プログラミングコンテスト基礎テクニック

情報知識ネットワーク研究室　M2　谷 陽太

目次

- テクニックの前に
  - HCPC勉強会に来たときの心得
  - 役立つ本の紹介
  - 時間計算量の気持ち
  - C++の便利機能入門
- 全探索
  - 線形探索
  - 深さ優先探索
- 簡単な算数
  - 最大公約数と最小公倍数
  - 素数判定
- 貪欲法
- シミュレーション

C言語の基礎的なプログラミングが
できることを前提に進めます

(Python勢やJava勢の人、ゴメンネ！)

目次

- **テクニックの前に**
  - **HCPC勉強会に来たときの心得**
  - **役立つ本の紹介**
  - **時間計算量の気持ち**
  - **C++の便利機能入門**
- 全探索
  - 線形探索
  - 深さ優先探索
- 簡単な算数
  - 最大公約数と最小公倍数
  - 素数判定
- 貪欲法
- シミュレーション

HCPC勉強会に来たときの心得

分からなくなったら
必ずその場で質問すること！

HCPC勉強会に来たときの心得

分からなくなったら
必ずその場で質問すること！

以上！

役立つ本の紹介

もし、プロコンのために
1冊だけ本を読むとしたら……

プログラミングコンテスト チャレンジブック

- 通称「蟻本」
- 基礎的なものから上級テクニックまで
  例題と合わせて紹介されている
- 先輩に頼めば貸してくれる
- ただ、いきなり読むにはちょっとムズいかも

時間計算量 (Time Complexity) の気持ち

時間計算量: この方法で問題を解くと、
実行にどのくらい時間かかるの？ という指標

プロコンでは、実行時間に関する制限があるので、
時間計算量を見積もることが大切 (1〜2秒程度のことが多い)

時間計算量 (Time Complexity) の気持ち

時間計算量: この方法で問題を解くと、
実行にどのくらい時間かかるの？ という指標

プロコンでは、実行時間に関する制限があるので、
時間計算量を見積もることが大切 (1〜2秒程度のことが多い)

- 「このループは何回回ってるかな？」
  「この関数は何回実行されてるかな？」という感じで見積もる

- だいたいN回の計算で終わることを **O(N)時間** と書く

例でみる時間計算量

- O(N)時間の例:

```
int sum = 0;
for(int i = 0; i < N; i++)
  sum += i;
```

- $O(N^2)$時間の例:

```
int a[N][N];
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    a[i][j] = i * j;
```

- O(logN)時間の例:

```
int count = 0;
while(N > 1){
  N /= 2;
  ++count;
}
```

- O(1)時間の例:

```
N *= 2;
printf("%d\n", N);
```

Oは大雑把のオー (と覚えればOK)

- 時間計算量は、細かな違いを無視して**大雑把に見積もる**

例1: ループ1周毎が重くても、
$O(N+N+N)$ みたいになっても、
定数倍は気にせず $O(N)$

```
int sum = 0;
for(int i = 0; i < N; i++)
  sum += i;

for(int i = 0; i < N; i++)
  sum += i;

for(int i = 0; i < N; i++){
   sum += i;
   sum += i;
   sum += i;
   sum += i;
}
```

例2: $O(N^2 + N)$ みたいになったら、
明らかに他より計算が軽い項は
無視して $O(N^2)$

```
int a[N][N];
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    a[i][j] = i * j;

for(int i = 0; i < N; i++)
  a[i][i] = 0;
```

実際の実行時間の概算

- 時間計算量を求めたら、**実際の実行時間を概算しよう！**

  プロコンの問題には、入力される数値の制約が必ず書いてあるので、
  それの最大値を、さっき求めた計算量にぶち込んでみる

実際の実行時間の概算

- 時間計算量を求めたら、**実際の実行時間を概算しよう！**

プロコンの問題には、入力される数値の制約が必ず書いてあるので、
それの最大値を、さっき求めた計算量にぶち込んでみる

基準: **今のパソコンは1秒間に10^8回程度の計算が可能**

時間計算量	プロコン的にOKなNの範囲
$O(N^2)$	$N \leq 10^4$ 程度
$O(NlogN)$	$N \leq 10^6$ 程度
$O(N)$	$N \leq 10^8$ 程度
$O(1)$	なんでもござれ

C++の便利機能入門

C++には、Cには無かった便利機能が満載

C++の便利機能入門

C++には、Cには無かった便利機能が満載

→ **なので使おう！**

C++の便利機能入門

C++には、Cには無かった便利機能が満載

→ **なので使おう！**

- たくさん紹介したいところだけど、**今回は2つだけ紹介**

いろいろ知りたい人は
HCPCのホームページへ！

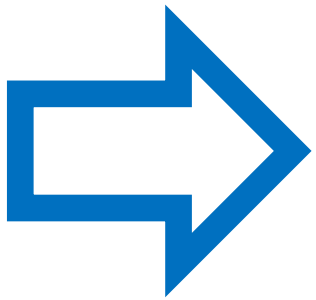
HCPC活動記録 (初めてのSTL)
http://hcpc.web.fc2.com/activities.html

C++の便利機能その1: cin / cout

- **標準入出力が手軽にできる！**
  - scanf/prinf: 型によって %d とか %f とかを使い分けないといけない
  - cin/cout　：型を意識する必要がない！ 便利！

```c
#include <stdio.h>

int main(void){
  int a;
  double b;
  long long int c;
  scanf("%d%lf%lld", &a, &b, &c);
  printf("%d%f%lld\n", a, b, c);
  return 0;
}
```



```c
#include <iostream>
using namespace std;

int main(void){
  int a;
  double b;
  long long int c;
  cin >> a >> b >> c;
  cout << a << b << c << endl;
  return 0;
}
```

C++の便利機能その2: vector

- **可変長配列！**

C言語での配列の
完全上位互換と考えてOK

```cpp
#include <vector>
using namespace std;

int main(void){
  int s, n = 30, m = 20;

  vector<int> a;     // int型での基本的な宣言の方法（最初は長さ0の状態）
  vector<int> b(n); // int型で、最初から長さnの領域を確保する
  vector<double> c(n, 6.5); // double型で長さnの領域を確保しつつ、そのすべてを6.5で初期化
  vector<vector<int> > d(n, vector<int>(m, 0)); // 入れ子にすることで多次元にできる

  a.push_back(3); // 末尾への要素の追加 O(1)
  c[20] = 100;    // 配列と同様にランダムアクセス可能 O(1)
  s = a.size();   // 要素数の取得 O(1)

  return 0;
}
```

目次

Q1.

3つの整数a, b, cが与えられるので、
aからbまでの中に、cの約数がいくつあるか求めよう！

- **Constraints**
  $1 \leq a, b, c \leq 1000$
  $a \leq b$

- **Time Limit**
  1 sec

例: $a = 5,\ b = 14, c = 80$ の場合 → **3つ**

5	6	7	**8**	9	**10**	11	12	13	14

Q1.

3つの整数a, b, cが与えられるので、
aからbま

- Cons
  1 ≤
  a ≤
- Time
  1 sec

aからbまで順番に見ていって
cを割り切れるかチェックすればいい！

												13	14

Q1.

3つの整数a, b, cが与えられるので、
aからb

- Cons
  1 ≤
  a ≤
- Time
  1 sec

aからbまで順番に見ていって
cを割り切れるかチェックすればいい！

この手法を **線形探索** とよぶ

13	14

実装例

- 時間計算量: **O(b)時間**

  $b \leq 1000$ なので間に合う！

  基本的に線形探索は
  ループを回すだけなので
  実装が簡単

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main(void){
 5   int a, b, c;
 6   cin >> a >> b >> c;
 7
 8   int ans = 0;
 9   for(int i = a; i <= b; i++)
10     if(c % i == 0)
11       ++ans;
12
13   cout << ans << endl;
14   return 0;
15 }
```

Q2.

地図が描かれたH×Wの盤面が与えられるので、
家から魚屋まで辿り着けるかどうか判定しよう！

- **Input**

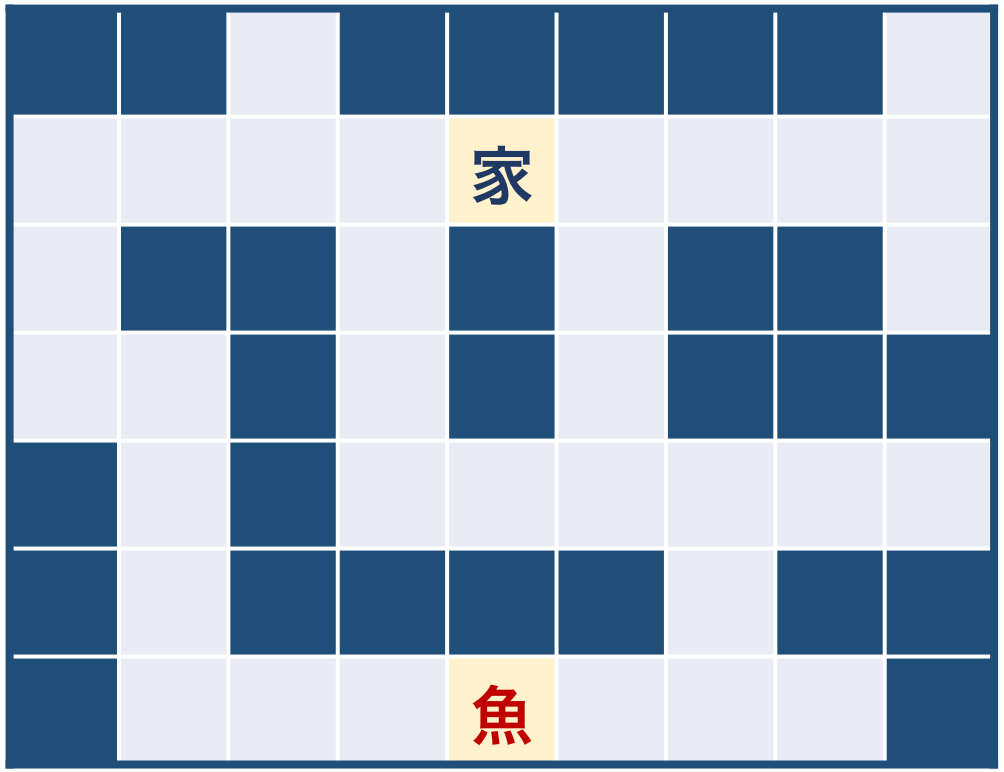
  1行目 H W
  2行目 盤面の情報 $c_{(1,1)}c_{(1,2)}\cdots c_{(1,W)}$
  ⋮
  H+1行目 盤面の情報 $c_{(H,1)}c_{(H,2)}\cdots c_{(H,W)}$

- **Constraints**

  $1 \leq \mathrm{H}, \mathrm{W} \leq 500$

- **Time Limit**

  2 sec



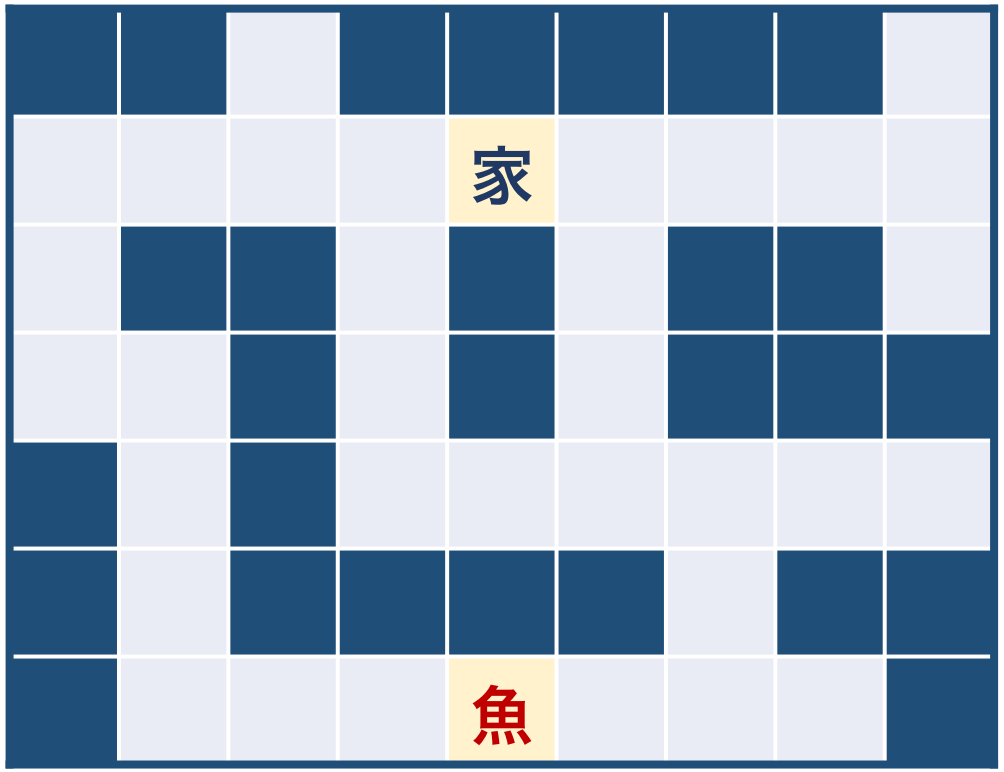
Q2.

地図が描かれたH×Wの盤面が与えられるので、
家から魚屋まで辿り着けるかどうか判定しよう！

- **Sample Input**

```
7 9
##.######.
....s....
.##.#.##.
..#.#.###
#.#......
#.#####.##
#...g...#
```

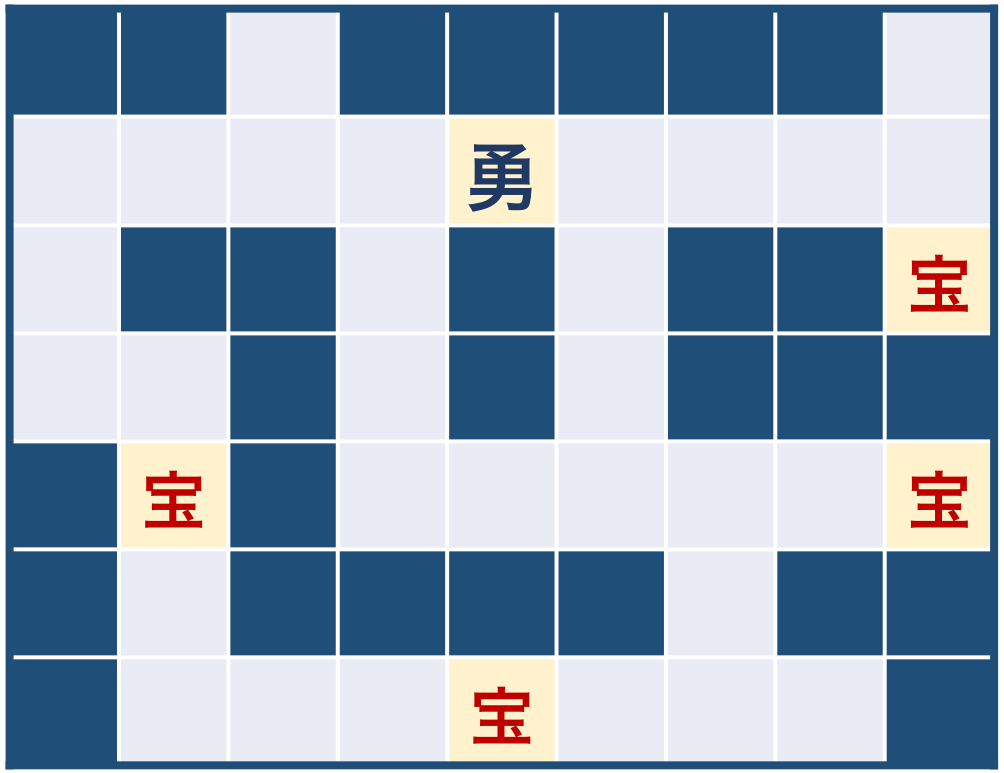
.：通路
#：壁
s：家
g：魚屋



どうやって解けばいいだろう？

- 例えば、この地図が**ダンジョン**だったとしよう

あなたは勇者なので、
このダンジョンのすべての宝箱を
開けたいと思っている

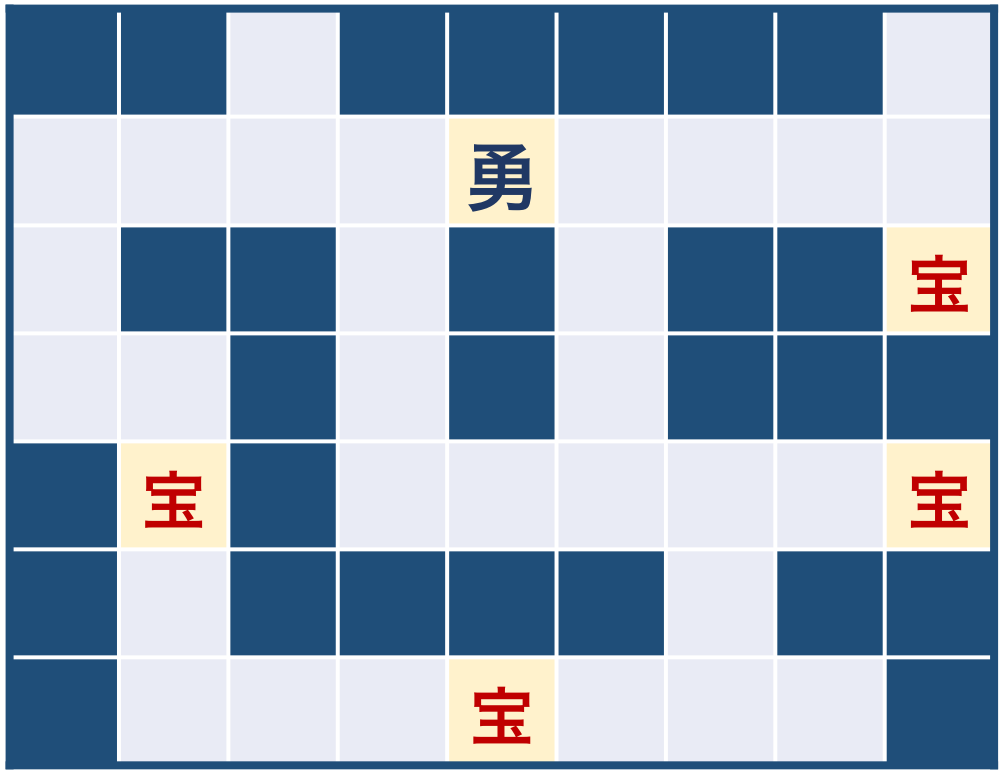


どうやって解けばいいだろう？

- 例えば、この地図が**ダンジョン**だったとしよう

あなたは勇者なので、
このダンジョンのすべての宝箱を
開けたいと思っている

Q. どうやって宝箱を探す？



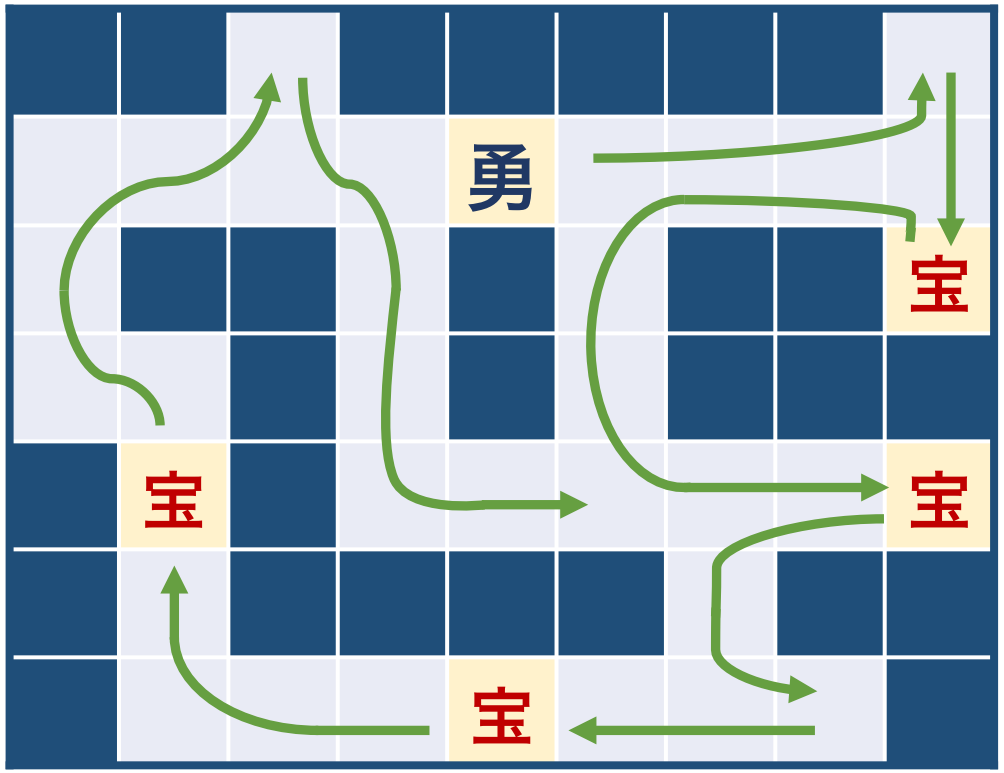
どうやって解けばいいだろう？

- 例えば、この地図が**ダンジョン**だったとしよう

あなたは勇者なので、
このダンジョンのすべての宝箱を
開けたいと思っている

Q. どうやって宝箱を探す？

A. こんな感じで探す！



どうやって解けばいいだろう？

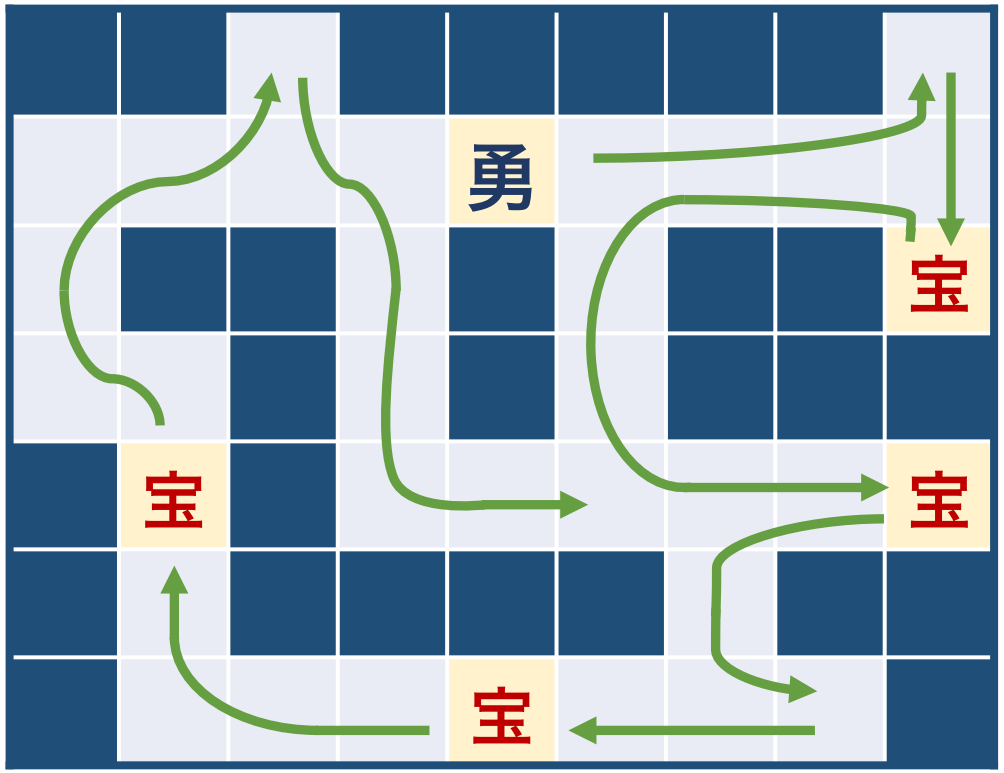
- 例えば、この地図が**ダンジョン**だったとしよう

あなたは勇者なので、
このダンジョンのすべての宝箱を
開けたいと思っている

Q. どうやって宝箱を探す？

A. こんな感じで探す！

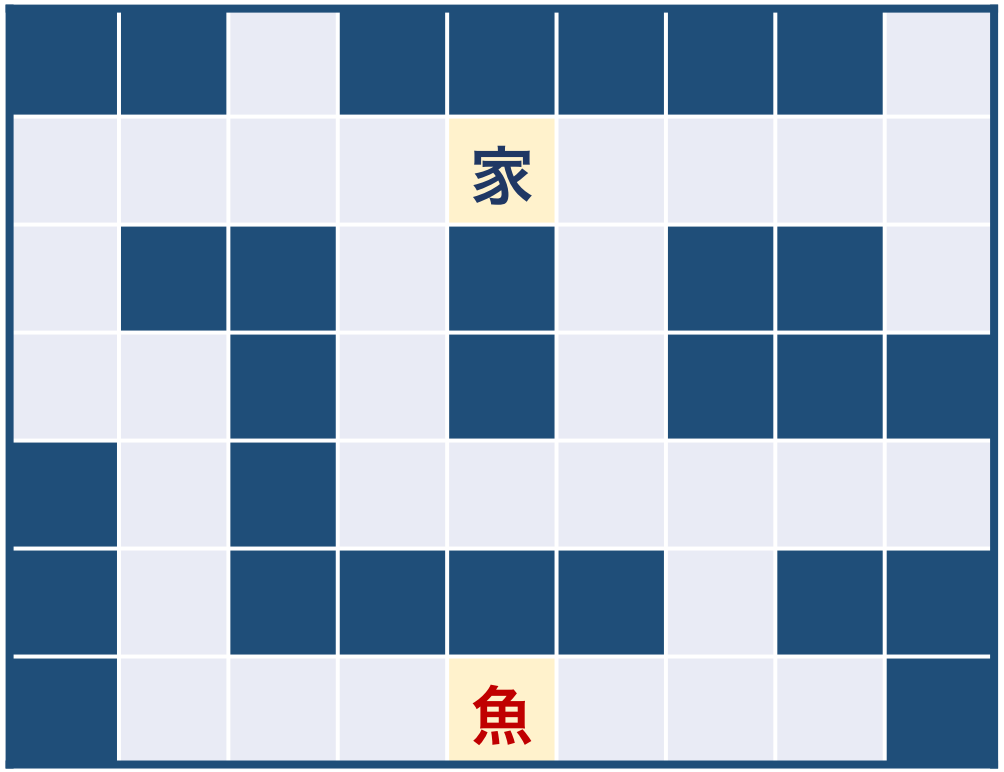
この手法を
深さ優先探索 とよぶ



深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

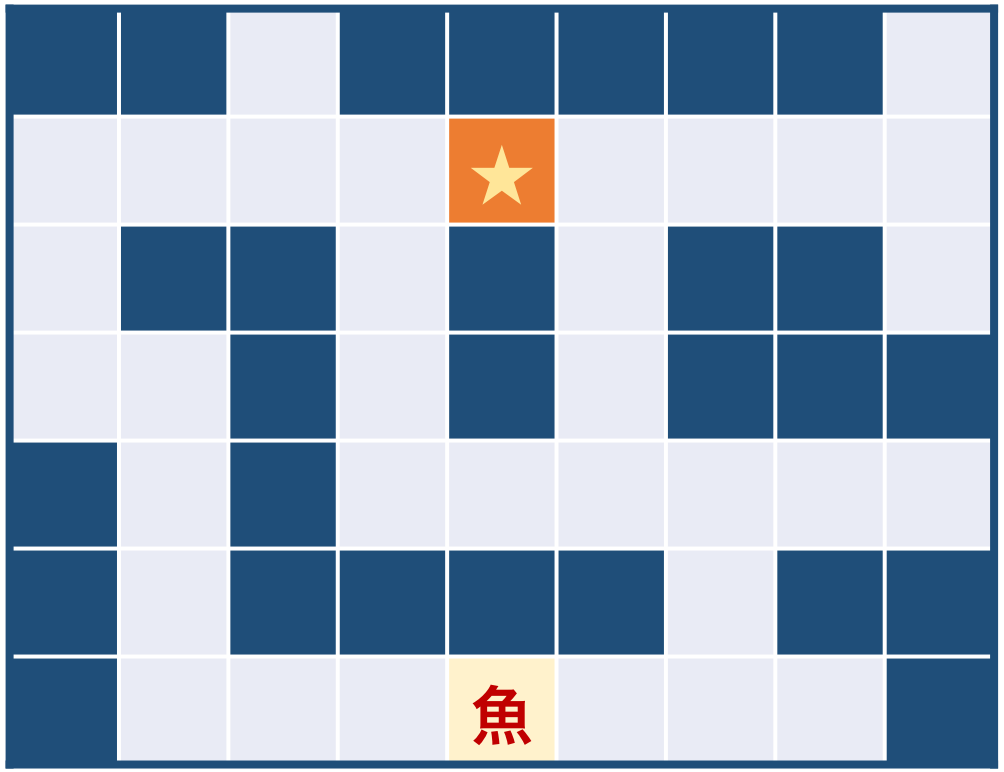


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

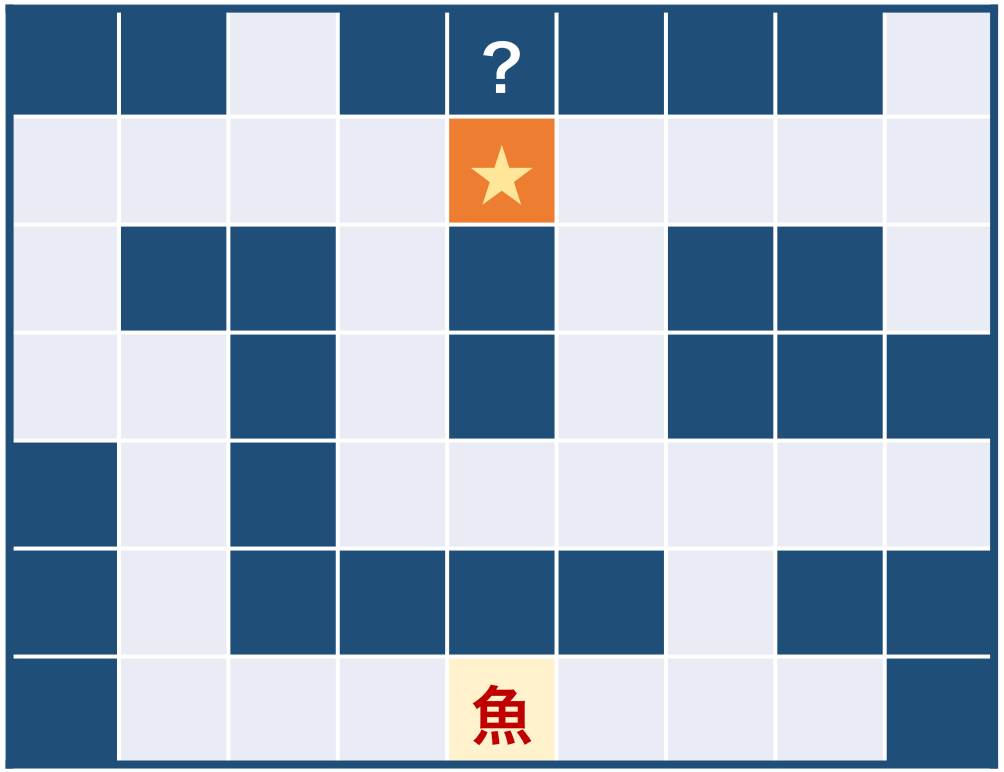
最初に、スタート地点へ
探索済みフラグを立てる



深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合



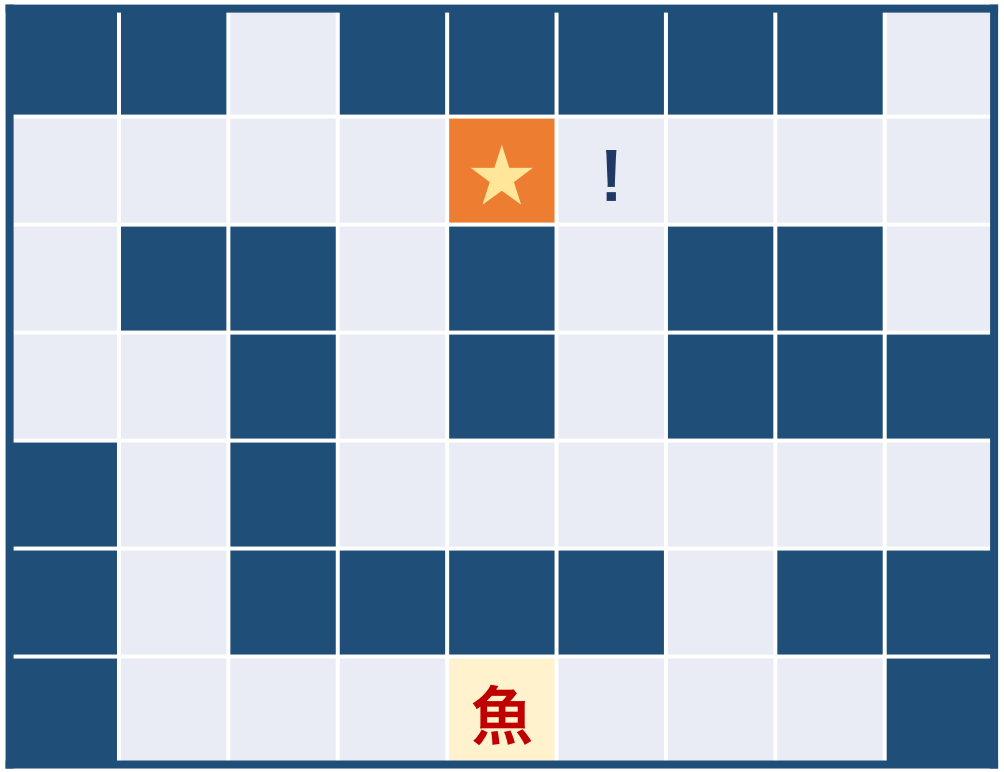
優先順位に従って上を見るが
壁なので行けない

深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

次に右を見る → **行ける！**

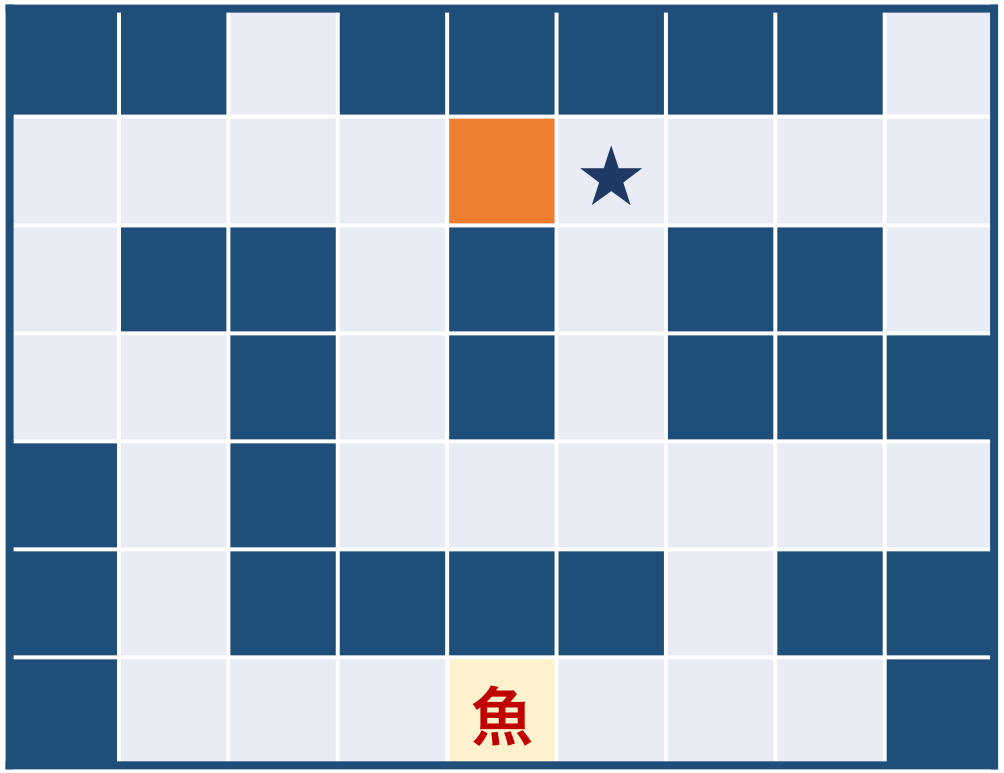


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

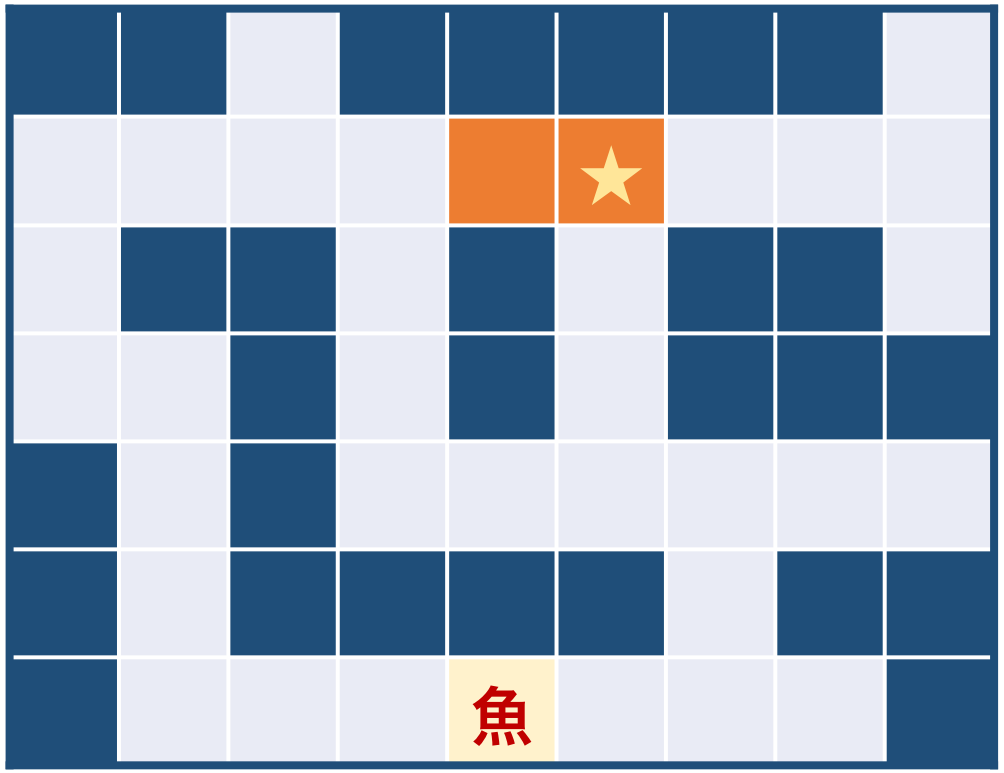


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進んだら、
その地点を「探索済み」にする

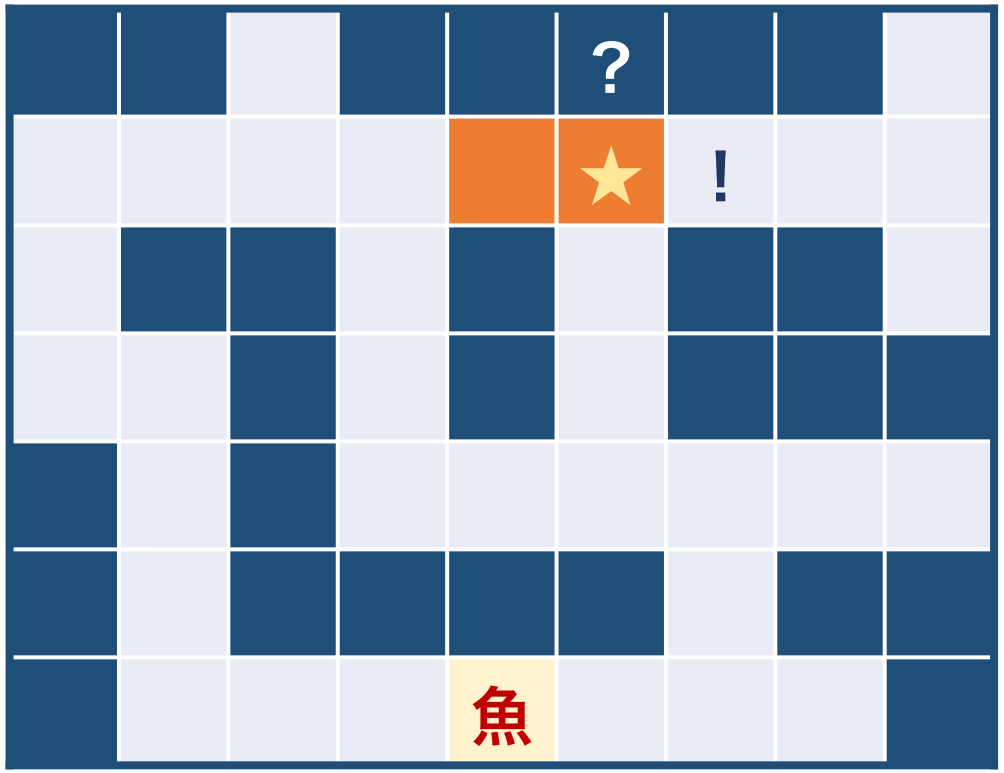


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

優先順位に従って
上を見るが行けず、
次に右を見ると行けそう

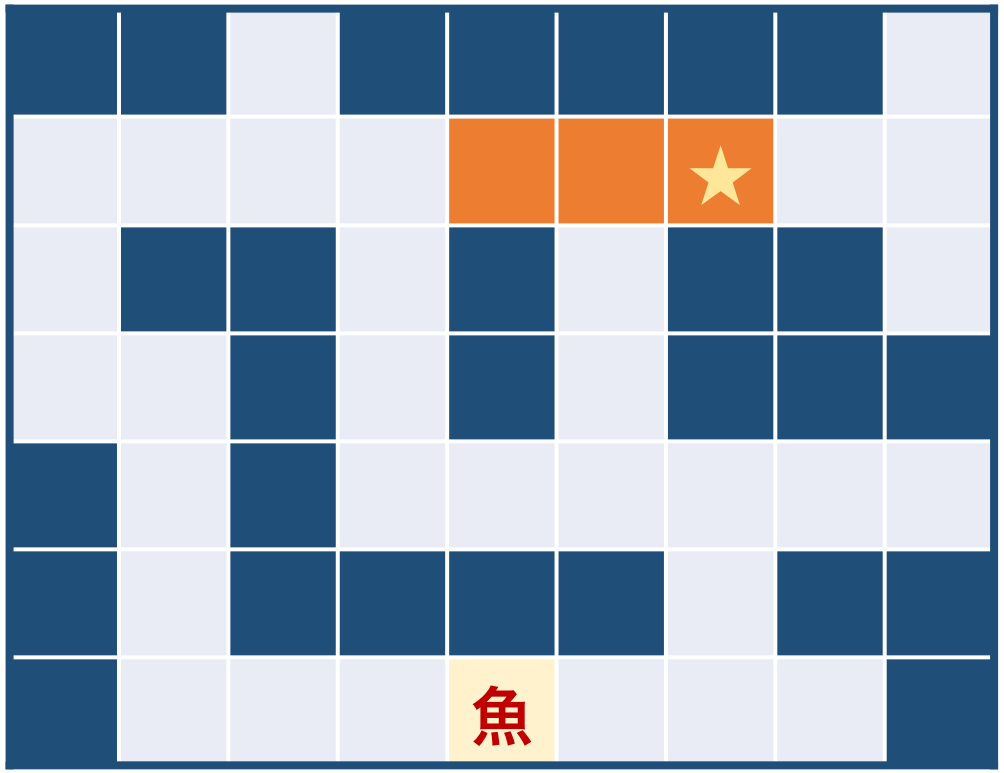


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

右に進んで、
そこを「探索済み」にする

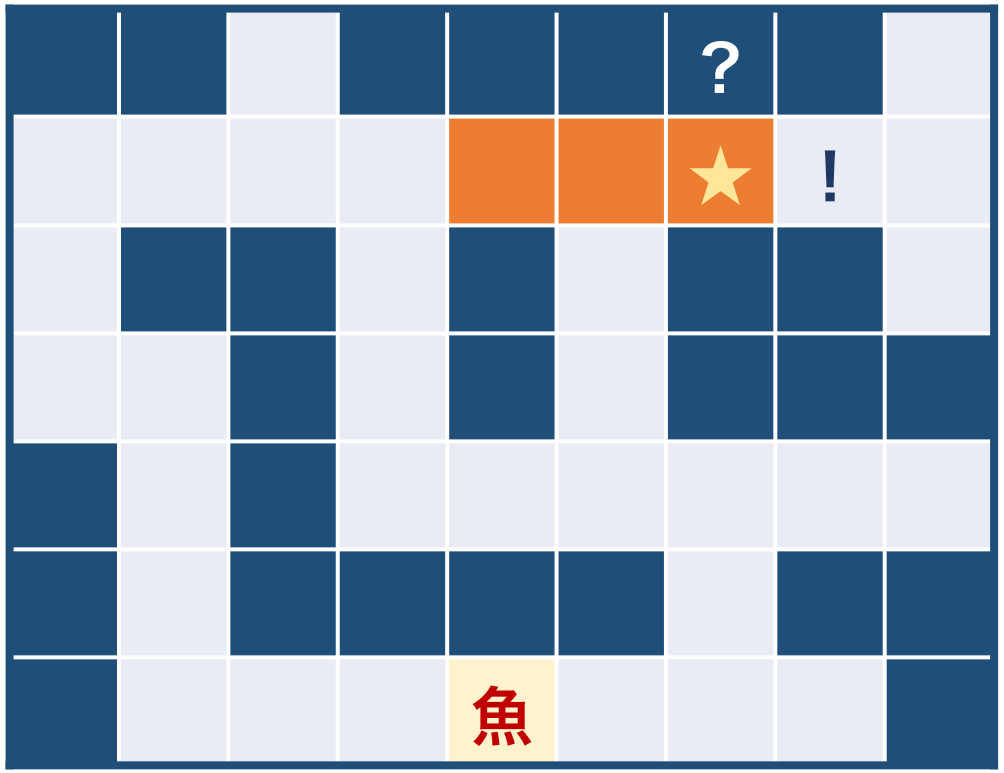


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

同様に進む

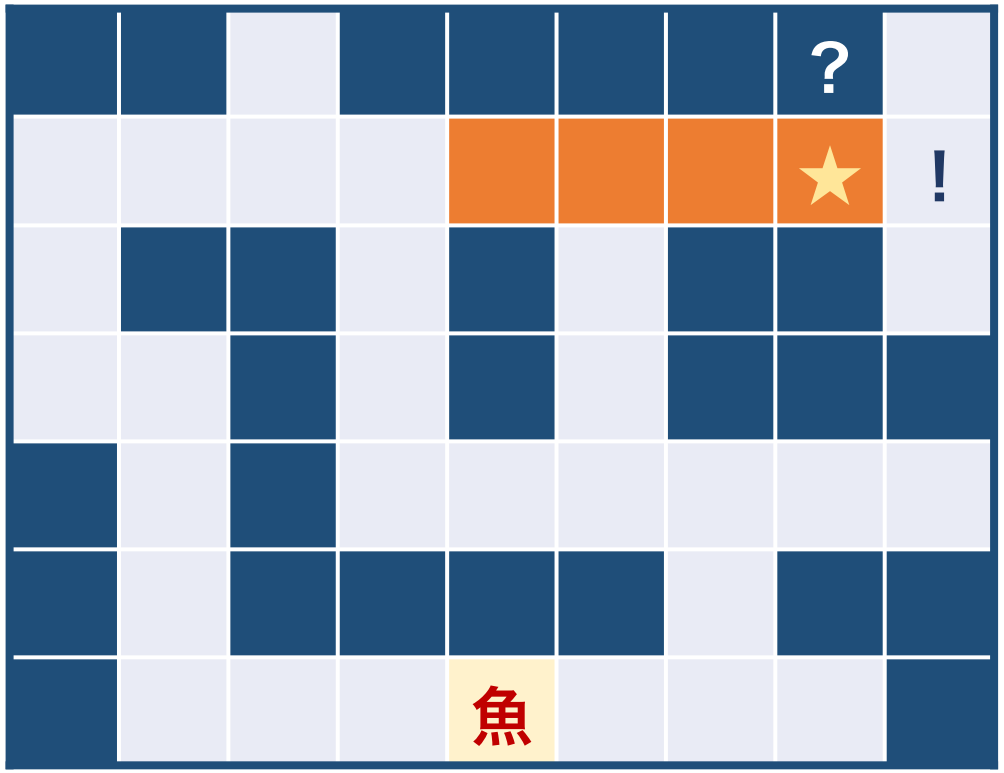


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

同様に進む

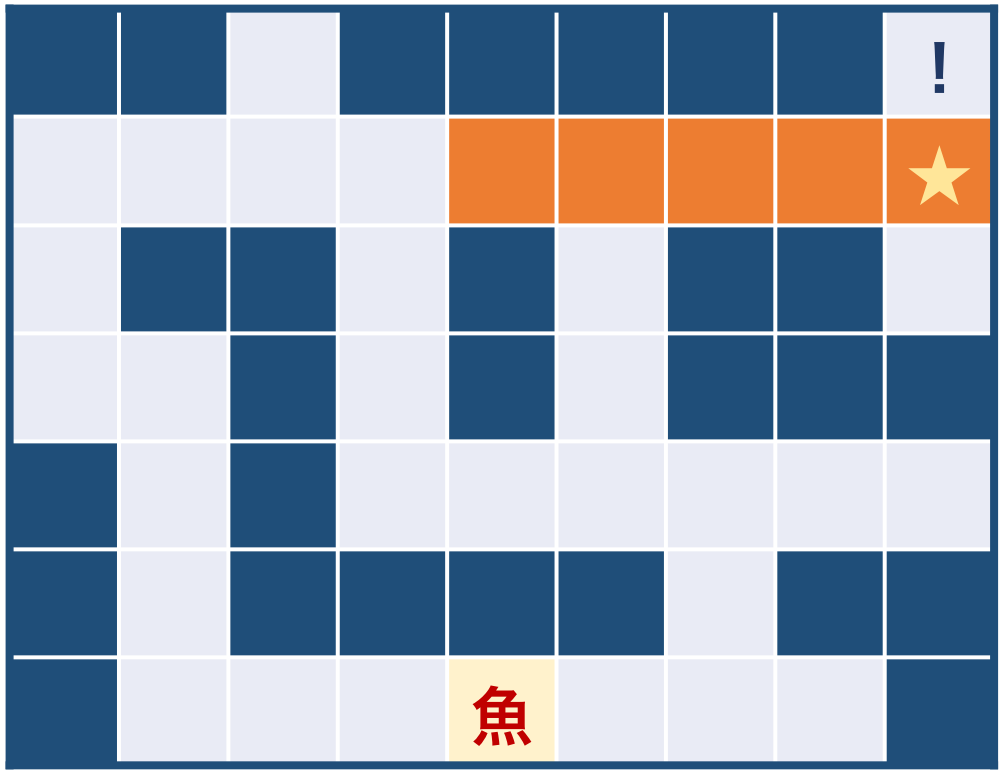


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

同様に進む

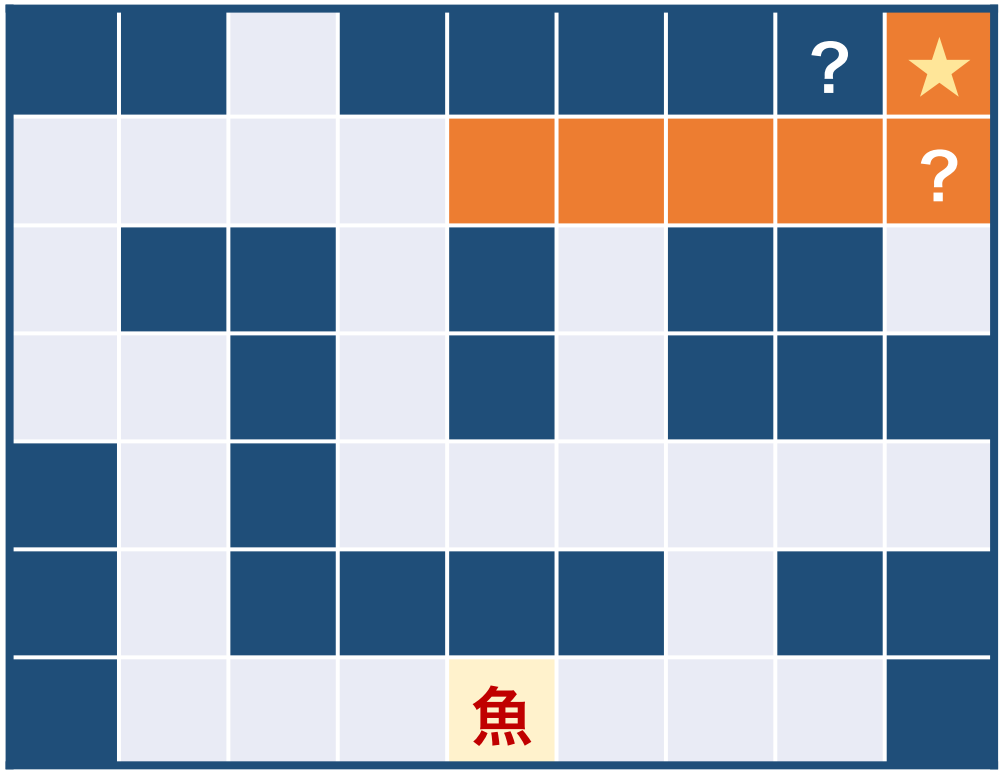


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

4方向どこも行けなくなったら……

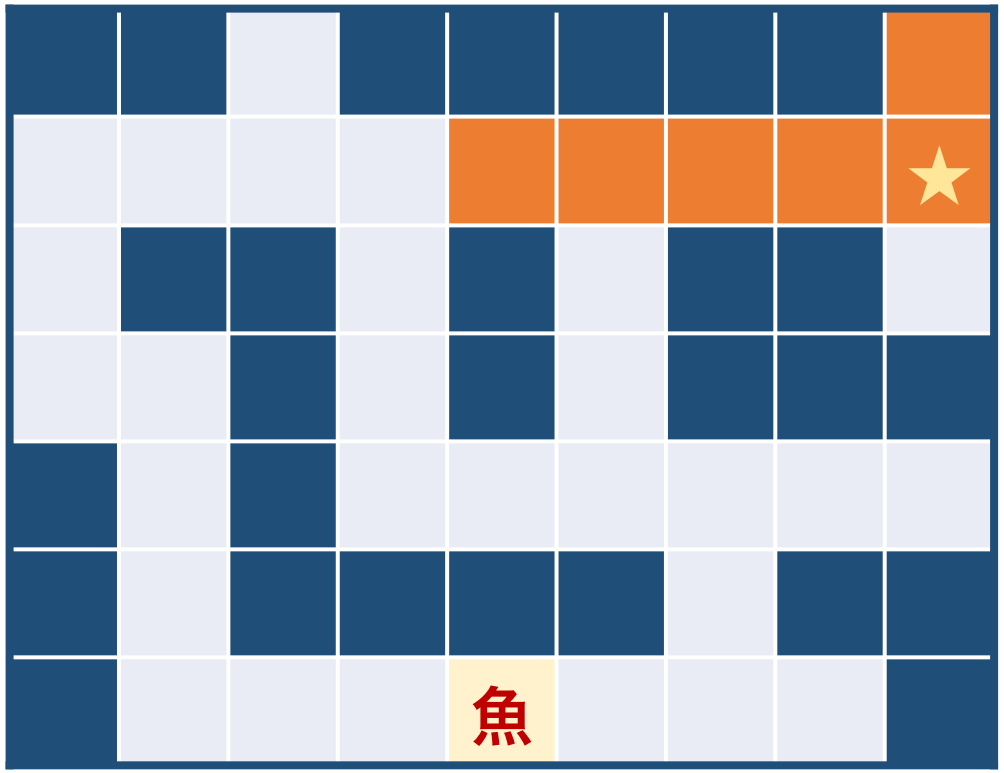


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

一歩戻る

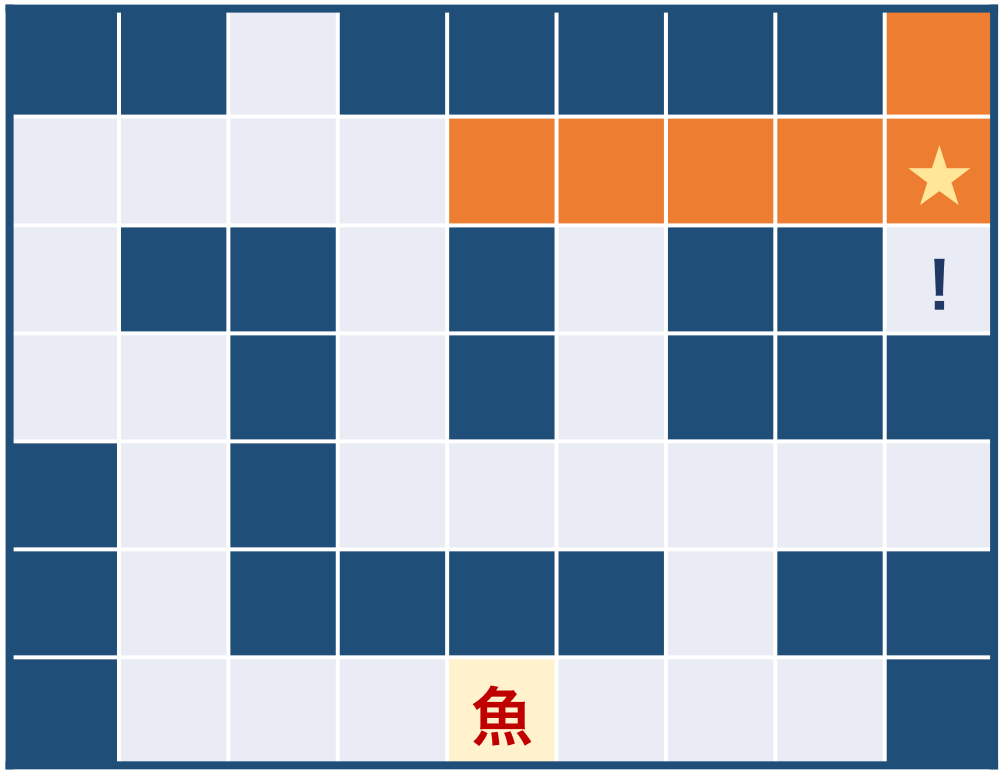


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

この地点で上はもう試したので
右から順に試す
→ 下に行けそう

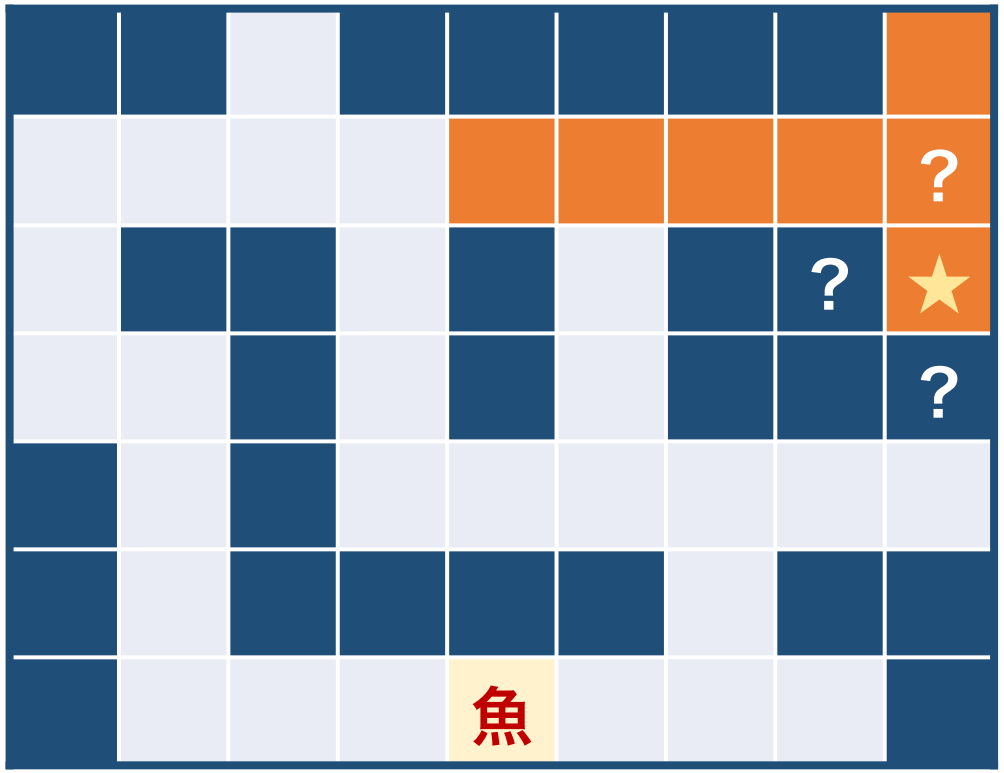


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む → 再び行き止まり

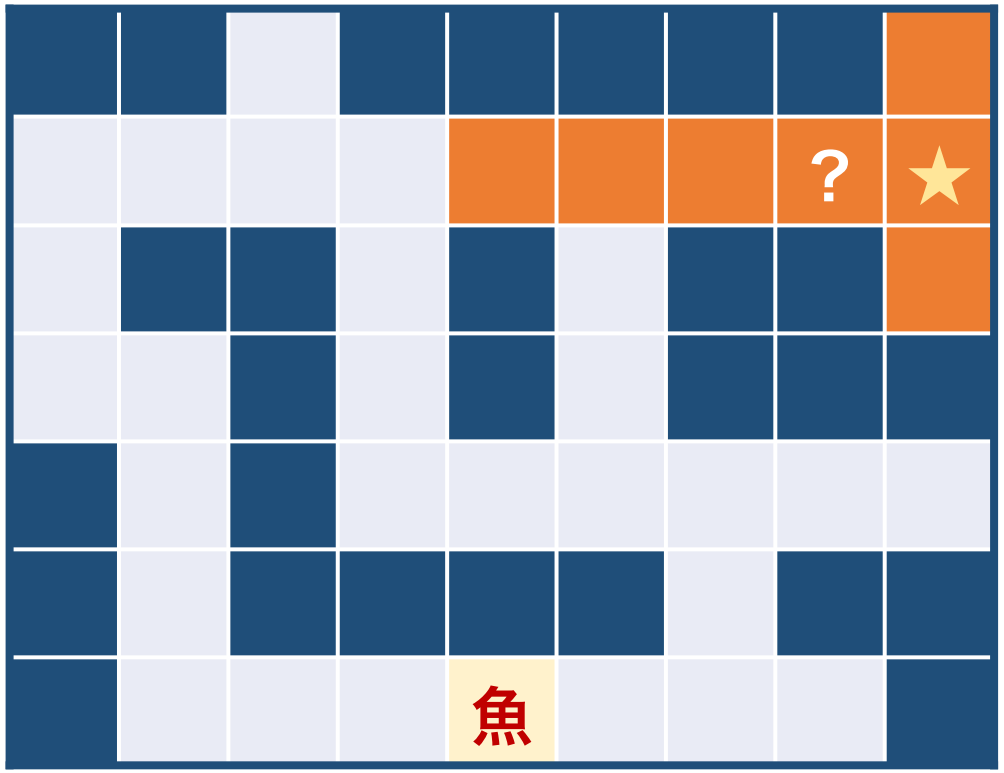


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

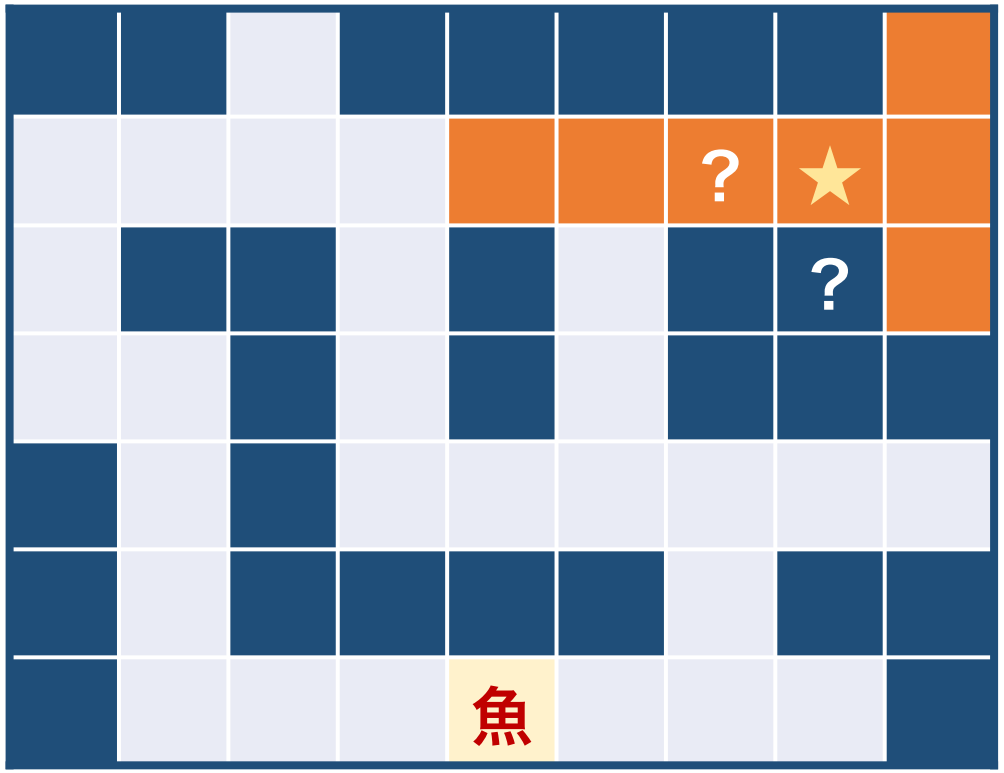


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

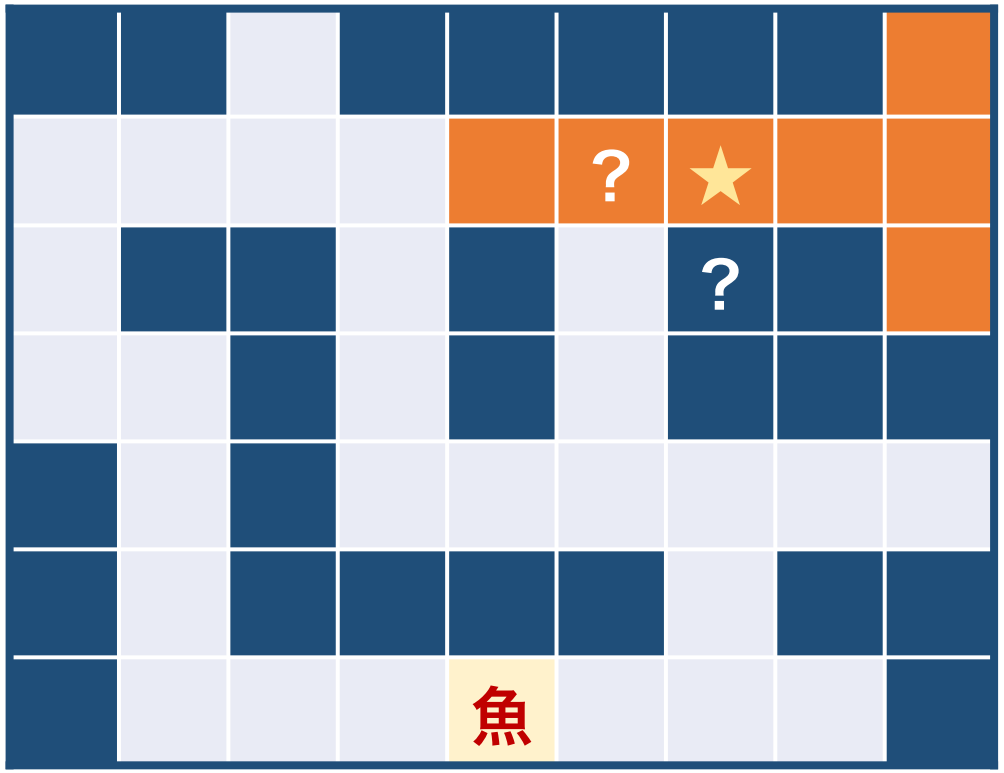


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

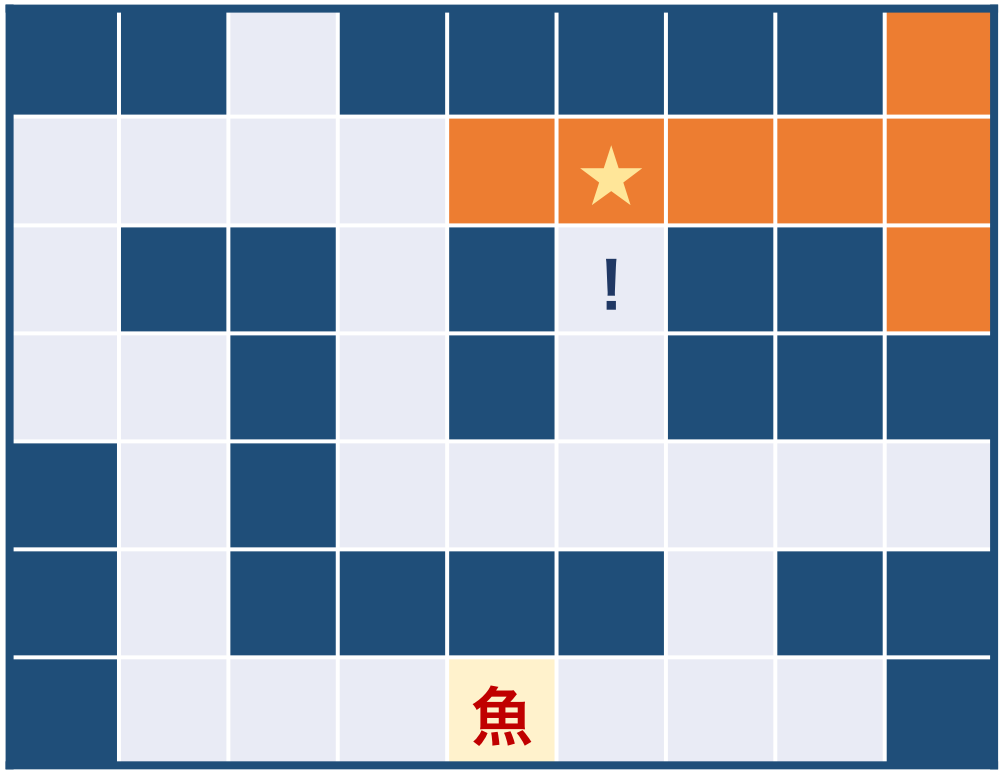


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

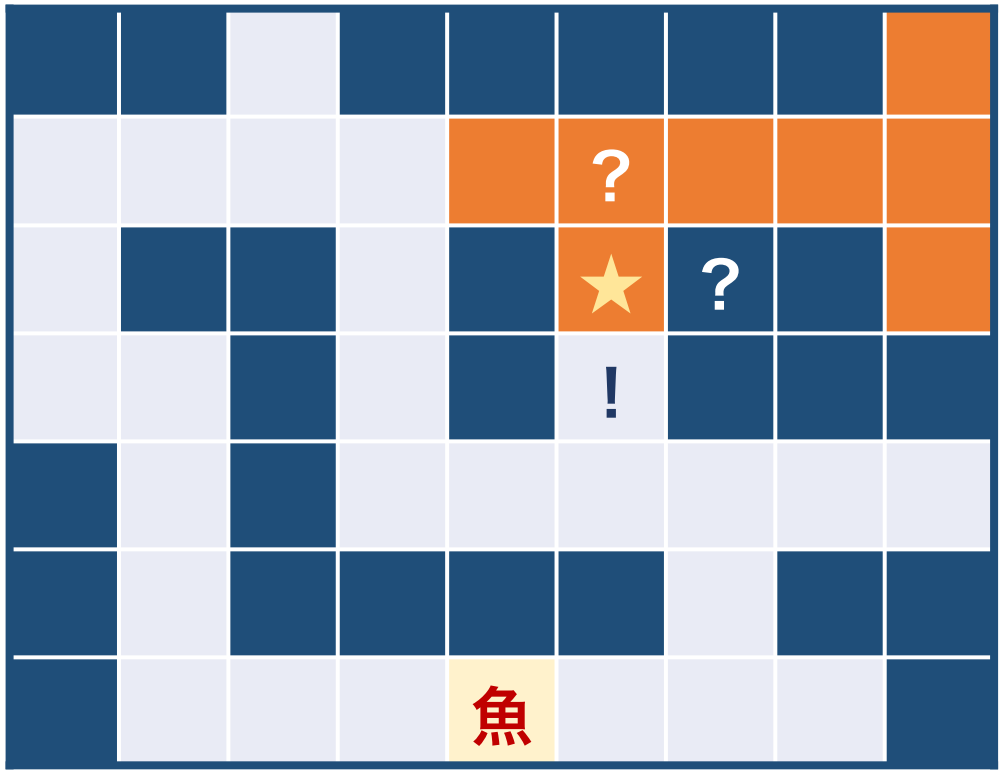


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

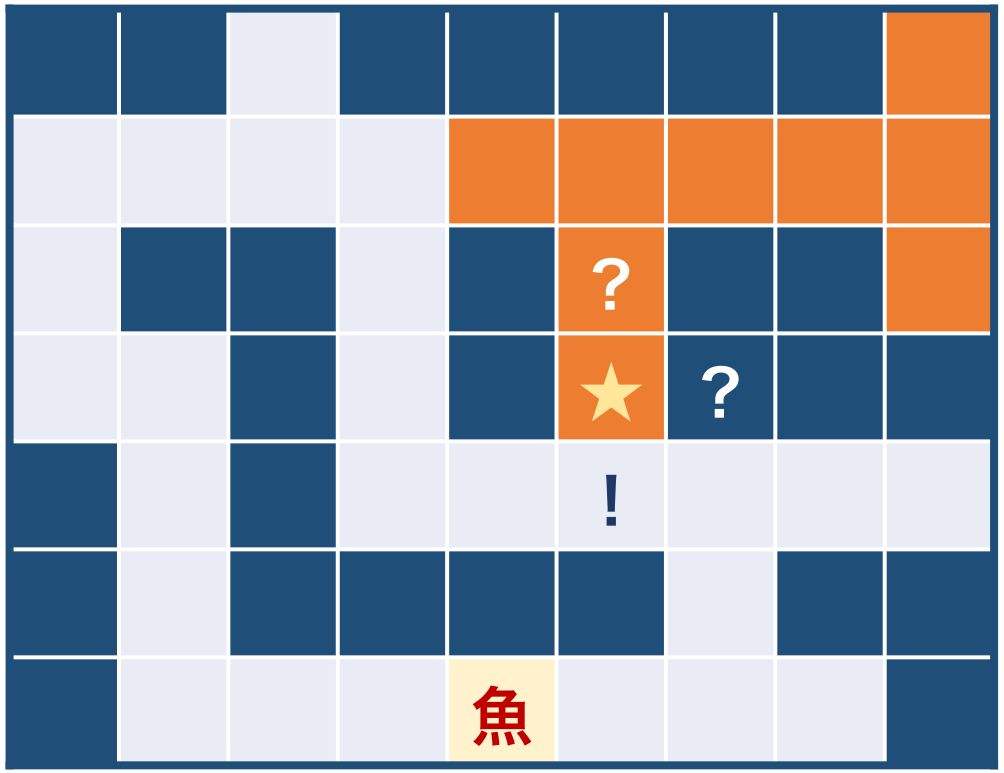


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

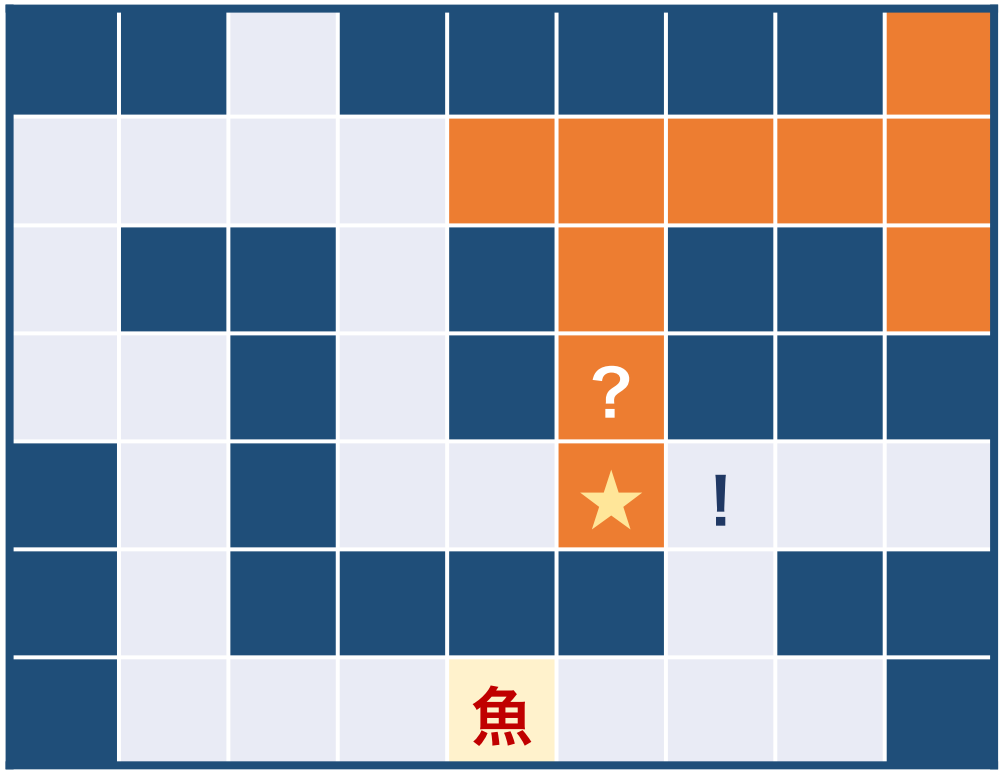


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

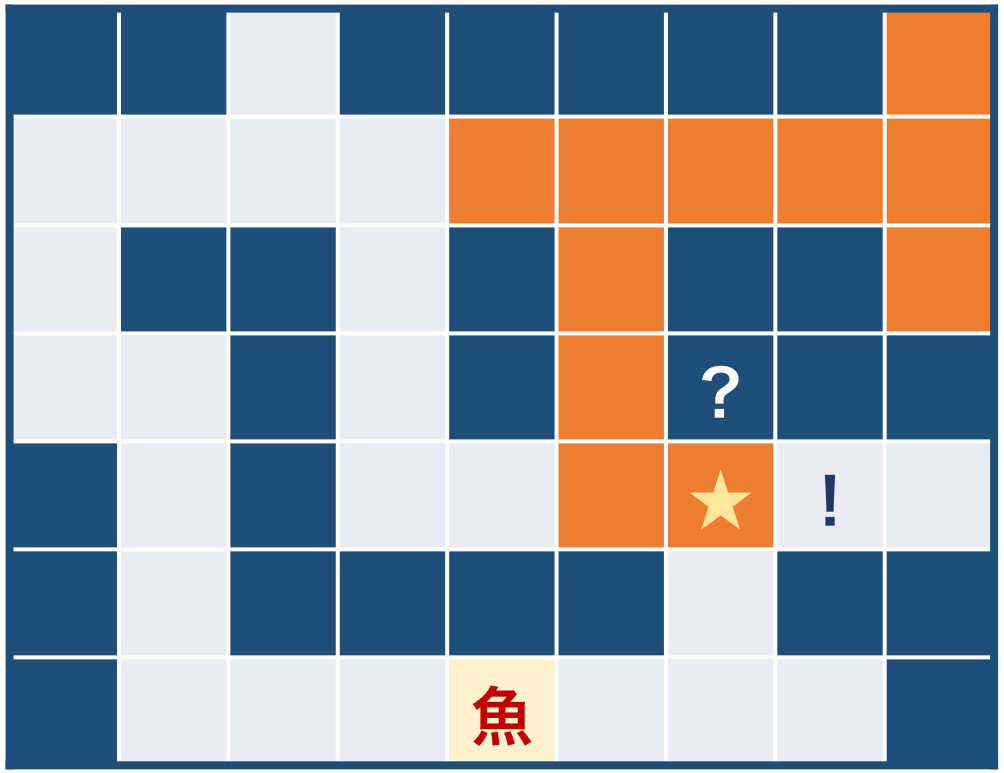


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

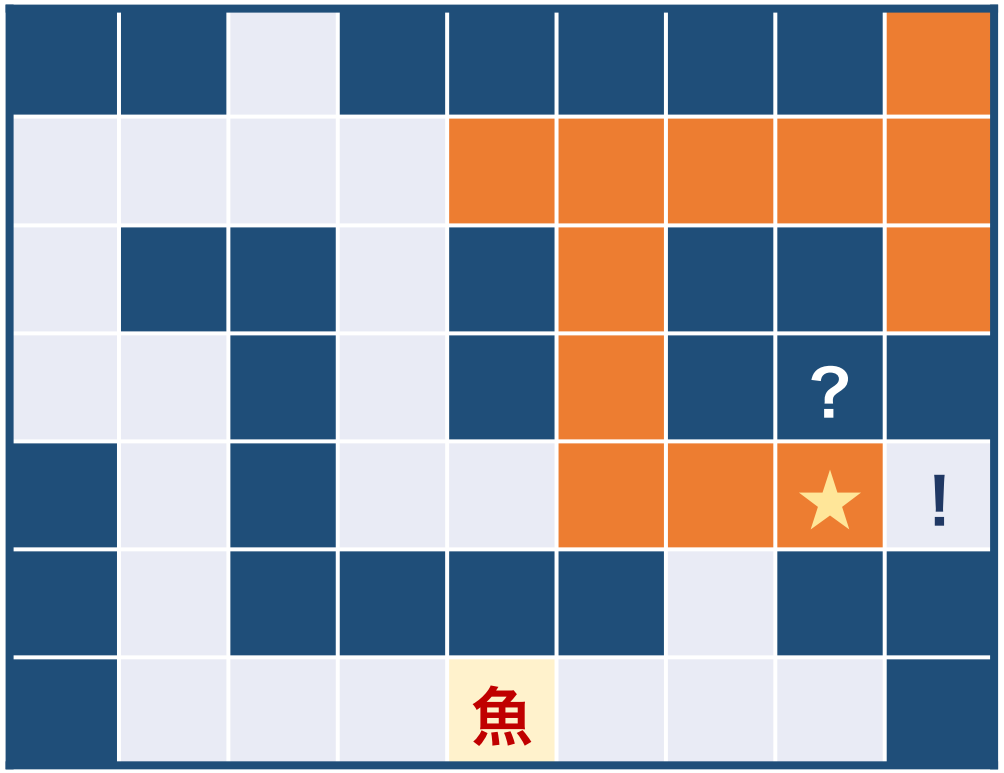


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

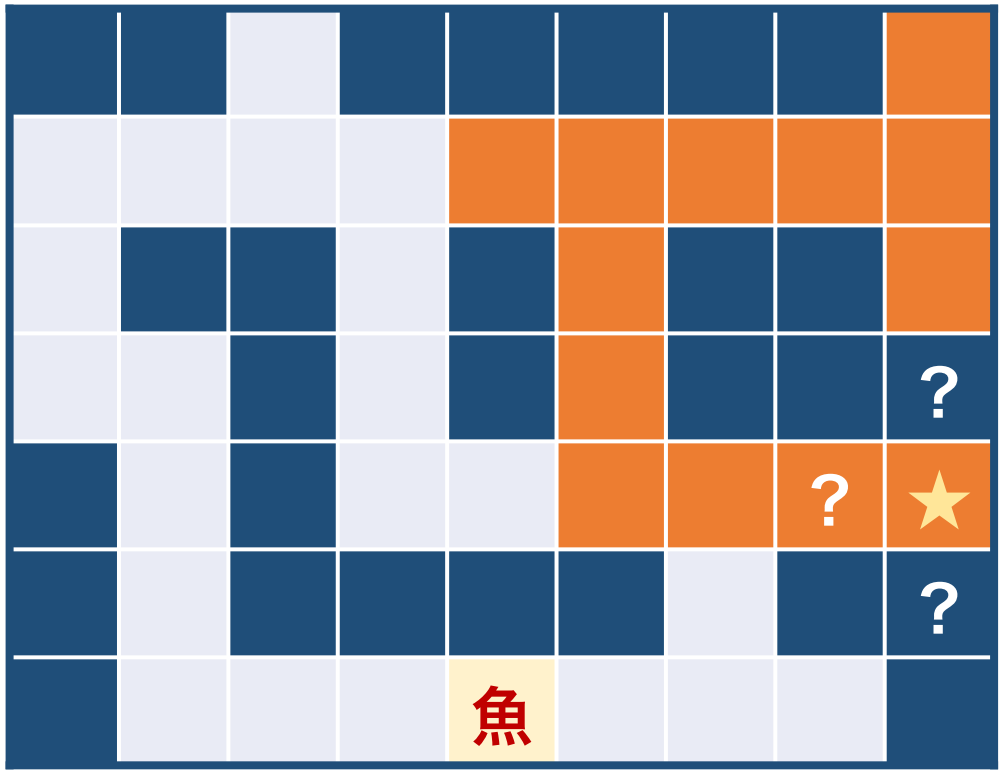


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

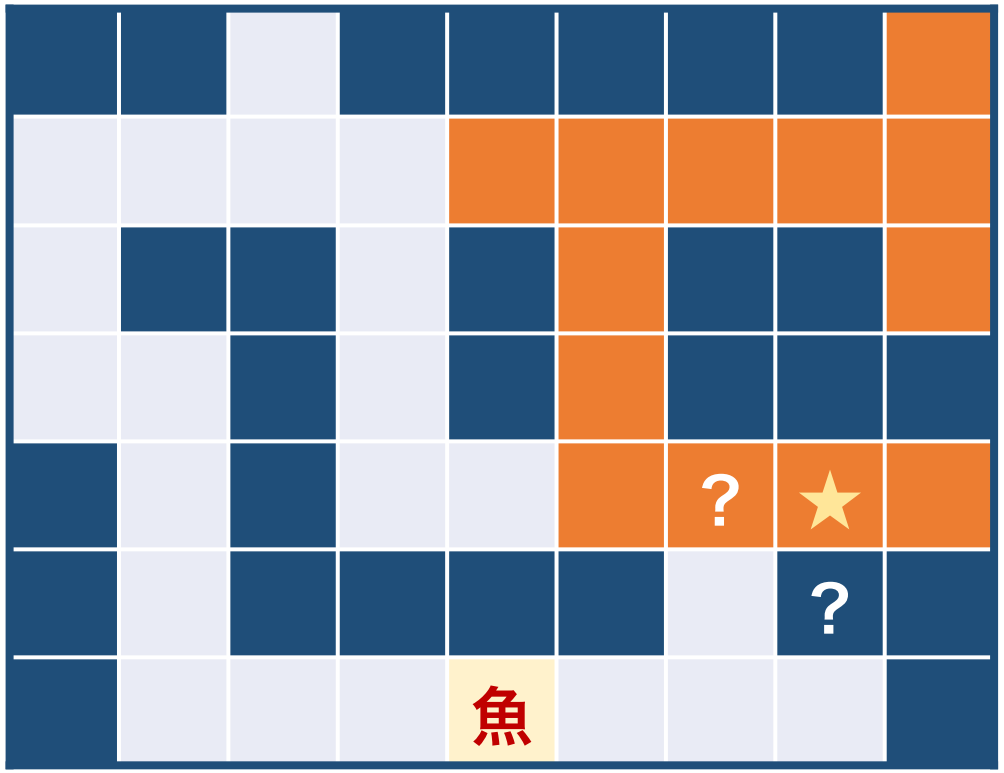


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

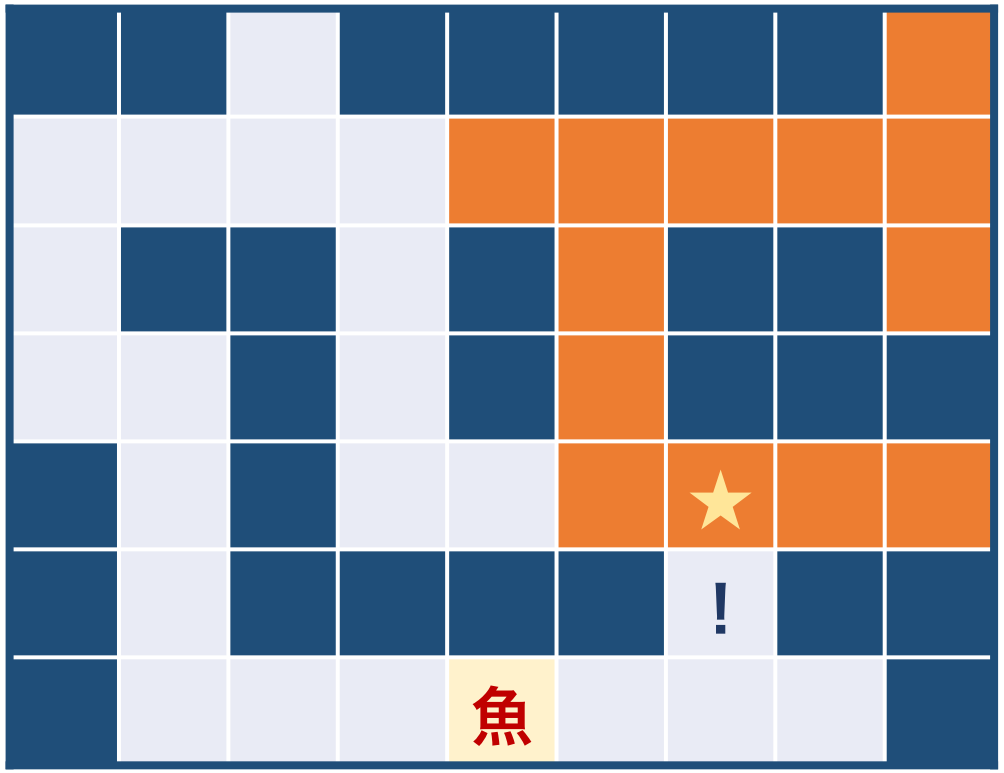


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

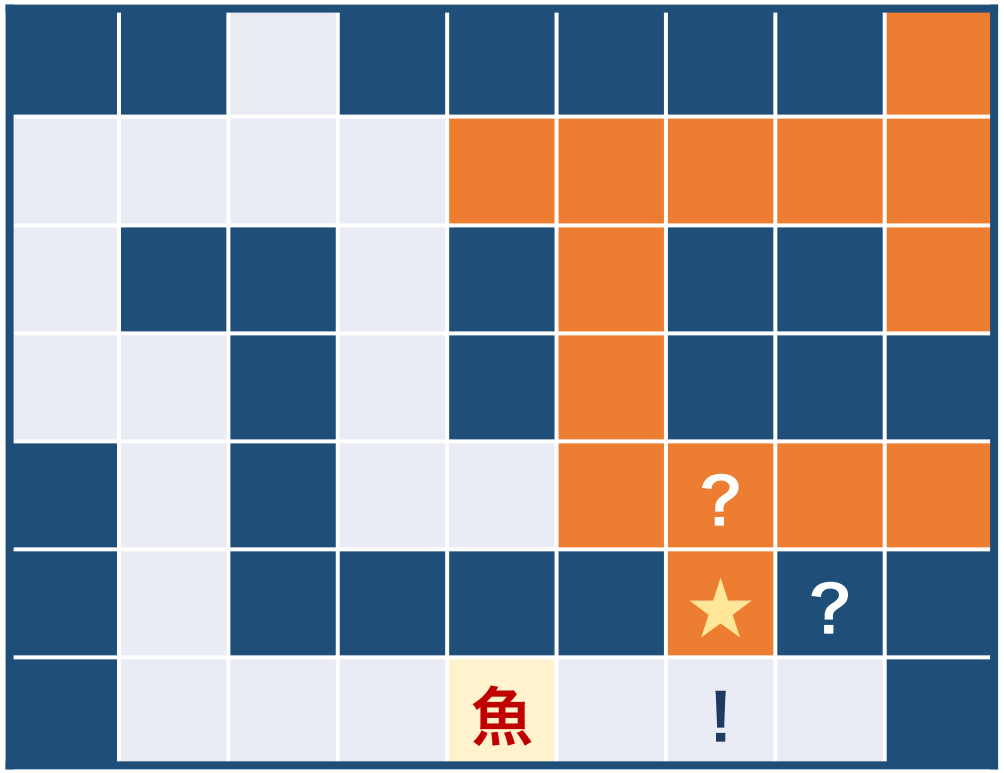


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

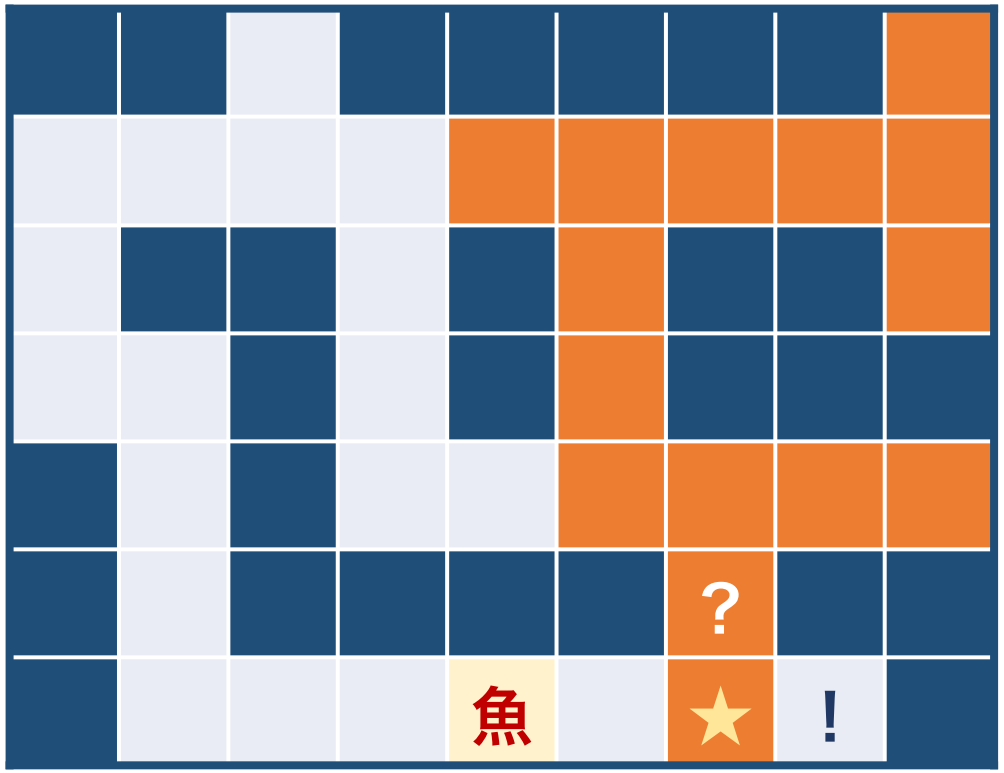


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

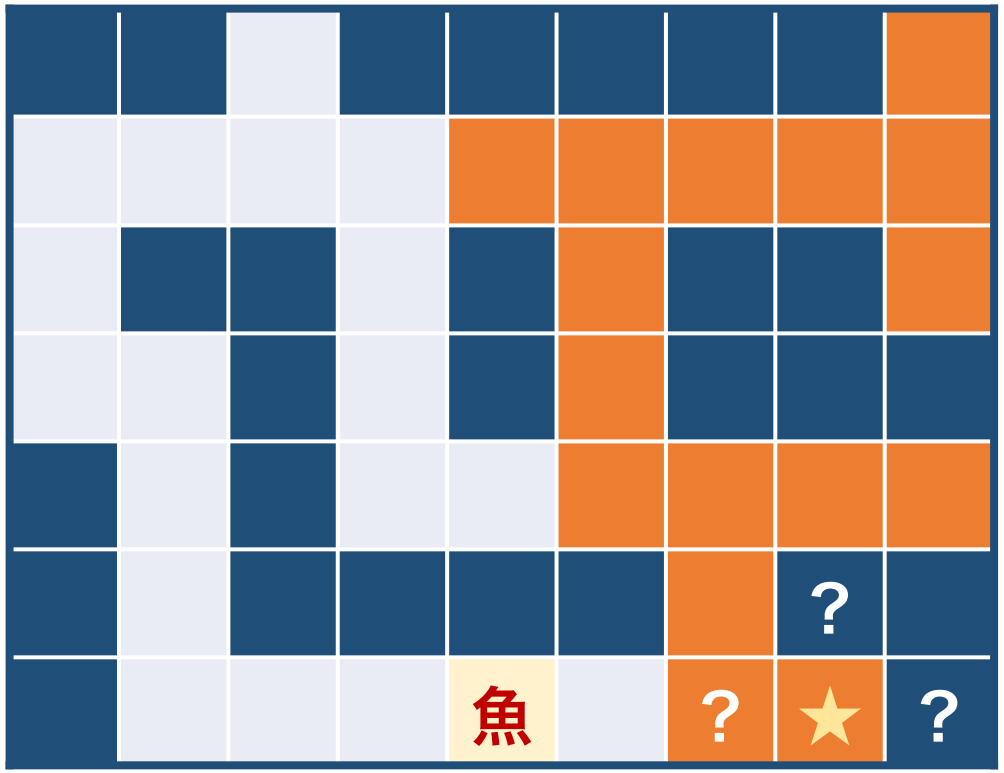


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

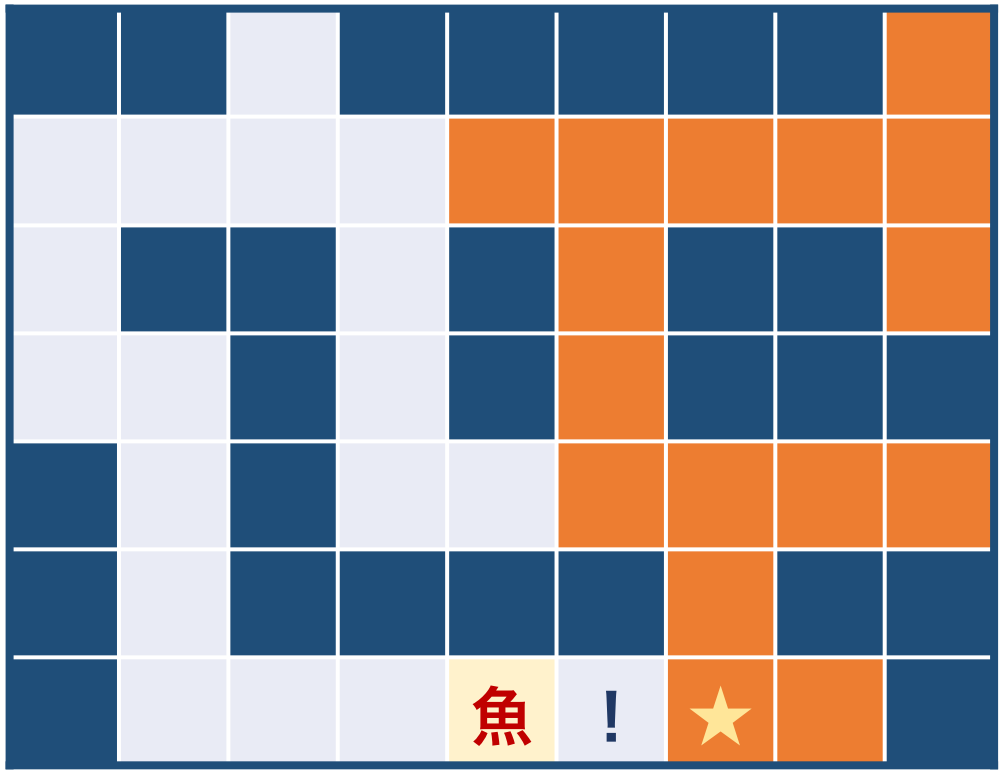


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

戻る

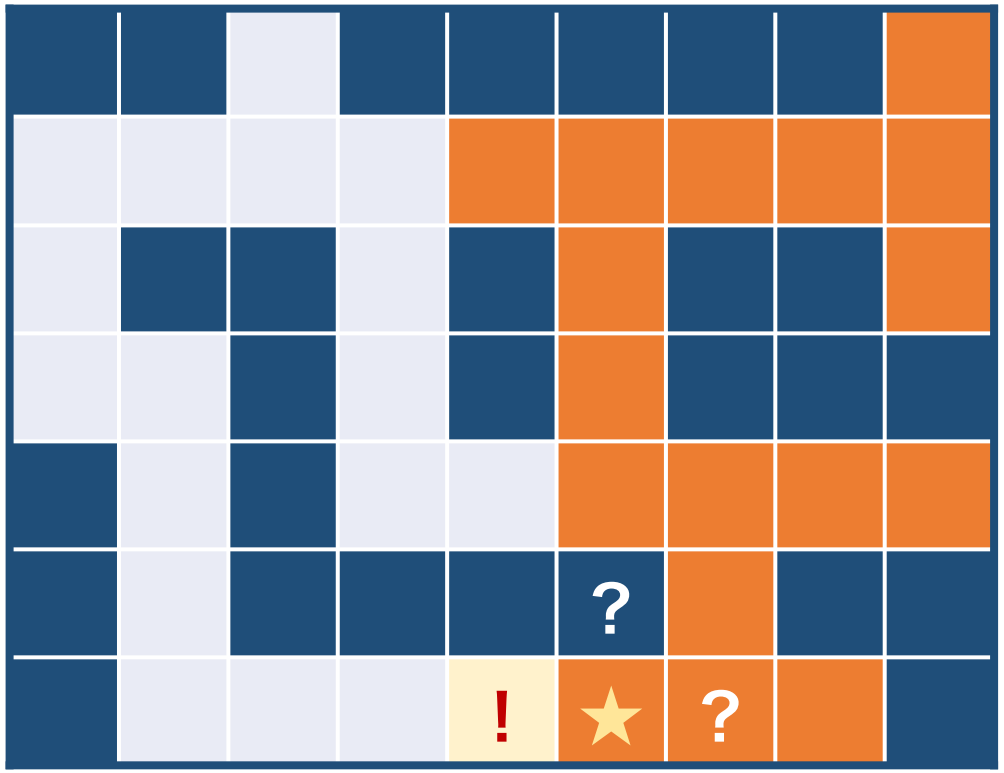


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

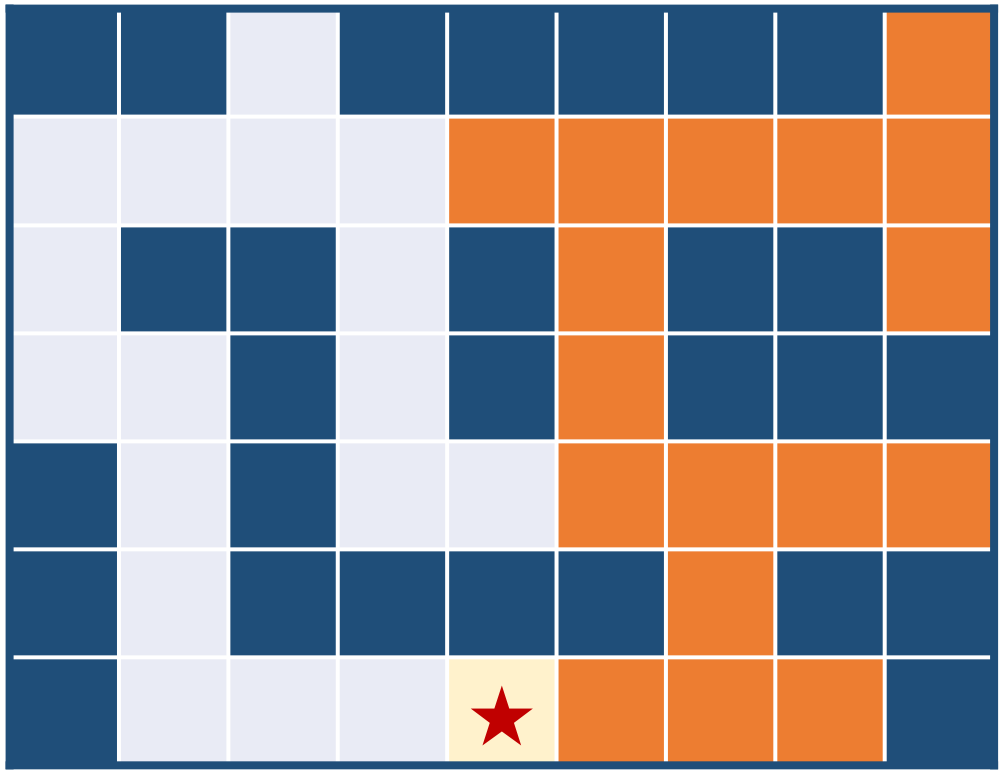


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む

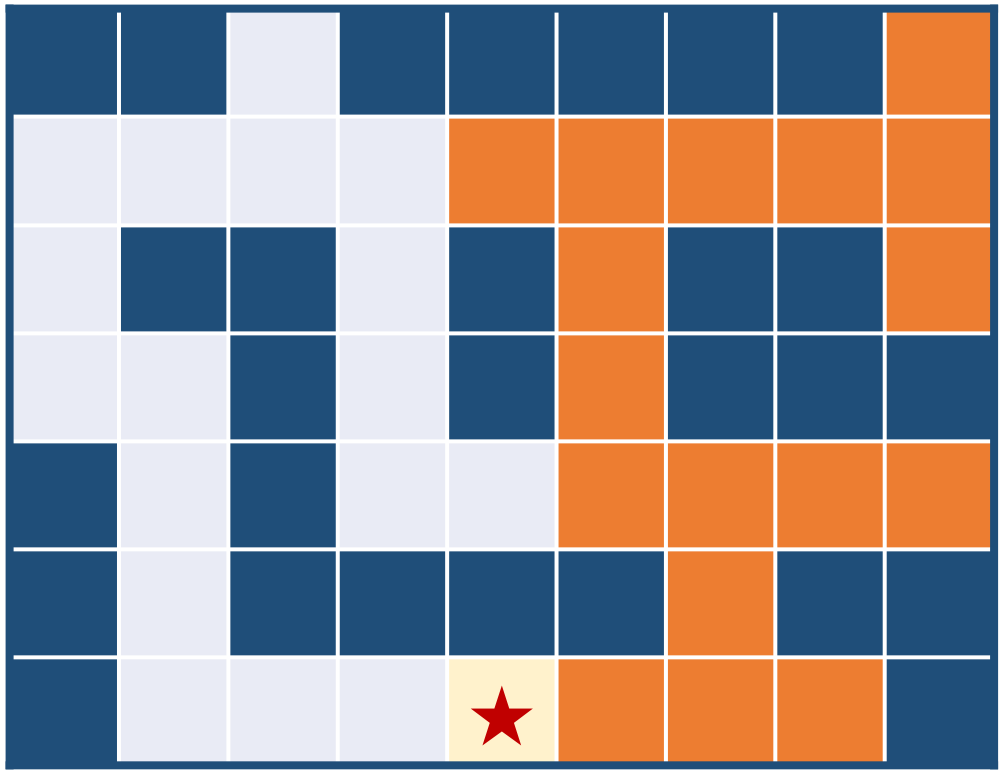


深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の場合

進む → **あ、ここ魚屋じゃん**



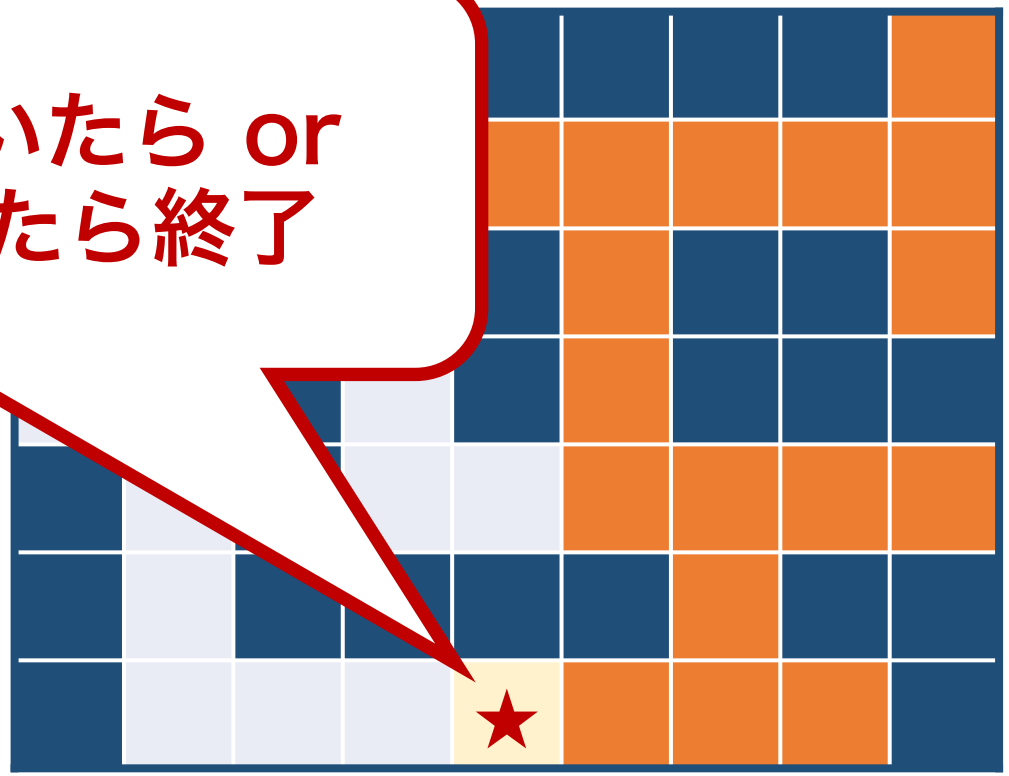
深さ優先探索 (Depth First Search) の動き

方針: ダンジョンの壁沿いにどんどん突き進む！
行き止まりにぶつかったら、
ちょっと戻って別の道を行く！

優先順位
上, 右, 下, 左 の

ゴール地点に着いたら or
全地点を探索したら終了

進む → **あ、ここ魚屋じゃん**



深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

例: a = 2の場合のイメージ

func(2)

“Hello” と出力し、func(1) を実行

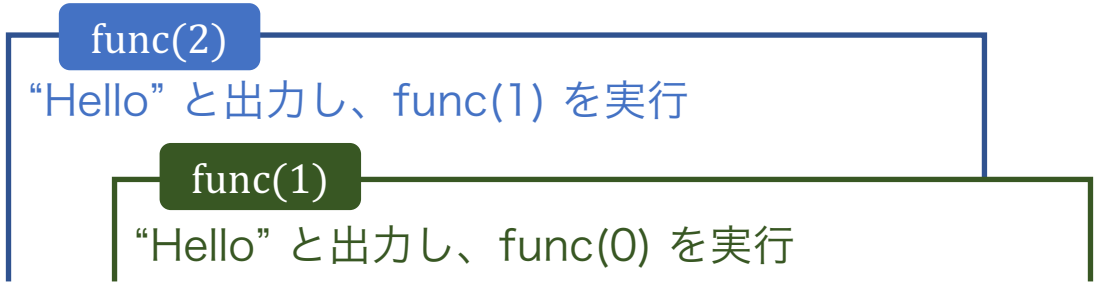
```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

例: a = 2の場合のイメージ




```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

例: a = 2の場合のイメージ

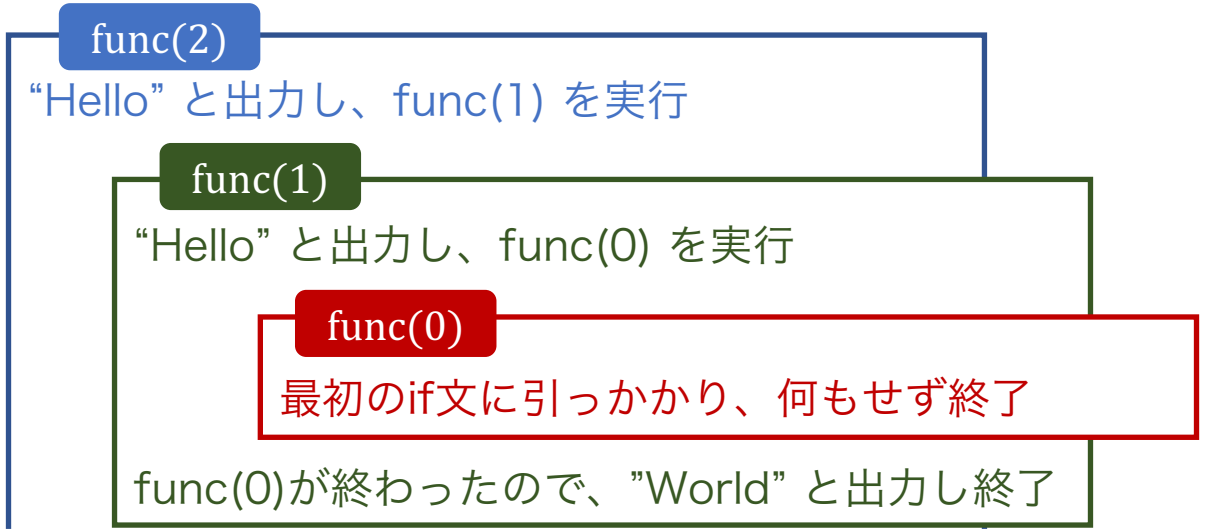

```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

例: a = 2の場合のイメージ




```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

深さ優先探索の実装方法

各地点で同じ処理をやっているが、**進む方向が4つある**
→ 普通のループでは厳しそう
→ 関数の**再帰呼び出し**を使おう！ (関数が自分自身を呼び出す方法)

- まずは、右図の関数の実行イメージをつかもう

例: a = 2の場合のイメージ


```
 4 void func(int a){
 5   if(a == 0){ return; }
 6
 7   cout << "Hello" << endl;
 8   func(a - 1);
 9   cout << "World!" << endl;
10
11   return;
12 }
```

実装例 (1/2)

- main関数部

  入力を受け取るのに二重ループ

  → O(HW)

```
34 int main(void){
35   int h, w;
36   cin >> h >> w;
37
38   char input;
39   int sx, sy, gx, gy;
40   vector<vector<bool> > canGo(h + 2, vector<bool>(w + 2, false));
41   for(int i = 1; i <= h; i++){
42     for(int j = 1; j <= w; j++){
43       cin >> input;
44       if(input != '#'){ canGo[i][j] = true; }
45       if(input == 's'){
46         sx = i;
47         sy = j;
48       }
49       if(input == 'g'){
50         gx = i;
51         gy = j;
52       }
53     }
54   }
55
56   // 深さ優先探索をして結果を出力
57   bool result = dfs(sx, sy, gx, gy, canGo);
58   if(result){ cout << "Yes" << endl; }
59   else{ cout << "No" << endl; }
60
61   return 0;
62 }
```

実装例 (2/2)

- 探索部

この関数が
各地点につき
1回実行される

→ O(HW)

```cpp
#include <iostream>
#include <vector>
using namespace std;

int needleX[4] = {-1, 0, 1, 0};
int needleY[4] = {0, 1, 0, -1};

bool dfs(int x, int y, int goalx, int goaly, vector<vector<bool> >& canGo){
  // ゴールにたどり着いたらtrue
  if(x == goalx && y == goaly){ return true; }

  // 今いる地点を探索済みにする
  canGo[x][y] = false;

  // 自分から見て4方向に着いて
  for(int i = 0; i < 4; i++){
    int next_x = x + needleX[i];
    int next_y = y + needleY[i];

    // 進めそうだったら進む
    if(canGo[next_x][next_y]){
      bool flag = dfs(next_x, next_y, goalx, goaly, canGo);
      //「こっち方向に魚屋あるよー」って返ってきたらtrue
      if(flag)
        return true;
    }
  }

  // ダメだったらfalse
  return false;
}
```

実装例 (2/2)

- 探索部

この関数が
各地点につき
1回実行される

→ O(HW)

合計 O(HW)

```cpp
#include <iostream>
#include <vector>
using namespace std;

int needleX[4] = {-1, 0, 1, 0};
int needleY[4] = {0, 1, 0, -1};

bool dfs(int x, int y, int goalx, int goaly, vector<vector<bool> >& canGo){
  // ゴールにたどり着いたらtrue
  if(x == goalx && y == goaly){ return true; }

  // 今いる地点を探索済みにする
  canGo[x][y] = false;

  // 自分から見て4方向に着いて
  for(int i = 0; i < 4; i++){
    int next_x = x + needleX[i];
    int next_y = y + needleY[i];

    // 進めそうだったら進む
    if(canGo[next_x][next_y]){
      bool flag = dfs(next_x, next_y, goalx, goaly, canGo);
      //「こっち方向に魚屋あるよー」って返ってきたらtrue
      if(flag)
        return true;
    }
  }

  // ダメだったらfalse
  return false;
}
```

実装上のテクニック

- **方位配列 (命名 僕):**
  「自分からみた4方向」をforループで手軽に回せる！

```
int needleX[4] = {-1, 0, 1, 0};
int needleY[4] = {0, 1, 0, -1};
```

- **地図領域を余分に確保:**
  一番外側を壁にすることで配列外アクセスを防げる！

```
vector<vector<bool> > canGo(h + 2, vector<bool>(w + 2, false));
```

- **参照渡し:**
  これをやらないとやばい！

```
vector<vector<bool> >& canGo){
```

実装上のテクニック

- **方位配列 (命名 僕):**
  「自分から
  ```
  int need
  int need
  ```

- **地図領域を**
  一番外側を
  ```
  vector<v                                    + 2, false));
  ```

- **参照渡し:**
  これをやらないとやばい！
  ```
  vector<vector<bool> >& canGo){
  ```

深さ優先探索の仲間に
「幅優先探索」というのもあるので
気になった人はホームページへ！

HCPC活動記録 (最短路・最小全域木)
http://hcpc.web.fc2.com/activities.html

目次

Q3.

2つの自然数x, yが与えられるので、最大公約数を求めよう！

- **Input**
  $x\ y$
- **Constraints**
  $1 \leq x, y \leq 10^9$
- **Time Limit**
  1 sec

Q3.

2つの自然数x, yが与えられるので、最大公約数を求めよう！

- **Input**
  $x\ y$
- **Constraints**
  $1 \leq x, y \leq 10^9$
- **Time Limit**
  1 sec

1つずつ順番に
「これxとyの公約数？」って
試していけばいいのでは？

Q3.

2つの自然数x, yが与えられるので、最大公約数を求めよう！

- **Input**
  x y
- **Constraints**
  $1 \leq x, y \leq 10^9$
- **Time Limit**
  1 sec

1つずつ順番に
「これxとyの公約数？」って
試していけばいいのでは？

**線形探索じゃ
間に合わない！**

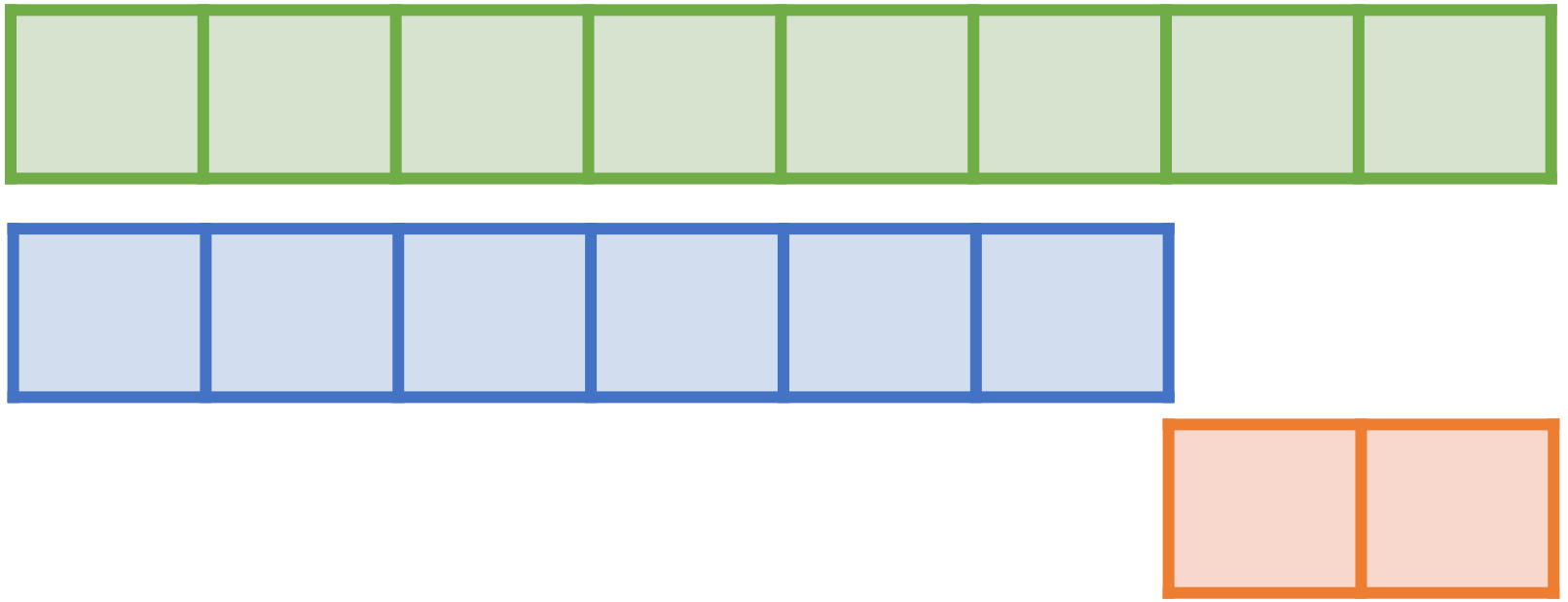
公約数の特徴

- xとyの公約数は、必ずx%yの約数になっている (逆も然り)

x=8

y=6

x%y=2



→ つまり、xとyの最大公約数は、
yとx%yの最大公約数に一致する！

ユークリッドの互除法 (Euclidean Algorithm)

「aとbの最大公約数は
bとa%bの最大公約数である」

を、**そのまま再帰で書くだけ**

(bが0になったときのaが答え)

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int gcd(int a, int b){
 5   if(b == 0){ return a; }
 6   return gcd(b, a % b);
 7 }
 8
 9 int main(void){
10   int x, y;
11   cin >> x >> y;
12   cout << gcd(x, y) << endl;
13   return 0;
14 }
```

ユークリッドの互除法 (Euclidean Algorithm)

「aとbの最大公約数は
bとa%bの最大公約数である」

を、**そのまま再帰で書くだけ**

(bが0になったときのaが答え)

Q. 時間計算量はいくつになるの？

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int gcd(int a, int b){
 5   if(b == 0){ return a; }
 6   return gcd(b, a % b);
 7 }
 8
 9 int main(void){
10   int x, y;
11   cin >> x >> y;
12   cout << gcd(x, y) << endl;
13   return 0;
14 }
```

ユークリッドの互除法 (Euclidean Algorithm)

「aとbの最大公約数は
bとa%bの最大公約数である」

を、**そのまま再帰で書くだけ**

(bが0になったときのaが答え)

Q. 時間計算量はいくつになるの？

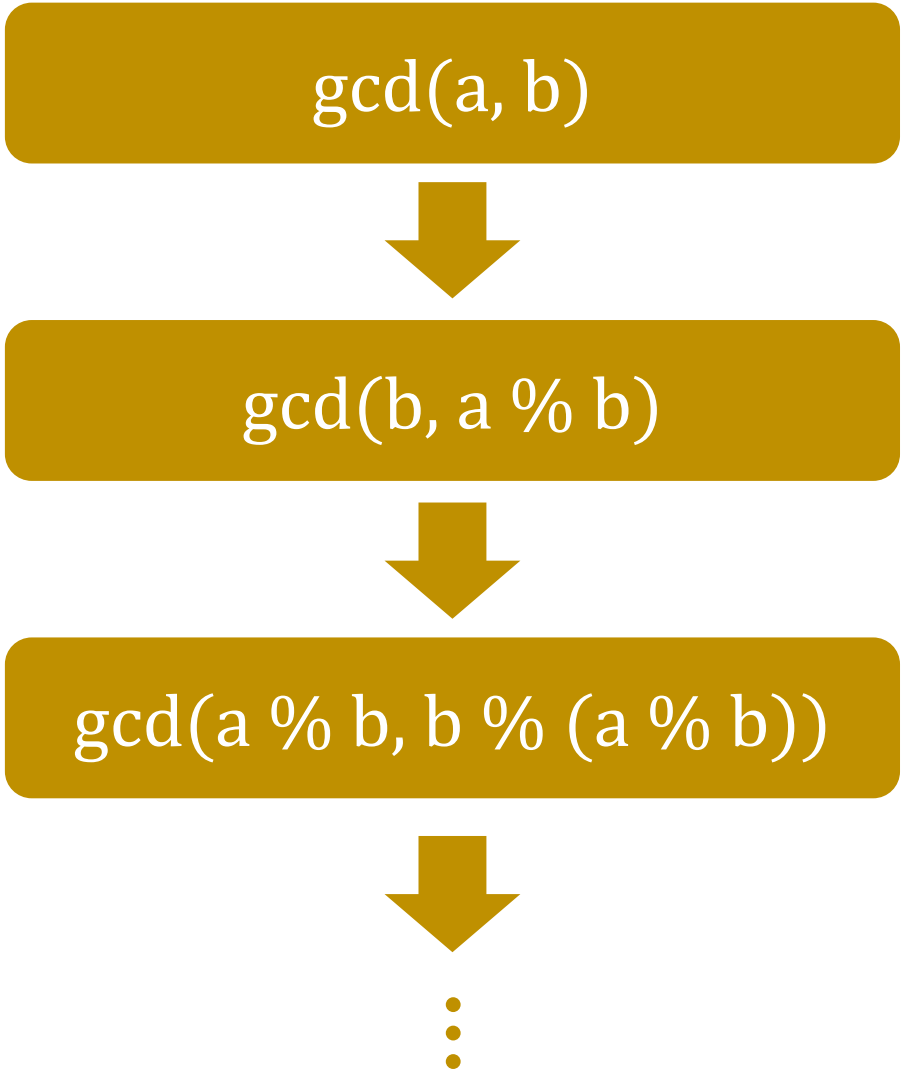
ちょっと解析してみよう

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int gcd(int a, int b){
 5   if(b == 0){ return a; }
 6   return gcd(b, a % b);
 7 }
 8
 9 int main(void){
10   int x, y;
11   cin >> x >> y;
12   cout << gcd(x, y) << endl;
13   return 0;
14 }
```

ユークリッドの互除法の解析

ここで着目すべき性質はコレ

$$a\%b < \frac{a}{2}$$

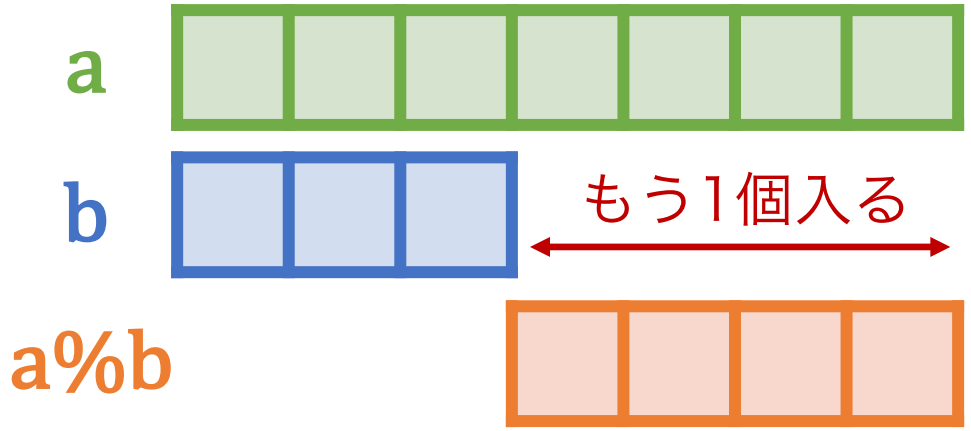


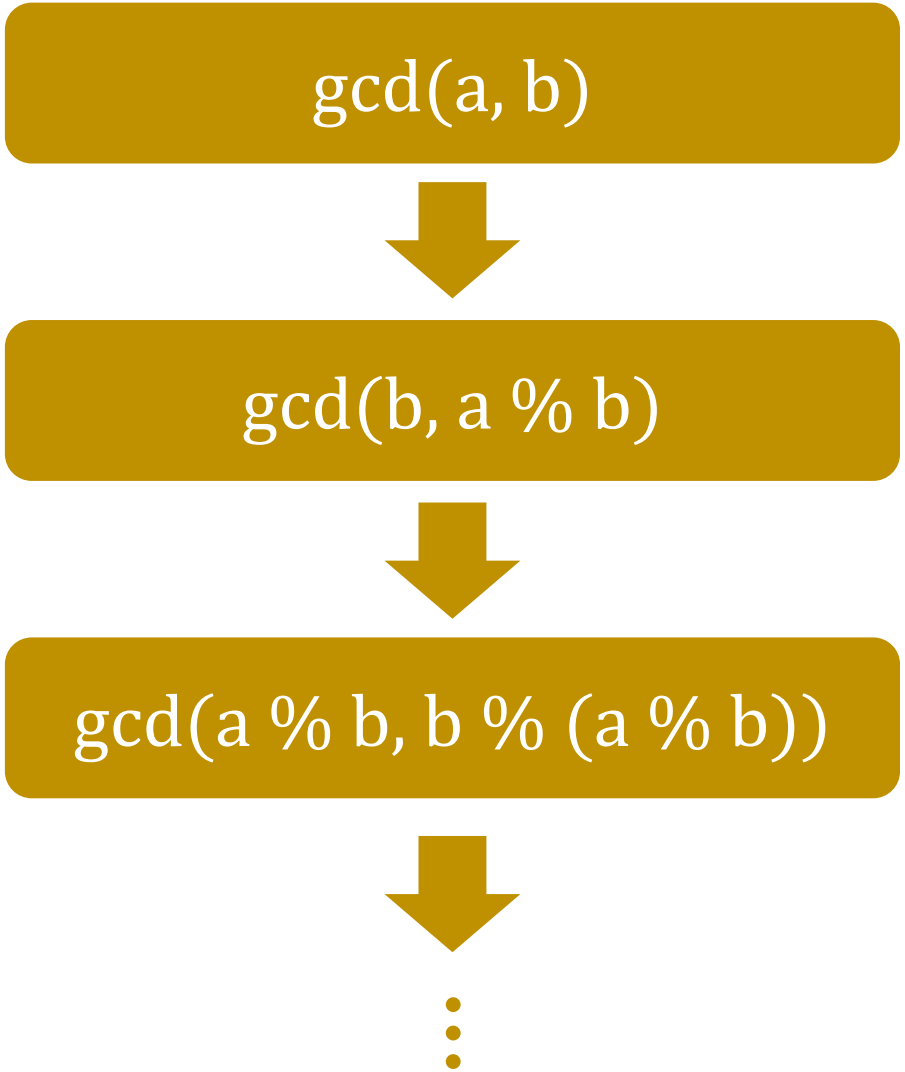
ユークリッドの互除法の解析

ここで着目すべき性質はコレ

$$a\%b < \frac{a}{2}$$

- 仮にこれが満たされなかったら……





ユークリッドの互除法の解析

ここで着目すべき性質はコレ

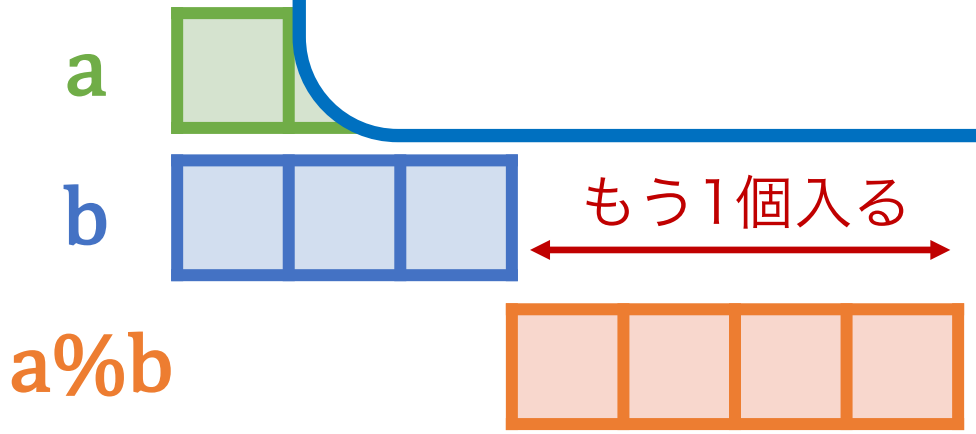
gcd(a, b)

a%b

再帰を2段階進むごとに
引数の値が半分以下に

→ 関数の呼び出し回数は高々2loga程度

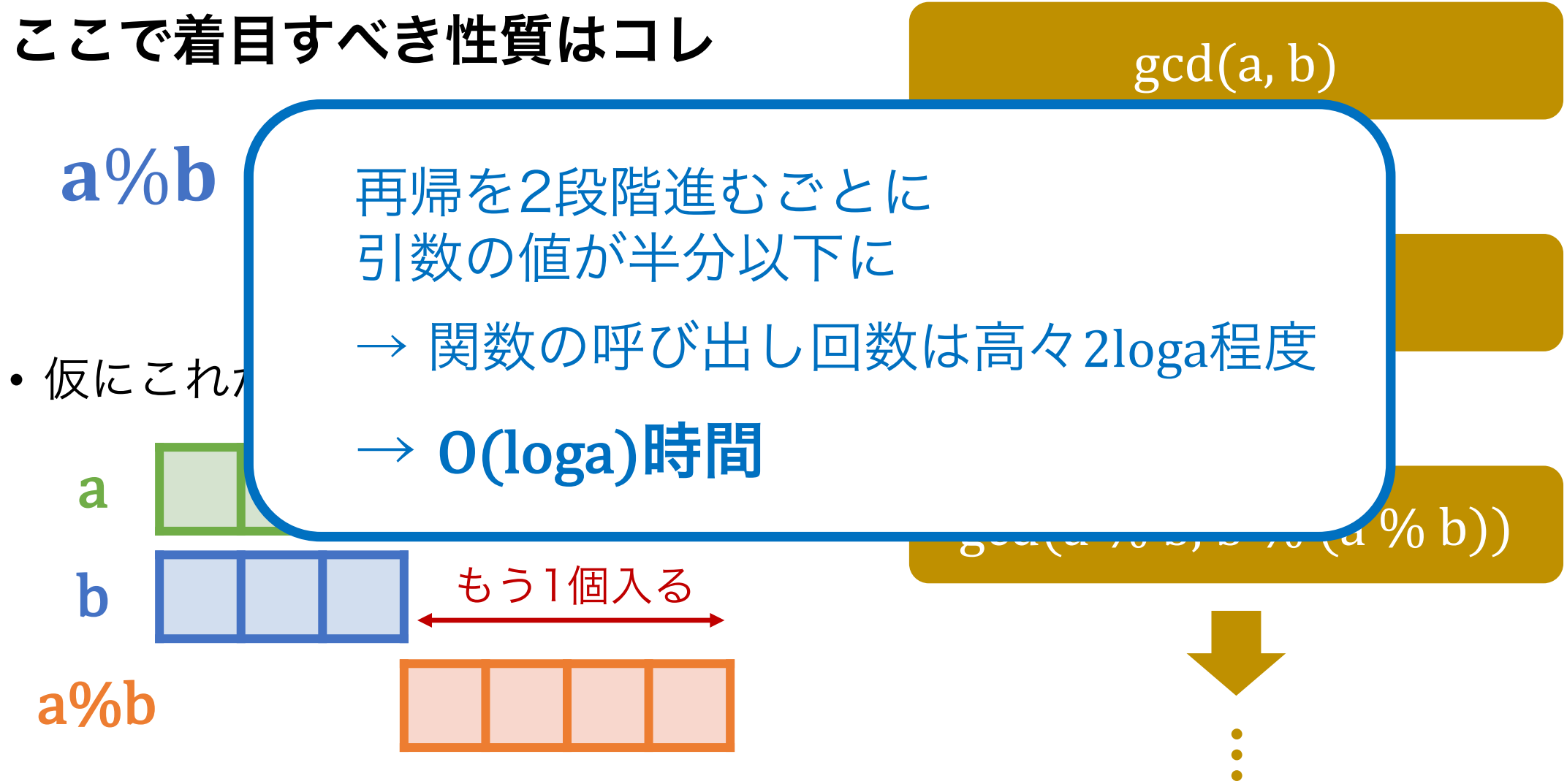
- 仮にこれ



gcd(a % b, b % (a % b))

ユークリッドの互除法の解析

ここで着目すべき性質はコレ



ところで、最小公倍数は？

ところで、最小公倍数は？

- xとyの最大公約数をGCD(x, y)
  xとyの最小公倍数をLCM(x, y) とすると

LCM(x, y)GCD(x, y) = xy という性質がある

導出のポイント: x=nGCD(x, y), y=mGCD(x, y) と書いたとき
nとmは互いに素である

ところで、最小公倍数は？

- xとyの最大公約数をGCD(x, y)
  xとyの最小公倍数をLCM(x, y) とすると

$LCM(x, y)GCD(x, y) = xy$ という性質がある

導出のポイント: $x=nGCD(x, y)$, $y=mGCD(x, y)$ と書いたとき
nとmは互いに素である

→ **最大公約数さえ求まれば、あとはO(1)で求まる！**

Q4.

n個の自然数が与えられるので、
そのうち素数だったものの個数を求めよう！

- **Input**

  1行目 N
  2行目 1番目の自然数 a_1
  ⋮
  N+1行目 N番目の自然数 a_N

- **Constraints**

  $1 \leq \mathrm{N} \leq 10^6$
  $1 \leq a_i \leq 10^6$

- **Time Limit**

  1 sec

普通の素数判定

- 普通の素数判定は、
  線形探索で $O(\sqrt{a})$ でできる
  - a=bc という合成数だった場合
    bとcの一方は必ず$\sqrt{a}$以下

```
 4 bool isPrim(int a){
 5   if(a <= 1){ return false; }
 6
 7   for(int i = 2; i * i <= a; i++)
 8     if(a % i == 0)
 9       return false;
10
11   return true;
12 }
```

普通の素数判定

- 普通の素数判定は、
  線形探索で $O(\sqrt{a})$ でできる
  - a=bc という合成数だった場合
    bとcの一方は必ず$\sqrt{a}$以下

```
 4 bool isPrim(int a){
 5   if(a <= 1){ return false; }
 6
 7   for(int i = 2; i * i <= a; i++)
 8     if(a % i == 0)
 9       return false;
10
11   return true;
12 }
```

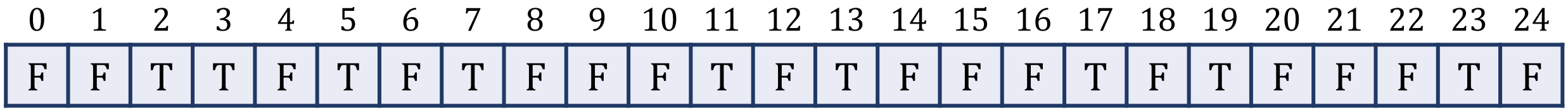
今回はこれをN回やる必要があるので $O(N\sqrt{a})$

およそ10^9回の計算 → 厳しい

もっと高速なアルゴリズムが必要

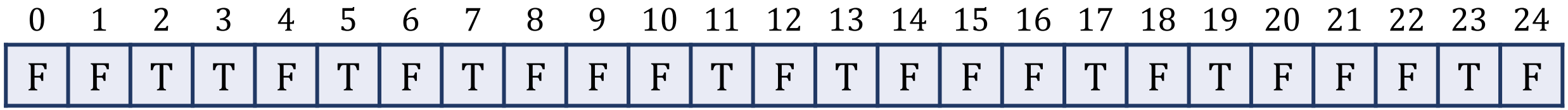
エラトステネスの篩 (Sieve of Eratosthenes)

- **方針**: あらかじめ**「素数かどうかテーブル」**を作っておけば、
  あとはそのテーブルを O(1) で参照すればいい



エラトステネスの篩 (Sieve of Eratosthenes)

- **方針**: あらかじめ**「素数かどうかテーブル」**を作っておけば、
  あとはそのテーブルを O(1) で参照すればいい



とはいえ、このテーブルが高速に作れなければ意味がない

エラトステネスの篩 (Sieve of Eratosthenes)

- **方針**: あらかじめ**「素数かどうかテーブル」**を作っておけば、
  あとはそのテーブルを $O(1)$ で参照すればいい

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F

とはいえ、このテーブルが高速に作れなければ意味がない

ポイント: a以下の合成数は必ず$\sqrt{a}$以下の約数を持つ

エラトステネスの篩 (Sieve of Eratosthenes)

- **方針**: あらかじめ**「素数かどうかテーブル」**を作っておけば、
  あとはそのテーブルを O(1) で参照すればいい

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F

とはいえ、このテーブルが高速に作れなければ意味がない

ポイント: a以下の合成数は必ず$\sqrt{a}$以下の約数を持つ

→ **$\sqrt{a}$以下の全自然数の倍数をfalseにしておけばいい**

テーブル構築の動き

- $a \leq 80$ の場合

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

まず1だけfalseに初期化

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

次に2をチェック → 素数らしい

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

2の倍数をすべてfalseに

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

次に3をチェック → 素数らしい

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

3の倍数をすべてfalseに

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

4は素数じゃないらしいのでスルー

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

5は素数らしいので、その倍数を全部false

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

6はスルー

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

7は素数らしいので、その倍数を全部false

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

8はスルー

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

テーブル構築の動き

- $a \leq 80$ の場合

$\sqrt{80} < 9$ なので終了

(青背景: true / 黄背景: false)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

実装例

- テーブルの構築時間は
  O(alogloga)時間
  であることが知られている
  - 調和級数の性質を用いると
    O(aloga) 以下であることは
    簡単に証明できる
  - 素数の逆数和の発散速度の
    性質を用いると
    O(alogloga) が証明できるらしい

    (誰かこの証明理解したら教えてね)

```cpp
#include <iostream>
#include <vector>
using namespace std;

#define MAX_N (1e6)+1

int main(void){
  // テーブルを作る
  vector<bool> eratos(MAX_N, true);
  eratos[0] = eratos[1] = false;
  for(int i = 2; i * i < MAX_N; i++)
    if(eratos[i])
      for(int j = 2; i * j < MAX_N; j++)
        eratos[i * j] = false;

  // クエリ処理
  int n, input, ans = 0;
  cin >> n;
  for(int i = 0; i < n; i++){
    cin >> input;
    if(eratos[input])
      ++ans;
  }

  // 出力
  cout << ans << endl;
  return 0;
}
```

目次

Q5.

1円玉・5円玉・10円玉・50円玉・100円玉・500円玉が、
それぞれc_1枚・c_5枚・c_{10}枚・c_{50}枚・c_{100}枚・c_{500}枚ある

なるべく少ない枚数の硬貨でA円のものを買いたい！
最低何枚の硬貨で買い物ができるか

- **Input**
  A
  $c_1 \ c_5 \ c_{10} \ c_{50} \ c_{100} \ c_{500}$
- **Constraints**
  $1 \leq \mathrm{A},\ c_i \leq 10^9$
- **Time Limit**
  $1\ \mathrm{sec}$

Q5.

1円玉・5円玉・10円玉・50円玉・100円玉・500円玉が、
それぞれc枚・枚・枚・枚・枚・枚ある

なるべく
最低何枚

500円から順番に
使えるだけ使っていけばいい！

- **Input**
  A
  c_1 c_5
- **Cons**
  $1 \leq A$
- **Time Limit**
  1 sec

Q5.

1円玉・5円玉・10円玉・50円玉・100円玉・500円玉が、
それぞれc枚・枚・枚・枚・枚・枚ある

なるべく
最低何枚

- **Input**
  A
  c_1 c_5
- **Cons**
  $1 \leq A$
- **Time Limit**
  1 sec

500円から順番に
使えるだけ使っていけばいい！

この手法を 貪欲法 とよぶ

実装例

- **時間計算量: O(1)**

正確には
O(硬貨の種類数)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

#define N 6

int main(void){
  // 入力を受け取る
  int a;
  cin >> a;
  vector<int> c(N);
  for(int i = 0; i < N; i++)
    cin >> c[i];

  // 各コインについてループ
  int ans = 0;
  int coin[6] = {1, 5, 10, 50, 100, 500};
  for(int i = N - 1; i >= 0; i--){
    // コインを使えるだけ使う
    int cnt = min(a / coin[i], c[i]);
    a -= cnt * coin[i];
    ans += cnt;
  }

  // 結果を出力
  cout << ans << endl;
  return 0;
}
```

目次

Q6.

H×5の盤面に石が埋め込まれて、以下の手順で進行する

1. 同じ色の石が3つ以上水平に並んでいたら、その石は消える
2. 石が消滅したマスの上に石があれば、空きを埋めるように石が落ちる
3. すべての石が落ちきった後、1の条件を満たす石があれば1に戻る

最終的に、消滅した石の数字の合計がスコアになる
パズルの盤面が与えられるので、得られるスコアを求めよう！

- **Input**
  1行目 行数 H
  2行目 行1の石 $s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,4}$ $s_{1,5}$
  :
  H+1行目 行Hの石 $s_{H,1}$ $s_{H,2}$ $s_{H,3}$ $s_{H,4}$ $s_{H,5}$

- **Constraints**
  $1 \leq \mathrm{H} \leq 10$
  $1 \leq s_{i,j} \leq 9$

- **Time Limit**
  8 sec

Q6.

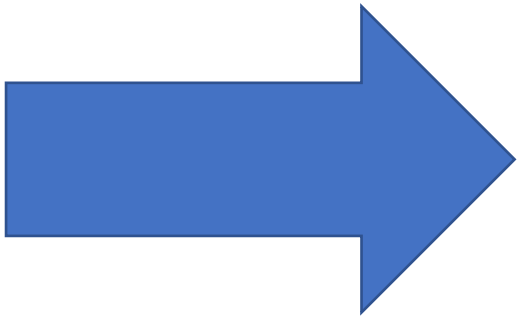
- Sample Input

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

Q6.

- Sample Input

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```



5	9	5	5	9
5	5	6	9	9
4	6	3	6	9
3	3	2	9	9
2	2	1	1	1

スコア: 0

Q6.

- **Sample Input**

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

5	9	5	5	9
5	5	6	9	9
4	6	3	6	9
3	3	2	9	9
2	2	1	1	1

スコア: 0

Q6.

- **Sample Input**

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

5	9			
5	5	5	5	9
4	6	6	9	9
3	3	3	6	9
2	2	2	9	9

スコア: 3

Q6.

- Sample Input

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

5	9			
5	**5**	**5**	**5**	9
4	6	6	9	9
3	**3**	**3**	6	9
2	**2**	**2**	9	9

スコア: 3

Q6.

- Sample Input

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

				9
			9	9
5	9		6	9
4	6	6	9	9

スコア: 38

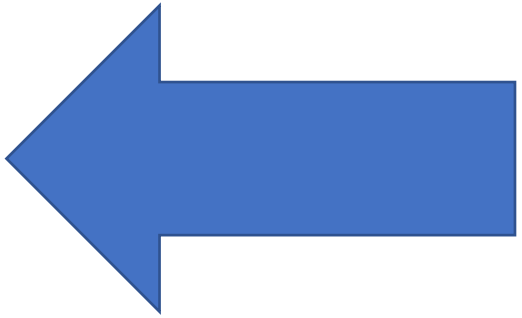
Q6.

- **Sample Input**

```
5
5 9 5 5 9
5 5 6 9 9
4 6 3 6 9
3 3 2 9 9
2 2 1 1 1
```

- **Sample Output**

```
38
```



				9
			9	9
5	9		6	9
4	6	6	9	9

スコア: 38

Q6.

- Sample Input

  5
  5 9
  5 5
  4 6
  3 3
  2 2

- Sample Output

  38

この様子をひたすら再現すればいい！

スコア: 38

Q6.

- Sample Input

  5
  5 9
  5 5
  4 6
  3 3
  2 2

- Sample Output

  38

この様子をひたすら再現すればいい！

この手法を **シミュレーション** とよぶ

スコア: 38

```cpp
#include <iostream>
#include <vector>
using namespace std;

#define W 5
#define DELETE -1

int main(void){
  while(true){
    int H;
    cin >> H;
    if(H == 0){ break; }

    // 盤面を受け取る
    vector<vector<int> > puyo(H, vector<int>(W));
    for(int i = 0; i < H; i++)
      for(int j = 0; j < W; j++)
        cin >> puyo[i][j];

    // 石が消えるまで繰り返す
    int flag, ans = 0;
    do{
      // 最初にフラグを折っておく
      flag = false;

      // すべての地点について
      for(int i = 0; i < H; i++){
        for(int j = 0; j < W; j++){
          if(puyo[i][j] == DELETE){ continue; }

          // 位置[i][j]をスタートとして
          // 色が同じである区間を求める
          int seg;
          for(seg = 1; j + seg < W; seg++)
            if(puyo[i][j] != puyo[i][j + seg])
              break;
          // 連続区間が3以上だったら消す
          if(seg >= 3){
            for(int k = 0; k < seg; k++){
              // 消す場合はフラグを立て、コストを加算する
              flag = true;
              ans += puyo[i][j + k];
              puyo[i][j + k] = DELETE;
            }
          }
        }
      }

      // DELETE になった石を上に持ち上げていく
      // (バブルソートチック)
      for(int k = 0; k < H; k++){
        for(int i = 1; i < H; i++){
          for(int j = 0; j < W; j++){
            if(puyo[i][j] == DELETE){
              int tmp = puyo[i][j];
              puyo[i][j] = puyo[i - 1][j];
              puyo[i - 1][j] = tmp;
            }
          }
        }
      }

    }while(flag);

    // 答えを出力
    cout << ans << endl;
  }

  return 0;
}
```

$$O\left(HW\left(HW^2 + H^2W\right)\right) = O(H^3)$$

```cpp
#include <iostream>
#include <vector>
using namespace std;

#define W 5
#define DELETE -1

int main(void){
  while(true){
    int H;
    cin >> H;
    if(H == 0){ break; }

    // 盤面を受け取る
    vector<vector<int> > puyo(H, vector<int>(W));
    for(int i = 0; i < H; i++)
      for(int j = 0; j < W; j++)
        cin >> puyo[i][j];

    // 石が消えるまで繰り返す
    int flag, ans = 0;
    do{
      // 最初にフラグを折っておく
      flag = false;

      // すべての地点について
      for(int i = 0; i < H; i++){
        for(int j = 0; j < W; j++){
          if(puyo[i][j] == DELETE){ continue; }

          // 位置[i][j]をスタートとして
          // 色が同じである区間を求める
          int seg;
          for(seg = 1; j + seg < W; seg++)
            if(puyo[i][j] != puyo[i][j + seg])
              break;

          // 連続区間が3以上だったら消す
          if(seg >= 3){
            for(int k = 0; k < seg; k++){
              // 消す場合はフラグを立て、コストを加算する
              flag = true;
              ans += puyo[i][j + k];
              puyo[i][j + k] = DELETE;
            }
          }
        }
      }

      // DELETE になった石を上に持ち上げていく
      // (バブルソートチック)
      for(int k = 0; k < H; k++){
        for(int i = 1; i < H; i++){
          for(int j = 0; j < W; j++){
            if(puyo[i][j] == DELETE){
              int tmp = puyo[i][j];
              puyo[i][j] = puyo[i - 1][j];
              puyo[i - 1][j] = tmp;
            }
          }
        }
      }

    }while(flag);

    // 答えを出力
    cout << ans << endl;
  }

  return 0;
}
```

まとめ

プロコンで使う基礎テクニックを学ぶことができた！

- とはいえ、頭では分かっていても、実際にコードが書けるかどうかは**別問題！**
- 今日の内容に関連する問題を付録につけておいたので、頑張って解こう！

付録 (1/3)

このスライドの写経で解ける問題

- 約数の数
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_3_D
- 深さ優先探索
  http://atc001.contest.atcoder.jp/tasks/dfs_a
- 最大公約数
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_1_B&lang=jp
- 素数判定 (MAX_Nを修正する必要あり)
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_1_C&lang=jp
- 連鎖消滅パズル
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1193&lang=jp

付録 (2/3)

頑張って解こう！

- 線形探索
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_4_A&lang=jp
- Block
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0207
- 島の数
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0067
- おつり
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0521

付録 (3/3)

頑張って解こう！！

- 整数の和
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0030&lang=jp
- 竹の花
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1610&lang=jp
- Old Bridges
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1052&lang=jp
- Ennichi
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2232&lang=jp