プログラミング応用 第4回

河瀬 康志

2017年7月2日

アウトライン

演習問題

問 1

リスト [1,2,3] を Shuffle1 によってシャッフルしたとき，[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1] はそれぞれどれだけの確率で出現するか？

- プログラムを用いずに答えてもよい．
- 余力がある人は Shuffle2 についても計算せよ．

問 2

大文字または小文字のアルファベット 8 文字からなるランダムなパスワードを生成するプログラムを作成せよ．

演習問題

問 3

http://yambi.jp/lecture/advanced_programming2018/data.csvには，各行が 2 つのデータの組からなる csv ファイルが置いてある．このファイルを読み込み，そのデータたちの関係を表すもっともらしい直線を最小二乗法を用いて求めよ．また，もっともらしい直線であることを，プロットすることにより確かめよ．

問 4

確率 1/100 で欲しいキャラが出るガチャをまわしたとき，何回で初めてそのキャラがでるか．10000 回試してその頻度分布をヒストグラムで表せ．

授業スケジュール

	日程	内容
第 1 回	6/11	ガイダンス・復習
第 2 回	6/18	文字列操作（文字列整形，パターンマッチ，正規表現） 平面幾何（線分の交差判定，点と直線の距離，凸包）
第 3 回	6/25	乱数（一様分布，正規分布への変換，乱数生成） 統計（データ処理，フィッティング）
第 4 回	7/2	計算量（オーダー表記） スタックとキュー（幅優先探索，深さ優先探索）
第 5 回	7/9	ソートアルゴリズム バックトラック（N クイーン問題，数独）
第 6 回	7/23	動的計画法（ナップサック問題） 最短経路探索（Warshall-Floyd, Bellman-Ford, Dijkstra）
第 7 回	7/30	巡回セールスマン問題
期末試験	8/6?	—

アウトライン

データサイズと計算時間

	$\sqrt{n}$	n	$n \log n$	n^2	2^n	$n!$
1 秒	$1.0e+16$	$1.0e+8$	$6.4e+6$	10000	26	11
1 分	$3.6e+19$	$6.0e+9$	$3.1e+8$	77500	32	12
1 時間	$1.3e+23$	$3.6e+11$	$1.5e+10$	600000	38	15
1 日	$7.5e+25$	$8.6e+12$	$3.3e+11$	$2,9e+6$	42	16
1 月	$6.7e+28$	$2.6e+14$	$8.7e+12$	$1.6e+7$	47	17
1 年	$9.7e+30$	$3.1e+15$	$9.7e+13$	$5.6e+7$	51	18
1 世紀	$9.7e+34$	$3.1e+17$	$8.5e+15$	$5.6e+8$	58	20

- 制限時間内にどのサイズまで計算できるか
- 1 秒で 10^8 回の計算ができると仮定

ランダウの記法

n が十分に大きい時

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^{100} < 2^n < 3^n < n! < n^n$$

この順序を表す記法を導入する $\Rightarrow$ オーダー表記

$f(n) = O(g(n))$ とは

$\exists k > 0, \exists n_0, \forall n > n_0, f(n) \leq k \cdot g(n)$

$f(n) = \Omega(g(n))$ とは

$\exists k > 0, \exists n_0, \forall n > n_0, f(n) \geq k \cdot g(n)$

- $f(n) = O(g(n))$ かつ $f(n) = \Omega(g(n))$ のとき，$f(n) = \Theta(g(n))$ とかく
- 一般にオーダーが小さいほど計算が速く終わる
- ただし，実用的には定数係数も重要

例

- $1000n = O(n)$
- $18n^2 + 5n + 1 = O(n^2)$
- $10(n+5)^8 = O(n^8)$
- $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log n)$　　($\log$ の底は無視できる)
- $n = O(2^n)$　　(上から抑えていればよい)
- $5 \cdot 2^{n+3} = O(2^n)$
- $10^{\log_{10} n} = O(n)$
- $(\log n)^{100} = O(n^{0.001})$
- $\log(n!) = O(n \log n)$

多項式時間アルゴリズム

- 理論的には,「計算時間が入力サイズ n の多項式である」ことが効率的に解けるかの境目 (多項式時間アルゴリズム)
- 組合せ爆発，指数爆発 $\Rightarrow$ 解くことが困難
  - 組合せ爆発の恐ろしさ `https://youtu.be/Q4gTV4r0zRs`
- 実際に問題を解くときは，解きたい問題サイズの考慮が重要

多項式時間 ($O(n^k)$)

$$\log n, (\log n)^{100}, n, n \log n, n^2, n^3, n^{100}, \ldots$$

超多項式時間 (多項式時間ではない)

$$2^n, 3^n, n!, n^n, n^{\log n}, 2^{(\log n)^2}, (\log n)^{\log n}, 2^{2^n}, \ldots$$

Advanced Topic: P vs NP

正誤判定する問題について

クラス P　多項式時間アルゴリズムの存在する問題のクラス

クラス NP　正しい場合には，うまい証拠があって，多項式時間でその正しさを検証できる問題のクラス
(Nondeterministic Polynomial-time)

- P⊆NP は明らか
- P≠NP かどうかは未解決
  - 2000 年に Clay Mathematics Institute によって選ばれた 7 大数学の未解決問題 (Millennium Prize Problems) の一つ
  - 解けたら 100 万ドル

計算量の例 (1/3)

べき乗の剰余

37^n を 127 で割った余りは何になるか？

単純な計算法

```
>>> def simple(n):
...     res=1
...     for i in range(n): res=(res*37)%127
...     return res
...
>>> list(map(simple,range(1,10)))
[37, 99, 107, 22, 52, 19, 68, 103, 1]
>>> simple(100000000) # 少し遅い
37
```

- 計算量は $O(n)$ (入力サイズは $\log n$ ビットなので指数時間)
- (37**100000000)%127 はもっと遅い (巨大整数の計算が必要)

計算量の例 (2/3)

べき乗の剰余

37^n を 127 で割った余りは何になるか？

効率の良い計算法

```
>>> def efficient(n):
...     if n==1: return 37
...     elif n%2==0: return (efficient(n//2)**2)%127
...     else: return (37*efficient(n//2)**2)%127
...
>>> list(map(efficient,range(1,10)))
[37, 99, 107, 22, 52, 19, 68, 103, 1]
>>> efficient(100000000) # 一瞬
37
```

- 計算量は $O(\log n)$
- pow(37,n,127) でも同じ方法で計算をしてくれる

計算量の例 (3/3)

べき乗の剰余

37^n を 127 で割った余りは何になるか？

もっと効率の良い計算法

```
>>> def ultrafast(n): return (1, 37, 99, 107, 22, 52, 19, 68, 103)[n%9]
...
>>> list(map(ultrafast,range(1,10)))
[37, 99, 107, 22, 52, 19, 68, 103, 1]
>>> ultrafast(100000000) # 一瞬
37
```

- 計算量は $O(1)$
- 余りの周期が 9 であることを利用している

分割統治法の計算量

分割統治法 (Divide-and-Conquer)

- 分割：問題を小さな問題 (部分問題) に分割しそれぞれを解く
  - サイズ n の問題をサイズ n/b の問題 a 個に分割すると仮定
- 統治：部分問題の解を合わせることで元問題の解を求める
  - $O(n^c)$ 時間でできると仮定
- 分割と統治は再帰的に行う

計算時間 $T(n)$ について以下の漸化式が成立

$$T(n) = \begin{cases} aT(n/b) + O(n^c) & (n > 1), \\ O(1) & (n = 1). \end{cases}$$

漸化式を解くと以下が成立

$$T(n) = \begin{cases} O(n^c) & (\log_b a < c), \\ O(n^c \log n) & (\log_b a = c), \\ O(n^{\log_b a}) & (\log_b a > c). \end{cases}$$

分割統治法の計算量

分割統治法 (Divide-and-Conquer)

- 分割：問題を小さな問題 (部分問題) に分割しそれぞれを解く
  - サイズ n の問題をサイズ n/b の問題 a 個に分割すると仮定
- 統治：部分問題の解を合わせることで元問題の解を求める
  - $O(n^c)$ 時間でできると仮定
- 分割と統治は再帰的に行う

計算時間 $T(n)$ について以下の漸化式が成立

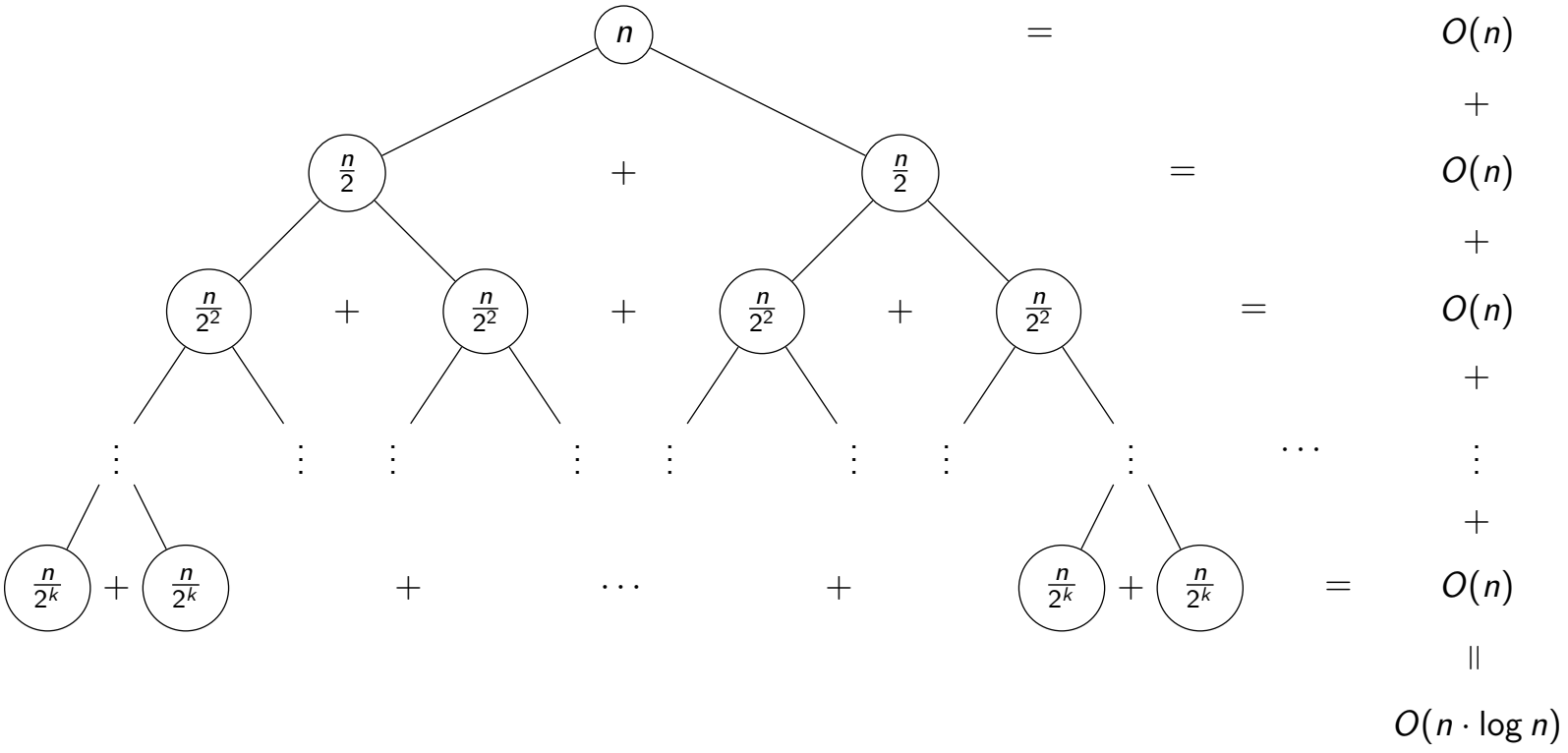
$$T(n) = \begin{cases} aT(n/b) + O(n^c) & (n > 1), \\ O(1) & (n = 1). \end{cases}$$

漸化式を解くと以下が成立

$$T(n) = \begin{cases} O(n^c) & (\log_b a < c), \\ O(n^c \log n) & (\log_b a = c), \\ O(n^{\log_b a}) & (\log_b a > c). \end{cases}$$

分割統治法の計算量

$a = 2, b = 2, c = 1$ の場合 $(T(n) = 2T(n/2) + O(n))$



分割統治法の計算量 — 例

漸化式	計算時間
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = 2 \cdot T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = 2 \cdot T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 3 \cdot T(n/3) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 3 \cdot T(n/2) + O(n^2)$	$T(n) = O(n^2)$
$T(n) = 4 \cdot T(n/2) + O(n^2)$	$T(n) = O(n^2 \log n)$

分割統治法の例 1

与えられたリストに指定された数があるか

```
def find(l,x):
    return _find(l,x,0,len(l))

# l[i]==x なる s<=i<t があるか
def _find(l,x,s,t):
    if s>=t: return -1
    if s+1==t:
        if l[s]==x: return s
        else: return -1
    m = (s+t)/2
    return max(_find(l,x,s,m),_find(l,x,m,t))
```

n 個の要素から探索するために $n/2$ 個の要素の問題を 2 個解く

$$T(n) = \begin{cases} 2T(n/2) + O(1) & (n > 1) \\ O(1) & (n = 1) \end{cases}$$

$$T(n) = O(n^{\log_2 2}) = O(n)$$

分割統治法の例 2

与えられた昇順に値の並んだリストに指定された数があるか

```
def bfind(l,x):
    return _bfind(l,x,0,len(l))

# l[i]==x なる s<=i<t があるか
def _bfind(l,x,s,t):
    if s>=t: return -1
    m = (s+t)/2
    if l[m]<x: return _bfind(l,x,m+1,t)
    if l[m]>x: return _bfind(l,x,s,m-1)
    return m
```

n 個の要素から探索するために $n/2$ 個の要素の問題を 1 個解く

$$T(n) = \begin{cases} T(n/2) + O(1) & (n > 1) \\ O(1) & (n = 1) \end{cases}$$

$$T(n) = O(\log n) = O(n)$$

実行時間の計測

```
import time
start = time.time()
f() # 重たい処理
end = time.time()
print('elapsed time: {}'.format(end-start))
```

time.time(): 1970 年 1 月 1 日 0 時 0 分 0 秒 (世界標準時) からの経過秒数

対話環境から実行する場合は下記のように実行すれば良い

```
>>> import time
>>> start = time.time(); f(); end = time.time()
>>> print('elapsed time: {}'.format(end-start))
```

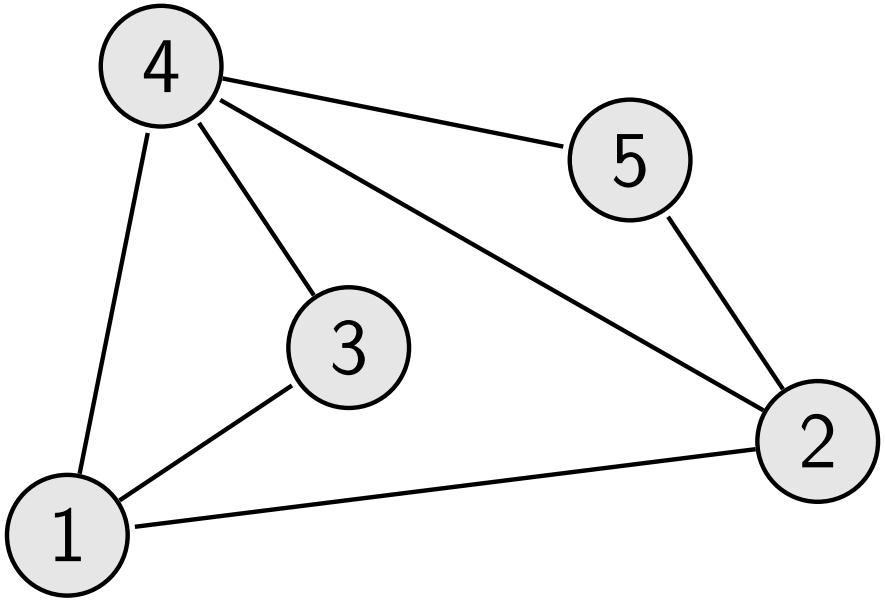
アウトライン

グラフ

- 点の集合 V と二点間を結ぶ辺の集合 E のペア
  - 鉄道ネットワーク，web のリンク構造，電気回路，友人関係など
  - 点数を n，辺数を m であらわすことが多い

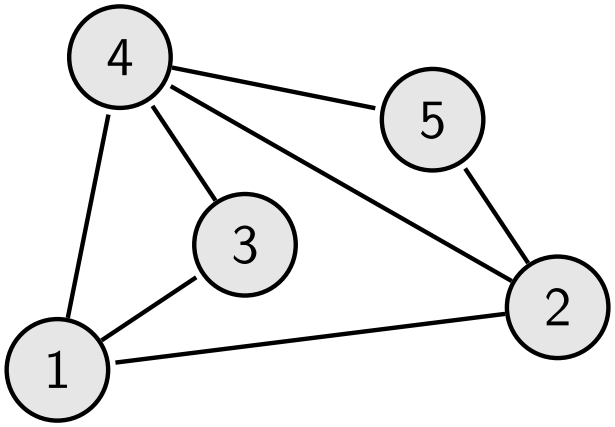
例：

- $V = \{1, 2, 3, 4, 5\}$
- $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{4, 5\}\}$

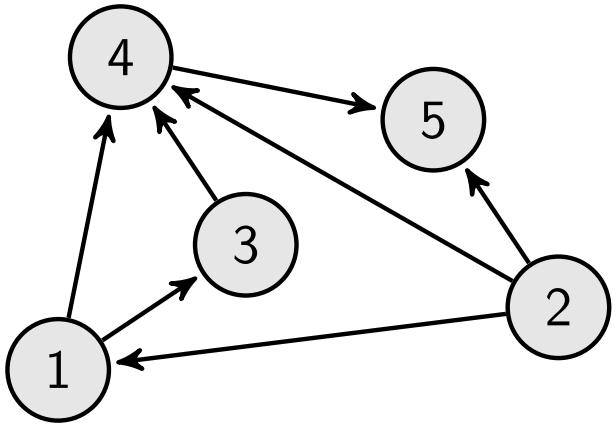


有向グラフと無向グラフ

無向グラフ
辺に向きがないグラフ



有向グラフ
辺に向きがあるグラフ



グラフの表現

隣接行列

- ⋆ 頂点 v,w 間の枝の本数を，行列の (v, w) 成分で表す
- 領域計算量 $O(n^2)$
- v,w 間に枝があるか？ $O(1)$
- v に接続する枝の列挙． $O(n)$
- 密なグラフを表すのに適する

隣接リスト

- ⋆ 各頂点に対してどの頂点への枝があるかを覚える
- 領域計算量 $O(n + m)$
- v,w 間に枝があるか？ $O(d)$
- v に接続する枝の列挙． $O(d)$
- 疎なグラフを表すのに適する

ただし，d は v から出ている枝の数（次数) である

グラフの表現 — 例

隣接行列

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

隣接リスト

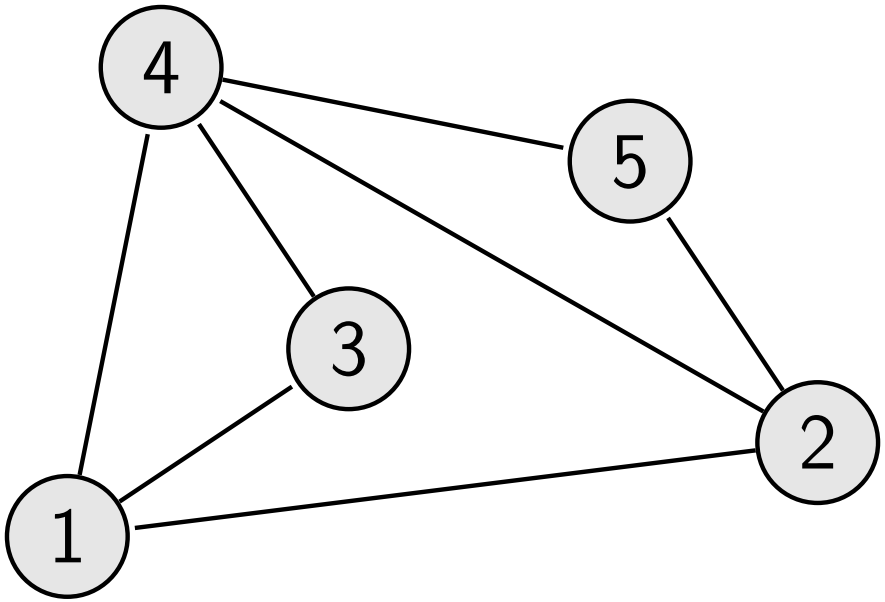
1 [2,3,4]

2 [1,4,5]

3 [1,4]

4 [1,2,3,5]

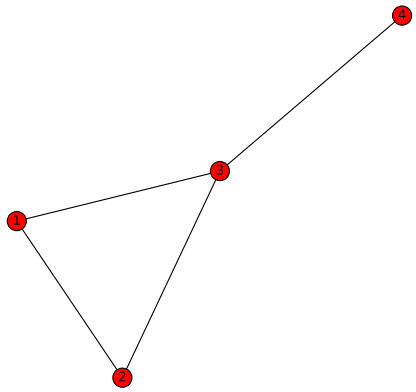
5 [2,4]



おまけ：グラフライブラリ

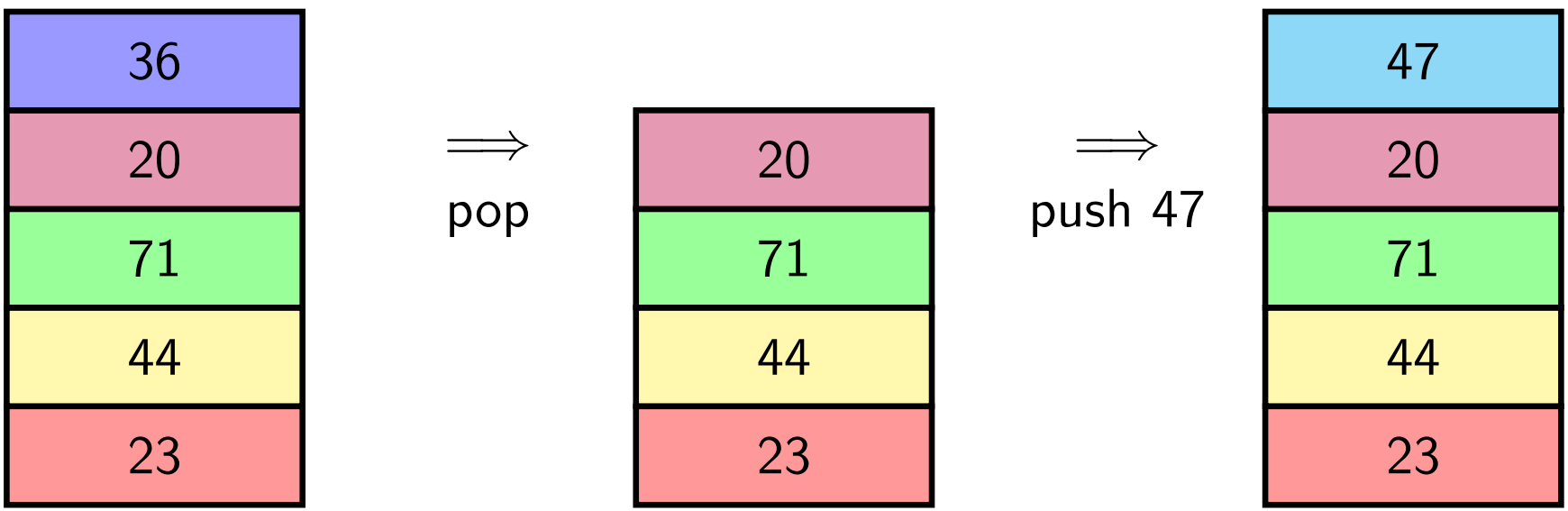
- グラフに対する本格的な処理を行いたい場合は networkx ライブラリが便利です
- https://networkx.github.io/index.html
- 例えば，以下のコードでグラフを描画することができます

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()

G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(3,1)
G.add_edge(3,4)
pos = nx.spring_layout(G)
nx.draw(G,pos,with_labels=True)
plt.show()
```



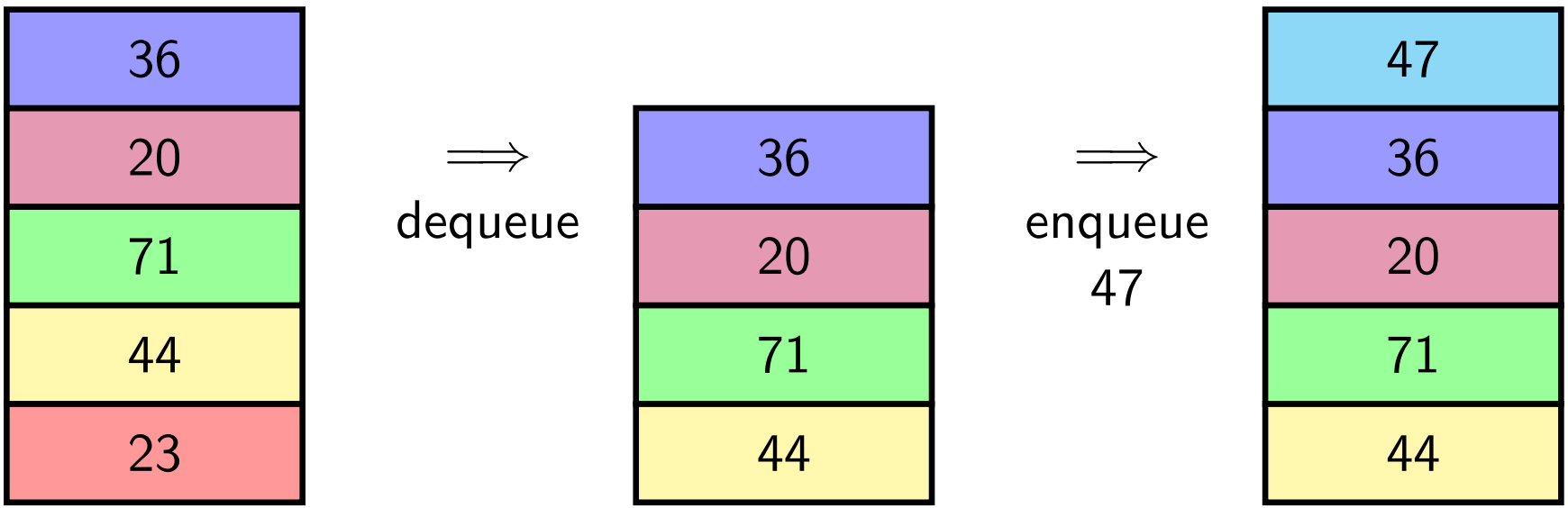
スタック (stack)

- データを先入れ後出し (FILO) の構造で保持するもの
- 2 つの基本操作をもつ
  - push: データを上に積む，計算量は $O(1)$
  - pop: 一番上のデータを取り出す，計算量は $O(1)$
- Python ではリストの append と pop を用いることでスタックとなる



キュー (queue)

- データを先入れ先出し (FIFO) の構造で保持するもの
- 2 つの基本操作をもつ
  - enqueue: キューにデータを入れる，計算量は $O(1)$
  - dequeue: キューからデータを出す，計算量は $O(1)$
- Python では次スライドの deque を用いる
- リストの append と pop(0) を使うと遅い



両端キュー (deque)

- スタックとキューを兼ね備えたデータ構造
- Python では from collections import deque で使えるようになる
  - append: 新しい要素を右につけたす (push, enqueue に対応)
  - appendleft: 新しい要素を左につけたす
  - pop: 一番右の要素を削除し返す
  - popleft: 一番左の要素を削除し返す (dequeue に対応)

```
>>> from collections import deque
>>> d = deque([1,2,3,4,5])
>>> d
deque([1, 2, 3, 4, 5])
>>> d.append(0)
>>> d
deque([1, 2, 3, 4, 5, 0])
>>> d.pop()
0
>>> d.popleft()
1
>>> d
deque([2, 3, 4, 5])
```

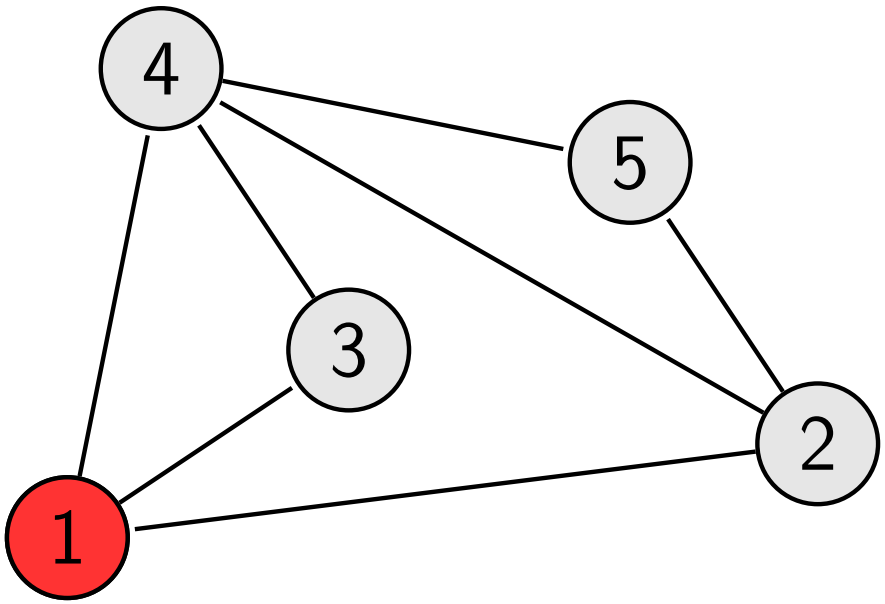
深さ優先探索 (Depth First Search)

- スタック (FILO) を用いたグラフ探索
- 次に訪れる候補をスタックに積んでいく
- 計算量は $O(m+n)$
- 再帰関数によって簡単に書けることが多い

```
adj = {1:[2,3,4], 2:[1,4,5], 3:[1,4], 4:[1,2,3,5], 5:[2,4]}

prev = {}
stack = [(1,'start')]
i = 0
while len(stack)>0:
    (v,p) = stack.pop()
    if v in prev: continue
    prev[v] = p
    for u in adj[v]:
        if u not in prev:
            stack.append((u,v))
print(prev) # {1: 'start', 2: 5, 3: 4, 4: 1, 5: 4}
```

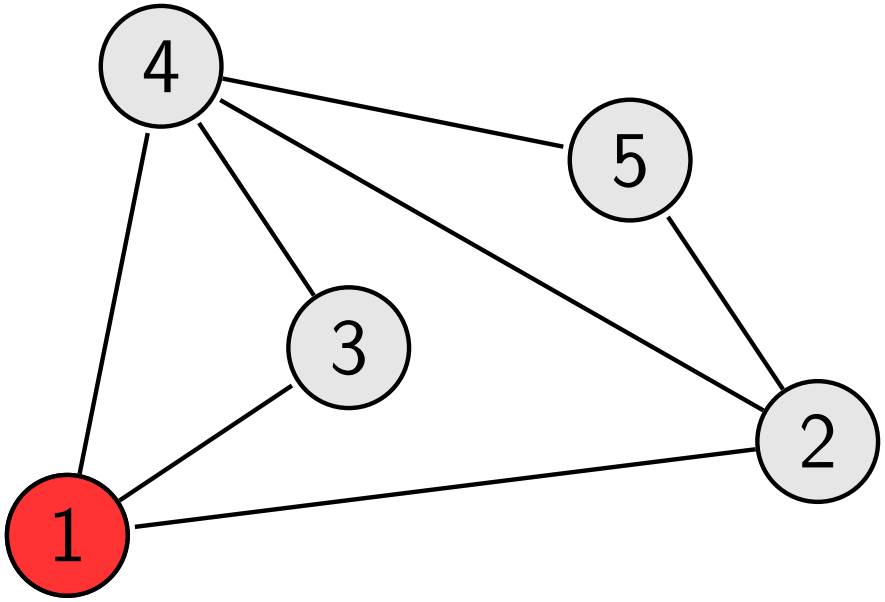
深さ優先探索

- stack=[]
- $v = 1$
-



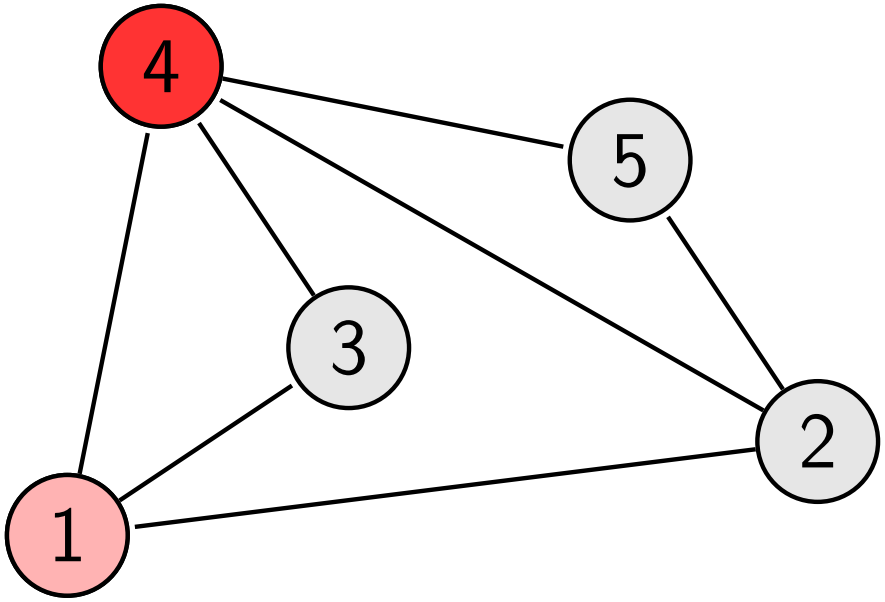
深さ優先探索

- stack=[2, 3, 4]
- $v = 1$
- push 2, 3, 4



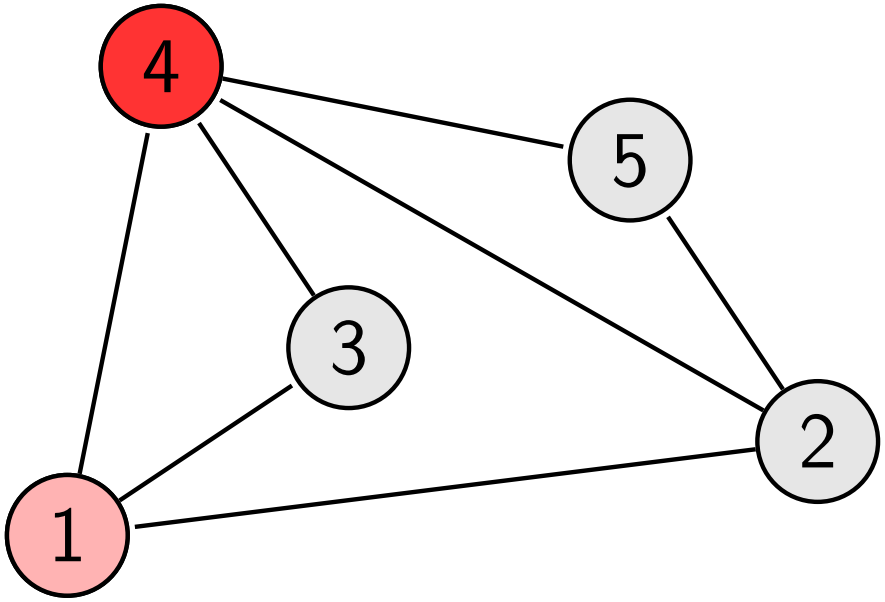
深さ優先探索

- stack=[2, 3]
- $v = 4$
- pop



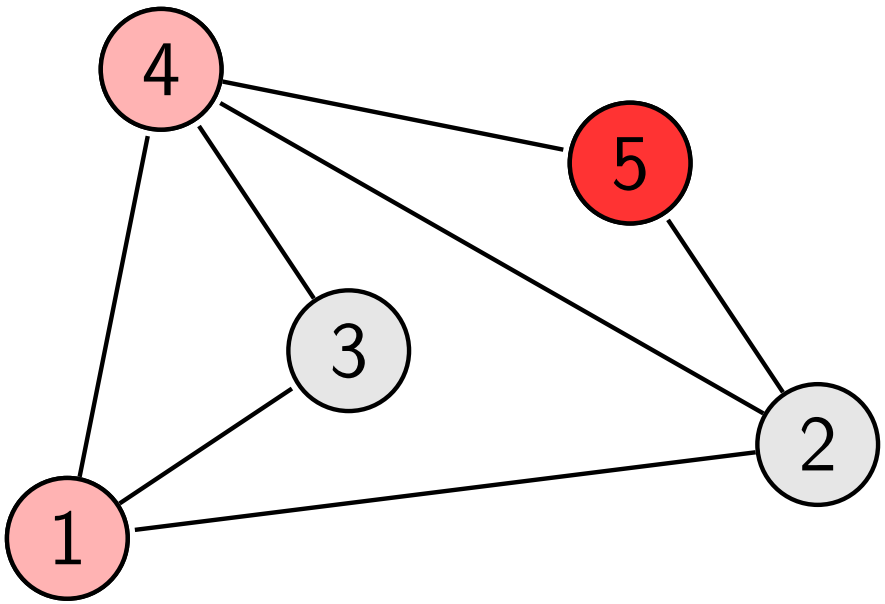
深さ優先探索

- stack=[2, 3, 2, 3, 5]
- $v = 4$
- push 2,3,5



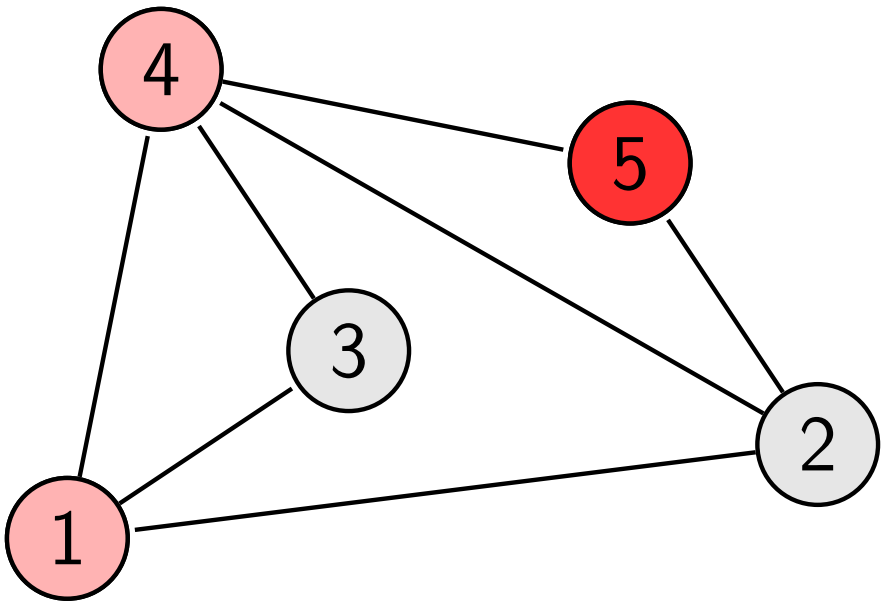
深さ優先探索

- stack=[2, 3, 2, 3]
- $v = 5$
- pop



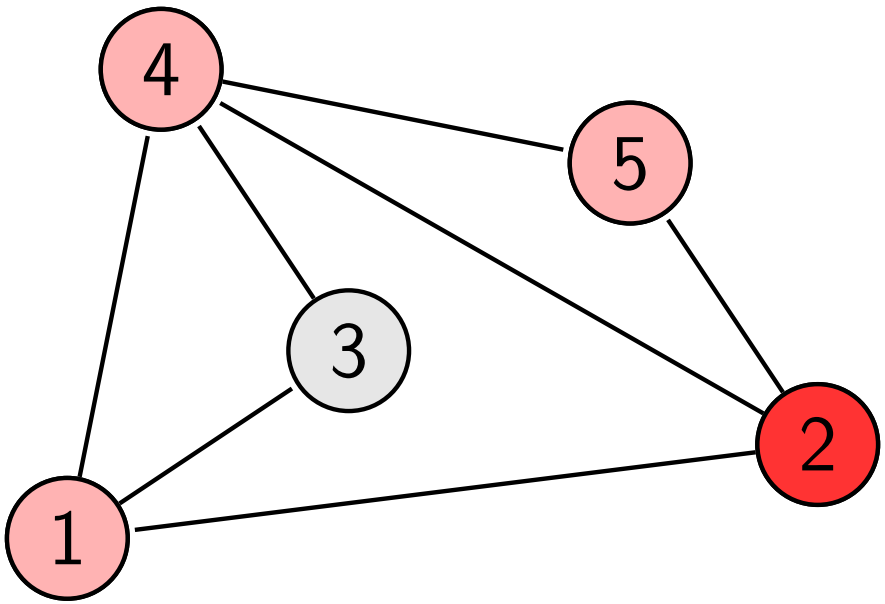
深さ優先探索

- stack=[2, 3, 2, 3, 2]
- $v = 5$
- push 2



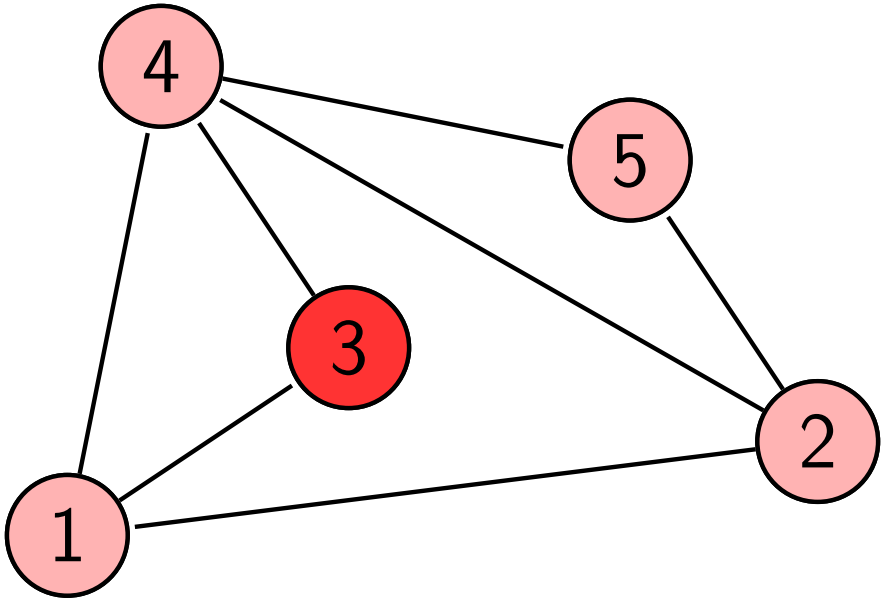
深さ優先探索

- stack=[2, 3, 2, 3]
- $v = 2$
- pop



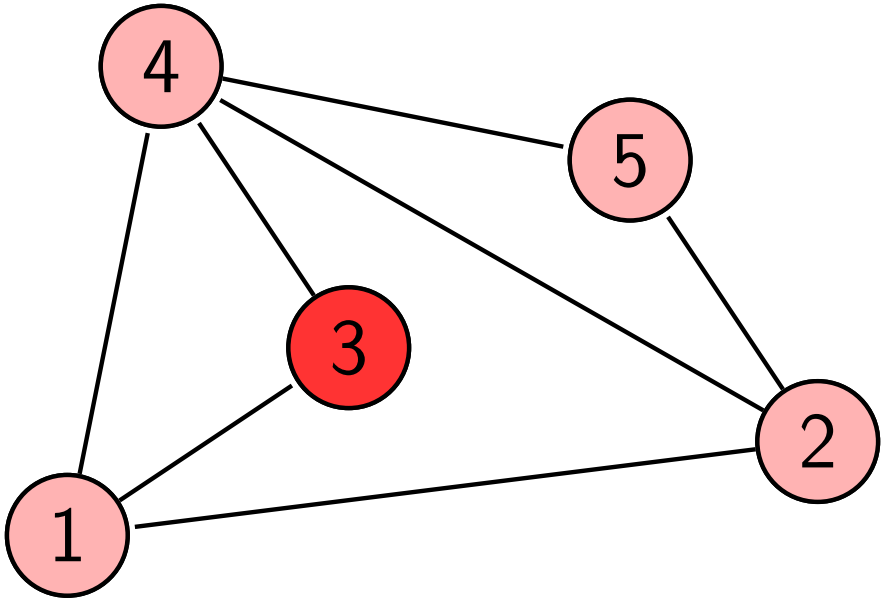
深さ優先探索

- stack=[2, 3, 2]
- $v = 3$
- pop



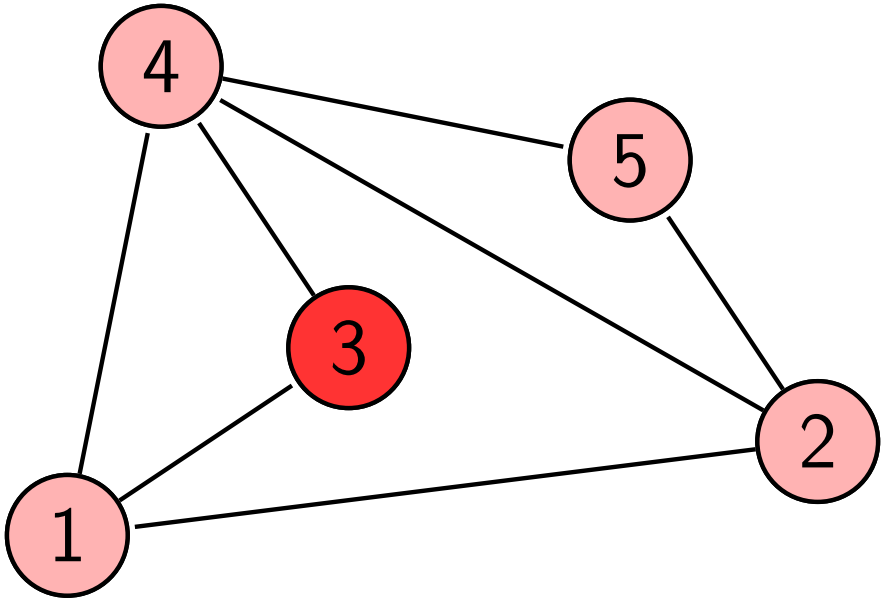
深さ優先探索

- stack=[2, 3]
- $v = 3$
- pop



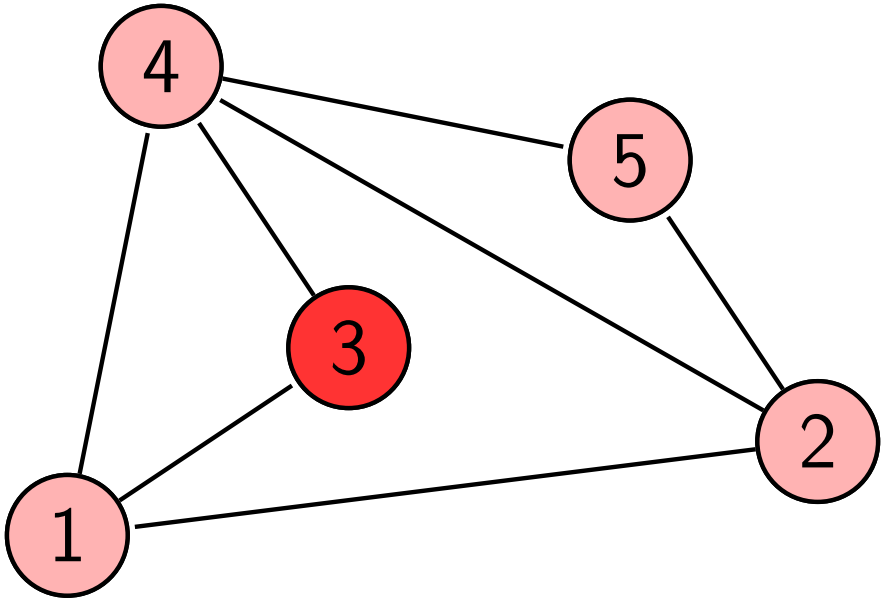
深さ優先探索

- stack=[2]
- $v = 3$
- pop



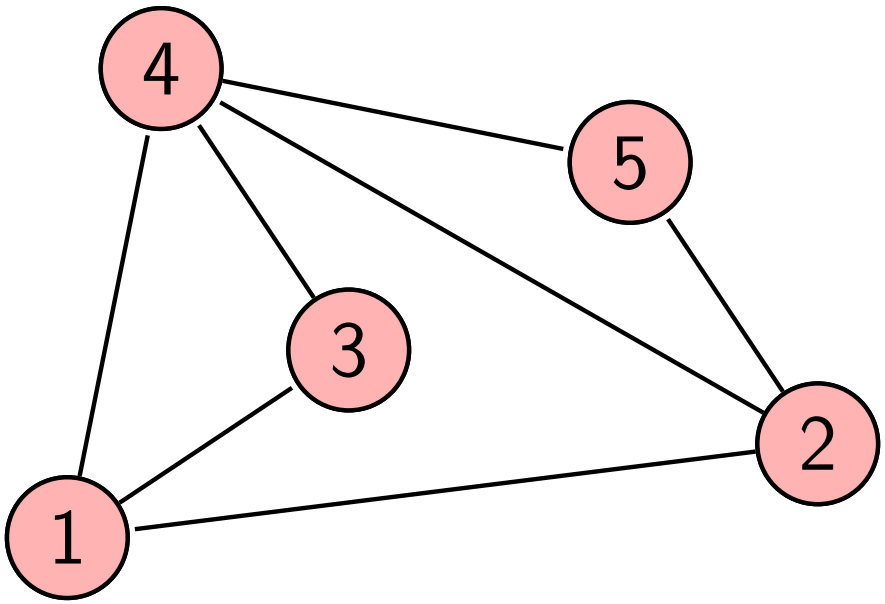
深さ優先探索

- stack=[]
- $v = 3$
- pop



深さ優先探索

- stack=[]
- $v = 3$
-



再帰関数を用いた深さ優先探索

- 再帰関数を用いることで深さ優先探索は実装できる
- スタックを用いるよりも簡単に書けることが多い

```
adj = {1:[2,3,4], 2:[1,4,5], 3:[1,4], 4:[1,2,3,5], 5:[2,4]}

prev = {}
def dfs(v,p):
    if v in prev: return
    prev[v] = p
    for u in adj[v]:
        if u not in prev:
            dfs(u,v)
dfs(1,'start')
print(prev) # {1: 'start', 2: 1, 3: 4, 4: 2, 5: 4}
```

深さ優先探索の応用

迷路のスタートからゴールまでたどり着けるか？

- S: スタート
- G: ゴール
- #: 壁
- .: 通路

maze.txt

```
....#.....
.##.#.###.
.#..#.#...
.#.#..###.
.##..#....
....##.###
.###.#.#.#
.#........
.#.#######
S#.......G
```

再帰関数を用いた深さ優先探索

maze_dfs.py

```
# file 読み込み
f = open('maze.txt','r')
field = [list(l.rstrip()) for l in f]
f.close()

w = len(field[0])
h = len(field)
for i in range(h):
    for j in range(w):
        if field[i][j]=='S': s=(i,j)
        if field[i][j]=='G': g=(i,j)

# 再帰関数による深さ優先探索
prev = {}
def dfs(v,p):
    if v in prev: return
    (x,y)=v
    if (0<=x<w) and (0<=y<h) and field[x][y]!='#':
        prev[v]=p
        for (dx,dy) in [(1,0),(-1,0),(0,1),(0,-1)]:
            dfs((x+dx,y+dy),v)

dfs(s,'start')
```

再帰関数を用いた深さ優先探索

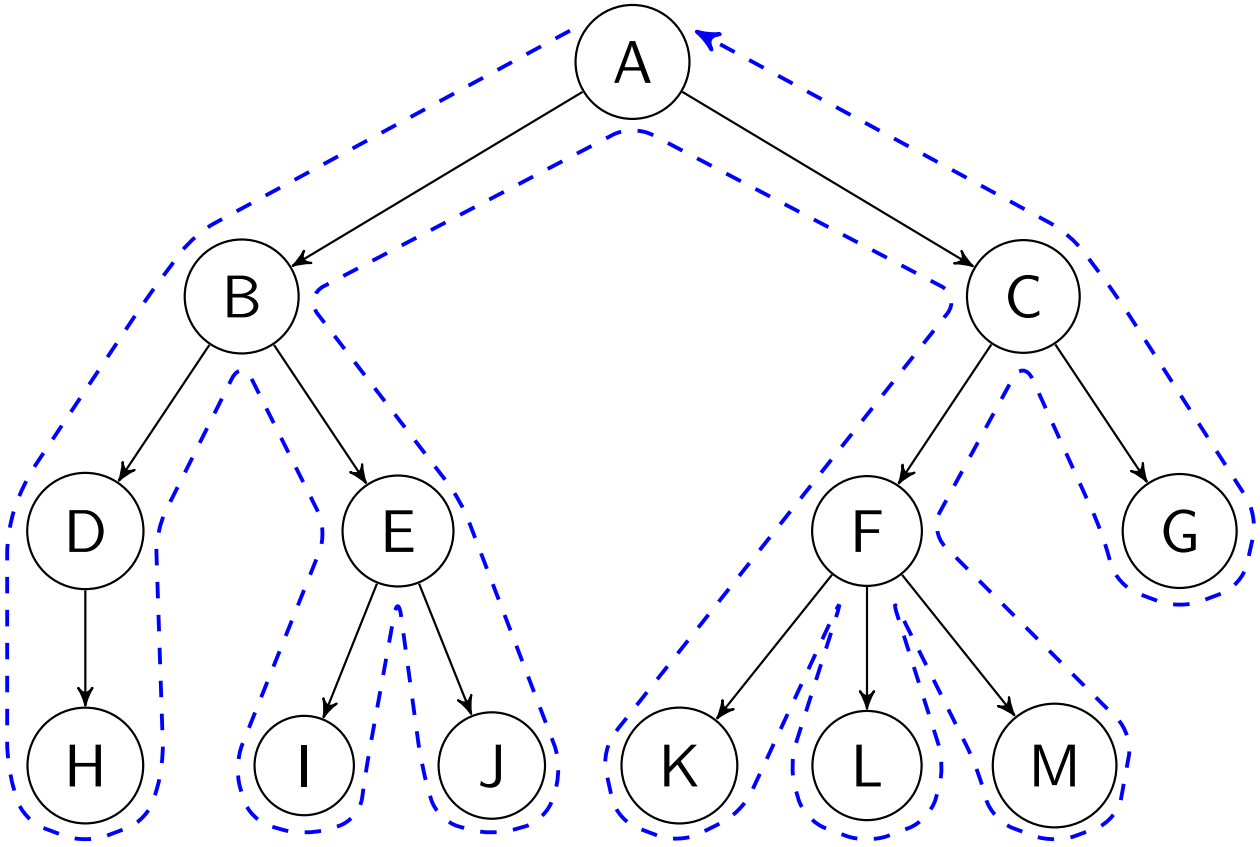
maze_dfs.py の続き

```
def print_field():
    for l in field:
        print(''.join(l))
    print()

print_field()
v=g
while v!='start':
    (i,j)=v
    field[i][j]='x'
    v=prev[v]
print_field()
```

```
....#.....
.##.#.###.
.#..#.#...
.#.#..###.
.##..#....
....##.###
.###.#.#.#
.#........
.#.#######
S#.......G
```

```
....#xxxxx
.##.#x###x
.#..#x#..x
.#.#xx###x
.##xx#xxxx
xxxx##x###
x###.#x#.#
x#xxxxx...
x#x#######
x#xxxxxxxx
```

深さ優先探索の探索順



- 行きがけ順 (pre-order): A,B,D,H,E,I,JC,F,K,L,M,G
- 帰りがけ順 (post-order): H,D,I,J,E,B,K,L,M,F,G,C,A

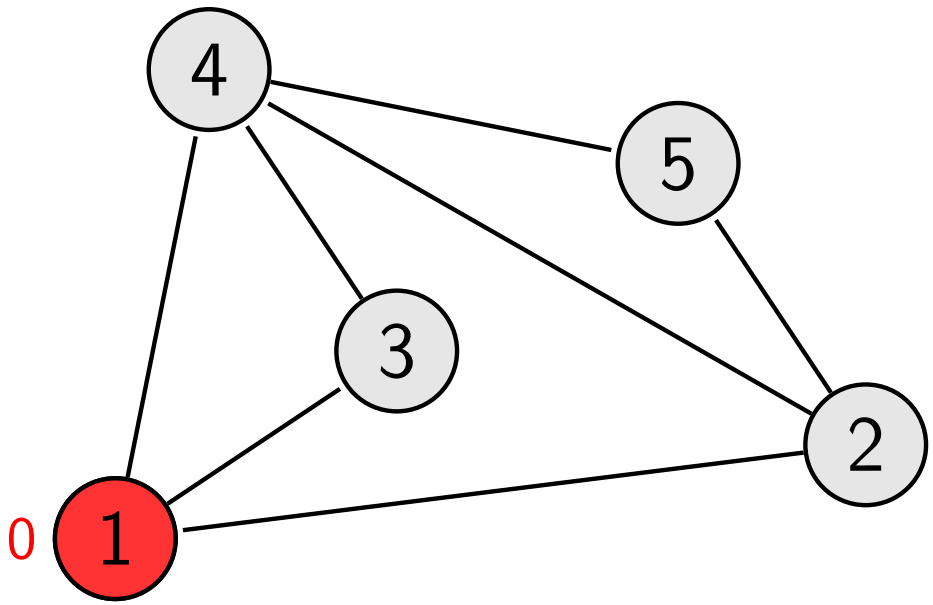
幅優先探索 (Breadth First Search)

- キュー (FIFO) を用いたグラフ探索
- 次に訪れる候補をキューに積んでいく
- 計算量は $O(m+n)$
- 距離を求めることもできる

```
from collections import deque
adj = {1:[2,3,4], 2:[1,4,5], 3:[1,4], 4:[1,2,3,5], 5:[2,4]}

dist = {}
queue = deque([(1,0)]) # 1 までの距離が 0
while len(queue)>0:
    v, d = queue.popleft()
    if v in dist: continue
    dist[v] = d
    for u in adj[v]:
        if u not in dist:
            queue.append((u,d+1))
print(dist) # {1: 0, 2: 1, 4: 1, 5: 2}
```

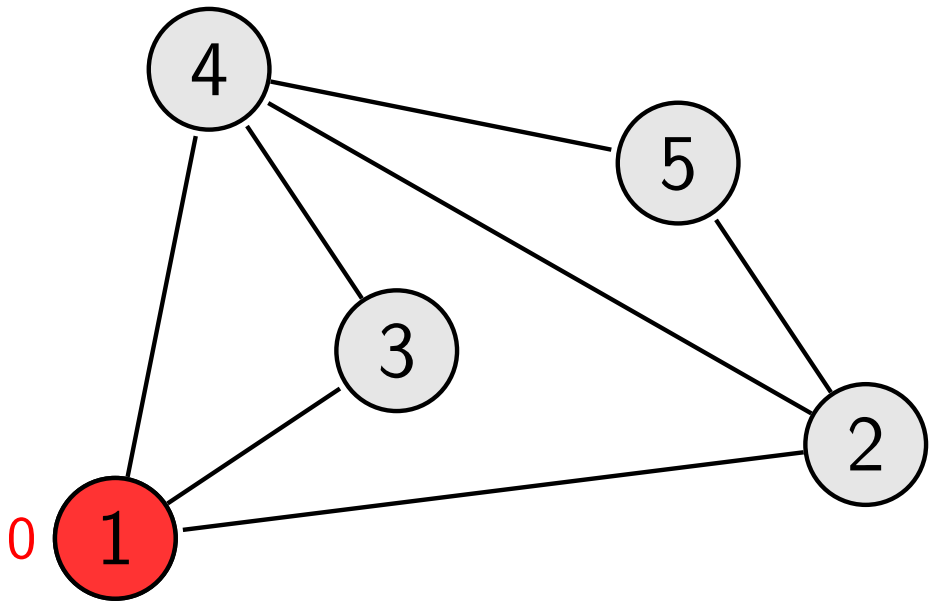
幅優先探索

- queue=[]
- $(v, d) = (1, 0)$
-



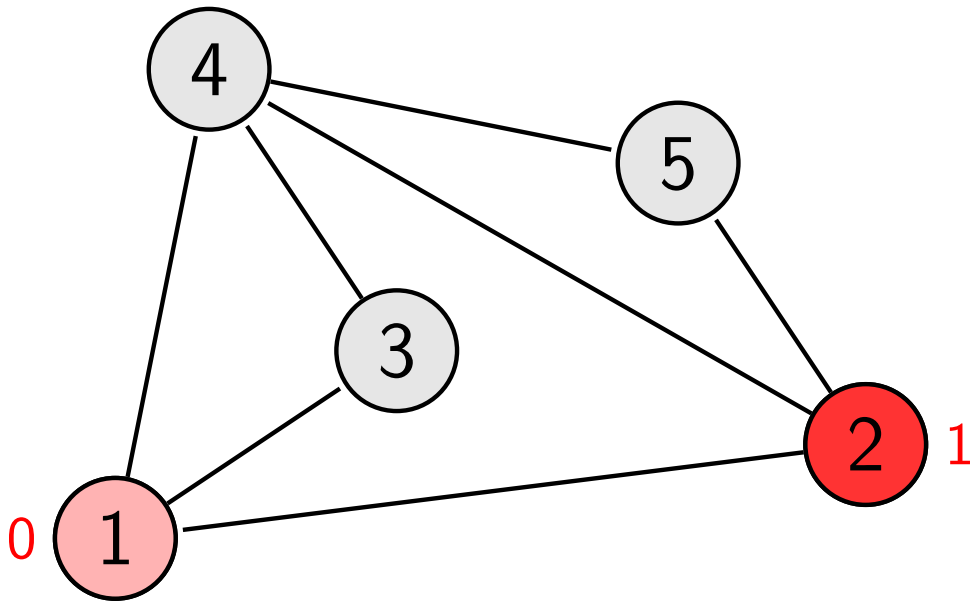
幅優先探索

- queue=$[(2,1),(3,1),(4,1)]$
- $(v,d)=(1,0)$
- enqueue $2,3,4$



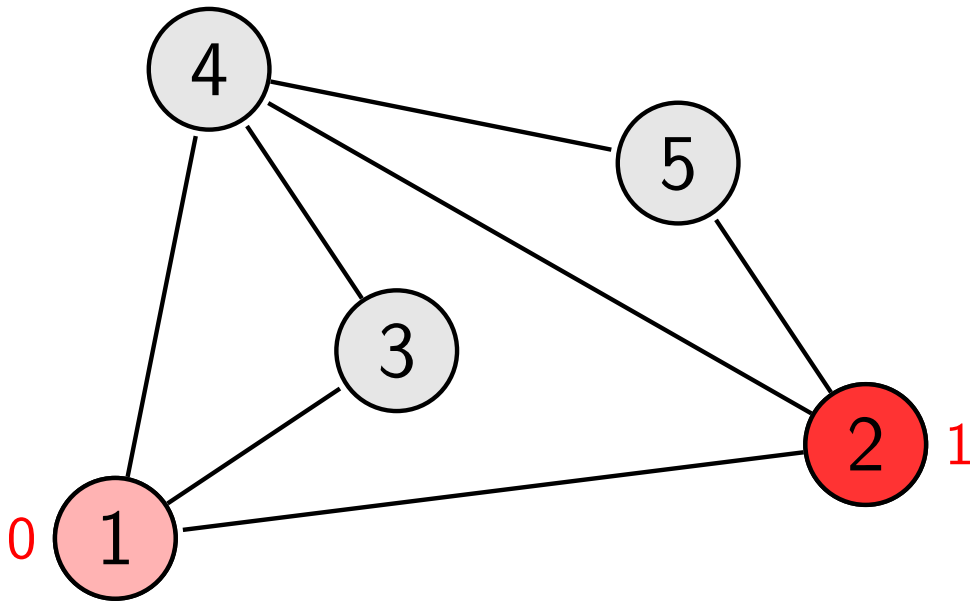
幅優先探索

- queue=$[(3,1),(4,1)]$
- $(v,d)=(2,1)$
- dequeue



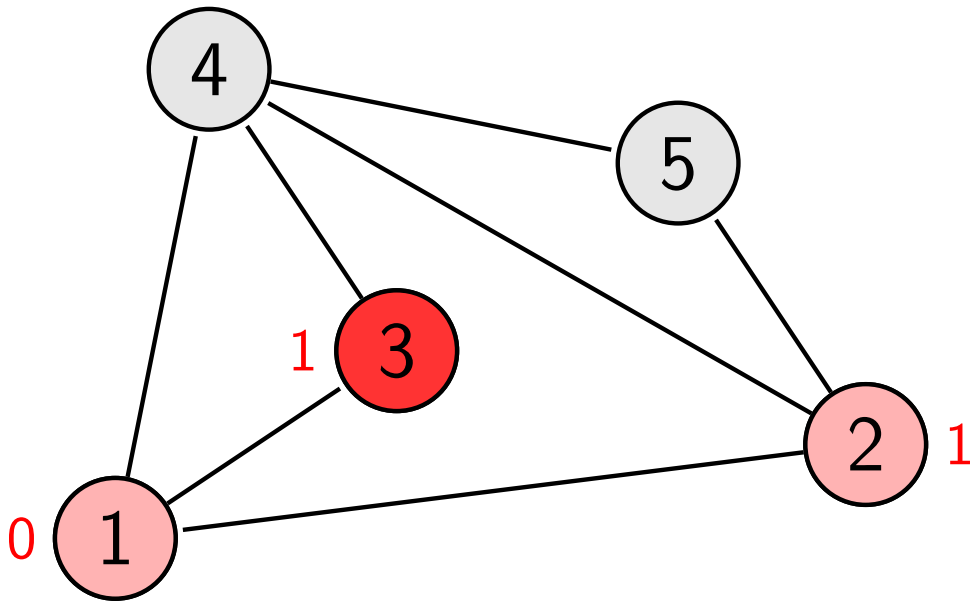
幅優先探索

- queue=$[(3,1),(4,1),(4,2),(5,2)]$
- $(v,d)=(2,1)$
- enqueue $4,5$



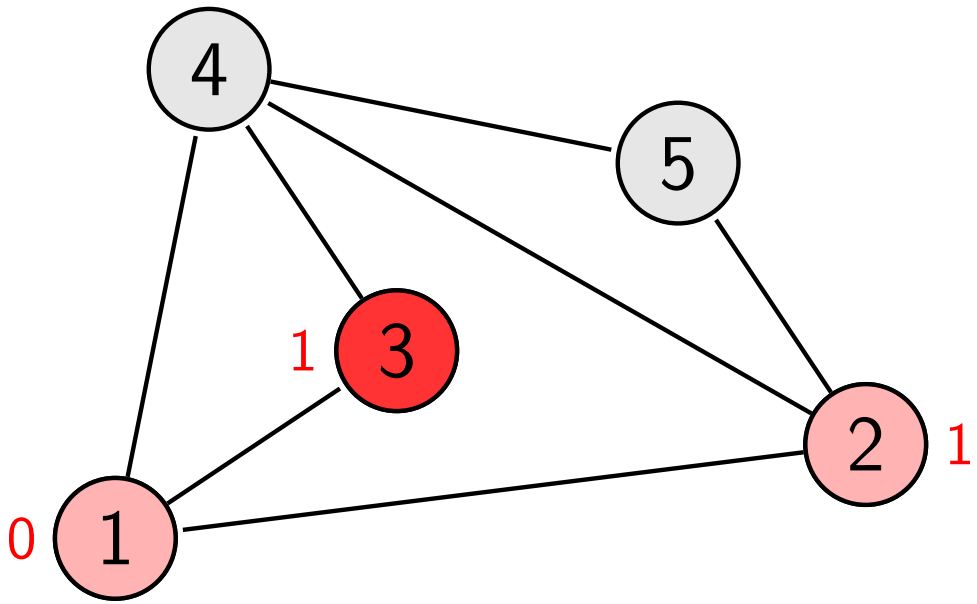
幅優先探索

- queue=$[(4,1),(4,2),(5,2)]$
- $(v,d)=(3,1)$
- dequeue



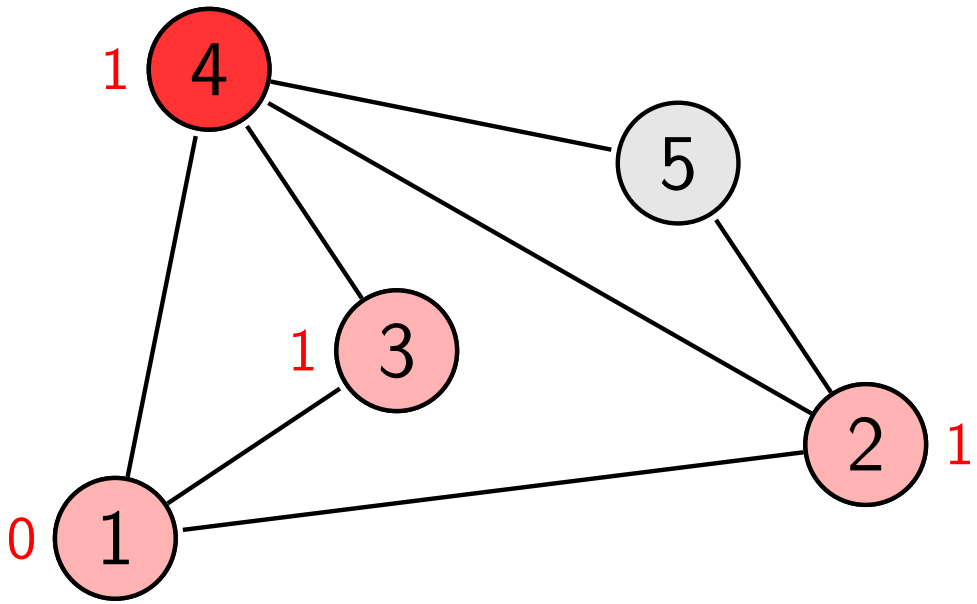
幅優先探索

- queue=$[(4,1),(4,2),(5,2),(4,2)]$
- $(v,d)=(3,1)$
- enqueue 4



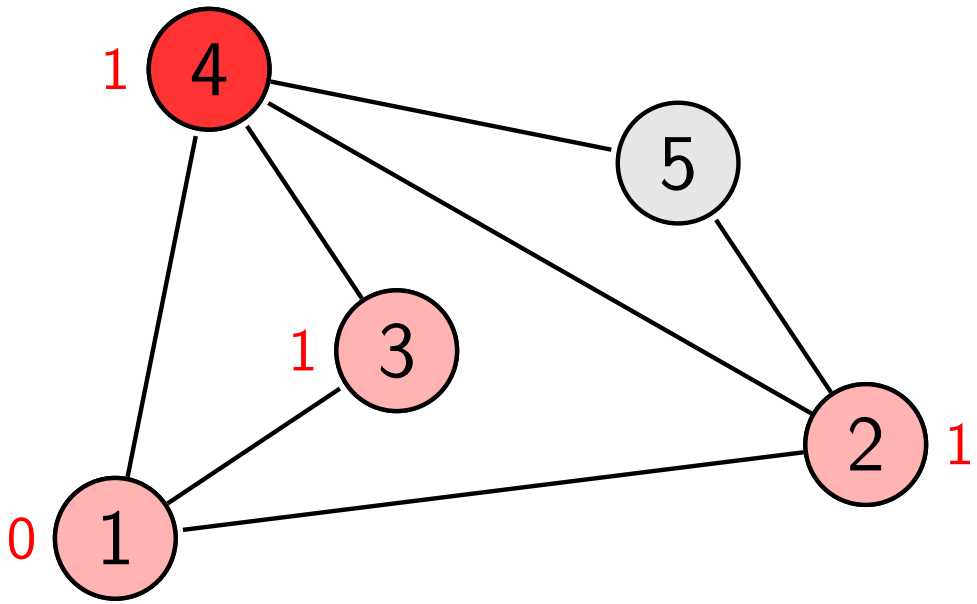
幅優先探索

- queue=$[(4,2),(5,2),(4,2)]$
- $(v,d)=(4,1)$
- dequeue



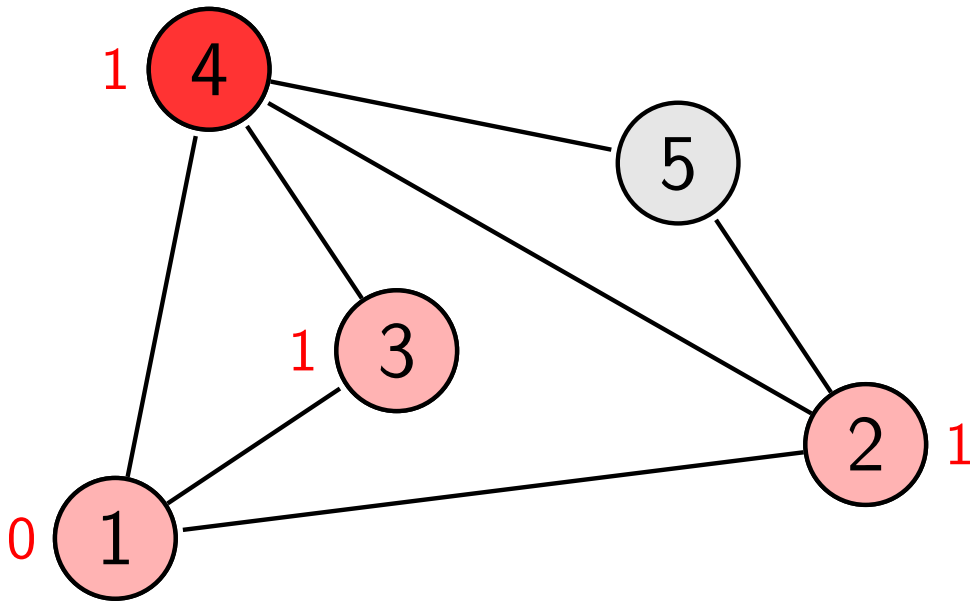
幅優先探索

- queue=$[(4,2),(5,2),(4,2),(5,2)]$
- $(v,d)=(4,1)$
- enqueue 5



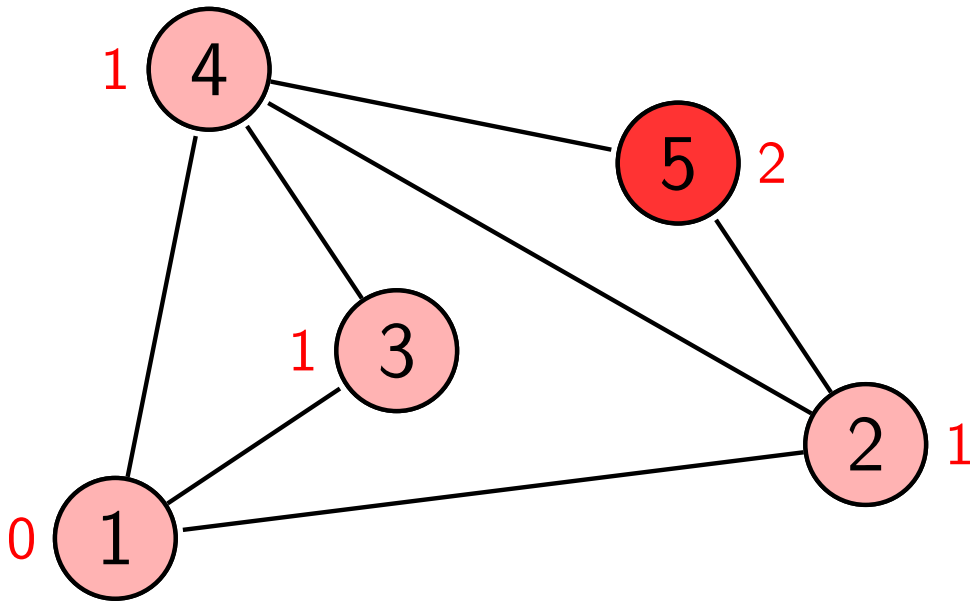
幅優先探索

- queue=$[(5,2),(4,2),(5,2)]$
- $(v,d)=(4,1)$
- dequeue



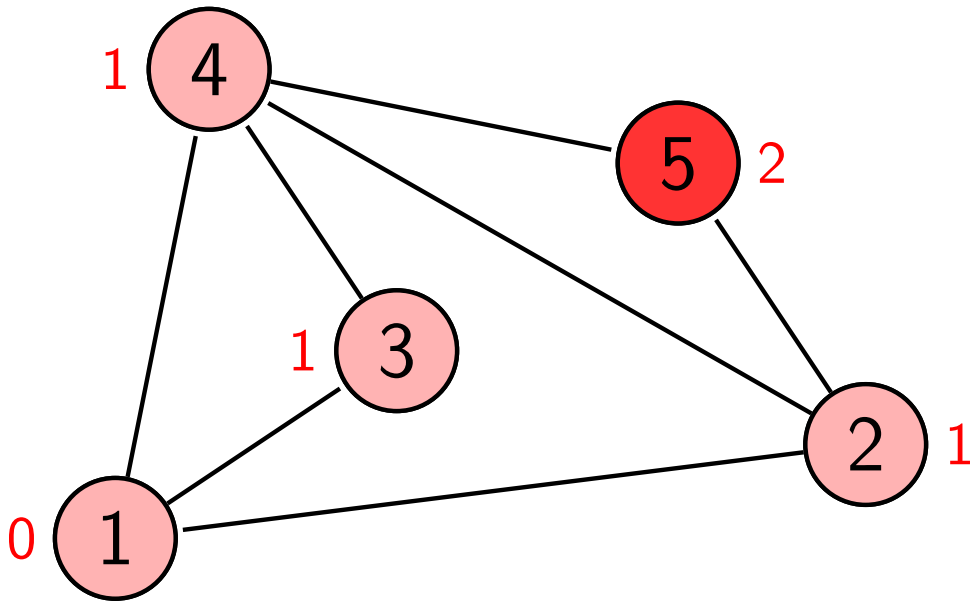
幅優先探索

- queue=$[(4,2),(5,2)]$
- $(v,d)=(5,2)$
- dequeue



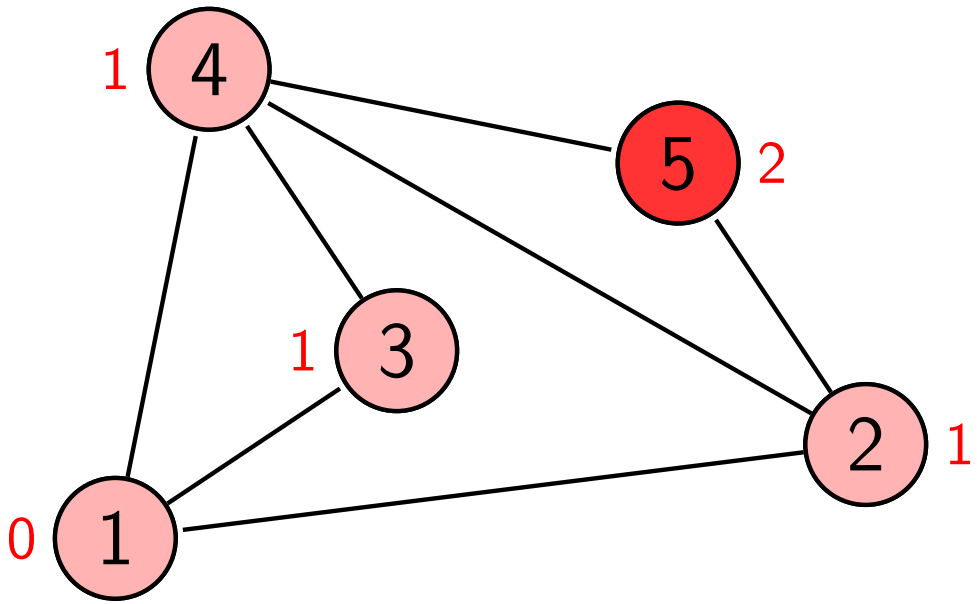
幅優先探索

- queue=[(5, 2)]
- $(v, d) = (5, 2)$
- dequeue



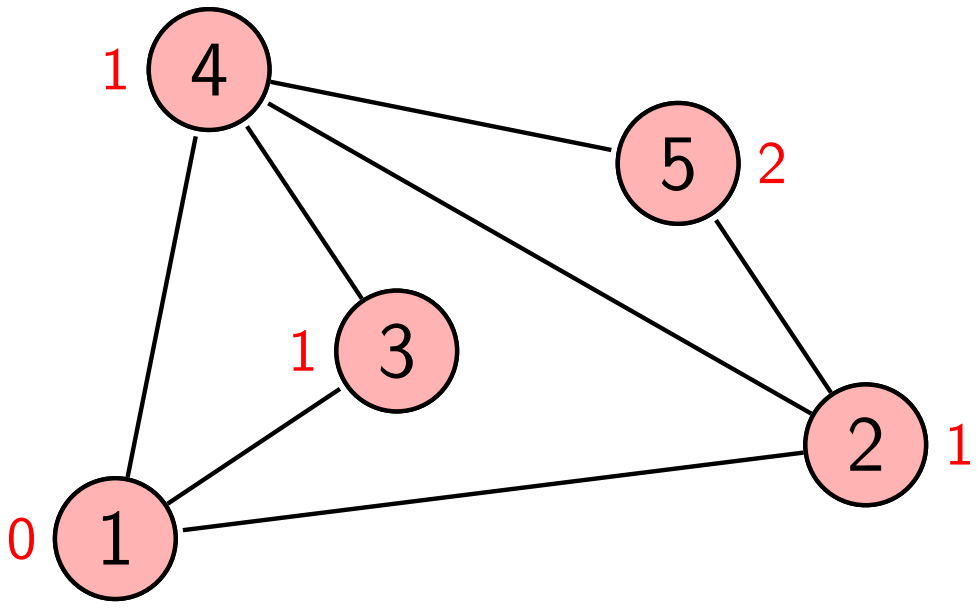
幅優先探索

- queue=[]
- $(v, d) = (5, 2)$
- dequeue



幅優先探索

- queue=[]
- $(v, d) = (5, 2)$
-



幅優先探索の応用

迷路のスタートからゴールまでの最短ルートは何か？

- S: スタート
- G: ゴール
- #: 壁
- .: 通路

maze2.txt

```
....#.....
.##.#.###.
.#....#...
.#.#..###.
.##..#....
....##.#.#
.###...#.#
.#...#....
.#.#######
S#.......G
```

幅優先探索による迷路の探索

maze_bfs.py

```
from collections import deque

f = open('maze2.txt','r')
# 読み込み部は maze_dfs.py と同じなので省略

prev = {}
dist = {}
queue = deque([(s,'start',0)])
while len(queue)>0:
    v, p, d = queue.popleft()
    if v in dist: continue
    (x,y)=v
    if (0<=x<w) and (0<=y<h) and field[x][y]!='#':
        dist[v] = d
        prev[v] = p
        for (dx,dy) in [(1,0),(-1,0),(0,1),(0,-1)]:
            queue.append(((x+dx,y+dy),v,d+1))
```

幅優先探索による迷路の探索

maze_bfs.py の続き

```
def print_field():
    for l in field:
        print(''.join(l))
    print()

print_field()
v=g
while v!='start':
    (i,j)=v
    field[i][j]=str(dist[v]%10)
    v=prev[v]
print_field()
```

```
....#.....
.##.#.###.
.#....#...
.#.#..###.
.##..#....
....##.#.#
.###...#.#
.#...#....
.#.#######
S#.......G
```

```
....#45678
.##.#3###9
.#..12#..0
.#.#0.###1
.##89#5432
4567##6#.#
3###987#.#
2#210#....
1#3#######
0#45678901
```

深さ優先探索と幅優先探索

深さ優先探索

- スタックを用いる
- 比較的書きやすいことが多い
- メモリは比較的使わない
- 最短路は分からない

幅優先探索

- キューを用いる
- 比較的書きにくいことが多い
- メモリを比較的多く使う
- 最短路が分かる

アウトライン

演習問題提出方法

解答プログラムをまとめたテキストファイルを作成して，OCW-i で提出

- ファイル名は practice4.txt
- 次回授業の開始時間が締め切り
- ファイルの最初に学籍番号と名前を書く
- どの演習問題のプログラムかわかるように記述
- 出力結果もつける（描画する問題の場合はどのような結果が得られたか一言で説明）
- 途中までしかできなくても，どこまでできてどこができなかったかを書けば部分点を付けます
- 問 1 の出力結果は PDF で提出してください．

演習問題 (1/2)

問 1

次の関数の計算量のオーダーを求めよ．また，$n = 1000, 2000, \ldots, 10000$ の場合について実行時間を計測し，matplotlib を用いてプロットせよ．さらに，関数 ax^b を用いて，最小二乗法により計測結果を近似した結果もプロットせよ．

```
def calc(n):
    res = 0
    for i in range(n):
        for j in range(i):
            res+=j
    return res
```

問 2

次の迷路について，スタートから到達可能なマスの数と，スタートからゴールまでの最短距離を求めよ．

http://yambi.jp/lecture/advanced_programming2018/maze3.txt

演習問題 (2/2)

問 3 (おまけ)

大きさが $h \times w$ マスの庭がある．そこに雨が降り，水たまりができたとするとき，全部でいくつの水たまりがあるか数えたい．ただし，水たまりは 8 近傍で隣接している場合につながっているとみなす．

入力例

```
W........WW.
.WWW.....WWW
.W..WW...WW.
..W......WW.
........WW..
..W......W..
.W.W......W.
W.W.WW...W.W
..WW........
```

W は水たまりを表すとする．
この例の答えは 3 個である．

次の入力について数えよ．

http://yambi.jp/lecture/advanced_programming2018/lake.txt