Wyszukiwanie binarne

#find #binary_search
#lower_bound
#upper_bound

Zajmiemy się teraz wyszukiwaniem elementu o określonej wartości w kontenerze `vector`. (Analogicznie można wyszukiwać elementy w tablicy.) Pokażemy, jak zrobić to po Bożemu, ale też zaproponujemy Czytelnikowi bardzo efektywny sposób wyszukiwania – ważny, jeśli wektor jest okazałych rozmiarów, a zapytań/wyszukań jest dużo.

„Zwykłe” wyszukiwanie

Załóżmy, że mamy wektor liczb całkowitych V i chcemy sprawdzić, czy występuje w nim określona wartość *value*. Jeśli zawartość wektora nie wykazuje żadnego uporządkowania, można zrobić to zwykłą pętlą, która przespaceruje się po całym wektorze, a w razie znalezienia wyznaczonej wartości zwróci położenie (indeks) tego elementu. W przeciwnym razie powinniśmy otrzymać jakiś umowny indeks, który na pewno nie może wystąpić, na przykład -1. To się da zrobić:

```
int find_index(const & vector<int> V, int value)
{
  for(int i = 0; i < V.size(); i++)
    if(V[i] == value)
      return i;
  return -1;
}
```

Pamiętamy, że instrukcja `return` oznacza wyjście z funkcji, stąd taka konstrukcja algorytmu.

OK, to działa. Ale może warto rozszerzyć horyzonty i posłużyć się biblioteczną funkcją `find()` (wymawiaj: *fajnd*).* Argumentami tej funkcji są dwa iteratory (wskaźniki) określające odkąd-dokąd wyszukujemy oraz wyszukiwana wartość:

```
find(V.begin(), V.end(), value);
```

Tylko co jest jej rezultatem? Otóż jest to iterator wskazujący znaleziony element. Jeśli natomiast wartość *value* nie zostanie znaleziona, funkcja `find()` zwróci wartość $V.end()$.†

Napiszmy przykładowy program ilustrujący działanie tej funkcji. Utworzymy niewielki wektor (o rozmiarze 5), którego elementy wpiszemy z klawiatury i zapytamy, czy istnieje w nim

*Ta funkcja istnieje w kilku odmianach, dla różnych typów danych.
†Przypominamy, że jest to wartość iteratora wskazująca poza wektor V.

element o wartości 3. Jeśli element będzie istniał, wtedy program wypisze indeks tego elementu, a w przeciwnym razie wypisze komunikat **NOPE** (wymawiaj: *nołp*):

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> V(5);
  for(int i = 0; i < 5; i++)
    cin >> V[i];
  vector<int>::iterator p = find(V.begin(), V.end(), 3);
  if(p == V.end())
    cout << "NOPE" << endl;
  else
    cout << p - V.begin() << endl;
}
```

Zauważmy, w jaki sposób wyliczany jest indeks znalezionego elementu: od iteratora p odejmowany jest iterator wskazujący na element $V[0]$. W wyniku daje to ilość pozycji dzielących oba elementy.[‡]

Po uruchomieniu programu i wpisaniu przykładowych liczb:

```
10 4 7 3 12
```

otrzymamy wynik:

```
3
```

Jeśli natomiast wpiszemy liczby:

```
4 5 6 7 8
```

otrzymamy wynik:

```
NOPE
```

Funkcje `find_index()` oraz `find()` działają w czasie liniowym: ilość porównań jest pesymistycznie równa długości wektora – dla nieuporządkowanej struktury danych tak po prostu musi być.

[‡]W języku C/C++ obowiązuje tak zwana *arytmetyka wskaźników*, o czym już wspominaliśmy w podrozdziale *Wskaźniki i iteratory*.

Funkcję `find()` można z powodzeniem zastosować do zwykłej tablicy, na przykład:

```
int A[5];
for(int i = 0; i< 5; i++)
  cin >> A[i];
int *p = find(A, A + N, value);
if(p == A + N)
  cout << "NOPE" << endl;
else
  cout << p - A << endl;
```

Jak widać, zamiast iteratorów wykorzystaliśmy zwykłe wskaźniki. (Zmienna tablicowa A jest – jak pamiętamy – wskaźnikiem do $A[0]$.)

Wyszukiwanie binarne

Jeśli przeszukiwany wektor jest naprawdę spory, a do tego musimy dokonać wielkiej liczby wyszukań, wtedy opłaca się przed przystąpieniem do serii przeszukań najpierw przygotować wektor przez jego posortowanie. Wtedy bowiem do wyszukiwania można zastosować algorytm pokrewny do metody bisekcji opisanej w jednym z wcześniejszych podrozdziałów.

Załóżmy, że mamy strukturę danych (wektor czy tablicę) zawierającą 15[§] przypadkowych liczb i wyszukamy w niej element o wartości 3 (dokładniej: sprawdzimy, czy taki element istnieje):

11	2	14	15	7	6	3	1	20	19	-1	9	19	10	0

Posortujemy wstępnie te liczby niemalejąco (wszakże mogą się powtarzać):

-1	0	1	2	3	6	7	9	10	11	14	15	19	19	20

I teraz wykonamy „strzał” w środek:

-1	0	1	2	3	6	7	9	10	11	14	15	19	19	20

Liczba 9 jest większa od poszukiwanej, więc ograniczamy się do lewej części tablicy:

-1	0	1	2	3	6	7

I znów strzelamy w środek:

-1	0	1	2	3	6	7

[§] Algorytm działa oczywiście dla struktury o dowolnej wielkości, ale dla rozmiaru $2^k - 1$ obrazek jest ładniejszy (dlaczego?).

Tym razem trafiliśmy w liczbę 2 (mniejszą od 3), zatem ograniczamy się do prawej części tablicy:

Jeszcze jeden strzał w środek:

Mamy liczbę 6 (za dużą), więc zajmujemy się lewą częścią tablicy, w której zresztą został tylko jeden element – jest to akurat oczekiwana liczba, czyli nasze wyszukiwanie zakończyło się sukcesem:

Oczywiście gdybyśmy poszukiwali wartości, która nie występuje w tablicy, wtedy zawężenie obszaru do jednego elementu (o wartości innej od wyszukiwanej) byłoby sygnałem, że elementu nie znaleziono.

Proszę zwrócić uwagę na ilość operacji k, jaka jest konieczna do przeprowadzenia wyszukiwania: przy każdym kroku zawężamy wycinek tablicy z grubsza o połowę, zatem spełniona jest przybliżona relacja (N jest rozmiarem tablicy/wektora):

$$N \approx 2^k,$$

zatem złożoność tego algorymu jest logarytmiczna:

$$k \approx \lg_2 N.$$

Rzecz jasna, może się zdarzyć, że na którymś etapie wyszukiwania po prostu trafimy w poszukiwany element, ale pesymistyczna złożoność jest taka, jak powyżej.

Mamy dobrą wiadomość: algorytm wyszukiwania binarnego jest realizowany przez gotową funkcję (nawet jest ich kilka). Funkcja `binary_search()` zwraca wartość logiczną informującą, czy wyszukiwany element został znaleziony. Wywołuje się ją z trzema argumentami: dwa pierwsze określają zakres wyszukiwania (odkąd-dokąd), natomiast trzecim argumentem jest poszukiwana wartość. Oto przykładowy program ilustrujący działanie tej funkcji:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<int> V = {11, 2, 14, 15, 7, 6, 3, 1, 20, 19, -1, 9, 19, 10, 0};
  sort(V.begin(), V.end());
  if(binary_search(V.begin(), V.end(), 3)
    cout << "YEP!" << endl;
  else
    cout << "NOPE!" << endl;
}
```

Użycie tej funkcji dla wyszukiwania wartości *value* w zwykłej (posortowanej) tablicy $A[]$ o rozmiarze N wyglądałoby następująco:

```
binary_search(A, A + N, value)
```

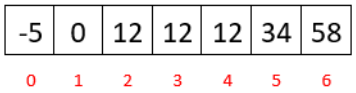
Należy podkreślić, że opisana powyżej funkcja `binary_search()` nie podaje pozycji znalezionego elementu – służą do tego funkcje, które zaprezentujemy w następnej sekcji.

Funkcje `lower_bound()` i `upper_bound()`

Funkcje wymienione w tytule tej sekcji są niezwykle użytecznymi narzędziami wyszukiwania binarnego. Angielskie zwroty (wymawiaj: *lołer baund*, *aper baund*) oznaczają odpowiednio: dolną i górną granicę.

Wywołuje się je analogicznie, jak funkcję `binary_search()`, natomiast ich rezultatem jest iterator/wskaźnik do pewnego szczególnego miejsca w przeszukiwanej strukturze.

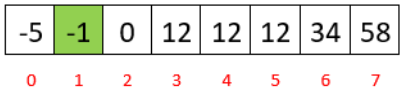
Wyobraźmy sobie następujący posortowany wektor W:



oraz wywołanie funkcji „dolna granica” z wyszukiwaniem wartości -1:

```
lower_bound(W.begin(), W.end(), -1)
```

Rezultatem tej funkcji będzie iterator do najwcześniejszego miejsca w wektorze W, gdzie możemy zgodnie z regułami sztuki[¶] wstawić nowy element. Wartość -1 można bez problemu wstawić pomiędzy -5 oraz 0, czyli na pozycji oznaczonej indeksem 1. Tym samym elementy począwszy od tego miejsca przesunęłyby się o jedną pozycję w prawo i otrzymalibyśmy:



Notabene, taką operację można zrealizować przy pomocy funkcji `insert()`:

```
W.insert(lower_bound(W.begin(), W.end(), -1), -1);
```

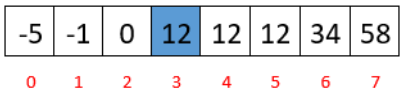
Jak widać, pierwszym jej argumentem jest iterator wskazujący miejsce, gdzie coś należy wstawić, a drugim – wstawiana wartość. (Funkcja `insert()` ma zresztą kilka odmian.)

No dobra, załóżmy że wstawiliśmy tę minus jedynkę i mamy teraz ośmioelementowy wektor W. Pytanie za 50 punktów – jaki będzie rezultat następującego wywołania funkcji:

```
lower_bound(W.begin(), W.end(), 12)
```

[¶] Czyli nie zaburzając porządku elementów.

Zgodnie z tym, co wcześniej było napisane, wynikiem będzie iterator wskazujący na pierwszą dwunastkę:



Możemy także odczytać pozycję tego elementu, odejmując od tego wyniku wartość iteratora wskazującego na początek wektora:

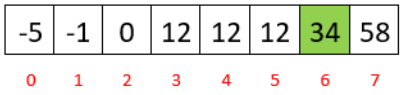
```
lower_bound(W.begin(), W.end(), 12) - W.begin()
```

W naszym przypadku wartość tego wyrażenia wynosi po prostu 3.

A co dostaniemy w wyniku wywołania tej drugiej funkcji – „górnej granicy”?

```
upper_bound(W.begin(), W.end(), 12)
```

Rezultatem będzie iterator wskazujący miejsce za ostatnią dwunastką (czyli najdalszą pozycję, na której możemy umieścić daną wartość):



I znów indeks tego elementu (tutaj mamy 6) można wyliczyć odejmując dwa iteratory:

```
upper_bound(W.begin(), W.end(), 12) - W.begin()
```

Ciekawe, czy Czytelnik domyśli się, jaki wynik dostaniemy, kiedy odejmiemy od siebie wartości obydwu opisywanych tu funkcji, na przykład:

```
upper_bound(W.begin(), W.end(), 12) - lower_bound(W.begin(), W.end(), 12)
```

Jeśli zamiast wektora mamy zwykłą tablicę, wtedy postępujemy tak, jak opisaliśmy to wcześniej: używając zwykłych wskaźników, a nie iteratorów.