データ科学基礎演習B（4）

データ科学科目部会

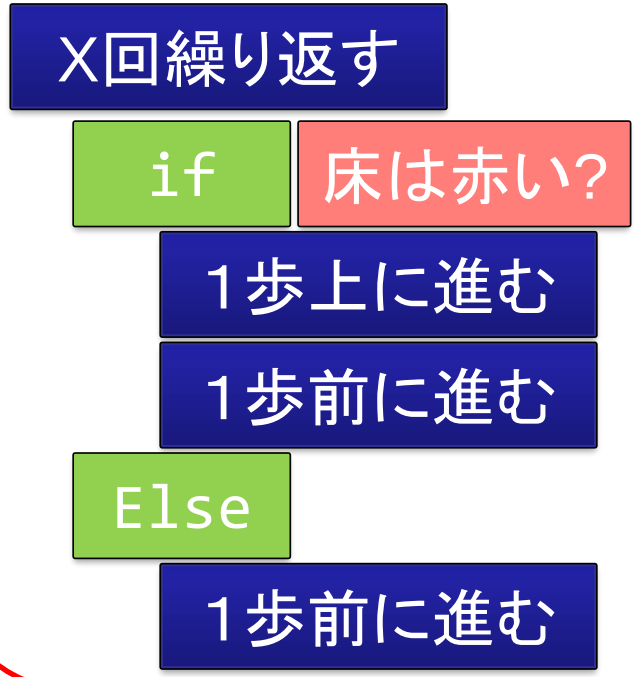
関数

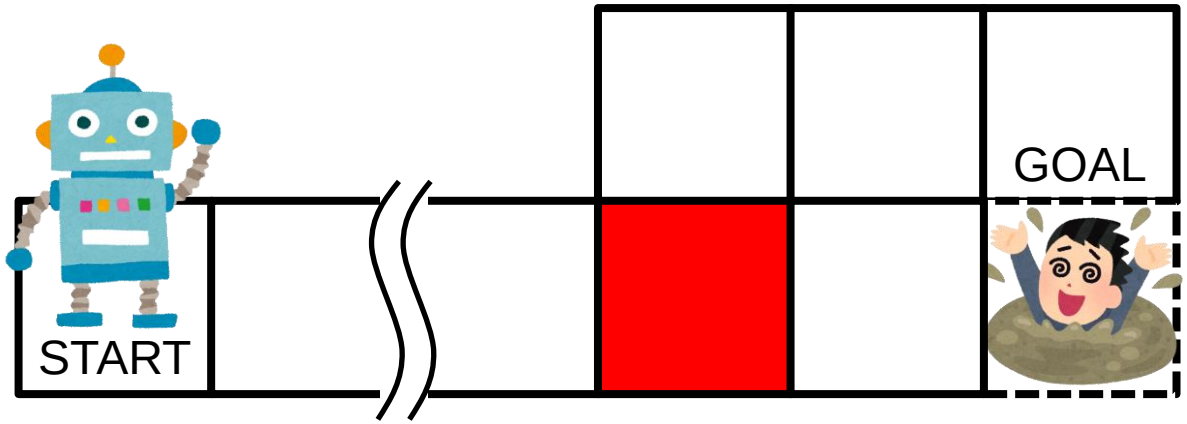
関数自作できるようになろう

どうすればロボットは目的地に着けるでしょう？

- 利用できる命令セット
  - ① 1歩前に進む
  - ② 1歩上に進む
  - ③ X回繰り返す
  - ④ 条件分岐（if-else）
    - ① True なら命令1
    - ② False なら命令2
  - ⑤ 床は赤い?

プログラム1





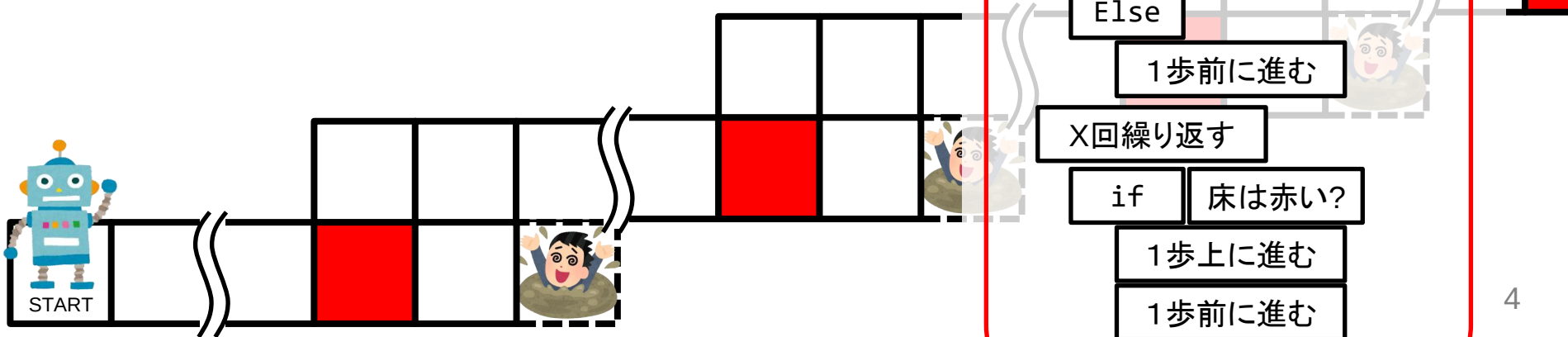
どれだけ同じことをすれば目的地に着ける？

- 利用できる命令セット
  - ① 1歩前に進む
  - ② 1歩上に進む
  - ③ X回繰り返す
  - ④ 条件分岐(if-else)
    - ① True なら命令1
    - ② False なら命令2
  - ⑤ 床は赤い?

何度も同じことを
書くのは無駄！

プログラム2

```
X回繰り返す
if 床は赤い?
    上に進む
    1歩前に進む
    床は赤い?
    1歩上に進む
    1歩前に進む
Else
    1歩前に進む
X回繰り返す
if 床は赤い?
    1歩上に進む
    1歩前に進む
```



関数とは？

- 一定の処理を再利用可能な形でまとめたもの

- 関数の使いみち
  - いろいろな所で同じ計算を何度もする場合
    - パラメータを変えて同じコードを使い回せます
  - 処理のまとまりに意味のある名前をつけたい場合
    - 何の処理をしているかがわかりやすくなります
  - 再帰的な処理をしたい場合(フィボナッチ数列など)

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{n+2} = F_n + F_{n+1}$$

0, 1, 1, 2, 3, 5, 8, 13, …

関数の定義方法

- 関数を定義する場合は先頭に def を付ける

先頭に必ず付ける

末尾はコロン(:)

```
def 関数名(引数1,引数2,…):
    …処理1…
    …処理2…
    return 戻り値
```

関数の定義方法

- 関数を定義する場合は先頭に def を付ける
- 引数は複数指定できる(カンマ区切り)

引数が0個の場合も括弧は付ける

```
def 関数名(引数1,引数2,…):
    …処理1…
    …処理2…
    return 戻り値
```

関数の定義方法

- 関数を定義する場合は先頭に def を付ける
- 引数は複数指定できる(カンマ区切り)
- 関数の戻り値は return で返す
- 関数を定義しただけでは実行されない

```
def 関数名(引数1,引数2,…):
    …処理1…
    …処理2…
    return 戻り値
```

if-elif-elseと
組み合わせると
条件に合わせて
戻り値を返せる

関数の定義と呼び出し（仮引数と実引数）

- 仮引数：関数（定義）が受け取る引数
- 実引数：関数呼び出し時に渡す引数

引数名は違ってもOK

仮引数

```
def func(a,b):
    z = a + b
    return z
```

関数の定義

```
m = 1
n = 10
k = func(m,n)
```

実引数

関数の呼び出し

関数の定義と呼び出し（仮引数と実引数）

[1]
```
1  def func(arr):
2      for a in arr:
3          print(a)
```

[2]
```
1  data1 = [1, 2, 3]
2  data2 = [9, 8, 7]
```

[3]
```
1  func(data1)
```

```
1
2
3
```

[4]
```
1  func(data2)
```

```
9
8
7
```

デフォルト引数

- 仮引数に対応する実引数が省略された時に使う値を指定する方法

デフォルト引数

```
def func(a,b=1,c=2):
    z = a + b + c
    return z
```

```
[1]  1  def func(a, b=1, c=2):
     2      z = a + b + c
     3      return z

[2]  1  func(1)

     4

[3]  1  func(1, 2)

     5

[4]  1  func(1, 2, 3)

     6
```

キーワード引数

- 特定の仮引数に対して直接値を指定する方法
  - 引数の順番を無視できる

```
def func(a,b=1,c=2):
    z = a + b + c
    return z
```

```
[1]  1  def func(a, b=1, c=2):
     2      z = a + b + c
     3      return z
```

```
[2]  1  func(1)
```

```
4
```

```
[3]  1  func(1, c=5)
```

```
7
```

```
[4]  1  func(1, b=2, c=3)
```

```
6
```

関数の戻り値

- 関数の計算結果を呼び出し元に返す方法
  - 関数内で return は何回も使うことができる
    - `if-elif-else` との組み合わせで戻り値の条件分け
  - 戻り値を返す必要がなければ return を省略できる
  - 計算結果をそのまま戻り値にできる

```
return x+y
```

  - 戻り値をタプルにすると複数の値を同時に返せる

```
return x,y
```

タプルの括弧は省略可

関数の戻り値がタプルの場合

- 戻り値のタプルの受け取り方は3種類

```
def func(a,b):
    return a,b

z    = func(1,2)
x,y  = func(1,2)
_,w  = func(1,2)
```

戻り値をそのまま変数に代入（zはタプル）

戻り値のタプルを展開して変数に代入（xとyは1と2）

タプルを展開した後で使わないものはアンダーバーを指定（Pythonのお作法）

関数の戻り値がタプルの場合

- 戻り値のタプルの受け取り方は3種類

```
def func(a,b):
    return a,b

z    = func(1,2)
x,y  = func(1,2)
_,w  = func(1,2)
```

```
[1]  1  def func(a,b):
     2      return a,b

[2]  1  z = func(1,2)
     2  z

     (1, 2)

[3]  1  x,y = func(1,2)
     2  x

     1

[4]  1  y

     2

[5]  1  _,w = func(1,2)
     2  _

     1
```

変数のスコープ

- 関数内で定義(代入)された変数は関数内でのみ利用できる
  - 関数の外から関数の内は見えない
  - 関数の内から関数の外は見える

```
g = 10

def func():
    return g+10
```

gは10

```
def func():
    g = 10
    return g+10

print(g)
```

func内のgは見えないのでエラー

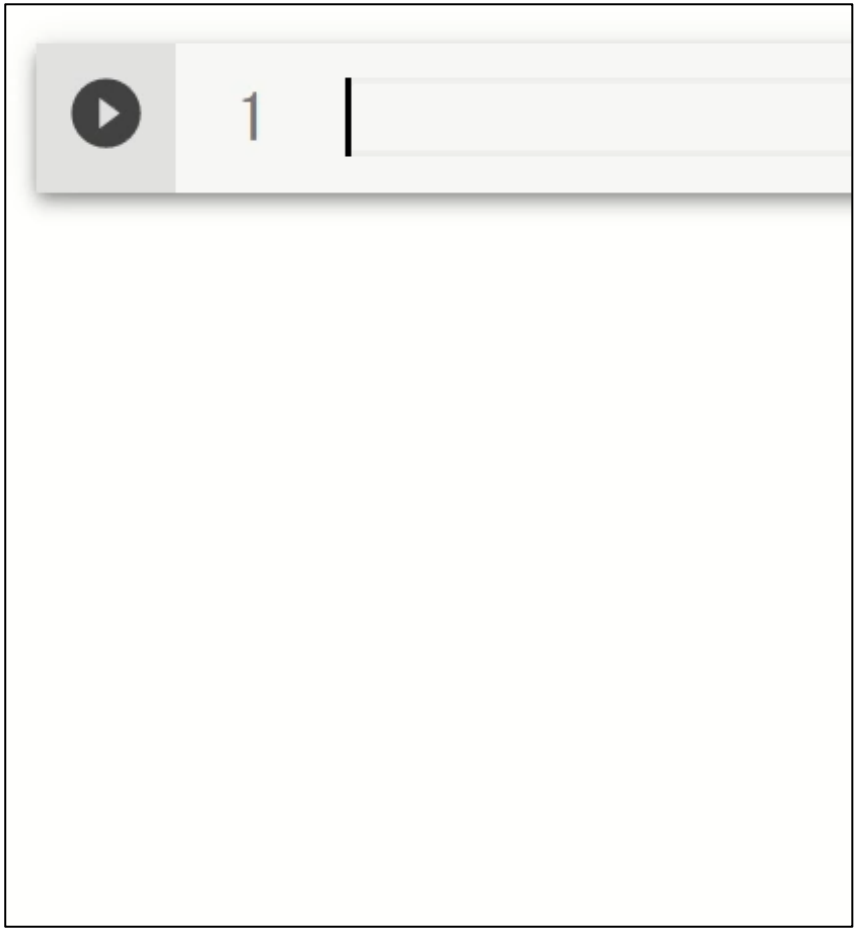
変数のスコープ

- 同じ変数名でも関数内と関数外は別物として扱われる

```
a = 10

def func():
    a = 5
    print(a)

func()
print(a)
```



（補足）コメントアウトの方法

- コメントアウト ＝ コメント化して実行から除外
  - プログラムの説明や一時的な処理の無効化に使う
- Pythonでのコメントアウトにはシャープ（#）を使う
  - シャープ（#）の記号から行末までをコメント化

```
def func():
    #a = 5
    a = 10 #コメント
    print(a)
```

緑色の部分は
実行されない

（補足）print関数で改行しない方法

- print関数のデフォルト動作は値を出力後に改行
  - 仮引数endを指定することで出力後の動作を変更可

```
print(値, end='¥n')
```

デフォルトは改行が指定されている

```
[1]  1  for i in range(5):
     2  |  print(i)
```

```
0
1
2
3
4
```

print関数のデフォルト動作

```
[2]  1  for i in range(5):
     2  |  print(i, end=',')
```

```
0,1,2,3,4,
```

仮引数endにカンマを指定した場合

再帰的な関数(再帰関数)

- 関数の中から自分自身(関数)を呼び出す関数

  例)1,2,…,N の和を求める場合

$$a_0 = 0$$
$$a_n = n + a_{n-1}$$

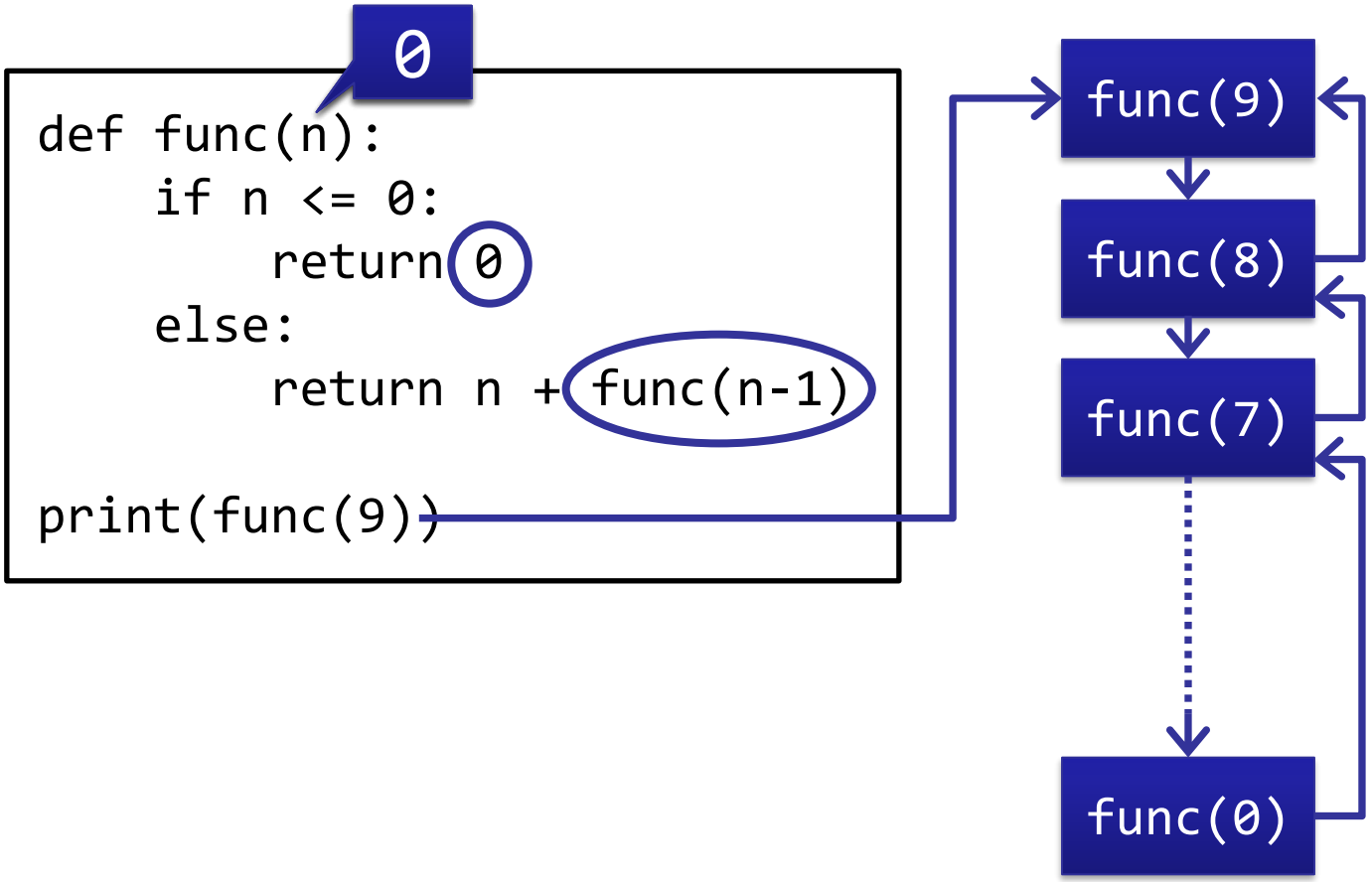
```
def func(n):
    if n <= 0:
        return 0
    else:
        return n + func(n-1)

print(func(9))
```

再帰的な関数（再帰関数）

- 関数の中から自分自身（関数）を呼び出す関数

例）1,2,…,N の和を求める場合



フィボナッチ数列

- 「隣り合う2つの数の合計が次の数」となる数列
  – 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

$$a_1 = a_2 = 1$$
$$a_n = a_{n-1} + a_{n-2} \quad (n \geq 3)$$

[1]
```
def fibo(a):
    if a <= 2:
        return 1
    else:
        return fibo(a-1)+fibo(a-2)

for i in range(1,11):
    print(fibo(i), end=',')
```

```
1,1,2,3,5,8,13,21,34,55,
```

課題

- データ集合内の要素を高速に探してみよう

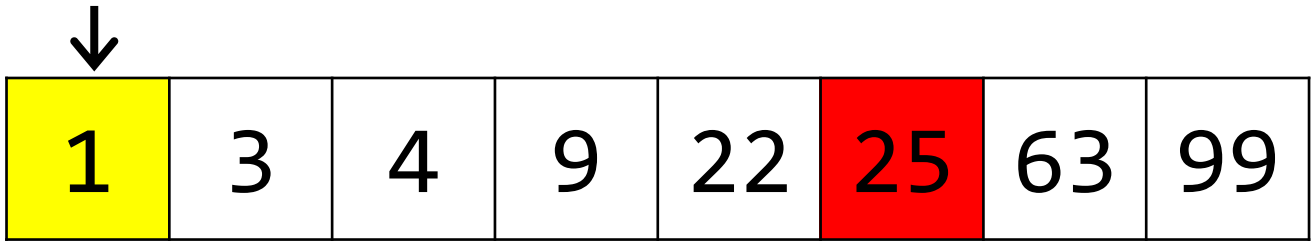
- 下記URLから演習課題のノートブックを自身のGoogle Driveにコピーし，Google Collaboratoryを起動して各設問に回答すること
  - https://bit.ly/2UBY1bf

- 演習課題を提出する際は，ノートブックのURLで共有し，そのURLを提出すること
  - 課題のヒントは次ページで紹介

課題のヒント（2分探索）

- データ集合から目的の値を高速に探す方法
  - データ集合が予めソートされていればO(logN)の計算量で目的の値を探索できるアルゴリズム

- 要素数N=8のデータ集合から値25を探す場合
  - 単純な探索方法
    - 左端から順番に目的の値を探す
    - 探す値が右端の方にあると時間がかかる



課題のヒント（2分探索）

- 2分探索はデータ集合を2分割しながら対象の範囲を絞り込む探索方法
- データが昇順に並んでいる場合の手順
  - 探索する値をYとし，以下を繰り返す

① データ集合の真ん中の値をXとする

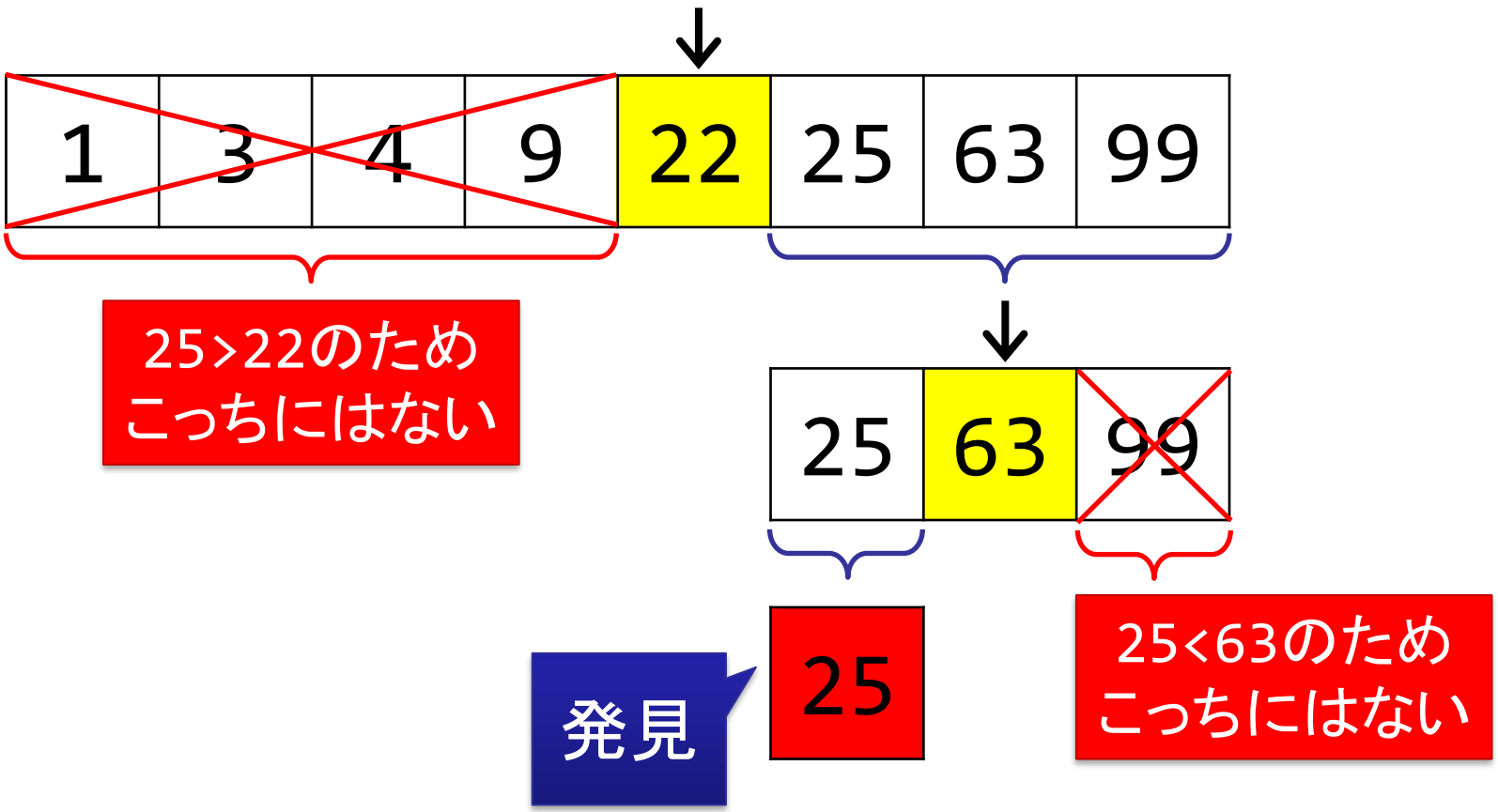
② Y==Xの場合
- 探索する値が見つかったのでXを返す

③ Y<Xの場合
- Xの左側のデータに対して①～④を適用

④ Y>Xの場合
- Xの右側のデータに対して①～④を適用

課題のヒント（2分探索）

- 2分探索の処理の流れ（値25を探す場合）



課題提出時の注意点

- 課題提出の際には提出前に必ず以下2点を確認すること
  - Google Collaboratoryの共有設定の際には『リンクを知っている全員』にチェックを入れる
  - 課題提出時に貼り付けたURLの末尾が「?usp=sharing」となっている（共有設定を開き、「リンクのコピー」ボタンを押して共有用のURLをコピー＆ペーストする）