Wyszukiwanie

Wejście: posortowana, n-elementowa tablica liczbowa T oraz liczba p.
Wyjście: liczba naturalna, określająca pozycję elementu p w tablicy T,
bądź –1, jeżeli element w tablicy nie występuje.

Wyszukiwanie binarne

Wyszukiwanie binarne polega na tropieniu fragmentu tablicy, o którym wiemy, że musi zawierać element p, o ile element ten znajduje się w tablicy T. Początkowo tym fragmentem jest cała tablica. Przedział kurczy się po porównaniu środkowego elementu ze zmienną p i odrzuceniu odpowiedniej połowy tego przedziału. Proces trwa do chwili odnalezienia p w tablicy lub do momentu, gdy wiadomo, że przedział, w którym musiałby się on znajdować, jest pusty. Złożoność obliczeniowa: $O(\log_2 n)$.
(Na analogicznej zasadzie działa np. algorytm bisekcji, służący do znajdywania miejsc zerowych funkcji.)

Sformułowanie algorytmu

Wiemy, że jeżeli *p* znajduje się w gdziekolwiek w tablicy *T*, to musi być w określonym przedziale. Oznaczmy ten fakt skrótem:
`musi_być(przedział)`.

Szkic programu:

```
zainicjuj przedział jako 1...n
pętla
   // niezmiennik: musi_być(przedział)
   jeżeli przedział pusty
      elementu p nie ma w tablicy T
      koniec
   oblicz wartość środek, środka przedziału
   użyj środek do testów, aby zmniejszyć przedział
   jeżeli p znaleziono podczas testu
      wynik = pozycja p w tablicy
      koniec
```

Przedział reprezentujemy przy pomocy dwóch indeksów: *dół*, *góra*.
Oznaczenie `musi_być(dół, góra)` oznacza, że jeżeli element znajduje się gdzieś w tablicy, to znajduje się w przedziale domkniętym
T`[dół ... góra]`.

Inicjowanie: jakie wartości muszą mieć zmienne *góra* i *dół*, aby warunek `musi_być(dół, góra)` był spełniony? Oczywiście `1` oraz *n*. A zatem:

```
dół  = 1
góra = n
```

Następnie sprawdzamy, czy przedział jest pusty:

```
jeżeli dół > góra
   wynik = -1   // elementu nie ma w tablicy
   koniec
```

Obliczamy wartość środka przedziału:

```
środek = (dół + góra) div 2  // Dzielenie całkowite
```

Szkic programu wygląda teraz tak:

```
dół  = 1
góra = n
pętla
   // niezmiennik: musi_być(dół, góra)
   jeżeli dół > góra
      wynik = -1
      koniec
   środek = (dół + góra) div 2
   użyj środek do testów, aby zmniejszyć przedział
   [dół ... góra]
   jeżeli p znaleziono podczas testu
      wynik = pozycja p w tablicy
      koniec
```

Teraz porównujemy *p* i *T*[*środek*] oraz podejmujemy odpowiednie działania w celu zachowania niezmiennika (4 ostatnie wiersze). Piszemy:

```
jeżeli T[środek] <  p: przypadek A
jeżeli T[środek] == p: przypadek B
jeżeli T[środek] >  p: przypadek C
```

Przypadek B oznacza, że znaleziono element: przypisujemy *wynik* = *środek* i kończymy program.

Analiza przypadku A: Jeżeli *T*[*środek*] < *p*, to *T*[*dół*] ≤ *T*[*dół*+1] ≤ ... ≤ *T*[*środek*] < *p*. Innymi słowy, *p* nie może znajdować się w przedziale *T*[*dół*... *środek*]. Zatem, jeżeli *p* znajduje się w tablicy *T*, to znajduje się w przedziale [*środek* + 1 ... *góra*], co zapisujemy: musi_być(*środek* + 1, *góra*). Przywracamy zatem prawdziwość niezmiennika musi_być(*dół*, *góra*) poprzez podstawienie: *dół* = *środek* + 1.

Analogicznie, w przypadku C przywracamy niezmiennik podstawiając: *góra* = *środek* – 1.

Ostateczna postać pseudokodu wyszukiwania binarnego:

```
dół  = 1
góra = n
pętla
   // niezmiennik: musi_być(dół, góra)
   jeżeli dół > góra
      wynik = -1
      koniec
   środek = (dół + góra) div 2
   jeżeli T[środek] <  p: dół   = środek + 1
   jeżeli T[środek] == p: wynik = środek; koniec
   jeżeli T[środek] >  p: góra  = środek - 1
```

Dowód poprawności algorytmu

Metoda niezmienników Floyda (przypomnienie)

– wyróżnić newralgiczne punkty w algorytmie,

– określić warunki (niezmienniki), jakie mają być spełnione w każdym wyróżnionym punkcie,

– udowodnić poprawność kolejnych warunków, zakładając poprawność warunków poprzedzających,

– własność stopu udowodnić np. metodą liczników iteracji lub metodą malejących wielkości.

```
1. // musi_być(1, n)
2. dół = 1; góra = n
3. // musi_być(dół, góra)
4. pętla
5.     // musi_być(dół, góra)
6.     jeżeli dół > góra
7.         // dół > góra i musi_być(dół, góra)
8.         // p nie ma w tablicy T
9.         wynik = -1;    koniec pętli
10.    // dół ≤ góra i musi_być(dół, góra)
11.    środek = (dół + góra) div 2
12.    // dół ≤ środek ≤ góra i musi_być(dół, góra)
13.    jeżeli T[środek] <  p
14.        // musi_być(dół, góra) i nie_może_być(dół, środek)
15.        // musi_być(środek+1, góra)
16.        dół = środek + 1
17.        // musi_być(dół, góra)
18.    jeżeli T[środek] == p
19.        // T[środek] == p
20.        wynik = środek; koniec pętli
21.    jeżeli T[środek] >  p
22.        // musi_być(dół, góra) i nie_może_być(środek, góra)
23.        // musi_być(dół, środek-1)
24.        góra = środek -1
25.        // musi_być(dół, góra)
26.    // musi_być(dół, góra)
```

Dowód poprawności algorytmu wyszukiwania binarnego będzie się składał się z 3 części:

1. Inicjowanie.
Niezmiennik jest prawdziwy podczas pierwszego wykonania pętli.

2. Zachowanie prawdziwości niezmiennika.
Jeżeli niezmiennik jest prawdziwy na początku iteracji i treść pętli zostanie wykonana, to niezmiennik pozostanie prawdziwy po jej zakończeniu.

3. Zakończenie.
Pętla się skończy i da pożądany skutek (u nas: nadanie zmiennej *wynik* odpowiedniej wartości).

Szczegółowa analiza (uwaga – nudne!).

Prawdziwość warunku (asercji) z wiersza 1, czyli `musi_byc(1,n)`, wynika z definicji tego warunku: jeśli *p* znajduje się w tablicy, to musi być w przedziale `T[1...n]`. Przypisania w wierszu 2 zapewniają więc prawdziwość asercji z wiersza 3, czyli `musi_być(dół, góra)`.

Inicjowanie pętli: wiemy, że asercja w wierszu 3 jest taka sama, jak w wierszu 5.

Jeżeli przeprowadzimy odpowiednią analizę dla wierszy 6-26 będziemy wiedzieli, że prawdziwość niezmiennika jest w pętli zachowana, bo asercje w wierszach 5 i 26 są identyczne.

Pozytywny wynik testu w wierszu 6 prowadzi do asercji w wierszu 7: jeśli *p* znajduje się gdziekolwiek w tablicy, to musi być między *dół* i *góra* dla *dół* < *góra*. Z tych warunków wynika wiersz 8: *p* nie ma w tablicy. Tak więc poprawnie kończymy pętlę w wierszu 9 po nadaniu zmiennej *wynik* wartości `-1`.

Jeżeli test w wierszu 6 da wynik negatywny, to przechodzimy do wiersza 10. Niezmiennik jest nadal zachowany (nie zrobiliśmy nic, co mogłoby go zmienić), a w wyniku testu wiemy, że *dół* <= *góra*. Wiersz 11 to nadanie zmiennej *środek* wartości średniej z *dół* i *góra*, zaokrąglonej w dół do najbliższej liczby całkowitej. Ponieważ średnia zawsze znajduje się pomiędzy dwiema wartościami a zaokrąglenie nie zmniejszy jej poniżej *dół*, prawdziwa jest asercja w wierszu 12.

Dowód poprawności instrukcji warunkowych w wierszach 13-25 dotyczy każdego z trzech możliwych przypadków.

Przypadek drugi (wiersz 18): ponieważ prawdziwa jest asercja z wiersza 19, nadanie zmiennej *wynik* wartości *środek* i zakończenie pętli są poprawne. To jest drugie miejsce, w którym pętla może się zakończyć.

Pozostałe przypadki (czyli pierwszy i trzeci) są symetryczne. Rozpatrzmy dla przykładu warunek z wierszy 21-25. Pierwszy człon asercji z wiersza 22 nie został naruszony podczas działania programu. Drugi człon jest prawdziwy, bo $p < T[środek] \leq T[środek+1] \leq ... \leq T[n]$ – wiemy, że *p* nie może się znajdować w tablicy powyżej indeksu *środek*-1, co wyrażone zostało warunkiem nie_może_być(*środek, góra*). Logika podpowiada, że jeżeli *p* musi być między *dół* a *góra*, i nie może być pod indeksem *środek* ani wyżej, to musi być znajdować się między *dół* a *środek*-1 (o ile w ogóle się oczywiście w tablicy znajduje). Stąd się bierze wiersz 23. Wykonanie instrukcji z wiersza 24 przy prawdziwej asercji z wiersza 23, daje prawdziwość asercji w wierszu 25. Zatem niezmiennik w wierszu 26 pozostaje prawdziwy.

Rozumowanie dla wierszy 13-17 wygląda identycznie.

Wykazaliśmy więc, że jeżeli pętla się kończy, to zmienna *wynik* ma właściwą wartość. Może się jednak pojawić pętla nieskończona.
W dowodzie własności stopu wykorzystamy metodę malejących wielkości. Przedział *dół*... *góra* ma początkowy rozmiar *n*. Z wierszy 6-9 wynika, że pętla się skończy, jeżeli przedział zawiera mniej niż jeden element. Musimy więc wykazać, że przedział się zmniejsza przy każdym obiegu pętli. Na podstawie wiersza 12 wiemy, że *środek* znajduje się zawsze w bieżącym przedziale. Pierwsza i trzecia instrukcja warunkowa wykluczają indeks *środek* z bieżącego przedziału i w ten sposób zmniejszają jego rozmiar co najmniej o 1. A zatem pętla – a co za tym idzie cały program – musi się zatrzymać.