競プロ勉強会07

elzup

概要

- bitDP

- ICPC に向けて

bitDP

- n 個の要素の組み合わせを 1つのパラメーター(bit 列) で表現する
- 2^n
- dp[bits]

- dp[0b000001]　　oooooo
  dp[0b000010]　　ooooox
  dp[0b000011]　　oooxx
  dp[0b000100]　　oooxoo

- n枚からいくつかのカードを選んでいる状態

- n種類のチケットからいくつかのカードを使用済みの状態

問題

- AOJ Traveling Stagecoach
- http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1138&lang=jp

Sample1

3チケット, 4都市, 3エッジ

from1to4

3 4 3 1 4

3 1 2
1 2 10
2 3 30
3 4 20
2 4 4 2 1

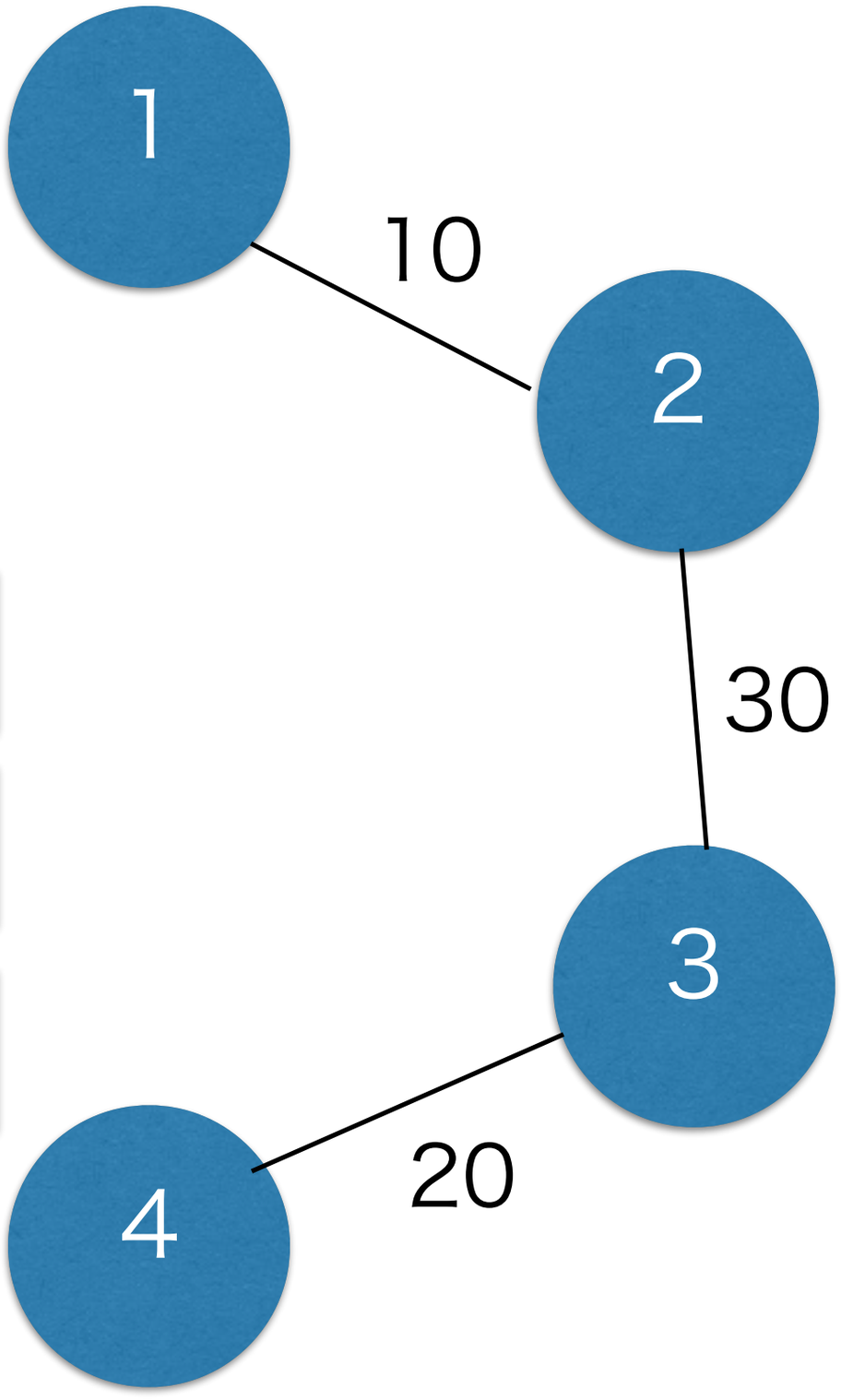
チケットリスト

エッジリスト

x3

x1

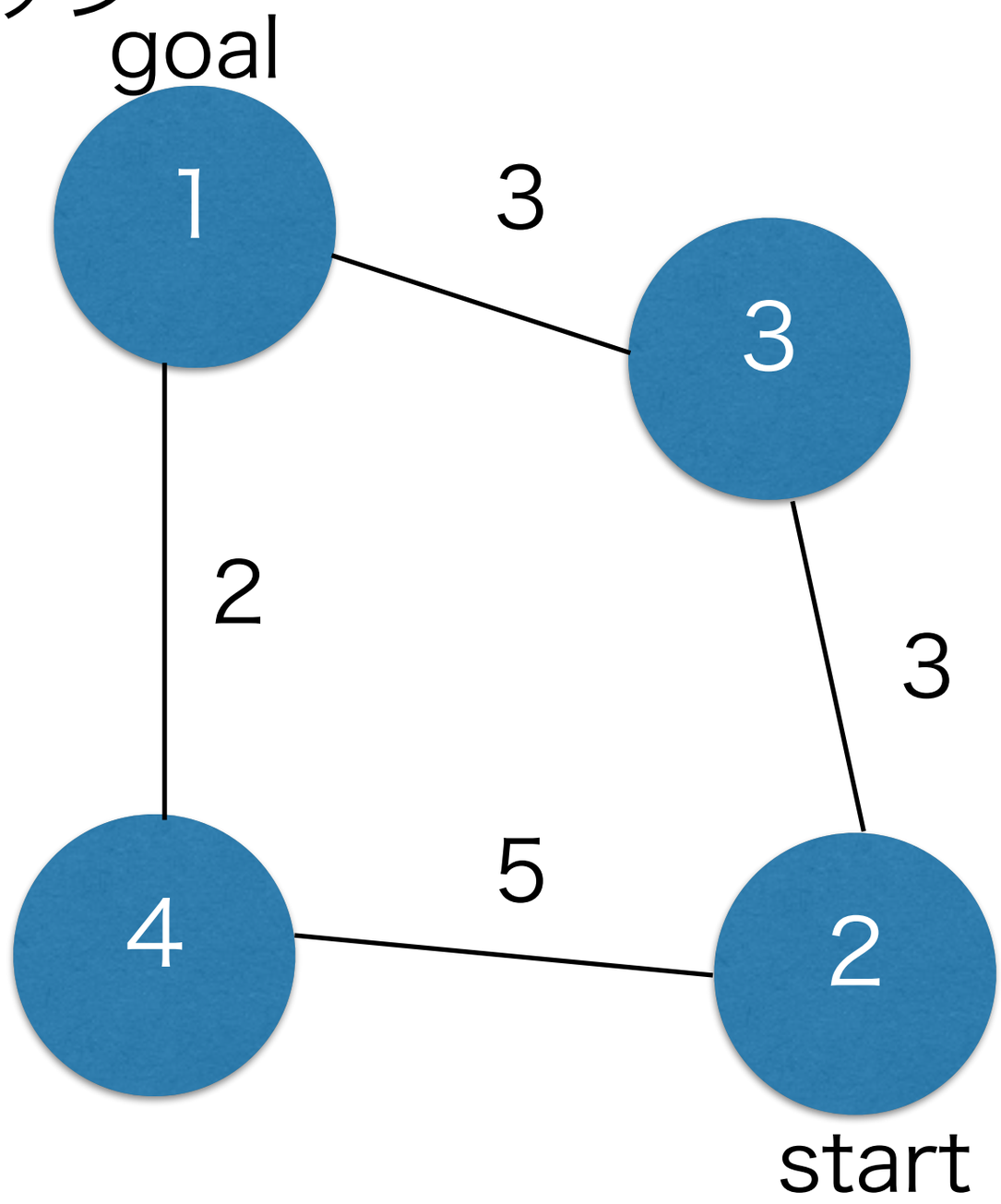
x2



Sample2

2チケット, 4都市, 4エッジ

from2to1



考え方

- 単一始点(スタート地点が決まっている)
  + 負コストのエッジ無し
  + m <= 30

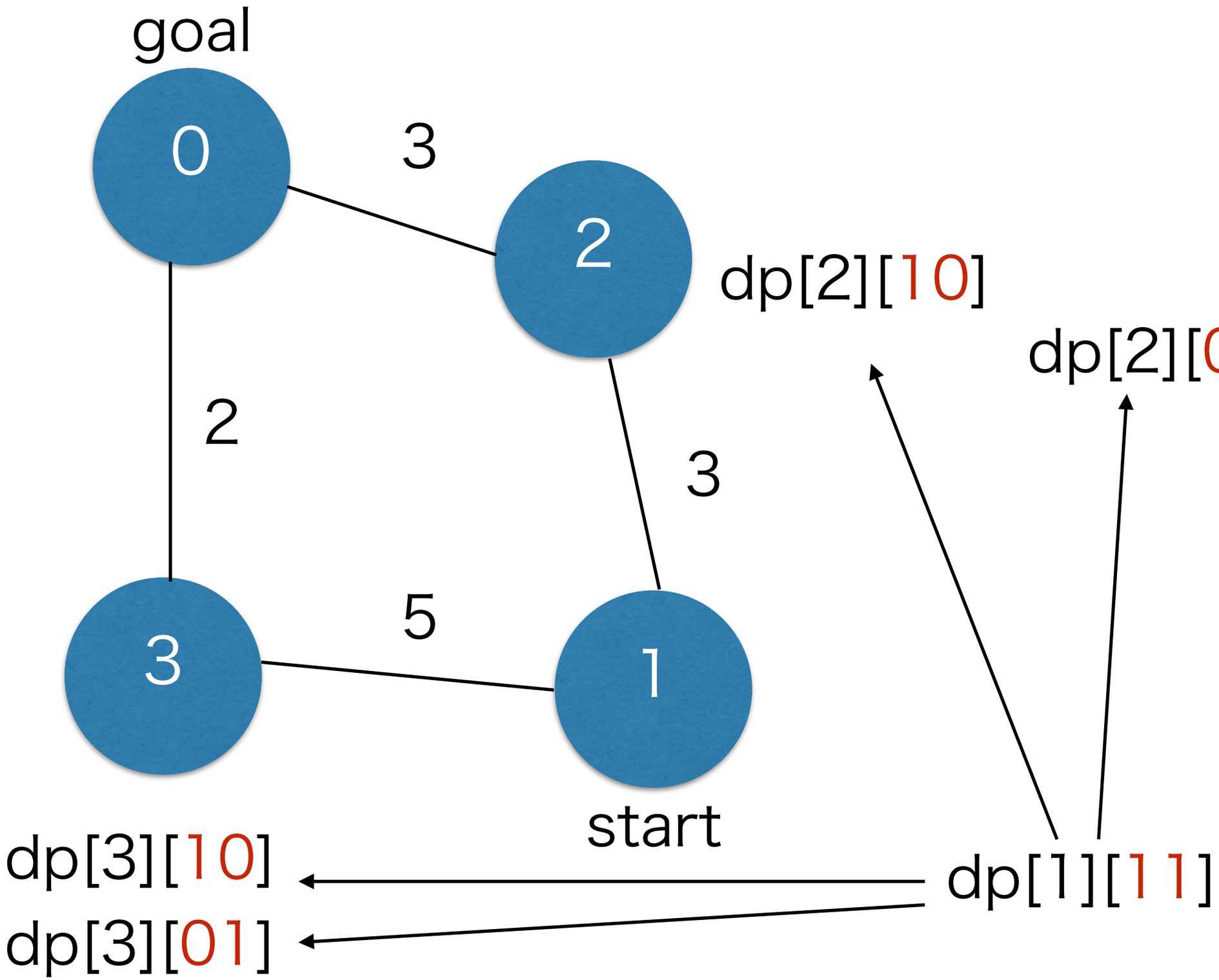
  - -> ダイクストラ O(e + v log(v))

- 各ノードに「各チケットの有無(消費済みかどうか)」という状態が存在する
  + n <= 8

  - -> bitDP O(2^n)

- (e = 30 * (2^8)) ^ 2 = 60,000,000

dp[頂点][チケットbit列] => コスト



状態を表す構造

* 頂点番号
* チケットbit列
* コスト

実装流れ

- 入力
- ダイクストラ
- 出力

ダイクストラ

- pq(PriorityQueue)初期化 + 初期状態追加
  pq が空になるまで {
    - pq.poll()
      繋がるエッジ全て pq に追加

- }

bitダイクストラ

- pq(PriorityQueue)初期化 + 初期状態追加
  pq が空になるまで {
  - pq.poll()
    繋がるエッジについて {
    - 残っているチケットそれぞれについて {
      - 使った場合の次の状態を pq に追加
    - }
  - }
- }

実装

```
· static class P implements Comparable<P> {
      int v, tickets;
      double cost;
     P(int v, double cost, int tickets) {
          this.v = v;
          this.cost = cost;
          this.tickets = tickets;
      }

·     @Override public int compareTo(P o) {
        return (int) (this.cost - o.cost);        // コストで比較
      }
  }
```

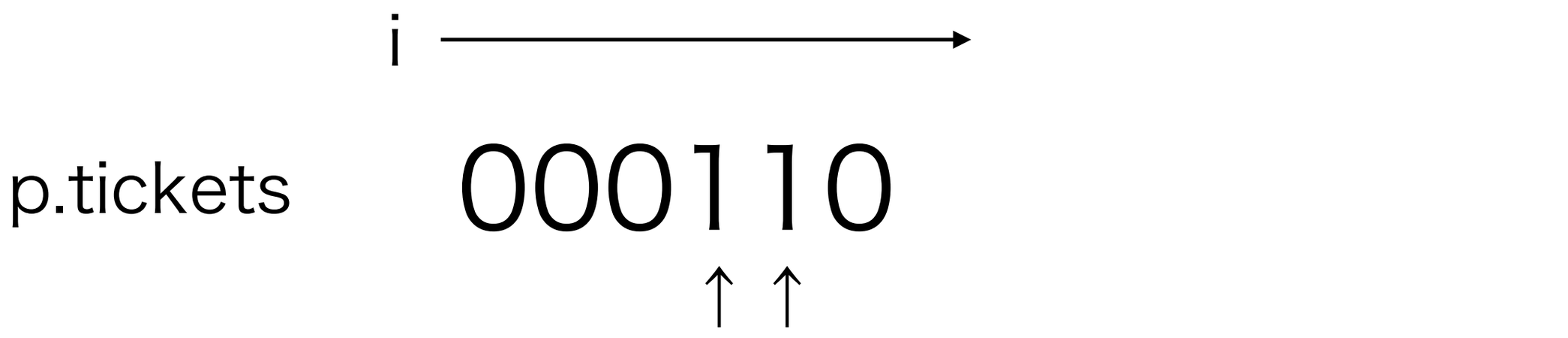
実装

```
・ static class Edge {
        int to;
        double cost;
        Edge(int to, double cost) {
            this.to = to;
            this.cost = cost;
        }
}
```

実装

- List<Integer> is = p.ticket_index();
  for (Edge e : edges[p.v]) {
      for (int i = 0; i < n; i++) {
          if (((p.tickets >> i) & 1) == 0) { continue; }
          // 残っているチケット [i] を使った場合
          int at = p.tickets - (1 << i);
          // コストが少ないならば更新して pq を追加
          if (dp[e.to][at] > dp[p.v][p.tickets] + e.cost / tickets[i]) {
              dp[e.to][at] = dp[p.v][p.tickets] + e.cost / tickets[i];
              que.add(new P(e.to, dp[p.v][p.tickets] + e.cost / tickets[i], at));
          }
      }
  }

- if (((p.tickets >> i) & 1) == 0) { continue; }



- // 消費した後のチケット組み合わせ
- int at = p.tickets - (1 << i);

000110 - (1 << 2)

000110 - 000100

000010

- dp[e.to][at]
- dp[p.v][p.tickets] + e.cost / tickets[i]

- dp[e.to][at]
- dp[p.v][p.tickets]

出力

- double min = Double.MAX_VALUE;

```
// goal 頂点で各残り枚数の中で最小コスト
for (int i = 0; i < ALL_B; i++) {
    min = Math.min(dp[g][i], min);
}
```

ACM-ICPC

- A　全探索, 総当り, 考えたら負け

- B　シミュレーション, パース問題
- C　豆にデバッグ, 着実に実装
　詰まったらドツボにハマる前に最初から実装

- D
- E　たぶん同難易度
　DP, グラフ問題, 幾何問題(数学必須)
- F

……

- G　怖い

ACM-ICPC

↓最初に問題のアルファベット7つを1枚の紙に書く

問題のラベル付　　担当中　　状態

- A　　柴原　　考察 -> 実装 -> デバッグ
- B　幾何　　高橋　　考察 -> 実装
- C　文字列操作　　千葉　　考察
- D　最小全域木
- E　　無理
- F
- G

勉強会以外の知識

- セグメントツリー
- 逆元
- 最長増加部分列, 最大部分和
- いもす法
- (a*)
- フロー

ちょっとしたアルゴリズム

- 素数列挙, 素数判定
- 約数, 最大公約数, 最小公倍数
- xor