Vol.2022-SE-210 No.15
2022/3/11

ARTを用いたアルゴリズミックデバッグの効率化手法

冨久 剛志[1,a)]　大久保 弘崇[,b)]　粕谷 英人[,c)]　山本 晋一郎[,d)]

概要：アルゴリズミックデバッグはプログラムのトレース情報を用いて計算過程の正誤をユーザに質問し，ユーザの回答からバグを発見する手法である．Haskell ではアルゴリズミックデバッガとして hat-detect が提供されているが，バグを発見する操作において，質問順の問題から質問数が多くなってしまうことや冗長な質問が発生する課題をもつ．本研究ではこの課題に対し，プログラムに定義されている関数の特徴から優先するべき質問の選定や，デバッガに質問と回答結果を記録することにより解決する．これらの提案手法を hat-detect と比較し，その有効性を確認した．

1. はじめに

1.1 背景

デバッグの手法の一つにアルゴリズミックデバッグがある．アルゴリズミックデバッグは対象となるプログラムのトレースから関数の呼び出し関係の計算木を作成し，計算木に記録された関数呼び出しとその計算結果の正誤をプログラマに尋ねることでバグを発見する．プログラマは発見されたバグに対して修正を行うことでデバッグする．

アルゴリズミックデバッグではプログラマの回答と計算木を使いプログラムのバグを発見するが，バグを発見するために必要な回答数は計算木の規模に左右される．デバッガが計算木を探索するとき，計算木の各頂点ごとに質問を行うため，計算木が巨大になるとデバッガによる質問数が増える．これはプログラマの回答数が増えることであり，これはプログラマの負担になる．

Haskell で提供されているアルゴリズミックデバッガに hat-detect[1] がある．hat-detect は先に述べた課題を持つ．また hat-detect における計算木の探索は実行の評価順に行われ，その後プログラマの回答によって分岐する．探索は計算木の根から始まり，リーフへと向かって行われ，デバッガによってバグが特定されたとき探索を終了する．このとき hat-detect では質問数が多くなる問題が発生する．その原因として，定義された関数の構造を考慮しない質問順序や同じ質問が 2 度される場合があることが挙げられる．

[1] 愛知県立大学大学院情報科学研究科
Graduate School of Information Science and Technology, Aichi Prefectural University
[a)] tomihisa@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp

Haskell は関数型プログラミング言語であり，しばしば再帰を用いて関数が定義される．hat-detect では再帰関数を使ったプログラムをアルゴリズミックデバッグする場合，計算木を評価順に探索するため再帰部から探索が始まり，基底部へと向かう．このとき再帰部が誤っている場合，hat-detect の仕様では基底部まで質問する必要がある．しかし hat-detect は計算木の規模による質問の増減という問題をもつため，再帰処理が多くなると必然的に質問も多くなる．また基底部が誤りの場合も同様に質問が多くなる．再帰関数における再帰部と基底部の正誤を考えると，基底部が正しいかどうかを調べることが重要だろう．しかし hat-detect では基底部は最後に確認するようになっている．

hat-detect では質問の回答を記録しないため，全く同じ質問が発生する場合がある．hat-detect は計算木から質問を行うため，頂点の関数ラベルやその引数に関して考慮しない．そのため，計算式とその結果が正しいかどうかという質問が重複して現れる場合がある．Haskell は参照透過性を持つため，ある関数のある引数に対する結果は常に同じ結果となる．そのため，ある関数のある引数における計算結果の正誤は別の関数呼び出しであっても同じになる．これは冗長であると考えられる．

本稿では，質問の順番の改善と冗長な質問の削除を目的とする．本稿で実装されたデバッガは加藤らによって作成された Augmented Redex Trail（以下 ART）を用いる [2]．ART はプログラムの計算過程が記録されたグラフである．

2. アルゴリズミックデバッグ

Shapiro らは，Prolog プログラムを対象として，誤り箇所を特定する手法であるアルゴリズミックデバッグを提案した [3]．この手法は論理言語 Prolog にプログラムの誤り

```
1  -- merge test
2  mergesort :: [Int] -> [Int]
3  mergesort [] = []
4  mergesort [x] = [x]
5  mergesort xss =
6      let xs = take l xss ;
7          ys = drop l xss ;
8          l = div (length xss) 2 ;
9      in merge (mergesort xs) (mergesort ys)
10
11 merge :: [Int] -> [Int] -> [Int]
12 merge [] xs = []
13 merge xs [] = xs
14 merge xss@(x:xs) yss@(y:ys)
15         = if (x < y)
16                 then (x:merge xs yss)
17                 else (y:merge xss ys)
18
19 main :: IO ()
20 main = do
21         print (mergesort [3,1,2])
```

図 1 merge sort の実装

を特定する手法である．Haskell では hat-detect や Hoed といったアルゴリズミックデバッガが提供されている．

アルゴリズミックデバッグはプログラムのトレース結果から，計算木を作成する．計算木の頂点は計算式とその計算式の結果を持ち，辺は計算式の呼び出し関係を示す．その後デバッガは計算木の探索を始め，計算木の各頂点の正誤をプログラマに尋ねる．プログラマの回答が十分に集まったとき探索を終了し，誤りを持つと判断された頂点の計算式を提示する．

2.1 hat-detect

Hat-detect は Haskell プログラムをトレースするためのツール Hat[4] の機能の一部である．Hat は Haskell プログラムの呼び出し関係や実行過程を記録できるようにソースコードを変換する．変換したプログラムを実行すると元のプログラムと同じ処理を行なった後に，実行過程が保持された hat ファイルが作成される．hat ファイルには Augmented Redex Trail（ART）と呼ばれるグラフ構造によって計算が記憶される．

hat-detect によるアルゴリズミックデバッグの例を示す．図 1 のプログラムがあるとする．

これは誤りを持つマージソートを実装したプログラムである．プログラムには 3 つの関数，`mergesort`と `merge`，`main`が定義されている．`mergesort`は長さ 2 以上のリストを二つに等分し，それぞれを再帰的に `mergesort`した結果を `merge`に渡す．`merge`は整列された二つのリストを受け取り，マージする．`main`は，`mergesort`の引数にリスト `[3,1,2]`渡した計算結果を出力する．

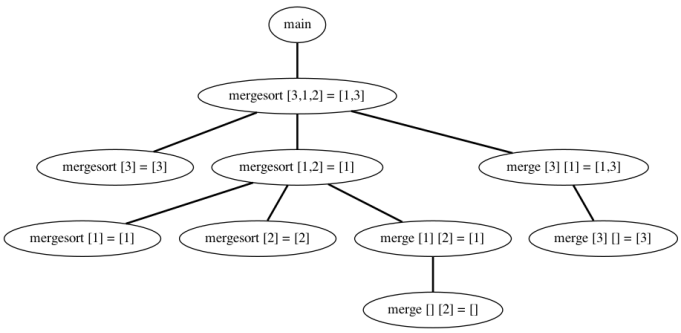

図 2 ソースコード 1 を実行したときの mergesort の計算木

```
1  1  mergesort [3,1,2] = [1,3]
2  hat-detect> n
3  2  mergesort [3] = [3]
4  hat-detect> y
5  3  mergesort [1,2] = [1]
6  hat-detect> n
7  4  mergesort [1] = [1]
8  hat-detect> y
9  5  mergesort [2] = [2]
10 hat-detect> y
11 6  merge [1] [2] = [1]
12 hat-detect> n
13 7  merge [] [2] = []
14 hat-detect> n
15 bug found at ac8
16     merge [] [2] = []
17 Done.
```

図 3 ソースコード 2 を hat-detect でアルゴリズミックデバッグした例

このプログラムは 12 行目 `merge [] xs = []`が間違っており，正しい記述は `merge [] xs = xs`となる．そのため，このプログラムを実行すると，標準出力に `[1,3]`と出力される．ただし，期待される出力は `[1,2,3]`である．

このプログラムの実行結果に対する計算木は図 2 となる．この計算木を元にアルゴリズミックデバッグを行うと図 3 のやりとりが行われる．これはデバッガからの質問とプログラマの回答を示している．1 行目，3 行目，5 行目，... では，デバッガが計算式とその結果について表示する．それに対し 2 行目，4 行目，6 行目，... では，プログラマが `y`または `n`で回答を行う．表示された計算式の結果が正しければ，プログラマは Yes として `y`と入力し，誤りなら No として `n`を入力する．

デバッガはプログラマの回答から計算木を巡回し，最終的に誤りだと判断した計算式を提示する．このとき hat-detect

における回答の分岐と終了判定は次の規則で行う．

(1) デバッガは式に対する回答が Yes のとき，質問している頂点の親が質問している頂点以外の子を持つ場合はその子を質問する．
(2) 親が質問している頂点以外の子を持たないとき，質問している頂点以外の子に対する回答が Yes の場合，質問している頂点の親を誤りとして提示する．
(3) 回答が No のとき，質問している頂点が子を持つ場合は質問する．
(4) 質問している頂点が子を持たない場合は質問している頂点を誤りとして報告する．

規則に従いながら図 2 を探索する場合は次のようになる．図 2 の main から探索を始め，`mergesort [3,1,2] = [1,3]`の頂点へ移動し，質問を行う．質問の回答が Noであり，子を持つため 3 により次の頂点に移る．次に子の `mergesort [3] = [3]`の質問を行う．回答が Yesであり，親が別の子を持つので 1 により子に移動し，質問を行う．次に別の子である `mergesort [1,2] = [1]`を質問する．回答が Noであり，子を持つため 3 によりその子へと移動する．`mergesort [1] = [1]`を質問する．回答は Yesであり，別の子を持つため 1 によりその頂点を質問する．`mergesort [2] = [2]`も同様である．そして `merge [1] [2] = [1]`を質問する．回答は Noであるため，その頂点の子へと移り，質問を続ける．`merge [] [2] = []`を質問する．回答は Noであり，かつ子を持たないため 4 よりこの頂点を誤りの原因とする．よって，`merge [] [2] = []`の式を誤りとして提示する．

プログラマは誤りと提示された式 `merge [] [2] = []` の処理から正しい処理について考える．そして `merge`では，どちらか片方のリストが空リストの場合，空リストでないリストをそのまま出力するのが正しい操作であると結論を出す．そこからソースコード 1 の 12 行目が間違いだと判断し，`merge [] xs = xs`と書き換えることでバグを修正する．

これが，hat-detect を用いたアルゴリズミックデバッグの一連の流れである．

3. hat-detect の課題

アルゴリズミックデバッガである hat-detect にはデバッグの際にいくつか課題が残されている．

3.1 プログラマへの質問の順序

アルゴリズミックデバッグでは，プログラムの誤りを特定するために質問を行う．この質問は計算過程を記録した ART を巡回することで行われる．Hat-detect でアルゴリズミックデバッグを行う場合，この巡回による質問の提示される順序はプログラムの実行順となる．

図 1 において，誤りは 12 行目の `merge [] xs = []`である．再帰関数である `merge`において基底部である定義が誤りであるとき，例え再帰部の定義が正しいとしてもその関数自体の計算は誤りとなってしまう．再帰関数は定義に基底部と再帰部という二つの処理を持ち，再帰関数作成においてもそれぞれの定義が重要となる．どちらか一方の定義が正しいがもう一方の定義が誤りである場合，その関数は正しい結果を示すことはない．

ここで，それぞれの定義の正誤とそのときの関数の正誤について考える．まず，再帰部と基底部の定義がそれぞれ，正しい場合，誤りを持つ場合，また引数によって正誤が変わる場合とする．これらの関係は表 1 となる．

表 1 再帰関数の構造の関係

基底部の定義	再帰部の定義	関数の正誤
正しい	正しい	正しい
正しい	誤り	引数によって正しい
正しい	引数に依存	引数によって正しい
誤り	正しい	誤り
誤り	誤り	誤り
誤り	引数に依存	誤り
引数に依存	正しい	引数によって正しい
引数に依存	引数に依存	引数によって正しい
引数に依存	誤り	誤り

これは基底部が誤りであるときは，再帰部がどのような場合であってもその関数は誤りである．再帰関数において基底部の重要性は高いと考えられる．また再帰部が基底部を使い処理を行うこと，hat-detect が評価順に質問を行うことから探索の際，少なくとも基底部まで質問を行わなければ例え再帰部が正しいかったとしても誤りを持つ箇所について判断ができない場合が多いということを示す．hat-detect において，評価通り `main`関数から質問を始めることはプログラムの実行の流れを追いやすいという点においてメリットを持つ．だが Haskell のように再帰関数を多用する言語において，基底部の正誤を確認することは重要である．引数に依存するが，ある再帰関数が誤りを持つときは少なくとも基底部が正しいかどうか確認しなければわからない場合が多い．しかし hat-detect の探索方法では関数の構造まで把握しないため，質問の回数が多くなると考えられる．

3.2 冗長な質問

hat-detect では同じ式を質問する場合がある．図 1 のプログラムを少し変更して例を示す．プログラムを図 4 に書き換える．図 4 は図 1 の 15 行目 **`if`**`(x < y)` を **`if`**`(x <= y)` とし，21 行目の `print (mergesort [3,1,2])` を `print (mergesort [10,10,10,1])` と変更する．これを実行すると hat-detect の質問において図 5 のようなことが起こる．

```
1  -- merge test
2  mergesort :: [Int] -> [Int]
3  mergesort [] = []
4  mergesort [x] = [x]
5  mergesort xss =
6      let xs = take l xss ;
7          ys = drop l xss ;
8          l = div (length xss) 2 ;
9      in merge (mergesort xs) (mergesort ys)
10
11 merge :: [Int] -> [Int] -> [Int]
12 merge [] xs = []
13 merge xs [] = xs
14 merge xss@(x:xs) yss@(y:ys)
15         = if (x <= y)
16                 then (x:merge xs yss)
17                 else (y:merge xss ys)
18
19 main :: IO ()
20 main = do
21         print (mergesort [10,10,10,1])
```

図 4 merge sort の実装

```
1  1  mergesort [10,10,10,1] = [1,10]
2  hat-detect> n
3  2  mergesort [10,10] = [10]
4  hat-detect> n
5  3  mergesort [10] = [10]
6  hat-detect> y
7  4  mergesort [10] = [10]
8  hat-detect> y
9  5  merge [10] [10] = [10]
10 hat-detect> n
11 6  merge [] [10] = []
12 hat-detect> n
13 bug found at ac1
14     merge [] [10] = []
15 Done.
```

図 5 ソースコード 2 を変更したプログラムのアルゴリズミックデバッグ

2 行目と 9 行目の質問を確認するとそれぞれ `mergesort [10,10] = [10]`と表示されている．また，5 行目と 7 行目の質問を確認すると `mergesort [10] = [10]`となっている．表示されている計算式とその結果は全く同じである．

Haskell は参照透過性を持つため，ある引数のときのある関数の計算結果は常に同じに結果になる．そのため，ある関数のある引数における計算結果の正誤は別の関数呼び出しであっても同じになる．よってこの質問は冗長であり，プログラマにとって負担となる．

しかし hat-detect ではデバッガ自身がした質問の内容である計算式とその結果，そしてプログラマによる回答を保存しない．これは hat-detect が現在質問している計算式の構造や引数，その等式の関係，さらに次に質問する計算式や過去にした質問の計算式の情報は全く使わず，計算木の呼び出し関係のみを利用するからである．

4. 改善手法の提案

本章では前章の課題に対し，改善について提案する．

4.1 質問の順番の改善

ここでは再帰に着目し，改善について行う．

再帰関数のデバッグでは一般的にまず，基底部が正しい値を返すことを確認することや想定している処理をするかどうかをみる．基底部が正しく動いたどうかを確認し，誤りを持つ場合は修正する．正しい場合は次に再帰部を確認する．再帰部では任意の値を用いて正しく動いているか考える．基底部は正しく動いていることを確認しているため，再帰の処理のみに注目すれば良い．そして誤りを持つ箇所を特定し修正する．

この流れをアルゴリズミックデバッグに適用する．現在は関数の処理の始めである再帰部を使用する計算式から探索が始まるが，逆に基底部を利用する計算式を先に探索する．基底部を利用する計算式が正しいことを確認できれば，再帰部を利用する計算式における正誤のみに注目すれば良くなる．また基底部を使用する計算式が間違いであれば，再帰部を使用する計算式の正誤にかかわらずその式を誤りとして提示することができる．よって基底部を使用する計算式を先に確認するよう変更する．

4.2 冗長な質問

Haskell は参照透過性を持つため，ある引数のときのある関数の計算結果は常に同じに結果になる．そこで，ある計算式が質問の対象になり，かつ記録された質問の中に同じ計算式が含まれているとき，その計算式の回答と同じ回答とする．デバッガでそれらの処理を行うことで，プログラマにとっては質問が行われない振る舞いに見えるため，質問が省略されることとなる．

4.2.1 関数構造と正誤結果の利用

ある計算式が全く同じ場合に質問を省略するが，このとき先に述べた再帰関数の構造について考えると，次のことが言える．一般に関数は一つの機能を持つ．よって，ある関数の結果が誤りであるとき，次に現れる関数の結果が誤りである可能性がある．同様にある関数の結果が正しいとき，次に現れる関数の結果が正しい可能性がある．再帰関

数は構造に基底部と再帰部を持つ．これは，関数が持つ機能は一つだが，処理を行う部分が異なることを意味する．例えば，定義した再帰関数の再帰部は間違っているが基底部が正しい場合，その再帰関数は正しい計算を行う場合と誤った計算を行う場合がある．そのため再帰関数の計算式においてある計算結果から同じ関数を使う別の計算結果の正誤を決定することは難しい．そこで，過去の回答を利用する場合，その回答の対象の式の構造を把握する必要がある．

アルゴリズミックデバッグを行うとき，現在質問している計算式が基底部であるか再帰部であるかをデバッガが判断するよう設計する．そして，プログラマが回答した計算式の関数名と構造，そして正誤結果を保持する．次に質問される計算式が記録された計算式と同じ関数名かつ同じ構造であるとき，記録された正誤結果を利用してデバッガが回答を行う．

5. 実験と評価

本章では，先に述べた手法を実装したアルゴリズミックデバッガと hat-detect を利用して実験を行う．

5.1 実験

バグを含む Haskell プログラムは，プログラミングコンテストサイト AtCoder[5] に提出されたものから収集する．開催されているコンテストのうち，1〜200 の A 問題と B 問題に提出されているプログラムを対象とする．ただし，明らかにアルゴリズミックデバッグに適さないバグを含むプログラムは対象とならない．

実験のために集めたプログラムを実験に適用する際に一つ修正を加えた．それは入出力である．AtCoder ではプログラムを作成するとき，必ず標準入力から値を受けるような形式にしなければならない．しかし，元の hat-detect では入力を扱えず，また今回実装したデバッガにおいても，入力の成否を尋ねる冗長な質問が発生する．そのため，元のプログラムを `let`などで束縛した値を関数に渡すように変更した．これによって，バグを含む関数の処理に対して質問数が増減することはない．

比較する手法として，hat-detect，基底部を優先するアルゴリズミックデバッグ，質問を省略するアルゴリズミックデバッグ，機能を複合したアルゴリズミックデバッグを対象に実験を行なった．機能を複合したアルゴリズミックデバッガは基底部を優先し，かつ過去に同じ計算式があるとき省略するアルゴリズミックデバッガである．質問を省略するあるゴリズミックデバッガと異なり，ある関数に対しある引数のときの計算結果の質問回答が記録されているとき，同じ関数と引数の式は過去の回答を使うというものだ．そのため，単に質問を省略するあるゴリズミックデバッガと処理が異なる．

表 2 37 のプログラムを対象に各アルゴリズムミックデバッガで実験を行なった結果

	正答率	平均回答数	最大回答数
hat-detect	0.76	4.28	16
基底部を優先する改良	1.00	2.67	22
質問を省略する改良	0.84	2.15	5
機能を複合した改良	1.00	2.13	7

これらデバッガがバグを含むプログラムに対し，アルゴリズミックデバッグを開始してから誤りを持つ計算式を特定するまでの回答数を比較する．実験結果は，表 2 の通りである．

hat-detect では，特定の関数の計算式を計算木に含めない．そのため，hat-detect を起動しても質問が現れない場合がある．今回の実験では，作成したアルゴリズミックデバッガでデバッグ可能だが hat-detect デバッグできないものも含まれる．その場合，私が新たに実装した hat-detect と同じアルゴリズムのデバッガを用いて，評価し回答数を記録する．

5.2 評価・考察

5.2.1 質問の優先順位を入れ替える

基底部を優先して質問するアルゴリズミックデバッガは，hat-detect に比べて平均の回答数が少ない．今回集めたプログラムの計算木のノードの平均は 5.68 個となる．計算ノードの多くが再帰な処理であり，その再帰処理が間違っていると，ほぼ全てのノードが間違っているパターンとなる．そのため，平均の回答数がノードの平均と近い値になる．

基底部を優先するアルゴリズミックデバッグの質問回数が，2.67 となる．これは，再帰関数を定義するとき，基底部は正しく再帰部が間違っているというパターンが多い傾向にあるためだと考えられる．今回作成した基底部を優先するアルゴリズミックデバッガは，基底部を尋ねた後に基底部を呼び出している再帰部の計算式を確認する．そのとき，再帰部に誤りがあり，誤っていると指摘すると質問の回数が 2 回で終わる．

これらからわかることは，バグを含む再帰関数をアルゴリズミックデバッグする場合は，基底部から基底部を呼び出す再帰処理の方向へと計算木を巡回するのが良いということである．

5.2.2 質問の解答を自動で決定する

質問を省略するアルゴリズミックデバッグは平均質問回数が少ない．これは，再帰関数をデバッグするとき，再帰部を 1 回尋ね，その後，基底部を 1 回質問するといった極端な挙動をするためである．しかし，必ずしもバグを見つけられるとは限らない．再帰関数定義において基底部が正しく，再帰部に誤りを持つ定義があるとする．このとき再帰部は小さい値では正しく動くが，値が大きくなるにつれ

て誤差が発生する問題を持つ場合，質問をデバッガ側で省略すると誤りを持つ計算式が省略されてバグを発見できない場合がある．そのため，アルゴリズミックデバッグにおいてなるべくプログラマが計算式を確認することが望ましい．

5.2.3 機能を複合したアルゴリズミックデバッグ

機能を複合したアルゴリズミックデバッグにおいて，最も平均回答数が少ない．これは基底部を優先するアルゴリズミックデバッグと同じ理由で少なくなっていると考えらえれる．しかし，基底部を優先するあるゴリズミックデバッグに比べて，少ない理由はもう一つあると考えられる．それは，全く同じ計算式をデバッガ側で省略するという点である．基底部の処理が複数箇所に現れる場合，基底部を優先するデバッガでは同じ質問に対し繰り返し，回答をしなければならない．しかし，Haskell は参照透過性を持つため同じ計算式が現れた場合，過去の回答を使い質問を省略することができる．そのため，基底部を優先する質問より，平均回答数が少なくなる．

5.2.2 において，なるべくプログラマが計算式を確認することが望ましいとしたが，これは引数が異なる場合のみである．ある引数における関数の正誤が，他の引数においても適用される場合がほとんどであるが，必ずしもそうではい．やはり，関数の引数で正誤は異なってくる．質問を省略する場合は，全く同じ計算式でなければならない．

6. 終わりに

本論文では，再帰関数の構造に着目し，ART を用いたアルゴリズミックデバッグの効率化について提案した．

再帰関数のデバッグ手法をアルゴリズミックデバッグに取り入れることで，質問数の削除を行なった．再帰関数を構成する基底部と再帰部の関係に着目し，それぞれを関数における別の定義として扱うことで，提案手法を実現した．また，アルゴリズミックデバッグを行うとき，プログラマが回答した結果とそのときの計算式を保存することで，同じ式が現れたときデバッガがその正誤を利用することで，質問の省略を可能にした．hat-detet と比べて，回答数を最大で平均 2.15 回ほど減らすことができた．

今後の課題として，今回比較対象とならなかった Hoed との比較，また，AtCoder での規模のプログラムではなく，より巨大なプログラムにおいての実験などが挙げられる．

謝辞

参考文献

[1] Manpage of HAT-DETECT. https://www.cs.york.ac.uk/fp/hat/hat-detect.1.html.
[2] 加藤 知樹, Haskell を対象としたトレーサ HatLight の GHC における拡張と実装, 修士研究, 愛知県立大学大学院, 2021
[3] E. Y. Shapiro. Algorithmic program debugging. MIT press, 1983.
[4] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multipleview tracing for Haskell: a new Hat. 2001 ACM SIGPLAN Haskell Workshop,pp. 151–170, 2001.
[5] AtCoder. https://atcoder.jp/ (2022 年 2 月 14 日閲覧)