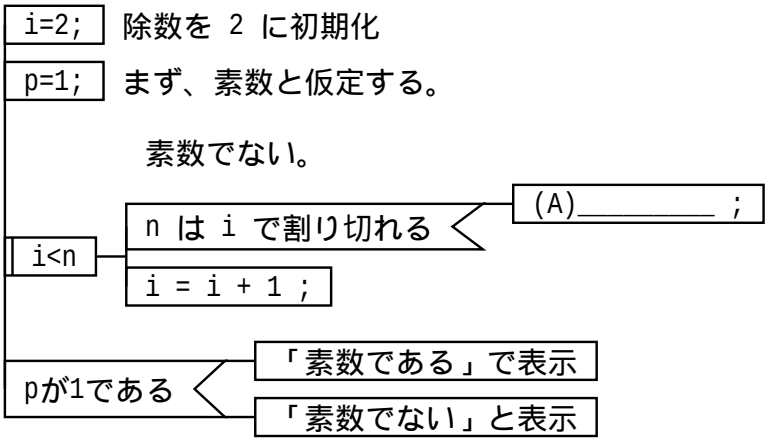


図 1: 素数の判断を定義通り行なうアルゴリズム

## 1 基本的考え方

ある自然数 $n$ が素数 (prime number) であるとは、 1 とそれがその数自身では割り切れないことである。したがって、 $n$ が与えられた場合は、変数 $i$ を 2 から $n-1$ までの数で順に割って、どれも割り切れなければ素数であり、一つでも割り切れる数があったらそれは素数でないと判定できる。

## 2 そのままの方法

そのままの方法によるアルゴリズムの PAD を 図 1に示す。まず素数であるか否かに応じて値 1,0 をとる変数 $p$ を用いる。変数 $p$ は、はじめは $n$ が素数であると仮定して 1 に初期化する。 $i = 2$ からはじめて $i < n$ である間、割り切れるか否かを判定し、割り切れた場合は $p = 0$ にする。反復処理の終了時点の $p$ の値で $n$ が素数であるか否かの結果を得ることができる。

## 3 無駄な判定をしない方法

ある数で割り切れた場合は、素数でないことがその時点で判明するにも関わらず、2の方法では、最後まで割り切れるかどうかを調べ続ける。これは無駄である。これを回避するには、繰り返しの条件に 「p が 1 に等しい」をつけ加えればよい。このアルゴリズムの PAD を図 2に示す。

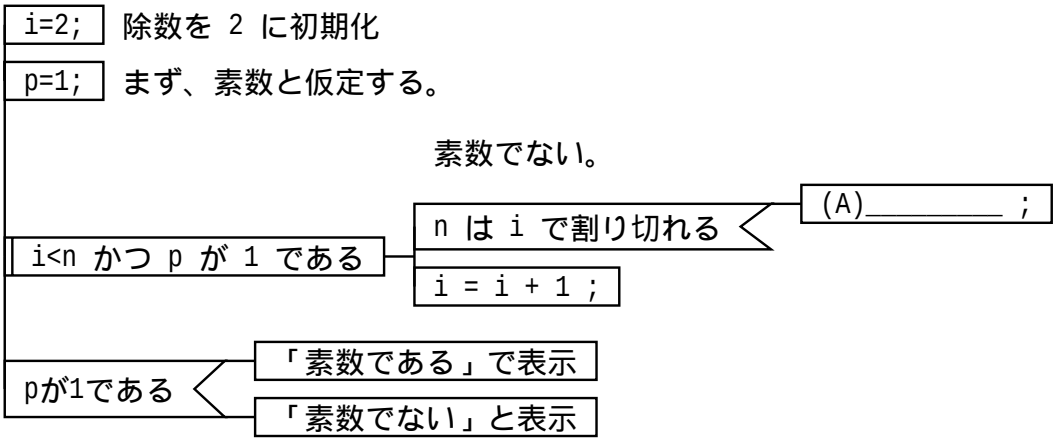


図 2: 無駄な判定をやめるアルゴリズム

## 4　繰り返し処理の抜け出しを用いる方法

3のアルゴリズムでは、繰り返しで毎回 $p$ の判定を行なっているが $p$ の値がいつ変化するかを考えれば、 $p$ に $0$ を代入しているところで直接終了すればよいことがわかる。すると繰り返し判定は $i < n$ の判定だけでよい。この場合、繰り返しを途中で終了した場合は素数でないことがわかるので、変数 $p$ を用いないで、繰り返し処理終了後に $i < n$ であるか否かで素数の判定ができることになる。このアルゴリズムの擬似コードを図 3に示す。

## 5　「番兵」を用いる方法

これまでのアルゴリズムの反復処理には、実質的に二つの条件判断がある。 $n$ が $i$ で割り切れるかどうかと終りまで調べたかどうかである。この問題に関してはもっと便利な判断方法がある。 $n$ は $n$ で常に割り切れるので、繰り返し条件を「$n$ が $i$ で割り切れない」にすると、 $i$ を一つずつ増している限りは必ず $i$ が $n$ に等しくなって繰り返しは正しく終了する。したがって、とくに抜け出しを用いる必要はない。このアルゴリズムの PAD を図 4に示す。これは、「番兵」と同じ効果が自動的に得られた例となっている。

```
{
   i=2;
   while( i<n ){
      if( n は i で割り切れる ){
         (B)_______; /* while から抜け出す */
      }
      i = i + 1 ;
   }
   if( i<n ){
      「素数でない」と表示
   }else{
      「素数である」と表示
   }
}
```

図 3: 繰り返しの抜け出しを用いるアルゴリズム

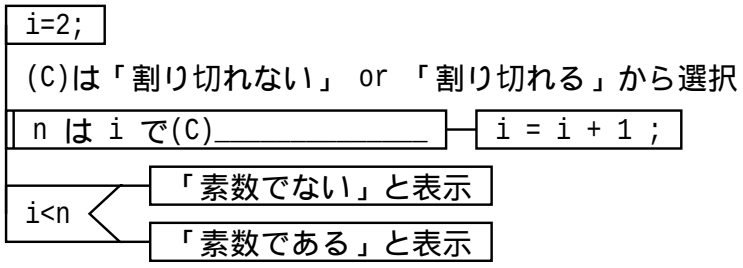


図 4: 「番兵」を用いるアルゴリズム

```
{
   i=2;
   while( i*i<=n ){
      if( n は i で割り切れる ){
         (B)_______; /* while から抜け出す */
      }
      i = i + 1 ;
   }
   if( i*i<=n ){
      「素数でない」と表示
   }else{
      「素数である」と表示
   }
}
```

図 5: より効率的なアルゴリズム

## 6 より効率のよい方法

これまでのアルゴリズムでは、$i = n - 1$ まで、割り切れるかどうかを調べてきた。$n$ を $i$ で割った商を $j$ とすると、$n$ が $i$ で割り切れるれる時には、$j$ でも割り切れるはずである。したがって、$i$ を小さい値から順に変えていった場合、$i > j$ となれば、それ以上調べる必要がない。なぜならそれ以降の $i$ で割り切れる時とすれば、すでに割り切れていたはずだからである。したがって、$i \leqq \sqrt{n}$ の場合だけ調べればよい。これにより 大きい $n$ に対して調べる回数が大幅に減少することが期待できる。このアルゴリズムの疑似コードを図 5に示す。この場合、5で示した「番兵」を用いることはできない。