Algorytmy - podstawy

Opracował: Andrzej Nowak

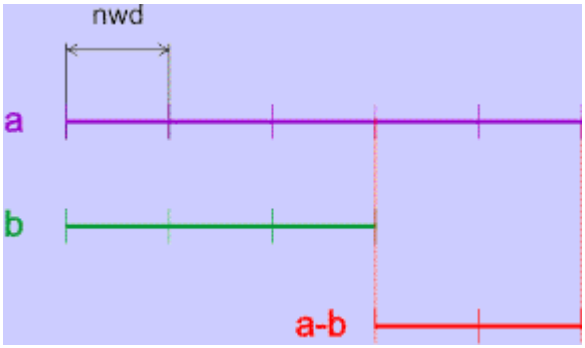
Sposoby przedstawiania algorytmu

Opis słowny

Algorytm opisujemy słowami przedstawiając kolejne operacje oraz sposób ich wykonania. Dla przykładu przedstawimy algorytm Euklidesa znajdowania NWD.

NWD - **największy wspólny dzielnik** (ang. GCD - Greatest Common Divisor) liczb *a* i *b* jest największą liczbą naturalną, która jednocześnie dzieli *a* i *b* bez reszty.

Euklides zauważył, że NWD liczb *a* i *b* dzieli również ich różnicę. Fakt ten można prosto wyjaśnić geometrycznie:



Przy obliczaniu NWD liczb *a* i *b* postępujemy zatem w sposób następujący:

Dopóki liczby *a* i *b* są różne, odejmujemy od większej mniejszą. Gdy liczby *a* i *b* staną się równe, to NWD(*a*,*b*) jest wartością dowolnej z tych liczb.

Przykład:

Mamy liczby 21 i 18. Dopóki się nie zrównają od większej odejmujemy mniejszą:

(21,18) → (21-18,18)
(3,18) → (3,18-3)
(3,15) → (3,15-3)
(3,12) → (3,12-3)
(3,9) → (3,9-3)
(3,6) → (3,6-3)
(3,3) - koniec, NWD(21,18) = 3

Lista kroków

Wykonanie algorytmu opisujemy przedstawiając kolejne kroki tego procesu. W każdym kroku opisujemy zwięźle wykonywaną operację. Istnieją pewne zasady tego opisu, które poznasz analizując kolejne algorytmy. Kroki są numerowane i wykonywane zgodnie z numerami, o ile nie zostanie nakazane inaczej. Przed listą kroków należy umieścić tzw. specyfikację danych. Jest to opis danych wejściowych i wyjściowych algorytmu. Dane wejściowe to informacja, którą musi otrzymać algorytm w celu rozwiązania problemu. Dane wyjściowe to wyniki pracy algorytmu.

Algorytm Euklidesa wyznaczania NWD dwóch liczb a i b

Wejście:

a,*b* - liczby naturalne, których NWD oblicza algorytm

Wyjście:

a lub *b* - wartość NWD pierwotnych liczb *a* i *b*.

K01:	**Czytaj** *a*,*b*	*wczytujemy dane wejściowe*
K02:	**Jeśli** *a* = *b*, **to idź do** kroku 5	*jeśli a = b, to NWD jest a lub b*
K03:	**Jeśli** *a* > *b*, **to** *a* ← *a* - *b*. **Inaczej** *b* ← *b* - *a*	*jeśli a jest różne od b, to od większej liczby odejmujemy mniejszą*
K04:	**Idź do** kroku 2	*wracamy do sprawdzania warunku w kroku 2*
K05:	**Pisz** *a*	*wypisujemy NWD*
K06:	**Zakończ**	*koniec algorytmu*

Użyte operacje:

Czytaj - powoduje odczyt danych i przypisanie ich podanym symbolom.
Pisz - powoduje wypisanie informacji
Idź do kroku n - powoduje, że następna operacja zostanie wykonana od kroku n.
Jeśli warunek, **to** operacja$_1$. **Inaczej** operacja$_2$ - jeśli warunek jest spełniony, to zostaje wykonana operacja$_1$. Inaczej wykonana zostanie operacja$_2$.
Zakończ - powoduje zakończenie wykonywania algorytmu.

Schemat blokowy

Algorytm opisywany jest w sposób graficzny za pomocą następujących symboli:

Symbol startowy, od którego rozpoczyna się wykonanie algorytmu

Symbol końca algorytmu

Strzałka określa kierunek wykonania. Prowadzi do następnego symbolu w algorytmie.

Symbol przetwarzania danych

Symbol operacji wprowadzania danych lub wyprowadzania wyników.

Symbol decyzyjny. W zależności od wyniku testu idziemy drogą TAK, jeśli test jest spełniony lub drogą NIE, jeśli test nie jest spełniony.

Schemat blokowy również wymaga specyfikacji danych wejściowych i wyjściowych.

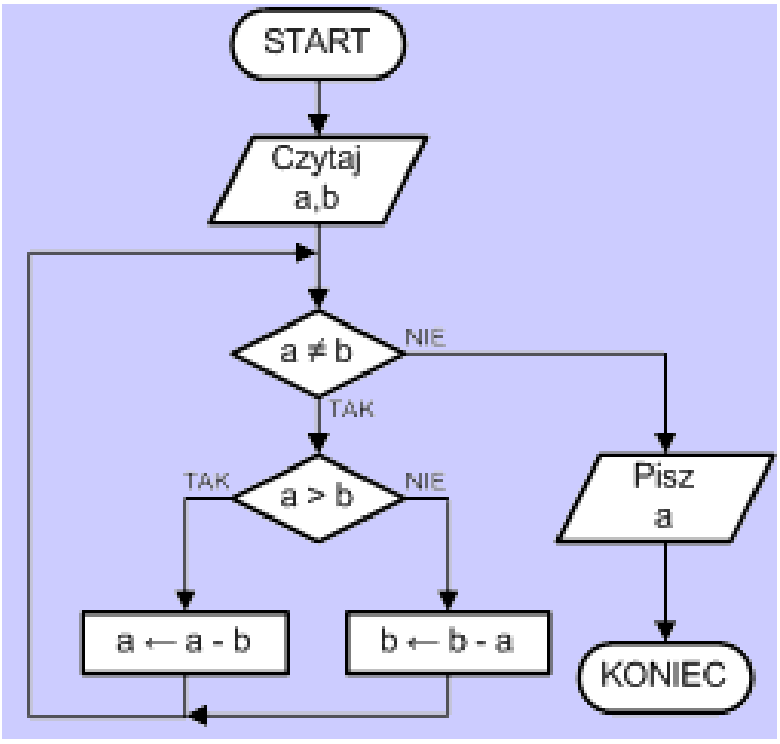
Algorytm Euklidesa wyznaczania NWD dwóch liczb a i b

Wejście:

a,*b* - liczby naturalne, których NWD oblicza algorytm

Wyjście:

a lub *b* - wartość NWD pierwotnych liczb *a* i *b*.



Program komputerowy

Program komputerowy, np. w języku C++, jest również zapisem algorytmu przy pomocy dostępnych w danym języku instrukcji i struktur danych. W przypadku programu specyfikacja danych jest umieszczona wewnątrz jego kodu.

```
#include <iostream>

using namespace std;

int main()
{
    int a,b;
    cin >> a >> b;
    while(a != b)
        if(a > b) a -= b;
        else      b -= a;
    cout << a << endl;

    return 0;
}
```

Optymalizacja algorytmu Euklidesa

Podany wyżej algorytm Euklidesa nie jest najlepszy, chociaż oczywiście działa. Rozważmy prosty przypadek:

$a = 2000000000$
$b = 2$

Aby znaleźć NWD, należy wykonać 1000000000 odejmowań! Zastanówmy się, czy nie można przyspieszyć pracy tego algorytmu.

Od liczby większej można odjąć mniejszą tyle razy, ile ta mniejsza się w niej mieści. To co zostanie z większej liczby jest resztą z dzielenia jej przez liczbę mniejszą.

Zatem po wykonaniu tej operacji z poprzednich dwóch liczb zostaje zawsze liczba mniejsza (dzielnik) oraz reszta z dzielenia większej liczby przez mniejszą:

(liczba większa) (liczba mniejsza) → (liczba mniejsza) (reszta z dzielenia liczby większej przez mniejszą)

Jeśli reszta wynosi zero, to NWD jest równy liczbie mniejszej. Sprawdźmy:

40 36 → 36 reszta z 40:36 równa 4
36 4 → 4 reszta z 36:4 równa 0
4 0 czyli NWD(40,36) = 4

Z powyższych rozważań otrzymujemy następujący algorytm:

Algorytm Euklidesa wyznaczania NWD dwóch liczb a i b

Wejście:

a,*b* - liczby naturalne, których NWD oblicza algorytm

Wyjście:

a - wartość NWD pierwotnych liczb *a* i *b*.

Dane **pomocnicze*:***

r - przechowuje wartość reszty z dzielenia *a* przez *b*

K01: **Czytaj** *a*,*b* *// wczytujemy dane wejściowe*

K02: **Jeśli** *b* = 0, **to idź do** kroku 7 *// jeśli b = 0, to a jest równe NWD*

K03: *r* ← reszta z dzielenia *a* przez *b*

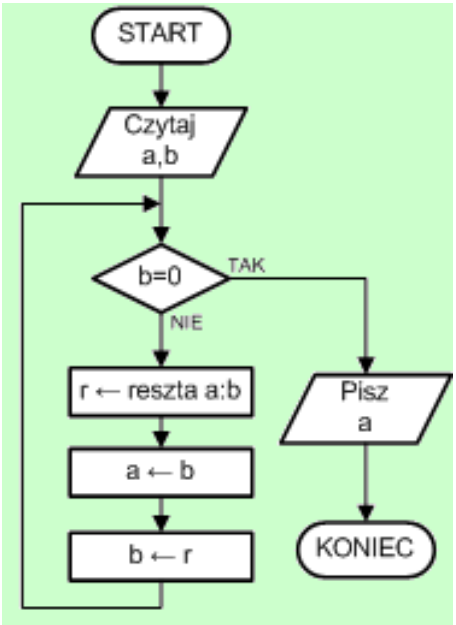
K04: *a* ← *b* *// liczbę większą zastępujemy mniejszą*

K05: *b* ← *r* *// liczbę mniejszą zastępujemy resztą*

K0 6: **Idź do** kroku 2

K07: **Pisz** *a* *// wyprowadzamy wynik*

K0 8: **Zakończ**



```cpp
#include <iostream>

using namespace std;

int main()
{
    int a,b,r;
    cin >> a >> b;

    while(b)
    {
        r = a % b;
        a = b;
        b = r;
    }
    cout << a << endl;

    return 0;
}
```

Wyszukiwanie liniowe (ang. linear search), zwane również **sekwencyjnym** (ang. sequential search) polega na przeglądaniu kolejnych elementów tablicy *T*. Jeśli przeglądany element posiada odpowiednie własności (np. jest liczbą o poszukiwanej wartości), to zwracany jest jego indeks w tablicy i algorytm się kończy. W przeciwnym razie poszukiwania są kontynuowane aż do przejrzenia wszystkich pozostałych elementów. Jeśli poszukiwany element nie zostanie znaleziony, to jest zwracany indeks niemożliwy, np. -1.

Często chcemy znaleźć wszystkie wystąpienia w zbiorze poszukiwanej wartości elementu. W takim przypadku algorytm na wejściu powinien otrzymywać dodatkowo pozycję (indeks) elementu, od którego ma rozpocząć wyszukiwanie. Pozycję tę przy kolejnym przeszukiwaniu podajemy zawsze o 1 większą od ostatnio znalezionej. Dzięki temu nowe poszukiwanie rozpocznie się tuż za poprzednio znalezionym elementem.

Zadanie znajdowania elementu maksymalnego lub minimalnego jest typowym zadaniem wyszukiwania, które rozwiązujemy przy pomocy algorytmu wyszukiwania liniowego. Za tymczasowy maksymalny (minimalny) element przyjmujemy pierwszy element zbioru. Następnie element tymczasowy porównujemy z kolejnymi elementami. Jeśli któryś z porównywanych elementów jest większy (mniejszy) od elementu tymczasowego, to za nowy tymczasowy element maksymalny (minimalny) przyjmujemy porównywany element zbioru. Gdy cały zbiór zostanie przeglądnięty, w elemencie tymczasowym otrzymamy element maksymalny (minimalny) w zbiorze.

Poniżej podajemy algorytm wyszukiwania max. Wyszukiwanie min wykonuje się identycznie, zmianie ulega tylko warunek porównujący element tymczasowy z elementem zbioru w kroku K03.

Algorytm wyszukiwania

elementu maksymalnego w zbiorze

Wejście

n – liczba elementów w tablicy *T*, $n \in N$

T – tablica zawierająca elementy do zliczania. Indeksy elementów rozpoczynają się od 0, a kończą na *n* - 1.

Wyjście:

tmax – wartość elementu maksymalnego.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów *T*. $i \in C$

tmax – tymczasowy element maksymalny

Lista kroków:

K01: *tmax* ← *T*[0] *// za tymczasowy element maksymalny bierzemy pierwszy element*

K02: **Dla** *i* = 1,2,...,*n*-1 **wykonuj** K03 *// przeglądamy następne elementy zbioru*

K03: **Jeśli** *T*[*i*] > *tmax*, **to** *tmax* ← *T*[*i*] *// jeśli natrafimy na większy od tmax, to zapamiętujemy go w tmax*

K04: **Zakończ**

Czasami zależy nam nie tylko na wartości elementu maksymalnego (minimalnego), lecz również na jego pozycji w zbiorze. W tym przypadku musimy, oprócz wartości samego elementu maksymalnego (minimalnego), zapamiętywać jego pozycję, którą na końcu zwracamy jako wynik pracy algorytmu.

Algorytm wyszukiwania

pozycji elementu maksymalnego w zbiorze

Wejście

n – liczba elementów w tablicy *T*, $n \in N$

T – tablica zawierająca elementy do zliczania. Indeksy elementów rozpoczynają się od 0, a kończą na *n* - 1.

Wyjście:

pmax – pozycja (indeks) elementu maksymalnego
tmax – wartość elementu maksymalnego.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów *T*. $i \in C$

Lista kroków:

K01: *tmax* ← *T*[0] *// za tymczasowy element maksymalny bierzemy pierwszy element*

K02: *pmax* ← 0 *// zapamiętujemy jego pozycję*

K03: **Dla** *i* = 1,2,...,*n*-1 **wykonuj** K04...K06 *// przeglądamy następne elementy zbioru*

K04: **Jeśli** *T*[*i*] ≤ *tmax,* **to następny obieg pętli** K03 *// sprawdzamy, czy trafiliśmy na element większy od tmax*

K05: *tmax* ← *T*[*i*] *// jeśli tak, to zapamiętujemy wartość elementu w tmax*

K06: *pmax* ← *i* *// oraz jego pozycję w pmax*

K07: **Zakończ**

Jednoczesne wyszukiwanie max i min

Jeśli do wyszukiwania elementu *max* i *min* zastosujemy standardowy algorytm, to otrzymamy:

Wejście

n – liczba elementów w tablicy *T*, $n \in N$

T – tablica zawierająca elementy do zliczania. Indeksy elementów rozpoczynają się od 0, a kończą na *n* - 1.

Wyjście:

tmax – wartość elementu maksymalnego.
tmin – wartość elementu minimalnego.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów *T*. $i \in C$

Numer	Operacja	Liczba wykonań
K01	*tmin* ← *T*[0]	1
K02	*tmax* ← *T*[0]	1
K03	*i* ← 1	1
K04	**Jeśli** *i* = *n*, **to idź do** K09	n
K05	**Jeśli** *T*[*i*] < *tmin*, **to** *tmin* ← *T*[*i*]	n - 1
K06	**Jeśli** *T*[*i*] > *tmax*, **to** *tmax* ← *T*[*i*]	n - 1
K07	*i* ← *i* + 1	n - 1
K08	**Idź do** K04	n - 1
K09	**Zakończ**	1

Zastanówmy się, czy można uzyskać lepszy wynik. Algorytm porównuje każdy element zbioru z *tmin* oraz z *tmax*. Tutaj tkwi nieefektywność. Obie operacje z kroków K05 i K06 są wykonywane po *n* - 1 razy, co daje w sumie 2*n* - 2 porównania w całym zbiorze.

Wyobraźmy sobie, iż ze zbioru bierzemy parę 2 elementów i porównujemy je ze sobą. Element większy nie może być minimalnym, zatem nie musimy go już porównywać z *tmin*. Wystarczy porównanie z *tmax*. To samo dotyczy elementu mniejszego – porównujemy go tylko z *tmin*. Otrzymujemy zysk – poprzednio dwa elementy wymagały wykonania 4 porównań (każdy z *tmin* i *tmax*). Teraz mamy:

Jedno porównanie dwóch elementów, które rozdzieli je na element mniejszy i element większy.
Element mniejszy porównujemy z *tmin*.
Element większy porównujemy z *tmax*.

Daje to 3 porównania zamiast 4. Ponieważ ze zbioru pobieramy po dwa kolejne elementy, to ich liczba musi być parzysta. Jeśli zbiór ma nieparzystą liczbę elementów, to po prostu dublujemy ostatni element i dostaniemy parzystą liczbę elementów.

Porównanie pary elementów dzieli zbiór na dwa podzbiory – zbiór elementów większych i mniejszych. W podzbiorze elementów większych szukamy elementu maksymalnego, a w podzbiorze elementów mniejszych szukamy minimalnego. Ponieważ ich liczebność jest o połowę mniejsza od liczebności zbioru wejściowego, to wykonamy mniej operacji porównań.

Metoda rozwiązywania problemów przez podział na mniejsze części w informatyce nosi nazwę

Dziel i zwyciężaj (ang. Divide and Conquer). Dzięki niej często udaje się zmniejszyć złożoność obliczeniową wielu algorytmów.

Algorytm jednoczesnego wyszukiwania

elementu maksymalnego i minimalnego w zbiorze

Wejście

n – liczba elementów w tablicy *T*, $n \in N$

T – tablica zawierająca elementy do zliczania. Indeksy elementów rozpoczynają się od 0, a kończą na *n* - 1. Tablica musi umożliwiać dodanie elementu, jeśli *n* jest nieparzyste.

Wyjście:

tmax – wartość elementu maksymalnego.
tmin – wartość elementu minimalnego.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów *Z*. $i \in C$

Lista kroków:

K01: **Jeśli** *n* **mod** 2 = 1, *T*[*n*] ← *T*[*n*-1] *// jeśli nieparzysta liczba elementów, dublujemy ostatni*

K02: *tmin* ← największa liczba całkowita *// inicjujemy tmin i tmax*

K03: *tmax* ← najmniejsza liczba całkowita

K04: *i* ← 0

K05: **Dopóki** *i* < *n* **wykonuj** K06...K12 *// wybieramy pary kolejnych elementów*

K06: **Jeśli** *T*[*i*] > *T*[*i*+1], **to idź do** K10 *// rozdzielamy je na element mniejszy i większy*

K07: **Jeśli** *T*[*i*] < *tmin*, **to** *tmin* ← *T*[*i*] *// T[i] - mniejszy, T[i+1] - większy*

K08: **Jeśli** *T*[*i*+1] > *tmax*, **to** *tmax* ← *T*[*i*+1]

K09: **Idź do** K12

K10: **Jeśli** *T*[*i*] > *tmax*, **to** *tmax* ← *T*[*i*] *// T[i] - większy, T[i+1] - mniejszy*

K11: **Jeśli** *T*[*i*+1] < *tmin*, **to** *tmin* ← *T*[*i*+1]

K12: *i* ← *i* + 2 *// przechodzimy do następnej pary elementów*

K13: **Zakończ**

Algorytm wyszukiwania liniowego/sekwencyjnego

Wejście

n - liczba elementów w tablicy T, $n \in N$

T - tablica zawierająca elementy do przeszukania. Indeksy elementów rozpoczynają się od 0, a kończą na n-1

p - indeks pierwszego elementu tablicy T, od którego rozpoczniemy poszukiwania. $p \in C$

k- poszukiwana wartość, czyli tzw. klucz, wg którego wyszukujemy elementy w Z

Wyjście:

Indeks elementu tablicy T o kluczu k lub -1 w przypadku nie znalezienia elementu.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów T. $i \in C$

Lista kroków:

K01: **Dla** $i = p,p+1,...,n-1$: **wykonuj** K02 *//przeglądamy kolejne elementy w tablicy*

K02: **Jeśli** $T[i] = k$, **to zakończ z wynikiem** i *//jeśli napotkamy poszukiwany element, zwracamy jego pozycję*

K03: **Zakończ z wynikiem** -1 *// jeśli elementu nie ma w tablicy, zwracamy -1*

W algorytmie wyszukiwania liniowego sprawdzamy kolejne elementy zbioru aż do napotkania jego końca lub poszukiwanego elementu. Zachodzi pytanie, czy algorytm ten można przyspieszyć. Aby na nie odpowiedzieć, zapiszmy algorytm w poniższej postaci:

Wejście

N - liczba elementów w tablicy T, $n \in N$

T - tablica zawierająca elementy do przeszukania. Indeksy elementów rozpoczynają się od 0, a kończą na n-1

k - poszukiwana wartość, czyli tzw. klucz, wg którego wyszukujemy elementy w T

Wyjście:

pozycja elementu zbioru T o kluczu k lub -1 w przypadku nie znalezienia elementu.

Zmienne pomocnicze

i - przebiega przez kolejne indeksy elementów T. $i \in C$

Numer	Operacja
1	$i \leftarrow 0$
2	**Jeśli** $i \geq n$, **to zakończ z wynikiem** -1
3	**Jeśli** $T[i] = k$, **to zakończ z wynikiem** i
4	$i \leftarrow i + 1$
5	**Idź do** 2

Sprawdzimy teraz ile operacji wykonuje ten algorytm w dwóch charakterystycznych przypadkach:

Przypadek pierwszy: poszukiwanej liczby nie ma w zbiorze. Poniżej przedstawiamy liczbę wykonań poszczególnych operacji w algorytmie:

Numer	Operacja	Liczba wykonań
1	$i \leftarrow 0$	1
2	**Jeśli** $i \geq n$, **to zakończ z wynikiem** -1	$n + 1$
3	**Jeśli** $T[i] = k$, **to zakończ z wynikiem** i	n
4	$i \leftarrow i + 1$	n
5	**Idź do** 2	n
	RAZEM:	**$4n + 2$**

Przypadek drugi: poszukiwana liczba statystycznie znajduje się w środku zbioru – czasem znajdziemy ją wcześniej, a czasem później, zatem średnio będzie w środku.

Numer	Operacja	Liczba wykonań
1	$i \leftarrow 0$	1
2	**Jeśli** $i \geq n$, **to zakończ z wynikiem** -1	$^{n}/_{2} + 1$
3	**Jeśli** $T[i] = k$, **to zakończ z wynikiem** i	$^{n}/_{2} + 1$
4	$i \leftarrow i + 1$	$^{n}/_{2}$
5	**Idź do** 2	$^{n}/_{2}$
	RAZEM:	**$2n + 3$**

Zwróć uwagę, iż w każdym obiegu pętli nasz algorytm wykonuje dwa testy – w instrukcji numer 2 i 3. Usprawnienie pracy algorytmu będzie polegało na eliminacji testu 2. Jednakże test ten jest niezbędny, aby zakończyć przeglądanie tablicy w przypadku, gdy poszukiwanego elementu nie ma w zbiorze. Skoro tak, to wstawmy poszukiwany element na koniec zbioru, wtedy test 2 stanie się zbędny, nieprawdaż. Algorytm w zmienionej postaci wygląda następująco:

Numer	Operacja
1a	$T[n] \leftarrow k$
1b	$i \leftarrow 0$
2	usunięta
3	**Jeśli** $T[i] = k$, **to idź do** 6
4	$i \leftarrow i + 1$
5	**Idź do** 3
6	**Jeśli** $i = n$, **to zakończ z wynikiem** -1
7	**Zakończ z wynikiem** i

Zmiany w stosunku do pierwotnego algorytmu są następujące:

1a – instrukcja umieszcza na końcu zbioru T poszukiwany element o wartości k. Dzięki tej operacji dostajemy gwarancję, iż zawsze znajdziemy w zbiorze element k.

2 – test osiągnięcia końca zbioru stał się zbędny, ponieważ element o wartości k zawsze znajdziemy w zbiorze.

3, 6, 7 – znalezienie elementu o wartości k wymaga sprawdzenia, czy nie jest on elementem wstawionym do zbioru w operacji 1a. Jeśli tak, to zbiór faktycznie nie zawierał poszukiwanego elementu.

Przypadek pierwszy: poszukiwanej liczby nie ma w zbiorze. Poniżej przedstawiamy liczbę wykonań poszczególnych operacji w algorytmie:

Numer	Operacja	Liczba wykonań
1a	$T[n] \leftarrow k$	1
1b	$i \leftarrow 0$	1
2		
3	**Jeśli** $T[i] = k$, **to idź do** 6	$n + 1$
4	$i \leftarrow i + 1$	n
5	**Idź do** 3	n
6	**Jeśli** $i = n$, **to zakończ z wynikiem** -1	1
7	**Zakończ z wynikiem** i	0
	RAZEM:	**$3n + 4$**

Przypadek drugi: poszukiwana liczba statystycznie znajduje się w środku zbioru – czasem znajdziemy ją wcześniej, a czasem później, zatem statystycznie będzie w środku.

Numer	Operacja	Liczba wykonań
1a	$T[n] \leftarrow k$	1
1b	$i \leftarrow 0$	1
2		
3	**Jeśli** $T[i] = k$, **to idź do** 6	$^{n}/_{2} + 1$
4	$i \leftarrow i + 1$	$^{n}/_{2}$
5	**Idź do** 3	$^{n}/_{2}$
6	**Jeśli** $i = n$, **to zakończ z wynikiem** -1	1
7	**Zakończ z wynikiem** i	1
	RAZEM:	$^{3}/_{2}n + 5$

Porównajmy teraz wyniki z pierwszej i drugiej wersji algorytmu w poniższej tabelce (wartości ułamkowe z ostatniej kolumny należy rozumieć jako wartości statystyczne).:

n	Przypadek pierwszy		Przypadek drugi	
	Algorytm podstawowy	Algorytm usprawniony	Algorytm podstawowy	Algorytm usprawniony
	4n + 2	3n + 4	2n + 3	3/2n + 5
1	6	7	5	6,5
2	10	10	7	8
3	14	13	9	9,5
4	18	16	11	11
5	22	19	13	12,5
6	26	22	15	14
...	...	...	...	...
10	42	34	23	20
100	402	304	203	155
1000	4002	3004	2003	1505
10000	40002	30004	20003	15005
...	...	...	...	...

Chociaż początkowo algorytm pierwszy wygrywa w ilości operacji, to przy wzroście liczby elementów zbioru widzimy wyraźnie, iż algorytm usprawniony wykonuje mniej operacji (począwszy od n > 3), zatem działa szybciej.

Opisana metoda wyszukiwania nosi nazwę **wyszukiwania liniowego z wartownikiem** (ang. Search with Sentinel). Wartownikiem jest dodany na końcu zbioru element równy poszukiwanemu. Dzięki niemu uzyskujemy zawsze pewność znalezienia poszukiwanego elementu w zbiorze. Jeśli jest to wartownik, to elementu poszukiwanego w zbiorze nie ma i

zwracamy pozycję -1. Jeśli nie jest to wartownik, to znaleźliśmy poszukiwany element w zbiorze i zwracamy jego pozycję *i*.

Należy podkreślić, iż wyszukiwanie z wartownikiem można stosować tylko wtedy, gdy do zbioru da się dołączyć jeszcze jeden element.

Algorytm wyszukiwania liniowego z wartownikiem

Wejście

n - liczba elementów w tablicy T, $n \in N$

T - tablica zawierająca elementy do przeszukania. Indeksy elementów rozpoczynają się od 0, a kończą na n. Ostatnia pozycja $T[n]$ będzie zajęta przez wartownika, zatem nie może być używana do innych celów

p - indeks pierwszego elementu T, od którego rozpoczniemy poszukiwania. $p \in C$

k - poszukiwana wartość, czyli tzw. klucz, wg którego wyszukujemy elementy w T

Wyjście:

Pozycja elementu zbioru T o kluczu k lub -1 w przypadku nie znalezienia elementu.

Zmienne pomocnicze

i - przebiega przez kolejne indeksy elementów T. $i \in C$

Lista kroków:

K01: $T[n] \leftarrow k$ *//na końcu zbioru umieszczamy wartownika*

K02: $i \leftarrow p$ *//przeszukiwanie rozpoczynamy od pozycji p*

K03: **Jeśli** $T[i] = k$, **to idź do** K06 *//sprawdzamy kolejne elementy zbioru*

K04: $i \leftarrow i + 1$ *//jeśli element nie znaleziony, przechodzimy do następnego elementu w zbiorze*

K05: **Idź do** K03

K06: **Jeśli** $i = n$, **to zakończ z wynikiem** -1 *//sprawdzamy, czy znaleziony element jest wartownikiem*

K07: **Zakończ z wynikiem** i *//jeśli nie, to zwracamy pozycję znalezionego elementu*

Algorytm wyszukiwania lidera w zbiorze

W *n*-elementowym zbiorze *T* znaleźć element, którego wartość występuje więcej niż $^{n}/_{2}$ razy.

Element o takich własnościach nosi nazwę **lidera**. Lidera można znaleźć przy pomocy jednego z opisanych wcześniej algorytmów wyszukiwania najczęstszej wartości w zbiorze. Po znalezieniu takiej wartości sprawdzamy, czy liczba jej wystąpień jest większa od liczby połowy elementów zbioru *T*. Jeśli tak, to mamy lidera. Jeśli nie, to zbiór *T* nie posiada lidera.

Istnieje jednakże prostszy i szybszy algorytm, który korzysta z następującego twierdzenia:

> Jeśli zbiór *T* posiada lidera, to usunięcie z niego pary elementów różnych daje zbiór z tym samym liderem.

Dowód tego twierdzenia jest bardzo prosty. Oznaczmy przez n_L liczbę elementów będących liderami. Zbiór T posiada n elementów, zatem liczba pozostałych elementów wynosi $n - n_L$. Zachodzi nierówność:

$$n_L > n - n_L$$

Przypadek 1

Ze zbioru *T* usuwamy dwa elementy, które nie są liderami. W tym przypadku n_L nie zmniejsza się, lecz zmniejsza się o 2 liczba elementów n. Otrzymamy zatem:

$$n_L > (n - 2) - n_L$$
$$n_L > (n - n_L) - 2$$

Jeśli pierwsza nierówność była prawdziwa (a była z założenia, iż n_L jest liczebnością liderów), to tym bardziej będzie prawdziwa druga nierówność. Wynika z tego, że usunięcie dwóch elementów nie będących liderami nie wpływa na występowanie lidera.

Przypadek 2

Ze zbioru T usuwamy jeden element lidera oraz jeden element nie będący liderem. Zmniejszeniu o 1 ulega zatem liczba liderów, a liczba elementów maleje o 2. Otrzymujemy:

$$n_L - 1 > (n - 2) - (n_L - 1)$$
$$n_L - 1 > n - 2 - n_L + 1$$
$$n_L - 1 > (n - n_L) - 1$$

Otrzymaliśmy nierówność wyjściową, w której od obu stron odjęto tę samą liczbę -1. Zatem nierówność jest wciąż prawdziwa, z czego wynika, że usunięcie ze zbioru jednego lidera i jeden element nie będący liderem nie wpływa na występowanie lidera.

Innych przypadków nie ma, zatem dowiedliśmy prawdziwości twierdzenia.

Z powyższego twierdzenia bezpośrednio wynikają dwie dalsze własności:

Jeśli zbiór posiada lidera, to usunięcie z niego wszystkich par elementów różnych daje zbiór zawierający tylko elementy będące liderem.

Jeśli w wyniku takiej operacji otrzymujemy jednak zbiór pusty, to lidera w zbiorze wejściowym nie było.

Zamiast faktycznie wyrzucać ze zbioru elementy różne – co jest dosyć trudne, możemy wykonać operację odwrotną. Będziemy zliczali elementy o równych wartościach – wystarczy wtedy zapamiętać wartość elementu oraz ilość jej wystąpień. Algorytm pracuje w sposób następujący:

Licznik elementów równych L ustawiamy na zero. Rozpoczynamy przeglądanie elementów zbioru od pierwszego do ostatniego.

Jeśli licznik elementów równych L jest równy 0, to kolejny element zbioru zapamiętujemy, zwiększamy licznik L o 1 i wykonujemy kolejny obieg pętli dla następnego elementu.

Jeśli licznik L nie jest równy zero, to sprawdzamy, czy bieżący element jest równy zapamiętanemu.

Jeśli tak, to mamy dwa elementy równe – zwiększamy licznik L o 1. W przeciwnym razie licznik L zmniejszamy o 1 – odpowiada to wyrzuceniu ze zbioru dwóch elementów różnych. W obu przypadkach wykonujemy kolejny obieg pętli.

Jeśli zbiór posiadał lidera, to liczba elementów równych jest większa od liczby elementów różnych. Zatem licznik L powinien mieć zawartość większą od zera.

Jeśli zawartość licznika L jest równa zero, to lidera w zbiorze nie ma. W przeciwnym razie zapamiętany element należy przeliczyć w zbiorze – jest to kandydat na lidera. Jeśli liczba jego wystąpień jest większa od liczby połowy elementów, to wytypowany element jest liderem. W przeciwnym razie zbiór T nie posiada lidera.

Algorytm posiada liniową klasę złożoności obliczeniowej O(n),

Algorytm wyszukiwania lidera w zbiorze

Wejście

n – liczba elementów w tablicy T, $n \in N$

T – tablica do wyszukania najczęstszego elementu Elementy są liczbami całkowitymi. Indeksy od 0 do n - 1..

Wyjście:

Element będący liderem, liczba jego wystąpień w T lub informacja o braku lidera.

Zmienne pomocnicze

i – przebiega przez kolejne indeksy elementów T. $i \in C$

L – licznik wystąpień wartości równych, $L \in C$

W – wartość lidera

Lista kroków:

K01: $L \leftarrow 0$ *// licznik wartości równych*

K02: **Dla** i = 0,1,...,n-1 **wykonuj** K03...K07 *// przeglądamy kolejne elementy*

K03: **Jeśli** $L > 0$, **to idź do** K07 *// sprawdzamy, czy licznik równy 0*

K04: $W \leftarrow T[i]$ *// jeśli tak, zapamiętujemy bieżący element*

K05: $L \leftarrow 1$ *// zaliczamy ten element*

K06: **Następny obieg pętli** K02

K07: **Jeśli** $W = T[i]$, to $L \leftarrow L + 1$ *// elementy równe zliczamy*
inaczej $L \leftarrow L - 1$ *// elementy różne wyrzucamy*

K08: **Jeśli** $L = 0$, **to idź do** K15 *// sprawdzamy, czy jest kandydat na lidera*

K09: $L \leftarrow 0$ *// jeśli tak, to sprawdzamy, czy jest liderem*

K10: **Dla** $i = 0,1,...,n-1$ **wykonuj** K11 */ zliczamy jego wystąpienia w zbiorze*

K11: **Jeśli** $T[i] = W$, **to** $L \leftarrow L + 1$

K12: **Jeśli** $L \leq [n:2]$, **to idź do** K15 *// lider?*

K13 **Pisz** W,L *// wyprowadzamy lidera oraz liczbę jego wystąpień*

K14: **Zakończ**

K15: **Pisz** "BRAK LIDERA"

K16: **Zakończ**