第7回演習課題

提出時の注意

- student_id及びファイル名に学籍番号を設定すること。
- Pythonファイル中の型指定を守ること。
- 提出用のPythonファイルのグローバルスコープには、余分な関数呼び出しを書かないこと。デバッグ用に書いたものはすべてコメントアウトすること。Pythonファイルを実行した際に何も実行されないのが正しい状態です。

```
def func(a, b):
    ...
func(1, 2) <--これが呼び出しです。これをグローバルスコープに書かないでください。
```

- オプション演習に取り組まない場合でも、関数の定義を削除しないこと。

疎ベクトルドット積

ベクトルの大半がゼロの場合、通常のリスト形式で保存するのは非効率的です。そのため、非零要素とその添え字で表現する、疎ベクトル表現を定義します。例えば、`[0, 0, 1, 8, 0, 0, 0]`は、`[[2, 3], [1, 8]]`と表現します。

２つの疎ベクトルのドット積（内積）を求める関数を書いてください。

例：

```
dot([[1,2,3,4], [1,2,3,4]], [[3,4,5], [3,4,5]])
25
```

ヒント：

- リストのメソッドについてのドキュメントを参照すること。
- 添え字は0以上の整数とする。
- 入力のフォーマットは正しいものと想定してよい（添え字と非零要素の数が等しいなど）。

```
def dot(vec1, vec2):
    """
    引数：vec1: list[list[int], list[T]], vec2: list[list[int], list[T]]
          #ジェネリック型Tとは加算と乗算が定義済みの任意の型（int/floatなど）
    返値：T
```

```
    """
    pass
```

ヨセフスの問題

紀元370年ごろに書かれたヨセフスの問題のシミュレーションをします。問題は以下の通りです。

- 窮地に陥ったn人の住人が、人口を減らすために次のような作戦を実行します。
- 彼らは円形に自分たちを配置し、 執行人（住人には含まない）が住人1人が残るまでm人ごとに処刑しながら円を回っていきます。
- ヨセフスは、処刑されないようにどこに座ればいいかを考えます。

2つの整数の入力mとnを受け取り、人々が処刑される順序を表示し、円のどこに座るべきかヨセフスに示すプログラムを書きなさい。 なお、ヨセフスにわかりやすいように、最後に"<= sit here!"と表示すること。

テストケース

```
josephus(5, 9)
5 1 7 4 3 6 9 2 8 <= sit here!
# この場合、ヨセフスは8番目の席に座れば助かることになる
josephus(11, 8)
3 7 5 6 2 8 1 4 <= sit here!
```

ヒント：

- この問題には様々なアプローチがありますが、とりあえずリストに住人１～ｎをいれてしまうことから始めてみましょう。
- 処刑された住人はこのリストから削除することで再現できます。
- 執行人を変数として巡回させる方法もありますが、執行人を固定し（常にlist[0]を処刑する）住人をリスト内で巡回させることも可能です。

```
def josephus(m, n):
    """
    引数：m: int, n: int
    返値：None
    """
    pass
```

カッコの対応付け

この問題では、文字列中の対応するカッコ（`[`と`]`）のペアを調べ、辞書にして返す関数generate_bracket_mapを作ります。 例えば、`s = "x[x + [y]]"`という文字列があった場合、s[1]

とs[9]のカッコ、s[6]とs[8]のカッコが対応しているため、{1: 9, 6: 8, 8: 6, 9: 1}が出力されます。`[`と`]`の双方のキーが対応するペアのインデックスをバリューに持っていることに注意してください。

今回実装してもらうアルゴリズムには、「スタック」というデータ構造を用います。スタックは、Last In First Out (LIFO)とも呼ばれるデータ構造で、PushとPopの2つの操作を受け付け、Popは最後にPushされたデータを吐き出します。

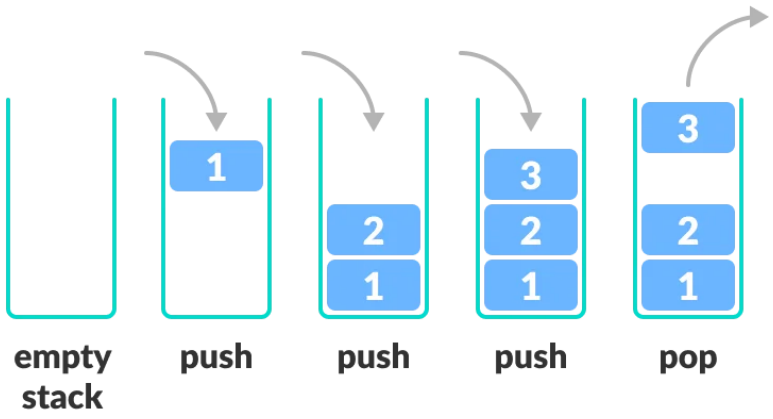


image credit: https://www.programiz.com/dsa/stack

今回は、Pythonのリストを用い、Pushをlist.append(xxx)、Popをlist.pop()で再現します。つまり、listは最初にPushされたもの（古いもの）から先頭にデータを保持し、Popでは最後のデータを返すとともにリストから削除します。

カッコの対応付けアルゴリズムは以下の通りです。

```
sのそれぞれの文字のインデックスiについてループ：
    もし、文字が[の場合、スタックにiをPushします。
    もし、文字が]の場合、
        まずスタックからPopします。これが対応する[のインデックスになります。
        bracket_map[i]にポップしたインデックスを設定します（] --> [）。
        同様に、bracket_mapに逆方向のペアを設定します（[ --> ]）。
```

プログラムを書く前に、一度図に起こしてアルゴリズムがどう動くか確認してみましょう！

テストケース

```
generate_bracket_map("x[x + [y]]")
{1: 9, 6: 8, 8: 6, 9: 1}
generate_bracket_map("a[b[[c]d[e]]f]")
{1: 13, 3: 11, 4: 6, 6: 4, 8: 10, 10: 8, 11: 3, 13: 1}
```

```
def generate_bracket_map(string):
    """
    引数：string: str
    返値：dict[int, int]
    """
    stack = []
    bracket_map = {}
    # 処理を以下に記入

    return bracket_map
```

オプション演習：BrainFuck

この演習では、皆さんにプログラミング言語とそれが動く仮想マシン（インタプリタ）を実装してもらいます。でも決して難しいわけではありません（Wordleよりかんたん！）

ここで実装するプログラミング言語は、BrainFuckと呼ばれる8文字しか使わない頭のおかしい（？）言語です。頭のおかしいとは言っても、実はBrainFuckはチューリング完全な言語です。

> **参考** チューリング完全な言語は、皆さんがお持ちのコンピュータ（チューリングマシン）で実行可能なすべての処理を記述できます。さらに、チューリング完全な言語は、（チューリングマシンと等価なため）チューリングマシン自体をシミュレート（再現）できるという特徴があります。PythonもBrainFuckもチューリング完全なので、PythonでBrainFuckが動く仮想マシンを作ることも、さらにはその逆を行うことも、理論的には可能なのです。

構成要素の準備

さて、BrainFuck仮想マシンはstr型のプログラムを受け取り実行します。仮想マシンに必要な要素はたった３つです。

- **メモリ**：データを置いておくところです。今回はリストを使います。メモリは以下のようにして初期化します。

  ```
  memory = [0] * 1024
  ```

- **ポインタ（idx）**：リストmemoryのインデックスを保存するCPU内の記憶領域です。int型の変数を使います。皆さんはプロ１で習うC言語のポインタでさんざん悩むと思いますが、「メモリというリスト」のインデックスだと思えば万事解決します。
- **プログラムカウンタ（pc）**：こちらは入力プログラム（文字列）のインデックスです。int型の変数を使います。

最初にこれらの要素を宣言し、初期化（idxとpcは0）してください。

処理系

次に、処理系を書きます。

```
pcがlen(prog)より小さい限りループ：
    progのインデックスpcに対応する文字(prog[pc])をcとする。
    cが > のとき、idxをインクリメント(+1)する。
    cが < のとき、idxをデクリメント(-1)する。
    cが + のとき、memory[idx]をインクリメントする。
    cが - のとき、memory[idx]をデクリメントする。
    cが . のとき、memory[idx]をプリントする。（注意１）
    cが , のとき、入力をmemory[idx]に代入する。（注意２）
    cが [ のとき、memory[idx]が0であれば、progの対応する]にジャンプする。（注意３）
    cが ] のとき、memory[idx]が0でなければ、progの対応する[にジャンプする。（注意３）
    cがそれ以外の時はコメントとして無視する。
    最後に、pcをインクリメントする。
```

- （注意１）メモリの内容をプリントするときは、シーザー暗号の時のようにメモリ内のユニコードを文字に変換する。また、print(xxx, end="")として、改行をなくす。
- （注意２）入力はinputでとり、シーザー暗号の時のように文字からユニコードに直してからメモリに格納する。
- （注意３）ジャンプはpcを書き換えることにより可能。対応するカッコのインデックスを探すのにgenerate_bracket_mapの結果を使ってよい（使用する場合、ループに入る前にbracket_mapを生成）。

たったこれだけで、仮想マシンが出来上がります！実はこの十数行のプログラムに、CPUの核である「命令フェッチ」「命令デコード」「演算実行」「メモリインタフェース」などがモデル化されています。

> **参考** ほとんど全てのCPUはこの5つのステージで構成されています。
>
> 1. 命令フェッチ（Instruction Fetch）：機械語のプログラムを読み込みます。prog[pc]の部分です。
> 2. 命令デコード（Instruction Decode）：機械語から、どの演算器を使用するか判断します。if文の部分です。
> 3. 演算実行（Execution）：加算器などでレジスタの値を変更します。今回はインクリメント／デクリメント操作です。
> 4. メモリインターフェイス：メモリにアクセスします。memory[idx]の部分です。

> 5. ライトバック：演算結果をレジスタに書き戻します。今回は、idxを更新する部分です。

早速、以下のプログラムを実行してみましょう。

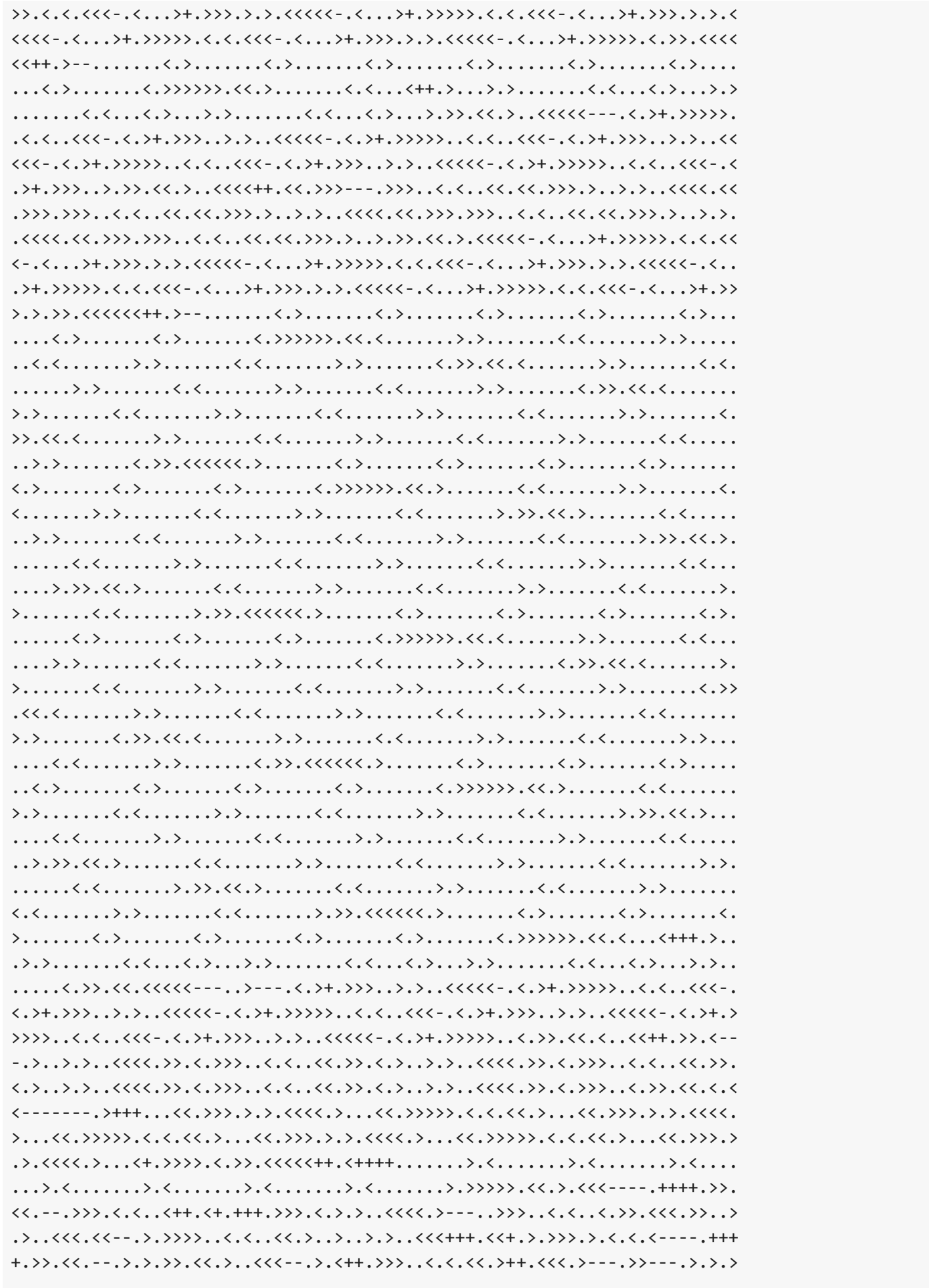
```
prog = """
+++++++++[>++++++++>++++++++++>+++>+<<<<-]>.>++.+++++++..+++.
>+++++.<<++++++++++++++++.>.+++.------.--------.>+.>+.
"""
exec_brainfuck(prog)
Hello World!
```

```
def exec_brainfuck(prog):
    """
    引数：prog: str
    返値：None
    """
    pass

prog = """
+++++++++[>++++++++>++++++++++>+++>+<<<<-]>.>++.+++++++..+++.
>+++++.<<++++++++++++++++.>.+++.------.--------.>+.>+.
"""
```

```
program1 = """
++++++++++[>++++++++++<-]>>++++++++++>->>>>>>>>>>-->+++++++[->++++++++
++<]>[->+>+>+>+<<<<]+++>>+++>>>++++++++[-<++++<++++<++++>>>]+++++[-<++
++<++++>>]>-->++++++[->++++++++++<]>[->+>+>+>+<<<<]+++++>>+>++++++>++
++++>++++++++[-<++++<++++<++++>>>]++++++[-<+++<+++<+++>>>]>-->+++-+[-<+
]-<<[->>>+>++[-->++]-->+++[---<-->+>-[<<++[>]]>++[--+[-<+]->>[-]+++++[
---->++++]-->[->+<]>>[.>]>++]-->+++]---+[-<+]->>-[+>++++++++++<<[->+>-
[>+>>]>[+[-<+>]>+>>]<<<<<<]>>[-]>>>++++++++++<[->->+<<]>[-]>[<++++++[-
>++++++++<]>.[-]]<<++++++[-<++++++++>]<.[-]<<[-<+>]]<<<.<]"""

program2 = """
++++++++++++++[>++>+++>+++>++++++>++>+++++++>++++>+<<<<<<<<-]>>--.>....
...<.>.......<.>.......<.>.......<.>.......<.>.......<.>.......<.>......
.<.>>>>>>-----.<<+++++++++++++++++.<++.<+.++++.>>.<<.--.>.>.>--..<<<++
.<-.+++.>>>.>.<.<..<<.>---..>..>.>..<<<.>>.<<<.>>>>..<.<..<.<<.>.>>..>.>
..<<<<.>..>>>..<.<..<+++.<<+.>.>>>.<.>.>.<<<----.++++.>>.<<.--.>>>.<.>>.
<<.<..<--.<<<+++++.>>>++.>..>.>.<<<<.>++.<<<.>---.>>---.>>>.<.<..<.<<<.>
>.>>..>.>..<<<<<-.<.>+.>>>>>..<.<..>-.<+++.>++.<---..>-.>..<<<.<<<.>>.>>
>>..<.<.<<.>+++.<<<.>.>>---.>.>.>..<<<-.<<<.>>>++.>>>..<.>>.<<.<..<--.<<
<.>>>++.>..>.>..<<<<.<<.>>>-..>>>.<.<..<<<-.<.>+.>>>..>.>..<<<<<-.<.>+.>
>>>>..<.<..>-.<+++.>++.<---..>-.>..<<<<<-.<.>+.>>>>>..<.<..<<.<<.>>>..>.
>.>..<<<-.<<<.>>>++.>>>..<.>>.<<.<.<<<-.<...>+.>>>.>.>.<<<<<-.<...>+.>>>
```



```
..<<<.>.<<.>>>>..<.<..<<<-.<.>+.>>>..>.>..<-.<.>++.>..<-.<..<.>.<<.>>..>
.>.<<<<.>+++.<<<.>.>>---.>>>.<.<..<-.>.<++.>..>.>>.<<.>..<<<--.>.<++.>>>
..<.<..<<.>>.<-..>.>.>..<<<<<-.<.>+.>>>>>..<.<..<<<-.<.>+.>>>..>.>..<-.<
.>++.>..<-.<..<<<-.<.>+.>>>..>.>..<<<<.>>.<..>>>.<.<..<-.>.<++.>..>.>>.<
<.>.<<<<-------.>++...<<.>>>>>.<.<.<<.>...<<.>>>.>.>.<<<<.>...<<.>>>>>.<
.<.<<.>...<<.>>>.>.>.<<<<.>...<<.>>>>>.<.<.<<.>...<<.>>>.>.>.<<<<.>...<<
.>>>>>.<.<.<<.>...<<.>>>.>.>>.<<<<<<++.++.......>+++.<.......>.<.......>
.<.......>.<.......>.<.......>.<....>>----.<--.<<.>++.>>+.<<.>>.>.<<-.<-
-..--.>>>>>>."""

#program3は途中でエラーになります。なぜでしょう。
program3 = """
>>>>++>+>++>+>>++<+[
  [>[>>[>>>>]<<<<[[>>>>+<<<<-]<<<<]>>>>>>]+<]>-
  >>--[+[+++<<<<--]++>>>>--]+[>>>>]<<<<[<<+<+<]<<[
    >>>>>>[[<<<<+>>>>-]>>>>]<<<<<<<<[<<<<]
    >>-[<<+>>-]+<<[->>>>[-[+>>>>-]-<<-[>>>>-]++>>+[-<<<<+]+>>>>]<<<<[<<<<]]
    >[-[<+>-]]+<[->>>>[-[+>>>>-]-<<<-[>>>>-]++>>>+[-<<<<+]+>>>>]<<<<[<<<<]]<<
  ]>>>+[>>>>]-[+<<<<--]++[<<<<]>>>+[
    >-[
      >>[--[++>>+>>--]-<[-[-[+++<<<<-]+>>>>-]]++>+[-<<<<+]++>>+>>]
      <<[>[<-<<<]+<]>->>>
    ]+>[>>>>]-[+<<<<--]++<[
      [>>>>]<<<<[
        -[->--[<->+]++<[[>-<+]++[<<<<]+>>+>>-]++<<<<-]
        >-[+[<+[<<<<]>]<+>]+<[->->>>[-]]+<<<<
      ]
    ]>[<<<<]>[
      -[
        -[
          +++++[>+++++++++<-]>-.>>>-[<<<----.<]<[<<]>>[-]>->>+[
            [>>>>]+[-[->>>>+>>>>>>>>-[-[+++<<<<[-]]+>>>>-]++[<<<<]]+<<<<]>>>
          ]+<+<<
        ]>[
          -[
            ->[--[++>>>>>--]->[-[-[+++<<<<-]+>>>>-]]++<+[-<<<<+]++>>>>]
            <<<<[>[<<<<]+<]>->>
          ]<
        ]>>>>[--[++>>>>>--]-<--[+++>>>>--]+>+[-<<<<+]++>>>>]<<<<<[<<<<]<
      ]>[>+<<++<]<
    ]>[+>[--[++>>>>--]->--[+++>>>>--]+<+[-<<<<+]++>>>>]<<<[<<<<]]>>
  ]>
]
"""

program4 = """
[ This program prints Sierpinski triangle on 80-column display. ]
                                 >
                                + +
                               +   +
                              [ < + +
```

```
                          +       +
                         + +     + +
                        >   -   ]   >
                       + + + + + + + +
                      [               >
                     + +             + +
                    <   -           ]   >
                   > + + >         > > + >
                  >       >       +       <
                 < <     < <     < <     < <
                <   [   -   [   -   >   +   <
               ] > [ - < + > > > . < < ] > > >
              [                               [
             - >                             + +
            +   +                           +   +
           + + [ >                         + + + +
          <       -                       ]       >
         . <     < [                     - >     + <
        ]   +   >   [                   -   >   +   +
       + + + + + + + +                 < < + > ] > . [
      -               ]               >               ]
     ] +             < <             < [             - [
    -   >           +   <           ]   +           >   [
   - < + >         > > - [         - > + <         ] + + >
  [       -       <       -       >       ]       <       <
 < ]     < <     < <     ] +     + +     + +     + +     + +
 +   .   +   +   +   .   [   -   ]   <   ]   +   +   +   +   +
* * * * * M a d e * B y : * N Y Y R I K K I * 2 0 0 2 * * * * *
"""
```

時間つぶし用の課題：やることリスト

タスクの数とその依存関係のリストを入力にとり、そのすべてのタスクが実行可能か判定する関数を書いてください。

例えば、タスク1がタスク0より前に実行されなければならない場合、その依存関係は[0, 1]と表されます。

実行例

```
can_finish(2, [[1, 0]])
True
can_finish(2, [[1, 0], [0, 1]])
False
can_finish(4, [[1, 0], [2, 1], [3, 2]])
True  # 0->1->2->3 で可能
test_code(can_finish)
# テストケースに全問正解の場合メッセージが表示
```

```
def can_finish(num_tasks, prerequisites):
    pass

'''
検証用
'''
def gen_random_tasks(n, seed):
    import random
    random.seed(seed)
    prerequisites = set()
    while len(prerequisites) < n // 4 * 3:
        v1 = random.randrange(n)
        v2 = random.randrange(n)
        while v1 == v2:
            v2 = random.randrange(n)
        prerequisites.add((v1, v2))
    return list(prerequisites)

def test_code(fn):
    test_seeds = [3*n + 1 for n in range(20)]
    n = 1000
    results = []
    for test_seed in test_seeds:
        prerequisites = gen_random_tasks(n, test_seed)
        results.append(fn(n, prerequisites))
    x = "".join(map(str, results))

    import hashlib
    cypher = 214676177229716215358805913591335380330
    key = hashlib.md5(bytes(x, 'utf-8')).hexdigest()
    decrypted_hex = cypher ^ int(key, 16)
    try:
        return bytes.fromhex(hex(decrypted_hex)[2:]).decode()
    except:
        return "Decoding error"
```