アルゴリズム入門　共通問題（2023年度Aセメスター試験）

試験日時：2024年1月31日第5限（60分）　答案用紙：1部　持ち込み一切不可

裏面にも問題があるので確認すること

内容に関する質問は受け付けない．問題の記述が曖昧な場合は，適切な仮定をおいて回答し，どのような仮定をおいたのか明記せよ．`make1d(n,v)` はすべての要素が `v` で長さが `n` の配列を作る関数，`make2d(n,m)` はすべての要素が `0` で `n` 行 `m` 列の2次元配列を作る関数であり，これらは定義済みだとしてよい．

問題1

(1) 右記の関数 `is_seven(n)` は整数 `n` がちょうど 7 であるかどうかを `True` または `False` で返す．つぎの (ア) ～ (オ) の中で，評価した結果が `True` となるものをすべて選べ．

```
def is_seven(n):
  return n == 7
```

(ア) `is_seven(6) and True`
(イ) `is_seven(7) or False`
(ウ) `is_seven(8) or True`
(エ) `is_seven(14) or is_seven(7)`
(オ) `is_seven(77) and is_seven(7)`

(2) 非負整数 `n` が偶数かどうか (2で割り切れるかどうか) を知りたい．非負整数 `n` を正の整数 `m` で割った余りは `n % m` により求められる．右記の関数 `is_even_number(n)` の空欄を埋めよ．

```
def is_even_number(n):
  return [ A ]
```

(3) 2以上の整数 `n` について，その1でない最小の約数を求める右記の関数 `min_factor(n)` の空欄を埋めよ．なお，素数の場合は `n` を返すものとする．

```
def min_factor(n):
  for d in range(2, n):
    if [ B ]:
      return d
  return n
```

(4) 右記の関数 `mark_even` についての，以下の説明中の空欄を埋めよ．なお，この問や以降の小問では，正の整数 `N` と `len(A)` = `N` であるような配列 `A` を考える．配列 `A` の要素は真偽値 (`True` または `False`) であると想定する．

`mark_even(A)` を実行したとき，`is_even_number` が呼び出される回数は [C] 回である．`mark_even` の実行前に `A` の要素がすべて `False` だったとして，実行終了後に値が変化する要素は [D] 個である．

```
def mark_even(A):
  for k in range(0, len(A)):
    A[k] = is_even_number(k)
```

(5) 右記の関数 `mark_even2` についての，以下の説明中の空欄を埋めよ．

`mark_even2(A)` を実行したとき，`A[k] = True` の文を実行する回数は [E] 回である．`mark_even2` の実行前に `A` の要素がすべて `False` だったとして，実行終了後に値が変化する要素は [F] 個である．

```
def mark_even2(A):
  k = 0
  while k < len(A):
    A[k] = True
    k = k + 2
```

(6) 合成数 (素数でない数) を True とする方針を考える．具体的には，配列 A の要素は初めすべて False として，計算終了後には $2 \leq$ n $<$ N となる整数 n に対して， n が素数なら A[n] は False，合成数なら True としたい．なお，計算終了後の A[0] や A[1] の値は何であっても差し支えない．

(a) 右記の mark_multiples(A, p) の動作を，(5) の mark_even2(A) と比較しながら 1 ～ 2 行で説明せよ．

(b) 右記の mark_composites(10) を計算したときの， A の結果と，mark_multiples 内の A[k] = True の文が合計何回実行されるかを，簡潔な理由とともに答えよ．

(c) mark_composites では，実行後に A[j] が True であれば A[j*2] も True である (j は $2 \leq$ j $<$ N/2 の整数)．この理由を mark_multiples のプログラムと対応させつつ 2 ～ 3 行で説明せよ．

```
def mark_multiples(A, p):
  k = p * 2
  while k < len(A):
    A[k] = True
    k = k + p

def mark_composites(N):
  A = make1d(N, False)
  for p in range(2, len(A)):
    mark_multiples(A, p)
  return A
```

(7) mark_composites 内を変更して mark_multiples の実行回数を減らすことで，効率を改善しつつ，まったく同じ計算結果を得たい．以下の観点で可能な工夫について， プログラムに対する変更を具体的に述べ，さらに変更前と同じ結果を得られる理由を 2 ～ 3 行で説明せよ．

(a) for ループ内に if 条件: を挿入して mark_multiples(A, p) を一部の p にのみ実行する．

(b) for ループの range(2, len(A)) の部分を工夫して，検討する p の範囲を狭くする．

問題 2

チェスで用いるクイーンの駒は，縦・横・斜めの全ての方向の任意のマスに移動できる．クイーンを別の駒のあるマスに移動させることで，その駒を取ることができるとする．$n \times n$ のチェス盤において，n 個のクイーンを，どのクイーンも他のクイーンに取られることがないように安全に配置する（図 1 参照）問題を考える．

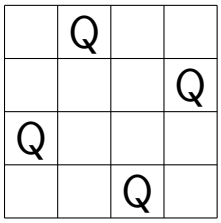

図 1: クイーンの安全な配置の一例．Q はクイーンを表す．

0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

図 2: 図 1 に対応する 2 次元配列 board の例．

```
def is_safe(board, i, j):
  n = len(board)
  # 同じ行と列にクイーンがあるか確認する
  for k in range(0, n):
    if board[ A ][ B ] == 1:
       return False
    if board[ C ][ D ] == 1:
       return False
  # 左上，右下にクイーンがあるか確認する
  for k in range(0, n):
    if (0 <= i - k < n and 0 <= j - k < n and
        board[ E ][ F ] == 1):
      return False
    if (0 <= i + k < n and 0 <= j + k < n and
        board[ G ][ H ] == 1):
      return False
  # 左下，右上にクイーンがあるか確認する
  # （省略）
  return True
```

(1) 上記の is_safe 関数は，チェス盤上の特定の位置にクイーンを置いても安全か確認する．2 次元配列 board

は，i 行 j 列にクイーンがある場合は board[i][j] が 1，ない場合は 0 である（図 2 参照）．新たに i 行 j 列にクイーンを置いても，他のクイーンに取られることがない場合は，is_safe は True を返す．それ以外は False を返す．空欄を埋めて関数 is_safe を完成させよ．

(2) 下記の solve_nq_naive 関数は，各行と各列にクイーンは一つしか配置できないことを踏まえて，全ての候補を探索し，解が見つかった時点でそれを返す．ここで，get_permutations(n) は 0 から n-1 までの整数の順列を返し，例えば get_permutations(4) は [[0,1,2,3],[0,1,3,2],...,[3,2,1,0]] を返す．(*) の行において，queens が [0,2,3,1] のとき，solve_nq_nsub(board, queens) の呼び出し終了時に返される値と，その時の board の値を答えよ．board の値は図 1 のように図示しても良い．

```
def solve_nq_naive(n):
  candidates = get_permutations(n)
  for k in range(0, len(candidates)):
    queens = candidates[k]
    board =  make2d(n, n)
    if solve_nq_nsub(board, queens): # (*)
      return board
```

```
def solve_nq_nsub(board, queens):
  for j in range(0, len(queens)):
    i = queens[j]
    if is_safe(board, i, j):
      board[i][j] = 1
      # i 行 j 列にクイーンを配置する
    else:
      return False
  return True
```

(3) 下記の solve_nq 関数は，solve_nq_sub 関数を再帰的に実行することで，solve_nq_naive 関数よりも効率的に解を探索する．solve_nq_sub(board,j) は，盤面 board の j 列目以降にクイーンを配置し，解が見つかったら True を，見つからなかったら False を返す．図 3 に，solve_nq(4) における，solve_nq_sub の最初の 2 回の実行を示す．1 回目は，空の盤面に対して，solve_nq_sub(board,0) を実行する．0 行 0 列は安全なのでクイーンを配置して，2 回目は solve_nq_sub(board,1) を実行する．solve_nq_sub を 3 回目と 4 回目に呼び出す際の引数（board と j) の値をそれぞれ示せ．board の値は図 1 のように図示しても良い．

(4) solve_nq_naive 関数と比較して，solve_nq 関数がどのように効率的であるかを述べよ．

```
def solve_nq(n):
  board =  make2d(n, n)
  if solve_nq_sub(board, 0):
    return board

def solve_nq_sub(board, j):
  # 全ての列にクイーンを配置できたら True
  n = len(board)
  if j >= n:
    return True
  for i in range(0, n):
    if is_safe(board, i, j):
      # i 行 j 列にクイーンを配置する
      board[i][j] = 1
      # j+1 列目以降を探索
      if solve_nq_sub(board, j+1):
        return True
      else:
        # i 行 j 列へのクイーンの配置を止める
        board[i][j] = 0
  return False
```

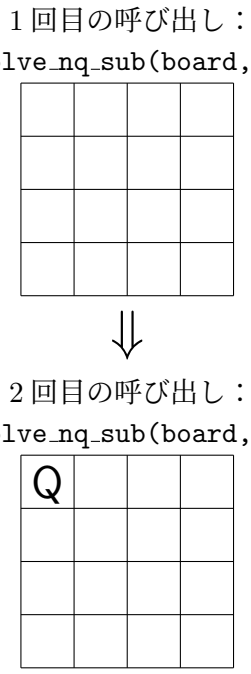

図 3: solve_nq 関数における，solve_nq_sub 関数の実行順

問題 3

(1) x の n 乗（n は非負整数）を求める右記の関数 pow の空欄を埋めよ．ただし，pow や**を用いてはならない．

(2) 関数 pow の n に関する時間計算量オーダを示せ．

(3) 多項式の計算を行う右記 poly1・poly2 の空欄を埋めよ．入力 keisu は数値の配列であり，先頭に近い要素ほど高い次数の係数を，末尾要素は定数項（0 次の係数）を表す．例えば，[1,2,3] は多項式 x^2+2x+3 を，[-2,0,0,5] は多項式 $-2x^3+5$ を表す．また，入力 x は多項式の変数 x の値である．例えば，poly1([1,2,3,4],5) であれば $1\cdot5^3+2\cdot5^2+3\cdot5^1+4$ を計算する．関数 poly2 は poly1 とはアルゴリズムが異なる．例えば，poly2([1,2,3,4],5) であれば，多項式を $((1x+2)x+3)x+4$ と変形し，$((1\cdot5+2)\cdot5+3)\cdot5+4$ を計算する．

```
def pow(x, n):
  if n == 0:
    return 1.0
  r = pow(x, n // 2)
  if n % 2 == 0:
    return [ A ]
  else:
    return [ B ]

def poly1(keisu, x):
  r = 0
  for i in range(0, len(keisu)):
    r = r + keisu[i] * pow(x, [ C ])
  return r

def poly2(keisu, x):
  r = keisu[0]
  for i in range(1, len(keisu)):
    r = [ D ]
  return r
```

(4) keisu の長さ m に関する時間計算量オーダを関数 poly1 と関数 poly2 のそれぞれについて示せ．

(5) 多項式 $f_1(x) = 2.0^{128}x^{128} + \sum_{i=0}^{127} 2.0^{i-53}x^i$ に対し，$f_1(0.5) = 1 + 128\cdot2^{-53} \fallingdotseq 1.0000000000000142$ を計算することを考える．

(a) 関数 poly1 を用いて計算したところ，1.0 という結果が得られた．誤差の主な理由を説明せよ．桁落ち，情報落ち，丸め誤差との関連があるならその点に言及すること．

(b) 関数 poly2 を用いて計算した結果に含まれる誤差は，関数 poly1 で計算した場合と比べてどうなると予想されるか．誤差がかなり少なくなる・同程度の誤差を含む・誤差がかなり大きくなるのいずれかを選んだ上で，その理由を説明せよ．

(6) 多項式 $f_2(x) = x^{11} - 1.9999999x^{10}$ に対し，$f_2(2.0) = 0.0000001 \times 2^{10} = 0.0001024$ を計算することを考える．

(a) 関数 poly1 を用いて計算したところ，0.000102400000059788 という結果が得られた．誤差の主な理由を説明せよ．桁落ち，情報落ち，丸め誤差との関連があるならその点に言及すること．

(b) 関数 poly2 を用いて計算した結果に含まれる誤差は，関数 poly1 で計算した場合と比べてどうなると予想されるか．誤差がかなり少なくなる・同程度の誤差を含む・誤差がかなり大きくなるのいずれかを選んだ上で，その理由を説明せよ．