Instytut Informatyki
Politechnika Śląska

Algorytmy i struktury danych

Opracował: Zbigniew Czech

Materiały dydaktyczne
Gliwice, wrzesień 2011

Spis treści

1 Dowodzenie poprawności algorytmów

Poprawność częściowa

Przed dowodem poprawności

$\{(n > 0)$ **and** $(m > 0)\}$ warunek wstępny (asercja wejściowa)
begin
  $iloczyn := 0;$
  $N := n;$
  repeat
    $iloczyn := iloczyn + m;$
    $N := N - 1;$
  until $N = 0;$
end;
$\{iloczyn = n * m\}$ warunek ostateczny (asercja wyjściowa)

Po przeprowadzeniu dowodu

$\{(n > 0)$ **and** $(m > 0)\}$
begin
  $iloczyn := 0;$
  $N := n;$
  repeat
    {Niezmiennik: $(iloczyn + N * m = n * m)$ **and** $(N > 0)\}$
    $iloczyn := iloczyn + m;$
    $N := N - 1;$
    $\{(iloczyn + N * m = n * m)$ **and** $(N \geqslant 0)\}$
  until $N = 0;$
  $\{(iloczyn + N * m = n * m)$ **and** $(N = 0)\}$
end;
$\{iloczyn = n * m\}$

Proces obliczeniowy

Instrukcje		Wyniki obliczeń	Niezmiennik iteracji
$n = 5$ $m = 8$			
$iloczyn := 0;$	$N := n;$	$iloczyn = 0, N = 5, m = 8$	$(0 + 5 * 8 = 40)$ **and** $(N > 0)$
$iloczyn := iloczyn + m$	$N := N - 1$	$iloczyn = 8, N = 4, m = 8$	$(8 + 4 * 8 = 40)$ **and** $(N > 0)$
$iloczyn := iloczyn + m$	$N := N - 1$	$iloczyn = 16, N = 3, m = 8$	$(16 + 3 * 8 = 40)$ **and** $(N > 0)$
$iloczyn := iloczyn + m$	$N := N - 1$	$iloczyn = 24, N = 2, m = 8$	$(24 + 2 * 8 = 40)$ **and** $(N > 0)$
$iloczyn := iloczyn + m$	$N := N - 1$	$iloczyn = 32, N = 1, m = 8$	$(32 + 1 * 8 = 40)$ **and** $(N > 0)$
$iloczyn := iloczyn + m$	$N := N - 1$	$iloczyn = 40, N = 0, m = 8$	$(40 + 0 * 8 = 40)$ **and** $(N = 0)$

Procedura assert oraz przykład jej użycia

procedure *assert*(b: *Boolean*; t: *string*);
  if not b **then write**(t);

assert($(n > 0)$ **and** $(m > 0)$, "not 1");
begin
  $iloczyn := 0;$
  $N := n;$
  repeat
    assert($(iloczyn + N * m = n * m)$ **and** $(N > 0)$, "not 2");
    $iloczyn := iloczyn + m;$
    $N := N - 1;$
    assert($(iloczyn + N * m = n * m)$ **and** $(N \geqslant 0)$, "not 3");

until $N = 0$;
$assert((iloczyn + N * m = n * m)$ **and** $(N = 0)$, "not 4");
end;
$assert(iloczyn = n * m$, "not 5");

Skończoność algorytmu (warunek stopu)

Przykład 1

$\{(n > 0)$ **and** $(m > 0)\}$
begin
  $iloczyn := 0$;
  $N := n$;
  repeat
    $\{0 < N = N_0\}$
    $iloczyn := iloczyn + m$;
    $N := N - 1$;
    $\{0 \leqslant N < N_0\}$
  until $N = 0$;
end;

Przykład 2

$\{(x \geqslant 0)$ **and** $(y > 0)\}$
begin
  $q := 0$;
  $r := x$;
  while $r \geqslant y$ **do**
    $\{r > 0\}$
    $r := r - y$; $\{r$ zmniejsza się pozostając nieujemne bo $r \geqslant y\}$
    $q := 1 + q$;
    $\{r \geqslant 0\}$
  end while;
end;

2 Złożoność obliczeniowa algorytmów

Tabela 1: Maksymalny rozmiar danych, jaki można przetworzyć w zadanym czasie

Algorytm	Złożoność czasowa	Maksymalny rozmiar danych 1 sek	1 min	1 godz
A_1	n	10^9	60×10^9	3.6×10^{12}
A_2	$n \log n$	40×10^6	2×10^9	10^{11}
A_3	n^2	32×10^3	250×10^3	2×10^6
A_4	n^3	10^3	4×10^3	15×10^3
A_5	2^n	30	36	42

2.1 Obliczanie wartości wielomianu

$$W(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$$

Algorytm 1: Bezpośredni (na podstawie wzoru)

Tabela 2: Czasy przetwarzania danych o zadanym rozmiarze

Algorytm	Złożoność czasowa	Czas przetwarzania danych o rozmiarze $n = 200000$
A_1	n	0.2 ms
A_2	$n \log n$	3.5 ms
A_3	n^2	40 sek
A_4	n^3	3 miesiące
A_5	2^n	2^{200000}ns =?
		2^{70} ns $\cong$ 36 mln lat

Tabela 3: 10-krotne przyspieszenie komputera

Algorytm	Złożoność czasowa	Maksymalny rozmiar danych przed przyspieszeniem	Maksymalny rozmiar danych po przyspieszeniu
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	$10s_2$ (dla dużych s_2)
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

begin
  $W := a[0]$;
  for $i := 1$ **to** n **do**
    $p := x$;
    for $j := 1$ **to** $i - 1$ **do**
      $p := p * x$;   {Potęgowanie zastąpione mnożeniami.}
    end for;
    $W := a[i] * p + W$;
  end for;
end;

Algorytm 2: Hornera

$$
\begin{aligned}
W(x) &= (a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_2 x + a_1)x + a_0 = \\
&= ((a_n x^{n-2} + a_{n-1} x^{n-3} + \ldots + a_2)x + a_1)x + a_0 = \\
&\quad \ldots \\
&= ((\ldots (a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_2)x + a_1)x + a_0
\end{aligned}
$$

begin
  $W := a[n]$;
  for $i := n - 1$ **downto** 0 **do**
    $W := W * x + a[i]$;
  end for;
end;

2.2 Znajdowanie maksymalnego (minimalnego) elementu

Algorytm 1

begin

```
  Max := A[1];
  i := 1;
  while i < n do
    i := i + 1;
    if A[i] > Max then
      Max := A[i];
    end if;
  end while;
end;
```

Algorytm 2 (z wartownikiem)

```
begin
  i := 1;
  while i ⩽ n do
    Max := A[i];
    A[n + 1] := Max;      {Ustawienie wartownika.}
    i := i + 1;
    while A[i] < Max do
      i := i + 1;
    end while;
  end while;
end;
```

2.3 Jednoczesne znajdowanie maksymalnego i minimalnego elementu

Zadanie: Należy znaleźć jednocześnie maksymalny i minimalny element w n-elementowym zbiorze S

Algorytm 1: Szukanie elementu maksymalnego i minimalnego oddzielnie

```
begin
  Max := dowolny element zbioru S;
  for pozostałych elementów x ze zbioru S do
    if x > Max then
      Max := x;
    end if;
  end for;
end;
```

W podobny sposób można znaleźć element minimalny.

Algorytm 2: Metoda „dziel i zwyciężaj" (ang. divide and conquer)

Ograniczenie: Liczba elementów zbioru S winna być potęgą liczby 2, tj. $n = 2^k$ dla pewnego k.

```
procedure MaxMin(S);
begin
  if #S = 2 then
    Niech S = {a, b};
    return(MAX(a, b), MIN(a, b));
  else
    Podzielić S na dwa równe podzbiory S1 i S2;
    (max1, min1) := MaxMin(S1);
    (max2, min2) := MaxMin(S2);
    return(MAX(max1, max2), MIN(min1, min2));
  end if;
end;
```

2.4 Sortowanie przez łączenie

procedure $SORT(i,\ j\colon integer)$;
begin
  if $i = j$ **then**
    return(x_i);
  else
    $m := (i + j - 1)/2$;
    return(*ŁĄCZENIE*($SORT(i,\ m),\ SORT(m + 1,\ j)$));
  end if;
end;

3 Programowanie dynamiczne

Algorytm 1: Wyznaczanie wartości współczynnika dwumianowego metodą „dziel i zwyciężaj”

function $wsp(n, m : integer) : integer$;
$\{0 \leqslant m \leqslant n\}$
begin
  if $(n = m)$ **or** $(m = 0)$ **then**
    $wsp := 1$;
  else
    $wsp := wsp(n - 1, m) + wsp(n - 1, m - 1)$
end;

Algorytm 2: Wyznaczanie wartości współczynnika dwumianowego metodą programowania dynamicznego

for $j := 0$ **to** n **do** {inicjalizacja}
  $pom[j] := 1$;
end for;
for $i := 2$ **to** n **do** {kolejne fazy}
  $\{pom[j] = \binom{i-1}{j}$, dla $0 \leqslant j \leqslant i - 1\}$
  for $j := i - 1$ **downto** 1 **do**
    $pom[j] := pom[j] + pom[j - 1]$;
  end for;
  $\{pom[j] = \binom{i}{j}$, dla $0 \leqslant j \leqslant i\}$
end for;
$\{pom[j] = \binom{n}{j}$, dla $0 \leqslant j \leqslant n\}$

4 Kopce i kolejki priorytetowe

4.1 Procedury operujące na kopcu

procedure *przemieść_w_górę*(p: *integer*);
  {Element $A[p]$ przemieszczany jest w górę kopca.
    Warunek wstępny: *kopiec*$(1,\ p - 1)$ i $p > 0$.
    Warunek ostateczny: *kopiec*$(1,\ p)$.}
  var d, r: *integer*;   {d — dziecko, r — rodzic}
begin
  $d := p$;
  loop
    {Niezmiennik: *kopiec*$(1,\ p)$ z wyjątkiem być może
      relacji pomiędzy $A[d]$ i jego rodzicem. $1 \leqslant d \leqslant p$.}

```
    if d = 1 then
      break;
    end if;
    r := d div 2;
    if A[r] ⩽ A[d] then
      break;
    end if;
    zamiana(A[r], A[d]);
    d := r;
  end loop;
end; {przemieść_w_górę}
```

Procedura *przemieść_w_górę* **z użyciem wartownika**

```
procedure przemieść_w_górę(p: integer);
  var d: integer;
      x: ... ;   {Typ zgodny z typem A.}
begin
  x := A[p];
  A[0] := x;      {Ustawienie wartownika.}
  d := p;
  loop
    r := d div 2;
    if A[r] ⩽ x then
      break;
    end if;
    A[d] := A[r];
    d := r;
  end loop;
  A[d] := x;
end; {przemieść_w_górę}

procedure przemieść_w_dół(p: integer);
  {Element A[1] przemieszczany jest w dół kopca.
    Warunek wstępny: kopiec(2, p) i p ⩾ 0.
    Warunek ostateczny: kopiec(1, p).}
  var d, r: integer;
begin
  r := 1;
  loop
    {Niezmiennik: kopiec(1, p) z wyjątkiem być może relacji
      pomiędzy A[r] i jego dziećmi. 1 ⩽ r ⩽ p.}
    d := 2 * r;
    if d > p then
      break;
    end if;
    {d jest dzieckiem lewym r}
    if (d + 1 ⩽ p) cand (A[d + 1] < A[d]) then
      d := d + 1;
    end if;
    {d jest najmniejszym dzieckiem r}
    if A[r] ⩽ A[d] then
      break;
    end if;
    zamiana(A[r], A[d]);
    r := d;
  end loop;
end; {przemieść_w_dół}
```

4.2 Abstrakcyjny opis kolejki

1. Operacja *zeruj*

procedure *zeruj*;
{Warunek wstępny: brak.
Warunek ostateczny: $K = \emptyset$.}

2. Operacja *wstaw*

procedure $wstaw(t)$;
{Warunek wstępny: $|K| < max\_rozmiar$.
Warunek ostateczny: Bieżąca K = Poprzednia $K \cup \{t\}$.}

3. Operacja *usuń_min*

procedure $usuń\_min(t)$;
{Warunek wstępny: $|K| > 0$.
Warunek ostateczny: Poprzednia K = Bieżąca $K \cup \{t\}$ oraz
$t = \min(\textit{Poprzednia } K)$.}

4.3 Budowowanie kolejki za pomocą kopca

1. Operacja *zeruj*

procedure *zeruj*;
begin
  $n := 0$;
end;

2. Operacja *wstaw*

procedure $wstaw(t)$;
begin
  if $n \geqslant max\_rozmiar$ **then**
    błąd;
  else
    $n := n + 1$;
    $A[n] := t$;
    $\{kopiec(1,\ n-1)\}$
    przemieść_w_górę(n);
    $\{kopiec(1,\ n)\}$
  end if;
end; $\{wstaw\}$

3. Operacja *usuń_min*

procedure $usuń\_min(t)$;
begin
  if $n < 1$ **then**
    błąd;
  else
    $t := A[1]$;
    $A[1] := A[n]$;
    $n := n - 1$;
    $\{kopiec(2,\ n)\}$
    przemieść_w_dół(n);
    $\{kopiec(1,\ n)\}$
  end if;
end; $\{usuń\_min\}$

4.4 Sortowanie przez kopcowanie

```
type
  typ_rekordowy = record
                    klucz: typ_klucza;     {Typ skalarny.}
                    {Pozostałe składowe rekordu.}
                  end;
var
  A: array[1 .. n] of typ_rekordowy;     {Sortowana tablica.}
```

Zmodyfikowana procedura przemieść_w_dół

```
procedure przemieść_w_dół(l, p: integer);
  {Element A[l] przemieszczany jest w dół kopca.
    Warunek wstępny: kopiec(l + 1, p) i l ⩽ p.
    Warunek ostateczny: kopiec(l, p).}
var d, r: integer;
    t: typ_rekordowy;
begin
  r := l;  t := A[r];
  loop
    {Niezmiennik: kopiec(l, p) z wyjątkiem być może relacji
      pomiędzy r i jego dziećmi.  l ⩽ r ⩽ p.}
    d := 2 * r;
    if d > p then
      break;
    end if;
    {d jest dzieckiem lewym r.}
    if (d < p) cand (A[d + 1].klucz > A[d].klucz) then
      d := d + 1;
    end if;
    {d jest największym dzieckiem r.}
    if t.klucz ⩾ A[d].klucz then
      break;
    end if;
    A[r] := A[d];
    r := d;
  end loop;
  A[r] := t;
end; {przemieść_w_dół}

procedure sortowanie_przez_kopcowanie;
  {Sortowana jest tablica A[1 .. n].}
  var i: integer;
begin
  {Faza 1: Budowanie kopca.}
  for i := n div 2 downto 1 do
    {Niezmiennik: kopiec(i + 1, n).}
    przemieść_w_dół(i, n);
    {kopiec(i, n).}
  end for;
  {kopiec(1, n).}
  {Faza 2: Właściwe sortowanie.}
  for i := n downto 2 do
    {Niezmiennik: kopiec(1, i) i posortowane(i + 1, n)
      i A[1 .. i].klucz ⩽ A[i + 1 .. n].klucz.}
    zamiana(A[1], A[i]);
```

```
    {kopiec(2, i − 1) i posortowane(i, n)
      i A[1 .. i − 1].klucz ⩽ A[i .. n].klucz.}
    przemieść_w_dół(1, i − 1);
    {kopiec(1, i − 1) i posortowane(i, n)
      i A[1 .. i − 1].klucz ⩽ A[i .. n].klucz.}
  end for;
  {posortowane(1, n)}
end; {sortowanie_przez_kopcowanie}
```

5 Wyszukiwanie

5.1 Wyszukiwanie liniowe

```
procedure wyszukiwanie_liniowe_1(x: typ_klucza;
                                 var jest: Boolean;
                                 var i: 1 .. n + 1);
begin
  i := 1;
  while (i ⩽ n) cand (A[i].klucz <> x) do
    i := i + 1;
  end while;
  jest := i <> (n + 1);
end;
```

5.2 Wyszukiwanie liniowe z zastosowaniem wartownika

```
procedure wyszukiwanie_liniowe_2(x: typ_klucza;
                                 var jest: Boolean;
                                 var i: 1 .. n + 1);
begin
  A[n + 1].klucz := x;     {Wartownik.}
  i := 1;
  while A[i].klucz <> x do
    i := i + 1;
  end while;
  jest := i <> (n + 1);
end;
```

5.3 Wyszukiwanie liniowe w uporządkowanej tablicy

```
procedure wyszukiwanie_liniowe_3(x: typ_klucza;
                                 var jest: Boolean;
                                 var i: 1 .. n + 1);
begin
  i := 1;
  A[n + 1].klucz := ∞;     {∞ > x}
  while A[i].klucz < x do
    i := i + 1;
  end while;
  jest := A[i].klucz = x;
end;
```

5.4 Wyszukiwanie binarne (logarytmiczne)

```
procedure wyszukiwanie_binarne(x: typ_klucza;
                               var jest: Boolean;
                               var m: 1 .. n);
```

```
  var lewy, prawy: 0 .. n + 1;
begin
  lewy := 1;
  prawy := n;
  jest := false;
  repeat
    m := (lewy + prawy) div 2;
    if A[m].klucz = x then
      jest := true;
    else
      if  A[m].klucz < x then
        lewy := m + 1;
      else
        prawy := m − 1;
      end if;
    end if;
  until jest or (lewy > prawy);
end; {wyszukiwanie_binarne}
```

5.5 Drzewa poszukiwań binarnych

```
type
  wierzchołek_drzewa = record
                         klucz: typ_klucza;
                         lewy, prawy: ^wierzchołek_drzewa;
                       end;
  odsyłacz = ^wierzchołek_drzewa;


procedure szukaj(x: klucz; var t: odsyłacz);
  {Szukanie klucza x w drzewie wskazanym przez t.}
  var v: odsyłacz;
begin
  v := t;
  while (v ≠ nil) cand (v^.klucz ≠ x) do
    if x < v^.klucz then
      v := v^.lewy;
    else
      v := v^.prawy;
    end if;
  end while;
  return(v); {jeśli v = nil to x nie występuje w drzewie}
end; {szukaj}

procedure wstaw(x: klucz; var t: odsyłacz);
  {Wstawianie klucza x do drzewa wskazanego przez t.}
  var v: odsyłacz;
begin
  v := t;
  while (v ≠ nil) cand (v^.klucz ≠ x) do  {szukania klucza x}
    if x < v^.klucz then
      v := v^.lewy;
    else
      v := v^.prawy;
    end if;
  end while;
  if v ≠ nil then  {x jest w drzewie}
```

```
    return;
  else
    Wstawienie x na koniec ścieżki;
  end if;
end; {wstaw}

procedure usuń(x: klucz; var t: odsyłacz);
  {Usuwanie klucza x z drzewa wskazanego przez t.}
  var v, w: odsyłacz;
begin
  v := t;
  while (v ≠ nil) cand (v^.klucz ≠ x) do  {szukania klucza x}
    if x < v^.klucz then
      v := v^.lewy;
    else
      v := v^.prawy;
    end if;
  end while;
  if v = nil then  {x nie występuje w drzewie}
    return;
  else
    if (v^.lewy = nil) or (v^.prawy = nil) then {brak lewego lub prawego poddrzewa}
      Usuń wierzchołek v^ z drzewa;
    else
      w := v^.prawy;
      while w^ nie jest końcem ścieżki w drzewie do
        w := w^.lewy; {szukanie minimalnego elementu w prawym poddrzewie}
      end while;
      Wstaw w^.klucz do v^.klucz i usuń wierzchołek w^ z drzewa;
    end if;
  end if;
end usuń;
```

Analiza wyszukiwania w drzewach binarnych

Wyznaczymy średnią liczbę porównań wymaganych dla wyszukania klucza w drzewie wyszukiwań binarnych. Zakładamy, że:

1. Dane jest n różnych kluczy o wartościach $1, 2, ..., n$.
2. Klucze pojawiają się w losowej kolejności.
3. Pierwszy klucz (a więc korzeń tworzonego drzewa) ma wartość i z prawdopodobieństwem $1/n$. Wówczas lewe poddrzewo będzie zawierać $i - 1$ wierzchołków, a prawe poddrzewo $n - i$ wierzchołków.

Oznaczmy:

- T_{i-1} — średnia liczba porównań konieczna do wyszukania dowolnego klucza w lewym poddrzewie.
- T_{n-i} — ta sama wielkość dla prawego poddrzewa.
- $T_n^{(i)}$ — średnia liczba porównań wymagana dla wyszukania klucza w drzewie o n wierzchołkach i o korzeniu i.

Wobec tego

$$T_n^{(i)} = (T_{i-1} + 1)p_{i-1} + 1 \cdot p_i + (T_{n-i} + 1)p_{n-i}$$

gdzie p_{i-1}, p_i i p_{n-i} to prawdopodobieństwa, że będziemy szukać odpowiednio klucza w lewym poddrzewie, klucza i-tego, oraz klucza w prawym poddrzewie. Zakładając, że prawdopodobieństwa szukania poszczególnych kluczy są równe, otrzymujemy:

$$p_{i-1} = \frac{i-1}{n},\ p_i = \frac{1}{n},\ p_{n-i} = \frac{n-i}{n}.$$

Wobec tego

$$\begin{aligned} T_n^{(i)} &= (T_{i-1}+1)\frac{i-1}{n} + 1\cdot\frac{1}{n} + (T_{n-i}+1)\frac{n-i}{n} = \\ &= \frac{1}{n}\left[(T_{i-1}+1)(i-1) + 1 + (T_{n-i}+1)(n-i)\right]. \end{aligned}$$

Wielkość $T_n^{(i)}$ jest ważna tylko dla drzewa o korzeniu i. Należy ją uśrednić, tj. uwzględnić przypadek, że dowolny klucz może wystąpić w korzeniu. Oznaczmy tę uśrednioną wartość przez T_n. Wówczas

$$\begin{aligned} T_n &= \frac{1}{n}\sum_{i=1}^{n} T_n^{(i)} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{n}\left[(T_{i-1}+1)(i-1) + 1 + (T_{n-i}+1)(n-i)\right] \\ &= \frac{1}{n^2}\sum_{i=1}^{n}\left[T_{i-1}(i-1) + i - 1 + 1 + T_{n-i}(n-i) + n - i\right] \\ &= 1 + \frac{1}{n^2}\sum_{i=1}^{n}\left[T_{i-1}(i-1) + T_{n-i}(n-i)\right]. \end{aligned}$$

Mamy

$$\sum_{1\leqslant i\leqslant n} T_{i-1}(i-1) = \sum_{0\leqslant i+1\leqslant n} T_{i+1-1}(i+1-1) = \sum_{0\leqslant i\leqslant n-1} T_i\cdot i$$

$$\sum_{1\leqslant i\leqslant n} T_{n-i}(n-i) = \sum_{1\leqslant n-i\leqslant n} T_{n-n+i}(n-n+i) = \sum_{0\leqslant i\leqslant n-1} T_i\cdot i$$

a więc

$$T_n = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i\cdot T_i.$$

Mnożąc obie strony równania przez n^2 otrzymujemy

$$n^2T_n = n^2 + 2\sum_{i=0}^{n-1} i\cdot T_i. \tag{1}$$

To samo równanie dla $n-1$

$$(n-1)^2T_{n-1} = (n-1)^2 + 2\sum_{i=0}^{n-2} i\cdot T_i. \tag{2}$$

Odejmując stronami (1) i (2) otrzymujemy

$$n^2T_n - (n-1)^2T_{n-1} = n^2 - (n-1)^2 + 2(n-1)T_{n-1}$$

$$n^2T_n = T_{n-1}((n-1)^2 + 2(n-1)) + (n^2 - n^2 + 2n - 1)$$

$$n^2T_n = T_{n-1}(n-1)(n+1) + (2n-1) \tag{3}$$

Zastosujemy *metodę czynnika sumacyjnego* [10, str. 41] (ang. summation factor method), która mówi, że rozwiązaniem równania rekurencyjnego o postaci

$$a_n P_n = b_n P_{n-1} + c_n$$

gdzie $a_i, b_i \neq 0$ dla $i = 1, \ldots, n$, jest

$$P_n = \frac{1}{a_n s_n}(s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k),$$

gdzie s_n to *czynnik sumacyjny* o wartości

$$s_n = \frac{a_{n-1} \cdot a_{n-2} \cdot \ldots \cdot a_1}{b_n \cdot b_{n-1} \cdot \ldots \cdot b_2}.$$

Równanie (3) ma postać

$$n^2 T_n = (n+1)(n-1)T_{n-1} + (2n-1)$$

a więc $a_n = n^2, b_n = (n+1)(n-1), c_n = 2n-1$. Wyliczmy wartość s_n:

$$s_n = \frac{a_{n-1} \cdot a_{n-2} \cdot \ldots \cdot a_1}{b_n \cdot b_{n-1} \cdot \ldots \cdot b_2} = \frac{(n-1)^2 \cdot (n-2)^2 \cdot (n-3)^2 \cdot \ldots \cdot 3^2 \cdot 2^2 \cdot 1^2}{(n+1)(n-1) \cdot n(n-2) \cdot (n-1)(n-3) \cdot \ldots \cdot 4 \cdot 2 \cdot 3 \cdot 1}$$

$$a_{n-1} \cdot a_{n-2} \cdot \ldots \cdot a_1 = (n-1)^2 \cdot (n-2)^2 \cdot \ldots \cdot 3^2 \cdot 2^2 \cdot 1^2 = (n-1)! \cdot (n-1)!$$

$$b_n \cdot \ldots \cdot b_2 = (n+1)(n-1) \cdot n(n-2) \cdot (n-1)(n-3) \cdot (n-2)(n-4) \cdot \ldots \cdot 5 \cdot 3 \cdot 4 \cdot 2 \cdot 3 \cdot 1$$

Wartość iloczynu $b_n \cdot b_{n-2} \cdot \ldots \cdot b_2$ jest $(n+1)!/2$, a wartość iloczynu $b_{n-1} b_{n-3} \cdot \ldots \cdot b_1$ jest $(n-1)!$. Wobec tego otrzymujemy

$$s_n = \frac{(n-1)! \cdot (n-1)!}{\frac{(n+1)!}{2} \cdot (n-1)!} = \frac{2}{n(n+1)}.$$

Ponieważ $T_0 = 0$, to

$$T_n = \frac{1}{s_n a_n} \sum_{k=1}^{n} s_k c_k = \frac{1}{\frac{2}{n(n+1)} n^2} \sum_{k=1}^{n} \frac{2}{k(k+1)}(2k-1) = \frac{n+1}{n} \sum_{k=1}^{n} \frac{2k-1}{k(k+1)}.$$

Rozkładając wyrażenie $\frac{2k-1}{k(k+1)}$ na ułamki proste otrzymujemy

$$\frac{2k-1}{k(k+1)} = \frac{3}{k+1} - \frac{1}{k}.$$

Wobec tego

$$\begin{aligned} T_n &= \frac{n+1}{n}\left(3\sum_{k=1}^{n} \frac{1}{k+1} - \sum_{k=1}^{n} \frac{1}{k}\right) = \frac{n+1}{n}\left(3\frac{1}{n+1} - 3 + 3\sum_{k=1}^{n} \frac{1}{k} - \sum_{k=1}^{n} \frac{1}{k}\right) = \\ &= \frac{n+1}{n}\left(\frac{3(1-n-1)}{n+1} + 2\sum_{k=1}^{n} \frac{1}{k}\right) = 2\frac{n+1}{n} H_n - 3. \end{aligned}$$

5.6 Haszowanie

5.6.1 Haszowanie otwarte

```
const a = {Odpowiednia stała.}
type słowo = array[1 .. 10] of char;
    element_listy = record
                        r: słowo;
                        nast: ^element_listy;
                    end;
    typ_listowy = ^element_listy;
    słownik = array[0 .. a − 1] of typ_listowy;
```

Funkcja haszująca

```
function h(x: słowo): 0 .. a − 1;
  var i, suma: integer;
begin
  suma := 0;
  for i := 1 to 10 do
    suma := suma + ord(x[i]);
  end for;
  return(suma mod a);
end;
```

Procedury operujące na słowniku

```
procedure ZERUJ(var A: słownik);
  var i: integer;
begin
  for i := 0 to a − 1 do
    A[i] := nil;
  end for;
end;

function JEST(x: słowo; var A: słownik): Boolean;
  var bieżący: typ_listowy;
begin
  bieżący := A[h(x)];
  while bieżący <> nil do
    if bieżący^.r = x then
      return(true);    {Słowo znaleziono.}
    else
      bieżący := bieżący^.nast;
    end;
  end while;
  return(false);    {Słowa nie znaleziono.}
end;

procedure WSTAW(x: słowo; var A: słownik);
  var kubełek: 0 .. a − 1;
      stara_głowa: typ_listowy;
begin
  if not JEST(x, A) then
    kubełek := h(x);
    stara_głowa := A[kubełek];
    new(A[kubełek]);
    A[kubełek]^.r := x;
    A[kubełek]^.nast :=stara_głowa;
```

```
  end if;
end;

procedure USUŃ(x: słowo; var A: słownik);
  var bieżący: typ_listowy;
      kubełek: 0 .. a − 1;
begin
  kubełek := h(x);
  if A[kubełek] <> nil then
    if A[kubełek]^.r = x then
      {Słowo x jest w pierwszym elemencie.}
      A[kubełek] := A[kubełek]^.nast;    {Usunięcie słowa
                                            z listy.}
    else
      {Słowo x, jeśli występuje, nie jest zawarte
        w pierwszym elemencie.}
      bieżący := A[kubełek]; {Zmienna bieżący wskazuje
                                na poprzedni element.}
      while bieżący ^.nast <> nil do
        if bieżący ^.nast^.r = x then
          {Usunięcie słowa z listy.}
          bieżący ^.nast :=bieżący ^.nast^.nast;
          return;    {Wyjście z procedury.}
        else
          {Słowa x jeszcze nie znaleziono.}
          bieżący := bieżący ^.nast;
        end if;
      end while;
    end if;
  end if;
end;
```

Rozkład kluczy w tablicy

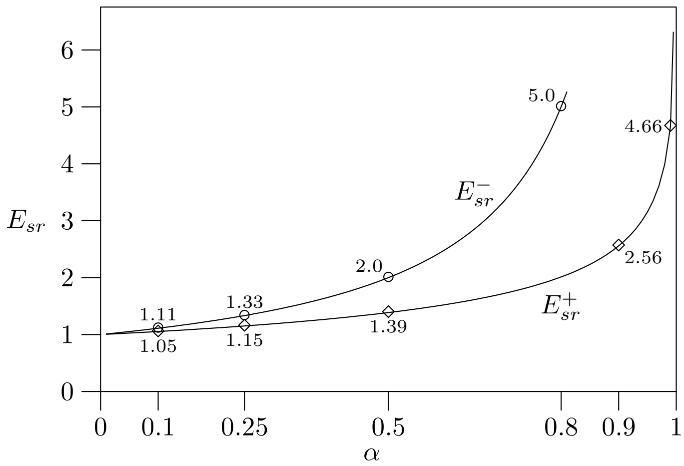
i	$A0$				
$i+1$	$A1$				
$i+2$	$A2$				
$i+3$	$A3$				
$i+4$	$A4$				
$i+5$	$A5$				
$i+6$	$A6$				
$i+7$	$A7$				
$i+8$	$A8$				
$i+9$	$A9$				
...	...				
j	$A10$				
$j+1$	$A11$	$A20$			
$j+2$	$A12$	$A21$	$A30$		
$j+3$	$A13$	$A22$	$A31$	$A40$	
$j+4$	$A14$	$A23$	$A32$	$A41$	$A50$

$j+5$	$A15$	$A24$	$A33$	$A42$	$A51$	$A60$			
$j+6$	$A16$	$A25$	$A34$	$A43$	$A52$	$A61$	$A70$		
$j+7$	$A17$	$A26$	$A35$	$A44$	$A53$	$A62$	$A71$	$A80$	
$j+8$	$A18$	$A27$	$A36$	$A45$	$A54$	$A63$	$A72$	$A81$	$A90$
$j+9$	$A19$	$A28$	$A37$	$A46$	$A55$	$A64$	$A73$	$A82$	$A91$
$j+10$		$A29$	$A38$	$A47$	$A56$	$A65$	$A74$	$A83$	$A92$
$j+11$			$A39$	$A48$	$A57$	$A66$	$A75$	$A84$	$A93$
$j+12$				$A49$	$A58$	$A67$	$A76$	$A85$	$A94$
$j+13$					$A59$	$A68$	$A77$	$A86$	$A95$
$j+14$						$A69$	$A78$	$A87$	$A96$
$j+15$							$A79$	$A88$	$A97$
$j+16$								$A89$	$A98$
$j+17$									$A99$

5.6.2 Haszowanie zamknięte

```
const puste = '          ';   {10 odstępów}
      usunięte = '**********';   {10 gwiazdek}
type słowo = packed array[1 .. 10] of char;
     słownik = array[0 .. a − 1] of słowo;

procedure ZERUJ(var A: słownik);
  var i: integer;
begin
  for i := 0 to a − 1 do
    A[i] := puste;
  end for;
end;

function szukaj(x: słowo; var A: słownik): integer;
    {Przeszukiwana jest tablica A poczynając od kubełka
      h0(x) do momentu, gdy x zostaje znalezione albo
      zostaje znaleziony kubełek pusty albo też cała tablica
      zostanie przeszukana. Wartością funkcji jest numer
      kubełka, na którym szukanie zostaje zakończone,
      wskutek jednego z wymienionych wyżej powodów.}
  var pocz, i: integer;
    {Zmienna pocz przechowuje wartość h0(x);
      zmienna i zawiera liczbę dotychczas przeszukanych
      kubełków.}
begin
  pocz := h0(x);
  i := 0;
  while (i < a) and (A[(pocz + i) mod a] <> x) and
        (A[(pocz + i) mod a] <> puste) do
    i := i + 1;
  end while;
  return((pocz + i) mod a);
end;

function szukaj_1(x: słowo; var A: słownik): integer;
    {Działanie podobne do funkcji szukaj, z tą różnicą,
      że przeszukiwanie tablicy A zostaje także zakończone
      po napotkaniu kubełka z wartością usunięte.}

function JEST(x: słowo; var A: słownik): Boolean;
```

```
begin
   if A[szukaj(x, A)] = x then
      return(true);
   else
      return(false);
   end if;
end;

procedure WSTAW(x: słowo; var A: słownik);
   var kubełek: 0 .. a − 1;
begin
   if A[szukaj(x, A)] = x then
      return;    {x jest w tablicy.}
   end if;
   kubełek := szukaj_1(x, A);
   if (A[kubełek] = puste) or (A[kubełek] = usunięte) then
      A[kubełek] := x;
   else
      błąd('błąd w procedurze WSTAW: tablica jest pełna');
   end if;
end;

procedure USUŃ(x: słowo; var A: słownik);
   var kubełek: 0 .. a − 1;
begin
   kubełek := szukaj(x, A);
   if A[kubełek] = x then
      A[kubełek] := usunięte;
   end if;
end;
```

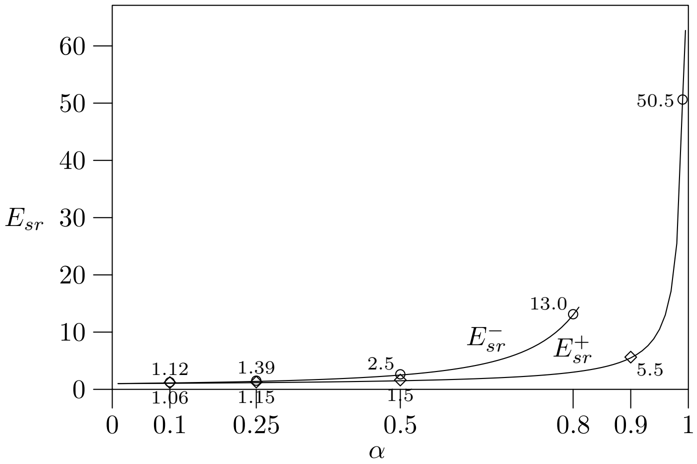


Rysunek 1: Złożoność średnia haszowania zamkniętego przy założeniu idealnych funkcji haszujących; α – współczynnik zapełnienia tablicy; $E^{-}_{sr} = 1/(1-\alpha)$, $E^{+}_{sr} = -(1/\alpha)\ln(1-\alpha)$

5.7 Minimalne, doskonałe funkcje haszujące

1. Niech dany będzie zbiór skrótów angielskich nazw miesięcy $W = \{$JUN, SEP, DEC, AUG, JAN, FEB, JUL, APR, OCT, MAY, MAR, NOV$\}$

Wartości liczbowe przyporządkowane poszczególnym literom:



Rysunek 2: Złożoność średnia haszowania zamkniętego przy założeniu liniowej metody rozwiązywania kolizji; α – współczynnik zapełnienia tablicy; $E_{sr}^- = 1/2 + 1/(2(1-\alpha)^2))$, $E_{sr}^+ = 1/2 + 1/(2(1-\alpha))$

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
4	5	2	0	0	0	3	0	0	0	0	6	0	0	5	1	0	6

S	T	U	V	W	X	Y	Z
0	6	0	6	0	0	5	0

Jeżeli nazwę każdego miesiąca uważać za ciąg trzech liter $x_1x_2x_3$, to minimalna, doskonała funkcja haszująca ma postać:

$$h^d(x_1x_2x_3) = \text{liczba przyporządkowana literze } x_2 + \text{liczba przyporządkowana literze } x_3$$

$$h^d(JUN) = 0$$
$$h^d(SEP) = 1$$
$$h^d(DEC) = 2$$
$$\ldots$$
$$h^d(NOV) = 11$$

2. Niech dany będzie zbiór słów zastrzeżonych języka Pascal W = {DO, END, ELSE, CASE, DOWNTO, GOTO, TO, OTHERWISE, TYPE, WHILE, CONST, DIV, AND, SET, OR, OF, MOD, FILE, RECORD, PACKED, NOT, THEN, PROCEDURE, WITH, REPEAT, VAR, IN, ARRAY, IF, NIL, FOR, BEGIN, UNTIL, LABEL, FUNCTION, PROGRAM}, $card(W) = 36$.

Wartości liczbowe przyporządkowane poszczególnym literom:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
11	15	1	0	0	15	3	15	13	0	0	15	15	13	0	15	0	14

S	T	U	V	W	X	Y	Z
6	6	14	10	6	0	13	0

$$h^d(x_1x_2 \ldots x_n) = \text{długość słowa } n + \text{liczba przyporządkowana literze } x_1 + \text{liczba przyporządkowana literze } x_n$$

Np.:

$$h^d(\text{DO}) = 2 + 0 + 0 = 2$$
$$h^d(\text{PROGRAM}) = 7 + 15 + 15 = 37$$

Wartości h^d dla słów DO..PROGRAM są z zakresu 2..37. Funkcja jest więc doskonała ale nie minimalna.

3. Minimalna, doskonała funkcja haszująca dla słów zastrzeżonych języka Pascal 6000

słowa:

$\textbf{var } w : \textbf{array}[1 \mathinner{..} N] \textbf{ of char}; \{W = \{w_1, w_2, \dots w_N\}\}$
$h^d(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \textbf{ mod } N$
$N = card(W)$
$h_0(w) =$ długość(w) + (Σ ord $(w[i])$, $i := 1$ do długość(w)
z krokiem 3)
$h_1(w) =$ (Σ ord $(w[i])$, $i := 1$ do długość(w) z krokiem 2) **mod** 16
$h_2(w) =$ ((Σ ord $(w[i])$, $i := 2$ do długość(w) z krokiem 2) **mod** 16) + 16
g — funkcja zadana za pomocą tablicy

Przyjęto kod EBCDIC:

ord('A') = 193, *ord*('B') = 194, ... , *ord*('I') = 201;
ord('J') = 209, *ord*('K') = 210, ... , *ord*('R') = 217;
ord('S') = 226, *ord*('T') = 227, ... , *ord*('Z') = 233;

Zbiór słów zastrzeżonych języka Pascal 6000 ($card(W) = 36$):

W = { AND, ARRAY, BEGIN, CASE, CONST, DIV, DO, DOWNTO, ELSE, END,
FILE, FOR, FUNCTION, GOTO, IF, IN, LABEL, MOD, NIL, NOT, OF,
OR, PACKED, PROCEDURE, PROGRAM, RECORD, REPEAT, SEGMENTED,
SET, THEN, TO, TYPE, UNTIL, VAR, WHILE, WITH }

i	$g(i)$	i	$g(i)$	i	$g(i)$	i	$g(i)$
0	0	9	12	18	0	27	28
1	0	10	16	19	30	28	28
2	0	11	22	20	0	29	3
3	31	12	0	21	24	30	26
4	12	13	16	22	25	31	30
5	12	14	16	23	35		
6	15	15	11	24	15		
7	33	16	0	25	27		
8	11	17	29	26	3		

$h^d(\text{NOT}) = 0$
$h^d(\text{MOD}) = 35$

6 Operacje na tekstach

tekst: $\textbf{array}[1 \mathinner{..} n]$ **of** *char*;
wzorzec: $\textbf{array}[1 \mathinner{..} m]$ **of** *char*;
{Zwykle zachodzi $1 \leqslant m \leqslant n$.}

6.1 Wyszukiwanie „naiwne”

```
function wyszukiwanie_naiwne: integer;
  var i, j: integer;
begin
  i := 1;
  j := 1;
  repeat
    if tekst[i] = wzorzec[j] then
      i := i + 1;
      j := j + 1;
    else
      i := i − j + 2;
      j := 1;
    end if;
  until (j > m) or (i > n);
  if j > m then
    return(i − m);
  else
    return(i);     {Wzorca nie znaleziono (i = n + 1).}
  end if;
end;
```

6.2 Algorytm Knutha, Morrisa i Pratta

```
function wyszukiwanie_KMP: integer;
  var i, j: integer;
begin
  i := 1;
  j := 1;
  repeat
    if (j = 0) cor (tekst[i] = wzorzec[j]) then
      i := i + 1;
      j := j + 1;
    else
      j := nast[j];
    end if;
  until (j > m) or (i > n);
  if j > m then
    return(i − m);
  else
    return(i);
  end if;
end;

procedure inicjacja_nast;
  var i, j: integer;
begin
  i := 1;
  j := 0;
  nast[1] := 0;
  repeat
    if (j = 0) or (wzorzec[i] = wzorzec[j]) then
      i := i + 1;
      j := j + 1;
      nast[i] := j;
    else
```

```
      j := nast[j];
    end if;
  until i > m;
end;
```

Algorytm wyszukiwania KMP z *wbudowanym* wzorcem 10100111

```
  i := 0;
0: i := i + 1;
1: if tekst[i] <> '1' then  goto 0; i := i + 1;
2: if tekst[i] <> '0' then  goto 1; i := i + 1;
3: if tekst[i] <> '1' then  goto 1; i := i + 1;
4: if tekst[i] <> '0' then  goto 2; i := i + 1;
5: if tekst[i] <> '0' then  goto 3; i := i + 1;
6: if tekst[i] <> '1' then  goto 1; i := i + 1;
7: if tekst[i] <> '1' then  goto 2; i := i + 1;
8: if tekst[i] <> '1' then  goto 2; i := i + 1;
  if i > n + 1 then
    return(n + 1);
  else
    return(i − 8);
  end if;
```

6.3 Wyszukiwanie niezgodnościowe

```
type znak = (' ', 'A', 'B', ... , 'Z');
var skok1: array[znak] of integer;

function wyszukiwanie_niezgodnościowe: integer;
  var i, j: integer;
begin
  i := m;
  j := m;
  repeat
    if tekst[i] = wzorzec[j] then
      i := i − 1;
      j := j − 1;
    else
      i := i + skok1[tekst[i]];
      j := m;
    end if;
  until (j < 1) or (i > n);
  if j < 1 then
    return(i + 1);
  else
    return(n + 1);
  end if;
end;

procedure inicjacja_skok1;
  var j: znak;
      k: integer;
begin
  for j := ' ' to 'Z' do
    skok1[j] := m;
  end for;
  for k := 1 to m do
```

```
      skok1[wzorzec[k]] := m − k;
   end for;
end;
```

Modyfikacja instukcji *repeat* w algorytmie wyszukiwania niezgodnościowego

```
repeat
   if tekst[i] = wzorzec[j] then
      i := i − 1;
      j := j − 1;
   else
      i := i + max(skok1[tekst[i]], skok2[j]);
      j := m;
   end if;
until (j < 1) or (i > n);
```

6.4 Algorytm Boyera i Moora

```
procedure wyszukiwanie_BM;
   var duży: integer := n + m;    {Ogólnie winno zachodzić:
                                     duży ⩾ n + m.}
       i, j, k: integer;
begin
   {m ⩽ n}
   i := m;
   loop
      repeat
         i := i + skok1[tekst[i]];
      until i > n;
      if i ⩽ duży then
         return(n + 1);    {Wzorca nie znaleziono.}
      end if;
      {Zachodzi: tekst[i − duży] = wzorzec[m].}
      i := (i − duży) − 1;
      j := m − 1;
      loop
         if j = 0 then
            return(i + 1);    {Wzorzec znaleziono.}
         end if;
         if tekst[i] = wzorzec[j] then
            i := i − 1;
            j := j − 1;
         else
            break;
         end if;
      end loop;
      k := skok1[tekst[i]];
      if k = duży then
         i := i + skok2[j];
      else
         i := i + max(k, skok2[j]);
      end if;
   end loop;
end;
```

6.5 Algorytm Karpa i Rabina

procedure *KR*;
begin
  $h_1 := h(w[1..m])$;
  $h_2 := h(t[1..m])$;
  $d := r^{m-1}$;
  for $i := 1$ **to** $n - m + 1$ **do**
    if $h_2 = h_1$ **then**
      if $w[1..m] = t[i..i + m - 1]$ **then**
        $write(i)$; {wzorzec znaleziono}
      end if;
    else
      $h_2 := ((h_2 - t[i] * d) * r + t[i + m]) \bmod q$;
    end if;
  end for;
end;

7 Sortowanie

type
  typ_rekordu = **record**
    klucz: *typ_klucza*;{W zbiorze wartości typu *typ_klucza* musi być zdefiniowana relacja liniowego porządku.}
  end;
  indeks = 0 .. *n*;
var
  A: **array**[1 .. *n*] **of** *typ_rekordu*; {Tablica podlegająca sortowaniu.}

7.1 Sortowanie przez proste wstawianie

klucze początkowe	44	55	12	42	94	18	06	67
i = 2	44	55	12	42	94	18	06	67
i = 3	12	44	55	42	94	18	06	67
i = 4	12	42	44	55	94	18	06	67
i = 5	12	42	44	55	94	18	06	67
i = 6	12	18	42	44	55	94	06	67
i = 7	06	12	18	42	44	55	94	67
i = 8	06	12	18	42	44	55	67	94

Abstrakcyjny zapis algorytmu

for $i := 2$ **to** n **do**
  {Niezmiennik: $A[1 .. i - 1]$ jest posortowane.}
  $x := A[i]$;
  Wstaw x w odpowiednim miejscu w $A[1 .. i - 1]$;

```
end for;
```

```
procedure proste_wstawianie_1;
  var
    i, j: indeks;
    x: typ_rekordu;
begin
  for i := 2 to n do
    {Niezmiennik: A[1 .. i − 1] jest posortowane.}
    x := A[i];
    j := i − 1;
    while (j > 0) cand (x.klucz < A[j].klucz) do
      A[j + 1] := A[j];
      j := j − 1;
    end while;
    A[j + 1] := x;
  end for;
end;
```

```
procedure proste_wstawianie_2;
  var
    i, j: indeks;
    x: typ_rekordu;
begin
  for i := 2 to n do
    {Niezmiennik: A[1 .. i − 1] jest posortowane.}
    x := A[i];
    A[0] := x; {Ustawienie wartownika.}
    j := i − 1;
    while  x.klucz < A[j].klucz do
      A[j + 1] := A[j];
      j := j − 1;
    end while;
    A[j + 1] := x;
  end for;
end;
```

Nieco lepsze ustawienie wartownika

```
⋮
begin

  {A[0].klucz := −∞;}
  for i := 2 to n do
    {Niezmiennik: ...}
    x := A[i];
    j := i − 1;
    while ...
⋮
```

```
procedure połówkowe_wstawianie;
  var
    i, j, l, m, p: indeks;
    x: typ_rekordu;
begin
  for i := 2 to n do
```

```
    {Niezmiennik: A[1 .. i − 1] jest posortowane.}
    x := A[i];
    l := 1;
    p := i − 1;
    while l ⩽ p do
      m := (l + p) div 2;
      if x.klucz < A[m].klucz then
        p := m − 1;
      else
        l := m + 1;
      end if;
    end while;
    for j := i − 1 downto l do
      A[j + 1] := A[j];
    end for;
    A[l] := x;
  end for;
end;
```

7.2 Algorytm Shella, czyli sortowanie za pomocą malejących przyrostów

```
type
  indeks: −h[1] .. n;
var
  A: array[−h[1] + 1 .. n] of typ_rekordu;

procedure sortowanie_Shella;
  const
    t = 4;
  var
    m: 1 .. t;
    i, j, k, w: indeks;
    h: array[1 .. t] of integer; {Tablica przyrostów.}
    x: typ_rekordu;
begin
  h[1] := 40; h[2] := 13; h[3] := 4; h[4] := 1;
  for m := 1 to 4 do
    k := h[m];
    w := −k; {Miejsce wartownika.}
    for i := k + 1 to n do
      x := A[i];
      if w = 0 then
        w := −k;
      end if;
      w := w + 1;
      A[w] := x; {Ustawienie wartownika.}
      j := i − k;
      while x.klucz < A[j].klucz do
        A[j + k] := A[j];
        j := j − k;
      end while;
      A[j + k] := x;
    end for;
  end for;
end;
```

Prosta wersja sortowania Shella

```
procedure sortowanie_Shella_1;
  var
    i, j, przyr: integer;
    x: typ_rekordu;
begin
  przyr := n div 2;
  while przyr > 0 do
    for i := przyr + 1 to n do
      j := i − przyr;
      while j > 0 do
        if A[j].klucz > A[j + przyr].klucz then
          zamiana(A[j], A[j + przyr]);
          j := j − przyr;
        else
          j := 0; {Wyjście z pętli.}
        end if;
      end while;
    end for;
    przyr := przyr div 2;
  end while;
end;
```

7.3 Sortowanie przez proste wybieranie

44	55	12	42	94	18	06	67
06	55	12	42	94	18	44	67
06	12	55	42	94	18	44	67
06	12	18	42	94	55	44	67
06	12	18	42	94	55	44	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

```
for i := 1 to n − 1 do
  {Niezmiennik: A[1 .. i − 1] jest posortowane.}
  Przypisz zmiennej k indeks rekordu o najmniejszym kluczu
    spośród rekordów A[i .. n];
  Wymień A[i] z A[k];
end for;
```

```
procedure proste_wybieranie;
  var
    i, j, k: indeks;
```

```
    x: typ_rekordu;
begin
  for i := 1 to n − 1 do
    {Niezmiennik: A[1 .. i − 1] jest posortowane.}
    k := i;
    x := A[i];
    for j := i + 1 to n do
      if A[j].klucz < x.klucz then
        k := j;
        x := A[j];
      end if;
    end for;
    A[k] := A[i];
    A[i] := x;
  end for;
end;

procedure proste_wybieranie(A[i..j]); { wersja rekursywna}
begin
  if i = j then
    return;
  else
    Znajdź indeks k elementu minimalnego
      w tablicy A[i..j];
    A[i] ⇔ A[k];
    proste_wybieranie(A[i + 1..j]);
  end if;
end;

procedure sortowanie_przez_wybieranie_2;
  var
    i, j, k, m, M, p, q: 1 .. n;
    min, Max: typ_rekordu;
begin
  i := 1;
  j := n;
  while i < j do
    min := Max := A[j]; {m i M są wskaźnikami,}
    m := M := j;        {odpowiednio, elementu }
    k := i;             {minimalnego i maksymalnego.}
    repeat
      if A[k].klucz ⩽ A[k + 1].klucz then
        p := k;
        q := k + 1;
      else
        p := k + 1;
        q := k;
      end if;
      if A[p].klucz < min.klucz then
        min := A[p];
        m := p;
      end if;
      if A[q].klucz > Max.klucz then
        Max := A[q];
        M := q;
      end if;
      k := k + 2;
```

```
    until k ⩾ j;
    if M = i then
      A[m] := A[j];
    else
      A[m] := A[i];
      A[M] := A[j];
    end if;
    A[i] := min;
    A[j] := Max;
    i := i + 1;
    j := j − 1;
  end while;
end;
```

```
for i := n downto 2 do
  {Niezmiennik: A[i + 1 .. n] jest posortowane.}
  Przypisz zmiennej k indeks rekordu o największym kluczu
    spośród rekordów A[1 .. i] ;
  Wymień A[i] z A[k] ;
end for;
```

7.4 Sortowanie przez prosta zamianę

klucze	numer przejścia						
początkowe	k=1	k=2	k=3	k=4	k=5	k=6	k=7
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

```
procedure sortowanie_bąbelkowe;
  var
    i, j: indeks;
    t: typ_rekordu;
begin
  for i := 2 to n do
    for j  := n downto i  do
      if A[j].klucz < A[j − 1].klucz then
        t := A[j];
        A[j] := A[j − 1];
```

```
          A[j − 1] := t;
        end if;
      end for;
    end for;
  end;
```

```
procedure sortowanie_mieszane;
  var
    j, k, l, p: indeks;
    t: typ_rekordu;
begin
  l := 2;
  p := n;
  k := n; {Pozycja ostatniej zamiany.}
  repeat
    for j := p downto l do
      if A[j].klucz < A[j − 1].klucz then
        t := A[j];
        A[j] := A[j − 1];
        A[j − 1] := t;
        k := j;
      end if;
    end for;
    l := k + 1;
    for j := l to p do
      if A[j].klucz < A[j − 1].klucz then
        t := A[j];
        A[j] := A[j − 1];
        A[j − 1] := t;
        k := j;
      end if;
    end for;
    p := k − 1;
  until l > p;
end;
```

7.5 Sortowanie szybkie – algorytm Quicksort

```
procedure Quicksort(i, j: integer);
1.  if A[i] do A[j] zawierają co najmniej dwa różne klucze then
2.    Niech v będzie większym z dwóch różnych kluczy najbardziej
        na lewo położonych w ciągu;
3.    Dokonać permutacji A[i], ..., A[j] tak, że dla pewnego k ∈ [i + 1, j]
        podciąg A[i], ..., A[k − 1] składa się z kluczy mniejszych od v,
        a podciąg A[k], ..., A[j] — z kluczy większych lub równych v;
4.    Quicksort(i, k − 1);
5.    Quicksort(k, j);
6.  end if;
```

```
function znajdź_klucz_osiowy(i, j: integer): integer;
  {Wartością funkcji jest indeks większego z dwóch najbardziej na lewo położonych,
    różnych kluczy ciągu A[i], ..., A[j],
    bądź 0, gdy ciąg składa się z identycznych kluczy.}
```

```
  var
    pierwszy: typ_klucza;
    k: integer;
begin
  pierwszy := A[i].klucz;
  for k := i + 1 to j do
    if A[k].klucz > pierwszy then
      return(k);
    else
      if A[k].klucz < pierwszy then
        return(i);
      end if;
    end if;
  end for;
  return(0); {Nie znaleziono różnych kluczy.}
end;


function podział(i, j: integer; klucz_osiowy: typ_klucza): integer;
  {Dokonuje się podziału ciągu A[i], ..., A[j] tak, że klucze < klucz_osiowy
    znajdą się po lewej stronie, a klucze ⩾ klucz_osiowy po prawej stronie ciągu.
    Wartością funkcji jest początek prawej grupy kluczy.}
  var
    l, p: integer;
begin
  l := i;
  p := j;
  repeat
    zamiana(A[l], A[p]);
    while A[l].klucz < klucz_osiowy do
      l := l + 1;
    end while;
    while A[p].klucz ⩾ klucz_osiowy do
      p := p − 1;
    end while;
  until l > p;
  return(l);
end;


procedure Quicksort(i, j: integer);
  {Sortowana jest tablica A[i], ..., A[j].}
  var
    klucz_osiowy: typ_klucza;
    indeks_klucza_osiowego: integer;
    k: integer; {Początek grupy kluczy ⩾ klucz_osiowy.}
begin
  indeks_klucza_osiowego := znajdź_klucz_osiowy(i, j);
  if indeks_klucza_osiowego ≠ 0 then
    klucz_osiowy := A[indeks_klucza_osiowego].klucz;
    k := podział(i, j, klucz_osiowy);
    Quicksort(i, k − 1);
    Quicksort(k, j);
  end if;
end;
```

```
procedure quicksort;
   procedure sort(l, p: indeks);
      var
         i, j: indeks;
         x, t: typ_rekordu;
   begin
      i := l;
      j := p;
      Wybierz losowo rekord x; {np. x := A[(l + p) div 2]}
      repeat
         while A[i].klucz < x.klucz do
            i := i + 1;
         end while;
         while x.klucz < A[j].klucz do
            j := j − 1;
         end while;
         if  i ⩽ j then
            t := A[i];
            A[i] := A[j];
            A[j] := t;
            i := i + 1;
            j := j − 1;
         end if;
      until i > j;
      if  l < j then
         sort(l, j);
      end if;
      if  i < p then
         sort(i, p);
      end if;
   end;
begin
   sort(1, n);
end;
```

```
procedure Quicksort_1(i, j: integer);
   {Sortowana jest tablica A[i .. j].}
   var
      klucz_osiowy: typ_klucza;
      indeks_klucza_osiowego: integer;
      k: integer; {Początek grupy kluczy ⩾ klucz_osiowy.}
      x: typ_rekordu;
      l, m: integer;
begin
   if  j − i + 1 ⩽ 9 then
      for l := i + 1 to j  do
         {Niezmiennik: A[i .. l − 1] jest posortowane.}
         x := A[l];
         m := l − 1;
         while (m > i − 1) cand (x.klucz < A[m].klucz) do
            A[m + 1] := A[m];
            m := m − 1;
         end while;
         A[m + 1] := x;
      end for;
```

```
  else
    indeks_klucza_osiowego := znajdź_klucz_osiowy(i, j);
    if indeks_klucza_osiowego ≠ 0 then
      klucz_osiowy := A[indeks_klucza_osiowego].klucz;
      k := podział(i, j, klucz_osiowy);
      Quicksort_1(i, k − 1);
      Quicksort_1(k, j);
    end if;
  end if;
end;
```

7.6 Inna wersja algorytmu Quicksort

Dana jest następująca wersja algorytmu sortowania szybkiego

```
procedure qsort(d, g: indeks);
  var
    i, s: indeks;
    t: typ_rekordu;
begin
  if d < g then
    t := A[d]; {t.klucz jest kluczem osiowym}
    s := d;
    for i := d + 1 to g do
      {Niezmiennik: A[d + 1 .. s] < t ⩽ A[s + 1 .. i − 1] }
      if A[i].klucz < t.klucz then
        s := s + 1;
        Zamień(A[s], A[i]);
      end if;
    end for;
    Zamień(A[d], A[s]);
    {Niezmiennik: A[d .. s − 1] < A[s] ⩽ A[s + 1 .. g] }
    qsort(d, s − 1);
    qsort(s + 1, g);
  end if;
end;
```

Wyznaczymy złożoność średnią tego algorytmu. W tym celu założymy, że:

1. Ciąg kluczy rekordów w tablicy A jest losową permutacją liczb całkowitych $1 \ldots n$.
2. Wystąpienie każdej permutacji jest jednakowo prawdopodobne.
3. Pierwsze wywołanie procedury ma postać $qsort(1,\ n)$.
4. Operacją dominującą jest porównanie kluczy rekordów.

Zauważmy, że dla $d \geqslant g$ wykonuje się 0 porównań. Wobec tego

$$T_{śr}(0) = T_{śr}(1) = 0.$$

Porównania kluczy rekordów wykonywane są $g-d$ razy, wobec tego przy wywołaniu $qsort(1,\ n)$ porównań wykonanych będzie $n-1$. Załóżmy, że kluczem osiowym zostaje wartość i. Wówczas na lewo od klucza osiowego trafi $i-1$ rekordów, a na prawo od klucza osiowego $n-i$ rekordów. Wobec tego wykonamy wówczas

$$n - 1 + T(i-1) + T(n-i)$$

porównań. Wartość tę musimy uśrednić po wszystkich wartościach i z przedziału $1..n$. Ponieważ każda permutacja jest jednakowo prawdopodobna, to prawdopodobieństwo, że kluczem osiowym będzie wartość i wynosi $\frac{1}{n}$. Wobec tego

$$\begin{aligned} T_{śr}(n) &= \sum_{i=1}^{n} \frac{1}{n}(n-1+T_{śr}(i-1)+T_{śr}(n-i)) = \\ &= n-1+\frac{1}{n}\sum_{1\leqslant i\leqslant n} T_{śr}(i-1)+\frac{1}{n}\sum_{1\leqslant i\leqslant n} T_{śr}(n-i) \end{aligned}$$

Ponieważ

$$\sum_{1\leqslant i\leqslant n} T_{śr}(i-1) = \sum_{1\leqslant i+1\leqslant n} T_{śr}(i+1-1) = \sum_{0\leqslant i\leqslant n-1} T_{śr}(i)$$

$$\sum_{1\leqslant i\leqslant n} T_{śr}(n-i) = \sum_{1\leqslant n-i\leqslant n} T_{śr}(n-(n-i)) = \sum_{0\leqslant i\leqslant n-1} T_{śr}(i)$$

otrzymujemy

$$T_{śr}(n) = n-1+\frac{2}{n}\sum_{i=0}^{n-1} T_{śr}(i). \tag{4}$$

Mnożąc obie strony równania (4) przez n otrzymujemy

$$nT_{śr}(n) = n^2-n+2\sum_{i=0}^{n-1} T_{śr}(i). \tag{5}$$

Równanie (5) dla $n-1$ ma postać

$$(n-1)T_{śr}(n-1) = (n-1)^2-(n-1)+2\sum_{i=0}^{n-2} T_{śr}(i). \tag{6}$$

Odejmując stronami (5) i (6) otrzymujemy

$$\begin{aligned} nT_{śr}(n)-(n-1)T_{śr}(n-1) &= n^2-n-(n-1)^2+n-1+2\sum_{i=0}^{n-1} T_{śr}(i)-2\sum_{i=0}^{n-2} T_{śr}(n) \\ nT_{śr}(n) &= (n+1)T_{śr}(n-1)+2(n-1). \end{aligned} \tag{7}$$

Do znalezienia rozwiązania (6) zastosujemy *metodę czynnika sumacyjnego* [10, str. 41]. Z (7) wynika, iż $a_n = n,\ b_n = n+1,\ c_n = 2(n-1)$. Wobec tego

$$s_n = \frac{a_{n-1}a_{n-2}\cdots a_1}{b_n b_{n-1}\cdots b_2} = \frac{(n-1)(n-2)\cdots 3\cdot 2\cdot 1}{(n+1)n(n-1)(n-2)\cdots 3} = \frac{2}{(n+1)n}$$

Ponieważ $T_{śr}(0) = 0$, otrzymujemy

$$\begin{aligned} T_{śr}(n) &= \frac{1}{s_n c_n}\cdot\sum_{k=1}^{n} s_k c_k = \\ &= \frac{1}{\frac{2}{n(n+1)}\cdot n}\cdot\sum_{k=1}^{n}\frac{2}{k(k+1)}\cdot 2(k-1) = \\ &= 2(n+1)\sum_{k=1}^{n}\frac{k-1}{k(k+1)} = \\ &= 2(n+1)\left(-\sum_{k=1}^{n}\frac{1}{k}+2\sum_{k=1}^{n}\frac{1}{k+1}\right) = \\ &= 2(n+1)\left(-\sum_{k=1}^{n}\frac{1}{k}+2\sum_{k=1}^{n}\frac{1}{k}+\frac{2}{n+1}-2\right) = \\ &= 2(n+1)\left(\sum_{k=1}^{n}\frac{1}{k}-\frac{2n}{n+1}\right) = \\ &= 2(n+1)H_n-4n. \end{aligned}$$

7.7 Czasy wykonania programów sortowania

Tabela I.

Metoda sortowania	Ciągi uporządk.		Ciągi losowe		Ciągi odwrot. uporządk.	
Wstawianie proste	12	23	366	1444	704	2836
Wstawianie połówkowe	56	125	373	1327	662	2490
Wybieranie proste	489	1907	509	1956	695	2675
Sortowanie bąbelkowe	540	2165	1026	4054	1492	5931
Sortowanie bąbelkowe ze znacznikiem	5	8	1104	4270	1645	6542
Sortowanie mieszane	5	9	961	3642	1619	6520
Sortowanie metodą Shella	58	116	127	349	157	492
Sortowanie stogowe	116	253	110	241	104	226
Sortowanie szybkie	31	69	60	146	37	79
Sortowanie przez łączenie	99	234	102	242	99	232

Tabela II. (klucze wraz ze związanymi z nimi danymi)

Metoda sortowania	Ciągi uporządk.		Ciągi losowe		Ciągi odwrot. uporządk.	
Wstawianie proste	12	46	366	1129	704	2150
Wstawianie połówkowe	46	76	373	1105	662	2070
Wybieranie proste	489	547	509	607	695	1430
Sortowanie bąbelkowe	540	610	1026	3212	1492	5599
Sortowanie bąbelkowe ze znacznikiem	5	5	1104	3237	1645	5762
Sortowanie mieszane	5	5	961	3071	1619	5757
Sortowanie metodą Shella	58	186	127	373	157	435
Sortowanie stogowe	116	264	110	246	104	227
Sortowanie szybkie	31	55	60	137	37	75
Sortowanie przez łączenie	99	196	102	195	99	187

8 Selekcja

A: **array**[1 .. *n*] **of** *typ_klucza*;

function *selekcja*(*i*, *j*, *k*: *integer*): *typ_klucza*;
  {Znajdowany jest *k*-ty najmniejszy klucz w tablicy *A*[*i* .. *j*].}
var
  klucz_osiowy: *typ_klucza*;
  indeks_klucza_osiowego: *integer*;
  m: *integer*; {Początek grupy kluczy $\geqslant$ *klucz_osiowy*.}
begin
  indeks_klucza_osiowego := *znajdź_klucz_osiowy*(*i*, *j*);
  if *indeks_klucza_osiowego* $\neq 0$ **then**
    klucz_osiowy := *A*[*indeks_klucza_osiowego*];
    m := *podział*(*i*, *j*, *klucz_osiowy*);
    if $k \leqslant m - i$ **then**
      return(*selekcja*(i, $m-1$, k));
    else
      return(*selekcja*(m, j, $k-m+i$));
    end if;

```
  else
    return(A[i]);
  end if;
end;
```

9 Sortowanie przy uwzględnieniu szczególnych własności kluczy

```
type typ_rekordu = record
                     klucz: 1 .. n;
                     ...
                   end;
 var A, B: array[1 .. n] of typ_rekordu;
```

Algorytm 1

```
for i := 1 to n do
  B[A[i].klucz] := A[i];
end for;
```

Algorytm 2

```
for i := 1 to n do
  while A[i].klucz ≠ i do
    zamiana(A[i], A[A[i].klucz]);
  end while;
end for;
```

Algorytm 3. Sortowanie przez zliczanie

```
type typ_rekordu = record
                     klucz: 1 .. m;
                     ...
                   end;
 var A, B: array[1 .. n] of typ_rekordu;
     C: array[1 .. m] of integer;

for i := 1 to m do
  C[i] := 0;
end for;
for j := 1 to n do
  C[A[j].klucz] := C[A[j].klucz] + 1; {C[k] zawiera liczbę kluczy równych k}
end for;
for i := 2 to m do {oblicznie sum prefiksowych}
  C[i] := C[i] + C[i − 1]; {C[i] zawiera liczbę kluczy mniejszych lub równych i}
end for;
for j := n downto 1 do {właściwe sortowanie}
  B[C[A[j].klucz]] := A[j];
  C[A[j].klucz] := C[A[j].klucz] − 1;
end for;
```

9.1 Sortowanie kubełkowe

```
type
  typ_klucza = 1 .. m; {lub znakowy; ogólnie typ dyskretny,
                        w zbiorze wartości którego istnieje
                        relacja porządku}
  typ_rekordu = record
                  klucz: typ_klucza;
                  ...
                end;
  element_listy = record
                    r: typ_rekordu;
                    nast: ^element_listy
                  end;
  typ_listowy = ^element_listy;
var
  A: array[1 .. n] of typ_rekordu;       {Tablica rekordów do sortowania.}
  B: array[typ_klucza] of typ_listowy;   {Tablica kubełków.
                                            W kubełku rekordy sa połączone w listę.}
  C: array[typ_klucza] of typ_listowy;   {Tablica odsyłaczy
                                            do końców list w kubełkach.}
```

```
procedure sortowanie_kubełkowe;
  var
    i: integer;
    p, v: typ_klucza;
begin
  for v := niski_klucz to wysoki_klucz do
    B[v] := nil;
    C[v] := nil;
  end for;
  for i := 1 to n do
    WSTAW(A[i], B[A[i].klucz], C[A[i].klucz]);
  end for;
  p := niski_klucz;
  while B[p] = nil do
    p := succ(p);
  end while;
  if p ≠ wysoki_klucz  then
    for v := succ(p) to wysoki_klucz do
      if B[v] ≠ nil then
        POŁĄCZ(C[p], B[v], C[v]);
      end if;
    end for;
  end if;
end;
```

```
procedure WSTAW(x: typ_rekordu; var l, m: typ_listowy);
  var
    temp: typ_listowy;
begin
  temp := l;
  new(l);
```

```
  l^.r := x;
  l^.nast := temp;
  if temp = nil then
    m := l;
  end if;
end; {WSTAW}
```

```
procedure POŁĄCZ(var lC1: typ_listowy; lB2, lC2: typ_listowy);
  {Parametry są odsyłaczami do początków i końców łączonych list.}
begin
  lC1^.nast := lB2;
  lC1 := lC2;
end; {POŁĄCZ}
```

9.2 Sortowanie pozycyjne

```
procedure sortowanie_pozycyjne;
  {Sortowana jest lista A o n rekordach, których klucze
    składają się z pól f1, f2, ..., fk
    o typach odpowiednio t1, t2, ..., tk.}
begin
  for i := k downto 1 do
    for każdej wartości v typu ti do
      Bi[v] := nil; {Opróżnianie kubełków.}
    end for;
    for każdego rekordu r na liście A do
      przesuń r z A na koniec listy kubełka Bi[v],
      gdzie v jest wartością pola fi klucza rekordu r;
    end for;
    for każdej wartości v typu ti, od najmniejszej do
      największej do
      dołącz Bi[v] na koniec listy A;
    end for;
  end for;
end;
```

Praktyczna implementacja algorytmu sortowania pozycyjnego

Sortowanie listy rekordów A według dat:

```
type
  typ_rekordu = record
                  dzień: 1 .. 31;
                  mies: (sty, ..., gru);
                  rok: 2000 .. 2099;
                  {Inne pola rekordu.}
                end;
  element_listy = record
                    r: typ_rekordu;
                    nast: ^element_listy;
                  end;
  typ_listowy = ^element_listy;
var
  A: typ_listowy;
```

```
procedure sortowanie_wg_dat;
  {Sortowana jest lista A o n rekordach, których klucze są datami.}
var
  p3, v3: 2000 .. 2099;
  p2, v2: (sty, ..., gru);
  p1, v1: 1 .. 31;
  B3, C3: array[2000 .. 2099] of typ_listowy;
  B2, C2: array [(sty, ..., gru)] of typ_listowy;
  B1, C1: array[1 .. 31] of typ_listowy;
begin
  {Sortowanie rekordów według dni.}
  for v1 := 1 to 31 do
    B1[v1] := nil;
    C1[v1] := nil;
  end for;
  while A ≠ nil do
    {Wstawienie rekordu A^.r na koniec listy kubełka
      określonego przez klucz rekordu.}
    DOŁĄCZ(A^.r, B1[A^.r.dzień], C1[A^.r.dzień]);
    A := A^.nast;
  end while;
  p1 := 1;
  while B1[p1] = nil do
    p1 := succ(p1);
  end while;
  if p1 ≠ 31 then
    for v1 := succ(p1) to 31 do
      {Przyłączenie list wszystkich kubełków do końca listy
        wskazanej przez B1[p1].}
      if B1[v1] ≠ nil then
        POŁĄCZ(C1[p1], B1[v1], C1[v1]);
      end if;
    end for;
  end if;
  A := B1[p1];
  {Dokonać podobnie sortowania kubełkowego listy A według pól A^.r.mies
    oraz A^.r.rok.}
end;
```

```
procedure DOŁĄCZ(x: typ_rekordu; var lB, lC: typ_listowy);
var
  temp: typ_listowy;
begin
  if lB = nil then
    new(lC);
    lC^.r := x;
    lC^.nast := nil;
    lB := lC;
  else
    temp := lC;
    new(lC);
    lC^.r := x;
    lC^.nast := nil;
    temp^.nast := lC;
  end if;
```

```
end; {DOŁĄCZ}
```

10 Algorytmy zachłanne

10.1 Kolorowanie zachłanne

```
procedure kolorowanie_zachłanne(G: graf;
                                var nowy_kolor: zbiór);
  {Procedura dołącza do zbioru nowy_kolor te
    wierzchołki grafu, którym można przyporządkować
    ten sam kolor.}
  var jest: Boolean;
      v, w: integer;
begin
  nowy_kolor := ∅;
  v := pierwszy niepokolorowany wierzchołek w G;
  while v <> null do
    jest := false;
    w := pierwszy wierzchołek w zbiorze nowy_kolor;
    while w <> null cand not jest do
      if istnieje krawędź pomiędzy v i w w G then
        jest := true;
      end if;
      w := następny wierzchołek w zbiorze nowy_kolor;
    end while;
    if jest = false then
      Oznacz v jako pokolorowany;
      Dołącz v do zbioru nowy_kolor;
    end if;
    v := następny niepokolorowany wierzchołek w G;
  end while;
end;
```

10.2 Problem komiwojażera

```
procedure komiwojażer(G: graf; var droga: ciąg_krawędzi);
  {Wyznacza się drogę o minimalnym koszcie w grafie G.}
  var i: integer;
      min: real;
begin
  min := ∞;
  for i := 1 to (n − 1)! do
    p := i–ta permutacja wierzchołków v1, v2, ... , v(n−1);
    {Niech d(p) jest drogą w grafie G odpowiadającą
        permutacji p jego wierzchołków.}
    if koszt(d(p)) < min then
      droga := d(p);
      min := koszt(d(p));
    end if;
  end for;
end;
```

10.3 Znajdowanie najkrótszych dróg — algorytm Dijkstry

```
procedure Dijkstra;
  {Obliczane są koszty najkrótszych dróg z wierzchołka 1
    do każdego innego wierzchołka w grafie zorientowanym.}
begin
  S := {1};
  for i := 2 to n do
    D[i] := C[1, i];      {Inicjalizacja tablicy D.}
  end for;
  for i := 1 to n − 1 do
    Wybrać wierzchołek w ze zbioru V − S taki,
       że D[w] jest minimalne;
    Dołączyć w do S;
    for każdego wierzchołka v w zbiorze V − S do
      D[v] := min(D[v], D[w] + C[w, v]);
    end for;
  end for;
end;
```

11 Algorytmy grafowe

11.1 Przeszukiwanie grafu w głąb

```
procedure wg(v);
  {Przeszukiwanie w głąb z wierzcholka v.}
begin
  znacznik[v] := odwiedzony;
  for u ∈ listy_incydencji[v] do
    if znacznik[u] = nieodwiedzony then
      {Krawędź (v, w) wstawiana jest do drzewa
        rozpinającego.}
      wg(u);
    end if;
  end for;
end;

begin
  {Przeszukiwanie grafu G = (V, E) w głąb.}
  for v ∈ V do
    znacznik[v] := nieodwiedzony;
  end for;
  for v ∈ V do
    if znacznik[v] = nieodwiedzony then
      wg(v);
    end if;
  end for;
end;
```

Zastosowanie metody przeszukiwania w głąb — wyszukiwanie składowych spójności

```
var compnum: array [1 .. |V|] of integer;      {compnum[v] jest
     numerem składowej spójności, do której należy wierzchołek v}
```

```
for v ∈ V do
  compnum[v] := 0
end for;
c := 0;

for v ∈ V do
  if compnum[v] = 0 then
    c := c + 1;
    COMP(v)
  end if;
end for;

procedure COMP(x: wierzchołek);
begin
  compnum[x] := c;
  for w ∈ adj[x] do
    if compnum[w] = 0 then
      COMP(w)
    end if;
  end for;
end;
```

Nierekursywna wersja procedury przeszukiwania w głąb

```
procedure wg1(v);
{Na początku procesu przeszukiwania P[t] = wskaźnik
  do pierwszego rekordu listy adj[t] dla każdego u.}
begin
  stos ⇐ ∅; stos ⇐ v;  odwiedź v;  nowy[v] := false;
  while stos <> ∅ do
    t := top(stos);        {odczytaj szczytowy
                             element stosu}
    while (P[t] <> nil) cand (not nowy[P[t]^.wierzch]) do
      P[t] := P[t]^.nast;       {znajdź "nowy" wierzchołek
                                  na liście adj[t]}
    end while;
    if P[t] <> nil then         {nowy wierzchołek znaleziony}
      t := P[t]^.wierzch; stos ⇐ t;
      odwiedź t; nowy[t] := false;
    else                    {wierzchołek t został wykorzystany}
      t ⇐ stos;             {zdejmij szczytowy wierzchołek ze stosu}
    end if;
  end while;
end;
```

11.2 Przeszukiwanie grafu wszerz

```
procedure wsz(v);
  {Przeszukiwanie wszerz z wierzchołka v.}
  var K: kolejka wierzchołków (typu FIFO);
begin
  znacznik[v] := odwiedzony;
  wstaw_do_kolejki(v, K);
  while not pusta_kolejka(K) do
    x := pierwszy(K);
```

```
      usuń_pierwszy_z_kolejki(K);
      for y ∈ listy_incydencji[x] do
        if znacznik[y] = nieodwiedzony then
          znacznik[y] := odwiedzony;
          wstaw_do_kolejki(y, K);
          {Krawedź (x, y) wstawiana jest do drzewa
            rozpinającego.}
        end if;
      end for;
    end while;
  end;

begin
  {Przeszukiwanie grafu G = (V, E) wszerz.}
  for v ∈ V do
    znacznik[v] := nieodwiedzony;
  end for;
  for v ∈ V do
    if znacznik[v] = nieodwiedzony then
      wsz(v);
    end if;
  end for;
end;
```

Alternatywna wersja BFS

```
kolejka ⇐ ∅; kolejka ⇐ v; nowy[v] := false;
while kolejka <> ∅ do
  p ⇐ kolejka;          {pobierz p z kolejki (tj. z usunięciem)}
  for u ∈ adj[p] do
    if nowy[u] then
      kolejka ⇐ u;
      nowy[u] := false;
    end if;
  end for;
end while;
```

Zastosowanie przeszukiwania grafu wszerz — znajdowanie najkrótszej drogi pomiędzy wierzchołkami v_0 i x w grafie

```
var pred: array[−1 .. |V|] of 1 .. |V|;    {wierzchołki poprzedzające
                                              na najkrótszej drodze}
    odl: array[1 .. |V|] of integer;      {długość najkrótszej drogi}
begin
  for v ∈ V do
    nowy[v] := true;
  end for;
  kolejka := pusta kolejka; kolejka ⇐ v0; nowy[v0] := false;
  pred[v0] := − 1;
  odl[v0] := 0;
  while nowy[x] do
    p ⇐ kolejka;
    for u ∈ adj[p] do
      if nowy[u] then
        kolejka ⇐ u; nowy[u] := false;
        pred[u] := p; odl[u] := odl[p] + 1;
```

```
      end if;
    end for;
  end while;
end;
```

11.3 Wyznaczanie cykli podstawowych (fundamentalnych)

```
begin
  for x ∈ V do
    num[x] := 0;          {inicjacja numeracji wierzchołków}
  end for;
  i := 0;        {licznik do numeracji kolejnych wierzchołków
                    odwiedzanych w głąb}
  j := 0;        {licznik cykli fundamentalnych}
  k := 0;        {wskaźnik stosu}
  for x ∈ V do
    if num[x] = 0 then
      k := k + 1;
      stos[k] := x;
      ZNAJDŹ_CYKLE(x, 0);
      k := k − 1;
    end if;
  end for;
end;
```

```
procedure ZNAJDŹ_CYKLE(v, u);        {v — wierzchołek aktualny,
                                        u — wierzchołek poprzedni}
begin
  i := i + 1;          {Odwiedzone wierzchołki zostają ponumerowane}
  num[v] := i;         {rosnąco zgodnie z licznikiem i.}
  for w ∈ adj[v] do
    if num[w] = 0 then
      k := k + 1;
      stos[k] := w;
      ZNAJDŹ_CYKLE(w, v);
      k := k − 1
    else
      if num[w] < num[v] cand w <> u then
        j := j + 1;       {licznik cykli}
        “s_j jest cyklem (w, stos[k], stos[k − 1], ... , stos[t])
          gdzie stos[t] = w, stos[k] = v”
      end if;
    end if;
  end for;
end;
```

11.4 Minimalne drzewa rozpinające

Algorytm Kruskala

T – zbiór krawędzi minimalnego drzewa rozpinającego; VS – rodzina (zbiór) rozłącznych zbiorów wierzchołków;

```
begin
```

```
  T := ∅;
  VS := ∅;
  Skonstruuj kolejkę priorytetową Q zawierającą
    wszystkie krawędzie ze zbioru E ;
  for v ∈ V do
    Dodaj {v} do VS;
  end for;
  while |VS| > 1 do
    Wybierz z kolejki Q krawędź (v, w) o najmniejszym koszcie;
    Usuń (v, w) z Q ;
    if v i w należą do różnych zbiorów W1 i W2
      należących do VS then
      Zastąp W1 i W2 w VS przez W1 ∪ W2 ;
      Dodaj (v, w) do T ;
    end if;
  end while;
end;
```

Algorytm najbliższego sąsiedztwa (Prima i Dijkstry)

```
(n = |V|)
var near: array[1 .. n] of 1 .. n;     {dla każdego wierzchołka v,
          który nie znalazł się jeszcze w drzewie, element
          near[v] zapamiętuje najbliższy mu wierzchołek drzewa}
    dist: array[1 .. n] of real;   {dist[v] zapamiętuje
          odległość v od najbliższego mu wierzchołka drzewa}

W = [w_ij]    {macierz kosztów}

begin
  Wybierz arbitralnie v0 ;
  for v ∈ V do
    dist[v] := W[v, v0];    {koszt krawędzi (v0, v)}
    near[v] := v0;
  end for;
  Vt := {v0};       {wierzchołki drzewa}
  Et := ∅;          {krawędzie drzewa}
  Wt := 0;          {sumaryczny koszt drzewa}
  while Vt <> V do
    v := wierzchołek z V − Vt o najmniejszej
         wartości dist[v];
    Vt := Vt ∪ {v};
    Et := Et ∪ {(v, near(v))};
    Wt := Wt + dist[v];
    for x ∈ V − Vt do
      if dist[x] > W[v, x] then
        dist[x] := W[v, x];
        near[x] := v
      end if;
    end for;
  end while;
end;
```

11.5 Wyznaczanie cykli Eulera

begin

```
  STOS ⇐ ∅; {opróżnianie}
  CE ⇐ ∅; {stosów}
  v := dowolny wierzchołek grafu;
  STOS ⇐ v;
  while STOS ≠ ∅ do
    v := szczyt(STOS); {v jest szczytowym elementem stosu}
    if inc[v] ≠ ∅ then {lista incydencji v jest niepusta}
      u := pierwszy wierzchołek listy inc[v];
      STOS ⇐ u;
      {usuwanie krawędzi (u, v) z grafu}
      inc[v] := inc[v] − {u};
      inc[u] := inc[u] − {v};
    else {lista incydencji v jest pusta}
      v ⇐ STOS; {przeniesienie szczytowego wierzchołka stosu STOS}
      CE ⇐ v; {do stosu CE}
    end if;
  end while;
end;
```

12 Wyszukiwanie wyczerpujące. Algorytm z powrotami

```
procedure wyszukiwanie_z_powrotami;
begin
  Wyznacz S_1; {na podstawie A_1 i ograniczeń}
  k := 1;
  while k > 0 do
    while S_k <> ∅ do
      a_k := element z S_k;
      S_k := S_k − {a_k};
      if (a_1, a_2, ... , a_k) jest rozwiązaniem then
        write((a_1, a_2, ... , a_k));
      end if;
      k := k + 1;
      Wyznacz S_k; {na podstawie A_k i ograniczeń}
    end while;
    k := k − 1; {powrót}
  end while;
end;
```

13 Przeszukiwanie heurystyczne

13.1 Przeszukiwanie wszerz

```
procedure BFS;
begin
  Q ⇐ ∅; {zerowanie kolejki}
  Q ⇐ s_0; {stan początkowy do kolejki}
  loop
    if kolejka Q pusta then
      return(porażka);
    end if;
    s ⇐ Q; {pobranie stanu z kolejki}
    Wygenerowanie następników s_j stanu s na podstawie
      operatora dla stanu s;
```

```
    Wstawienie stanów s_j do kolejki;
    if dowolny stan s_j jest stanem końcowym then
      return(sukces);
    end if;
  end loop;
end BFS;
```

13.2 Przeszukiwanie w głąb z iteracyjnym pogłębianiem

```
procedure DFS(s, głębokość, odcięcie);
  {Przeszukiwanie w głąb z odcięciem. s — bieżący stan,
    głębokość — bieżąca głębokość.}
begin
  for s_j ∈ następniki(s) do
    if s_j jest stanem końcowym then
      return(sukces); {znaleziono rozwiązanie}
    end if;
    if głębokość + 1 < odcięcie then
      DFS(s_j, głębokość + 1, odcięcie)
    end if;
  end for;
end DFS;
```

```
for odcięcie := 1 to odcięcie_max do
  DFS(s_0, 0, odcięcie);
end for;
return(porażka);
```

13.3 Przeszukiwanie według strategii „najlepszy wierzchołek jako pierwszy”

```
      OTWARTE ⇐ (s, f̂(s)); {wierzchołek początkowy na listę}
      while lista OTWARTE nie pusta do
(*)     Pobranie z listy OTWARTE wierzchołka i o minimalnej wartości f̂(i);
        ZAMKNIĘTE ⇐ (i, f̂(i));
        if i jest wierzchołkiem końcowym then
          return(sukces);
        end if;
        Rozszerzenie wierzchołka i przez wyznaczenie wszystkich jego
          następników j i obliczenie f̂(j);
        if j nie występuje na listach OTWARTE i ZAMKNIĘTE then
          OPEN ⇐ (j, f̂(j));
          Ustanowienie wskaźnika od wierzchołka j do jego poprzednika i
            (aby umożliwić odtworzenie rozwiązania końcowego);
        else
(**)      if j występuje na liście OTWARTE lub ZAMKNIĘTE then
            if nowe f̂(j) < stare f̂(j) then
              stare f̂(j) := nowe f̂(j);
              Ustanowienie wskaźnika od j do nowego poprzednika i;
            end if;
            if wierzchołek j jest na liście ZAMKNIĘTE then
              Przesunięcie (j, f̂(j)) na listę OTWARTE;
            end if;
```

```
    end if;
  end if;
end while;
return(porażka);
```

Literatura

[1] Aho, A.V., Ullman, J.D., *Projektowanie i Analiza Algorytmów Komputerowych,* PWN, Warszawa, 1983.

[2] Alagić, S., Arbib, M.A., *Projektowanie Programów Poprawnych i Dobrze Zbudowanych,* WNT, Warszawa, 1982.

[3] Banachowski, L., Diks, K., Rytter, W., *Algorytmy i Struktury Danych,* WNT, Warszawa, 1996.

[4] Banachowski, L., Kreczmar, A., *Elementy Analizy Algorytmów,* WNT, Warszawa, 1982.

[5] Banachowski, L., Kreczmar, A., Rytter, W., *Analiza Algorytmów i Struktur Danych,* WNT, Warszawa, 1987.

[6] Bentley, J., *Perełki Oprogramowania,* WNT, Warszawa, 1986.

[7] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., *Wprowadzenie do Algorytmów,* WNT, Warszawa, 2007, wyd. 8.

[8] Czech, Z.J., Deorowicz, S., Fabian, P., *Algorytmy i struktury danych. Wybrane zagadnienia*, Wydawnictwo Politechniki Śląskiej, Gliwice, 2010.

[9] Debudaj-Grabysz, A., Deorowicz, S., Widuch, J., *Algorytmy i struktury danych. Wybór zaawansowanych metod*, Wydawnictwo Politechniki Śląskiej, Gliwice, 2010.

[10] Graham, R.L., Knuth, D.E., Patashnik, O., *Matematyka Konkretna,* PWN, Warszawa, 1996.

[11] Harel, D., *Rzecz o Istocie Informatyki. Algorytmika,* WNT, Warszawa, 1992.

[12] Lipski, W., *Kombinatoryka dla Programistów,* WNT, Warszawa, 1987.

[13] Papadimitriou, C. H., *Złożoność obliczeniowa,* WNT, Warszawa, 2002.

[14] Reingold, E.M., Nievergelt, J., Deo, N., *Algorytmy Kombinatoryczne,* PWN, Warszawa, 1985.

[15] Stańczyk, P., *Algorytmika praktyczna. Nie tylko dla mistrzów,* PWN, Warszawa, 2009.

[16] Wirth, N., *Algorytmy + Struktury Danych = Programy,* WNT, Warszawa, 1980.