (l2.a - l1.a) * (l3.b - l1.b) - (l3.a - l1.a) * (l2.b - l1.b);
}

vector<line_t> T[SZ<<1];
line_t L[SZ]; int P[SZ<<1];

void Build(int node, int s, int e){
    T[node].clear();
    if(s == e){ T[node].push_back(L[s]); P[node] = 0; return; }
    int m = (s + e) / 2;
    Build(node<<1, s, m); Build(node<<1|1, m+1, e);
    auto &hull = T[node]; hull = T[node<<1];
    for(auto line : T[node<<1|1]){
        while(hull.size() >= 2 && CCW(hull[hull.size()-2], hull.back(), line) <=
0) hull.pop_back();
        hull.push_back(line);
    }
    P[node] = (int)hull.size() - 1;
}

ll Get(int node, ll x){
    while(P[node] > 0 && T[node][P[node]].f(x) <= T[node][P[node]-1].f(x))
P[node]--;
    return T[node][P[node]].f(x);
}

ll Query(int node, int s, int e, int l, int r, ll x){
    if(r < s || e < l) return 0;
    if(l <= s && e <= r) return Get(node, x);
    int m = (s + e) / 2;
    return max(Query(node<<1, s, m, l, r, x), Query(node<<1|1, m+1, e, l, r,
x));
}
```

```
ll N, A[SZ], B[SZ], AM[SZ], BM[SZ], R;
void Swap(){ for(int i=1; i<=N; i++) swap(A[i], B[i]); }
void Reverse(){ reverse(A+1, A+N+1); reverse(B+1, B+N+1); }

void Calc1(const int st, const int m, const int ed){
    for(int s=m, e=m; s>=st; s--){
        while(e + 1 <= ed && AM[s] <= A[e+1] && BM[s] <= B[e+1]) e++;
        R = max(R, AM[s] * BM[s] * (e-s+1));
    }
}

void Calc2(const int st, const int m, const int ed){
    static pair<int,int> Range[SZ];
    for(int s=m, l=m, r=m; s>=st; s--){
        while(l <= ed && BM[s] < B[l]) l++;
        while(r + 1 <= ed && AM[s] <= A[r+1]) r++;
        Range[s] = {l, r};
    }
    for(int e=m; e<=ed; e++) L[e] = line_t(BM[e], e * BM[e]);
    Build(1, m, ed);
    for(int s=m; s>=st; s--) R = max(R, AM[s] * Query(1, m, ed, Range[s].first,
Range[s].second, -s+1));
}

void Solve(int st, int ed){
    if(st > ed) return;
    if(st == ed){ R = max(R, A[st] * B[st]); return; }
    int m = (st + ed) / 2;
    AM[m] = A[m]; BM[m] = B[m];
    for(int i=m-1; i>=st; i--) AM[i] = min(AM[i+1], A[i]), BM[i] = min(BM[i+1],
B[i]);
    for(int i=m+1; i<=ed; i++) AM[i] = min(AM[i-1], A[i]), BM[i] = min(BM[i-1],
B[i]);
    Calc1(st, m, ed); Calc2(st, m, ed);
    Solve(st, m-1); Solve(m+1, ed);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i] >> B[i];
    Solve(1, N); Swap(); Solve(1, N);
    Reverse();
    Solve(1, N); Swap(); Solve(1, N);
    cout << R;
}
```