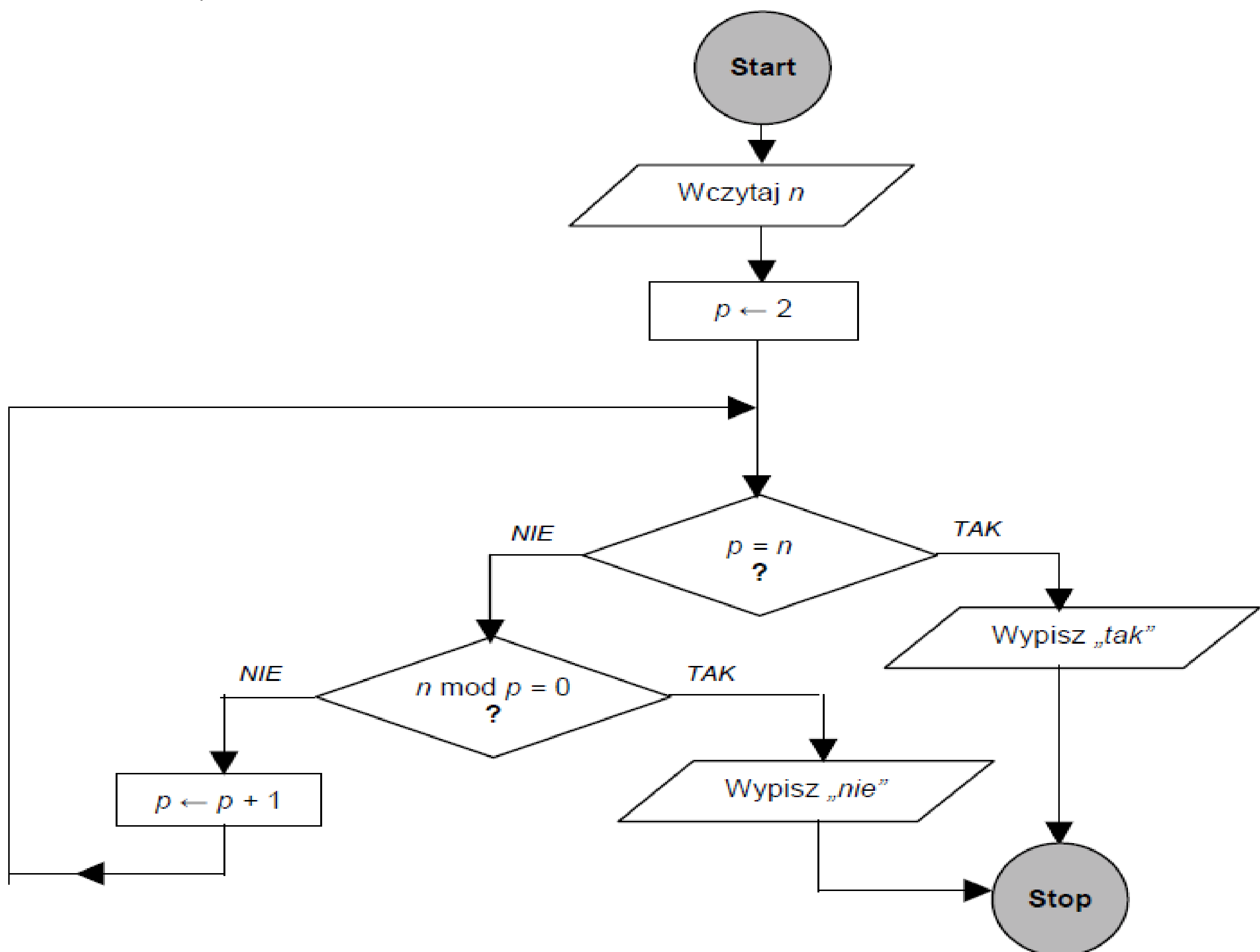
Sprawdzenie czy podana liczba jest liczbą pierwszą

Algorytm sprawdzania, czy *n* jest liczba pierwszą.

Specyfikacja algorytmu
Dane:
n - liczba naturalna większa od 1
Wynik:
komunikat „tak” lub „nie” w zależności od tego, czy n jest liczbą pierwszą

Lista kroków
Krok 1. Wczytaj liczbę n.
Krok 2. Zmiennej p przypisz wartość 2.
Krok 3. Jeśli p jest równe n, wtedy wypisz komunikat „tak” i zakończ działanie algorytmu. W przeciwnym razie przejdź do następnego kroku.
Krok 4. Jeśli n mod p jest równe 0, wtedy wypisz komunikat „nie” i zakończ działanie algorytmu.
Krok 5. Powiększ wartość zmiennej p o 1 i przejdź do kroku 3.

Schemat blokowy



Algorytm można poprawić przypisując p kolejno 2,3,5, …, $\lceil\sqrt{n}\rceil$

<u>Algorytm szukania połówkowego (binarnego)</u>

Załóżmy że mamy daną tablicę n-elementów i chcemy odnaleźć w niej zadany element x. Niech będzie to tablica a o indeksach od 0 do n. Czyli kolejne jej elementy oznaczymy: a[0], a[1], a[2], ..., a[n-1. Jeżeli elementy w tej tablicy są posortowane, to możemy użyć bardzo szybkiego algorytmu wyszukiwania połówkowego, zwanego też wyszukiwaniem binarnym. Poniżej przedstawię jego wersję dla tablicy posortowanej rosnąco. Nazwa tego algorytmu pochodzi od sposobu w jaki szukany jest element.

Opis słowny algorytmu:
Załóżmy, że początkowy indeks naszej tablicy oznaczony jest indeksem *l* a końcowy indeks tablicy oznaczony jest *p*. Na początku oczywiście $l = 0$, a $p = n-1$. Dzielimy tą tablicę na pół elementem o indeksie równym $sr = (l + p) / 2$, przy czym dzielenie to jest dzieleniem bez reszty. Sprawdzamy czy element o indeksie *sr* jest szukanym elementem, jeżeli tak to kończymy działanie algorytmu, gdyż odnaleźliśmy szukany element. W przeciwnym razie sprawdzamy czy element pod indeksem sr jest większy od szukanego. Jeżeli tak jest, to wiemy, że elementy o indeksach *sr+1, sr+2*, ..., *p* również są większe (wynika, to z faktu, że tablica jest posortowana rosnąco). Zatem wiemy już, że w tamtej części nie ma co szukać, ustawiamy zatem indeks końca tablicy *p* na wartość *sr-1* i powtarzamy dzielenie tej tablicy na pół tak, jak już to było opisane wcześniej. Gdy element pod indeksem *sr* jest mniejszy od szukanego, to wiemy, że również elementy o indeksach *sr-1*, *sr-2*, ..., *l* są mniejsze od elementu szukanego (wynika, to z faktu, że tablica jest posortowana rosnąco). Zatem wiemy już, że tamtej części nie ma co szukać, ustawiamy zatem indeks początku tablicy *l* na wartość *sr+1* i powtarzamy dzielenie tej tablicy na pół tak, jak już to było opisane wcześniej. Opisane czynności wykonujemy tak długo aż znajdziemy szukany element bądź wskaźnik początku tablicy *l* będzie większy od wskaźnika końca tablicy *p*. Jeżeli tak się stanie oznacza to, że szukanego elementu nie ma w przeszukiwanej tablicy.

```
void szukaj(int a[],int n,int liczba)
int l = 0,p = n-1,sr;
      while (l < p)
      {
           sr = (l+p)/2;
           if (a[sr] == liczba)
           {
               cout << "Odnaleziono liczbe " << liczba << " pod indeksem "
               << sr+1 << endl;
               break;
           }
           else if (a[s] < liczba)
               l = sr+1;
           else
               p = sr-1;
      }
      if (l > p)
           cout << "Nie odnaleziono szukanego elementu" << endl;
}
```

Algorytm wyznaczania miejsca zerowego funkcji

Algorytm dotyczy znajdywanie miejsca zerowego funkcji czyli takiego punktu x, że f(x)=0. Aby można było zastosować metodę bisekcji dla funkcji ciągłej, funkcja musi przyjmować różne znaki na końcach przedziału <a,b> czyli f(a)f(b) < 0.

Opis algorytmu:
Krok 1. Sprawdzić, czy pierwiastkiem równania jest punkt c = (a+b)/2, czyli czy f(c) = 0.
Krok 2. Jeżeli tak jest, algorytm kończy się, a punkt c jest miejscem zerowym. W przeciwnym razie c dzieli przedział <a,b> na dwa mniejsze przedziały <a,c> i <c,b>.
Krok 3. Wybierany jest ten przedział, dla którego spełnione jest założenie, tzn. albo f(c)f(a) < 0 albo f(c)f(b) < 0.
Krok 3. Jeżeli nie osiągnięto zadowalającej dokładności przybliżenia pierwiastka należy wrócić do kroku 1.

```
int main()
{// program znajduje pierwiastek równania metoda połówkowa (bisekcji)

	clrscr();
	double a,b,c;
	cout<<"Podaj krance przedzialow: "
	cin >> a >> b;
	double fa = f(a),fb = f(b),fc,eps;
	cout<<"Podaj dokladnosc: ";cin>>eps;
	do
	{
		c = (a+b)/2;fc = f(c);
		if (fc*fa<0)
		{
			b = c; fb = fc;
		}
		else
		{
			a = c; fa = fc;
		}
	}while (fabs(fc)>eps);
	cout << setprecision(15) << "\nx="<<c;
	cout <<" f(x)=" << setprecision(15) << fc;
}
```

Schemat Hornera obliczania wartości wielomianu

Dany wielomian

$$W(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

przekształcamy do postaci

$$W(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x a_n) \ldots)).$$

Następnie definiujemy:

$$\begin{aligned} b_n &:= a_n, \\ b_{n-1} &:= a_{n-1} + b_n x, \\ &\vdots \\ b_0 &:= a_0 + b_1 x. \end{aligned}$$

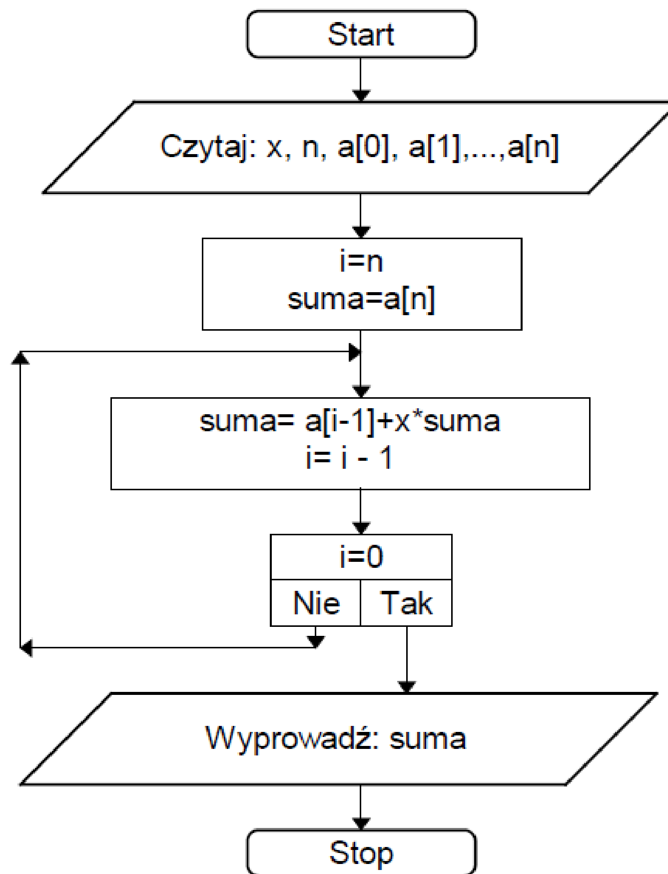
Tak otrzymane b_0 będzie równe W(x). Rzeczywiście, jeśli podstawimy kolejno do tego wielomianu b_n, …, b_0, otrzymamy

$$\begin{aligned} W(x) &= a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1} + b_n x))), \\ W(x) &= a_0 + x(a_1 + x(a_2 + \cdots x b_{n-1})), \\ &\vdots \\ W(x) &= a_0 + x b_1, \\ W(x) &= b_0. \end{aligned}$$

```
double horner(double x,int n,double a[])
{/* horner oblicza wartość wielomianu w punkcie algorytmem Hornera
   a[0],...,a[n] - kolejne współczynniki wielomianu
   x - punkt, w którym obliczana jest wartość
   n - stopień wielomianu*/

   int i = n;
   double b = a[i];
   while(i-->0)
     b = a[i] + b*x;
   return b;
}
```

Poniżej znajduje się schemat blokowy dla tego algorytmu.



Algorytm Euklidesa

Algorytm dotyczy znajdowania największego wspólnego dzielnika (NWD) dwóch liczb naturalnych. Opiera się na twierdzeniu

$$NWD(a,b) = \begin{cases} a & \text{dla } b = 0 \\ NWD(b, a \bmod b) & \text{dla } b \geqslant 1 \end{cases}$$

Krok 1. Oblicz c jako resztę z dzielenia a przez b.
Krok 2. Zastąp a liczbą b, a b liczbą c.
Krok 3. Jeżeli b = 0, to szukane NWD = a, w przeciwnym wypadku przejdź do kroku 1.

```
int nwd (int a, int b)
{
	int c;
	while (b != 0)
	{
		c = a % b;
		a = b;
		b = c;
	}
	return a;
}
```

Można również zrealizować ten algorytm wykorzystując wyłącznie operacje odejmowania.

```
int nwd(int a,int b)
{
   while (a!=b)
        if (a>b) a=a-b;
        else b=b-a;
   return a;
}
```

Sito Eratostonesa

Sito Eratostenesa jest algorytmem służącym do wyznaczenia wszystkich liczb pierwszych z zakresu od 2 do danego n. Działanie algorytmu polega na wykreślaniu z danego zbioru (2, 3, 4, ..., n) wszystkich wielokrotności kolejnych liczb (większych od nich), które nie zostały do tej pory wykreślone, przy czym wystarczy wziąć pod uwagę tylko liczby z zakresu od 2 do wartości całkowitej z pierwiastka z n. Wszystkie liczby, które pozostaną niewykreślone, są liczbami pierwszymi.

```
bool lnp[n + 1]; // tablica o indeksach od 0 do 100 - wszystkie false

int main()
{
        for (int i = 2; i*i <= n; i++ ) // przeszukuj liczby od 2 do
        {                  sqrt(n), 0 i 1 nie są liczbami pierwszymi
             if (lnp[i]) // jeżeli dana liczb jest już wykreślona
                       continue; // to przejdź do kolejnej
             for (int j = 2 * i ; j <= n; j += i)
                // przejdź od liczby 2 * i do n przesuwając się o i
                       lnp[j] = true; // i każdą z nich usuwaj ze zbioru
        }
        cout << "Liczby pierwsze z przedziału od 2 do n:" << endl;
        for (int i = 2; i <= n; i++) // przeszukaj liczby od 2 do n
              if (!nlp[i]) // jeśli liczba nie została usunięta ze zbioru
                       cout << i << endl; // to ją wypisz
        return 0;
}
```

Algorytmy sortowania

```
void prostewstawianie(int a[],int n)
{/* sortowanie tablicy a[] metodą prostego wstawiania – sposób stosowany
przez grających w karty*/
     int j;
     int x;
     for(int i=2;i<=n;i++)
     {
          x=a[i]; //wzięcie kolejnej liczby
          a[0]=x;
          //a[0] pełni rolę wartownika, liczby są w a od a[1] do a[n]
          j=i-1;
          while (x<a[j])
          {
                 a[j+1]=a[j];j=j-1; //przesuwanie liczb w prawo
          }
          a[j+1]=x;//wstawienie liczby na zwolnione dla niej miejsce
     }
}

void prostewybieranie(int a[],int n)
{/* sortowanie tablicy a[] metodą prostego wybierania*/
     int k,x;
     for(int i=0;i<n-1;i++) //n-1 obrotów
     {
          k=i;
          for(int j=i+1;j<n;j++)
          //szukanie położenia najmniejszej liczby w ciągu a[i],...,a[n-1]
                 if (a[j]<a[k])
                         k=j;
          x=a[k];a[k]=a[i];a[i]=x;//zamiana liczby najmniejszej z a[i]
     }
}

void sortowaniebabelkowe(int a[],int n)
{/* sortowanie tablicy a[] metodą sortowania bąbelkowego*/
     int x;
     for(int i=1;i<n;i++)
          for(int j=n-1;j>=i;j--)
               if (a[j-1]>a[j])
               {
                    x=a[j-1];a[j-1]=a[j];a[j]=x;/*zamiana sąsiednich liczb
                    stojących w niewłaściwej kolejności*/
               }
}
```

```
void podzial(int a[],int l,int p)
{/*podzielenie tablicy na dwie części a następnie
sortowanie rekurencyjnie obu części*/
      int i,j;
      int x,w;
      i=l;j=p;x=a[(l+p)/2]; //wybranie środkowej liczby
      do
      {
             while(a[i]<x) //szukanie kandydata do przerzutu z lewej stony
               i++;
             while( x<a[j]) //szukanie kandydata do przerzutu z prawej
               j--;
             if (i<=j)
             {
               w=a[i];a[i]=a[j];a[j]=w;//wykonanie zamiany
               i=i+1;j=j-1;
             }
      }while(i<=j);
      if (l<j) podzial(a,l,j);//sortowanie lewej części
      if (i<p)  podzial(a,i,p);//sortowanie prawej części
}

void sortowanieszybkie(int a[],int n)
{/* sortowanie tablicy a[] metodą szybkiego sortowania */
  podzial(a,0,n-1);
}
```

Sortowanie przez scalanie

Algorytm ten jest dobrym przykładem algorytmów typu *dziel i zwyciężaj*, których ideą działania jest podział problemu na mniejsze części, których rozwiązanie jest już łatwiejsze.

Wyróżnić można trzy podstawowe kroki:

1. Podziel zestaw danych na dwie, równe części (w przypadku nieparzystej liczby wyrazów jedna część będzie o 1 wyraz dłuższa);
2. Zastosuj sortowanie przez scalanie dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element;
3. Połącz posortowane podciągi w jeden.

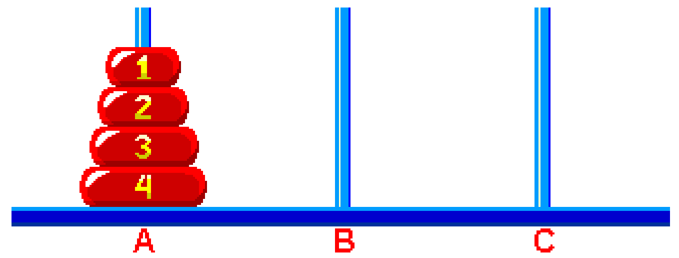
Procedura scalania dwóch ciągów a[1..n] i b[1..m] do ciągu c[1..m+n]:

1. Utwórz indeksy na początki ciągów **a** i **b**: i=0, j=0.
2. Jeżeli ciąg **a** wyczerpany (i>n), dołącz pozostałe elementy ciągu **b** do **c** i zakończ pracę.
3. Jeżeli ciąg **b** wyczerpany (j>m), dołącz pozostałe elementy ciągu **a** do **c** i zakończ pracę.
4. Jeżeli $a[i] \leq b[j]$ dołącz a[i] do **c** i zwiększ i o jeden, w przeciwnym przypadku dołącz b[j] do **c** i zwiększ j o jeden.
5. Powtarzaj od kroku 2 aż wszystkie wyrazy **a** i **b** trafią do **c**.

```
/* Scalanie dwóch posortowanych ciągów
tab[pocz...sr] i tab[sr+1...kon] i
wynik zapisuje w tab[pocz...kon] */
void merge(int pocz, int sr, int kon)
{
    int i, j, q;
    for (i = pocz; i <= kon; i++)
         t[i] = tab[i];// Skopiowanie danych do tablicy pomocniczej
    i = pocz; j = sr + 1; q = pocz;// Ustawienie indeksów
    while (i <= sr && j <= kon) // Przenoszenie danych z sortowaniem ze
    {                            zbiorów pomocniczych do tablicy głównej
         if (t[i] < t[j])
              tab[q++] = t[i++];
         else
              tab[q++] = t[j++];
         }
    while (i <= sr)
    tab[q++] = t[i++];// Przeniesienie nie skopiowanych danych ze zbioru
    pierwszego w przypadku, gdy drugi zbiór się skończył
}
```

```
/* sortowanie tab[pocz...kon] */
void mergesort(int pocz, int kon)
{
    int sr;
    if (pocz < kon) {
         sr = (pocz + kon) / 2;
         mergesort(pocz, sr);// Dzielenie lewej części
         mergesort(sr + 1, kon);// Dzielenie prawej części
         merge(pocz, sr, kon);// Łączenie części lewej i prawej
    }
}
```

Wieże Hanoi



Problem polega na przełożeniu krążków ze słupka A na C zachowaniem kształtu, wieży z krążków o różnych średnicach, przy czym podczas przekładania wolno się posługiwać dodatkowym słupkiem B, jednak przy ogólnym założeniu, że nie wolno kłaść krążka o większej średnicy na mniejszy ani przekładać kilku krążków jednocześnie. Najczęściej rozwiązanie przedstawia się w postaci rekurencyjnej.

```
void hanoi(int n, char A, char B, char C)
{
     if (n > 0)
     {
         hanoi(n-1, A, C, B); //przełożenie n-1 krążków z A na B
         cout << A << " -> " << C << endl; //przełożenie największego na C
         hanoi(n-1, B, A, C); //przełożenie n-1 krążków z B na C
     }
}

int main()
{
     int n;
     cin>>n;
     hanoi(n, 'A', 'B', 'C');
     return 0;
}
```