SCPC 2회 2차예선 풀이 + SCPC 3회 2차예선 풀이

서울대학교 컴퓨터공학부 18학번 김동현

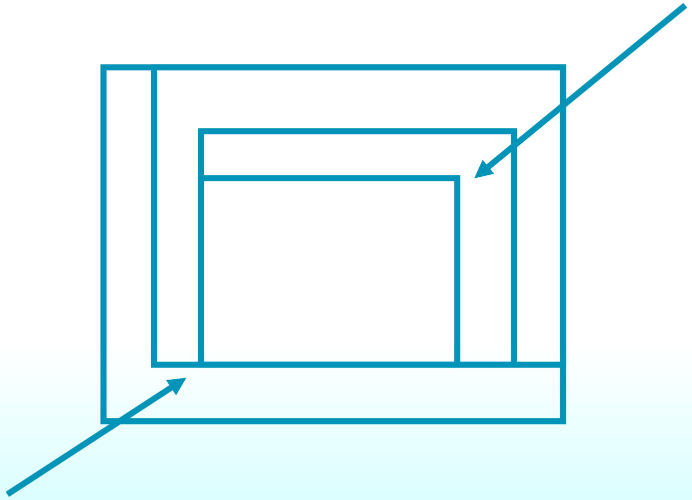
SCPC 2회 2차예선 풀이

1번 - Rectangles

- 좌표축에 평행하고 꼭짓점의 좌표가 모두 정수인 직사각형들이 N개 주어집니다.
- 직사각형들 중 몇 개를 골라서 부분집합을 만들었을 때, 임의의 두 원소를 골라도 둘 중 하나가 다른 하나에 포함되도록 했다고 합시다.
- 이 조건을 만족시킬 수 있는 부분집합의 최대 크기를 출력하세요.
- $1 \leq N \leq 5{,}000$

1번 - Rectangles

- 순서를 잘 정하면 뭔가 DP가 될 것 같습니다.
- 순서를 어떻게 잘 정할 수 있을까요?
- 집합의 원소를 바깥쪽에 있는 것부터 고른다고 해 봅시다.
- 바깥쪽에서 안쪽으로 들어갈 때, 왼쪽 끝점의 좌표와 오른쪽 끝점의 좌표가 이동하는 방향을 따라 정렬을 잘 할 수 있을까요?



1번 - Rectangles

- 이런 식으로 정렬하면 됩니다.
- 왼쪽 아래 끝점은 pair 기준으로 오름차순, 오른쪽 위 끝점은 pair 기준으로 내림차순입니다.
- 비교함수를 짜기가 귀찮아서 그냥 오른쪽 위 끝점 좌표에 -1을 곱해버렸습니다.

```
for(int i = 0; i < n; i++) {
  cin >> a[i].x.x >> a[i].x.y >> a[i].y.x >> a[i].y.y;
  a[i].y.x *= -1;
  a[i].y.y *= -1;
}
sort(all(a));
for(int i = 0; i < n; i++) {
  a[i].y.x *= -1;
  a[i].y.y *= -1;
}
```

1번 - Rectangles

- 순서를 잘 매긴 다음에는 간단한 DP로 chain이 최대 몇 개까지 이어질 수 있는지 계산할 수 있습니다.
- 포함되는 조건을 따져주면서 DP 값을 갱신하면 됩니다.

```
vint d(n);
int ans = 0;
for(int i = 0; i < n; i++) {
  d[i] = 1;
  for(int j = 0; j < i; j++) {
    if(
      a[j].x.x <= a[i].x.x
      && a[j].x.y <= a[i].x.y
      && a[i].y.x <= a[j].y.x
      && a[i].y.y <= a[j].y.y
    ) {
      d[i] = max(d[i], d[j] + 1);
    }
  }
  ans = max(ans, d[i]);
}
cout << ans << '\n';
```

2번 - 프리랜서

- P사와 Q사가 총 N주동안 번역을 요청합니다.
- 각 주차마다 번역 수수료가 다릅니다.
- P사가 i주차에 요청한 문서는 i주차를 소모하여 완료할 수 있습니다.
- Q사가 i주차에 요청한 문서는 (i-1), i주차를 모두 소모하여 완료할 수 있습니다. (단, 1주차에 요청한 문서는 1주차에 끝낼 수 있습니다)
- 얻을 수 있는 최대의 번역 수수료를 구하세요.
- $1 \le N \le 10{,}000$

2번 - 프리랜서

- 매우 간단한 DP를 설계할 수 있습니다.
- D[i] : i주차가 끝났을 때 얻을 수 있는 최대 수수료.
- D[1] = max(P[1], Q[1])
- D[i] = max(D[i−1] + P[i], D[i−2] + Q[i])

- 이게 끝입니다.
- 1번보다 쉬운 것 같네요..

```cpp
void solve() {
  int n;
  cin >> n;

  vint a(n + 1), b(n + 1);
  for(int i = 1; i <= n; i++) cin >> a[i];
  for(int i = 1; i <= n; i++) cin >> b[i];

  vint d(n + 1);
  d[1] = max(a[1], b[1]);
  for(int i = 2; i <= n; i++) {
    d[i] = d[i - 1] + a[i];
    d[i] = max(d[i], d[i - 2] + b[i]);
  }
  cout << d[n] << '\n';
}
```

3번 – 땅 나누기

- 2차원 평면 상에 광산이 N개 있습니다. 각 광산은 철 광산 또는 구리 광산입니다.
- 어떤 지점 p를 원점으로 정해서, (1 또는 3사분면에 있는 구리 광산의 개수) + (2 또는 4사분면에 있는 철 광산의 개수) 를 최대화하고자 합니다.
- 최대로 가능한 값을 출력하세요.
- $1 \leq N \leq 100{,}000$

3번 – 땅 나누기

- 기본적으로, Plane Sweeping의 아이디어를 적용할 수 있습니다.
- 수직선을 특정 위치에 긋는다고 하면, 그 오른쪽에 있는 철 광산은 구리 광산으로, 구리 광산은 철 광산으로 바꿔서 왼쪽으로 대칭이동시켰다고 생각해도 됩니다.
- 이제, 문제는 최적의 수평선의 위치를 빠르게 결정하는 것입니다.
- 수평선을 기준으로 (위쪽의 철 광산 개수) + (아래쪽의 구리 광산 개수)를 최대화하면 됩니다.

3번 – 땅 나누기

- Plane Sweeping을 한다면 중간에 업데이트 연산이 빠르게 이루어져야 하므로, 세그먼트 트리 같은 자료 구조를 쓸 수 있을지 생각해 봅시다.
- 어떤 문제를 세그먼트 트리로 풀 수 있도록 만들기 위해서는 분할 정복으로 문제를 어떻게 풀지 고민해 보면 좋습니다.
- 각 노드가 다음의 세 가지 값을 가지고 있다고 합시다.
  - 내 구간에 있는 구리 광산의 개수
  - 내 구간에 있는 철 광산의 개수
  - 기준선을 구간 내 어딘가에 잡았을 때 (위쪽의 철 광산 개수) + (아래쪽의 구리 광산 개수) 의 최댓값

3번 – 땅 나누기

- 세그먼트 트리의 어떤 노드에서 두 자식 노드의 세 가지 값을 각각 알고 있다고 하면, 부모 노드의 세 가지 값을 O(1)에 계산할 수 있습니다.
  - 구리 광산의 개수, 철 광산의 개수는 그냥 두 자식 노드의 값을 더하면 됩니다.
  - (위쪽의 철 광산 개수) + (아래쪽의 구리 광산 개수) 의 최댓값은 기준선이 왼쪽 자식 노드에 있을 경우와 오른쪽 자식 노드에 있을 경우로 나누면 됩니다.

```
struct Node { int zero, one, ans; };

Node mrg(const Node &l, const Node &r) {
  return {
    l.zero + r.zero,
    l.one + r.one,
    max(l.zero + r.ans, l.ans + r.one)
  };
}
```

3번 – 땅 나누기

- 수직선의 좌표를 맨 왼쪽부터 오른쪽까지 한번 쭉 훑으면서, 세그먼트 트리에 적절히 갱신을 해 주면서 어느 시점에 답이 가장 커지는지를 보면 됩니다.

```
void solve() {
  int n;
  cin >> n;

  Seg::i(n);
  vector<vpii> mines(n + 1);
  for(int i = 0; i < n; i++) {
    int x, y, z;
    cin >> x >> y >> z;
    mines[x].emplace_back(y, z);
    Seg::u(y, !z);
  }

  int ans = Seg::d[1].ans;
  for(int i = 1; i <= n; i++) {
    for(pii &p : mines[i]) Seg::u(p.x, p.y);
    ans = max(ans, Seg::d[1].ans);
  }
  cout << ans << '\n';
}
```

3번 – 땅 나누기

- 세그먼트 트리의 구현입니다.
- mrg 함수가 핵심입니다.
- 어떤 구간에서 답을 구하는 쿼리도 O(logn)에 할 수 있습니다.
  - 이 문제에서는 필요 없어서 구현하지 않았습니다.

```
namespace Seg {
  struct Node { int zero, one, ans; };
  int sz;
  vector<Node> d;

  void i(int n) {
    for(sz = 1; sz <= n; sz *= 2);
    d = vector<Node>(2 * sz);
  }

  Node mrg(const Node &l, const Node &r) {
    return {
      l.zero + r.zero,
      l.one + r.one,
      max(l.zero + r.ans, l.ans + r.one)
    };
  }

  void u(int x, int v) {
    d[x += sz] = {!v, v, 1};
    for(x /= 2; x; x /= 2) d[x] = mrg(d[2 * x], d[2 * x + 1]);
  }
}
```

4번 - 독재자

- N개의 집과 M개의 가중치 있는 양방향 도로가 주어진다.
- 1번 집은 독재자의 집으로, 어느 날 독재자는 1번 집에서 다른 모든 집으로 가는 최단경로에 속하는 N-1개의 도로만 남기기로 했다. (그 방법이 유일함이 보장됨)
- 이제 없어진 도로 중 정확히 하나를 복구할 것인데, 복구 할 수 있는 도로의 조건과 그 때 이동 규칙이 어떻게 되는지는 다음 슬라이드에 있다.
- 복구할 도로를 잘 선택하여 모든 (a, b) 쌍에 대해 a->b로 가는 이동거리의 총합이 최대한 많이 줄어들도록 하여라. (줄어들지 않는다면 최대한 덜 늘어나도록 하여라.)
- $1 \le N \le 10{,}000$, $1 \le M \le 100{,}000$

4번 - 독재자

원래 있던 도로들 중 단 하나를 다시 복구할 수 있는 비용을 마련하였다. 복구하는 도로는 $i \neq 1$이고 $j \neq 1$인 어떤 두 집 i, j를 연결하는 길이 (현재 상태에서) 독재자의 집을 **중간에** 거쳐가는 경우만 선택될 수 있다고 한다. 두 노드 i, j를 연결하는 도로가 복구된 경우 어떤 노드 a에서 b로 가는 길을 찾는 방법은 다음과 같다. 사람들은 전체 도로망에 대한 정보를 다 가지고 있는 것이 아니라서 약간 단순한 방법을 사용한다.

1. 짧은 길을 찾고 싶은 것이므로 동일한 도로를 두 번 사용하는 경우는 없다.
2. 현재 상태에서 a와 b를 연결하는 길이 1번 집을 포함하지 않는 경우 길을 변경하지 않고 그대로 사용한다. (여기서 1번 집을 포함한다는 말은 a나 b가 1번인 경우도 **허용**하는 것이다.)
3. 현재 상태에서 a와 b를 연결하는 길이 1번 집을 포함하는 경우 다음 규칙을 사용한다.
   A. 집 a에서 출발하여 세 집 $1, i, j$중 하나도 지나지 않은 경우는 세 집 중 가장 가까운 집 방향을 항상 선택하여 움직인다.
      즉, 새로 생긴 길을 쓰는 쪽을 선호하는 것이다. 가장 가까운 집이 유일하지 않은 경우는 i, j 중 더 가까운 것의 방향을 선택한다.
      이 조건이 적용되는 경우에는 i, j 중 더 가까운 것이 항상 존재한다. 집 a가 $1, i, j$ 중 하나인 경우는 바로 B 혹은 C의 규칙을 사용한다.
      즉, 0개의 도로를 지나서 $1, i, j$ 중 하나에 도착한 것으로 간주하는 것이다.
   B. A의 규칙을 따르다가 세 집 $1, i, j$ 중 1에 가장 먼저 도착한 경우는 원래의 길을 그대로 사용한다.
   C. A의 규칙을 따르다가 세 집 $1, i, j$ 중 i 나 j 에 가장 먼저 도착한 경우는 i, j 를 연결하는 도로를 반드시 사용하고, 그 이후는 b까지의 가장 짧은 길을 이용한다. (단, 1번 규칙을 어기면 안 된다.)

4번 - 독재자

- 이렇게 풀기 싫게 생긴 문제는 처음입니다..
- 트리를 구축하는 것은 (그나마) 쉬우니 일단 트리를 만들었다고 합시다.
- 이제 문제 조건을 열심히 분석해 보면 다음 사실을 알 수 있습니다.
  - 1번 정점과 직접 연결된 정점들을 기준으로 서브트리들을 나눕시다.
  - T(x)를 x번 정점이 속한 (1번 정점과 연결된 정점 기준) 서브트리라고 합시다.
  - d(x,y)를 x번 정점과 y번 정점의 트리 상에서 거리라고 합시다.
  - u-v 간선이 이번에 복구할 간선이라고 합시다.
  - 영향을 받는 경로들은 아래의 두 가지입니다.
    - (1) T(u)에 속하고 $d(u,x) \le d(1,x)$인 x에서 출발해서 T(u)에 속하지 않는 y로 가는 경로
    - (2) T(v)에 속하고 $d(v,z) \le d(1,z)$인 z에서 출발해서 T(v)에 속하지 않는 w로 가는 경로

4번 – 독재자

- 아래와 같은 값들을 잔뜩 정의하면 어떤 도로를 복구했을 때 총 이동 거리의 변화량을 오른쪽과 같은 식으로 쓸 수 있습니다.
- subn[x] : T(x)의 루트
- nearest[x] : d(x,y)≤d(1,y)인 y 중 1번 정점에 가장 가까운 y
- cnt[x] : x를 조상으로 갖는 노드 y의 개수 (x 자신도 포함!)
- depsum[x] : x를 조상으로 갖는 y에 대해 d(1,y)의 합
- distsum[x] : T(x)에 속하는 y에 대해 d(x,y)의 합
- nearsum[x] : nearest[x]를 조상으로 갖는 y에 대해 d(x,y)의 합

```
ll ans = -INF;
for(Edge &u : edges) {
  if(!subn[u.x] || !subn[u.y] || subn[u.x] == subn[u.y]) continue;
  int xnear = nearest[u.x], ynear = nearest[u.y];
  int xsub = subn[u.x], ysub = subn[u.y];
  ll bef =
    depsum[xnear] * (n - cnt[xsub])
    + (depsum[1] - depsum[xsub]) * cnt[xnear]
    + depsum[ynear] * (n - cnt[ysub])
    + (depsum[1] - depsum[ysub]) * cnt[ynear];
  ll aft =
    nearsum[u.x] * (n - cnt[xsub])
    + u.c * cnt[xnear] * (n - cnt[xsub])
    + distsum[u.y] * cnt[xnear]
    + d[u.y] * (n - cnt[ysub] - cnt[xsub]) * cnt[xnear]
    + (depsum[1] - depsum[ysub] - depsum[xsub]) * cnt[xnear]
    + nearsum[u.y] * (n - cnt[ysub])
    + u.c * cnt[ynear] * (n - cnt[ysub])
    + distsum[u.x] * cnt[ynear]
    + d[u.x] * (n - cnt[xsub] - cnt[ysub]) * cnt[ynear]
    + (depsum[1] - depsum[xsub] - depsum[ysub]) * cnt[ynear];
  ans = max(ans, bef - aft);
}
cout << ans << '\n';
```

4번 - 독재자

- subn, cnt, depsum 배열은 간단하게 O(N)에 구할 수 있습니다.
- nearest, distsum, nearsum 배열은 조금 더 어렵습니다..
- 열심히 노력을 하면 O(NlogN)에 모두 구할 수 있습니다.

- 기본적인 아이디어는 각 T(x)의 루트에서 DFS를 수행하면서 스택에 있는 노드들을 관리하는 것입니다.
- nearset 배열은 스택 위에서 이분탐색을 통해 O(NlogN)에 계산됩니다.

4번 - 독재자

- distsum 배열은 중심이 되는 노드를 간선 하나를 타고 옮길 때 변화량을 쉽게 계산할 수 있습니다.
  - cnt 배열을 이용하면 됩니다.
- nearsum 배열을 계산하기 위해서는 DFS 과정에서 스택에 값을 한 종류 더 관리하면 됩니다.
  - wingsum[x] : 현재 DFS 과정에서 스택에 노드 x가 있고, 그 바로 다음에 방문한 노드가 y라고 할 때 x는 조상으로 갖지만 y는 조상으로 갖지 않는 노드 z들에 대해 d(x,z) - d(1,x) 를 모두 합한 값입니다.
  - distsum[x]와 nearest[x]를 알고 있다면 nearsum[x]를 wingsum의 prefix sum 등을 이용해 계산할 수 있습니다.

4번 - 독재자

- 전체 코드는 GitHub에 있습니다.
- nearest, distsum, nearsum을 계산하는 부분만 슬라이드에 수록하였습니다.

```
vint nearest(n + 1);
vll nearsum(n + 1), distsum(n + 1);
{
  vint stk;
  vll wprefix, wingsum(n + 1);
  function<void(int, int)> f = [&](int x, int y) {
    stk.push_back(x);
    {
      int l = 0, r = int(stk.size()) - 1;
      while(l < r) {
        int m = (l + r) / 2;
        if(d[stk[m]] < d[x] - d[stk[m]]) l = m + 1;
        else r = m;
      }
      nearest[x] = stk[l];
      nearsum[x] = distsum[x] - wprefix[l]
        - (cnt[subn[x]] - cnt[nearest[x]]) * d[x];
    }
    for(pil &p : te[x]) if(p.x != y) {
      distsum[p.x] = distsum[x] + (cnt[subn[x]] - 2 * cnt[p.x]) * p.y;
      wingsum[x] = depsum[x] - depsum[p.x] - 2 * (cnt[x] - cnt[p.x]) * d[x];
      wprefix.push_back(wprefix.back() + wingsum[x]);
      f(p.x, x);
      wprefix.pop_back();
    }

    stk.pop_back();
  };

  wprefix.push_back(0LL);
  for(pil &p : te[1]) {
    distsum[p.x] = nearsum[p.x] = depsum[p.x] - cnt[p.x] * d[p.x];
    f(p.x, 1);
  }
}
```

5번 - 난민촌

- N명의 난민이 보급품을 받습니다.
- 각 난민의 최소 지급량은 (x, y)의 정수 좌표로 나타납니다.
- K개의 추가 보급품이 있어서, 모두 소진해야 하며 1사람당 최대 1개를 받을 수 있습니다. ($K \le N$)
- 불평등지수란, 어떤 좌표 (a, b)를 잘 선택해서 다음 값들의 최댓값을 최소화했을 때 그 값입니다.
  - 각 난민에 대해 $|x-a|+|y-b|+$(그 난민이 받은 추가 보급품의 양)
- 추가 보급품을 적절히 배분하여 불평등지수를 최소화하세요.
- $1 \le N \le 20$

5번 – 난민촌

- 추가 보급품이 없다고 해 봅시다.
- 기준 좌표 (a, b)를 선택하게 되면, 그것을 중심으로 하고 모든 N개의 점을 포함하는 최소 크기의 마름모꼴(45° 기울어진 정사각형)을 구하는 것이 됩니다.
- 이런 세팅에서, Parametric search를 한다고 합시다. 즉, 마름모의 크기를 정합시다.
- 이제, 주어진 마름모꼴의 경계선 좌표를 무조건 주어진 N개의 좌표 중 하나라고 가정할 수 있습니다.
  - SCPC 1차 예선에서 이런 비슷한 세팅의 문제를 한 3번 정도 본 것만 같습니다…

5번 - 난민촌

- 추가 보급품 때문에 뭔가 접근이 힘듭니다.
- 추가 보급품에 해당하는 요소를 2차원 평면에 나타낼 수 있을까요?
- 정답에 해당하는 추가 보급품 배분과 기준 좌표 (a, b)를 알고 있다고 잠시 가정해 봅시다.
- 각 난민마다 자신이 받은 추가 보급품의 양이 k라고 할 때 (x, y+k)와 (x, y-k)에 점을 하나씩 더 찍읍시다.
- 이제 정답은 (a, b)를 중심으로 하고 추가로 찍은 모든 점을 포함하는 최소 크기의 마름모에 해당함을 알 수 있습니다!
  - 각 난민의 원래 y좌표가 b 이상일 때와 미만일 때로 나누어 보면 알 수 있습니다.

5번 - 난민촌

- 아쉽게도, 추가 보급품을 어떻게 배분하는 것이 최적인지를 바로 알 수 있는 방법은 딱히 없습니다.
- 그러면, 각 점마다 모든 보급품을 배급받는 K개의 경우에 해당하는 점을 다 찍어 보면 되지 않을까요?
- 이제 Parametric Search를 적용하면, 시도해야 하는 마름모 위치의 경우의 수가 O((NK)^2)개가 됩니다!
- 마름모 중심의 좌표가 k/2 꼴이 될 수도 있기 때문에, 모든 좌표 및 추가 보급품 양에 (또..) 2를 곱해 놓아야 합니다.

5번 - 난민촌

- 마름모의 크기 및 위치를 정했다고 할 때, 이 마름모가 실제 정답이 될 수 있는지는 어떻게 판정할까요?
- 다시 원래 문제로 돌아와서, 그리디 전략을 적용할 수 있습니다.
- 마름모의 중심에 가까울수록 많은 양의 추가 보급품을 주는 것이 무조건 최적임 ((거리)+(추가보급품)의 최댓값이 작음) 을 알 수 있습니다.
  - 이번(6회) 1차예선 1번과 유사한 논리입니다.
- O(NlogN) 시간에 판정할 수 있으므로 결정 문제의 시간복잡도가 O(N^3K^2logN)이 됩니다.

5번 – 난민촌

- 이것을 곧이곧대로 짜면 시간초과가 납니다.
- 그런데, 커팅을 하나 적용할 수 있습니다.
  - 추가 보급품을 적용하기 전에 중심에서 거리를 구했을 때 애초에 마름모 밖을 벗어난다면, 정렬을 할 필요가 없이 바로 “이 마름모는 아니다” 라고 판단해 줄 수 있습니다.
- 놀랍게도, 이 커팅을 적용하면 10초가 넘던 코드가 갑자기 1초 내에 돌아갑니다!

5번 - 난민촌

- 핵심 로직에 해당하는 (결정 문제를 푸는) 함수입니다.
- 마름모의 좌표계에 유의합시다.

```cpp
auto f = [&](int x) {
  vint xs, ys;
  for(pii &p : a) {
    xs.push_back(p.x + p.y);
    ys.push_back(p.x - p.y);
    for(int &u : b) {
      xs.push_back(p.x + p.y + u);
      xs.push_back(p.x + p.y - u);
      ys.push_back(p.x - p.y + u);
      ys.push_back(p.x - p.y - u);
    }
  }
  sort(all(xs));
  sort(all(ys));
  xs.erase(unique(all(xs)), xs.end());
  ys.erase(unique(all(ys)), ys.end());
  vint v(n);
  for(int A : xs) {
    for(int B : ys) {
      int X = (A + B) / 2, Y = (A - B) / 2 - x;
      int valid = 1;
      for(int i = 0; i < n; i++) {
        v[i] = abs(a[i].x - X) + abs(a[i].y - Y);
        if(v[i] > x) { valid = 0; break; }
      }
      if(!valid) continue;
      sort(all(v));
      for(int i = 0; i < n; i++) {
        if(v[i] + (i < k ? b[i] : 0) > x) { valid = 0; break; }
      }
      if(valid) return 1;
    }
  }

  return 0;
};
```

SCPC 3회 2차예선 풀이

1번 - Hanoi

- 원판이 N개인 하노이 문제를 푸는데, 다음과 같은 제약 조건이 걸려 있다.
  - 기둥 3개를 A, B, C라고 할 때, 원판을 옮기는 조작은 A→B, B→C, C→A 로만 가능하다.
- N개의 원판이 놓인 상태가 주어질 때, 이 상태가 원판 N개를 모두 A에서 B로 옮기는 최적의 이동 과정에서 나타날 수 있는지 없는지 판단하여라.
- $1 \leq N \leq 1{,}000{,}000$

1번 – Hanoi

- 왠지 모르게 1번부터 어렵습니다….
- 하노이 탑 최적의 이동은 보통 재귀함수로 나타나니, 이 문제 같은 경우에도 재귀함수를 설계해 봅시다.
- 기둥 이동에 걸린 제약 조건 때문에, 아마 케이스가 여러 가지로 나뉠 것입니다.
- 가장 간단하게 나누는 방법은 A → B / A → C 로 나누는 것입니다.
  - n번째 원판을 몇 번 옮겨야 하는지에 따라 나누는 것입니다.

1번 - Hanoi

- 두 가지 경우에 대한 재귀함수는 각각 아래와 같이 생겼습니다.

```
Hanoi(N, A, B):
    Hanoi(N-1, A, C)
    move N from A to B
    Hanoi(N-1, C, B)
```

```
Hanoi(N, A, C):
    Hanoi(N-1, A, C)
    move N from A to B
    Hanoi(N-1, C, A)
    move N from B to C
    Hanoi(N-1, A, C)
```

1번 – Hanoi

- N번 원판의 위치에 따라, 나머지 N−1개의 원판에 해당하는 재귀함수가 어느 것인지를 결정할 수 있습니다.

```
 Hanoi(N, A, B):
A       Hanoi(N-1, A, C)
        move N from A to B
B       Hanoi(N-1, C, B)
```

* N번 원판이 C에 있을 경우 : 불가능

```
Hanoi(N, A, C):
A      Hanoi(N-1, A, C)
       move N from A to B
B      Hanoi(N-1, C, A)
       move N from B to C
C      Hanoi(N-1, A, C)
```

1번 - Hanoi

- 간단한 반복문으로 N번에서 1번 원판까지 내려가면서 조건을 다 확인해 줄 수 있습니다.
- 기둥의 연결 관계가 원형이므로 처리에 주의합시다.

```
void solve() {
  int n;
  string s;
  cin >> n >> s;
  reverse(all(s));

  int st = 0, en = 1;
  for(char c : s) {
    c -= 'A';
    if(c < st) c += 3;
    if(en - st == 1) {
      if(c == st) en++;
      else if(c == en) st += 2;
      else { cout << "NO\n"; return; }
    }
    else {
      if(c == st + 1) swap(st, en);
    }
    st %= 3;
    en %= 3;
    if(st > en) en += 3;
  }

  cout << "YES\n";
}
```

2번 - 오래달리기

- N명의 선수들이 달리기를 합니다.
- 각 선수는 한 바퀴의 길이가 Li인 트랙에서 출발선에서 Di만큼 앞에서 출발하며, 1초당 Si만큼의 속도로 달립니다.
- 모든 선수가 결승선을 동시에 통과하는 최초의 자연수 시간이 언제인지 출력하세요.
- 그런 순간이 존재함이 보장됩니다.
- $2 \leq N \leq 5$

2번 - 오래달리기

- 우리가 구하는 것은 아래 식을 모두 만족하는 가장 작은 자연수 x입니다.

$$s_i x \equiv l_i - d_i \pmod{l_i} \quad (1 \le i \le N)$$

- 최악의 경우에 답이 64bit 정수 범위까지 갈 수 있어서 단순 반복문으로 확인하는 것은 역부족입니다.

2번 - 오래달리기

- 이 경우에 쓸 수 있는 유용한 함수가 하나 있습니다.
  - 다른 곳에도 많이 쓰입니다.
- (a, b)를 주면 ax+by=gcd(a,b)를 만족하는 (x, y)를 반환하는 함수입니다.
- extgcd 함수라고 알려져 있습니다.

```
function<pll(ll, ll)> extgcd = [&](ll a, ll b) {
  if(!b) return pll(1, 0);
  pll p = extgcd(b, a % b);
  return pll(p.y, p.x - a / b * p.y);
};
```

2번 - 오래달리기

- 우선, 각각의 식을 $x \equiv a \pmod{m}$ 꼴로 바꾸면 좋을 것 같습니다.
- 이것은 extgcd를 이용하여 할 수 있습니다.
- $ax \equiv b\ (mod\ m)$ 이라는 식이 있을 때, $ax' + my' = \mathrm{gcd}(a, m)$을 만족하는 x', y'을 생각합시다.
- 무조건 해가 있다고 했으므로, b는 gcd(a,m)의 배수여야 합니다.
- $c = \frac{b}{\mathrm{gcd}(a,m)}$이라고 하면, $a(cx') + m(cy') = b$가 되므로
  $x \equiv cx'(mod\ \frac{m}{\mathrm{gcd}(a,m)})$ 이라는 식으로 바꿀 수 있습니다.

2번 – 오래달리기

- 이제, $x \equiv a \ (mod\ m)$, $x \equiv b \ (mod\ n)$의 두 가지 식이 있을 때 이것을 합치는 것에 대해 생각해 봅시다.
- 첫 번째 식에서부터 $x = a + my$라고 쓸 수 있고, 이를 두 번째 식에 대입해서 생각해 보면 $my \equiv b - a \ (mod\ n)$이 됩니다.
- 앞 슬라이드와 유사한 방식으로 식을 $y \equiv c \ (mod\ \frac{n}{\gcd(n,m)})$ 꼴로 바꿀 수 있습니다.
- 이제 이 식을 다시 $x = a + my$에 넣으면 $x \equiv a + mc \ (mod\ \frac{nm}{\gcd(n,m)})$으로, 두 식이 하나로 합쳐졌습니다!

2번 - 오래달리기

- 전체 코드는 오른쪽과 같습니다.
- 답이 0일 경우에는 한 주기를 더해서 출력해야 함에 유의하세요.

```cpp
void solve() {
  int n;
  cin >> n;

  vll s(n), l(n), d(n);
  for(int i = 0; i < n; i++) {
    cin >> s[i] >> l[i] >> d[i];
    s[i] %= l[i];
    d[i] = (l[i] - d[i]) % l[i];
  }

  function<pll(ll, ll)> extgcd = [&](ll a, ll b) {
    if(!b) return pll(1, 0);
    pll p = extgcd(b, a % b);
    return pll(p.y, p.x - a / b * p.y);
  };

  ll A = 0, N = 1;
  for(int i = 0; i < n; i++) {
    pll p = extgcd(s[i], l[i]);
    ll g = gcd(s[i], l[i]);
    ll M = l[i] / g;
    ll B = (d[i] / g * p.x % M + M) % M;

    p = extgcd(N, M);
    ll C = ((B - A) % M + M) % M;
    g = gcd(N, M);
    ll k = (C / g * p.x % M + M) % (M / g);
    A = (A + N * k) % (N * M / g);
    N *= M / g;
  }

  cout << (A == 0 ? N : A) << '\n';
}
```

3번 – Divisor

- N개의 1 이상 100만 이하 정수들로 이루어진 배열이 주어진다.
- M개의 쿼리가 주어진다.
- 각 쿼리는 b, l, r의 세 정수로 나타나는데, b의 약수들 중 배열의 [l, r] 구간에 있는 수를 하나도 나누어 떨어트리지 못하는 것이 몇 개인지 세는 것이다.
- $1 \leq N, M \leq 100{,}000$

3번 - Divisor

- 1 이상 100만 이하의 숫자들 중 약수의 개수가 가장 많은 수는 720720 입니다. (240개)
- 각 쿼리를 약수의 개수에 비례하는 시간에 적절히 처리할 수 있다면 문제를 풀 수 있습니다.
- 쿼리가 들어올 때 마다 실시간으로 처리하려고 하면 힘듭니다.
- 미리 쿼리를 다 받아 놓고 한 번에 처리한다고 하면 좀 더 쉽습니다.

3번 – Divisor

- 배열을 앞에서부터 쭉 훑어 나간다고 합시다.
- 1부터 100만까지의 각 자연수마다, “그 수의 배수가 등장한 가장 마지막 위치”를 관리한다고 합시다. 이것을 전역 배열 하나로 놓읍시다.
- 수를 하나 볼 때 마다, 위에서 정의한 전역 배열에서 값이 바뀌는 곳은 그 수의 약수에 해당하는 값들입니다. 즉, 많아야 240개입니다.
- 이제 r이 현재 위치인 쿼리에 대해서, b의 약수들 각각에 대해 (마지막으로 배수가 등장한 위치) 〈 l 인지 아닌지를 O(1)에 판별할 수 있습니다.

3번 – Divisor

- 각 수의 약수를 저장하는 배열은 미리 전처리로 구해 둘 수 있습니다.
- 전처리의 시간 및 공간복잡도는 O(100만 * log(100만)) 입니다.

```
for(int i = 1; i < N; i++) {
  for(int j = i; j < N; j += i) dv[j].push_back(i);
}
```

```
void solve() {
  int n, q;
  cin >> n >> q;

  vint a(n + 1);
  for(int i = 1; i <= n; i++) cin >> a[i];

  vector<vpii> qv(n + 1);
  for(int i = 0; i < q; i++) {
    int x, y, z;
    cin >> x >> y >> z;
    qv[z].emplace_back(y, x);
  }

  vint lst(N);
  int ans = 0;
  for(int i = 1; i <= n; i++) {
    for(int d : dv[a[i]]) lst[d] = i;
    for(pii &p : qv[i]) {
      for(int d : dv[p.y]) if(lst[d] < p.x) ans++;
    }
  }
  cout << ans << '\n';
}
```

4번 - 중심

- N개의 정점을 가진 트리가 주어집니다.
- 이 트리에서 k개의 연결된 정점으로 이루어진 부분집합을 하나 골랐다고 합시다.
- 그 집합에 속하지 않는 각 노드에 대해, 집합에 속한 노드 중 하나에 도달하기 위한 최소 거리들 중 최댓값을 생각합시다.
- 집합을 적절히 골라서 최소 거리의 최댓값이 최소가 되도록 했을 때, 그 값을 출력하세요.
- $1 \leq N \leq 100{,}000$

4번 – 중심

- 이 문제 역시 Parametric search를 생각해 볼 수 있습니다.
- 결정 문제, 즉 “모든 노드가 집합에서 최대 거리 X 이하로 떨어져 있도록 k개의 노드들을 고를 수 있는가?” 라는 문제를 푼다고 해 봅시다.
- 트리의 바깥쪽에서부터 노드를 하나씩 떼낸다고 생각합시다.
- 어떤 노드를 떼낼 수 있을 조건은, 그 노드를 포함해서 더 바깥쪽에 있는 모든 노드들이 아직 떼지 않은 노드까지 X 이하의 거리로 도달할 수 있는 것입니다.
- 노드를 최대한 떼내었을 때 마지막에 k개 이하의 노드가 남았는지를 살펴 보면 됩니다.

4번 - 중심

- 조금 더 구체적으로,
  d[x] : 현재까지 x 바깥쪽에서 떼낸 노드들 중 x와 가장 멀리 떨어진 노드까지의 거리 라고 합시다.
- 트리의 리프부터 하나씩 노드를 떼 가면서 그 리프와 연결된 내부 노드에 d[x] 값을 갱신해 줍니다.
- 어떤 리프 노드에 대해 d[x] + (그 리프에 연결된 간선 길이) 〉X라면, 그 리프부터는 더 이상 떼면 안 됩니다.
- 노드를 떼는 과정에서 리프가 새로 생길 수 있는데, 이들은 queue를 통해 처리할 수 있습니다.

4번 - 중심

- 결정 문제를 푸는 함수의 구현을 살펴보는 것이 이해에 도움이 될 듯 합니다.

```cpp
auto f = [&](ll x) {
  queue<int> q;
  vll d(n + 1);
  vint deg(n + 1), vis(n + 1);
  for(int i = 1; i <= n; i++) {
    deg[i] = e[i].size();
    if(deg[i] == 1) q.push(i);
  }

  int cnt = n;
  while(!q.empty()) {
    int c = q.front();
    q.pop();
    for(pii &p : e[c]) {
      if(vis[p.x]) continue;
      if(d[c] + p.y > x) break;
      d[p.x] = max(d[p.x], d[c] + p.y);
      cnt--;
      vis[c] = 1;
      if(--deg[p.x] == 1) q.push(p.x);
    }
  }
  return (cnt <= k);
};
```

5번 - 자석

- N개의 막대기가 주어집니다.
- 각 막대기는 [s, e] 구간에 놓여 있으며, w 이상의 전류를 공급하면 자석이 됩니다.
- 자석이 아닌 막대기는 자신과 구간이 겹치는 (교집합이 공집합이 아닌) 자석 막대기가 하나라도 있으면 거기에 붙습니다.
- 모든 자석이 아닌 막대기가 적어도 하나의 자석에 붙도록 할 수 있는 최소의 전류 요구량을 구하세요.
- $1 \leq N \leq 100{,}000$

5번 - 자석

- 기본적인 관찰이 하나 있습니다.
- 두 막대기 [a, b]와 [c, d]에 대해 a ≤ c ≤ d ≤ b라면, [a, b]가 자석에 붙었는지에 대한 여부는 신경쓰지 않아도 됩니다.
  - 단, [a, b]를 자석으로 만드는 경우는 여전히 고려해야 합니다.
- 어차피 [c, d]가 자석에 붙는다면 [a, b]는 무조건 그 자석에 붙일 수 있기 때문입니다.
- 이런 식으로 자신이 완전히 포함하는 구간이 존재하는 막대기들을 다 없애고 나면, 남은 막대기들을 시작점이 증가하는 순으로 정렬했을 때 끝점 역시 증가하는 순서가 됩니다.

5번 - 자석

- 이런 전처리를 왜 했냐 하면, 이제 DP를 할 수 있기 때문입니다!
- 앞에서 말한 성질 (시작점 증가 순으로 정렬하면, 끝점 역시 증가 순으로 정렬된다) 때문에, 임의의 막대기 하나를 자석으로 만들었을 때 그 자석이 커버하는 막대기들의 집합이 항상 구간으로 나타납니다!
- 이제 이를 이용하여 DP 식을 세워줄 수 있습니다.
- D[i] : 1~i번째 막대기까지 커버하는 최소 비용.
- 각 막대기에 대해 자석으로 만들면 [s, e] 구간을 커버한다고 할때, D[e]에 (D[s−1], D[s], …, D[e−1] 중 최솟값) + (그 막대기의 전류량)을 갱신해 줄 수 있습니다.

5번 - 자석

- 구현은 간단한 세그먼트 트리를 사용하여 O(NlogN)만에 해 줄 수 있습니다.

```cpp
void solve() {
  int n;
  cin >> n;

  struct Bar { int s, e, c; };
  vector<Bar> v(n);
  for(Bar &b : v) cin >> b.s >> b.e >> b.c;
  sort(all(v), [](const Bar &a, const Bar &b) {
    return pii(a.s, -a.e) > pii(b.s, -b.e);
  });

  vint sp, ep;
  for(Bar &b : v) {
    if(!ep.empty() && ep.back() <= b.e) continue;
    sp.push_back(b.s);
    ep.push_back(b.e);
  }
  reverse(all(sp));
  reverse(all(ep));
```

```cpp
  int m = sp.size();
  vector<vpii> w(m + 1);
  for(Bar &b : v) {
    int s = int(lower_bound(all(ep), b.s) - ep.begin()) + 1;
    int e = int(upper_bound(all(sp), b.e) - sp.begin());
    w[e].emplace_back(s, b.c);
  }

  vll d(m + 1);
  Seg::i(m);
  Seg::u(0, 0);
  for(int i = 1; i <= m; i++) {
    d[i] = INF;
    for(pii &p : w[i]) d[i] = min(d[i], Seg::g(p.x - 1, i - 1) + p.y);
    Seg::u(i, d[i]);
  }
  cout << d[m] << '\n';
}
```