5.2 流れ図

流れ図 (flow chart) はアルゴリズムを図式化したもので、コンピュータの処理手順となるデータの流れ・判定条件・実行の推移などを**流れ図記号**[4]を用いて描きます。流れ図のようにアルゴリズムを図式化することで、問題の定義や分析または解法がより明確となり、プログラムの設計や作成に非常に役立ちます。また、第三者にも的確にアルゴリズムを伝えることができます。

それでは、流れ図について詳しく見ていきましょう。主な流れ図記号には表 5.1 のような記号があり、これらの記号をアルゴリズムの流れに従って実線で繋いで流れ図を描きます (流れの方向性を明示する場合は矢印で記述します)。

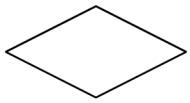
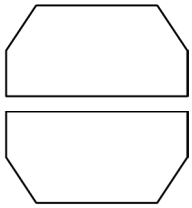
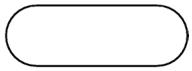
記号	記号名	記号の説明
	処理 (process)	任意の種類の処理を表します。特に、データの値・形・位置を変えるような定義された演算を表す場合に使用されます。
	定義済み処理 (predefined process)	サブルーチン (**??** 節参照) やモジュールなど、別の場所で定義された 1 つ以上の演算または命令群からなる処理を表します。
	準備 (preparation)	変数の初期設定など、その後の動作に影響を与えるための命令または命令群の修飾を表します。
	判断 (decision)	1 つの入力と幾つかの択一的な出口の中で記号中の条件の評価に従って唯一の出口を選ぶ判断を表します。
	ループ端 (loop limit) 上図: ループ始端 下図: ループ終端	1 組のループ始端とループ終端の記号中には同じループ名が入り、記号中の終了条件を満たすまでループ始端からループ終端に挟まれた区間の処理の繰り返しを表します (図 5.3 の⑥参照)。ただし、1 組のループが別のループを含むような場合は必ず**入れ子**構造になります。
	結合子 (connector)	一連の流れ図を分割して描いた場合など、他の部分へ継続していることを表します。流れ図の結合を表します。
	端子 (terminator)	流れ図の始まりと終わりを表します。なお、始まりには「開始 (start)」、終わりには「終了 (end)」と記号中に表記します。

表 5.1: 流れ図記号

[4] このテキストでは JIS X 0121 (1986 年) 情報処理用流れ図記号を用います。この流れ図記号は情報処理技術者試験で使用されています。

また、流れ図はアルゴリズムに従って基本的に上から下に左から右に記述します。具体例として、前節の平方根を求めるアルゴリズムと最大公約数を求めるアルゴリズムの流れ図を挙げておきます。

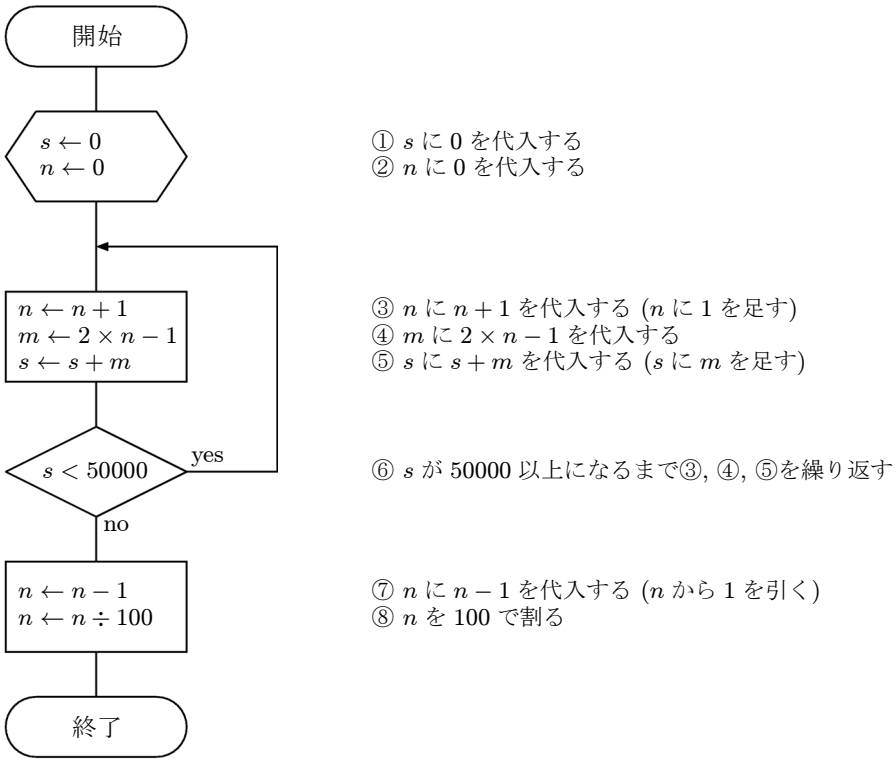


図 5.1: 5 の平方根を小数第 2 位まで求める流れ図

【注意】処理と準備の流れ図記号の中で使用されている「← (矢印)」は、右側の値 (右辺) を左側の変数 (左辺) に代入することを表します。また、判断の流れ図記号の中で使用されている式は命題で、それぞれ

$$
\begin{aligned}
&「A = B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ と変数 } B \text{ が等しい,}\\
&「A \neq B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ と変数 } B \text{ が異なる,}\\
&「A < B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ が変数 } B \text{ より小さい,}\\
&「A > B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ が変数 } B \text{ より大きい,}\\
&「A \leq B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ が変数 } B \text{ より小さいか等しい,}\\
&「A \geq B」 \quad \Longleftrightarrow \quad \text{変数 } A \text{ が変数 } B \text{ より大きいか等しい}
\end{aligned}
$$

を表し、真であれば yes、偽であれば no の方向に処理を進めます。

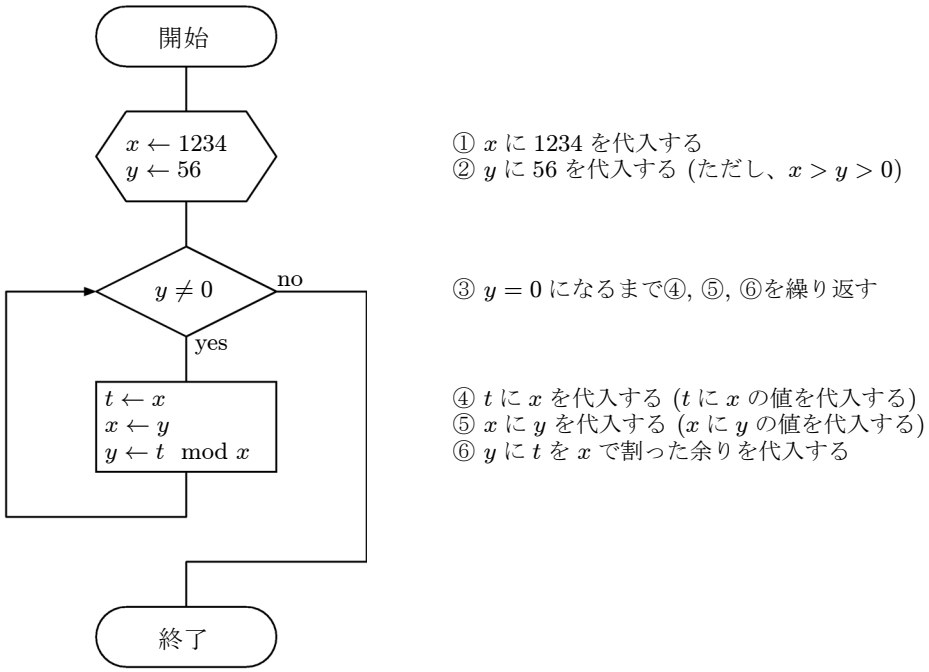


図 5.2: 正の整数 x, y の最大公約数を求める流れ図

現在、アルゴリズム (またはプログラム) の設計において主流となっているのは**構造化プログラミング** (structured programming) と呼ばれる方法です。構造化プログラミングが主流となる前は、特に制限がなかったため各設計者が独自にアルゴリズムの設計を行っており、第三者にも理解し辛いものでした[5]。このような背景と人間は誤解や過ちを起こしやすいという観点から、1966 年に C. Bohm と G. Jacopini によって**構造化定理**という理論が発表されました。この理論は、「1 つの入り口と 1 つの出口を持つように設計されていれば、三つの基本的な論理構造 (図 5.3 の① 順次, ②選択, ④繰り返し (前判定)) の組み合わせで、どんなアルゴリズムの理論も記述できる」というものです。構造化プログラミングはこの理論を取り入れたもので、現在使用されている多くのプログラム言語もこの理論に従っています。

また、構造化プログラミングに関していえば、前記の流れ図記号を用いて流れ図を記述しても構いませんが、階層化された構造を、さらに分かりやすく記述することのできる**構造化チャート**もあります。現在使用されている代表的な構造化チャートには **NS チャート** (Nassi Schneiderman chart) を改良した **PSD** (Program Structure Diagrams) や、日立製作所から発表された **PAD** (Problem Analysis Diagrams) などがあります。

[5]特に問題とされたのが、goto 文の多用によるプログラムの複雑化です。なお、このような状態をスパゲティが複雑に絡み合っている様子に比喩されて**スパゲティ構造**と呼ばれています。

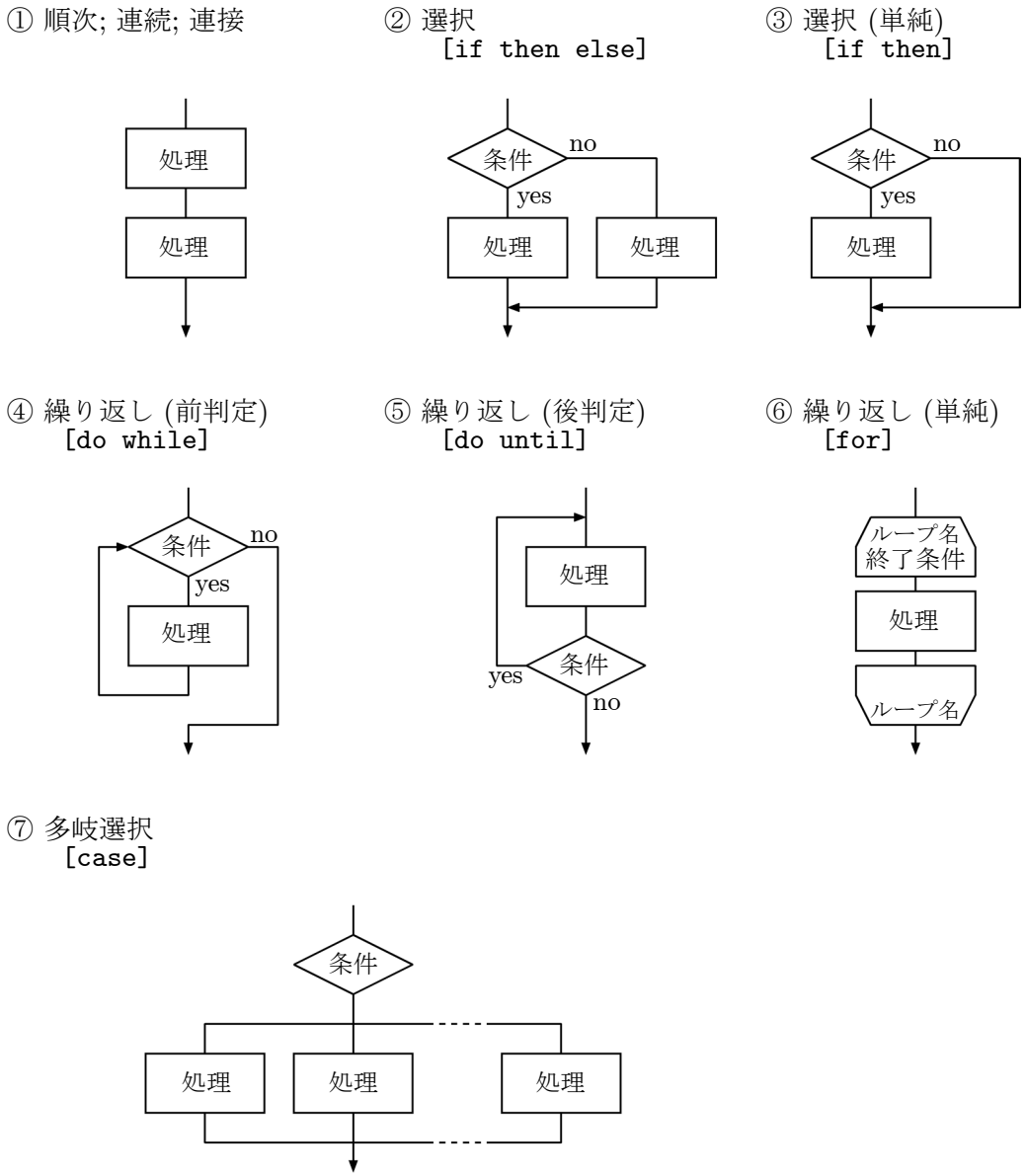


図 5.3: 構造化プログラミングの論理構造

問題 1　5.1.2 節の後半部分を参考に、効率よく平方根の値を求めるアルゴリズムの流れ図を描きなさい。

問題 2　最小公倍数を求めるアルゴリズムの流れ図を描きなさい。

問題 3　ハノイの塔のアルゴリズムの流れ図を描きなさい。

問題 4　図 5.1 の流れ図 (後判定) を繰り返し (前判定) を用いた流れ図に描き直しなさい。

問題 5　図 5.2 の流れ図 (前判定) を繰り返し (後判定) を用いた流れ図に描き直しなさい。

5.3 計算量

世の中には、以下のように解ける問題と解けない問題が存在します。

- **解ける問題**：問題を解決するためのアルゴリズムが存在する。
- **解けない問題**：問題を解決するためのアルゴリズムが存在しない (見つかっていない)。

なお、解ける問題の中には、その答えを見出すのが易しいものから困難なものまで様々です。困難なものとしては、

- 無理数の真の数値 ($\pi = 3.14\cdots$, $e = 2.71\cdots$, $\sqrt{2} = 1.41\cdots$ など)
- 組み合わせ数が非常に多い組み合わせ問題 (ハノイの塔, 巡回セールスマンなど)
- 非常に大きな 2 つの素数の積の因数分解 (公開鍵暗号の安全性を保障)

などがあります。これらの問題は、人間であろうがコンピュータであろうがその答えを導くのは非常に困難です (現時点では無理)。しかしながら、コンピュータが大量のデータを高速かつ正確に処理できるという利点を生かして、ある程度までの答えを見出すことが有意義で意味ある場合も数多く存在します。

情報科学の分野では、問題を解決するためのアルゴリズムをコンピュータで処理させようとするとき、その処理に必要な資源 (計算時間, 記憶容量など) を**計算量** (complexity) という抽象的な尺度で評価します。計算量には、**アルゴリズムの実行にどれだけ時間がかかるか**という尺度に基づいて評価する**時間計算量** (time complexity) と、**アルゴリズムの実行にどれだけ記憶領域 (メモリ) が必要か**という尺度に基づいて評価する**領域計算量** (space complexity) があります。通常は、単に計算量といえば時間計算量のことを指し、処理に必要な計算時間が評価の中心となります[6]。

また、計算量は最悪の入力データを想定して評価します。これを**最大時間計算量** (worst case time complexity) と呼びます。これに対して、全ての入力データに対する計算量の平均値を**平均時間計算量** (average case time complexity) と呼びます。一般には、計算量といえば最大時間計算量のことを指しますが、入力データの良し悪しによって計算量の評価が左右されるような場合は平均時間計算量が重要な意味を持ちます。

計算量を求めるには、n 個の入力データに対してアルゴリズムの処理に必要な時間を算出します。より詳しく述べれば、アルゴリズムの中で実行される

- 比較演算
- 四則演算

などの各演算の回数をカウントし、各演算を 1 回実行するのに必要な時間を掛け、その総和を求めます。例えば、1 個の入力データの処理に 0.001 秒かかる演算が 1000 回必要なとき、n 個のデータを処理するには $0.001 \times 1000 \times n$ $(= n)$ 秒の処理時間が必要となります。もう 1 つの例として、

[6] 領域計算量も時間計算量と同様の評価ができます。ただし、計算時間が増えることに比べれば、コンピュータの記憶領域は比較的小さく限りがあるため、大量にデータを扱う場合は記憶領域を節約するようなアルゴリズムが好まれます (または、記憶領域をなるべく節約するようにプログラムを設計します)。これを時間と空間のトレードオフといい、領域計算量は時間計算量に転換されて評価されます。

1 個の入力データの処理に 0.001 秒かかる演算が 10000 回と 0.005 秒かかる演算が 200 n 回必要なとき、n 個のデータを処理するには $0.001 \times 10000 \times n + 0.005 \times 200\ n \times n\ (= n^2 + 10\ n)$ 秒の処理時間が必要となります。

さらに、計算量には**オーダ記法**という n の値が十分大きなところで計算量を漸近的に評価する手段があります。オーダ記法を用いると、最初の例は $O(n)$, 2 つ目の例は $O(n^2)$ と表記することができます (O はオーダ (Order) の頭文字)。ただし、2 つ目の例が $O(n^2)$ と表記されているのは、n が非常に大きくなると n^2 に比べて 10 n は非常に小さくなり無視されるためです (これを計算量の第 1 次近似といいます)。

最後に、アルゴリズムの計算量による評価について述べておきます。大きく分けてアルゴリズムには、$O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ のように n の多項式のオーダとなる**多項式時間アルゴリズム** (polynomial time algorithm) と、$O(2^n)$, $O(3^n)$, $O(n!)$, $O(n^n)$ のように n の指数関数のオーダとなる**指数時間アルゴリズム** (exponential time algorithm) があります。前者は、n が多少大きくなっても計算量がそれほど大きくならないのに対して、後者は、n が少し増えただけで爆発的に計算量が大きくなります (表 5.2; 1 秒間に 10^9 回 (クロック数が 1GHz) の演算能力を持つコンピュータを使用したものとして算出)。従って、多項式時間アルゴリズムはコンピュータに適したアルゴリズムといえ、指数時間アルゴリズムはコンピュータには適さないといえます。

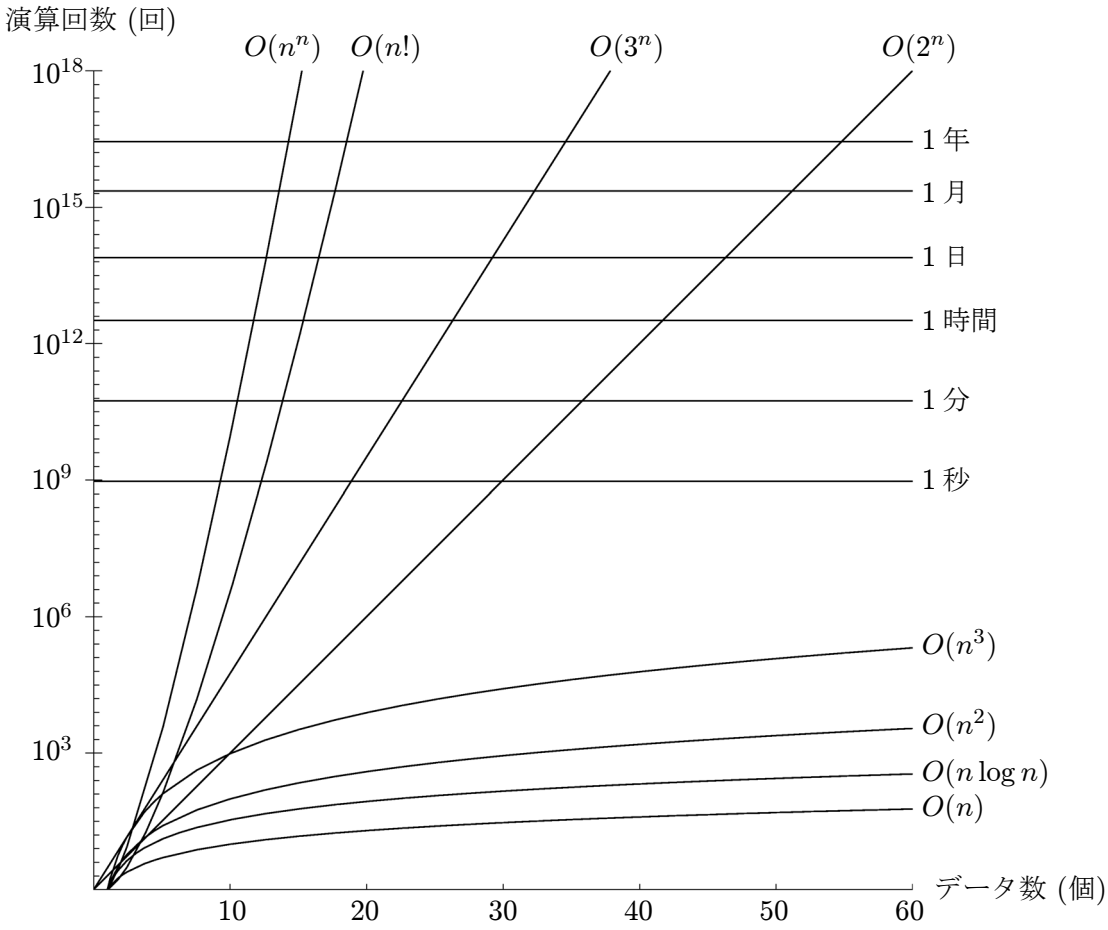


表 5.2: 計算量による評価