Algorytmy i Struktury Danych

Wyszukiwanie

(c) Marcin Sydow

Zawartość tego wykładu:

- reguła **"dziel i rządź"**
- wyszukiwanie
- **algorytm wyszukiwania binarnego**
- statystyki pozycyjne
- 2. najmniejsza wartość w ciągu (algorytm "turniejowy" - idea)
- algorytm Hoare'a (wyszukiwanie k-tej statystyki pozycyjnej)

"Dziel i rządź"

Jest to jedna z najskuteczniejszych **technik projektowania algorytmów**. Dzielimy problem na podproblemy (czyli dane wejściowe na mniejsze części) i wyjaśniamy jak z rozwiązań tych podproblemów otrzymać rozwiązanie oryginalnego problemu.

Często jest implementowana za pomocą rekursji, czyli **techniki programowania** polegającej na wywołaniu przez funkcję jej samej dla mniejszych danych wejściowych.

Nazwa reguły zapożyczona jest z identycznie brzmiącej nazwy strategii w polityce polegającej na podzieleniu sił przeciwnika na mniejsze części, czyniąc go słabszym i bardziej podatnym na wewnętrzne konflikty. Przypisywana oryginalnie Filipowi II Macedońskiemu, królowi Macedonii (382-336 przed Chrystusem) jest permanentnie stosowana w polityce, nie wyłączając czasu obecnego.

Problem wyszukiwania

`search(S, len, key)`

Input: S - ciąg liczb całkowitych; len - długość ciągu; key (klucz) - liczba całkowita (wyszukiwana w ciągu S)

Output: indeks (liczba naturalna mniejsza od len), pod którą w ciągu S znajduje się liczba key (S[index]==key) albo -1 jeśli klucz jest nieobecny w S

Przykład: S = (3,5,8,2,1,8,4,2,9):

- search(S, 9, 2) powinno zwrócić: 3
- search(S, 9, 7) powinno zwrócić: -1

Wyszukiwanie, c.d.

Naturalnym kandydatem na **operację dominującą** w algorytmach implementujących problem wyszukiwania jest **porównanie** (pomiędzy kluczem a elementami w ciągu S), natomiast **rozmiar danych** zawiera długość ciągu len (może też obejmować inne parametry, np. algorytm skoku co k)

Wyszukiwanie sekwencyjne:
Najprościej jest zrealizować algorytm wyszukiwania przez przeglądanie wszystkich indeksów np. od 0 do len-1. Taka realizacja skutkuje **liniową** pesymistyczną złożonością czasową $W(len) = len$.

UWAGA: Zauważmy, że pesymistycznej złożoności czasowej **nie można poprawić** zmieniając porządek przeglądania indeksów (gdyż zawsze może się zdarzyć, że np. szukany element jest pod ostatnim sprawdzanym indeksem).

Bardziej efektywne wyszukiwanie?

Jaka **dodatkowa własność** ciągu mogłaby umożliwić szybsze wyszukiwanie?

Bardziej efektywne wyszukiwanie?

Jaka **dodatkowa własność** ciągu mogłaby umożliwić szybsze wyszukiwanie?

Uporządkowanie wartości ciągu wejściowego (np. od najmniejszej do największej)

Zmienimy więc nieco specyfikację problemu, żeby opracować bardziej efektywne algorytmy.

Wyszukiwanie w ciągu posortowanym

Zmodyfikowana specyfikacja problemu:

Input: S - ciąg **niemalejąco posortowanych** (tzn. mogą być powtarzające się wartości) liczb całkowitych (indeksowanych od 0); len - naturalna liczba będącą długością ciągu S; key (klucz) - wyszukiwana liczba całkowita

Output: indeks (liczba naturalna mniejsza od len), pod którą w ciągu S znajduje się liczba key albo -1 jeśli klucz jest nieobecny w S.

Przy tak zmienionej specyfikacji (uporządkowanie ciągu wejściowego) można pokusić się o algortymy o pesymistycznej złożoności czasowej lepszej niż $W(len) = len$.

Algortym skoków co k; idea i analiza

Jeśli ciąg wejściowy jest posortowany, to możliwe jest wyszukiwanie poprzez sprawdzanie co k-tego indeksu (pomijając k-1 elementów podczas każdego "skoku"). W przypadku znalezienia pierwszego elementu większego od wyszukiwanego klucza, wystarczy sprawdzić jedynie ostatnio "przeskoczone" k-1 elementów.

Zauważmy, że dla $len \rightarrow \infty$ taki algorytm jest w przeciętnym przypadku *asymptotycznie* **k razy szybszy** w przeciętnym przypadku niż "normalny" algorytm sekwencyjny (dla niewielkich wartości k).
Również, przy odpowiednim doborze k[1], pesymistyczna złożoność czasowa tego algorytmu jest odpowiednio niższa: $W(len) = \frac{1}{k} \cdot \Theta(len)$, a więc też **k razy szybciej** niż sekwencyjnego.
Jest więc poprawa. Jednak wciąż jest to złożoność **liniowa**, czyli o tym samym **rzędzie złożoności** (czyli jesli dane wzrosną x razy, algorytm będzie działał x razy dłużej, etc.).

Czy możliwe jest uzyskanie **niższej niż liniowa** złożoności czasowej?

[1] ćwiczenie: udowodnić, że zachodzi to dla $k = \sqrt{len}$

Algorytm Wyszukiwania Binarnego - idea

Przykład zastosowania "dziel i rządź" w wyszukiwaniu.

`search(S, len, key)`

(wciąż zakładamy tu, że ciąg wejściowy jest posortowany niemalejąco)

Algorytm wyszukiwania binarnego:

1. dopóki długość ciągu jest dodatnia:
2. porównaj klucz ze środkowym elementem ciągu
3. jeśli jest równość, to zwróć wynik (bieżący indeks)
4. jeśli jest niższy niż element - ogranicz dalsze poszukiwania tylko do lewego podciągu (na lewo od bieżącego indeksu)
5. jeśli jest wyższy niż element - ogranicz dalsze poszukiwania tylko do prawego podciągu (na prawo od bieżącego indeksu)
6. wróć do kroku 1
7. jeśli długość ciągu spadła do zera, to nie ma klucza w ciągu

Kod algorytmu wyszukiwania binarnego


```
search(S, len, key){

  l = 0
  r = len - 1

  while(l <= r){
      m = (l + r)/2
      if(S[m] == key) return m
      else
        if(S[m] > key) r = m - 1
        else l = m + 1
  }

  return -1
}
```

Uwaga: zakłada się, że cały ciąg mieści się w szybkiej pamieci RAM (ang. random access memory), czyli pamięci o dostępie swobodnym, czyli, że sprawdzenie dowolnego indeksu `S[m]` ma czas stały (jest szybkie) i nie zależy od m.

Analiza algorytmu wyszukiwania binarnego

rozmiar danych: długość ciągu - len
operacja dominująca: porównanie - `(S[m]==key)`
(zakładamy, że cały ciąg jest w RAM)

Zauważmy, że z każdą iteracją bieżący ciąg staje się 2 razy krótszy. Algorytm zatrzymuje się, gdy długość bieżącego ciągu spadnie do 0 (lub wcześniej znajdziemy klucz).

$$W(len) = \Theta(log_2(len))$$

$$A(len) = \Theta(log_2(len))$$

$$S(len) = O(1)$$

(założenie o pamięci RAM jest istotne dla faktu, że każda operacja S(m)==key ma czas stały, w przeciwieństwie do sytuacji, gdyby np. dane były na "wolnym" dysku)

Statystyki pozycyjne

K-ta statystyka pozycyjna w ciągu to po prostu k-ty najmniejszy (lub największy) element w tym ciągu.

Przykład: szukanie minimum jest szczególnym przypadkiem, dla k=1

W przypadku, gdy ciąg jest posortowany zadanie byłoby trywialne: wynik byłby po prostu na k-tej pozycji posortowanego ciągu.

Dlatego, w dalszej części tego wykładu powrócimy znowu do wersji wyszukiwania dla ciągów nieuporządkowanych.

Wyszukiwanie 2-giego najmniejszego elementu w ciągu nieuporządkowanym

`second(S, len)`

(uwaga: nie ma już założenia o uporządkowaniu ciągu wejściowego, ale dla uproszczenia zakładamy, że wszystkie elementy są różne)

Input: S - ciąg liczb całkowitych; len - długość ciągu

Output: drugi najmniejszy element w ciągu S

Rozmiar danych: len; operacja dominująca: porównanie

Rozwiązanie proste: znajdź minimum, usuń je z ciągu, i znajdź ponownie minimum (wymaga $2 \cdot len - 1$ porównań)

Wyszukiwanie 2-giego najmniejszego elementu w ciągu nieuporządkowanym

`second(S, len)`

(uwaga: nie ma już założenia o uporządkowaniu ciągu wejściowego, ale dla uproszczenia zakładamy, że wszystkie elementy są różne)

Input: S - ciąg liczb całkowitych; len - długość ciągu

Output: drugi najmniejszy element w ciągu S

Rozmiar danych: len; operacja dominująca: porównanie

Rozwiązanie proste: znajdź minimum, usuń je z ciągu, i znajdź ponownie minimum (wymaga $2 \cdot len - 1$ porównań)

Jak zrobić to używając mniejszej liczby porównań?
(z mniejszą złożonością czasową)

Algorytm "turniejowy" - idea

Zastosujmy regułę "dziel i rządź":
Spójrzmy na ten problem jak na "turniej" rozgrywany przez elementy ciągu.

W każdej fazie turnieju bieżący zbiór elementów jest dzielony jest na pary elementów, które rozgrywają "grę": mniejszy element "wygrywa" i przechodzi do następnej fazy turnieju. W ten sposób w każdej fazie pozostaje tylko ok. połowy elementów z poprzedniej fazy, dopóki nie pozostanie jeden element ("zwycięzca") - jest to najmniejszy element w ciągu.

Gdzie jest 2-gi najmniejszy element? (którego szukamy)

Algorytm "turniejowy" - idea

Zastosujmy regułę "dziel i rządź":
Spójrzmy na ten problem jak na "turniej" rozgrywany przez elementy ciągu.

W każdej fazie turnieju bieżący zbiór elementów jest dzielony jest na pary elementów, które rozgrywają "grę": mniejszy element "wygrywa" i przechodzi do następnej fazy turnieju. W ten sposób w każdej fazie pozostaje tylko ok. połowy elementów z poprzedniej fazy, dopóki nie pozostanie jeden element ("zwycięzca") - jest to najmniejszy element w ciągu.

Gdzie jest 2-gi najmniejszy element? (którego szukamy)

Odpowiedź: pomiędzy elementami, które grały (i przegrały) ze "zwycięzcą" (tylko z nim mógł przegrać). Wystarczy więc w drugiej fazie przeszukać te elementy aby znaleźć wynik.

Analiza algorytmu turniejowego

Turniej może być naturalnie przedstawiony w postaci drzewa binarnego, gdzie na najniższym poziomie (liście) są wszystkie elementy oryginalnego ciągu a na górze (korzeń) jest zwycięzca. Każdy poziom odpowiada kolejnej fazie turnieju. Ile jest poziomów (jako funkcja len)?

Analiza algorytmu turniejowego

Turniej może być naturalnie przedstawiony w postaci drzewa binarnego, gdzie na najniższym poziomie (liście) są wszystkie elementy oryginalnego ciągu a na górze (korzeń) jest zwycięzca. Każdy poziom odpowiada kolejnej fazie turnieju. Ile jest poziomów (jako funkcja len)?

$\Theta(log_2(len))$ (bo każdy poziom zawiera 2 razy mniej elementów niż niższy)

Rozmiar danych: len

Operacja dominująca: porównanie między 2 elementami

Ile jest wszystkich porównań w pierwszej fazie turnieju?

Analiza algorytmu turniejowego

Turniej może być naturalnie przedstawiony w postaci drzewa binarnego, gdzie na najniższym poziomie (liście) są wszystkie elementy oryginalnego ciągu a na górze (korzeń) jest zwycięzca. Każdy poziom odpowiada kolejnej fazie turnieju. Ile jest poziomów (jako funkcja len)?

$\Theta(log_2(len))$ (bo każdy poziom zawiera 2 razy mniej elementów niż niższy)

Rozmiar danych: len

Operacja dominująca: porównanie między 2 elementami

Ile jest wszystkich porównań w pierwszej fazie turnieju?

len-1 (dlaczego?)

Analiza algorytmu turniejowego

Turniej może być naturalnie przedstawiony w postaci drzewa binarnego, gdzie na najniższym poziomie (liście) są wszystkie elementy oryginalnego ciągu a na górze (korzeń) jest zwycięzca. Każdy poziom odpowiada kolejnej fazie turnieju. Ile jest poziomów (jako funkcja len)?

$\Theta(log_2(len))$ (bo każdy poziom zawiera 2 razy mniej elementów niż niższy)

Rozmiar danych: len

Operacja dominująca: porównanie między 2 elementami

Ile jest wszystkich porównań w pierwszej fazie turnieju?
len-1 (dlaczego?) (bo z każdym porównaniem odpada dokładnie jeden element, a na końcu zostaje 1 element)

W drugiej fazie należy jeszcze wyszukać najmniejszy element spośród tych, które przegrały ze zwycięzcą (jest ich tyle ile poziomów turnieju)
Ostatecznie: $W(len) = len - 1 + \Theta(log_2(len))$ - jest to więc asymptotycznie 2 razy szybciej niż dwukrotne szukanie minimum.

K-ta statystyka pozycyjna - Algorytm Hoare'a (idea)

`kthSmallest(S, len, k)`

(nic nie zakładamy o ciągu S, może być nieuporządkowany)

Input: S - ciąg liczb całkowitych; len - długość ciągu; k - liczba naturalna dodatnia ($k <= len$) (

Output: element s, który jest k-tym najmniejszym elementem w ciągu S

Rozwiązanie naiwne: k razy wyszukujemy element minimalny oznacza to ok. $k \cdot len$ porównań.

Jednak zadanie może być znowu rozwiązane efektywniej za pomocą podejścia "dziel i rządź"

Procedura Partition

Przedstawimy teraz pewną pomocniczą procedurę:
`partition(S, l, r)`
input: S - ciąg liczb całkowitych; l, r - lewy i prawy skrajny indeks aktualnie przetwarzanego podciągu ciągu S.
output: i - finalna pozycja elementu "medianowego" (opis poniżej)

Działanie procedury: bierze dowolny (pierwszy) element ciągu m, i przestawia wszystkie elementy ciągu tak, że wszystkie elementy na lewo od elementu m są niewiększe (ale niekoniecznie posortowane) od niego a na prawo niemniejsze. Zwraca ostateczną pozycję *i* elementu m.

Wykorzystanie procedury Partition

Uwaga: dzięki powyższej specyfikacji, jeśli zwrócony indeks wynosiłby dokładnie k, to musi on zawierać k-ty najmniejszy element ciągu. (załóżmy dla uproszczenia, że indeksujemy ciąg od 0)

Jeśli natomiast indeks *i* jest mniejszy od k, to należy powtórzyć działanie partition na części podciągu na prawo od i a w przeciwnym wypadku na lewo od i ("dziel i rządź").
Jest to nieco podobny pomysł do wyszukiwania binarnego, ale tym razem i nie musi być (i rzadko jest) dokładnie w połowie ciągu.

Analiza procedury partition

Kod procedury podany będzie w innym wykładzie (przy okazji algorytmu QuickSort, gdzie również jest wykorzystywana). Natomiast ważne jest, że przy założeniach:

- **Operacja dominująca**: porównanie 2 elementów
- **Rozmiar danych**: długość oryginalnego ciągu $n = (r - l + 1)$

Procedura partition może być nietrudno zaprojektowana ze złożonością $W(n) = n + O(1)$ (i $S(n) = O(1)$)

Algorytm Hoare'a

(szybkie znajdowanie k-tego najmniejszego elementu w ciągu)

- wykonaj `partition` na ciągu
- jeśli zwrócony indeks i jest równy k, to koniec (zwróć: $S[k]$)
- w przeciwnym wypadku kontynuuj, na podciągu na lewo lub prawo (w zależności od porównania i z k) od wartości k, dopóki zwrócony indeks nie wyniesie dokładnie k

Dzięki liniowej złożoności procedury partition, *przeciętna* złożoność algorytmu Hoare'a jest liniowa ($\Theta(n)$) **niezależnie od wartości k**.

Uwaga: procedura partition ma jeszcze ważniejsze zastosowanie w innym algorytmie: sortowania szybkiego (QuickSort), omawianym w innym wykładzie.

Przykładowe pytania/problemy:

- podaj specyfikację problemu wyszukiwania klucza w ciągu
- algorytm skoków co k: podaj specyfikację, opisz działanie, dokonaj analizy poprawności i złożoności czasowej i zasymuluj jego działanie na danych wejściowych
- algorytm **wyszukiwania binarnego**: podaj specyfikację, opisz działanie, napisz z pamięci kod (wersja z wykładu), dokonaj analizy poprawności i złożoności, zasymuluj działanie na danych wejściowych.
- co to jest statystyka pozycyjna?
- algorytm turniejowy: specyfikacja, opis działania, analiza złożoności
- podaj specyfikację i złożoność czasową procedury Partition
- opisz ideę algorytmu Hoare'a.
- dlaczego algorytm Hoare'a ma **kwadratową** pesymistyczną złożoność czasową?

Dziękuję za uwagę