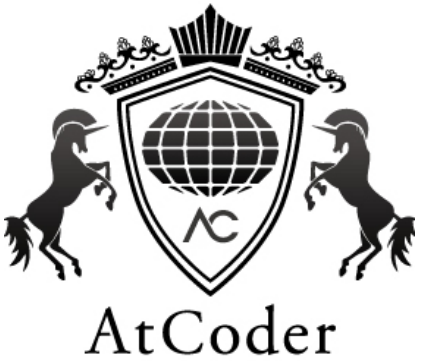
AtCoder Beginner Contest 010
解説



AtCoder株式会社 代表取締役
高橋 直大

競技プログラミングを始める前に

AtCoder

- 競技プログラミングをやったことがない人へ
  - まずはこっちのスライドを見よう！
  - http://www.slideshare.net/chokudai/abc004

A問題　高橋くんの研修

1. 問題概要
2. アルゴリズム

A問題　問題概要

AtCoder

- 文字列 S がA, Bが与えられる
- 文字数が多い方を出力せよ

- 制約
- $1 \leqq |A|, |B| \leqq 10$
  - |S| は文字列Sの長さを表します！
- $|A| \neq |B|$

A問題　アルゴリズム

AtCoder

- 基本的なプログラムの流れ
  - 標準入力から、必要な入力を受け取る
    - 今回の場合は、A,B という2つの文字列
  - 問題で与えられた処理を行う
    - 今回は、長い方を調べる
  - 標準出力へ、答えを出力する
    - 長い方を出力する

A問題　アルゴリズム

AtCoder

- 入力
  - 2つの文字列を、標準入力から受け取る
    - Cであれば、scanf(“%s”, &A);　など
    - C++であれば、cin >> A;
    - 入力の受け取り方は、下記の練習問題に記載があります。
      - http://practice.contest.atcoder.jp/tasks/practice_1
    - 个の例では一つ分しか書いてないよ！

A問題　アルゴリズム

AtCoder

- 今回の問題は、文字列の長さを比較するだけ
  - 文字列の長さを取得する関数は、どの言語にも基本的に入っている
    - もし見つからなかったら、終端文字（¥0）までループを回して文字数を数えよう！
- 不等号の向きを間違えないようにしよう！

A問題　アルゴリズム

AtCoder

- 出力
  - 求めた答えを、標準出力より出力する。
  - 言語によって違います。
    - 答えをSに入れたとする
    - printf(“%s¥n”, S); (C)
    - cout << S << endl; (C++)
    - System.out.println(S); (Java)
    - 各言語の標準出力は、下記の練習問題に記載があります。
      - http://practice.contest.atcoder.jp/tasks/practice_1

B問題　高橋くんの集計

1. 問題概要
2. アルゴリズム

B問題　問題概要

AtCoder

- N個の整数が与えられる
- 0以外の整数の平均値を出力しなさい

B問題　アルゴリズム

AtCoder

- 入力
  - 整数Nを受け取る
  - 数列Aの数字をN個受け取る
    - 受け取り方は複数いくつかある
      - 1行纏めて受け取って、スペースでsplitする
      - 1つずつ受け取る
    - 言語によって受け取りやすい書き方が違う！
    - 詳しくはpracticeで確認しよう！
      - http://practice.contest.atcoder.jp/tasks/practice_1

B問題　アルゴリズム

AtCoder

- 処理
  - やるべきことは2つ！
    - バグの総数を調べる
      - こちらはforループを回して足し算をしていくだけ
    - バグが存在するソフトウェアの個数を調べる
      - こちらは、ifなどで判定してカウントする
  - この2つが出来れば、あとは平均を取る！
    - 切り上げることに注意！
    - 切り上げる方法も多数
      - あまりが0でなかった場合、答えに1を足す
      - 一つの計算式で表すことも可能
        - » (sum + count – 1)/ count

B問題　アルゴリズム

AtCoder

- 出力
  - A問題と同じく、答えを出力するだけ
  - Print(ret)みたいな感じ

AtCoder

C問題　高橋くんのバグ探し

1. 問題概要
2. アルゴリズム

C問題　問題概要

AtCoder

- K択問題がN個あります。
- 全ての選択肢に内部的に整数が割り当てられており、選ばれた選択肢のXOR(排他的論理和)の値を利用するが、この値が0になるとバグが発生する。
- 選択肢に割り当てられた整数が与えられるので、バグが発生することがあり得るかを判定しなさい。

- 制約
  - $1 \leqq N \leqq 5$
  - $1 \leqq K \leqq 5$
  - $0 \leqq T_{i,j} \leqq 127$

C問題　アルゴリズム

AtCoder

- どう解くか？
  - 質問の数は5つ、選択肢も5つ
    - 非常に数が少ない！
  - つまり、全探索することが可能！
  - どうやって全探索をしよう？

C問題　アルゴリズム

AtCoder

- 1つの質問だったら？
  - Forループで K回回せば、全通り試せる
- 2つの質問だったら？
  - Forループを2重で組み合わせれば、全通り試せる
- 3つの質問だったら？
  - Forループを3重で組み合わせれば、全通り試せる

- ・・・・こんなのはやってられない！

C問題　アルゴリズム

AtCoder

- 1つの質問だったら？
  - Forループで K回回せば、全通り試せる
- 2つの質問だったら？
  - Forループを2重で組み合わせれば、全通り試せる
- 3つの質問だったら？
  - Forループを3重で組み合わせれば、全通り試せる

- ・・・・こんなのはやってられない！

C問題　アルゴリズム

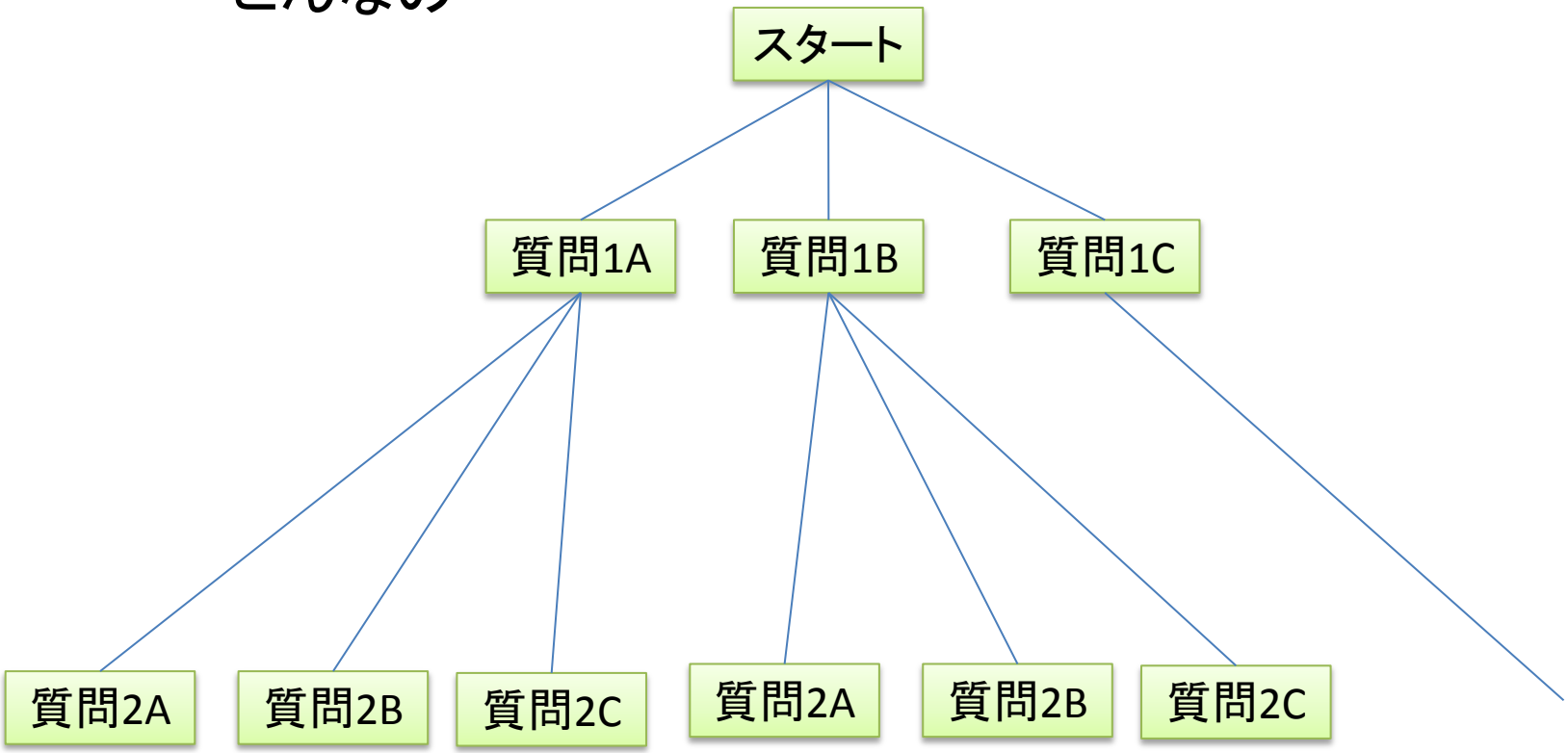
AtCoder

- ではどう解くか？
- 深さ優先探索を使おう！
  - 深さ優先探索って？

C問題　アルゴリズム

AtCoder

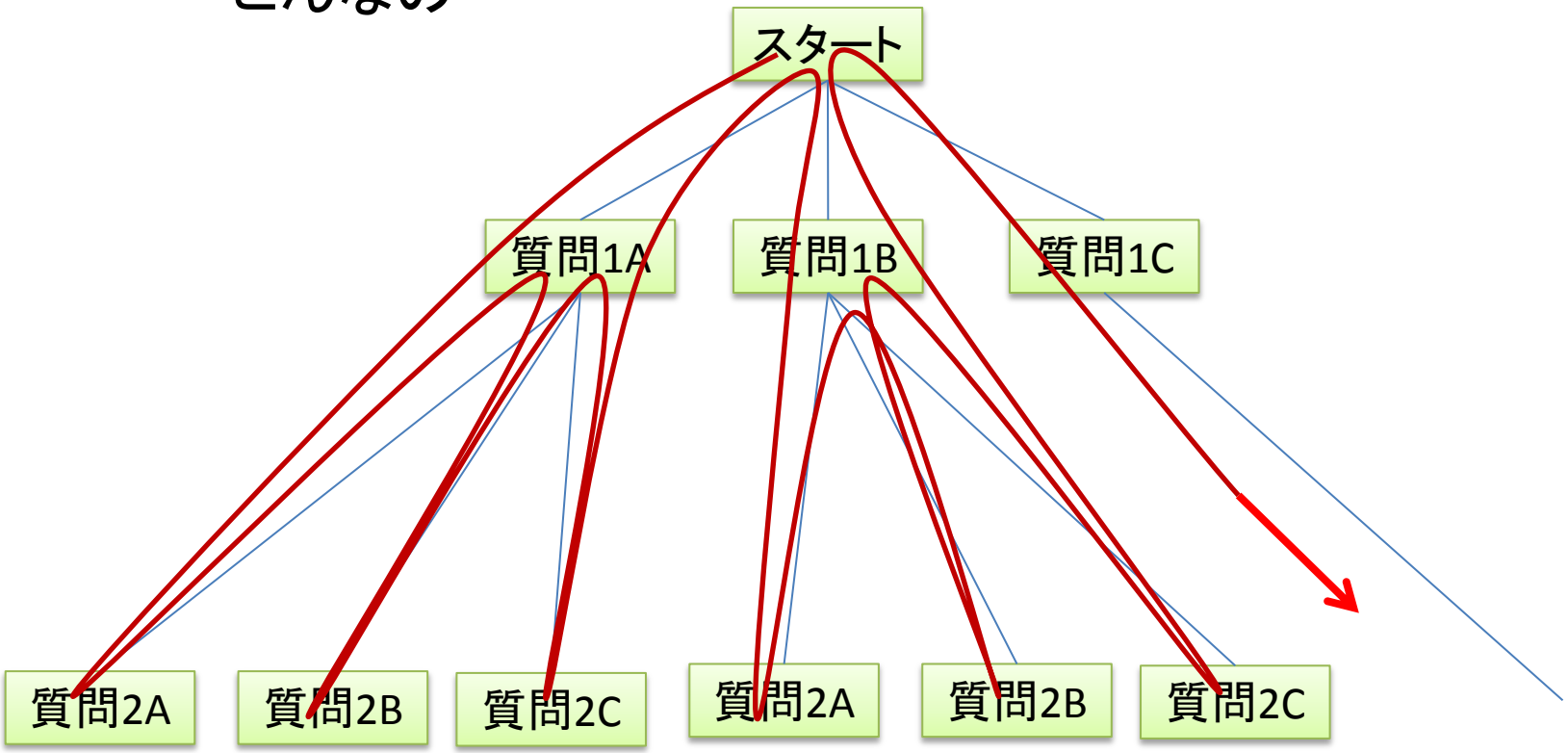
- 深さ優先探索
  - こんなの



C問題　アルゴリズム

AtCoder

- 深さ優先探索
  - こんなの



C問題　アルゴリズム

AtCoder

- 要するにどういうこと？
  - 質問や選択肢の数に合わせて、潜って行くように探索していく！
  - こうすることで、全ての質問の組み合わせに対して調べることが出来る！
    - でもどう組むの？

C問題　アルゴリズム

AtCoder

- 再帰関数を使おう！
  - 自分の関数の内部で、自分を呼び出す！
  - 例えばこんな感じ
    - 最初はdfs(0, 0);って感じで呼び出す

```
bool dfs(int numQ, int value){
    if(numQ == N) return (value == 0);
    for(int i = 0; i < K; i++){
        if(dfs(numQ+1, value^T[numQ,i])) return true;
    }
    return false;
}
```

今の質問数、値から
今どこにいるかを格納

質問がもうなければ
0になっているかを調べる

dfs関数の中から、
dfsをもう一度呼び出す！

探索した結果、0になる組
み合わせが無ければfalse

C問題　アルゴリズム

AtCoder

- 計算時間は大丈夫？
  - N回の質問で、K回の選択肢
  - 全ての組み合わせはK^N個
  - N,Kともに5以下なので、この組み合わせの数は大したことがない
    - 最も深いループが1億程度回らなければ、制限時間に間に合うことが多い

C問題　アルゴリズム

AtCoder

- 他の方法は？
  - 深さ優先探索で、もう少し工夫する方法
    - 「今までの質問数」「ここまでの値」が同じものを1度調べていれば、もう調べる必要がない
      - こうすることにより、計算量をK^NからNKまで減らせる
  - 幅優先探索などでももちろん良い
    - 一筆書きのように調べるのではなく、１手１手進めていく方法
    - 詳しくは他の幅優先探索を使う問題を見てね！

D問題　高橋くんの苦悩

1. 問題概要
2. アルゴリズム

C問題　問題概要

AtCoder

- 幅Wの領域に、幅A[i]、価値B[i]の、N枚のスクショを選択して貼りつける
- 貼りつけられるスクショは、
  - 合計枚数がK枚以下
  - 合計の幅がW以下
- でなければならない。価値の和の最大値を求めなさい。

- 制約
  - 1≦N≦50
  - 1≦W≦10000
  - 1≦A[i],B[i]≦1000

D問題　アルゴリズム

AtCoder

- どういう問題？
- 非常に有名な「ナップサック問題」の亜種です
  - 商品がN個、重さWまで持てる。
  - 各商品の重さA[i]と、価値B[i]が与えられる
  - 価値を最大化しなさい
- 今回は、それに加えて、「商品の個数」が制約に加わっている

D問題　アルゴリズム

AtCoder

- 深さ優先探索で解ける？
  - 今回は、C問題に合わせると、
    - N個の商品について、「選ぶ」「選ばない」を選ぶ必要があります。
    - XORではなく価値の和を最大化したいです。
    - いくつかの制約があります
  - みたいな問題になるため、似たような深さ優先探索で、答えを求めることは可能です。
    - ただし、制限時間内に終わるわけではない！
  - 計算量を考えよう！
    - 選択肢は「選ぶ」「選ばない」の2つ
    - 商品はN個、つまり50個以下
      - 2の50乗は10の15乗程度だが、これは組み合わせが多過ぎる

D問題　アルゴリズム

AtCoder

- では、どうするか？
  - 全部調べるのはなんとなくもったいない
  - 計算を上手く纏められると嬉しい。
  - そこで出てくるのが「動的計画法！」

D問題　アルゴリズム

AtCoder

- 動的計画法って？
  - ある状態に対して、「すでに訪れたか」や「最大値」「最小値」を計算することで、無駄に計算をしないこと！
  - 状態って何？
    - 「今持ってる価値」とか「何個目まで調べた」とか「今選んだスクショの合計の幅」とか、そのような、探索をする上で必要な情報！

D問題　アルゴリズム

AtCoder

- さっきと同じ感じで、深さ優先探索のイメージをしてみよう！
  - 自分が書くならこんな感じ
    - Int dfs(int useW, int useNum, int now) { 以下略
      - useWは、「今まで選んだスクショの合計幅」
      - useNumは、「今まで選んだスクショの数」
      - Nowは、「今選ぶかどうか調べたいスクショが何枚目か」
        - » これに対して、dfs(0,0,0)みたいな感じで呼び出したい
        - » で、最大値が返ってくると嬉しい
  - これで動いたら嬉しい！
    - でも実際は間に合わない

D問題　アルゴリズム

AtCoder

- どうすれば良いか？
  - 解法１：メモ化再帰を使う
    - メモ化再帰って？
      - 深さ優先探索で書いた時に、同じ計算を省略する！
  - 解法２：動的計画法を使う
    - » っていっても、メモ化再帰も動的計画法の一種ですが
    - 「幅」「使った枚数」「何枚目まで見たか」に対して、価値の最大値を計算する
      - 再帰ではなく、ボトムアップで計算する！

D問題　アルゴリズム

AtCoder

- 解法１：メモ化再帰を使う
  - どう計算を省略するの？
    - 要するに、同じ計算をしなければ良い
    - 結果メモ用配列dp[W][N][N]を用意する
    - Int dfs(int useW, int useNum, int now)とか書いたとする
      - もしdp[useW][useNum][now]に既に数が入っていたら、その数を返す
      - 入っていなければ普通に計算をし、returnするときに、dp[useW][useNum][now]に、returnする数を入れる
        - » こうすることにより、2回目から計算しなくて良くなる！
  - 計算量は？
    - 同じ計算をしなくなったので、useW, useNum, nowの組み合わせが何通りあるか調べれば良い
      - これはそれぞれW,N,Nであり、それに対して2択の選択肢
      - 5000万くらい！よってギリギリ間に合う！

D問題　アルゴリズム

AtCoder

- 解法2：動的計画法を使う！
  - dp[i][j][k]に、k番目まで調べた時の、幅合計i、使用枚数jの最大値を入れるとする
  - これは、dp[i][j][k-1]か、dp[i-A[k-1]][j-1][k-1]+B[k-1]のどちらかしかありえない！
    - なぜか？
    - K番目まで調べて、幅i、枚数jが最大になる、ということは、K番目のスクショを選んで最大になったか、選ばず最大になったか、どちらかしかない
      - K番目を使わない場合は、dp[i][j][k-1]
      - K番目を使う場合は、使う前がdp[i-A[k-1]][j-1][k-1]であり、そこにB[k-1]を足すことで、求めることが出来る！！

D問題　アルゴリズム

AtCoder

- 解法3：動的計画法を使う！工夫する版！
  - 解法2のkは実は省略できる
    - K番目を使わない場合→そのまま
    - K番目を使う場合→dp[i][j] = dp[i-A[k-1]][j-1]
  - みたいな感じで、同じメモリ上で回すことが出来る
  - ただし、この場合、「同じスクショを2回以上使わない工夫」をする必要がある。
    - 具体的には、ループの向きを逆にすると良い！
    - 逆にすれば、「直前に足したのをまた足してしまう」ということがなくなる
  - 詳しくは生放送で実装の時に説明するよ！！