レギオ講習会

目標:

1. 計算量の感覚を身につける
2. 便利なデータ構造・アルゴリズムを利用してみる
3. かしこいアルゴリズムのアイデアを勉強してみる

進め方

- 1. と 2. は、全員に講義します
- 各種アルゴリズムについては、余裕のある人向けに講義します

List の使い方

こちらにも、いくつか用例を載せてあります。
ただ、詳しい使い方は検索なりして調べましょう。

- 初期化

```
lst = [0, 1, 2, 3, 4]
lst = [0]*5  #=[0,0,0,0,0]
```

- k (0, 1, ..) 番要素への読み書き

```
print(lst[k])
lst[k] = 22
```

- 要素の追加（リストの最後）

```
lst.append(5)
```

- List を追加（最後につなげる）

```
lst.extend(lst2)
```

- 要素の追加・削除

```
lst.insert(1, 11) # 新1番目として 11を挿入
lst.pop(1)  # その1番要素を削除（先頭は0）
lst.pop()   # 最後尾の要素の削除
lst.pop(-k) # 最後からk番を削除（最後はk=1）
```

- 要素への順次アクセスいくつか

```
i = 0
for elem in lst: # 要素を順に
    print(i, elem)
    i += 1
for i in range(len(lst)):  # index を順に
    print(i, lst[i])
for i, elem in enumerate(lst): # ペアで
    print(i, elem)
```

計算量のイメージと賢い考え方１（動的計画法）

問題：1 年生 (A First Grader)

問題@AtCoder
(第10回の予選問題 D)

- 入力：
  - 1行目: N (3 以上 100以下)
  - N 個の数字からなる列

```
11
8 3 2 4 8 7 2 4 0 8 8
```

- 足し算・引き算で等式を作る
  - 途中で結果が 0 以上 20 以下に収まるような式の作り方が何通りあるか求める
  - 等式記号は、最後になるような式

```
8+3-2-4+8-7-2-4-0+8=8
```

正解の組み合わせは 2^{31}を越えることも
（6割分の問題は、2^{31}を越えない）

さて、
どう解こう？

問題：1 年生 (A First Grader)

問題＠AtCoder
(第10回の予選問題 D)

- 全部の式を作成してみる
  - 数字が 100 個あると、
    +/- の入る場所は98か所
    - 組み合わせは 2^{98}
  - 正解になるものでも 2^{31} を越える
    - 1G = 2^{30} なので、結構 1秒じゃ厳しいあたり

```
11
8 3 2 4 8 7 2 4 0 8 8
```

```
8+3-2-4+8-7-2-4-0+8=8
```

- そもそも、すべての組み合わせを作成するのも大変

現実計算機のスペック

再掲

- CPU のクロック：数GHz
  - 1秒間に数G回の命令をこなす（1命令1nano sec以下）
  - 1000命令かかる処理でも、数M回こなす
  - つまり「1秒」で解くとは、数G回以内の命令で解くということ
- メモリ容量
  - 主メモリ数GBは当たり前
    - ➢でも、OS の領域も残してね
  - ハードディスクはTBオーダー
    - ➢でも、disk は遅いけどね

但し、主メモリランダムアクセスは、CPUより数十倍遅い

1K = 1000 = 10^3
1M =1000,000 = 10^6
1G = 1000,000,000 = 10^9
1T = 10^{12}

でも、キャッシュは数MB程度以下

量感覚イメージ

再掲

- 累乗: 2^{10}=1024≒K, 2^{20} ≒ M, 2^{30} ≒ G
- 階乗:
  - 1x2x3x4x5=120, 1x2x...x10=3.6Mぐらい
  - ..x13 で4G越える
- 32bit 符号付整数で表現できる値:　-2G ～ +2G
- 時間
  - 1秒に数G回の演算が可能
  - 1分:60秒、1時間:3.6K秒、1日:86.4K秒、
  - 1M秒で11日半ぐらい

問題：1 年生 (A First Grader)

問題@AtCoder
(第10回の予選問題 D)

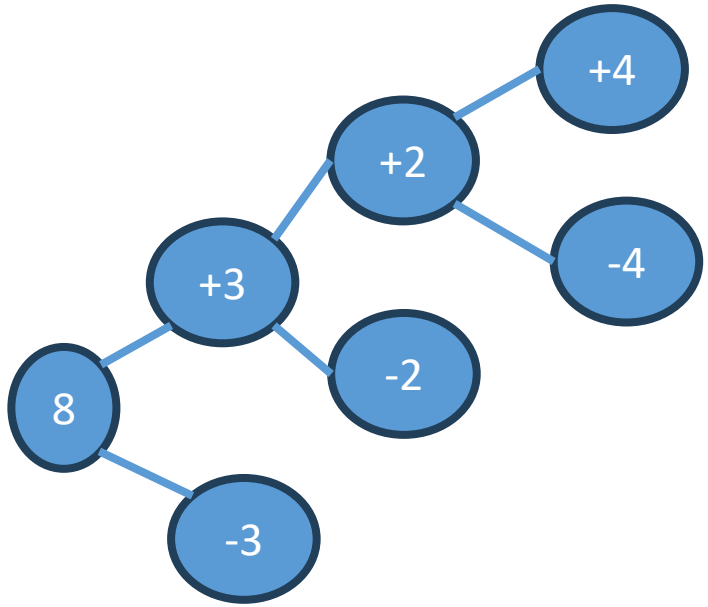
- ちなみに、全組み合わせ求めるってどんな感じ？

8+3−2−4+8−7−2−4−0+8=8

i 番目以降を全部チェックする関数 solve(i, current)

```
def solve(i, current):
    if current < 0 or current > 20:
        return 0
    if i == len(nums):
        if current == last:
            return 1
        else:
            return 0
    num = nums[i]
    result0 = solve(i+1, current+num)
    result1 = solve(i+1, current-num)
    return result0 + result1
```

解答例
（遅い）



i+1 番目以降をチェックする solve(i+1, ..) を
2回呼び出す

問題：1 年生 (A First Grader)

問題@AtCoder
(第10回の予選問題 D)

- じゃ、どうやって解く？
  - もし、i-1 番目までの計算結果が k になる式の個数が分かっていたら、i 番目の結果も分かるんじゃない？
  - i-1 番目までの計算結果が k となる数が dp[k] に格納されている
  - i 番の数が n だとする
  - i 番までの計算結果は、
    ndp[k] = dp[k-n] + dp[k+n]
    if　k – n >= 0 and k+n <= 20

8?3?2=3: 1通り

8?3?2=7: 1通り

8?3?2=9: 1通り

8?3?2=13: 1通り

8?3?2?4=k
(k= 0, .., 20)
はそれぞれ何通り

問題：1 年生 (A First Grader)

問題@AtCoder
(第10回の予選問題 D)

- じゃ、どうやって解く？
  - もし、i-1 番目までの計算結果が k になる式の個数が分かっていたら、i 番目の結果も分かるんじゃない？
  - i-1 番目までの計算結果が k となる数が dp[k] に格納されている
  - i 番の数が n だとする
  - i 番までの計算結果は、
    ndp[k] = dp[k-n] + dp[k+n]
    if　k – n >= 0 and k+n <= 20

できれば、ここから自作してみよう

解答例

```
for num in nums:
    ndp = [0] * 21
    for i in range(21):
        if i + num <= 20:
            ndp[i + num] += dp[i]
        if i - num >= 0:
            ndp[i - num] += dp[i]
    dp = ndp

print(dp[last])
```

計算回数は、数字の数 N x 20 ぐらい
N は 100 ぐらいだから、全然平気！！

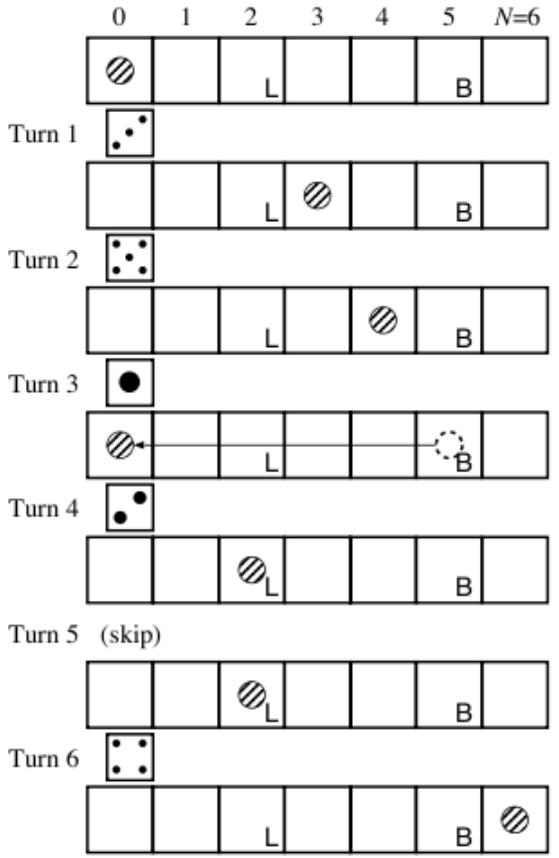
動的計画法(DP, Dynamic Programming)：
部分問題から順に解き、
部分問題の結果を用いてより大規模な問題を解く

類題

ACM ICPC 2007 アジア予選 東京大会

https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1277

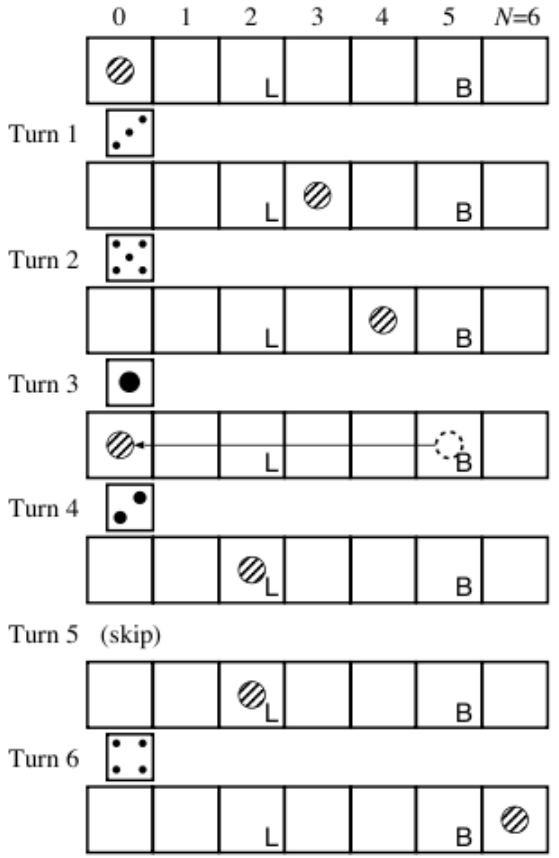
- ある種の双六ゲーム
  - 左端からスタート、右端がゴール
  - サイコロを振って、目の数だけ進む
    - ゴールを超えたら、その分戻る
  - L: 1回休み
  - B: スタートに戻る
- 質問： T 回以内にゴールにたどり着く確率は？(1≦T≦100)



どうやったら解けそう？

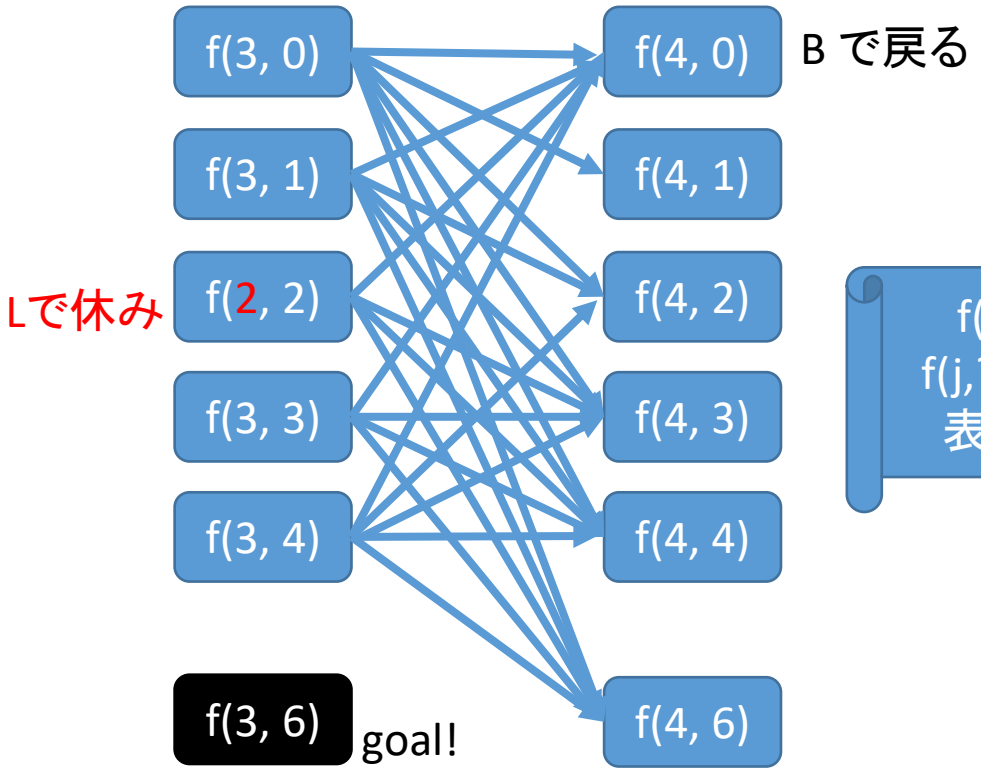
試行錯誤

- 最初は必ずスタート
- 1/6 の確率で 1 から 6 に移動
  - 但し、L についたら1回休み
  - B についたら、スタートに戻る
- サイコロのパターンを全部試す？
  - 10回振るパターン: 6^{10} ～60M
  - 20回だと、6^{20} ～ 3600ペタ？？
  - 100回なんて、、、、

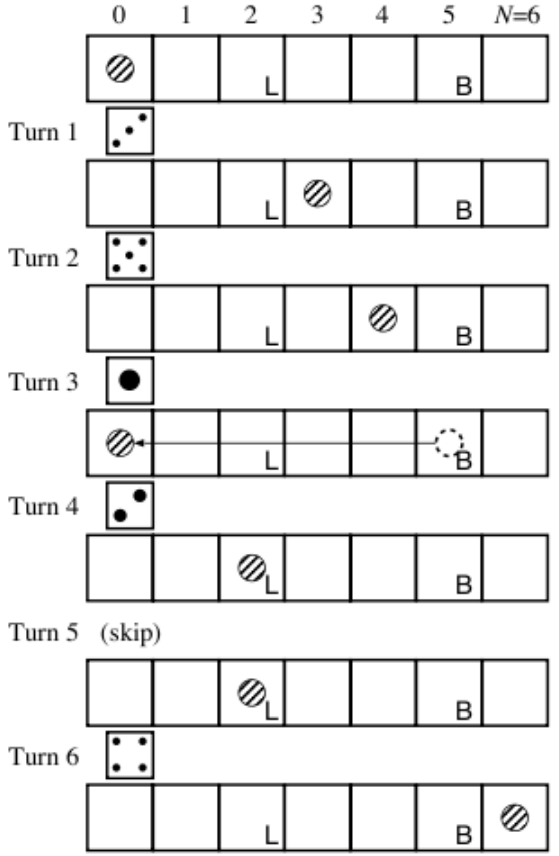


解法

- f(k, p): k回目に位置 p に到着する確率とする



f(k, p)は
f(j,?) (j<k)で
表現可能



便利なライブラリと
その利用

便利なライブラリ（Python)

- Python は各種データ構造を組み込み型やライブラリとして搭載
  (python3.10の標準ライブラリ)
  - 集合系
    - List（同種のデータの集まり、ソート可）
    - Tuple（長さ固定のデータの組、各要素は異種データでもOK）
    - Dict（マッピング型、キーから対応するオブジェクト（値）へのマッピング）
  - deque, heapq, complex などいろいろ

いくつか使ってみよう

問題：2番目に大きな数を返す (python)

問題Z
@AtCoder

- 5つの数字が与えられたとして、2番目に大きな数字を返す
- 解1：一番大きなものを探してから、それを省いてから、また一番大きなものを選んでみよう。
- 解2：面倒だから、ソートして2番目の要素をとればいいじゃん。
  - list のsort メソッド
    - ➢list の順番を変えて小さい順に
  - sorted(list) 関数
    - ➢list の順番は変更せず、小さい順のリストを新規作成

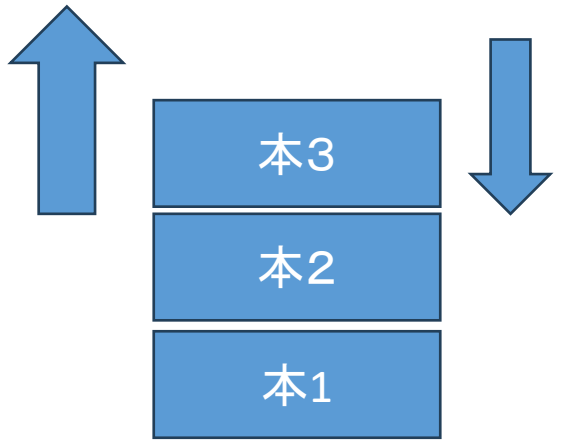
解答例@github

```
def main():
    n = 5
    numbers = [int(input()) for _ in range(n)]
    numbers.sort()
    print(numbers[n - 2])
    # あるいは、numbers をソートした結果を新規作成する場合
    # print(sorted(numbers)[n-2])
```

問題：図書館2

JOI 2021/2022
二次予選A

- ビ太郎君は、
  - 新たな本を積むか
  - 一番上に積まれた本を読んで返却するか
- List を使えば、簡単に
  - 試してみましょう！



解答例@github

ライブラリと計算量（Python）

ライブラリは、
「用途に応じて効率的に」
実装されています。

- List
  - 各要素へのアクセスは、要素数によらず一定
  - 要素の検索や先頭要素の削除は要素数に応じた時間必要
  - List の複製（コピー）は要素数に応じたコスト
- sort: 要素数 *n* に対して　log(*n*) x *n* に比例する計算量
  - $\log(2^{10}) = 10$, $\log(2^{20}) = 20$ なので、要素数が大きく増えても log 部分の増加は小さい　（参考：ソートのアルゴリズム）
- Dictionary (辞書型, dict): 要素の検索・追加・削除は要素数によらず一定コスト（参考：ハッシュテーブル(wiki) ）

順列生成

余談

こういうのを使うと、
予選D問題も簡単に解けたりします

- Python には、並び替えを作成してくれるライブラリがあります

```
import itertools

lst = list(range(5))
print('lst:', lst)

for v in itertools.permutations(lst, 3):
    print(v)
```

lst = [0, 1, 2, 3, 4]

lst から3要素つかった
順列を生成

実行結果

```
lst: [0, 1, 2, 3, 4]
(0, 1, 2)
(0, 1, 3)
(0, 1, 4)
(0, 2, 1)
(0, 2, 3)
(0, 2, 4)
(0, 3, 1)
(0, 3, 2)
(0, 3, 4)
(0, 4, 1)
....
(4, 3, 2)
```

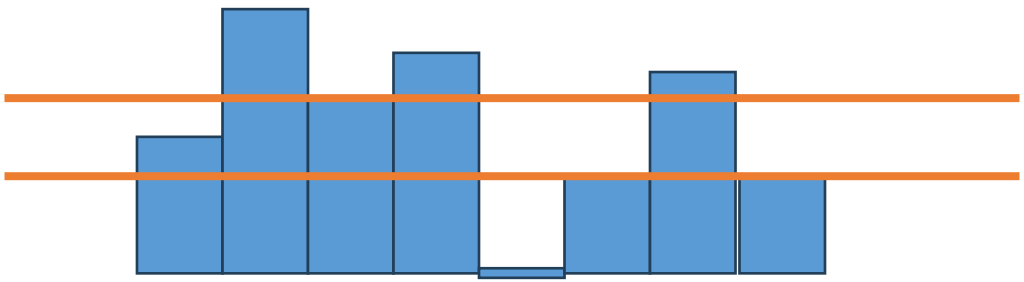
問題：日本沈没

JOI 2018/2019 予選D
ちょっと難しい

- 列島が線分としてあらわされ、標高が A[i] で表される
- 海水面が上昇するとき、水面にある線分区間（島）が何個あるか
- 区間の数: $N \leq 10^5$
- $0 \leq A[i] \leq 10^9$

線を引いて、島を数える

計算量的には、
$N^2 = 10^{10}$ は厳しい所

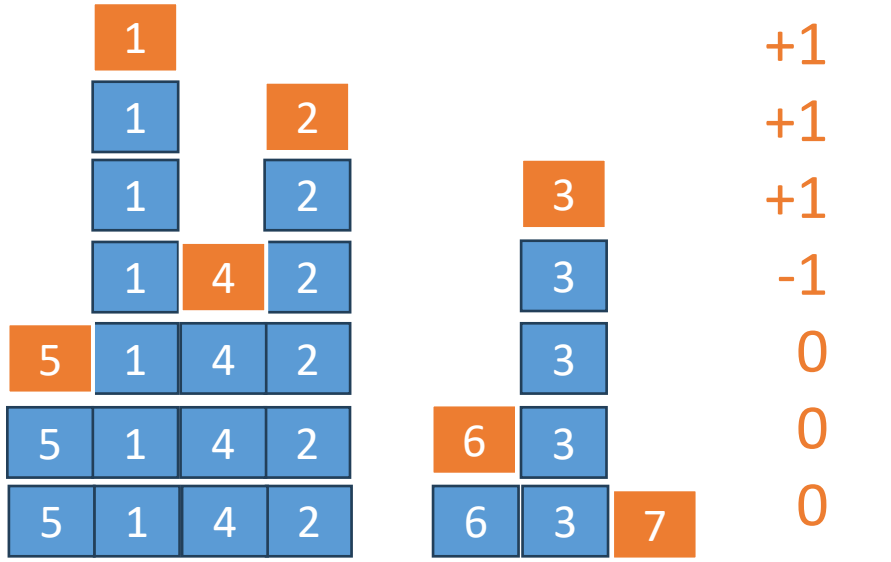


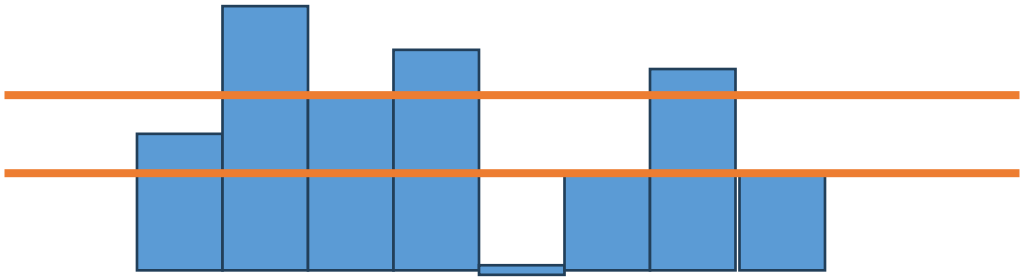
問題：日本沈没

JOI 2018/2019 予選D
ちょっと難しい

アイデア

- 例えば、一番高い領域から、順に区間に配置していけば？
- 前後が陸か海か確認しながら島の数を把握できそう？
  - 海陸海：+1
  - 陸陸陸：-1
  - 陸陸海、海陸陸：変化なし





- 陸地のソートまで
- 解答例

プログラム例@github

最短経路の探索

幅優先探索、最短経路探索

Deque, Heapq (Python)

全部リストの一種

- Deque
  - 先頭or最後に要素を追加
  - 先頭or最後から要素を削除
- HeapQueue
  小さいもの順で取り出す
  - 要素にリストを使うと、
    複数要素を辞書順にソート

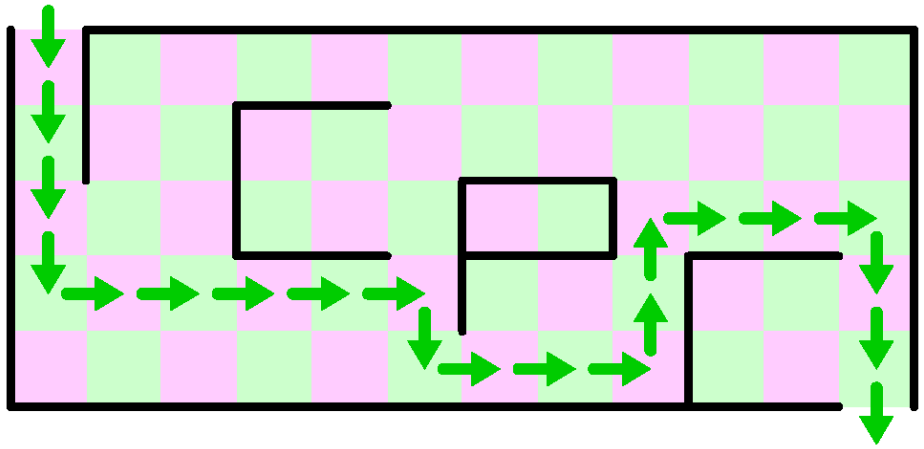
```
deque = deque()
deque.append(val)     # 最後に要素追加
deque.appendLeft(val)# 先頭に要素追加
deque.pop()           # 最後の要素取得＆削除
deque.popLeft()       # 先頭の要素取得＆削除
```

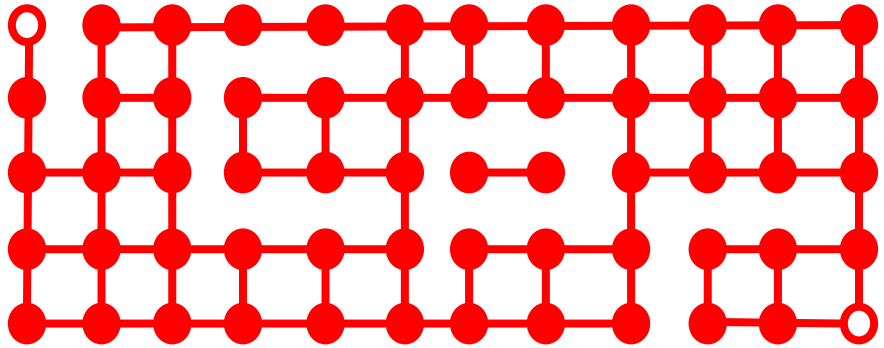
```
priq = []    # 作成するのは普通のリスト
heappush(priq, val) # 要素追加
val = heappop(priq) # 要素取得＆削除
# 優先度に2指標(abs(val), val.real)を利用したい場合
priq2 = []
heappush(priq2, [abs(val), val.real, val])
pri0, pri1, val = heappop(priq2)
```

Int 版program@github
complex 版program@github

お題（迷路と命ず）

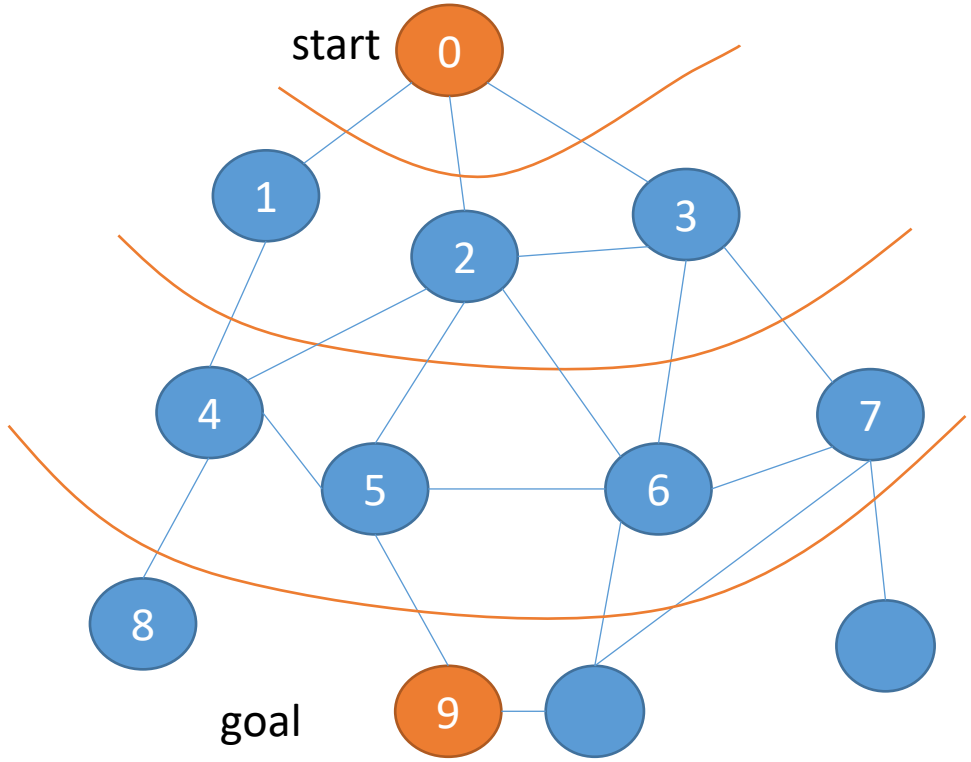
- 問題：ACM ICPC というプログラミングコンテストの2010年国内予選問題（参考 1, 2, AOJ）





幅優先探索という考え方

- 開始点から近い順に、隣接点を探索していく方法
  - 一度辿ったところは、再探索しない

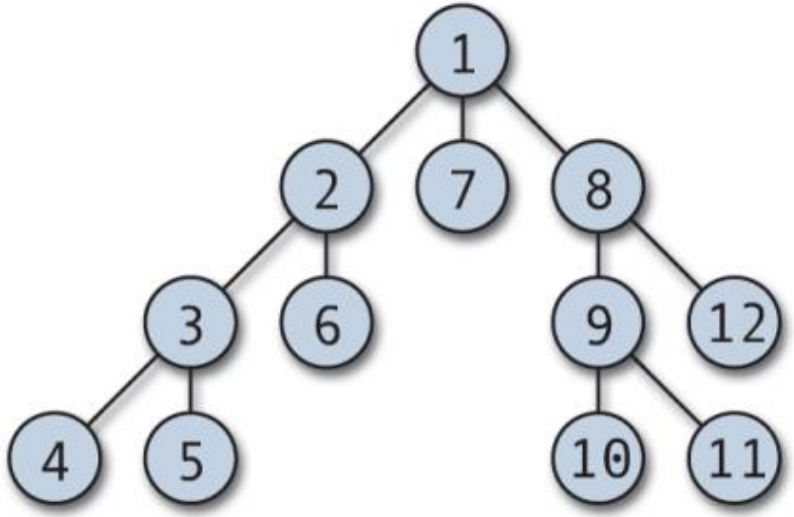


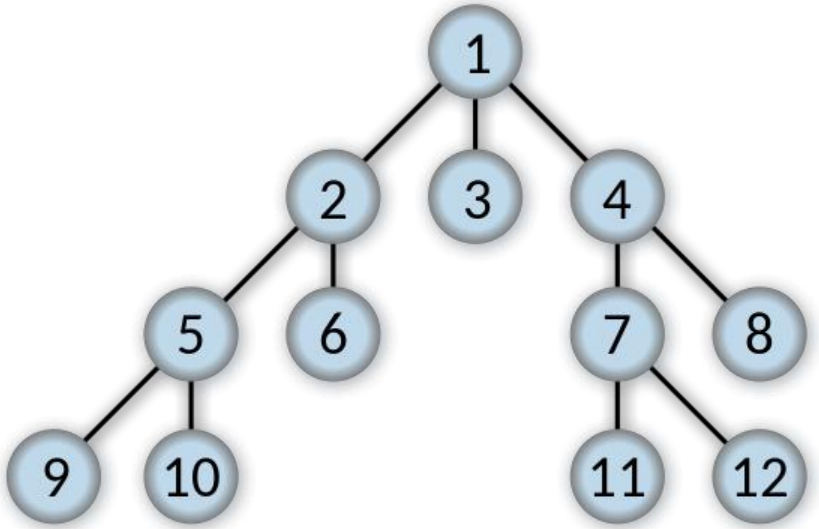

```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

深さ優先探索と幅優先探索

■深さ優先探索
- ●最近探索した点を優先
- ●Stack or 関数再帰呼出し利用

■幅優先探索
- ●開始点から近い順
- ●Queue を利用

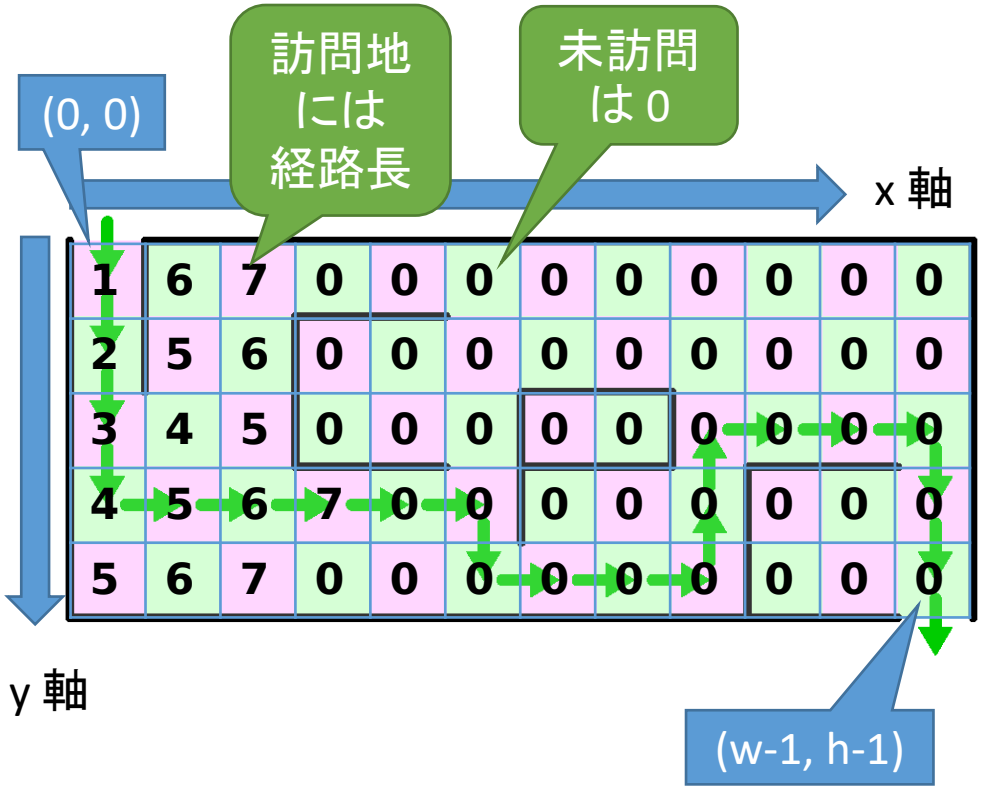




それぞれの図は Wikipedia から引用

幅優先探索してみる

- なにがいるかな？
  - 各方向に進めるか？
    - 壁判定がいりそう
  - 開始点から順番に探索
    - queue が便利
  - 各地点は訪問済み？
    ステップ数の記録は？
    - 2次元配列でOK



幅優先探索してみる

プログラム例@github

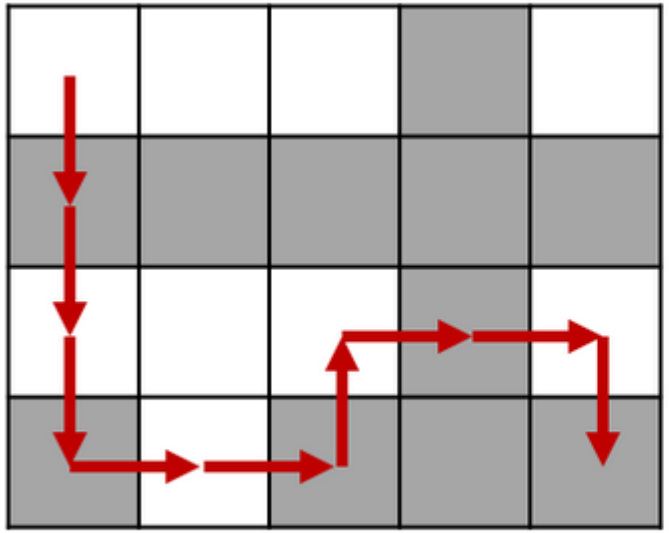
- なにがいるかな？
  - 各方向に進めるか？
    - 壁判定がいりそう
  - 開始点から順番に探索
    - queue が便利
  - 各地点は訪問済み？
    ステップ数の記録は？
    - 2次元配列でOK

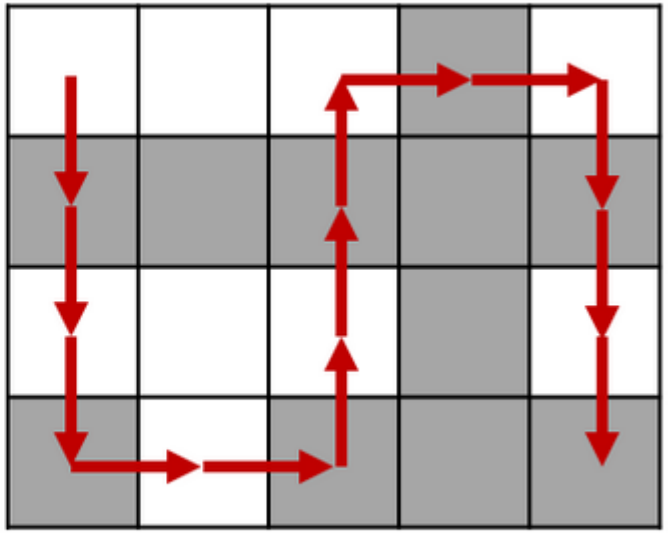
```
def solve() -> int:
    goal = [w - 1, h - 1]
    q = deque([[0, 0]])
    steps = [[0 for _ in range(w)] for _ in range(h)]
    steps[0][0] = 1
    while q:
        x, y = q.popleft()
        current_step = steps[y][x]
        for dx, dy in directions:
            if can_go(x, y, dx, dy):
                nx, ny = [x + dx, y + dy]
                if steps[ny][nx] > 0:
                    continue
                if [nx, ny] == goal:
                    return current_step + 1
                steps[ny][nx] = current_step + 1
                q.append([nx, ny])
    return 0
```

問題：カーペット (Carpet)

問題＠AtCoder
JOI 2021/2022 二次予選B

- 2色のタイル柄のカーペット上を左上から右下に移動する最短経路を見つける





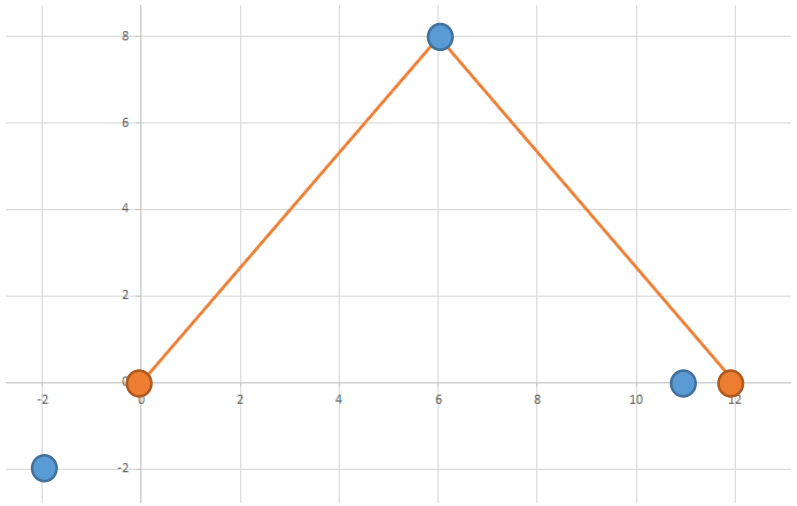
似たような考え方で
解けるので、
自分でトライ！

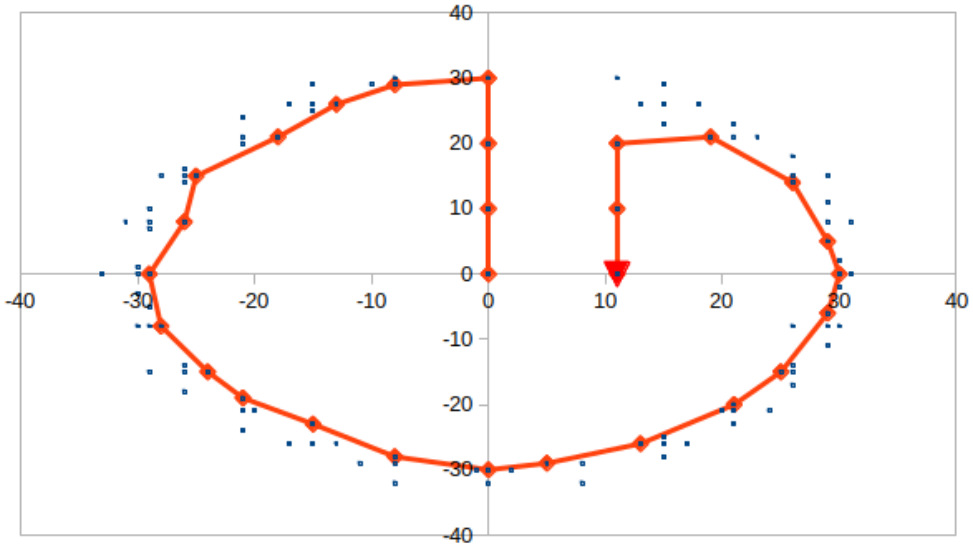
- 書きかけ
- 正解例

問題2つめ（最短経路問題）

- 盤面上にいくつか点がある
  - ●2点の距離が10以下の場合は移動可能
  - ●始点から終点まで、最短経路で移動したい

最短経路を
求めましょう

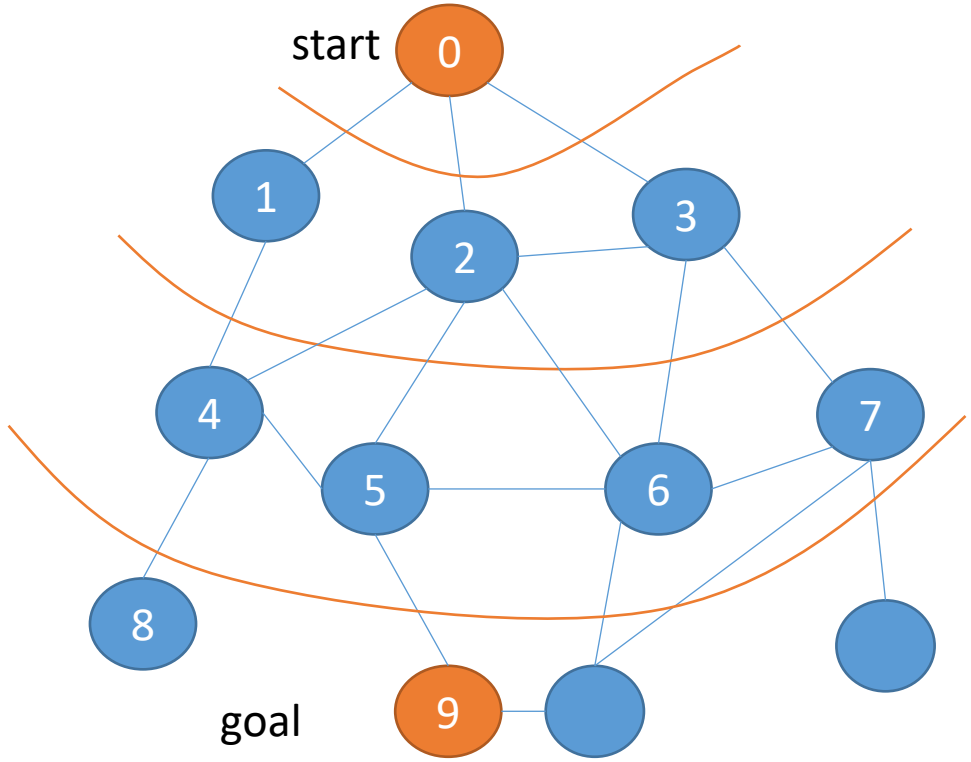




幅優先探索という考え方

再掲

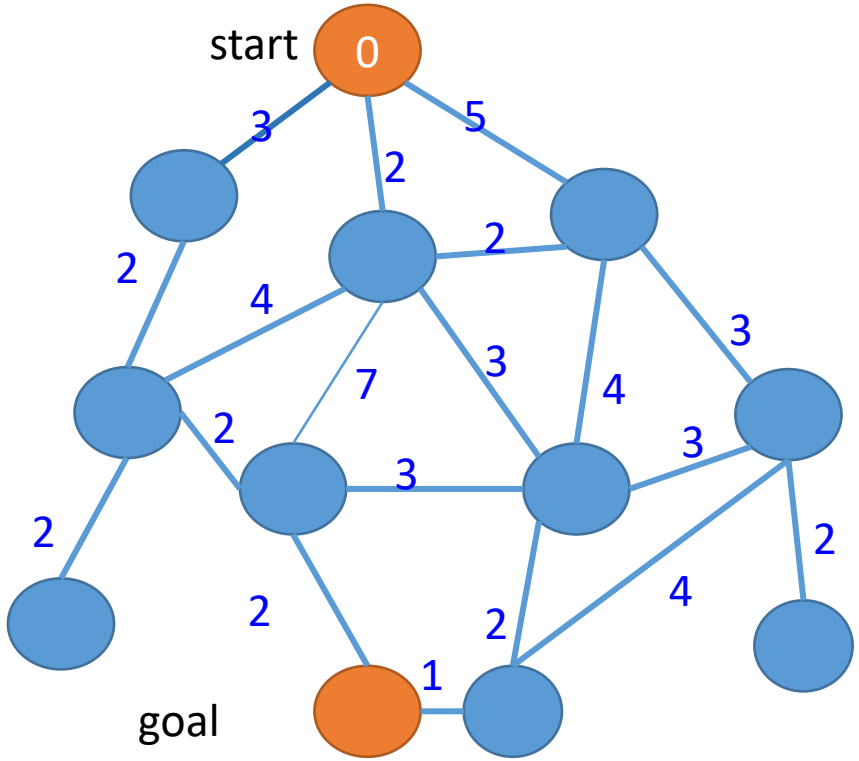
- 開始点から近い順に、隣接点を探索していく方法
  - 一度辿ったところは、再探索しない




```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

今回は枝に長さがある

- 一番近いところを探すのがちょっと面倒？
- でも、近いところからやらないと、何度もやり直し

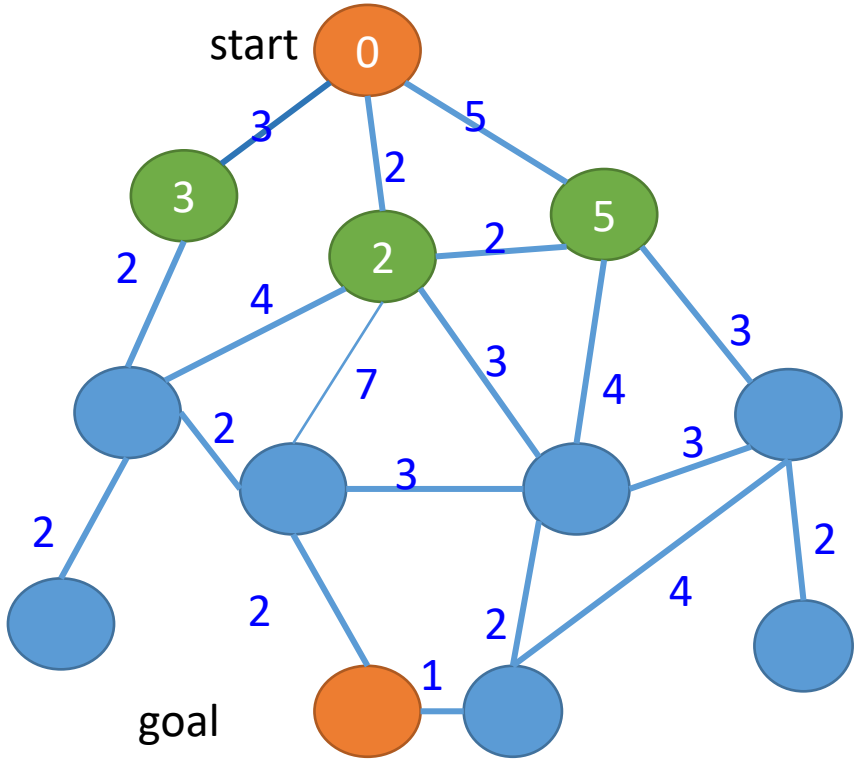



```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

プログラム例@github

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

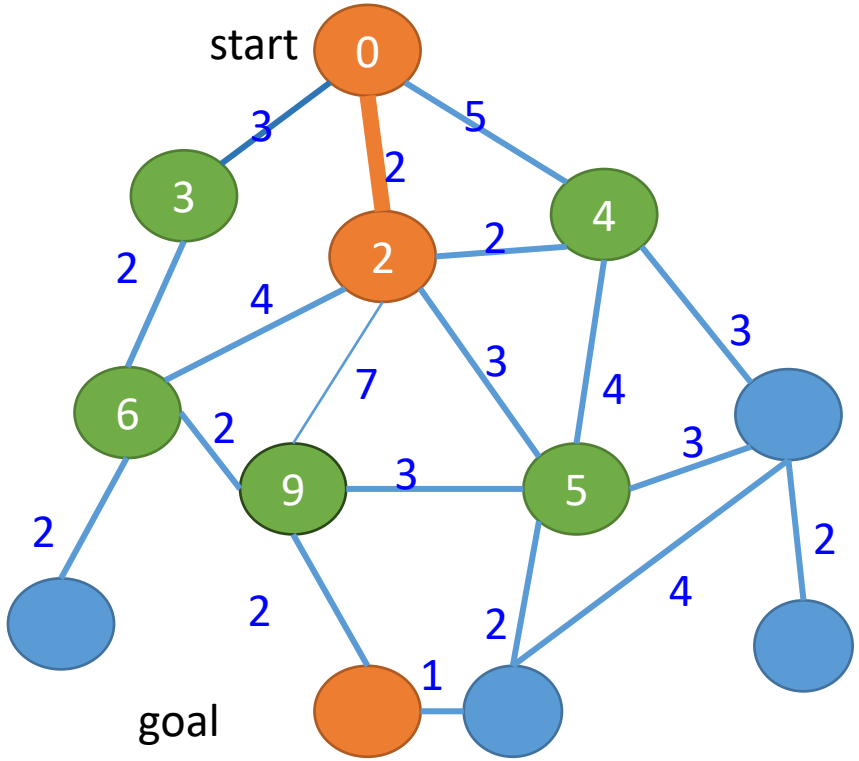


HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

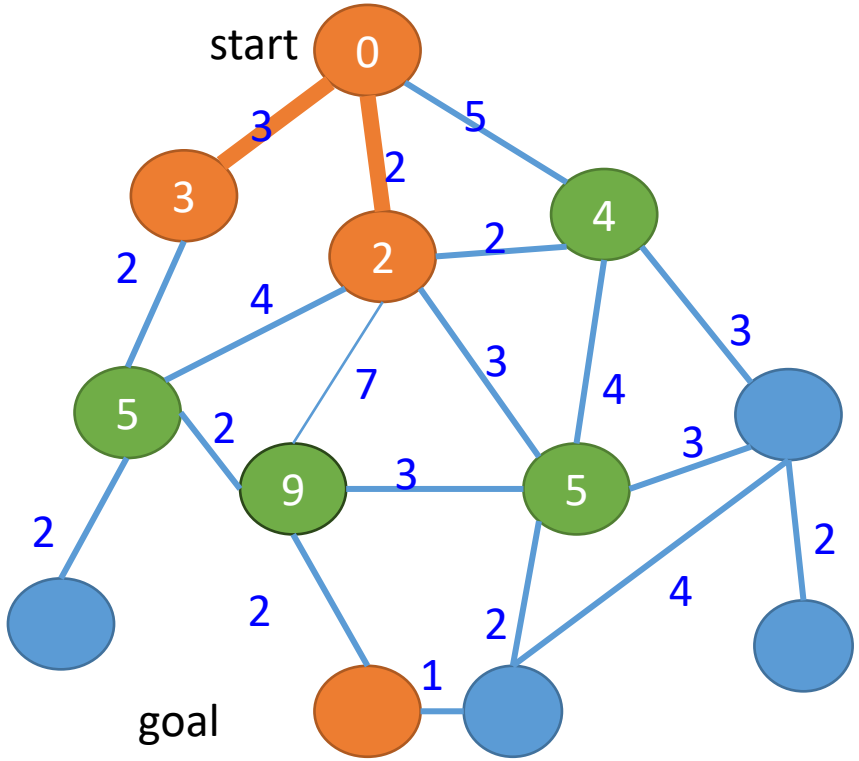


HeapQueue に「コスト」と「ノード」の組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

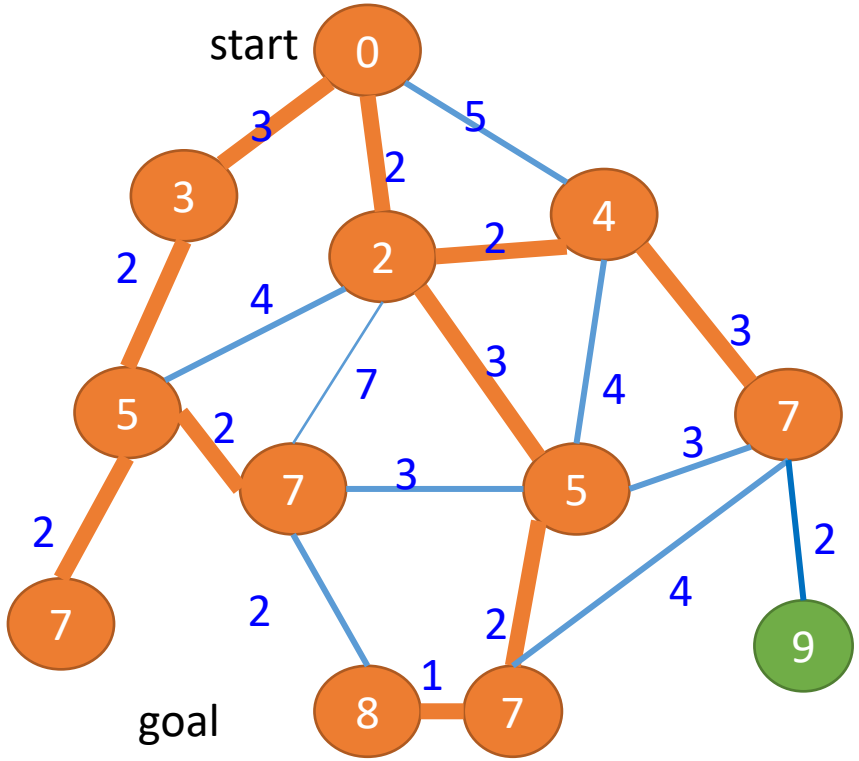


HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば



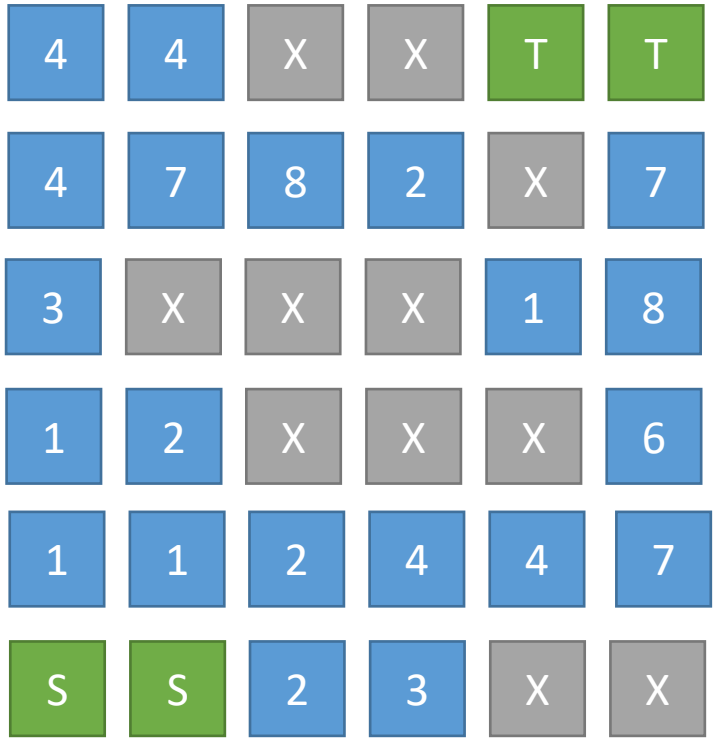
HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ACM ICPC 2007 国内予選

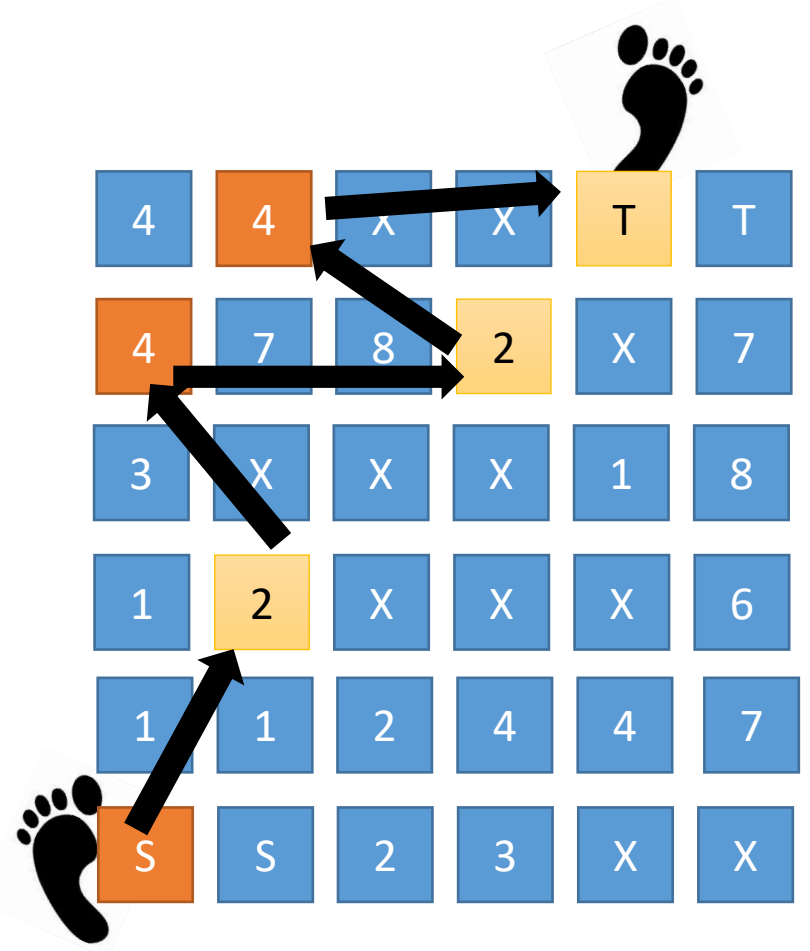
さまざまな最短経路問題

- スタートエリアからゴールエリアに移動
  - 各位置には、左足か右足を置ける
  - 各位置に書かれた分の時間を消費
    - 足を置けない場所もある
  - 次に置くのは逆の足
    - 置ける範囲が決まっている



解き方を考えてみよう

- スタートエリアからゴールエリアに移動
  - 各位置に、左足か右足を置く
  - 各位置にかかれた分の時間を消費
    - 足を置けない場所もある
  - 次に置くのは逆の足
    - 置ける範囲が決まっている
- 考え方？
  - 最短経路っぽい？
  - でも、左足右足って？
  - 同じ場所に両足置くことも？



考え方

プログラム例(C++)
プログラム例(python)

- [足の位置、左／右]をノードとするグラフをイメージすれば？
  - ノードをイメージできれば、最短経路問題