```
procedure Graham_Scan(Q);
begin
    znajdź w zbiorze Q punkt p0 o minimalnej współrzędnej y;
    jeśli jest więcej punktów o tej współrzędnej, wybierz punkt o najmniejszej współrzędnej x;
    pozostałe punkty Q posortuj rosnąco ze względu na współrzędną kątową w biegunowym;
    układzie współrzędnych o środku w p0 i zwrocie przeciwnym do ruchu wskazówek zegara,;
    oznacz kolejno przez p1, p2, ..., pn−1;
    wyzeruj stos S;
    Push(S, p0);
    Push(S, p1);
    Push(S, p2);
    for i := 3 to n − 1 do
        while kąt (Next_to_top(S), Top(S), pi) nie oznacza skrętu w lewo do
            Pop(S);
        end while;
        Push(S, pi);
    end for;
    return S;     {S zawiera szukaną otoczkę}
end.
```

1. Algorytm Grahama znajdowania minimalnej wypukłej otoczki

```
procedure przecinające_się_odcinki(S);
begin
    Init(T);
    sortuj końce odcinków w S wzg. współrz. x od lewej do prawej i współrz. y od dołu do góry;
    for każdy punkt p na posortowanej liście do
        if p jest lewym końcem odcinka s then
            Wstaw(T, s);
            if (Nad(T, s) istnieje i przecina s) or (Pod(T, s) istnieje i przecina s) then
                return true;
            end if;
        end if;
        if p jest prawym końcem odcinka s then
            if (Pod(T, s) istnieje i NAD(T, s) istnieje i przecina POD(T, s)) then
                return true;
            end if;
            Usuń(T, s);
        end if;
    end for;
    return false;
end.
```

1. Algorytm sprawdzania czy w zbiorze odcinków którekolwiek z nich przecinają się

```
procedure drzewo_maksimum(x);
begin
    while x.prawy_syn ≠ NIL do
        x := x.prawy_syn;
    end while;
    return x;
end.
```

1. Wyszukiwanie maksymalnego elementu w drzewie poszukiwań binarnych

```
procedure drzewo_minimum(x);
begin
    while x.lewy_syn ≠ NIL do
        x := x.lewy_syn;
    end while;
    return x;
end.
```

1. Wyszukiwanie minimalnego elementu w drzewie poszukiwań binarnych

```
procedure drzewo_następnik(x);
begin
    if x.prawy_syn ≠ NIL then
        return drzewo_minimum(x.prawy_syn);
    end if;
    y := x.rodzic;
    while y ≠ NIL and x = y.prawy_syn do
        x := y;
        y := y.rodzic;
    end while;
    return y;
end.
```

1. Wyszukiwanie następnika elementu w drzewie przeszukiwań binarnych

```
procedure drzewo_poprzednik(x);
begin
    if x.lewy_syn ≠ NIL then
        return drzewo_maximum(x.lewy_syn);
    end if;
    y := x.rodzic;
    while y ≠ NIL and x = y.lewy_syn do
        x := y;
        y := y.rodzic;
    end while;
    return y;
end.
```

1. Wyszukiwanie poprzednika elementu w drzewie przeszukiwań binarnych

```
procedure drzewo_szukaj(x, k);
begin
    while x ≠ NIL and k ≠ x.klucz do
        if k < x.klucz then
            x := x.lewy_syn;
        else
            x := x.prawy_syn;
        end if;
    end while;
    return x;
end.
```

1. Wyszukiwanie elementu o podanym kluczu w drzewie poszukiwań binarnych

```
procedure drzewo_usuń(D, e);
begin
    if e.lewy_syn = NIL or e.prawy_syn = NIL then
        y := e;
    else
        y := drzewo_następnik(e);
    end if;
    if y.lewy_syn ≠ NIL then
        x := y.lewy_syn;
    else
        x := y.prawy_syn;
    end if;
    if x ≠ NIL then
        x.rodzic := y.rodzic;
    end if;
    if y.rodzic = NIL then
        D.korzen := x;
    else
        if y = y.rodzic.lewy_syn then
            y.rodzic.lewy_syn := x;
        else
            y.rodzic.prawy_syn := x;
        end if;
    end if;
    if y ≠ e then
        z.klucz := y.klucz;
        {Kopiowanie pozostałych pól};
    end if;
    return y;
end.
```

1. Usuwanie elementu z drzewa poszukiwań binarnych

```
procedure drzewo_wstaw(D, e);
begin
    y := NIL;
    x := D.korzen;
    while x ≠ NIL do
        y := x;
        if e.klucz < x.klucz then
            x := x.lewy_syn;
        else
            x := x.prawy_syn;
        end if;
    end while;
    e.rodzic := y;
    if y = NIL then
        D.korzen := e;
    else
        if e.klucz < y.klucz then
            y.lewy_syn := e;
        else
            y.prawy_syn := e;
        end if;
    end if;
end.
```

1. Wstawianie nowego elementu do drzewa poszukiwań binarnych

```
procedure DSP_Ranga(D, x);
begin
    r := x.lewy_syn.rozmiar + 1;
    y := x;
    while y ≠ D.korzeń do
        if y = y.rodzic.prawy_syn then
            r := r + y.rodzic.lewy_syn.rozmiar + 1;
        end if;
        y := y.rodzic;
    end while;
    return r;
end.
```

1. Wyznaczanie rangi elementu w dynamicznej statystyce pozycyjnej

```
procedure DSP_Wybór(x, i);
begin
    r := x.lewy_syn.rozmiar + 1;
    if i = r then
        return x;
    else
        if i < r then
            return DSP_Wybór(x.lewy_syn, i);
        else
            return DSP_Wybór(x.prawy_syn, i − r);
        end if;
    end if;
end.
```

1. Algorytm wyszukiwania elementu o zadanej randze w dynamicznej statystyce pozycyjnej

```
procedure Fib1(n);
begin
    if n = 0 or n = 1 then
        return 1;
    end if;
    return Fib1(n−1) + Fib1(n−2);
end.
```

1. Rekurencyjny algorytm obliczania liczb Fibonacciego

```
procedure Fib2(n);
begin
    F[0] := 1;
    F[1] := 1;
    for i := 2 to n do
        F[i] := F[i−1] + F[i−2];
    end for;
    return F[n];
end.
```

1. Nierekurencyjny algorytm wyznaczania liczb Fibonacciego

```
procedure gen_komb(n, S, p: integer);
begin
    { n: nr kombinacji od 0, S: liczba elementów zbioru, p: liczba elementów podzb. }
    { v, lewy_r, prawy_r, i: zmienne cakowite }
    n := n + 1;    { ...teraz kombinacje są numerowane od 1}
    i := 0;    {nr pierwszego elementu zbioru}
    while S > 0 do
        v := \binom{S}{p};    {wartość bieżącego elementu trójkąta}
        prawy_r := \binom{S-1}{p};    { „prawy rodzic”}
        if p < S then
            lewy_r := v − prawy_r;
        else
            lewy_r := 1;    { „lewy rodzic”}
        end if;
        S := S − 1;    { „piętro” w górę}
        if n > lewy_r then    {idziemy w prawo do góry}
            n := n − lewy_r;
            writeln('element ',i, 'nie należy do podzbioru');
        else    {idziemy w lewo do góry}
            writeln('element ',i, 'należy do podzbioru');
            p := p − 1;    { zmniejszamy liczbę elementów podzbioru („k”)}
        end if;
        i := i + 1;    { nr kolejnego elementu zbioru}
    end while;
end.
```

1. Algorytm generowania k-elementowej kombinacji

```
procedure relax(u, v);
begin
    { u, v: nr wierzchołków }
    if d[u] + waga(u, v) < d[v] then
        d[v] := d[u] + waga(u, v);
        p[v] := u;
    end if;
end.
```

```
{s: źródło}
procedure dijkstra(V, E, s);
begin
    d[v] := ∞ dla v należących do V;
    d[s] := 0;
    p[v] := 0 dla v należących do V;
    S := zbiór pusty;
    Q := wszystkie wierzchołki ze zbioru V;
    while kolejka Q nie jest pusta do
        u := wierzchołek z Q o minimalnej wartości d;
        S := S + {u};
        for lista wierzchołków v sąsiadujących z u do
            relax(u, v);
        end for;
    end while;
end.
```

1. Algorytm Dijkstry wyszukiwania najkrótszych dróg w grafie z jednego wierzchołka do wszystkich pozostałych.

```
procedure graf_euler;
begin
    STOS ⇐ ∅;    {opróżnianie stosu}
    CE ⇐ ∅;    {opróżnianie stosu}
    v := dowolny wierzchołek grafu;
    STOS ⇐ v;
    while STOS ≠ ∅ do
        v := szczyt(STOS);
        if inc[v] ≠ ∅ then    {lista incydencji v nie jet pusta}
            u := usuń_pierwszy_wierzchołek_z_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 do stosu CE}
            CE ⇐ v;
        end if;
    end while;
end.
```

1. Wyznaczanie cykli Eulera w grafie

```
procedure Floyd-Warshall;
begin
    { Dane: macierz długości krawędzi D = {d_ij} }
    { Szukane: macierz długości najkrótszych dróg D = {d_ij} }
    for i := 1 to n do
        for j := 1 to n do
            if i = j then
                d[i, j] := 0;
            end if;
        end for;
    end for;
    for k := 1 to n do
        for i := 1 to n do
            if d[i, k] ≠ ∞ then
                for j := 1 to n do
                    d[i, j] := min(d[i, j], d[i, k] + d[k, j]);
                end for;
            end if;
        end for;
    end for;
end.
```

1. Algorytm Floyda-Warshalla wyszukiwania najkrótszych odległości między wszystkich parami wierzchołków w grafie.

```
procedure graf_kolorowanie_zachłanne (G : graf;  nowy_kolor : zbiór);
var  jest : boolean;
     kolor : integer;
     v, w : integer;
begin
   kolor := 0;
   while istnieją niepokolorowane wierzhołki w G do
      nowy_kolor := ∅;
      kolor := kolor + 1;
      v := pierwszy niepokolorowany wierzchołek w G;
      while v <> null do
         jest := false;
         w := pierwszy wierzchołek w zbiorze nowy_kolor;
         while w <> null 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 not jest then
            oznacz v jako pokolorowany kolorem kolor;
            dołącz v do zbioru nowy_kolor;
         end if;
         v := następny niepokolorowany wierzchołek w G;
      end while;
   end while;
end.
```

1. Kolorowanie grafu algorytmem zachłannym

```
procedure graf_kruskal;
begin
    {T – zbiór krawędzi minimalnego drzewa rozpinającego}
    {VS – rodzina (zbiór) rozłącznych zbiorów wierzchołków}
    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 and not pusta_kolejka(Q) 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.
```

1. Znajdowanie minimalnego drzewa rozpinającego metodą Kruskala

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

```
procedure graf_w_głąb;
begin
    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.
```

1. Przeszukiwanie grafu $G = (V, E)$ w głąb

```
procedure wsz(v);
var  K : kolejka wierzchoków 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 ∈ lista_incydencji[x] do
            if znacznik[y] = nieodwiedzony then
                znacznik[y] := odwiedzony;
                wstaw_do_kolejki(y, K);
                {Krawędź (x, y) jest wstawiana do drzewa rozpinającego}
            end if;
        end for;
    end while;
end.
```

```
procedure graf_wszerz;
begin
    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.
```

1. Przeszukiwanie grafu $G = (V, E)$ wszerz

```
procedure hash_al_szukaj(x);
begin
    for i := 0 to m − 1 do
        k := h(x, i);
        if A[k] = x then
            return true;
        end if;
        if A[k] jest puste then
            return false;
        end if;
    end for;
    return false;
end.
```

1. Operacja wyszukiwania elementu w tablicy mieszającej z adresowaniem liniowym jako metodą rozwiązywania kolizji

```
procedure hash_al_usuń(x);
begin
    for i := 0 to m − 1 do
        k := h(x, i);
        if A[k] = x then
            A[k] := usunięte;
            return true;
        end if;
        if A[k] jest puste then
            return false;
        end if;
    end for;
    return false;
end.
```

1. Operacja usuwania elementu z tablicy mieszającej z adresowaniem liniowym jako metodą rozwiązywania kolizji

```
procedure hash_al_usuń(x);
begin
    for i := 0 to m − 1 do
        k := h(x, i);
        if A[k] = x then
            A[k] := usunięte;
            return true;
        end if;
        if A[k] jest puste then
            return false;
        end if;
    end for;
    return false;
end.
```

1. Operacja usuwania elementu z tablicy mieszającej z adresowaniem liniowym jako metodą rozwiązywania kolizji

```
procedure hash_al_wstaw(x);
begin
    for i := 0 to m − 1 do
        k := h(x, i);
        if A[k] jest puste lub usunięte then
            A[k] := x;
            return true;
        end if;
    end for;
    return false;
end.
```

1. Operacja wstawiania elementu do tablicy mieszającej z adresowaniem liniowym jako metodą rozwiązywania kolizji

```
function hash_fun_float(f, m);
var  h : unsigned integer;
begin
    h := (0.616161 * f) mod m;
    return h;
end.
```

1. Przykładowa funkcja mieszająca dla liczb zmiennoprzecinkowych

```
function hash_fun_string1(s, m);
var  h, a : integer;
begin
    h := 0;
    a := 29;
    for i := 1 to length(s) do
        h := (a * h + s[i]) mod m;
    end for;
    return h;
end.
```

1. Przykładowa funkcja mieszająca dla ciągów znaków

```
function hash_fun_string2(s, m);
var  h, a, b : unsigned integer;
begin
    h := 0;
    a := 31415;
    b := 27183;
    for i := 1 to length(s) do
        h := (a * h + s[i]) mod m;
        a := (a * b) mod (m − 1);
    end for;
    return h;
end.
```

1. Wydajna funkcja mieszająca dla ciągów znaków

```
procedure hash_mł_szukaj(x);
begin
    k := h(x);
    if x występuje na liście zaczynającej się w A[k] then
        return true;
    else
        return false;
    end if;
end.
```

1. Operacja wyszukiwania elementu w tablicy mieszającej z łańcuchową metodą rozwiązywania kolizji

```
procedure hash_mł_usuń(x);
begin
    k := h(x);
    Usuń x z listy zaczynającej się w A[k] jeśli tam występuje;
end.
```

1. Operacja usuwania elementu z tablicy mieszającej z łańcuchową metodą rozwiązywania kolizji

```
procedure hash_mł_usuń(x);
begin
    k := h(x);
    Usuń x z listy zaczynającej się w A[k] jeśli tam występuje;
end.
```

1. Operacja usuwania elementu z tablicy mieszającej z łańcuchową metodą rozwiązywania kolizji

```
procedure hash_mł_wstaw(x);
begin
    k := h(x);
    Wstaw x na początek listy zaczynającej się w A[k];
end.
```

1. Operacja wstawiania elementu do tablicy mieszającej z metodą łańcuchową rozwiązywania kolizji

```
procedure kopiec_buduj(A, n);
begin
    heap_size(A) := n;
    for i := n div 2 downto 1 do
        kopiec_w_dół(A, i);
    end for;
end.
```

1. Algorytm budowania kopca

```
procedure kopiec_w_dół(A, i);
begin
    l := 2 * i;
    r := 2 * i + 1;
    if l ⩽ heap_size(A) and A[l] > A[i] then
        largest := l;
    else
        largest := i;
    end if;
    if r ⩽ heap_size(A) and A[r] > A[largest] then
        largest := r;
    end if;
    if largest ≠ i then
        zamiana(A[i], A[largest]);
        kopiec_w_dół(A, largest);
    end if;
end.
```

1. Algorytm przesuwania elementu w dół kopca

```
procedure kopiec_w_górę(A, x);
begin
    heap_size(A) := heap_size(A) + 1;
    i := heap_size(A);
    while i > 1 and A[i div 2] < x do
        A[i] := A[i div 2];
        i := i div 2;
    end while;
    A[i] := x;
end.
```

1. Algorytm przesuwania elementu w górę kopca

```
procedure max_frag1(A, n);
begin
    M := 0;
    for d := 1 to n do
        for g := d to n do
            S := 0;
            for i := d to g do
                s := s + A[i];    {s zawiera sumę elementów A[d..g]}
            end for;
            M := max(s, M);
        end for;
    end for;
    return M;
end.
```

1. Algorytm nr 1 wyznaczania spójnego fragmentu o największej sumie – złożoność $O(n^3)$.

```
procedure max_frag2a(A, n);
begin
    M := 0;
    for d := 1 to n do
        s := 0;
        for g := d to n do
            s := s + A[g];    {s zawiera sumę elementów A[d..g]}
            M := max(s, M);
        end for;
    end for;
    return M;
end.
```

1. Algorytm nr 2a wyznaczania spójnego fragmentu o największej sumie – złożoność $O(n^2)$.

```
procedure max_frag2b(A, n);
begin
    C[0] := 0;
    for i := 1 to n do
        C[i] := C[i − 1] + A[i];
    end for;
    M := 0;
    for d := 1 to n do
        for g := d to n do
            s := C[g] − C[d − 1];    {s zawiera sumę elementów A[d..g]}
            M := max(s, M);
        end for;
    end for;
    return M;
end.
```

1. Algorytm nr 2b wyznaczania spójnego fragmentu o największej sumie – złożoność $O(n^2)$.

```
procedure max_frag3(A, d, g);
begin
    if d > g then      {wektor zeroelementowy}
        return 0;
    end if;
    if d = g then      {wektor jednoelementowy}
        return max(0, A[d]);
    end if;
    p := (d + g) div 2;
    {{Znajdź maksymalny fragment po lewej stronie granicy}}
    s := 0; ml := 0;
    for i := p downto d do
        s := s + A[i];
        ml := max(ml, s);
    end for;
    {{Znajdź maksymalny fragment po prawej stronie granicy}}
    s := 0; mp := 0;
    for i := p + 1 to g do
        s := s + A[i];
        mp := max(mp, s);
    end for;
    {{Sprawdź, który fragment daje najlepszy wynik}}
    mo := ml + mp;
    mA := max_frag3(A, d, p);
    mB := max_frag3(A, p + 1, g);
    return max(mo, mA, mB);
end.
```

1. Algorytm nr 3 wyznaczania spójnego fragmentu o największej sumie – złożoność $O(n \log n)$.

```
procedure max_frag4(A, n);
begin
    dotąd_naj := 0;
    MK := 0;
    for i := 1 to n do
        MK := max(MK + A[i], 0);
        dotąd_naj := max(dotąd_naj, MK);
    end for;
    return dotąd_naj;
end.
```

1. Algorytm nr 4 wyznaczania spójnego fragmentu o największej sumie – złożoność $O(n)$.

```
procedure minmax1;
begin
    j := 1;
    for i := 2 to n do
        if A[i] > A[j] then
            j := i;
        end if;
    end for;
    k := 1;
    for i := 2 to n do
        if A[i] < A[k] then
            k := i;
        end if;
    end for;
end.
```

1. Algorytm znajdowania elementu minimalnego i maksymalnego (`minmax1`)

```
procedure minmax2;
begin
    j := 1; k := 1;
    for i := 2 to n do
        if A[i] > A[j] then
            j := i;
        else
            if A[i] < A[k] then
                k := i;
            end if;
        end if;
    end for;
end.
```

1. Algorytm znajdowania elementu minimalnego i maksymalnego (`minmax2`)

```
procedure minmax3;
begin
    {m jest indeksem elementu min, a M — elementu max}
    if A[2] ⩾ A[1] then
        m := 1; M := 2;
    else
        m := 2; M := 1;
    end if;
    i := 3;
    while i < n do
        k := i; l := k + 1;
        if A[k] > A[l] then
            k := l; l := i;     {k — el. mniejszy, a l — el. większy z pary i, i+1}
        end if;
        if A[m] > A[k] then
            m := k;     {poprawianie m}
        end if;
        if A[M] < A[l] then
            M := l;     {poprawianie M}
        end if;
        i := i + 2;
    end while;
    if i = n then     {jeśli n nieparzyste, to ostatnie porównanie}
        if A[n] < A[m] then
            m := n;
        else
            if A[n] > A[M] then
                M := n;
            end if;
        end if;
    end if;
end.
```

1. Algorytm znajdowania elementu minimalnego i maksymalnego (`minmax3`)

```
procedure dlugosc_NWP(X, Y);
begin
    m := length[X];
    n := length[Y];
    for i := 1 to m do
        c[i, 0] := 0;
    end for;
    for j := 1 to n do
        c[0, j] := 0;
    end for;
    for i := 1 to m do
        for j := 1 to n do
            if x_i = y_j then
                c[i, j] := c[i−1, j−1] + 1;
                b[i, j] := '↖';
            else
                if c[i−1, j] ⩾ c[i, j−1] then
                    c[i, j] := c[i−1, j];
                    b[i, j] := '↑';
                else
                    c[i, j] := c[i, j−1];
                    b[i, j] := '←';
                end if;
            end if;
        end for;
    end for;
    return c, b;
end.
```

1. Algorytm wyznaczania długości najdłuższego wspólnego podciągu (NWP)

```
procedure drukuj_NWP(b, X, i, j);
begin
    if i = 0 or j = 0 then
        return;
    end if;
    if b[i, j] = '↖' then
        drukuj_NWP(b, X, i−1, j−1);
        print x_i;
    else
        if b[i, j] = '↑' then
            drukuj_NWP(b, X, i−1, j);
        else
            drukuj_NWP(b, X, i, j−1);
        end if;
    end if;
end.
```

1. Algorytm drukowania najdłuższego wspólnego podciągu

```
procedure SilniaSystem;
begin
    { Dane wejściowe: liczba N}
    { Dane wyjściowe: ciąg cyfr d_0, d_1, ..., d_k będący zapisem N w silnia-systemie}
    d_0 := 0;     { najstarsza cyfra zawsze równa 0}
    q := N;
    k := 0;
    while q ≠ 0 do
        d_k := q mod r_k;     { kolejna cyfra }
        q := q  div r_k;
        k := k + 1;
    end while;
    if k ≠ 0 then
        k := k − 1;
    end if;
end.
```

1. Algorytm znajdowania zapisu liczby N w silnia-systemie

```
procedure sortowanie_bąbelkowe;
begin
    for i := 1 to n do
        for j := 2 to n do
            if A[j − 1] > A[j] then
                zamiana(A[j − 1], A[j]);
            end if;
        end for;
    end for;
end.
```

1. Algorytm sortowania bąbelkowego

```
procedure sortowanie_przez_kopcowanie(A, n);
begin
    kopiec_buduj(A, n);
    for i := n downto 2 do
        zamiana(A[1], A[i]);
        heap_size(A) := heap_size(A) − 1;
        kopiec_w_dół(A, 1);
    end for;
end.
```

1. Algorytm sortownia przez kopcowanie

```
procedure sortowanie_minmax(A[i..j]);
begin
    if j − i ⩾ 1 then
        {Stosując metodę porównań znajdź w tablicy A[i..j] element minimalny i zamień go}
        {z elementem A[i], następnie znajdź element maksymalny i zamień go z elementem A[j].}
        i := i + 1;
        j := j − 1;
        sortowanie_minmax(A[i..j]);
    end if;
end.
```

1. Prosty algorytm sortowania

```
procedure sortowanie_shella;
begin
    h := 1;
    while h < n/9 do      {wyszukiwanie maksymalnego h}
        h := 3 * h + 1;
    end while;
    while h > 0 do      {wykonuj h-sortowania tak długo, aż h zmaleje do 0}
        for i := h + 1 to n do
            x := A[i]; j := i;
            while j ⩾ h + 1 cand x < A[j − h] do
                A[j] := A[j − h]; j := j − h;
            end while;
            A[j] := x;
        end for;
        h := h div 3;      {zmniejszenie wartości h}
    end while;
end.
```

1. Algorytm sortowania Shella

```
procedure sortowanie_szybkie0(d, g);
begin
    if d < g then
        {wybierz klucz osiowy t};
        {przemieść klucze „wokół” klucza osiowego};
        sortowanie_szybkie0(d, s − 1);
        sortowanie_szybkie0(s + 1, g);
    end if;
end.
```

1. Ogólna wersja algorytmu sortowania szybkiego

```
procedure sortowanie_szybkie1(d, g);
begin
    if d < g then
        t := A[d];     {t jest kluczem osiowym}
        s := d;
        for i := d + 1 to g do     {przemieszczanie elementów wokół klucza osiowego}
            if A[i] < t then
                s := s + 1;
                zamiana(A[s], A[i]);
            end if;
        end for;
        zamiana(A[d], A[s]);
        sortowanie_szybkie1(d, s − 1);     {wywołania rekursywne dla obu części tablicy}
        sortowanie_szybkie1(s + 1, g);
    end if;
end.
```

1. Algorytm sortowania szybkiego

```
procedure proste_wstawianie1;
begin
    for i := 2 to n do
        x := A[i];
        j := i − 1;
        while (j > 0) cand (x < A[j]) do       {szukanie miejsca do wstawienia elementu x}
            A[j + 1] := A[j]; j := j − 1;
        end while;
        A[j + 1] := x;     {wstawienie element}
    end for;
end.
```

1. Algorytm sortowania przez proste wstawianie bez wartownika

```
procedure proste_wstawianie2;
begin
    for i := 2 to n do
        x := A[i];
        A[0] := x;      {ustawienie wartownika}
        j := i − 1;
        while x < A[j] do       {szukanie miejsca dla wstawienia elementu x}
            A[j + 1] := A[j]; j := j − 1;
        end while;
        A[j + 1] := x;      {wstawienie elementu x}
    end for;
end.
```

1. Algorytm sortowania przez proste wstawianie z wartownikiem

```
procedure proste_wybieranie;
begin
    for i := 1 to n − 1 do
        k := i; x := A[i];    {k — indeks minimalnego elementu w A[i..n], x — element minimalny}
        for j := i + 1 to n do
            if A[j] < x then    {czy bieżący element jest mniejszy niż dotychczasowe minimum}
                k := j; x := A[j];    {aktualizacja minimalnego elementu}
            end if;
        end for;
        A[k] := A[i]; A[i] := x;    {zamiana elementu A[i] ze znalezionym minimalnym}
    end for;
end.
```

1. Algorytm sortowania przez proste wybieranie

```
procedure wzorzec_KMP(T, W, n, m);
begin
    { Wyznaczanie tablicy P }
    P[0] := 0; P[1] := 0;
    t := 0;
    for j := 2 to m do
        while (t > 0) cand (W[t + 1] ≠ W[j]) do
            t := P[t];
        end while;
        if (W[t + 1] = W[j]) then
            t := t + 1;
        end if;
        P[j] := t;
    end for;
    for i := 1 to m do
        przes[i] := max(1, j − P[j]);
    end for;
    { Wyszukiwanie wzorca }
    i := 1; j := 0;
    while i ⩽ n − m + 1 do
        j := P[j];
        while j < m cand W[j + 1] = T[i + j] do
            j := j + 1;
        end while;
        if j = m then
            return i;
        end if;
        i := i + przes[j];
    end while;
    return 0;
end.
```

1. Algorytm Knutha-Morrisa-Pratta wyszukiwania wzorca w tekście

```
procedure wzorzec_naiwny(T, W, n, m);
begin
    i := 1;
    while i ⩽ n − m + 1 do
        j := 0;
        while j < m cand W[j + 1] = T[i + j] do
            j := j + 1;
        end while;
        if j = m then
            return i;
        end if;
        i := i + 1;
    end while;
    return 0;
end.
```

1. Algorytm naiwny wyszukiwania wzorca w tekście

```
procedure wiel1;
begin
    W := a[0];
    for i := 1 to n do
        p := x;
        for j := 1 to i − 1 do
            p := p ∗ x;
        end for;
        W := a[i] ∗ p + W;
    end for;
end.
```

1. Algorytm wyznaczania wartości wielomianu metodą bezpośrednią

```
procedure wiel2;
begin
    W := a[n];
    for i := n − 1 downto 0 do
        W := W ∗ x + a[i];
    end for;
end.
```

1. Algorytm wyznaczania wartości wielomianu metodą Hornera

```
procedure wyszukiwanie_z_powrotami_metoda_podziału_i_ograniczeń;
begin
    min_koszt := ∞;    {nieskończoność, bieżące minimum}
    koszt := 0;    {aktualny koszt}
    „wyznacz S_1 na podstawie A_1 i ograniczeń”;
    k := 1;
    while k > 0 do
        while (S_k ≠ ∅)  and (koszt < min_koszt) do
            a_k := element z S_k;
            S_k := S_k − {a_k};
            koszt := koszt(a_1, a_2, ..., a_k);
            if „(a_1, a_2, ..., a_k) jest rozwiązaniem”  and koszt < min_koszt then
                min_koszt := koszt((a_1, a_2, ..., a_k));
            end if;
            k := k + 1;    {następny „poziom”}
            „wyznacz S_k na podstawie A_k i ograniczeń”;
        end while;
        k := k − 1;    {powrót}
        koszt := koszt((a_1, a_2, ..., a_k));
    end while;
end.
```

1. Wyszukiwanie z powrotami – metoda podziału i ograniczeń

```
procedure szacowanie_motodą_Monte_Carlo;
begin
    {N oznacza liczbę eksperymentów do przeprowadzenia}
    średnia := 0;
    for i := 1 to N do
        iloczyn := 1;    {na iloczyn x_1 x_2 x_3 ...}
        suma := 0;    {na sumę iloczynów x_1 + x_1 x_2 + ...}
        „wyznacz S_1 na podstawie A_1 i ograniczeń”;
        k := 1;
        while S_k ≠ ∅ do
            iloczyn := iloczyn * |S_k|;
            suma := suma + iloczyn;
            a_k := losowy element z S_k;
            {nie musimy usuwać a_k z S_k, bo i tak nie będzie powrotu}
            k := k + 1;    {następny „poziom”}
            „wyznacz S_k na podstawie A_k i ograniczeń”;
        end while;
        {dotarliśmy do liścia, mamy oszacowanie liczby węzłów w tym eksperymencie}
        średnia := średnia + suma;
    end for;
    średnia := średnia / N ;
end.
```

1. Szacowanie efektywności metodą Monte Carlo

```
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
                „wypisz (a_1, a_2, ..., a_k)”;
            end if;
            k := k + 1;    {następny „poziom”}
            „wyznacz S_k na podstawie A_k i ograniczeń”;
        end while;
        k := k − 1;    {powrót}
    end while;
end.
```

1. Algorytm wyszukiwania wyczerpującego

```
procedure wysz_z_pow_rek(w: wektor; i: integer);
var  a: element_wektora; S: zbiór;
begin
    if „w jest rozwiązaniem” then
        „wypisz(w)”;
    end if;
    „wyznacz S”;
    for a ∈ S do
        wysz_z_pow_rek(w || (a), i + 1);
    end for;
end.
```

1. Rekurencyjny algorytm wyszukiwania wyczerpującego

```
procedure wybór(d, g, k);
begin
    if d = g then        {jeśli podtablica zawiera tylko jeden element to musi to być ten szukany}
        return A[d];
    end if;
    t := A[d];      {t jest kluczem osiowym}
    s := d;
    for i := d + 1 to g do       {przemieszczanie elementów wokół klucza osiowego}
        if A[i] < t then
            s := s + 1;
            zamiana(A[s], A[i]);
        end if;
    end for;
    zamiana(A[d], A[s]);
    if s = k then        {sprawdzenie czy klucz osiowy nie jest k-tym co do wielkości elementem}
        return A[s];
    else
        if s > k then        {sprawdzenie, w której części znajduje się szukany element}
            wybór(d, s − 1, k);
        else
            wybór(s + 1, g, k);
        end if;
    end if;
end.
```

1. Algorytm wyboru k-tego co do wielkości elementu w oczekiwanym czasie liniowym