```
student_number = XXXXXXXX
```

**2人でワードカウント！**

ある文章に含まれる単語を数えたい。文章は長いので、二人（二台のパソコン）で協力することにする。{単語: 出現数}の形式の辞書を返す関数word_countと、複数の結果をまとめる関数reduceを書いてください。

例：

```
wc1 = word_count("I think that that that is this")
wc2 = word_count("I think that this this is that")
reduced = reduce(wc1, wc2)
print(wc1, wc2)
# {'I': 1, 'think': 1, 'that': 3, 'is': 1, 'this': 1}
# {'I': 1, 'think': 1, 'that': 2, 'this': 2, 'is': 1}
print(reduced)
# {'I': 2, 'think': 2, 'that': 5, 'is': 2, 'this': 3}
```

＋ Code ＋ Text

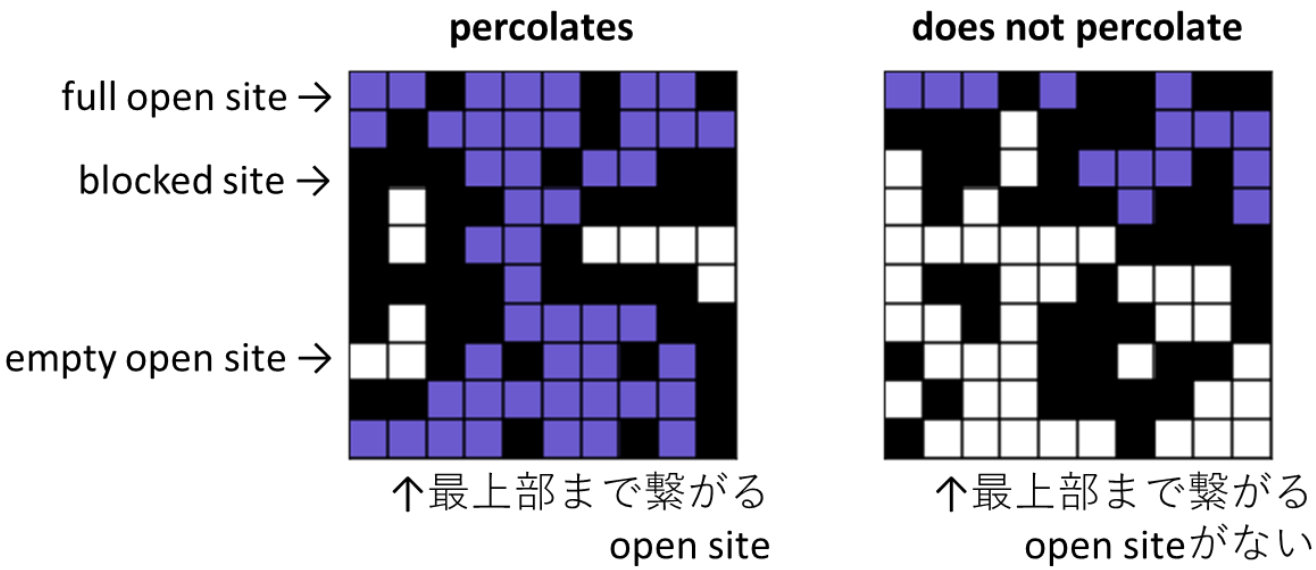
```
#%%
def word_count(string):
    """
    引数： string: str
    返値： dict[str, int]
    """
    pass

def reduce(wc1, wc2):
    """
    引数： wc1: dict[str, int], wc2: dict[str, int]
    返値： dict[str, int]
    """
    pass
```

## 総合演習：Percolation

この演習では、n x n の格子状のsiteとしてモデル化されるパーコレーションシステムについて取り上げます。モンテカルロ法を用いて、パーコレーションシステムのシミュレーションをしてみましょう！

さて、パーコレーションシステムでは、それぞれのsiteは塞がれている(blocked)か開放状態(open)のいずれかをとります。Full siteとは、隣接するopen siteをたどっていけば最上部のopen siteに接続されるようなsiteのことです。以下の図を参照してください。




もし、最下部のsiteのいずれかがfull siteであれば、その系はパーコレートしていると定義します。

この演習では、「もしそれぞれのsiteが開放率pによって一様にiid（独立同分布）で開放状態になるとき、その系がパーコレートする確率は何か」を求めます。実は、この問題を数学で解析的に解く方法は未だ解明されていません。

## データ表現

siteの状態は、以下のデータにより表現します。

- **isOpen[ ][ ]**: Bool型の行列です。siteが開放状態の時Trueで表現します。
- **isFull[ ][ ]**: siteがfullであるかどうか示すBool型行列です。

どちらの行列も、isOpen[0][x]は最上部の行を指し、isOpen[0][0]が左上のsiteを指すように行列を表現します。

今回の問題では、isOpen行列はこちらで用意したgenerate_percolator(n, p)で生成します。nは行列の大きさ(n x n)、pは開放率です。isOpen行列が与えられたとき、その系がパーコレートするか判断したいとき、どうすればいいでしょう？この課題では、それを導くため、シミュレーションによりisFullを完成させます。

```
#%%
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import sys
sys.setrecursionlimit(1500)
SEED = None

def generate_percolator(n, p):
    if SEED:
        np.random.seed(SEED)
```

```
    return np.random.binomial(1, p, n*n).reshape(n, n).astype(bool).tolist()

def PercolationVisualizer(isOpen, isFull):
    isOpen = np.array(isOpen)
    isFull = np.array(isFull)
    if any(isOpen[isFull == 1] == 0):
        raise ValueError("A closed cell is full.")

    data = isOpen.astype(float)
    data[np.logical_and(isFull == 1, isOpen == 1)] = np.nan
    data[np.logical_and(isFull == 1, isOpen == 0)] = 0.5

    current_cmap = mpl.cm.get_cmap('gray') #.copy()
    current_cmap.set_bad('slateblue', 1.)
    current_cmap.reversed()
    plt.imshow(data, cmap='gray', vmin=0, vmax=1, interpolation='none', aspect='equ

    ax = plt.gca()
    ax.set_xticks(np.arange(isOpen.shape[0]))
    ax.set_yticks(np.arange(isOpen.shape[1]))
    ax.set_xticks(np.arange(-.5, isOpen.shape[0], 1), minor=True)
    ax.set_yticks(np.arange(-.5, isOpen.shape[1], 1), minor=True)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
    ax.grid(which='minor', color='black', linestyle='-', linewidth=1.5)

    plt.show()
```

## Vertical Percolation

手始めに、単純な問題を考えます。isOpen行列が与えられたとき、垂直に開放状態のsiteが貫通しているか（垂直にパーコレートしているか）判断します。


## vertical_flow(isOpen)

まず、isOpen行列を引数にとり、上記のvertical percolationのルールに従ってisFull行列を生成するvertical_flow(isOpen)関数を書いてください。まずは、isFull行列を第7回例題でやった二次元リストの初期化方法を思い出し、isOpen行列と同じ大きさの二次元行列を作り、Falseで初期化するところから始めてみましょう。

**PercolationVisualizer**

今回の演習の一助とするため、PercolationVisualizerを準備しました。isOpen及びvertical_flowの結果のisFullを入力すると、状態をビジュアル化します。ただし、isOpenのblocked siteがisFullでfullであると指定されると、例外(Exception)が返ってきます。

## vertically_percolates(isOpen)

続いて、isOpenを受け取り、vertical_flowを実行したうえで、isFullの最下部の列を検査し、パーコレートしているかどうかのBoolを返す関数vertically_percolatesを書いてください。

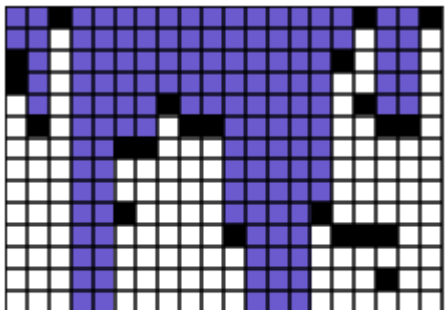
**run_vertical_percolation_simulation** は、上記二つの関数をテストするために設けられた単発シミュレーションです。pythonファイル上部のSEEDを指定すると、毎回同じ系が生成されるようになります。

```
#%%
def vertical_flow(isOpen):
    """
    引数：isOpen: list[list[bool]]
    返値：list[list[bool]]
    """
    pass

def vertically_percolates(isOpen):
    """
    引数：isOpen: list[list[bool]]
    返値：bool
    """
    pass

def run_vertical_percolation_simulation():
    isOpen = generate_percolator(20, 0.9)
    PercolationVisualizer(isOpen, vertical_flow(isOpen))
    print(vertically_percolates(isOpen))

# run_vertical_percolation_simulation()
```



## シミュレーション

### evaluate_vertical_percolation(n, p, trials)

この関数では、trial回シミュレーションを行い、垂直にパーコレートする確率を求めます。各trialでは、新しいisOpen行列をgenerate_percolatorでパラメータn, pから生成し、vertically_percolatesでシミュレーションします。そして、垂直にパーコレートした回数を数えます。

最後に、カウンタを試行回数で割った値を返します。

SEEDは必ずNoneに設定してください。

```
#%%
def evaluate_vertical_percolation(n, p, trials):
    """
    引数：行列の大きさ n: int, 開放率 p: float, 試行回数 trials: int
    返値：パーコレーション確立 float
    """
    pass

# evaluate_vertical_percolation(32, 0.9, 10000)  ## 0.66くらい
```

## Percolation

さて、先ほどは垂直な経路のみを考慮しましたが、全ての経路をたどる場合のシミュレーションを行う場合はどうすればよいでしょうか？

意外にも、この問題は「再帰的呼び出し」の「深さ優先探索」という方法であっさりと解くことができます。難しい字面ですが、一つ一つに着目すれば単純になります。

### 再帰関数の設計

まず、再帰的に呼び出される関数_flow(isOpen, isFull, i, j)を設計します。 この関数は、開放されているsiteに流入するときに呼び出され、さらに関数内から隣接するsiteに対して呼び出されます。

```
site[i][j]に対して呼び出し。
  ↓
isFull[i][j]をTrueに設定。
```

```
  ↓
site[i+1][j], site[i][j+1], site[i][j-1], site[i-1][j]に対して_flowを呼び出す。
```

ここで、isFull[i][j]を設定する前に、以下のことを確かめる必要があります。

- 0 <= i < n
- 0 <= j < n
- isOpen[i][j]がFalseでない

このいずれかに引っかかった場合、_flowは何も呼び出さずにreturnします。

さらに、呼び出されたマスがすでに評価されていて、fullの場合は、すでに隣接siteは評価済みであるから、

- isFull[i][j]がTrue

の場合も、_flowは何も呼び出さずにreturnします。

以上まとめると、

```
iとjの境界条件を満たさないならreturn。
isOpen[i][j]がOpenでないならreturn。
isFull[i][j]がFullならreturn。

isFull[i][j]をTrueに設定。

_flow(isOpen, isFull, i, j+1)を呼び出し。
同様に他3つの隣接siteに対しても呼び出し。
```

となる。

## 呼び出し関数の設計

次に、isFullを初期化し、_flowを呼び出す関数flowを設計します。

```
二次元ベクトルisFullをすべてFalseで初期化。
最上部のsiteすべてについて、_flowを呼び出す。
isFullをreturn。
```

## percolates(isOpen)

最後に、isOpenを受け取り、flowを実行したうえで、最下部の列を検査し、パーコレートしているかどうかのBoolを返す関数percolatesを書いてください。

**run_percolation_simulation** は、上記二つの関数をテストするために設けられた単発シミュレーションです。pythonファイル上部のSEEDを指定すると、毎回同じ系が生成されるようになります。

**再帰関数の経過観察** _flow関数で、isFullを設定した次の行に

```
 PercolationVisualizer(isOpen, isFull); input()
```

を挿入すると、_flow関数の再起呼び出しによりisFullが変更されていく様子をstep-by-stepで観察できるので、試してみましょう。

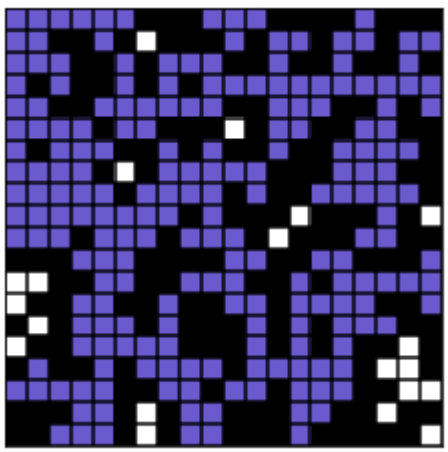
```
#%%
def _flow(isOpen, isFull, i, j):
    """
    引数：isOpen: list[list[bool]], i: int, j: int
    返値：None
    """
    pass

def flow(isOpen):
    """
    引数：isOpen: list[list[bool]]
    返値：list[list[bool]]
    """
    pass

def percolates(isOpen):
    """
    引数：isOpen: list[list[bool]]
    返値：bool
    """
    pass

def run_percolation_simulation():
    isOpen = generate_percolator(20, 0.6)
    PercolationVisualizer(isOpen, flow(isOpen))
    print(percolates(isOpen))

# run_percolation_simulation()
```



```
True
```

## シミュレーション

### evaluate_percolation(n, p, trials)

evaluate_vertical_percolationと同様に、このtrial回シミュレーションを行い、パーコレートする確率を求めます。各trialでは、新しいisOpen行列をgenerate_percolatorでパラメータn, pから生成し、percolatesでシミュレーションします。そして、パーコレートした回数を数えます。

最後に、カウンタを試行回数で割った値を返します。

SEEDは必ずNoneに設定してください。

```
#%%
def evaluate_percolation(n, p, trials):
    """
    引数：行列の大きさ n: int, 開放率 p: float, 試行回数 trials: int
    返値：パーコレーション確立 float
    """
    pass

# evaluate_percolation(32, 0.6, 1000)  ## 0.58くらい
```

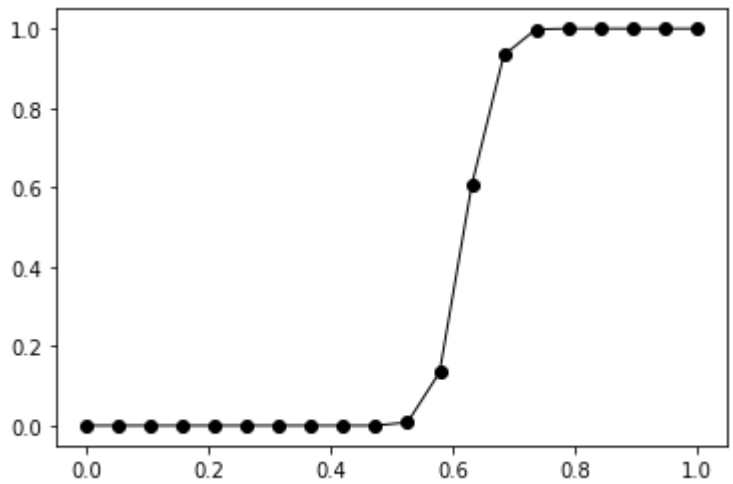
```
0.582
```

## 結果のプロット

x軸とy軸ののデータを用意して、linePlot関数に渡すと、線グラフが描写されます。

- x_data: 各データポイントの開放率pのリストです。0から0.05おきに1まで増加するように値を入れてください。例：[0, 0.05, 0.01, ..., 1]
- y_data: 各pに対して、evaluate_percolation(n, p, trial)を実行した結果を入れる。x_data と同じ要素数である。

nは32、trialは1000で評価したとき、次のような結果が得られるはずです。




この結果より、p = 0.6あたりで閾値が得られるでしょう。この閾値以下では、パーコレートしない場合が大半となり、閾値以上ではパーコレートする場合が大半となります。この現象は、phase transisiton（相転移）と呼ばれ、多くの物理システムで観察されます。

```
#%%
def linePlot(x, y):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    x = np.linspace(0,1,len(x))
    ax.plot(x, y, color='black',  linestyle='solid', linewidth = 1.0, marker='o')

def draw_plot():
    """
    引数：None
    返値：None
    """
    pass

# generate_plot()
```

## オプション課題

閾値はnの値に依存しないことがわかっています。しかし、nを大きくすると、乱数の影響が薄まりより正確な閾値がわかります。オプション課題として、より正確な閾値を求められるプログラムを書いた人は加点することにします。よい閾値を選ぶと、パーコレートする確率が0.5に漸近します。以下の方針が考えられますが、これらに限定するものではありません。

- 二分探索などをもちいて、pの範囲を狭める。初期の検索範囲は[0.58, 0.62]に限定してOK。
- 再帰部分を高速化させる（例えば、一度でもパーコレートすることが分かった場合、そのあとの計算をとばすことができる）

プログラムは別ファイル(exhard-percolation-option-xxxxxxxx.py)に記述し、以下のグローバル変数を記述してください。

- run_time: ITCのPCでの実行時間(秒)
- p_th: 求めた閾値（float型）

また、pythonファイル実行時に計算を実行しp_thと同じ値をprintするように設計してください。

加点は、p_thと正解値の差の桁数をもとに計算し、かつ実行時間の短い人トップｘ人を別に加点します。実行時間はこちらの環境で別に検証します。複数プロセスなど並列実行や外部ライブラリの使用、soファイルなどPython以外のファイルの提出は認めません。