Znajdowanie elementu w zbiorze nieuporządkowanym.

Algorytm	Ogólny opis problemu	Specyfikacja	Opis słowny
Znajdowanie elementu w zbiorze nieuporządkowanym [W n-elementowym zbiorze Z wyszukać element posiadający pożądane własności.]	Wyszukiwanie liniowe (ang. linear search), zwane również sekwencyjnym (ang. sequential search) polega na przeglądaniu kolejnych elementów zbioru Z. Jeśli przeglądany element posiada odpowiednie własności (np. jest liczbą o poszukiwanej wartości), to zwracamy jego pozycję w zbiorze i kończymy. W przeciwnym razie kontynuujemy poszukiwania aż do przejrzenia wszystkich pozostałych elementów zbioru Z. W przypadku pesymistycznym, gdy poszukiwanego elementu nie ma w zbiorze lub też znajduje się on na samym końcu zbioru, algorytm musi wykonać przynajmniej n obiegów pętli sprawdzającej poszczególne elementy. Wynika z tego, iż pesymistyczna klasa złożoności obliczeniowej jest równa O(n), czyli jest liniowa - stąd pochodzi nazwa metody wyszukującej. 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.	**Dane [Wejście]:** n - liczba elementów w tablicy Z[ ], n $\in$N Z[ ] - tablica zawierająca elementy do przeszukania. Indeksy elementów rozpoczynają się od 0, a kończą na n-1 p - indeks pierwszego elementu Z[ ], od którego rozpoczniemy poszukiwania. p $\in$C k - poszukiwana wartość, czyli tzw. klucz, według którego wyszukujemy elementy w Z[ ] **Wynik [Wyjście]:** Pozycja elementu zbioru Z[ ] o kluczu k lub -1 w przypadku nie znalezienia elementu. **Zmienne pomocnicze:** i - przebiega przez kolejne indeksy elementów Z[ ], i $\in$C	Lista kroków: Krok1: Dla i = p,p+1,...,n-1: wykonuj Krok2 [przeglądamy kolejne elementy w zbiorze]. Krok2: Jeśli Z[i] = k, to zakończ zwracając i [jeśli napotkamy poszukiwany element, zwracamy jego pozycję]. Krok3: Zakończ zwracając -1 [jeśli elementu nie ma w tablicy, zwracamy -1]

Free Pascal	Dev-C++

program prg;
 	
type Tint = array of integer;
 	
function Szukaj(var T : Tint; n,k,p : integer) : integer;
var i : integer;
begin
  Szukaj := -1;
  for i := p to n - 1 do
  if T[i] = k then
  begin
  Szukaj := i; break;
  end
end;
 	
var
  Z : Tint;
  n,k,i: integer;
 	
begin
  readln(n);
  SetLength(Z,n);
  for i := 0 to n - 1 do readln(Z[i]);
  readln(k);
  writeln;
  writeln(Szukaj(Z,n,k,0));
  writeln;
end.
	#include <iostream>
	
	using namespace std;
	
	int Szukaj(int T[], int n, int k, int p)
	{
	  for(int i = p; i < n; i++)
	  if(T[i] == k) return i;
	  return -1;
	}
	
	int main()
	{
	  int * Z,n,k,i;
	
	  cin >> n;
	  Z = new int [n];
	  for(i = 0; i < n; i++) cin >> Z[i];
	  cin >> k;
	  cout << endl << Szukaj(Z,n,k,0) << endl << endl;
	  delete [] Z;
	  return 0;
	}

Znajdowanie elementu w zbiorze nieuporządkowanym oraz w zbiorze uporządkowanym.

Algorytm	Ogólny opis problemu	Specyfikacja	Opis słowny
Znajdowanie elementu w zbiorze uporządkowanym. [W posortowanym rosnąco zbiorze Z wyszukać element o wartości (kluczu) k]	Postąpimy w sposób następujący. Wyznaczymy element środkowy zbioru. Sprawdzimy, czy jest on poszukiwanym elementem. Jeśli tak, to element został znaleziony i możemy zakończyć poszukiwania. Jeśli nie, to poszukiwany element jest albo mniejszy od elementu środkowego, albo większy. Ponieważ zbiór jest uporządkowany, to elementy mniejsze od środkowego będą leżały w pierwszej połówce zbioru, a elementy większe w drugiej połówce. Zatem w następnym obiegu zbiór możemy zredukować do pierwszej lub drugiej połówki - jest w nich o połowę mniej elementów. Mając nowy zbiór postępujemy w sposób identyczny - znów wyznaczamy element środkowy, sprawdzamy czy jest poszukiwanym elementem. Jeśli nie, to zbiór dzielimy znów na dwie połowy - elementy mniejsze od środkowego i elementy większe od środkowego. Poszukiwania kontynuujemy w odpowiedniej połówce zbioru aż znajdziemy poszukiwany element lub do chwili, gdy po podziale połówka zbioru nie zawiera dalszych elementów. Ponieważ w każdym obiegu pętli algorytm redukuje liczbę elementów o połowę, to jego klasa złożoności obliczeniowej jest równa O(log n). Jest to bardzo korzystna złożoność. Na przykład w algorytmie wyszukiwania liniowego przy 1000000 elementów należało wykonać aż 1000000 porównań, aby stwierdzić, iż elementu poszukiwanego nie ma w zbiorze. W naszym algorytmie wystarczy 20 porównań. Oczywiście algorytm wyszukiwania liniowego może być zastosowany dla dowolnego zbioru. Nasz algorytm można stosować tylko i wyłącznie w zbiorze uporządkowanym. Ze względu na podział zbioru na kolejne połówki, ćwierci itd. algorytm nosi nazwę wyszukiwania binarnego (ang. binary search).	**Wejście:** i_p - indeks pierwszego elementu w tablicy Z[ ], $i_p \in C$ i_k - indeks ostatniego elementu w tablicy Z[ ], $i_k \in C$ Z[ ] - tablica do wyszukania elementu. Indeksy od i_p do i_k k - wartość poszukiwanego elementu - tzw. klucz **Wyjście:** p = indeks elementu o kluczu k lub p = -1, jeśli nie odnalezienia elementu o takim kluczu. **Zmienne pomocnicze:** i_{sr} - indeks elementu środkowego. $i_{sr} \in C$	**Lista kroków:** K01: $p \leftarrow -1$ [zakładamy, iż k nie występuje w zbiorze] K02: **Dopóki** $i_p \le i_k$ **wykonuj** K02...K07 [w pętli poszukujemy elementu o wartości k] K03: $i_{sr} = \square \; i_p + i_k \; \square$ [wyznaczamy element środkowy] K04: **Jeśli** $k \neq Z[i_{sr}]$, **to idź do** K07 K05: $p \leftarrow i_{sr}$ [element znaleziony, kończymy] K06: **Idź do** K11 K07: **Jeśli** $k < Z[i_{sr}]$, **to idź do** K09 [wybieramy odpowiednią połówkę zbioru na dalsze poszukiwania] K08: $i_p \leftarrow i_{sr} + 1$ [$k > Z[i_{sr}] \rightarrow$ druga połówka] K09: $i_k \leftarrow i_{sr} - 1$ [$k < Z[i_{sr}] \rightarrow$ pierwsza połówka] K10: **Następny obieg pętli** K02 K11: **Zakończ zwracając** p

Free Pascal	Dev-C++

program prg;
const N = 100;
 	
var
  Z : array[0..N - 1] of integer;
  i,ip,ik,isr,k,L,p : integer;
 	
begin
  randomize;
 	
// wypełniamy tablicę Z[]
  Z[0] := random(5);
  for i := 1 to N - 1 do Z[i] := Z[i - 1] + random(5);
 	
// generujemy klucz k
  k := Z[0] + random(Z[N - 1] - Z[0] + 1);
 	
// poszukujemy binarnie elementu k
  L := 0; p := -1; ip := 0; ik := N - 1;
  while ip <= ik do
  begin
  inc(L);
  isr := (ip + ik) shr 1;
  if Z[isr] = k then
  begin
  p := isr; break;
  end
  else if k < Z[isr] then
  ik := isr - 1
  else
  ip := isr + 1;
  end;
 	
// wyświetlamy wyniki
  write(k,' : ');
  if p >= 0 then write(p) else write('BRAK');
  writeln(' : obiegi = ',L);
  writeln;
  for i := 0 to N - 1 do
  begin
  write(Z[i]:3);
  if p = i then write('<') else write(' ');
  end;
  writeln;
  writeln;
end.
	#include <iostream>
	#include <iomanip>
	using namespace std;
	
	const int N = 100;
	
	int main()
	{
	  int Z[N],i,ip,ik,isr,k,L,p;
	
	  srand((unsigned)time(NULL));
	
	// wypełniamy tablicę Z[]
	  Z[0] = rand() % 5;
	  for(i = 1; i < N; i++) Z[i] = Z[i - 1] + (rand() % 5);
	
	// generujemy klucz k
	  k = Z[0] + (rand() % (Z[N - 1] - Z[0] + 1));
	
	// poszukujemy binarnie elementu k
	  p = -1; L = ip = 0; ik = N - 1;
	  while(ip <= ik)
	  {
	  L++;
	  isr = (ip + ik) >> 1;
	  if(Z[isr] == k)
	  {
	  p = isr; break;
	  }
	  else if(k < Z[isr])
	  ik = isr - 1;
	  else
	  ip = isr + 1;
	  }
	
	// wyświetlamy wyniki
	  cout << k << " : ";
	  if(p >= 0) cout << p; else cout << "BRAK";
	  cout << " : obiegi = " << L << endl << endl;
	  for(i = 0; i < N; i++)
	  {
	  cout << setw(3) << Z[i];
	  if(p == i) cout << "<"; else cout << " ";
	  }
	  cout << endl << endl;
	  return 0;
	}