競技プログラミングの鉄則*

やさしい理系お兄ちゃん

February 3, 2024

Contents

*https://atcoder.jp/contests/tessoku-book

0　はじめに

この PDF は関数型言語 Haskell の習得のために、「競技プログラミングの鉄則」の問題に対して Haskell で記述したコード例をまとめた資料です。私のプログラミング力の向上を主目的とし、Haskell を学びたい人への手助けとなると良いと考えています。Haskell の I/O[1] は一筋縄でいかないので、アルゴリズム以前の問題で苦戦してしまうかもしれないでしょう。ですが、入出力に時間を取られるのは勿体無いため、そのような面でも Haskell 習得の支えになれば良いと願います。

1　アルゴリズムと計算量

1.1　導入問題

問題. 整数 N が与えられるので、一辺の長さが N であるような正方形の面積を出力するプログラムを作成してください。

Listing 1: A01.hs

```
1 main :: IO ()
2 main = do
3     n <- readLn :: IO Int
4     print $ n * n
```

解説. `n <- readLn` で 1 つの整数を受け取ることができます。このまま用いると `n` の型は `Integer` と推論されますが、`:: Int` と型注釈をつけることで `n` を `Int` 型とすることができます。`readLn` の型は `Read a => IO a` ですが、なぜ、`Integer` と推論されるのでしょうか？それは、`(*)` という演算の引数として `n` が使用されているのが理由の一つです。

正方形の面積である `n * n` を出力するためには、関数 `print` を使用します。関数 `print` の型は `print :: Show a => a -> IO ()` で、型クラス `Show` のインスタンスである型を持つ値を受け取って IO アクションを返します。

問題. 整数 A と B が与えられるので、$A + B$ の値を出力するプログラムを作成してください。ただし、制約は $1 \leqq A \leqq 100$、$1 \leqq B \leqq 100$ であるとします。

Listing 2: B01.hs

```
1 main :: IO ()
2 main = do
3     [a, b] <- map read . words <$> getLine :: IO [Int]
4     print $ a + b
```

解説. まずは 2 つの空白区切りの整数の入力から説明します。初学者にとってはいきなり難しい話になります。関数 `getLine` で文字列を 1 行分読み取ることができます。`getLine` の型は `getLine :: IO String` となっています。うーん... 私たちは、中身の `String` の部分を整数値のリストに変換して、IO モナドから中身を取り出したいのです。どうすれば良いのでしょうか。具体例を用いて説明します。

例えば、入力が「`2 3`」のとき、`IO String` の中身の `String` は`"2 3"`という文字列になっています。この文字列に対して関数 `words` を適用すると、`["3","2"]` というように空白

[1] I/O：Input(標準入力)， Output(標準出力)

区切りの文字列をリストに分解することができます。さらに、各要素に対して関数 `read` を適用することで、`[3,2]` と整数値のリストにすることができます。では、実際に `IO String` という型を持つ値にどのようにして `words` や `read` を適用させることができるのでしょうか？

それを実現する関数が `(<$>)` となります。`(<$>)` を用いることで、`IO` モナドに包まれた値に対して関数を適用することができます。`(<$>)` について、定義は以下の通りです。

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

`<$>`の型に注目して下さい。`IO` は `Functor` のインスタンスですので、`f` を `IO` に置き換えると、`<$> :: (a -> b) -> IO a -> IO b` となります。また、

- `getLine :: IO String`
- `map read . words :: Read b => String -> [b]`

と定義されています。以上を合わせると

```
map read . words <$> getLine :: (Read b) => IO [b]
```

となることが分かります。`IO [b]` については、`:: IO [Int]` と型注釈をつけてあげることで、型を指定することができます。慣れるまで入力を受け取るときは、どのような型を扱っているかを意識するためにも型注釈をつけておきましょう。最終的に、`(<-)` を使うことで、`IO` モナドの中身である `Int` 型のリストを `[a, b]` に束縛することができます。（➡**注** 入力が 2 つであると分かっているので、`[a, b]` としています。）あとは計算結果を A01 と同様に出力すれば完了です。

1.2　全探索 1

問題. N 個の整数 A_1, A_2, ..., A_N の中に、整数 X が含まれるかどうかを判定するプログラムを作成してください。

Listing 3: A02.hs

```
1 main :: IO ()
2 main = do
3     [n, x] <- map read . words <$> getLine :: IO [Int]
4     as <- map read . words <$> getLine :: IO [Int]
5     putStrLn $ if x `elem` as then "Yes" else "No"
```

解説. 3, 4 行目に関しては Listing 2 の解説で紹介しました。よく見たら `n` を使っていませんね。そのため、3 行目は

```
[_, x] <- map read . words <$> getLine
```

としても良いです。そして、リスト `as` の中に要素 `x` が含まれているかは、関数 `elem` で判定することができます。関数 `elem` は中置記法を用いることが多いです。関数 `putStrLn` は `String` を受け取って `IO ()` を返します。なぜ関数 `print` を使わないのかは、使ってみるとわかるでしょう！

問題. A 以上 B 以下の整数のうち、100 の約数であるものは存在しますか。答えを Yes か No で出力するプログラムを作成してください。

Listing 4: B02.hs

```
main :: IO ()
main = do
    [a, b] <- map read . words <$> getLine :: IO [Int]
    let xs = [a..b]
    putStrLn $ if 0 `elem` map (100 `mod`) xs then "Yes" else "No"
```

解説. 4 行目の `[a..b]` は、`[a, a+1, a+2, ..., b-1, b]` を表します。このリストのすべての要素について、100 から割った余りを考えます。その余りのリストの中に `0` が含まれてるか否かを判定します。

1.3　全探索 2

問題. 赤いカードが N 枚あり、それぞれ整数 P_1, P_2, ..., P_N が書かれています。また、青いカードが N 枚あり、それぞれ整数 Q_1, Q_2, ..., Q_N が書かれています。太郎くんは、赤いカードの中から 1 枚、青いカードの中から 1 枚、合計 2 枚のカードを選びます。選んだ 2 枚のカードに書かれた整数の合計が K となるようにする方法は存在しますか。答えを出力するプログラムを書いてください。

Listing 5: A03.hs

```
main :: IO ()
main = do
    [_, k] <- map read . words <$> getLine :: IO [Int]
    ps <- map read . words <$> getLine :: IO [Int]
    qs <- map read . words <$> getLine :: IO [Int]
    putStrLn $ if k `elem` [p + q | p <- ps, q <- qs] then "Yes"
       else "No"
```

解説. 6 行目の

```
[p + q | p <- ps, q <- qs]
```

では、リスト `ps` とリスト `qs` から一つずつ要素を取り出し、それらを足し合わせて可能性のある答えをすべて計算して提示しています。とりあえず、足した結果をすべて用意しておくということです。これは**非決定性計算**と呼ばれます。このとき、

```
(+) <$> ps <*> qs
```

と書くことができます。これについては**アプリカティブ**という用語を聞くまでは保留で構いません。非決定性計算という言葉も同様です。いずれ、後者の記法の方がきれいだと感じる日が来るはずです！私はリストの内包表記も好きです。

問題. N 個の商品があり、商品 i ($1 \leqq i \leqq N$) の価格は A_i 円です。異なる 3 つの商品を選び、合計価格をピッタリ 1000 円にする方法は存在しますか。答えを Yes か No で出力するプログラムを作成してください。ただし、制約は $3 \leqq N \leqq 1000$ であるとします。

Listing 6: B03.hs

```
just :: Int -> Int -> [Int] -> Bool
just _ _ [] = False
just 1 num xs = num `elem` xs
just 2 num (x : xs) = any (\m -> m + x == num) xs || just 2 num
   xs
just k num as@(x : xs) = length as >= k && k >= 0
   && (just (k - 1) (num - x) xs || just k num xs)

main :: IO ()
main = do
   _ <- read <$> getLine :: IO Int
   as <- map read . words <$> getLine :: IO [Int]
   putStrLn $ if just 3 1000 as then "Yes" else "No"
```

解説. 本問も非決定性計算だから、Listing 5 と同様に

```
1000 `elem` [x + y + z | x <- as, y <- as, z <- as]
```

で可能性のある答えを列挙しよう！としては... 方針は正しいですが、その手法に誤りが生じています。これでは、同じ商品を選ぶ場合も含まれてしまうからです。

それでは、関数 `just` について説明します。`just k num xs` とは、リスト `xs` から、異なる `k` 個の要素を取り出して足し合わせたとき、その答えが `num` になることがあるか否かを示す式です。したがって、本文では入力から得たリストに `just 3 1000` を適用します。

① `just _ _ []`
リストが空なら、問答無用で `False` を返します。

② `just 1 num xs`
リストから 1 個だけ取り出して、`num` に等しいかを判定します。おなじみの関数 `elem` を使えば解決です。

③ `just 2 num xs`
まず、`any (\m -> m + x == num) xs` で、先頭要素とその他の要素を足して `num` になるかを判定しています。そこで、`True` が得られなかった場合、`just 2 num xs` で先頭要素を捨てて残りのリストで同じことを試します。以降、再帰的に判定します。

④ `just k num xs`
`k` 個取り出すので、`k` が負でないこと、リストの長さが `k` 以上であることを判定しておきます。(➡**注** このような判定を書かずとも、AtCoder では正しい入力が与えられることが保証されているので問題ありません。しかし、一般的にはそうではないので、意図せぬ入力にも備えておく癖をつけましょう！とは言っても、AtCoder ではこれを書くのは手間ですので、やっぱり以降はサボります。)

あとは、③ と同様に、まずは先頭要素とその他の要素について条件を満たす組合せが存在するかを判定します。このとき、`just (k - 1) (num - x) xs` に帰着されることを理解しましょう。

1.4　2 進法

問題. 整数 N が 10 進法表記で与えられます。N を 2 進法に変換した値を出力するプログラムを作成してください。

Listing 7: A04.hs

```
toBinary :: Int -> String
toBinary 1 = "1"
toBinary n = toBinary (n `div` 2) ++ if even n then "0" else "1"

assignZero :: Int -> String -> String
assignZero 0 str = str
assignZero x str = "0" ++ assignZero (x - 1) str

main :: IO ()
main = do
    n <- readLn :: IO Int
    let binary = toBinary n
    putStrLn $ assignZero (10 - length binary) binary
```

解説. 関数 `toBinary` の動きを見てみましょう。

```
toBinary 12 -> toBinary 6 ++ "0" -> toBinary 3 ++ "00"
            -> toBinary 1 ++ "100" -> "1100"
```

どうでしょう？2 進数を計算する方法は数 A で習いますね！

問題. 整数 N（8 桁以内）が 2 進法表記で与えられます。N を 10 進法に変換した値を出力するプログラムを作成してください。

Listing 8: B04.hs

```
bin2list :: Int -> [Int] -> [Int]
bin2list 0 ys = ys
bin2list x ys = bin2list (x `div` 10) (x `mod` 10 : ys)

list2dec :: [Int] -> Int
list2dec [] = 0
list2dec (x : xs) = x * (2 ^ length xs) + list2dec xs

main :: IO ()
main = do
    n <- readLn :: IO Int
    print $ list2dec $ bin2list n []
```

解説. 与えられた入力（例えば 101011）を

```
101011 -> [1,0,1,0,1,1] -> 2^5 + 2^3 + 2^1 + 2^0 -> 43
```

という流れで計算しています。入力を `Int` として受け取り、それをリスト構造に変換しているなら、最初からリスト構造である `String` で受け取るべきかもしれませんね。

おっと、関数 `print` が初登場しました。型は以下のようになっています。`putStrLn` との違いを見てみましょう。

```
print :: Show a => a -> IO ()
putStrLn :: String -> IO ()
```

1.5　チャレンジ問題

問題. 赤・青・白の 3 枚のカードがあります。太郎くんは、それぞれのカードに 1 以上 N 以下の整数を書かなければなりません。3 枚のカードの合計を K にするような書き方は何通りありますか。

Listing 9: A05.hs

```
main :: IO ()
main = do
    [n, k] <- map read . words <$> getLine :: IO [Int]
    print $ length [(a, b, c) | a <- [1..n], b <- [1..n], let c =
        k - a - b, 1 <= c, c <= n]
```

解説. リスト内包表記のおかげで解説いらず！`1` 以上 `n` 以下の範囲で、`a + b + c = k` となるように `c` を定めてあげます。出来上がったトリプルのリストの長さが答えになります。

2　累積和

2.1　一次元の累積和 (1)

問題. ある遊園地では N 日間にわたるイベントが開催され、i 日目には A_i 人が来場しました。以下の Q 個の質問に答えるプログラムを作成してください。

- 質問 1：L_1 日目から R_1 までの来場者数は？

  ⋮

- 質問 Q：L_Q 日目から R_Q までの来場者数は？

Listing 10: A06_00.hs

```
import Control.Monad ( replicateM_ )

cumulativeSum :: [Int] -> [Int]
cumulativeSum = scanl (+) 0

solve :: [Int] -> [Int] -> Int
solve s [l, r] = (s !! r) - (s !! (l - 1))

main :: IO ()
main = do
    [_, q] <- map read . words <$> getLine :: IO [Int]
    as <- map read . words <$> getLine :: IO [Int]
    let cumSum = cumulativeSum as
    replicateM_ q $ do
        lr <- map read . words <$> getLine :: IO [Int]
        print $ solve cumSum lr
```

解説. 新しい関数がたくさん登場しました。一つずつ紹介します。

関数 `scanl` の定義は以下です。

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl = scanlGo
    where
        scanlGo :: (b -> a -> b) -> b -> [a] -> [b]
        scanlGo f q ls = q : (case ls of
                              []   -> []
                              x:xs -> scanlGo f (f q x) xs)
```

なぜ、scanlGo を挟んでいるのか !? よく分かっていませんが、実装自体は理解できそうですね！動作を見てみましょう。

```
scanl (+) 0 [1,2,3,4,5] -> [0,1,3,6,10,15]
```

これがまさに累積和になります。引数のリストの長さを N とすると、関数 cumulativeSum は $O(N)$ となります。上記の例で考えると、2 日目から 4 日目までの来場者数は、10 - 1 = 9（人）となります。3 日目から 5 日目までの来場者数は、15 - 3 = 12（人）となります。

リストの k 番目の要素にアクセスするために関数 (!!) が使えます。したがって、質問に回答する関数 solve は

```
solve s [l, r] = (s !! r) - (s !! (l - 1))
```

と定義できます。（「0 日目は累計 0 人」という無意味とも思える情報をリストの先頭に置いておくことで、リストのインデックスと日にちの数字が同じ数字になるので扱いやすくなっています。）これを、関数 replicateM_ を用いて q 個の質問に適用します。ここでは、使用例だけ紹介します。

```
ghci> replicateM_ 3 (putStrLn "a")
a
a
a
```

これで完成！いえ、実行時間超過です。なぜでしょうか... オーダーに問題があるそうです。プログラムとしては正しい出力が得られますが、このプログラムは $O(N^2+Q)$ となっています。$0 \leqq N \leqq 10^5$ ですので、最悪計算量が 10^{10} となり制限時間超過となります。問題文では、$O(N+Q)$ であることが望ましいと書かれていますので、それを目標にプログラムを変更しましょう。

関数 cumulativeSum は $O(N)$ で、関数 replicateM_ は $O(Q)$ です。したがって、関数 (!!) が $O(1)$ であれば、非常に嬉しい！ということになります。しかしながら、関数 (!!) は $O(N)$ であるようです。下記の定義からも確認できますね。

```
(!!) :: [a] -> Int -> a
xs     !! n | n < 0 =  errorWithoutStackTrace
                            "Prelude.!!: negative index"
[]     !! _         =  errorWithoutStackTrace
                            "Prelude.!!: index too large"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)
```

では、$O(1)$ でリストの要素にアクセスするにはどうしたらよいのでしょうか？ Data.Array モジュールを使いましょう。関数 listArray はインデックスの範囲とリストを引数に取り、インデックスの付いた配列を返してくれます。

```
listArray (0, 5) [0,1,3,6,10,15]
-> array (0, 5) [(0,0), (1,1), (2,3), (3,6), (4,10), (5,15)]
```

配列では、i 番目の要素に $O(1)$ でアクセスでき、アクセスする関数は `(!)` となっています。

これらを用いて変更したプログラムは Listing 11 のようになります。関数 `cumulativeSum` では、累積和を取ったのちに関数 `listArray` を適用しています。関数 `solve` では、関数 `(!!)` が関数 `(!)` に変わっただけですね！

Listing 11: A06_01.hs

```
import Control.Monad ( replicateM_ )
import Data.Array ( Array, (!), listArray )

cumulativeSum :: Int -> Int -> [Int] -> Array Int Int
cumulativeSum l r as = listArray (l, r) $ scanl (+) 0 as

solve :: Array Int Int -> (Int, Int) -> Int
solve s (l, r) = (s ! r) - (s ! (l - 1))

main :: IO ()
main = do
    [n, q] <- map read . words <$> getLine :: IO [Int]
    as <- map read . words <$> getLine :: IO [Int]
    let cumSum = cumulativeSum 0 n as
    replicateM_ q $ do
        [l, r] <- map read . words <$> getLine :: IO [Int]
        print $ solve cumSum (l, r)
```

問題. 太郎君はくじを N 回引き、i 回目の結果は A_i でした。$A_i = 1$ のときアタリ、$A_i = 0$ のときハズレを意味します。「L 回目から R 回目までの中では、アタリとハズレどちらが多いか？」という形式の質問が Q 個与えられるので、それぞれの質問に答えるプログラムを作成してください。計算量は $O(N + Q)$ であることが望ましいです。

Listing 12: B06.hs

```
import Control.Monad ( replicateM_ )
import Data.Array ( Array, (!), listArray )

cumulativeSum :: (Int, Int) -> [Int] -> Array Int Int
cumulativeSum lr xs = listArray lr $ scanl (+) 0 xs

solve :: Array Int Int -> (Int, Int) -> Int
solve s (l, r) = (s ! r) - (s ! (l - 1))

main :: IO ()
main = do
    n <- readLn :: IO Int
    as <- map read . words <$> getLine :: IO [Int]
    q <- readLn :: IO Int
    let cumSum = cumulativeSum (0, n) as
    replicateM_ q $ do
        [l, r] <- map read . words <$> getLine :: IO [Int]
        let t = solve cumSum (l, r)
        putStrLn $ case (r - l + 1 - t) `compare` t of
            GT -> "lose"
            EQ -> "draw"
            LT -> "win"
```

解説. 累積和を計算するところまで全問と全く同じです。`t = solve cumSum (l, r)` としていますが、この `t` は何を表しているでしょうか。具体例で考えてみましょう。`as` が

```
[0, 1, 1, 0, 1, 0, 0]
```

のとき、`cumSum` は

```
array (0,7) [(0,0),(1,0),(2,1),(3,2),(4,2),(5,3),(6,3),(7,3)]
```

となります。リストの長さが 1 増えていますが、前問と同様に「0 回目はアタリ 0 個」という情報を付加しているためです。いや、その情報は必要ないんだ！という場合は、累積和の計算を `scanl1 (+) xs` としましょう。ここで、2 回目から 5 回目までについて考えてみます。2, 3, 5 回目でアタリが出ているので、アタリ 3 個・ハズレ 1 個となります。このアタリ 3 個がまさに `t` の値で、`solve cumSum (2, 5)` によって求まります。このとき、ハズレの個数は `r - l + 1 - t` と表現できます。あとは、この 2 つを比較して出力する文字を定めれば完了です。

2.2　一次元の累積和 (2)

問題. ある会社では D 日間にわたってイベントが開催され、N 人が出席します。参加者 i は L_i 日目から R_i 日目まで出席する予定です。各日の出席者数を出力するプログラムを作ってください。

Listing 13: A07.hs

```
import Control.Monad ( replicateM )
import Data.Array ( accumArray, elems, Array )

add2PreviousDay :: [Int] -> [(Int, Int)]
add2PreviousDay [l, r] = [(l, 1), (r + 1, -1)]
add2PreviousDay _ = []

cumulativeSum :: Array Int Int -> [Int]
cumulativeSum xs = scanl (+) 0 $ elems xs

main :: IO ()
main = do
       d <- readLn
       n <- readLn
       lrs <- concatMap (add2PreviousDay . fmap read . words) <$>
          replicateM n getLine
       let accArray = accumArray (+) 0 (1, d + 1) lrs
       mapM_ print $ tail $ init $ cumulativeSum accArray
```

解説. 例えば、イベントが 5 日間開催されたとして、やさ理くんは 2 日目から 4 日目まで参加したとしましょう。このとき、リスト `[0,1,0,0,-1]` の累積和をとると、`[0,0,1,1,1,0]` となり、やさ理くんが出席した日のみ 1 となっていることが分かります。もちろん、ここでも「0 日目は出席者 0 人」という情報を付加しています。

以上のように、`L` 日目から `R` 日目まで参加した人に関して、`[(L,1),(R+1,-1)]` というリストを用意しておき、上手く累積和をとることができれば良いわけです。その上手くやる方法を説明していきます。

問題. あるコンビニは時刻 0 に開店し、時刻 T に閉店します。このコンビニでは N 人の従業員が働いており、i 人目の従業員の出勤時刻は L_i、退勤時刻は R_i です（L_i, R_i は整数）。$t = 0, 1, \ldots, T-1$ について、時刻 $t + 0.5$ には何人の従業員が働いているかを出力するプログラムを作成してください。計算量は $O(N + T)$ であることが望ましいです。

Listing 14: B07.hs

```
import Control.Monad ( replicateM )
import Data.Array ( accumArray, elems, Array )

attendAndLeave :: [Int] -> [(Int, Int)]
attendAndLeave [l, r] = [(l, 1), (r, -1)]
attendAndLeave _ = []

cumulativeSum :: Array Int Int -> [Int]
cumulativeSum xs = scanl (+) 0 $ elems xs

main :: IO ()
main = do
    t <- readLn
    n <- readLn
    lrs <- concatMap (attendAndLeave . fmap read . words) <$>
       replicateM n getLine
    let accArray = accumArray (+) 0 (0, t) lrs
    mapM_ print $ tail $ init $ cumulativeSum accArray
```

解説.

2.3　二次元の累積和 (1)

問題. $H \times W$ のマス目があります。上から i 行目、左から j 列目にあるマス (i, j) には、整数 $X_{i,j}$ が書かれています。以下の Q 個の質問に答えるプログラムを作成してください。

- 質問 1：左上 (A_1, B_1) 右下 (C_1, D_1) の長方形領域に書かれた整数の総和は？
- 質問 2：左上 (A_2, B_2) 右下 (C_2, D_2) の長方形領域に書かれた整数の総和は？

  ⋮

- 質問 Q：左上 (A_Q, B_Q) 右下 (C_Q, D_Q) の長方形領域に書かれた整数の総和は？

Listing 15: A08.hs

```
import Control.Monad ( replicateM )
import Data.Array ( Array, (!), listArray )

twoDimensionalSum :: Int -> Int -> [[Int]] -> Array (Int, Int)
   Int
twoDimensionalSum h w xs =
       listArray ((0, 0), (h, w))
       $ concat
       $ scanl (zipWith (+)) (replicate (w + 1) 0)
       $ map (scanl (+) 0) xs

solve :: Array (Int, Int) Int -> [[Int]] -> [Int]
```

```
solve twoDimSum = map (\[x, y, z, w] ->
      twoDimSum ! (x - 1, y - 1) + twoDimSum ! (z, w)
            - twoDimSum ! (z, y - 1) - twoDimSum ! (x - 1, w))

main :: IO ()
main = do
      [h, w] <- map read . words <$> getLine
      xs <- map (map read . words) <$> replicateM h getLine
      q <- readLn
      qs <- map (map read . words) <$> replicateM q getLine
      mapM_ print $ solve (twoDimensionalSum h w xs) qs
```

解説.

問題. 二次元平面上に N 個の点があります。i 個目の点の座標は $(X_i,\ Y_i)$ です。「x 座標が a 以上 c 以下であり、y 座標が b 以上 d 以下であるような点は何個あるか?」という形式の質問が Q 個与えられるので、それぞれの質問に答えるプログラムを実装してください。$N \leqq 100000,\ Q \leqq 100000,\ 1 \leqq X_i,\ Y_i \leqq 1500$ を満たすケースで、1 秒以内に実行が終わることが望ましいです。なお、入力される値はすべて整数です。

Listing 16: B08.hs

```
import Control.Monad ( replicateM )
import Data.Array ( Array, (!), listArray, accumArray, bounds )

twoDimensionalSum :: Int -> Int -> [[Int]] -> Array (Int, Int)
   Int
twoDimensionalSum h w xs =
      listArray ((0, 0), (h, w))
   $ concat
   $ scanl1 (zipWith (+))
   $ map (scanl1 (+)) xs

arrayToLists :: Array (Int, Int) Int -> [[Int]]
arrayToLists arr = [[arr ! (i, j) | j <- [0..jMax]] | i <- [0..
   iMax]]
   where rightBound = snd (bounds arr)
         iMax = fst rightBound
         jMax = snd rightBound

solve :: Array (Int, Int) Int -> [(Int, Int, Int, Int)] -> [Int]
solve twoDimSum = map (\(a, b, c, d) ->
      twoDimSum ! (a - 1, b - 1) + twoDimSum ! (c, d)
            - twoDimSum ! (c, b - 1) - twoDimSum ! (a - 1, d))

main :: IO ()
main = do
   n <- readLn
   xys <- map ((\[x, y] -> ((x, y), 1)) . map read . words) <$>
      replicateM n getLine
   q <- readLn
   qs <- map ((\[a, b, c, d] -> (a, b, c, d)) . map read . words
      ) <$> replicateM q getLine
   let h = maximum $ map (\((x, _), _) -> x) xys
   let w = maximum $ map (\((_, y), _) -> y) xys
   mapM_ print $ solve (twoDimensionalSum h w $ arrayToLists $
      accumArray (+) 0 ((0, 0), (h, w)) xys) qs
```

解説.

2.4　二次元の累積和 (2)

問題. ALGO 王国は $H \times W$ のマス目で表されます。最初は、どのマスにも雪が積もっていませんが、これから N 日間にわたって雪が降り続けます。上から i 行目・左から j 列目のマスを $(i,\ j)$ とするとき、i 日目には「マス $(A_i,\ B_i)$ を左上とし、マス $(C_i,\ D_i)$ を右下とする長方形領域」の積雪が 1 cm だけ増加することが予想されています。最終的な各マスの積雪を出力するプログラムを作成してください。

Listing 17: A09.hs

解説.

2.5　チャレンジ問題

3　二分探索

3.1　配列の二分探索

3.2　答えで二分探索

3.3　しゃくとり法

3.4　半分全列挙

3.5　チャレンジ問題

4　動的計画法

4.1　動的計画法の基本

4.2　動的計画法の復元

4.3　二次元の DP(1)：部分和問題

4.4　二次元の DP(2)：ナップザック問題

4.5　二次元の DP(3)：最長共通部分列問題

4.6　二次元の DP(4)：区間 DP

4.7　遷移形式の工夫

4.8　ビット DP

4.9　最長増加部分列問題

4.10　チャレンジ問題