2023 SCCC 봄 #6

BOJ 27940. 가지 산사태

> N층으로 구성된 농장에 비가 M번 내린다. i번째 비가 오면 농장의 1층부터 t_i층이 각각 r_i 만큼의 비를 받게 된다.
> 한 층에 K 초과 만큼의 빗물이 누적되면 그 층은 무너진다. 처음으로 농장이 무너지는 시점과, 그 때 무너지는 층의 번호를 출력하라.

매번 1층에는 비가 내리기 때문에 농장이 처음 무너지는 시점에 1층은 항상 무너집니다. 따라서 1층이 무너지는 시간을 구하면 문제를 해결할 수 있습니다.

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

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    ll N, Q, K, S=0; cin >> N >> Q >> K;
    for(int i=1; i<=Q; i++){
        ll p, v; cin >> p >> v;
        if((S += v) > K){ cout << i << " " << 1; return 0; }
    }
    cout << -1;
}
```

BOJ 27942. :danceplant:

> 각 칸에 정수가 적혀있는 $N \times N$ 크기의 격자의 정중앙에 2×2 크기의 가지가 있다. (N은 짝수)
>
> 가지는 매 순간 상하좌우 중 한 방향으로 자신의 길이를 1 만큼 늘리고, 늘어난 공간에 위치한 수들의 합만큼의 양분을 먹는다. 가지는 매 순간 양분을 가장 많이 먹는 방향으로 움직이고, 더 이상 크기를 늘릴 수 없거나 먹을 수 있는 양분이 0 이하가 되면 확장을 멈춘다. 가지가 크기를 확장하는 순서를 구하라.

2차원 격자에서 임의의 직사각형 영역의 합을 빠르게 구할 수 있다면 단순 시뮬레이션으로 문제를 해결할 수 있습니다.

2차원 누적 합 배열 $S(i,j) = \sum_{r=1}^{i}\sum_{c=1}^{j} A(r,c)$를 정의합시다.
$S(i,j) = S(i-1,j) + S(i,j-1) - S(i-1,j-1) + A(i,j)$가 성립함을 알 수 있고, 따라서 2차원 누적 합 배열 S는 $O(N^2)$ 시간에 전처리할 수 있습니다. 직사각형 영역 $[r_1, r_2] \times [c_1, c_2]$의 합은 $S(r_2, c_2) - S(r_1 - 1, c_2) - S(r_2, c_1 - 1) + S(r_1 - 1, c_1 - 1)$을 이용해 $O(1)$ 시간에 계산할 수 있습니다.
직접 종이에 2차원 격자를 그린 다음, 식의 각 항이 나타내는 칸을 색칠해 보면 이해하기 쉬울 것입니다.

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

int N, A[3030][3030]; string S="UDLR";
```

```
int Get(int r1, int r2, int c1, int c2){
    if(r1 < 1 || c1 < 1 || r2 > N || c2 > N) return 0;
    return A[r2][c2] - A[r1-1][c2] - A[r2][c1-1] + A[r1-1][c1-1];
}

vector<tuple<int,int,int,int,int,int>> Next(tuple<int,int,int,int> a){
    auto [r1,r2,c1,c2] = a;
    vector<tuple<int,int,int,int,int,int>> res;
    res.emplace_back(Get(r1-1, r1-1, c1, c2), -0, r1-1, r2, c1, c2);
    res.emplace_back(Get(r2+1, r2+1, c1, c2), -1, r1, r2+1, c1, c2);
    res.emplace_back(Get(r1, r2, c1-1, c1-1), -2, r1, r2, c1-1, c2);
    res.emplace_back(Get(r1, r2, c2+1, c2+1), -3, r1, r2, c1, c2+1);
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) cin >> A[i][j];
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) A[i][j] += A[i-1][j] + A[i]
[j-1] - A[i-1][j-1];

    int r1 = N / 2, r2 = N / 2 + 1;
    int c1 = N / 2, c2 = N / 2 + 1;
    int res = 0; string path;
    while(true){
        auto nxt = Next({r1, r2, c1, c2});
        auto mx = *max_element(nxt.begin(), nxt.end());
        if(get<0>(mx) <= 0) break;
        int v, p; tie(v,p,r1,r2,c1,c2) = mx;
        res += v; path += S[-p];
    }
    cout << res << "\n" << path;
}
```

BOJ 23280. 엔토피아의 기억강화

> 3×4 크기의 게임판의 각 칸을 눌러야 하는 순서가 정해져 있다. 게임판은 왼손 또는 오른손 엄지 손가락을 이용해 누를 수 있으며, 왼손 엄지 손가락이 한 칸을 이동할 때는 A만큼의 체력을, 오른쪽 엄지 손가락이 한 칸 이동할 때는 B만큼의 체력을 사용한다.
> 처음에 왼손은 1번 칸, 오른손은 3번 칸에 있을 때, 모든 칸을 순서대로 누르기 위해 필요한 체력의 최솟값을 구하라.

간단한 동적 계획법 문제입니다.

$D(i,j,k) :=$ 왼손 엄지 손가락은 j번 칸, 오른쪽 엄지 손가락은 k번 칸에 있고, i번째 칸까지 눌렀을 때, 지금까지 소모한 체력의 최솟값이라고 정의합시다. j와 k 중 하나는 V_i와 동일해야 하기 때문에 실제로 고려해야 하는 상태의 개수는 $24N$가지입니다. 아래와 같이 상태 전이를 하면 $O(288 \times N)$ 정도에 문제를 해결할 수 있습니다.

- $D(i, V_i, j) \leftarrow D(i-1, k, j) + Dist(k, V_i) \times A$
- $D(i, j, V_i) \leftarrow D(i-1, j, k) + Dist(k, V_i) \times B$

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

int N, X, Y, A[10101], D[10101][12][12];
```

```
int Dist(int a, int b){ return abs(a/3 - b/3) + abs(a%3 - b%3); }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> X >> Y;
    for(int i=1; i<=N; i++) cin >> A[i], A[i]--;
    memset(D, 0x3f, sizeof D); D[0][0][2] = 0;
    for(int i=1; i<=N; i++){
        for(int j=0; j<12; j++){
            for(int k=0; k<12; k++) D[i][A[i]][j] = min(D[i][A[i]][j], D[i-1][k]
[j] + Dist(A[i], k) + X);
            for(int k=0; k<12; k++) D[i][j][A[i]] = min(D[i][j][A[i]], D[i-1][j]
[k] + Dist(A[i], k) + Y);
        }
    }
    cout << *min_element(&D[N][0][0], &D[N][0][0] + 144);
}
```

BOJ 8286. Road Network 2

> 무향 그래프의 정점들의 차수 $d_1, d_2, \cdots, d_n$이 주어진다. 이러한 degree sequence를 갖는 트리가 존재하는지 확인하고, 존재하면 그러한 트리를 출력하라.

어떤 차수열이 트리의 차수열인지 판별하는 방법부터 생각해 봅시다. 우선 차수의 합이 $2(n-1)$이 아니거나 차수가 n 이상인 정점이 있다면 트리가 될 수 없습니다. 또한, $n > 1$이면 차수가 0인 정점이 존재하면 안 됩니다. 즉, $n = 1$이면 $d = \{0\}$이어야 하고, $n > 1$ 이면 $1 \le d_i \le n-1, \sum d_i = 2(n-1)$이 트리가 존재할 필요 조건입니다.

사실 이 조건은 필요 충분 조건입니다. 즉, $n \ge 2, 1 \le d_i < n, \sum d_i = 2n-2$이면 트리가 존재함을 보일 수 있습니다. 엄밀한 증명은 하지 않고, 트리를 실제로 구성할 수 있는 간단한 아이디어만 작성합니다.

차수 내림차순으로 정렬합시다. 이때 모든 $1 \le k \le n$에 대해 $d_1 + d_2 + \cdots + d_k \ge 2(k-1)$이 성립하고, 등호는 $k = n$일 때만 성립합니다. 따라서 귀납적으로 트리를 만들 수 있습니다.
구체적으로, 간선이 많이 달려야 하는 정점부터 순서대로 배치하면서, 각 정점마다 간선이 몇 개 더 필요한지를 관리하면 됩니다. $k < n$이면 $d_1 + d_2 + \cdots + d_k > 2(k-1)$을 만족하므로 간선을 요구하는 정점이 항상 1개 이상 존재하기 때문에 새로운 정점을 추가할 때마다 간선을 연결할 수 있습니다.

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

int N;
vector<pair<int,int>> V;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; V.resize(N);
    for(int i=0; i<N; i++) cin >> V[i].first, V[i].second = i + 1;
    sort(V.begin(), V.end(), [](auto a, auto b){ return a.first > b.first; });
    long long S = accumulate(V.begin(), V.end(), 0LL, [](auto a, auto b){ return
a + b.first; });
    if(V[0].first >= N || V.back().first < 1 || S != 2 * (N-1)){ cout << "BRAK";
return 0; }
    for(int i=0, j=1; i<N; i++){
        for(; V[i].first; V[i].first--, V[j++].first--) cout << V[i].second << "
" << V[j].second << "\n";
    }
}
```

BOJ 27953. 공룡 게임

> 크롬 공룡 게임에서 N개의 장애물이 주어진다. 플레이어는 점프와 슬라이딩을 할 수 있는데, 점프의 쿨타임은 A, 슬라이딩의 쿨타임은 B이고, 점프를 하면 X, 슬라이딩을 하면 Y 만큼의 패널티를 받는다. 모든 장애물을 넘을 수 있는지 판별하고, 가능하다면 모든 장애물을 넘기 위해 필요한 패널티의 최솟값을 구하라.

가장 먼저 생각하는 풀이는 $D(i,j,k) := i$번째 장애물까지, 마지막 점프는 j번째 장애물에서, 마지막 슬라이드는 k번째 장애물에서 했을 때의 최소 비용이라고 정의하는 것입니다. j와 k 중 하나는 i이고, 다른 하나는 i보다 작아야 합니다. 두 가지 경우로 나눠서 생각해 봅시다.

- $D(i,i,k)$
  - $k < i-1$ 이면 $D(i,i,k) = D(i-1,i-1,k) + X$
  - $k = i-1$ 이면 $D(i-1,t,i-1) + X$의 최솟값
- $D(i,j,i)$
  - $j < i-1$ 이면 $D(i,j,i) = D(i-1,j,i-1) + Y$
  - $j = i-1$ 이면 $D(i-1,i-1,t) + Y$의 최솟값

투 포인터 기법을 이용해 모든 위치에서의 t 범위를 $O(N)$에 계산할 수 있습니다.

상태를 이렇게 정의하면 $O(N^3)$이라서 문제를 해결할 수 없으니, $i = j$ or $i = k$ 조건을 이용해 상태의 개수를 줄여야 합니다.
$E(i,0,k) = D(i,i,k), E(i,1,j) = D(i,j,i)$라고 다시 정의합시다.

- $E(i,0,k)$
  - $k < i-1$ 이면 $E(i,0,k) = E(i-1,0,k) + X$
  - $k = i-1$ 이면 $E(i-1,1,t) + X$의 최솟값
- $E(i,1,j)$
  - $j < i-1$ 이면 $E(i,1,j) = E(i-1,i,j) + Y$
  - $j = i-1$ 이면 $E(i-1,0,t) + Y$의 최솟값

i마다 구간의 최솟값을 구하는 연산을 최대 2번 수행하기 때문에 최솟값을 naive하게 구해도 괜찮습니다. 시간 복잡도와 공간 복잡도 모두 $O(N^2)$이 되므로 문제를 해결할 수 있습니다.

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

ll N, A, B, X, Y, T[5050], S[5050], PA[5050], PB[5050], D[5050][2][5050];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> A >> B >> X >> Y; T[0] = -1e9;
    for(int i=1; i<=N; i++) cin >> T[i] >> S[i];
    for(int i=1, j=0, k=0; i<=N; i++){
        while(j + 1 <= i && T[i] - T[j+1] >= A) j++;
        while(k + 1 <= i && T[i] - T[k+1] >= B) k++;
        PA[i] = j; PB[i] = k;
    }

    memset(D, 0x3f, sizeof D); D[0][0][0] = D[0][1][0] = 0;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=i-2; j++){
```

```
            if((S[i] & 1) && T[i] - T[i-1] >= A) D[i][0][j] = D[i-1][0][j] + X;
            if((S[i] & 2) && T[i] - T[i-1] >= B) D[i][1][j] = D[i-1][1][j] + Y;
        }
        if(S[i] & 1) D[i][0][i-1] = *min_element(D[i-1][1], D[i-1][1]+PA[i]+1) +
X;
        if(S[i] & 2) D[i][1][i-1] = *min_element(D[i-1][0], D[i-1][0]+PB[i]+1) +
Y;
    }

    ll R = INF;
    for(int i=0; i<=N; i++) R = min({R, D[N][0][i], D[N][1][i]});
    cout << (R < INF ? R : -1);
}
```

BOJ 27491. Sum Over Zero

> 정수로 구성된 수열이 주어진다. 수열에서 서로 겹치지 않는 구간을 몇 개 만들 건데, 각 구간에 포함된 수들의 합은 0 이상이어야 한다. 만든 구간의 길이의 합을 최대화하라.

누적 합 배열 $S_i = A_1 + A_2 + \cdots + A_i$를 만들면 $S_r - S_{l-1} \geq 0$, 또는 $S_r \geq S_{l-1}$을 이용해서 구간의 합이 0 이상인지 빠르게 판별할 수 있습니다. 이 점을 이용하면 다음과 같은 점화식을 어렵지 않게 생각할 수 있습니다.

$D(n) = \max\{D(n-1), D(i) + n - i \text{ if } 0 \leq i < n, S_n \geq S_i\}$

단순하게 계산하면 $O(N^2)$이지만, 좌표 압축과 세그먼트 트리를 이용하면 $O(N \log N)$ 시간에 해결할 수 있습니다. 이때 세그먼트 트리에서 하는 연산이 prefix maximum query임을 이용하면 세그먼트 트리 대신 펜윅 트리를 사용할 수도 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int INF = 0xc0c0c0c0;

ll N, A[202020]; int D[202020], T[202020];
void Put(int x, int v){ for(x+=3; x<202020; x+=x&-x) T[x] = max(T[x], v); }
int Get(int x){ int r = INF; for(x+=3; x; x-=x&-x) r = max(r, T[x]); return r; }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i], A[i] += A[i-1];
    vector<ll> C(A, A+N+1);
    sort(C.begin(), C.end()); C.erase(unique(C.begin(), C.end()), C.end());
    for(int i=0; i<=N; i++) A[i] = lower_bound(C.begin(), C.end(), A[i]) -
C.begin() + 1;
    memset(D, 0xc0, sizeof D);
    memset(T, 0xc0, sizeof T);
    D[0] = 0; Put(A[0], 0);
    for(int i=1; i<=N; i++){
        D[i] = D[i-1];
        if(int v = Get(A[i]) + i; v > 0) D[i] = max(D[i], v);
        if(D[i] >= 0) Put(A[i], D[i]-i);
    }
    cout << D[N];
}
```

BOJ 11934. Fortune Telling 2

> 앞면에 A_i, 뒷면에 B_i가 쓰여진 N개의 카드가 주어진다. T가 주어지면 보이는 면에 적힌 수가 T 이하인 모든 카드를 뒤집는 쿼리를 K번 처리해야 한다. 처음에는 모든 카드가 앞면이 보이도록 세팅되어 있다. 모든 쿼리를 처리한 뒤, 보이는 면에 적힌 수들의 합을 구하라.

일관성을 잃지 않고, $A_i \le B_i$라고 생각합시다. $A_i > B_i$인 경우에는 두 수를 교환한 뒤, 처음에 B_i가 보이도록 세팅되어 있다고 생각하면 됩니다.

각 쿼리마다 주어지는 수 T에 대해서, 각 카드는 아래 세 가지 경우 중 한 가지에 해당합니다.

- $T < A_i$: 카드가 뒤집어지지 않음 => 무시해도 됨
- $A_i \le T < B_i$: 앞면이 보이는 경우에만 뒤집어짐 => 큰 수가 보이게 조정
- $B_i \le T$: 항상 뒤집어짐

두 번째 경우에 해당하는 카드는 앞에서 어떻게 조작했는지 전혀 신경쓰지 않고 큰 수가 보이도록 카드를 세팅합니다. 따라서 마지막으로 두 번째 경우에 해당하게 되는 시점 이전의 쿼리는 무시해도 됩니다. 결국 이 문제는 각 카드마다 $A_i \le T < B_i$가 성립하는 마지막 시점을 찾고, 그 이후로 $B_i \le T$인 경우가 몇 번 있었는지를 효율적으로 계산하는 문제로 바뀌게 됩니다.

j번째 쿼리에서 T_j가 주어졌다는 것을 merge sort tree나 persistent segment tree 등을 이용해 저장하면, 위에서 설명한 두 가지 연산을 모두 각 카드에 대해 $O(\log^2 K)$ 또는 $O(\log K)$ 시간에 구할 수 있습니다. 따라서 전체 문제를 $O(K \log K + N \log^2 K)$ 또는 $O((N+K) \log K)$에 해결할 수 있습니다.

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

int N, K, A[202020], B[202020];
vector<int> T[SZ<<1];

int Get(const vector<int> &v, int l, int r){
    return upper_bound(v.begin(), v.end(), r) - lower_bound(v.begin(), v.end(),
l);
}

int RangeQuery(int l, int r, int L, int R){
    int res = 0;
    for(l|=SZ, r|=SZ; l<=r; l>>=1, r>>=1){
        if(l & 1) res += Get(T[l++], L, R);
        if(~r & 1) res += Get(T[r--], L, R);
    }
    return res;
}

int RightQuery(int L, int R){
    int x = 1;
    while(x < SZ) x = x << 1 | (Get(T[x<<1|1], L, R) != 0);
    return x ^ SZ;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i] >> B[i];
    for(int i=1,t; i<=K; i++) cin >> t, T[i|SZ].push_back(t);
```

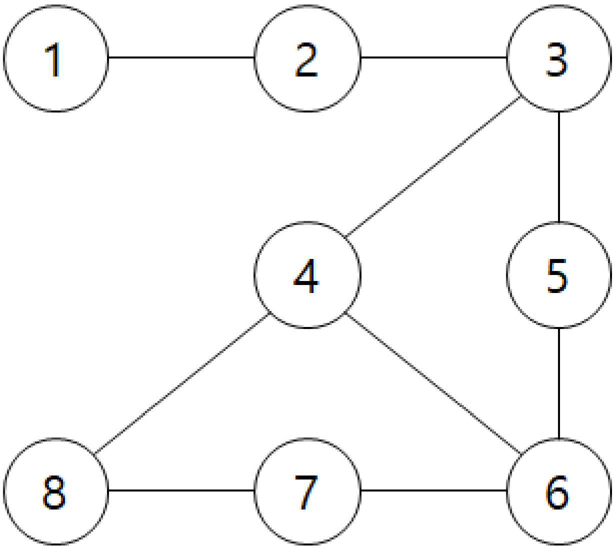
```
    for(int i=SZ-1; i; i--) merge(T[i<<1].begin(), T[i<<1].end(),
T[i<<1|1].begin(), T[i<<1|1].end(), back_inserter(T[i]));

    long long R = 0;
    for(int i=1; i<=N; i++){
        int idx = 0; // last A[i] <= T[j] < B[i]
        if(min(A[i],B[i]) != max(A[i],B[i])) idx = RightQuery(min(A[i],B[i]),
max(A[i],B[i])-1);
        int cnt = RangeQuery(idx, K, max(A[i],B[i]), 1e9); // # idx <= j, B[i]
<= T[j]
        if(idx != 0 && A[i] < B[i]) swap(A[i], B[i]);
        if(cnt % 2 == 1) swap(A[i], B[i]);
        R += A[i];
    }
    cout << R;
}
```

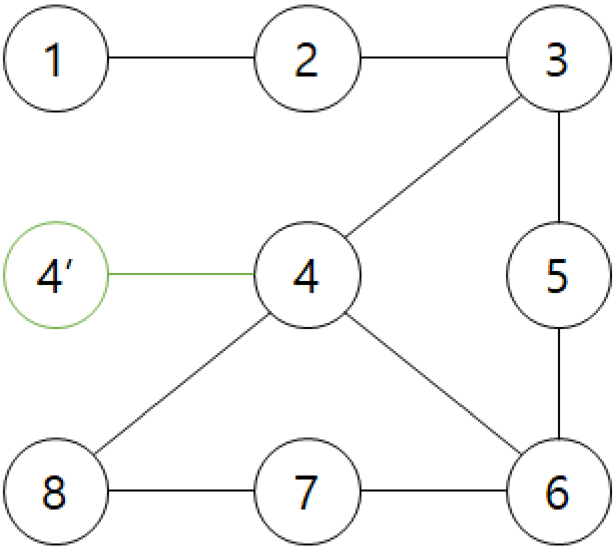
BOJ 11755. Routing a Marathon Race

> 정점에 가중치가 있는 무향 그래프가 주어진다. 1번 정점에서 N번 정점으로 가는 최소 가중치 경로를 구하라. 이때 경로의 가중치는 경로에 속한 정점, 그리고 경로에 속한 정점과 인접한 정점들의 가중치의 합으로 정의한다.

$N \le 40$이기 때문에 완전 탐색은 불가능하고, 적절한 전략을 통해 백트래킹을 해야 합니다.



위와 같은 그래프에서 `1 -> 2 -> 3 -> 5 -> 6`으로 이동했고, 6번 정점에서 어디로 갈지 결정해야 하는 상황을 생각해 봅시다. 먼저, 6번 정점에서 4번 정점으로 이동하는 경로는 탐색을 할 필요가 없습니다. 이미 3번 정점으로 인해 4번 정점의 가중치가 경로에 포함되었으므로 4번 정점으로 이동하면 손해만 보게 됩니다.



이 그림처럼 4번 정점에 추가로 정점이 붙어있을 때는 4'번 정점 때문에 고려를 해야할 것 같지만, 실제로는 탐색할 필요가 없습니다. 6번 정점에서 4번 정점으로 가는 것보다, 이전에 방문했던 4번 정점에서 3번 정점으로 가는 것이 항상 이득이기 때문입니다.

따라서 이런 상황에서는 4번 정점을 방문할 필요가 없습니다. 구체적으로, $C(i)$를 현재까지의 경로에서 i번 정점과 인접한 정점의 개수라고 할 때, $C(i) \geq 2$인 정점으로 이동하지 않아도 됩니다. 이 전략만 사용해서 가지치기를 해도 $O(3^{N/3})$ 시간이 보장되기 때문에 문제를 해결할 수 있습니다.

$T(N)$을 N개의 정점으로 구성된 그래프에서의 연산량이라고 정의합시다. 각 정점의 차수를 d_i라고 하면, $T(N) = \max_i \{d_i \times T(N - d_i)\}$가 성립합니다. i번 정점으로 인해 d_i개의 정점의 $C(*)$가 증가해서 탐색 범위를 좁힌 다음, 인접한 정점에 대해 재귀 호출을 하기 때문입니다.
이때 $T(N)$은 1부터 N까지의 정수를 적절히 사용해서 합을 N으로 만들 때의 곱의 최댓값을 구하는 것과 동일합니다. 이 값이 $O(3^{N/3})$임을 증명하면 됩니다.

정수 $a \geq 4$는 $2 + (a - 2)$로 쪼갤 수 있고, $2(a - 2) = 2a - 4 \geq a$이므로 4 이상의 정수는 2와 $a - 2$로 쪼개는 것이 이득입니다. 이제 $1, 2, 3$만 처리하면 되는데, $2^3 < 3^2$이므로 2보다는 3을 사용하는 것이 더 좋습니다. 따라서 합이 N인 양의 정수들의 곱의 최댓값은 $O(3^{N/3})$입니다.

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

int N, M, A[44], C[44], P[44], U[44], R=1e9;
vector<int> G[44];

void DFS(int v){
    if(v == N){
        memset(U, 0, sizeof U);
        for(int i=N; i; i=P[i]) for(auto j : G[i]) U[j] = A[j];
        R = min(R, accumulate(U+1, U+N+1, 0));
        return;
    }
    for(auto i : G[v]) C[i] += 1;
    for(auto i : G[v]) if(C[i] == 1) P[i] = v, DFS(i);
    for(auto i : G[v]) C[i] -= 1;
}

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

```
    for(int i=1,u,v; i<=M; i++) cin >> u >> v, G[u].push_back(v),
G[v].push_back(u);
    C[1] = 1; DFS(1);
    cout << R;
}
```

BOJ 27814. Emacs++

> 괄호 문자열이 주어진다. 커서를 i번째 문자에서 왼쪽으로 한 칸 옮기는데 L_i, 오른쪽으로 한 칸 옮기는데 R_i, 매칭되는 괄호의 위치로 옮기는데 P_i 만큼의 시간이 든다. S_j, E_j가 주어지면 커서를 S_j번째 문자에서 E_j번째 문자로 옮기는데 드는 최소 시간을 구하는 쿼리를 처리하라.

N각형을 그린 다음, 각 꼭짓점에 문자를 순서대로 하나씩 대응시킨 그림을 생각해 봅시다. 다각형의 각 변은 단방향 간선 2개로 구성되어 있으며, 왼쪽으로 가는 간선의 가중치는 L_i, 오른쪽으로 가는 간선의 가중치는 R_i, 그리고 대응되는 괄호로 가는 대각선의 가중치는 P_i인 그래프를 만들 것입니다. 이런 그래프에 가중치가 무한대인 간선을 적당히 추가하면 N각형을 삼각 분할한 그래프로 만들 수 있습니다.

이제, 다각형의 삼각 분할 그래프가 주어졌을 때 최단 경로 쿼리를 해결하는 문제로 바꿔서 생각합시다.

계산 기하를 공부하다 보면 삼각 분할과 함께 붙어다니는 키워드로 "듀얼 트리(Dual Tree)"라는 것이 있습니다. 평면 그래프의 듀얼 그래프와 비슷하게, 각 삼각형을 정점으로, 인접한 삼각형으로 연결한 트리를 듀얼 트리라고 부릅니다. 이 트리에서 각 정점의 차수는 최대 3이기 때문에 이진 트리의 좋은 성질을 기하 문제에서도 활용할 수 있도록 도와줍니다.

이진 트리의 유용한 성질 중 하나는 제거했을 때 나눠지는 두 컴포넌트의 크기를 모두 $2/3N$ 이하로 만드는 간선이 있다는 것입니다. 듀얼 그래프의 간선은 원본 그래프의 간선과 대응되기 때문에, 삼각 분할 그래프에는 부분 문제의 크기를 $2/3$ 이하로 만드는 대각선이 있음을 알 수 있습니다.

우리는 그러한 대각선을 찾은 다음 대각선을 넘나드는 쿼리를 처리하고, 대각선을 기준으로 한 쪽에 종속된 쿼리는 재귀적으로 처리하는 분할 정복을 할 것입니다. 대각선을 넘나드는 쿼리는 대각선의 양쪽 끝점에서 다익스트라 알고리즘을 돌리면 어렵지 않게 처리할 수 있습니다. 전체 시간 복잡도는 $O(N \log^2 N + Q \log N)$입니다.

또는 그래프가 outer planar graph라는 것, 즉 그래프의 treewidth가 2인 점을 이용해 $O(N \log N + Q \log N)$ 정도에 해결하는 방법도 존재합니다.

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

cost_t R[101010];

vector<cost_t> Dijkstra(int n, int s, const vector<vector<pair<int,cost_t>>> &g)
{
    vector<cost_t> dst(n, INF);
    priority_queue<pair<cost_t,int>, vector<pair<cost_t,int>>, greater<>> pq;
    pq.emplace(dst[s]=0, s);
    while(!pq.empty()){
        auto [c,v] = pq.top(); pq.pop();
        if(c == dst[v]) for(auto [i,w] : g[v]) if(dst[i] > c + w)
pq.emplace(dst[i]=c+w, i);
    }
    return dst;
}
```

```
void Solve(int n, vector<tuple<int,int,cost_t>> edges,
vector<tuple<int,int,int>> qry){
    if(n <= 3){
        vector<vector<cost_t>> dst(n, vector<cost_t>(n, INF));
        for(int i=0; i<n; i++) dst[i][i] = 0;
        for(auto [u,v,w] : edges) dst[u][v] = min(dst[u][v], w);
        for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++)
dst[i][j] = min(dst[i][j], dst[i][k] + dst[k][j]);
        for(auto [a,b,i] : qry) R[i] = dst[a][b];
        return;
    }

    int small = 0, u = 0, v = 0;
    for(auto [a,b,w] : edges){
        int x = min(a, b), y = max(a, b);
        int sz = min(y-x+1, n-(y-x-1));
        if(sz > small) u = x, v = y, small = sz;
    }

    vector<int> id(n), p1(n, -1), p2(n, -1);
    for(int i=0; i<n; i++){
        if(i == u || i == v) id[i] = 3;
        else if(u < i && i < v) id[i] = 1;
        else id[i] = 2;
    }
    for(int i=0, j=0; i<n; i++) if(id[(u+i)%n] & 1) p1[(u+i)%n] = j++;
    for(int i=0, j=0; i<n; i++) if(id[(v+i)%n] & 2) p2[(v+i)%n] = j++;

    vector<tuple<int,int,cost_t>> e1, e2;
    vector<tuple<int,int,int>> q1, q2, qq;
    for(auto [a,b,w] : edges){
        if((id[a] & 1) && (id[b] & 1)) e1.emplace_back(p1[a], p1[b], w);
        if((id[a] & 2) && (id[b] & 2)) e2.emplace_back(p2[a], p2[b], w);
    }
    for(auto [a,b,i] : qry){
        if((id[a] & 1) && (id[b] & 2)) qq.emplace_back(a, b, i);
        else if((id[a] & 2) && (id[b] & 1)) qq.emplace_back(a, b, i);
        else if((id[a] & 1) && (id[b] & 1)) q1.emplace_back(p1[a], p1[b], i);
        else q2.emplace_back(p2[a], p2[b], i);
    }
    Solve(v-u+1, e1, q1); Solve(n-(v-u-1), e2, q2);

    vector<vector<pair<int,cost_t>>> g1(n), g2(n);
    for(auto [a,b,w] : edges) g1[a].emplace_back(b, w), g2[b].emplace_back(a,
w);
    auto d1 = Dijkstra(n, u, g1), d2 = Dijkstra(n, v, g1);
    auto r1 = Dijkstra(n, u, g2), r2 = Dijkstra(n, v, g2);

    for(auto [a,b,i] : qry){
        R[i] = min(R[i], r1[a] + d1[b]);
        R[i] = min(R[i], r2[a] + d2[b]);
    }
}

void Solve(){
    int N, Q; string S; cin >> N >> Q >> S;
    vector<cost_t> Le(N), Ri(N), Pa(N);
    vector<int> St(Q), Ed(Q);
```

```
    for(auto &i : Le) cin >> i;
    for(auto &i : Ri) cin >> i;
    for(auto &i : Pa) cin >> i;
    for(auto &i : St) cin >> i;
    for(auto &i : Ed) cin >> i;

    vector<tuple<int,int,cost_t>> edges;
    vector<tuple<int,int,int>> qry(Q);

    for(int i=1; i<N; i++) edges.emplace_back(i-1, i, Ri[i-1]),
edges.emplace_back(i, i-1, Le[i]);
    for(int i=0; i<Q; i++) qry[i] = {St[i]-1, Ed[i]-1, i};

    stack<int> stk;
    vector<pair<int,int>> match;
    for(int i=0; i<N; i++){
        if(S[i] == '(') stk.push(i);
        if(S[i] == ')') match.emplace_back(stk.top(), i), stk.pop();
    }
    sort(match.begin(), match.end(), [](auto a, auto b){ return a.second -
a.first < b.second - b.first; });

    set<int> st;
    for(int i=0; i<N; i++) st.insert(i);
    for(auto [a,b] : match){
        edges.emplace_back(a, b, Pa[a]);
        edges.emplace_back(b, a, Pa[b]);
        if(a + 1 == b) continue;
        for(auto it=next(st.find(a)); *it!=b; it=st.erase(it))
edges.emplace_back(a, *it, INF), edges.emplace_back(*it, a, INF);
    }
    vector<int> vec(st.begin(), st.end());
    for(int i=2; i+1<vec.size(); i++) edges.emplace_back(0, vec[i], INF),
edges.emplace_back(vec[i], 0, INF);
    if(get<0>(match.back()) != 0 || get<1>(match.back()) + 1 != N)
edges.emplace_back(0, N-1, INF), edges.emplace_back(N-1, 0, INF);

    memset(R, 0x3f, sizeof(R[0]) * Q);
    Solve(N, edges, qry);
    assert(0 <= *min_element(R, R+Q) && *max_element(R, R+Q) <= 1e11);
    cout << accumulate(R, R+Q, 0LL) << "\n";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int TC; cin >> TC;
    for(int tc=1; tc<=TC; tc++) cout << "Case #" << tc << ": ", Solve();
}
```