＝＝＝＝＝＝＝＝＝第 1 章の解説＝＝＝＝＝＝＝

■問 1

ハノイの塔のパズルを解く上で必要となるアルゴリズムの理解を問う問題です。ハノイの塔のルールを知っていれば難しくはありません。

■問 2

問 1 のアルゴリズムをそのままプログラムとして書き下すだけです。

■問 3

ハノイの塔を解く場合の円盤の移動回数を問う問題です。

```
hanoi(n-1, source, auxiliary, target)
print(f"Move disk {n} from {source} to {target}")
hanoi(n-1, auxiliary, target, source)
```

というプログラムの核となる部分を知っておけば、（n－1）枚の円盤のハノイの塔の解法が 2 回あって、その間に「移動（Move)」の手順が 1 回あることから正解を導き出せます。

■問 4

いわゆる「不自由なハノイの塔」の問題です。これは、数学セミナー誌の 1984 年 12 月号で取り上げられています。プログラムで特にはグラフ理論を使うので超難問です。問 4 自体は「disk 2」が"B"に置けないことを理解していれば瞬殺で答えが分かります。

＝＝＝＝＝＝＝＝＝第 2 章の解説＝＝＝＝＝＝＝＝

■問 1

フィボナッチ数列を、その定義にしたがって求めて行けば、次の解答が得られます。

1 個目：fibonacci(0)=0
2 個目：fibonacci(1)=1
3 個目：fibonacci(2)=fibonacci(1)+fibonacci(0)=1+0=1
4 個目：fibonacci(3)=fibonacci(2)+fibonacci(1)=1+1=2
5 個目：fibonacci(4)=fibonacci(3)+fibonacci(2)=2+1=3
6 個目：fibonacci(5)=fibonacci(4)+fibonacci(3)=3+2=5
7 個目：fibonacci(6)=fibonacci(5)+fibonacci(4)=5+3=8
8 個目：fibonacci(7)=fibonacci(6)+fibonacci(5)=8+5=13
9 個目：fibonacci(8)=fibonacci(7)+fibonacci(6)=13+8=21
10 個目：fibonacci(9)=fibonacci(8)+fibonacci(7)=21+13=34
11 個目：fibonacci(10)=fibonacci(9)+fibonacci(8)=34+21=55

■問 2

基本的な再帰呼び出しの関数定義を答えさせる問題です。肝は、n＝0 の場合と n＝1 の場合を特別扱いして、n が 0 や 1 の場合はそれ以上の再帰呼び出しが発生しないようにすることです。

■問 3

これは Python だけの話かもしれませんが、メモ化をして、再帰呼び出しを高速化する方法を試す問題です。sum_fibonacci 関数の方は単純な再帰呼び出しで記述して、fibonacci 関数をメモ化で高速化します。これは、一種のキャッシュで、引数 n がメモ化した辞書である fibonacci_cache というリストに入っていれば、そこから結果を取り出します。これによって、関数の値を高速に計算できるようになります。関数の説明は次のようになります。

・fibonacci（n）：与えられた項数 n のフィボナッチ数を計算します。関数内で再帰呼び出しを使用しており、以下の手順で計算を行います。

・n が 0 以下の場合、0 を返します。

・n が 1 の場合、1 を返します。

・n が fibonacci_cache 辞書のキーとして存在する場合、キャッシュされた計算結果を取得して返します。

・上記の条件に該当しない場合、n－1 番目のフィボナッチ数と n－2 番目のフィボナッチ数の和を計算し、その結果を返します。また、計算結果を fibonacci_cache 辞書にキャッシュします。

・sum_fibonacci（n）：指定された項数 n までのフィボナッチ数列の合計を計算します。関数内で再帰呼び出しを使用しており、以下の手順で計算を行います。

・n が 0 以下の場合、0 を返します。

・n が 1 の場合、1 を返します。

・上記の条件に該当しない場合、n－1 番目までのフィボナッチ数列の合計と n 番目のフィボナッチ数を再帰的に計算し、その結果の和を返します。

sum_fibonacci 関数自体もメモ化可能です。余力のある人は挑戦してみてください。

■問 4

一見、再帰呼び出しができそうにない関数も再帰呼び出しが可能であるという例です。以下が関数の動作の概要です。

・n が負の場合、フィボナッチ数列に含まれないため False を返します。

・n が a と等しい場合、n はフィボナッチ数列に含まれるため True を返します。

・n が a より小さい場合、n はフィボナッチ数列に含まれないため False を返します。

つまり、フィボナッチ数列の数値（a）が n より大きくなると、それ以降のフィボナッチ

数列の値は必ず a より大きくなりますから、それ以降のフィボナッチ数列の値として n が出現することはありません。
・上記のいずれの条件にも該当しない場合は、再帰的に is_in_fibonacci 関数を呼び出し、新たな引数として b を a に、a＋b を b に渡します。
　この再帰的な呼び出しにより、与えられた数値 n がフィボナッチ数列に含まれるかどうかを判定することができます。
＝＝＝＝＝＝＝＝＝第 3 章の解説＝＝＝＝＝＝＝＝＝
■問 1
　これはデータのスワップを行う表現を問う問題です。基本的に、Python での表現ですが、Python を知らなくても選択肢を見れば正答が分かると思います。

■問 2
　これは、if 文が 1 回につき比較が 1 回だと分かれば難しい問題ではありません。

■問 3
　クイックソートのアルゴリズムで再帰呼び出しの箇所がどうなるかを質問しています。基本的に、左（left)、中央（middle)、右（right）の順番でリスト（配列）が結合されることを知っていれば難しくありません。

■問 4
　クイックソートの比較回数を質問しています。再帰呼び出しの前に、ピボット（pivot）より大きいかどうかの比較を行っているのでリストの要素の個数分に比較が行われます.「-1」となっているのは要素は 1 個の場合は比較が行われないためです。
＝＝＝＝＝＝＝＝＝第 4 章の解説＝＝＝＝＝＝＝＝＝
■問 1
　ユークリッドの互除法の説明を行っています。説明文をそのままプログラムに表しただけなので、ユークリッドの互除法を知らない人でも解答にたどり着けると思います。

■問 2
　最小公倍数の性質から答えは容易に分かります。サービス問題です。

■問 3
　頭を柔軟にしなと説明文の理解が難しいと思います。整数 a と b（a≧b）において、a を b で割った余りがと b が、ユークリッドの互除法の次のステップになることを理解している必要があります。

■問 4

このプログラムでは、extended_gcd 関数でユークリッドの互除法を実装し、find_solution 関数で mx＋ny＝gcd（m，n）の解となる x、y の組を見つけます。テストでは、m＝24、n＝18 の場合の解を求めています。

特に整数 a と b が互いに素である場合、その最大公約数は 1 になります。ax＋by＝1 となる整数 x と y を見つけることは ax＝1（mode b）となる x を見つけることに帰着できます。

＝＝＝＝＝＝＝＝＝第 5 章の解説＝＝＝＝＝＝＝＝＝

■問 1

関数 is_prime（n）は与えられた整数 n が素数かどうかを判定します。以下のアルゴリズムに従って判定しています。

・n が 1 以下の場合は素数ではないとして False を返す

・n が 2 または 3 の場合は素数として True を返す

・n が偶数または 3 の倍数である場合は素数ではないとして False を返す

・n が 5 以上の奇数の場合、6 の倍数を除く全ての素数は 6k±1 の形になることを利用して、6k±1 の形に合致するかを判定する

なお、6 の倍数を除く全ての素数は 6k±1 の形になることは次のように説明できます。

まず、整数 k に対して、6k、6k＋2、6k＋3、6k＋4 はすべて 6 の倍数です。つまり、6 の倍数は 2 と 3 で割り切れるため、素数ではありません。そこで、素数を見つける際には 6k±1 の形に注目します。

2 と 3 を除くすべての自然数 n は、

6k+0、6k+1、6k+2、6k+3、6k+4、6k+5

とあらわせます。これらについて。素数かどうかを検証します。

・6k＋0＝6k（6 の倍数）：6 の倍数は素数ではないため除外されます

・6k＋1＝6k＋1（素数）：例えば k＝0 のとき、6k＋1＝1 は素数です

※1 は素数ではありませんが、ここでは「素数的」に扱っています

・6k＋2＝2(3k＋1)（2 の倍数）：2 の倍数は素数ではないため、除外されます

・6k＋3＝3(2k＋1)（3 の倍数）：3 の倍数は素数ではないため、除外されます

・6k＋4＝2(3k＋2)（2 の倍数）：2 の倍数は素数ではないため、除外されます

・6k＋5＝6k＋5（素数）：例えば、k＝0 のとき、6k＋5＝5 は素数です

このように、6 の倍数を除く全ての素数は（6k±1）または（6k＋1、6k＋5）の形になることが分かります。この特性を利用して、6k±1 の形の数だけをチェックすれば、効率的に素数を見つけることができます。

例えば、エラトステネスのふるいでは、6k±1 の形の数だけを順番に見つけていくことで素数をふるいにかけることができます。

プログラムの中に

i＝5

```
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
```

とう部分がありますが、ここで、整数が 6k±1 での形をしているかどうかをチェックしています。チェックする整数は 5 と 5+2（これらは 6×1±1 です）から始まって、6 ずつ値を増やしてチェックしていきます。

また、i * i <= n の条件でループを終了させているのは、n が素数でない場合には、その約数が n の平方根以下に存在するためです。つまり、n が素数でない場合には、n の平方根以下の整数まで調べれば十分であり、それ以上の数を調べる必要がありません。

■問 2

これは、問 1 で作成した is_prime（n）関数を使って、総当たりで与えられた範囲の整数が素数かどうかをチェックし、その数が素数であれば、リスト prime に格納していきます。すべての候補をチェックし終わった後に、リスト prime を関数の戻り値としてます。

■問 3

素数を求めるアルゴリズムとして有名なエラトステネスのふるいをプログラムで実装します。問題にしている変数 j のループの開始は i*i でなくても i+i でも構いません。

つまり、

```
    for i in range(2, int(n**0.5) + 1):
        if sieve[i]:
            for j in range(i * i, n + 1, i):
                sieve[j] = False
```

でも

```
    for i in range(2, int(n**0.5) + 1):
        if sieve[i]:
            for j in range(i + i, n + 1, i):
                sieve[j] = False
```

でもよいこと n なります。for j in range(i * i, n + 1, i)としている理由は、エラトステネスの篩の効率性を向上させるためです。for j in range(i + i, n + 1, i)とすると、i の倍数を範囲内から取り除くときに i より小さい倍数（例えば、2 の倍数であれば 4、 6、8 など）は既に他の素数によって取り除かれている可能性があります。つまり、重複して取り除くことになります。

一方で、for j in range(i * i, n + 1, i)とすると、i より小さい倍数は既に他の素数によって取り除かれているため、重複を避けることができます。また、i の倍数のうち i より小さい

倍数は i * i 未満であるため、それらはすでに他の素数によって取り除かれていることになります。

したがって、for j in range(i * i, n + 1, i)とすることで、重複を避けながら効率的に非素数を取り除くことができ、エラトステネスのふるいの効率性が向上します。

■問 4

この問題は問 3 で作成した sieve_of_eratosthenes（n）関数を使って、与えられた偶数値を上限とするの素数のリストを作っておき、その素数リストのすべての要素を引き算してみて、それが素数であるかをチェックしています。いわば、総当たりです。

ゴールドバッハ予想は、多くの場合で確認されていますが、未だに正確な証明がなされていません。この予想は、数論の重要な未解決問題の一つとして知られており、多くの数学者が長年にわたって取り組んできました。これが証明されると、素数の性質に関する新たな洞察が得られる可能性があります。しかし、現在のところゴールドバッハ予想は未解決のままです。

＝＝＝＝＝＝＝＝＝第 6 章の解説＝＝＝＝＝＝＝＝

■問 1

平方根、三角関数、自然対数の底のべき乗など、超越関数の値を求める場合にニュートン・ラフソン法は多用されます。場合によっては 2 分探索法も使われますが、ニュートン・ラフソン法の方が有名です。平方根の場合も

f（x）＝x^2－2

という関数の解を求めることに帰着すればニュートン・ラフソン法で近似値を求めることができることを示す問題です。

■問 2

問 1 で示したアルゴリズムをプログラミングしたものです。問 1 が解っていれば難しくはありません。

■問 3

平方根を求める有名な方法である開平法（開平計算ともいう）を解説しています。これは計算問題の文章題です。地道な計算が求められます。

■問 4

問 3 の開平法のアルゴリズムをプログラミングしたものです。問 3 では平方根の最高位の数値は特別扱いされているように思えますが、ブロック内で引き算する数が

（（2× α）×10＋ β）× β

ここで、α が途中までの平方根、β が新しい平方の最下位を求めることであると一般化で

きれば解くことができます。開平法のアルゴリズムをよく理解していないと解くのは難しいかもしれません。

```
square_root_approximation
print(square_root_approximation(123))
print(square_root_approximation(3456))
print(square_root_approximation(56789))
print(square_root_approximation(200000000))
```

＝＝＝＝＝＝＝＝＝第 7 章の解説＝＝＝＝＝＝＝＝＝

■問 1

ベクトル（a0，a1，a2，…，an）と（b0，b1，b2，…，bn)の内積は

a0×b0＋a1×b1×a2×b2＋…＋an×bn

で表されます。この問題は sum 関数で総和を計算していますから、その各要素は zip 関数で対応させた要素の積となります。

■問 2

ベクトルの外積は、3 次元ベクトルに対して定義される演算であり、2 つのベクトルに垂直なベクトルを求める操作です。外積はクロス積とも呼ばれます。

逆に言うと、外積を計算すると 2 つのベクトルに垂直な 3 次元ベクトルが得られます。外積は主に物理学や 3D グラフィックスなどの応用分野で使われます。また、外積を用いて平面の法線ベクトルや面積を求めることができます。

2 つの 3 次元ベクトル A＝（a1，a2，a3）と B＝（b1，b2，b3）の外積 C＝A×B は以下のように定義されます。

（a2 ・ b3 — a3 ・ b2，　a3 ・ b1 — a1 ・ b3，　a1 ・ b2 — a2 ・ b1）

この式を知っているかどうかがキーポイントです。この式を知らなくても、外積の定義である 2 つのベクトルに垂直という性質を知っていれば次の連立方程式から外積（x，y，　z）を求めることが可能です.

a1・x ＋ a2・y ＋ a3・z ＝ 0 …（1） # 内積が 0

b1・x ＋ b2・y ＋ b3・z ＝ 0 …（2） # 内積が 0

これを解いてみましょう。まず、(1）を変形して y を求めます。

y ＝ —（a1/a2)・x — （a3/a2)・z …（3）

次に式（2）に式（3）を代入します。

b1・x ＋ b2・(—（a1/a2)・x — （a3/a2)・z） ＋ b3・z ＝ 0

すると z は

（(a2・b1）/a2)・x — （(a1・b2）/a2)・x — （(a3・b2）/a2)・z ＋ （(a2・b3）/a2)・z ＝ 0

から
（(a2・b1 — a1・b2）/a2)・x —　((a3・b2 — a2・b3）/a2)・z ＝ 0
なので
z ＝　((a2・b1 — a1・b2）/（a3・b2 — a2・b3))・x
となります。ここで、垂直なベクトルは何倍しても（0 倍以外）垂直なので、
x ＝ a3・b2 — a2・b3
と置きます。すると
z ＝　((a2・b1 — a1・b2）/（a3・b2 — a2・b3))・x
　＝ a2・b1 — a1・b2
となり、この x と z から式（3）により、
y ＝　—（a1/a2)・x —　(a3/a2)・z
　＝　(—a1・x — a3・z）/a2
　＝　(—a1・a3・b2 ＋ a1・a2・b3 — a3・a2・b1 ・ a3・a1・b2）/a2
　＝　(a1・a2・b3 —　a3・a2・b1）/a2
　＝ a1・b3 — a3・b1
となります。つまり、
（x,　y,　z）＝（a3・b2 — a2・b3,　a1・b3 — a3・b1,　a2・b1 — a1・b2）
です。ごれは外積の値と逆方向（各要素が—1 倍になっている）です。
　つまり、
x ＝ —1・(a3・b2 — a2・b3)
としておけば、外積と同じ値になります。
　外積はクロス積とも呼ばれますが、これはベクトルの要素をたすき掛け（クロス上）に掛け合わせて引き算することで容易に得られます。
　2 つのベクトルを（a1,　a2,　a3）と（b1,　b2,　b3）とする場合次のように要素を並べて隣同士掛け合わせます。

a1　a2　a3　a1
　×　×　×
b1　b2　b3　b1

そして掛けたもの同士を引き算します。つまり
a1・b2 — a2・b1　a2・b3 — a3・b2　a3・b1 — a1・b3
　これはそれぞれ、外積の第 3、第 1、第 2 要素の値になります。これを覚えておけば、掛け算と引き算の順番を忘れることはありません。

■問 3

2 つのベクトルが直角の関係にある（これを直交しているといいます）かどうかは、2 つのベクトルの内積が 0 であるかどうかで判別できます。内積が 0 の場合に直交します。これはその知識を問う問題です。

■問 4

2 つのベクトルの外積が、問 1〜問 3 で作成した関数を使って、元のベクトルと直交するかどうかを確認しています。問題文として外積を求めることを指示しているので、その通りにするだけです。

＝＝＝＝＝＝＝＝＝第 8 章の解説＝＝＝＝＝＝＝＝＝

■問 1

ここでは、直角三角形かどうかの判定には is_right_triangle 関数を使い、直角を挟む 2 辺の長さを求めるために find_right_triangle_sides 関数内で辺の長さを昇順にソートしています。テストケースとして a＝3、b＝4、c＝5 を与えた場合、この三角形は直角三角形であり、直角を挟む 2 辺の長さは 3 と 4 となります。

■問 2

このプログラムでは、is_pythagorean 関数を使用してピタゴラス数かどうかを判定し、find_pythagorean_numbers 関数で 1 辺の長さが 100 以下の全てのピタゴラス数を求めます。最終的に求められたピタゴラス数は、それぞれの a、b、c の値がタプルとしてリストに格納され、出力されます。

■問 3

このプログラムでは、triangle_area 関数を使用して与えられた 3 辺の長さから三角形の面積を計算しています。例として、a＝5、b＝12、c＝13 の三角形の面積を計算し、結果を出力しています。この問題のキーポイントでは

a**0.5

で a の平方根が求められることです。

ところで、ヘロンの公式以外にも三角形の面積を求める方法があります。例えば、3 辺の長さが a、b、c である三角形の面積は以下のように計算できます。

面積＝0.25×√(2a^2b^2＋2b^2c^2＋2c^2a^2＋a^4－b^4－c^4)

ただし、この方法は計算がやや複雑になる上、浮動小数点数の誤差が影響する可能性があります。三角形の面積を求めるためにヘロンの公式が一般的に使用される理由は、計算が簡単であり、誤差に対する頑健性があるためです。

■問 4

このプログラムでは、三角形の 3 つの辺の長さがそれぞれ変数 AB＝a、BC＝b、CA＝c、DE＝d、EF＝e、FD＝f で与えられています。関数 are_triangles_similar を使って、相似かどうかを判定し、結果を表示しています。

are_triangles_similar 関数では、与えられた 6 つの辺の長さを使って、三角形 ABC の辺 AB、BC、CA と三角形 DEF の辺 DE、EF、FD の比率が等しいかどうかをテストします。具体的には a≦b≦c の場合に 2 つのリスト

[a, b, c]

[b, c, a]

を作って、新たな比率のリスト

[a/b, b/c, c/a]

を計算して、三角形の各辺の比率を求めています。これを三角形 ABC と三角形 DEF に適用して、その比率のリストが等しければ True、等しくなければ False を返します。

三角形の相似を求める場合、比率を求める解法では、浮動小数点演算の誤差が気になります。浮動小数点演算の誤差を避けるために、分数を使って比率を求める方法があります。それは Fraction クラスを使って次のように書き換えることができます。

```
from fractions import Fraction

def are_triangles_similar(sides_ABC, sides_DEF):
    def get_ratios(sides):
        ratios = []
        for i in range(3):
            ratio = Fraction(sides[i], sides[(i + 1) % 3])
            ratios.append(ratio)
        return ratios
    ratio_ABC = get_ratios(sorted(sides_ABC))
    ratio_DEF = get_ratios(sorted(sides_DEF))
    print(ratio_ABC)
    print(ratio_DEF)
    return sorted(ratio_ABC) == sorted(ratio_DEF)

# テスト
sides_ABC = [5, 12, 13]
sides_DEF = [10, 24, 26]
print(are_triangles_similar(sides_ABC, sides_DEF))  # True
```

```
sides_ABC = [3, 4, 5]
sides_DEF = [6, 8, 10]
print(are_triangles_similar(sides_ABC, sides_DEF))  # True

sides_ABC = [3, 4, 6]
sides_DEF = [6, 8, 13]
print(are_triangles_similar(sides_ABC, sides_DEF))  # False
```

Fraction は Python の標準ライブラリに含まれるクラスで、有理数を表現するために使用されます。Fraction クラスを使用することで、分数を正確な値として扱うことができます。また、分数同士の加減乗除の演算も可能です。

Fraction クラスは fractions モジュールに含まれており、使用する前にインポートする必要があります。Fraction クラスは以下のようにして使用します。

```
from fractions import Fraction

# 分数の作成
frac1 = Fraction(1, 2)  # 1/2
frac2 = Fraction(3, 4)  # 3/4

# 四則演算
addition = frac1 + frac2    # 1/2 + 3/4 = 5/4
subtraction = frac1 - frac2    # 1/2 - 3/4 = -1/4
multiplication = frac1 * frac2    # 1/2 * 3/4 = 3/8
division = frac1 / frac2    # 1/2 / 3/4 = 2/3

# 分数の変換
decimal = float(frac1)    # 分数を浮動小数点数に変換
numerator = frac1.numerator    # 分子を取得
denominator = frac1.denominator    # 分母を取得
```

Fraction クラスを使用することで、浮動小数点演算による誤差を回避し、正確な分数の計算を行うことができます。特に、小数を分数に変換したり、分数の比較を行う場合に便利です。

＝＝＝＝＝＝＝＝＝＝第 9 章の解説＝＝＝＝＝＝＝＝＝

■問 1

モンテカルロ法を用いて円周率の近似値を求める手法は非常に有名です。しかし、その原理を知っている人は少ないのではないでしょうか。ここで示したように、面積の割合によって求める値の近似値を求めることが基本になります。問 3 などもこの考え方の応用です。ここでの設問は近似値を求める論理を説明しています。

■問 2

問 1 で示したアルゴリズムをプログラムに書き下します。半径が 1 である円は

x*x＋y*y＝1

という関係式で座標が求まることを知ってることが前提の問題です。

■問 3

問題文でアルゴリズムは示してありますから、それを素直にプログラミングするだけです。難しいのは y と－x*x＋2 の関係がどのようなときに、求める図形の中に入るかということです。この大小関係は間違いやすいので注意してください。

■問 4

モンテカルロ法で自然対数の底（e）の近似値を求めることができるということはあまり知られていません。その方法（問題文では定理して提示してある）は簡単なのですが、なぜその方法で自然対数の底が求まるのか証明が難しいからです。この証明は論文に記載されていますので興味のある方は見てみてください。

一様分布の和の平均到達時間 ―e を近似する確率シミュレーション―

https://aue.repo.nii.ac.jp/record/2871/files/epsilon427583.pdf

＝＝＝＝＝＝＝＝＝第 10 章の解説＝＝＝＝＝＝＝＝＝

■問 1

これは超難問です。論理的な思考が問われます。再帰的な考え方と最適な場合分けを考えないと答えにはたどり着けません。

■問 2

完全順列の性質から、いかにして総当たりを行う作れるかを問う問題です。キーポイントは全順列の組み合わせからいかにして完全順列を見つけ出すかです。Python では range(n) が 0～（n－1）の値を示すことを知っていなければ答えにたどり着けません。

■問 3

プログラムの構造としては問 2 と同じです。完全順列をリストに追加する代わりに個数を数えているだけです。

■問 4

これもプログラムの構造としては問 2 と同じです。len 関数を使って全順列の数を求めることに気づければ難しい問題ではありません。

＝＝＝＝＝＝＝＝＝第 11 章の解説＝＝＝＝＝＝＝＝＝

■問 1

M 系列ノイズを求める基本的なプログラムです。シフトの方向は、右シフトまたは左シフトの流派がありますが、今回は左シフトを採用しています。このプログラムは Python で書かれていますが、Python にはデータ内のビットを直接操作する命令がないので不便に感じます。

(product >> 1) & 1

など、verilog HDL のように

product[1]

と記述できれば、プログラムが見やすくなるのに残念です。もっとも、扱うデータをビットの塊ではなく、0 または 1 を要素とするリストにしてしまえば、もっとすっきりしたプログラムになるかもしれません。

■問 2

これは問 1 で作成したプログラムの実行結果を手計算で求める問題です。M 系列ノイズのアルゴリズムを理解できていれば難しいことはありません。

■問 3

これはサービス問題です。M 系列ノイズとランダムノイズのそれぞれの項の排他的論理和（XOR）を取るだけの問題です。Python における zip 関数の挙動を知っているかどうかがキーポイントです。

■問 4

M 系列ノイズは疑似乱数としても使われます。その周期を知っておくことは「乱雑さ」を確保する上で必要です。このプログラムは Python における集合型の理解がキーポイントです。

＝＝＝＝＝＝＝＝＝第 12 章の解説＝＝＝＝＝＝＝＝＝

■問 1

フル・アダーの論理を知る問題です。基本的には 2 つの数の排他的論理和（XOR）と論理積（AND）を計算できるかどうかの問題に帰着できます。

■問 2

最も基本的なリップル・キャリー・アダーをプログラミングで実現します。フル・アダーの論理が分かっていれば、それを数珠つなぎにした構成をそのままプログラミングで書き下すだけです。

■問 3

ここで紹介する P 信号や G 信号はより高速な加算器を構成するための基本となります。P 信号と G 信号を応用したキャリー・ルック・アヘッドをどう構成するかという問 4 への準備問題です。ここで挙げた 3 つのキャリーの性質から、P 信号や G 信号の論理が推測できるかどうかがキーポイントです。

■問 4

問 3 で求めた P 信号と G 信号を使って、4 ビットのキャリー・ルック・アヘッド・アダーをプログラムで実現します。問 3 の後半の設問文に答えはすでに出ています。式を Python のプログラムに書き換えることができるかどうかがキーポイントです。