

Algorytmy i struktury danych

dr hab. Adam Kiersztyn

Plan wykładu:

W1: Poprawność i złożoność obliczeniowa algorytmów

W2: Metody projektowania algorytmów

W3: Wyszukiwanie i sortowanie

W4: Algorytmy rekurencyjne. Rozwiązywanie niektórych równań rekurencyjnych

W5: Podstawowe struktury danych i abstrakcyjne struktury danych

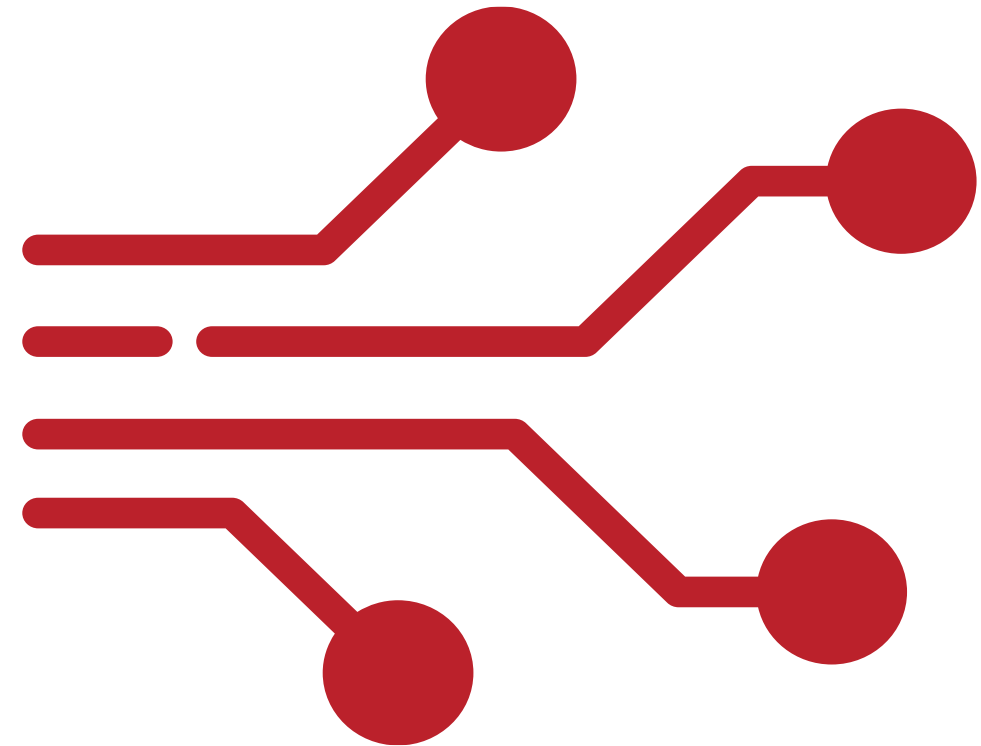
W6: Kolejka priorytetowa, słownik

W7: Podstawowe algorytmy grafowe

W8: Wybrane algorytmy tekstowe

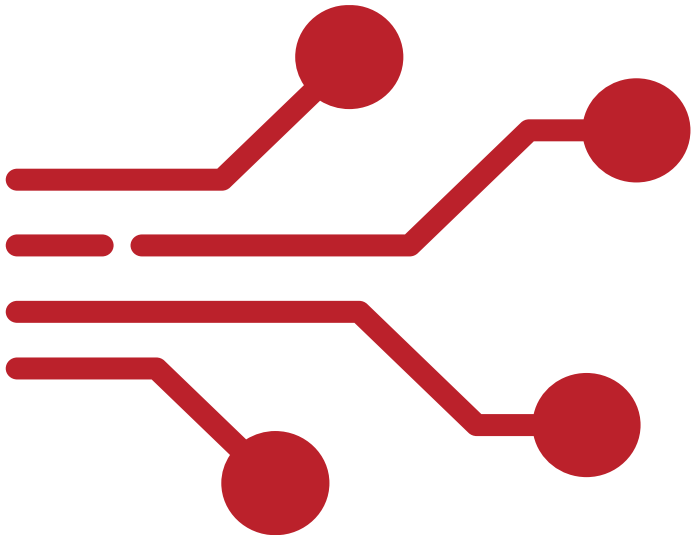


Poprawność i złożoność obliczeniowa algorytmów



Plan wykładu:

1. Wprowadzenie
2. Podstawowe pojęcia algorytmiki
3. Sposoby reprezentacji algorytmów
4. Poprawność algorytmów
5. Złożoność obliczeniowa
6. Przykłady algorytmów i ich analiza



Wprowadzenie

Algorytmy stanowią fundament współczesnej informatyki oraz nauk ścisłych. Badanie poprawności i złożoności algorytmów jest kluczowe zarówno z perspektywy teoretycznej, jak i praktycznej. Niezawodność oprogramowania zależy od dowiedzenia, że algorytm spełnia swoją specyfikację, zaś jego efektywność od określenia jego kosztu obliczeniowego.

Niniejszy wykład przedstawia formalne podstawy analizy algorytmów, metody dowodzenia ich poprawności oraz zasady oceny złożoności obliczeniowej. Prezentowane treści obejmują liczne przykłady i implementacje w dwóch językach: C# oraz Python, co pozwala na porównanie stylów programowania i konsekwencji dla analizy.

5

Podstawowe pojęcia algorytmiki

Algorytm można zdefiniować jako skończony ciąg dobrze zdefiniowanych instrukcji, które przekształcają dane wejściowe w dane wyjściowe. Warunki formalne algorytmu:

- **Skończoność,**
- **Jednoznaczność,**
- **Wykonalność,**
- **Określoność danych wejściowych i wyjściowych.**

Podstawowe pojęcia algorytmiki

1. Skończoność (Finiteness)

Algorytm musi wykonywać **skończoną liczbę kroków** – tzn. dla każdego dopuszczalnego wejścia musi się zakończyć po określonej liczbie instrukcji. Nie może prowadzić do nieskończonej pętli. Nie może wymagać nieskończonych zasobów lub czasu. Zapewnia możliwość uzyskania wyniku w praktyce.

Przykład:

Pętla z warunkiem malejącym (np. od n do 0) gwarantuje skończoność.

7

Podstawowe pojęcia algorytmiki

2. Jednoznaczność (Definiteness / Unambiguity)

Każda instrukcja algorytmu musi być precyzyjnie określona, tak aby nie pozostawiała miejsca na interpretację. Każdy krok powinien być zrozumiały i nie może mieć kilku możliwych znaczeń.

Algorytm powinien dawać zawsze ten sam wynik dla tych samych danych wejściowych. Instrukcje muszą być jasne nawet dla maszyny, nie tylko człowieka.

Podstawowe pojęcia algorytmiki

3. Wykonalność (Effectiveness / Feasibility)

Każdy krok algorytmu musi być realny do wykonania:

- Każda instrukcja powinna być wykonana w skończonym czasie.
- Instrukcja musi być możliwa do przeprowadzenia na podstawowych operacjach danego modelu obliczeniowego (np. RAM, maszyna Turinga).
- Nie mogą to być kroki wymagające „magii” lub czynności niemożliwych.

9

Sposoby reprezentacji algorytmów

Dlaczego potrzebujemy różnych form?

- Różni odbiorcy (programiści, analitycy, uczniowie)
- Różne poziomy szczegółowości
- Lepsze zrozumienie logiki
- Możliwość łatwiejszego wdrażania i testowania

10

Sposoby reprezentacji algorytmów

1. Opis słowny (język naturalny)

Charakterystyka:

- Algorytm jest przedstawiony w formie tekstu pisanego w języku naturalnym (np. polskim).
- Często stosowany w materiałach edukacyjnych, dokumentacji oraz podczas omawiania algorytmów na poziomie koncepcyjnym.

11

Sposoby reprezentacji algorytmów

1. Opis słowny (język naturalny)

Zalety:

- Czytelny również dla osób bez wiedzy programistycznej.
- Łatwo przekazać intuicję działania algorytmu.

12

Sposoby reprezentacji algorytmów

1. Opis słowny (język naturalny)

Wady:

- Może być nieprecyzyjny (ryzyko niejednoznaczności).
- Nie nadaje się do formalnych dowodów poprawności.

13

Sposoby reprezentacji algorytmów

1. Opis słowny (język naturalny)

Przykład:

- Weź pierwszy element listy.
- Porównaj z kolejnymi elementami.
- Jeśli znajdziesz większy, zapamiętaj go.
- Po przejściu całej listy zwróć największy element.

14

Sposoby reprezentacji algorytmów

2. Pseudokod

Charakterystyka:

Najpopularniejsza forma w nauce algorytmów.

Przypomina język programowania, ale jest uproszczony i niezależny od konkretnego języka.

Stosowany w książkach, artykułach naukowych, wykładach.

15

Sposoby reprezentacji algorytmów

2. Pseudokod

Zalety:

Jasny, jednoznaczny, zbliżony do programowania.

Łatwy do implementacji w dowolnym języku.

Doskonały do analizy poprawności oraz złożoności.

Wady:

Wymaga podstawowej wiedzy o strukturach programowania.

16

Sposoby reprezentacji algorytmów

2. Pseudokod

Przykład

$\text{max} \leftarrow A[1]$

for $i \leftarrow 2$ to n do

 if $A[i] > \text{max}$ then

$\text{max} \leftarrow A[i]$

return max

17

Sposoby reprezentacji algorytmów

3. Schemat blokowy (flowchart)

Charakterystyka:

- Graficzna reprezentacja algorytmu w postaci bloków i strzałek.
- Standardowe symbole:
 - ◇ – decyzja
 - – operacja
 - ⊥ – start/stop
 - ⊂ – wejście/wyjście

18

Sposoby reprezentacji algorytmów

3. Schemat blokowy (flowchart)

Zalety:

- Bardzo intuicyjne i zrozumiałe wizualnie.
- Pomocne w prezentowaniu logiki osobom mniej technicznym.
- Przydatne w analizie procedur biznesowych i automatyzacji.

Wady:

- Dla złożonych algorytmów schemat może być bardzo duży i nieczytelny.
- Mniej formalne niż pseudokod

19

Sposoby reprezentacji algorytmów

4. Diagram N-S (Nassi–Shneiderman diagram)

Charakterystyka:

- Reprezentacja algorytmów w postaci bloków ułożonych w prostokąty.
- Każdy blok odpowiada strukturze programistycznej: sekwencji, wyboru, pętli.
- Wygląda jak „program w pudełkach”.

20

Sposoby reprezentacji algorytmów

4. Diagram N-S (Nassi–Shneiderman diagram)

Zalety:

- Przedstawia strukturalny charakter algorytmu (odpowiada programowaniu strukturalnemu).
- Łatwy do konwersji na kod.

Wady:

- Rzadziej stosowany niż schematy blokowe.
- Mniej intuicyjny dla początkujących.

21

Sposoby reprezentacji algorytmów

5. Kod źródłowy (w języku programowania)

Charakterystyka:

- Najbardziej praktyczna forma – algorytm zapisany bezpośrednio w języku (np. C++, Python, Java).
- Najczęściej używana w praktyce inżynierskiej.

Sposoby reprezentacji algorytmów

5. Kod źródłowy (w języku programowania)

Zalety:

- Można bezpośrednio uruchomić, testować i optymalizować.
- Precyzyjny – kompilator wymusza poprawność składniową.

Wady:

- Różne języki mają różną składnię i możliwości.
- Kod jest często mniej przejrzysty niż pseudokod w celach dydaktycznych

23

Poprawność algorytmów

Algorytm jest poprawny, jeżeli:

- Dla każdego dopuszczalnych danych wejściowych:
 - kończy działanie (skończona liczba kroków),
 - zwraca wynik zgodny ze specyfikacją.
- Poprawność jest zawsze rozumiana względem specyfikacji (opis zadania).

24

Poprawność algorytmów

Specyfikacja – formalny opis tego, co algorytm ma robić:

- Opis problemu – np. „dla tablicy liczb znaleźć jej maksimum”.
- Dane wejściowe – typy, zakresy, dopuszczalne wartości.
- Wynik (dane wyjściowe) – co powinno zostać obliczone.

Często zapisywana za pomocą:

- warunków wstępnych (precondition, Pre),
- warunków końcowych (postcondition, Post).

25

Poprawność algorytmów

Warunek wstępny (precondition)

- Założenia dotyczące danych wejściowych.
- Muszą być spełnione przed uruchomieniem algorytmu.
- Przykład: „tablica A ma długość $n \geq 1$ ”.

Warunek końcowy (postcondition)

- Własność, którą powinien spełniać wynik po zakończeniu algorytmu.
- Przykład (maksimum):
 - wynik jest elementem tablicy A
 - dla każdego i: $A[i] \leq \text{wynik}$.

26

Poprawność algorytmów

Rodzaje poprawności

1. Poprawność częściowa (partial correctness)

- Jeśli algorytm się zatrzyma, to wynik jest zgodny z postcondition.
- Nie gwarantuje, że algorytm zawsze się zatrzyma.

2. Poprawność całkowita (total correctness)

- Algorytm zawsze się zatrzymuje dla danych spełniających precondition.
- Wynik jest zgodny z postcondition.
- = poprawność częściowa + skończoność.

27

Poprawność algorytmów

Sposoby dowodzenia poprawności

Najczęściej stosowane metody:

1. Indukcja matematyczna - przydatna m.in. do algorytmów rekurencyjnych i iteracyjnych.
2. Inwariant pętli - kluczowy dla dowodu poprawności pętli.
3. Dowód przez sprzeczność - zakładamy, że algorytm zwraca zły wynik i wyprowadzamy sprzeczność.
4. Specyfikacja Hoare'a.

28

Poprawność algorytmów

Inwariant pętli – własność, która:

- Jest prawdziwa przed wejściem do pętli,
- Jest zachowywana po każdej iteracji,
- Pomaga wykazać, że po zakończeniu pętli spełniony jest postcondition.

Stosujemy 3 kroki:

- Inicjalizacja – inwariant prawdziwy przed pierwszym wejściem do pętli.
- Zachowanie (utrzymanie) – jeśli był prawdziwy przed iteracją, to jest prawdziwy po iteracji.
- Zakończenie – po wyjściu z pętli z inwariantu i warunku zakończenia wynika postcondition.

29

Poprawność algorytmów

Poprawność a testowanie

- Dowód poprawności: Daje pewność formalną, że algorytm działa dla wszystkich danych spełniających precondition.
- Testowanie: Sprawdza działanie na wybranych przypadkach (np. przykłady, testy jednostkowe).

W praktyce:

- testy są niezbędne,
- ale dla kluczowych algorytmów (kryptografia, systemy krytyczne) pożądane są również formalnie udowodnione własności.

30

Złożoność obliczeniowa

Ocena, jak szybko rośnie czas działania lub zapotrzebowanie na pamięć algorytmu wraz ze wzrostem liczby danych wejściowych n .

- Ocenia się nie czas rzeczywisty, lecz liczbę operacji.
- Pozwala porównywać algorytmy niezależnie od sprzętu.
- Ważna dla dużych zbiorów danych.

31

Złożoność obliczeniowa

Rodzaje złożoności

Dwa najczęściej analizowane typy:

- Złożoność czasowa – liczba operacji potrzebnych do wykonania algorytmu.
- Złożoność pamięciowa – ilość dodatkowej pamięci niezbędnej do działania algorytmu.

Dodatkowo wyróżnia się:

- najlepszy przypadek,
- średni przypadek,
- najgorszy przypadek.

Złożoność obliczeniowa

Notacja asymptotyczna

Służy do opisu, jak algorytm skaluje się dla dużego n .

Najczęściej używane notacje:

- $O(g(n))$ – oszacowanie od góry (złożoność maksymalna).
- $\Omega(g(n))$ – oszacowanie od dołu.
- $\Theta(g(n))$ – ścisłe oszacowanie (zarówno od góry, jak i od dołu).

Najczęściej w praktyce stosuje się $O(\dots)$.

33

Złożoność obliczeniowa

Typowe klasy złożoności

- $O(1)$ — stała
- $O(\log n)$ — logarytmiczna
- $O(n)$ — liniowa
- $O(n \log n)$ — liniowo-logarytmiczna
- $O(n^2)$ — kwadratowa
- $O(2^n)$ — wykładnicza
- $O(n!)$ — silniowa (bardzo wolna)

34

Złożoność obliczeniowa

Przykłady algorytmów i ich złożoności

- $O(1)$ – dostęp do elementu tablicy
- $O(\log n)$ – wyszukiwanie binarne
- $O(n)$ – przegląd całej listy / znalezienie minimum
- $O(n \log n)$ – szybkie sortowanie [średnio], mergesort
- $O(n^2)$ – sortowanie bąbelkowe, sortowanie przez wstawianie
- $O(2^n)$ – rozwiązywanie problemu plecakowego metodą brute-force
- $O(n!)$ – pełne generowanie permutacji

35

Złożoność obliczeniowa

Analiza złożoności pętli

Zasady pomocnicze:

- Pętla wykonująca się n razy $\rightarrow O(n)$.
- Pętla wewnątrz pętli $\rightarrow$ mnożymy złożoności, np. $n \times n \rightarrow O(n^2)$.
- Instrukcje wykonywane stałą liczbą razy $\rightarrow O(1)$.

Złożoność obliczeniowa

Złożoność pamięciowa

Opisuje, ile dodatkowej pamięci (poza danymi wejściowymi) wymaga algorytm.

Przykłady:

- $O(1)$ – algorytmy in-place (np. iteracyjne liczenie maksimum).
- $O(n)$ – przechowywanie dodatkowej tablicy wyników.
- $O(n \log n)$ – np. niektóre algorytmy sortowania z rekurencją.
- $O(n^2)$ – macierze i tablice dynamiczne 2D.

Złożoność pamięciowa jest szczególnie istotna przy dużych zbiorach danych lub ograniczonej pamięci urządzenia.

37

Złożoność obliczeniowa

Dlaczego analiza złożoności jest ważna?

- Pozwala przewidzieć czas działania programu dla dużych danych.
- Umożliwia porównanie algorytmów rozwiązujących ten sam problem.
- Pomaga wybrać najbardziej efektywne rozwiązanie.
- Chroni przed projektowaniem algorytmów, które działają dobrze dla małych zbiorów, ale drastycznie spowalniają dla dużych.

38

Przykłady algorytmów i ich analiza

Wyszukiwanie liniowe (Linear Search)

Opis działania

- Sprawdzamy kolejne elementy tablicy od początku do końca.
- Jeśli znajdziemy szukaną wartość — zwracamy jej indeks.
- Jeśli nie — algorytm zgłasza brak wyniku.

Przykłady algorytmów i ich analiza

Pseudokod

LinearSearch(A, x):

 for i = 1 to n:

 if A[i] ==

x:

 return i

 return "brak"

Analiza

Złożoność czasowa

- najlepszy przypadek: $O(1)$
- najgorszy przypadek: $O(n)$
- średni przypadek: $O(n)$

Złożoność pamięciowa:
 $O(1)$

40

Przykłady algorytmów i ich analiza

Wyszukiwanie binarne (Binary Search)

Opis działania

- Działa tylko dla tablic posortowanych.
- Sprawdzamy środkowy element.
- W zależności od wyniku odrzucamy lewą lub prawą połowę.
- Powtarzamy aż znajdziemy element lub zakres się wyczerpie.

41

Przykłady algorytmów i ich analiza

BinarySearch(A, x):

```
left = 1    right = n
```

```
while left <= right:
```

```
    mid = (left + right) div 2
```

```
    if A[mid] == x:
```

```
        return mid
```

```
    else if A[mid] < x:
```

```
        left = mid + 1
```

```
    else:
```

```
        right = mid - 1
```

```
return "brak"
```

Analiza

Złożoność czasowa

- najlepszy przypadek:
 $O(1)$
- najgorszy przypadek:
 $O(n)$
- średni przypadek: $O(n)$

Złożoność pamięciowa:
 $O(1)$

Przykłady algorytmów i ich analiza

Sortowanie bąbelkowe (Bubble Sort)

Opis działania

- Porównujemy pary sąsiadujących elementów.
- Jeśli są w złej kolejności — zamieniamy miejscami.
- Powtarzamy cały proces aż do pełnego posortowania tablicy.

Przykłady algorytmów i ich analiza

BubbleSort(A):

for i = 1 to n-1:

for j = 1 to n-i:

if A[j] > A[j+1]:

zamień A[j] z A[j+1]

Analiza

Złożoność czasowa:

- najlepszy przypadek:
 $O(n)$

- najgorszy i średni
przypadek: $O(n^2)$

Złożoność pamięciowa:
 $O(1)$

44

Przykłady algorytmów i ich analiza

Sortowanie przez scalanie (Merge Sort)

Opis działania

- Dzielimy tablicę na dwie części, aż do elementów pojedynczych.
- Następnie scalane są dwie posortowane połowy w jedną całość.
- Opiera się na metodzie *dziel i zwyciężaj*.

45

Przykłady algorytmów i ich analiza

MergeSort(A):

```
if długość A > 1:  
    podziel A na L i R  
    MergeSort(L)  
    MergeSort(R)  
    A = Merge(L, R)
```

Merge(L, R):

```
wynik = pusty  
i = j = 0  
while i < długość L AND j <  
długość R:  
    if L[i] <= R[j]:  
        dodaj L[i] do wynik; i++  
    else:  
        dodaj R[j] do wynik; j++  
dodaj pozostałe elementy L i R  
return wynik
```

46

Przykłady algorytmów i ich analiza

Sortowanie przez scalanie (Merge Sort)

Analiza

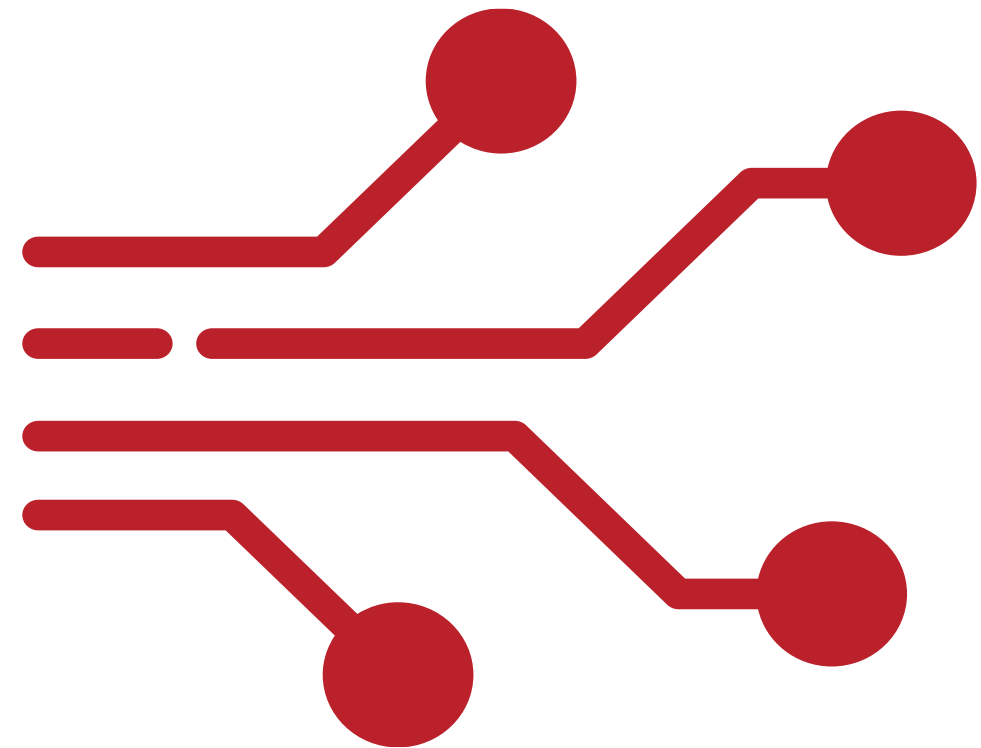
- Złożoność czasowa: $O(n \log n)$
- Złożoność pamięciowa: $O(n)$

Zalety: stabilne i szybkie.

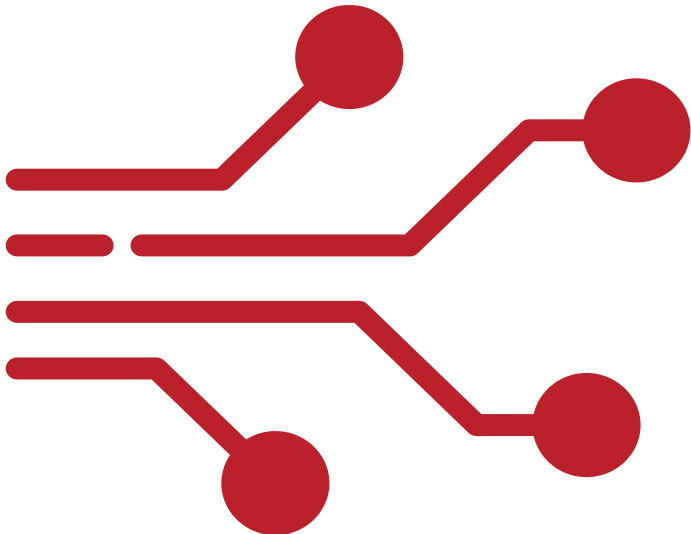
Wada: wymaga dodatkowej pamięci.

47

Metody projektowania algorytmów



Plan wykładu:



W ramach niniejszego wykładu omówione zostaną najważniejsze klasyczne metody projektowania algorytmów, które stanowią fundament współczesnej algorytmiki oraz praktycznego programowania. Każda metoda zostanie przedstawiona wraz z opisem idei, kluczowych właściwości, przykładami zastosowań, typowymi schematami oraz pseudokodami.

49

Plan wykładu

W szczególności zajmiemy się metodami:

- Metoda zachłanna
- Programowanie dynamiczne
- Dziel i zwyciężaj
- Przegląd zupełny (brute force)
- Heurystyki i metaheurystyki

Dodatkowo omówimy sposoby wyboru odpowiedniej metody dla danego problemu oraz porównamy ich właściwości.

Na czym polega projektowanie algorytmów?

Projektowanie algorytmów to proces tworzenia efektywnych i poprawnych metod rozwiązywania problemów obliczeniowych.

Obejmuje:

- **analizę struktury problemu**, jego ograniczeń i danych wejściowych,
- **wybór odpowiedniej strategii rozwiązania**,
- **sformułowanie algorytmu w formie pseudokodu lub schematu**,
- **dowód poprawności oraz ocena złożoności**,
- **dobór najefektywniejszej metody spośród dostępnych podejść**.

Celem jest skonstruowanie algorytmu, który jest nie tylko poprawny, ale również czasowo i pamięciowo optymalny.

51

Dlaczego potrzebujemy wielu metod projektowania?

Nie istnieje jedna uniwersalna metoda projektowania algorytmów dobra dla wszystkich problemów. Różne problemy mają różne własności, struktury i ograniczenia – dlatego wymagają odmiennych technik.

- Problemy optymalizacyjne często wymagają **programowania dynamicznego** lub **metod zachłannych**.

52

Dlaczego potrzebujemy wielu metod projektowania?

- Problemy związane z dużymi danymi i rekurencją naturalnie pasują do **dziel i zwyciężaj**.
- Problemy trudne obliczeniowo (NP-trudne) często rozwiązujemy za pomocą **heurystyk**, ponieważ metody dokładne byłyby zbyt wolne.
- Tam, gdzie potrzebna jest gwarancja znalezienia najlepszego rozwiązania, używamy metod **dokładnych**, nawet jeśli są kosztowne.

53

Klasyfikacja metod projektowania

Metody projektowania algorytmów można klasyfikować według różnych kryteriów:

1. Ze względu na deterministyczność

- deterministyczne — zawsze zwracają ten sam wynik dla tych samych danych (np. merge sort),
- niedeterministyczne lub probabilistyczne — mogą stosować losowość (np. algorytmy Monte Carlo).

Klasyfikacja metod projektowania

2. Ze względu na charakter optymalizacji

- dokładne — zawsze zwracają rozwiązanie optymalne (np. DP),
- przybliżone — wynik bliski optymalnemu (algorytmy aproksymacyjne),
- heurystyczne — dobre rozwiązanie, ale bez gwarancji jakości (np. hill climbing).

3. Ze względu na podejście strukturalne

- rekurencyjne (dziel i zwyciężaj),
- iteracyjne (zachłanne),
- hybrydowe (np. QuickSort + insertion sort w małych przedziałach).

55

Idea metody zachłannej

Metoda zachłanna opiera się na podejmowaniu w każdym kroku decyzji, która lokalnie wydaje się najlepsza według pewnego kryterium. Zakłada się, że lokalna optymalność prowadzi do globalnej optymalności.

Typowy krok zachłanny:

- wybieramy element o najlepszej wartości w danym momencie,
- utrwalamy dokonany wybór,
- nie cofamy się do wcześniejszych decyzji.

To podejście działa tylko dla problemów o tzw. strukturze zachłannej.

56

Kiedy metoda zachłanna działa poprawnie?

Aby metoda zachłanna dawała rozwiązanie optymalne, muszą zostać spełnione warunki:

- *Własność optymalnej podstruktury*: Optymalne rozwiązanie problemu składa się z optymalnych rozwiązań podproblemów.
- *Własność zachłannego wyboru*: Zawsze istnieje optymalne rozwiązanie, które zawiera element wybrany metodą zachłanną.
- *Brak potrzeby cofania decyzji*: Wszystkie wybory są nieodwracalne – raz podjęta decyzja nie wymaga korekty.

Niespełnienie tych warunków sprawia, że metoda zachłanna może być niepoprawna (np. klasyczny problem plecaka 0/1).

57

Ogólny schemat algorytmu zachłannego

1. Zdefiniuj zbiór dopuszczalnych rozwiązań.
2. Zdefiniuj funkcję oceny/priorytetu dla możliwości wyboru.
3. Wybierz element o najlepszym priorytecie.
4. Sprawdź, czy wybór jest zgodny z ograniczeniami.
5. Dodaj element do rozwiązania.
6. Zaktualizuj strukturę problemu (usuń element, zmodyfikuj zasoby).
7. Powtarzaj, aż problem zostanie rozwiązany lub nie ma dalszych wyborów.

58

Zalety i wady metod zachłannych

Zalety:

- prostota implementacji,
- często bardzo dobra złożoność — typowo **$O(n \log n)$** — związana z sortowaniem lub kolejkami priorytetowymi,
- szybkie działanie,
- idealne do problemów z dużymi wejściami.

Zalety i wady metod zachłannych

Wady:

- brak gwarancji rozwiązania optymalnego dla wszystkich problemów,
- trudność dowodu poprawności,
- lokalne decyzje mogą prowadzić do globalnie nieoptymalnych rozwiązań,
- wymuszona nieodwracalność wyborów.

60

Przykłady problemów rozwiązywanych zachłannie

- problem doboru zadań nienakładających się czasowo (interval scheduling),
- minimalne drzewo rozpinające (algorytm Kruskala i Prima),
- wydawanie reszty (w systemach kanonicznych),
- kompresja Huffmana,
- najkrótsze ścieżki w grafie z wagami nieujemnymi (Dijkstra).

W wielu problemach metoda zachłanna okazała się kluczem do znacznego uproszczenia algorytmów.

61

Przykład: dobór zadań (interval scheduling)

Opis problemu:

Dane są zadania, każde z określonym czasem rozpoczęcia i zakończenia.

Celem jest wybranie największej liczby zadań, które nie nakładają się na siebie.

Pseudokod:

Sortuj zadania niemalejąco wg czasu zakończenia

Wybrane $\leftarrow \emptyset$

OstatniKoniec $\leftarrow -\infty$

for każde zadanie z listy:

if start_zadania $\geq$

OstatniKoniec:

dodaj zadanie do Wybrane

OstatniKoniec $\leftarrow$

koniec_zadania

return Wybrane

Dowód poprawności metody zachłannej

Dowód poprawności zwykle opiera się na dwóch elementach:

- **Inwariant pętli** – w każdej iteracji zbiór *Wybrane* jest zbiorem zadań możliwych do wykonania (nienakładających się).
- **Argument zamiany (exchange argument)** – Można wykazać, że jeśli w optymalnym rozwiązaniu występuje zadanie kończące się później niż wybór zachłanny, to można je zamienić, uzyskując nie gorsze rozwiązanie.

Tego typu dowód jest klasycznym sposobem weryfikowania metod zachłannych.

63

Złożoność metody zachłannej

Najważniejsze elementy analizy:

- sortowanie zadań / zdarzeń / krawędzi $\rightarrow O(n \log n)$,
- jeden przegląd listy $\rightarrow O(n)$,
- ewentualnie operacje na kolejce priorytetowej $\rightarrow O(\log n)$ każda.

Łączna złożoność typowego algorytmu zachłannego:
 $O(n \log n)$.

Kiedy nie stosować metody zachłannej?

Metoda zachłanna może prowadzić do błędnych rozwiązań w sytuacjach, gdy:

- wybór lokalnie najlepszy ma negatywny wpływ na dalsze decyzje,
- brak jest właściwości optymalnej podstruktury,
- koszt wyboru zależy od decyzji przyszłych (niespełniona niezależność wyborów),
- problem wymaga rozważenia wielu kombinacji, które metoda zachłanna odrzuca zbyt wcześnie.

65

Kiedy nie stosować metody zachłannej?

Klasyczny kontrprzykład: **problem plecaka 0/1**, w którym wybór przedmiotu o najlepszym stosunku wartości do wagi nie zawsze prowadzi do optimum.

Podsumowanie sekcji o metodzie zachłannej

Metoda zachłanna jest jedną z najważniejszych i najczęściej stosowanych technik projektowania algorytmów. Jest szybka, prosta i intuicyjna, lecz wymaga bardzo starannej analizy poprawności. Nadaje się tylko dla wybranej klasy problemów o określonej strukturze optymalności. W dalszej części wykładu poznamy metody znacznie bardziej potężne, takie jak programowanie dynamiczne, które pozwalają rozwiązywać problemy bardziej złożone i wymagające.

67

Idea programowania dynamicznego

Programowanie dynamiczne (DP, dynamic programming) to technika projektowania algorytmów, która polega na rozwiązywaniu problemów poprzez dzielenie ich na mniejsze podproblemy, zapamiętywanie wyników tych podproblemów oraz wykorzystanie ich do obliczenia rozwiązania dla problemu głównego.

Kluczowe założenie: wiele podproblemów powtarza się, a ich wyniki można przechowywać, aby uniknąć wielokrotnego obliczania.

68

Kiedy stosujemy programowanie dynamiczne?

Aby zastosować DP, problem musi spełniać dwa kluczowe warunki:

1. Optymalna podstruktura

Rozwiązanie problemu można zbudować z optymalnych rozwiązań jego mniejszych części.

2. Nakładanie się podproblemów

Te same podproblemy pojawiają się wielokrotnie, co umożliwia zapamiętanie ich wyników.

Dodatkowo problem musi dawać się opisać za pomocą zależności rekurencyjnych — to one prowadzą do konstrukcji tabeli DP.

69

Podejścia: top-down i bottom-up

Programowanie dynamiczne można realizować na dwa sposoby:

Top-down (memoizacja)

- piszemy funkcję rekurencyjną,
- przy pierwszym obliczeniu wyniku zapamiętujemy go w tablicy,
- kolejne wywołania korzystają z wyników zapisanych.

Zaletą jest intuicyjność — odwzorowujemy naturalną rekurencyjną strukturę problemu.

70

Podejścia: top-down i bottom-up

Programowanie dynamiczne można realizować na dwa sposoby:

Bottom-up (tablicowanie)

- tworzymy tablicę DP,
- wypełniamy ją, zaczynając od stanów najprostszych,
- zajmujemy się coraz większymi podproblemami.

Zaletą jest przewidywalność złożoności i brak narzutu rekurencji.

Jak projektować algorytm DP?

1. **Precyzyjna analiza struktury problemu** – określenie, jakie podproblemy są kluczowe.
2. **Zdefiniowanie funkcji DP** – np. $dp[i][j]$ oznacza najlepszy wynik dla pierwszych i elementów i ograniczenia j .
3. **Ustalenie zależności rekurencyjnej** – formuły typu:
 $dp[x] = \min(dp[x-1], dp[x-2] + \text{koszt})$

Jak projektować algorytm DP?

1. **Ustalenie przypadków bazowych.**
2. **Wybór podejścia top-down lub bottom-up.**
3. **Rekonstrukcja rozwiązania**, jeśli problem tego wymaga (np. wybór elementów w plecaku).

To proces metodyczny, często wymagający eksperymentów i głębokiej analizy.

73

Przykładowy problem: liczby Fibonacciego

Wersja top-down z memoizacją:

`fib(n):`

`if  $n \leq 1$ : return n`

`if memo[n] istnieje: return memo[n]`

`memo[n] = fib(n-1) + fib(n-2)`

`return memo[n]`

74

Przykładowy problem: liczby Fibonacciego

Wersja bottom-up:

$\text{fib}(0)=0$, $\text{fib}(1)=1$

for $i = 2$ to n :

$\text{fib}(i) = \text{fib}(i-1) + \text{fib}(i-2)$

return $\text{fib}(n)$

Złożoność obu wersji: $O(n)$.

75

Przykład: plecak 0/1 (knapsack)

Opis problemu można ująć następująco: dany jest zbiór przedmiotów, z których każdy ma określoną wagę $w[i]$ oraz wartość $v[i]$. Do dyspozycji mamy plecak o maksymalnej pojemności W , której nie wolno przekroczyć. Naszym zadaniem jest wybrać taki podzbiór przedmiotów, aby łączna waga wybranych elementów nie była większa niż pojemność plecaka.

Jednocześnie celem nie jest zmieszczenie jak największej liczby przedmiotów, lecz osiągnięcie możliwie największej sumy ich wartości. Oznacza to, że czasem bardziej opłaca się wybrać jeden cenny przedmiot niż kilka mniej wartościowych. Problem ten polega więc na znalezieniu najlepszego kompromisu między wagą a zyskiem. Z tego powodu jest on jednym z najbardziej klasycznych zagadnień optymalizacyjnych w algorytmice. 76

Przykład: plecak 0/1 (knapsack)

Pseudokod bottom-up:

```
for i = 0..n:
    for w = 0..W:
        if i == 0 or w == 0:
            dp[i][w] = 0
        else if w[i] ≤ w:
            dp[i][w] = max(dp[i-1][w], v[i] + dp[i-1][w-w[i]])
        else:
            dp[i][w] = dp[i-1][w]
```

Złożoność: $O(n \cdot W)$ (pseudo-wielomianowa).

Koszt pamięci i optymalizacje DP

W wielu algorytmach DP wykorzystuje się tablice 2D o rozmiarach rzędu setek tysięcy elementów. Możliwe optymalizacje:

- redukcja wymiaru tablicy do 1D (np. w plecaku),
- przechowywanie tylko dwóch ostatnich wierszy,
- technika "divide & conquer optimization",
- optymalizacje Knutha,
- przejście do programowania z ograniczonymi stanami (state compression).

W praktyce wybór struktur danych jest równie ważny jak sam algorytm.

78

Zalety i wady DP

Zalety:

- zawsze daje rozwiązanie optymalne,
- bardzo szerokie zastosowanie,
- szczególnie skuteczne w problemach kombinatorycznych,
- podstawowa metoda rozwiązywania problemów NP-trudnych w wersjach ograniczonych.

Wady:

- wysoka złożoność pamięciowa,
- trudność w zaprojektowaniu stanu,
- rekurencje mogą być skomplikowane,
- algorytm jest często mniej intuicyjny niż metoda zachłanna.

79

Przykłady zastosowań DP

- analiza sekwencji DNA (LCS, edycja ciągów),
- kompresja danych (np. dynamiczne struktury Huffmana),
- grafy — najkrótsze ścieżki w DAG, drogi Hamiltona (DP po maskach),
- podziały macierzy (matrix chain multiplication),
- wyznaczanie ciągów o minimalnym koszcie,
- algorytmy tekstowe,
- wyznaczanie sposobów ułożenia kafelków, liczenia ścieżek itp.

DP to jedna z kluczowych technik w zawodach algorytmicznych.

80

Idea metody dziel i zwyciężaj

Metoda *divide and conquer* polega na dzieleniu problemu na mniejsze części, rozwiązywaniu ich niezależnie (najczęściej rekurencyjnie), a następnie scalaniu wyników.

Schemat:

- **Podziel** problem na podproblemy mniejsze.
- **Rozwiń** każdy podproblem (rekurencja).
- **Połącz** wyniki podproblemów w całość.

Metoda ta jest podstawą wielu najszybszych algorytmów sortowania, przeszukiwania czy przetwarzania sygnałów.

81

Przykłady klasyczne metody dziel i zwyciężaj

- wyszukiwanie binarne,
- sortowanie przez scalanie (merge sort),
- szybkie sortowanie (quick sort),
- algorytm Karacuby dla mnożenia liczb,
- algorytmy FFT,
- znajdowanie najbliższej pary punktów,
- drzewa segmentowe.

82

Złożoność metody

Złożoność wielu algorytmów D&Z można określić z ogólnej postaci rekurencji:

$$T(n) = a \cdot T(n / b) + f(n)$$

gdzie:

- a – liczba podproblemów,
- b – stopień zmniejszenia rozmiaru,
- $f(n)$ – koszt dzielenia i scalania.

Ta teoretyczna metoda pozwala w prosty sposób (bez pełnego rozwijania rekurencji) określić asymptotykę wielu algorytmów.

83

Zalety metody dziel i zwyciężaj

Zalety:

- naturalna struktura rekurencyjna,
- często bardzo dobra złożoność – nawet $O(n \log n)$,
- możliwość dużej paralelizacji,
- wykorzystywane w algorytmach szybkiego przetwarzania sygnałów.

Wady:

- narzut rekurencji,
- trudności w scalaniu wyników,
- problemy z pamięcią stosu dla głębokiej rekurencji.

Quick Sort — pseudokod

Quicksort to klasyczny algorytm D&Z.

QuickSort(A, p, r):

 if $p < r$:

$q = \text{Partition}(A, p, r)$

 QuickSort(A, p, $q-1$)

 QuickSort(A, $q+1$, r)

Złożoność:

- średnia: $O(n \log n)$
- najgorsza: $O(n^2)$

85

Metoda siłowa (brute force)

Metoda brute force polega na sprawdzeniu wszystkich możliwych rozwiązań problemu. Choć bardzo nieefektywna, jest jedyną metodą gwarantującą znalezienie optymalnego rozwiązania dla problemów, które nie mają efektywnych algorytmów.

Przykłady:

- generowanie permutacji,
- szukanie wzorca w tekście,
- przeszukiwanie drzew decyzyjnych,
- pełne przeszukanie rozwiązań NP-trudnych dla małych n .

86

Kiedy stosujemy brute force?

- gdy rozmiar danych jest mały,
- gdy potrzebujemy rozwiązania pewnego,
- by porównać wynik z algorytmem przybliżonym,
- w debugowaniu i testowaniu,
- jako element podejść hybrydowych.

87

Zalety i wady brute force

Zalety:

- prostota,
- gwarancja znalezienia optimum,
- łatwość implementacji.

Wady:

- złożoność potencjalnie rzędu $n!$, 2^n , n^n ,
- niepraktyczne dla dużych danych,
- wymaga silnego ograniczania przestrzeni przeszukiwania.

88

Pseudokod: generowanie permutacji

```
Permute(A, l, r):  
    if l == r:  
        wypisz A  
    else:  
        for i = l..r:  
            zamień A[l] z A[i]  
            Permute(A, l+1, r)  
            zamień A[l] z A[i]
```

89

Czym są heurystyki?

Heurystyki to metody projektowania algorytmów, które znajdują rozwiązania dobre (niekoniecznie optymalne) w rozsądnym czasie. Używa się ich, gdy problemy są zbyt złożone, aby stosować algorytmy dokładne — zwykle problemy NP-trudne.

Heurystyki stosuje się w:

- analizie grafów,
- logistyce i planowaniu,
- sztucznej inteligencji,
- przetwarzaniu obrazów,
- eksploracji danych.

90

Metaheurystyki

Metaheurystyki to bardziej ogólne struktury umożliwiające sterowanie procesem poszukiwania:

- algorytmy genetyczne,
- symulowane wyżarzanie,
- optymalizacja rojowa,
- tabu search,
- algorytmy ewolucyjne.

Cechy wspólne:

- stosują elementy losowości,
- przeszukują przestrzeń rozwiązań globalnie,
- potrafią unikać lokalnych minimów.

91

Przykład: symulowane wyżarzanie

Symulowane wyżarzanie imituje proces chłodzenia metalu.

- rozwiązania gorsze są czasem akceptowane z pewnym prawdopodobieństwem,
- z czasem „temperatura” spada, zmniejszając szansę na pogorszenie,
- umożliwia opuszczenie lokalnych minimów.

To jedna z najskuteczniejszych metod heurystycznych w optymalizacji kombinatorycznej.

92

Zalety i wady heurystyk

Zalety:

- bardzo szybkie,
- dają rozwiązania dobre jakościowo,
- działają tam, gdzie algorytmy dokładne są niepraktyczne.

Wady:

- brak gwarancji optimum,
- wyniki mogą zależeć od parametrów,
- trudne do analizy matematycznej.

93

Porównanie metod projektowania

Metoda	Gwarancja optimum	Złożoność	Typ problemów
Zachłanna	czasem	$n \log n$	struktura zachłanna
DP	zawsze	$n^2 - n^3$	nakładanie podproblemów
D&Z	zależy	$n \log n$	struktura rekurencyjna
Brute force	zawsze	wykładnicza	małe dane
Heurystyki	nie	szybka	NP-trudne

Podsumowanie

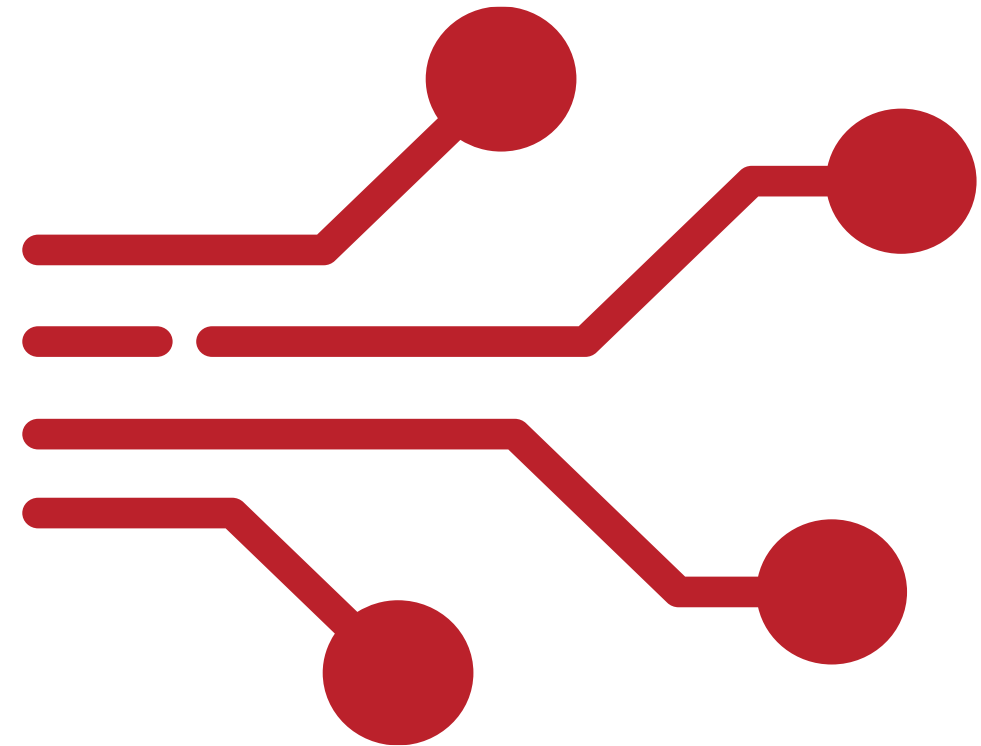
Metody projektowania algorytmów stanowią fundament nowoczesnej informatyki. Każda z nich jest odpowiednia dla innego typu problemów i często kluczowe znaczenie ma umiejętność oceny, która strategia będzie najefektywniejsza.

W praktyce często używa się podejść hybrydowych, łączących różne idee.

W kolejnych wykładach przedstawione zostaną bardziej zaawansowane techniki oraz ich zastosowania w sortowaniu, przeszukiwaniu, grafach i algorytmach tekstowych.

95

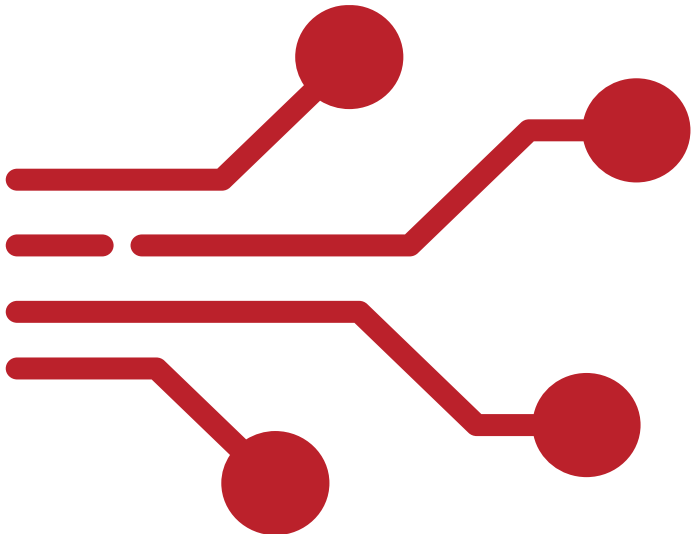
Wyszukiwanie i sortowanie



96

Plan wykładu

- Wprowadzenie
- Wyszukiwanie liniowe
- Wyszukiwanie binarne
- Sortowanie bąbelkowe
- Sortowanie przez wstawianie
- Sortowanie przez wybór
- Sortowanie przez scalanie
- Quicksort
- Heap sort
- Sortowanie kubełkowe
- Radix sort



Wprowadzenie do wyszukiwania

Wyszukiwanie to proces odnajdywania w strukturze danych elementu, który spełnia określone kryterium — najczęściej porównanie z wartością zapytania. Chociaż koncepcja wydaje się banalna, wybór odpowiedniej metody wyszukiwania ma ogromne znaczenie praktyczne. Dla bardzo dużych zbiorów różnica między wyszukiwaniem liniowym a binarnym może oznaczać sekundy zamiast godzin przetwarzania.

98

Wprowadzenie do wyszukiwania

Wyszukiwanie jest tak powszechne, że stało się inspiracją dla tworzenia nowych struktur danych dedykowanych przyspieszeniu tej operacji — od drzew BST i AVL, przez tablice mieszające, aż po złożone indeksy bazodanowe.

Rozpoczynamy jednak od klasycznych, prostych metod, które są fundamentem dla bardziej skomplikowanych rozwiązań.

Modele wyszukiwania danych

Aby świadomie dobierać algorytmy wyszukiwania, musimy rozumieć, w jakich warunkach będą działać. Pierwsze kluczowe pytanie brzmi: *czy dane są uporządkowane?*

Wyszukiwanie w nieposortowanej tablicy jest znacznie ograniczone — w praktyce jedyną metodą pozostaje przegląd kolejnych elementów. Natomiast uporządkowanie danych umożliwia zastosowanie znacznie efektywniejszych strategii.

100

Modele wyszukiwania danych

Kolejne pytanie dotyczy dynamiki danych: *czy elementy będą często dodawane, usuwane, aktualizowane?* Niektóre metody działają świetnie dla danych statycznych, lecz zupełnie tracą efektywność przy częstych modyfikacjach.

Wreszcie, istotny jest model pamięci oraz liczba zapytań: pojedyncze wyszukiwanie w dużym zbiorze danych wymaga innej strategii niż setki tysięcy wyszukiwań wykonywanych w krótkich odstępach czasu. Wszystkie te czynniki decydują o wyborze metody.

101

Wyszukiwanie liniowe – opis

Aby świadomie dobierać algorytmy wyszukiwania, musimy rozumieć, w jakich warunkach będą działać. Pierwsze kluczowe pytanie brzmi: *czy dane są uporządkowane?*

Wyszukiwanie w nieposortowanej tablicy jest znacznie ograniczone – w praktyce jedyną metodą pozostaje przegląd kolejnych elementów. Natomiast uporządkowanie danych umożliwia zastosowanie znacznie efektywniejszych strategii.

102

Wyszukiwanie liniowe – opis

Kolejne pytanie dotyczy dynamiki danych: *czy elementy będą często dodawane, usuwane, aktualizowane?* Niektóre metody działają świetnie dla danych statycznych, lecz zupełnie tracą efektywność przy częstych modyfikacjach.

Wreszcie, istotny jest model pamięci oraz liczba zapytań: pojedyncze wyszukiwanie w dużym zbiorze danych wymaga innej strategii niż setki tysięcy wyszukiwań wykonywanych w krótkich odstępach czasu. Wszystkie te czynniki decydują o wyborze metody.

103

Wyszukiwanie liniowe — pseudokod

```
LinearSearch(A, x):  
    for i = 1 to n:  
        if A[i] == x:  
            return i  
    return "brak"
```

104

Wyszukiwanie liniowe — pseudokod

Pseudokod wyszukiwania liniowego odzwierciedla jego prostotę: wykonujemy pojedynczą pętlę, która sprawdza każdy element. Kod jest tak intuicyjny, że stanowi doskonały punkt wyjścia do nauki analizy algorytmiki. Algorytm pokazuje, czym jest koszt iteracji, jak działa najgorszy, średni i najlepszy przypadek.

Zwróćmy uwagę, że zwrócenie wyniku następuje natychmiast po znalezieniu elementu, dlatego najlepszy przypadek to pojedyncze porównanie. Jednak średnio musimy przejść połowę tablicy, a jeśli elementu nie ma to całą.

105

Wyszukiwanie liniowe – analiza

Analiza wyszukiwania liniowego jest dobrym przykładem, jak ważna jest znajomość modeli złożoności czasowej. W najlepszym przypadku wynik znajduje się na samym początku tablicy, wtedy algorytm wykonuje tylko jedną operację porównania. Jest to sytuacja optymalna, ale jednocześnie mało reprezentatywna.

W praktyce częściej spotykamy przypadek średni, w którym zakładamy równomierny rozkład wartości, wówczas po średnio $n/2$ porównaniach osiągamy wynik. Gdy jednak szukany element nie istnieje, zmuszeni jesteśmy przejrzeć całą tablicę, co daje czas $O(n)$. 106

Wyszukiwanie liniowe – analiza

Zaletą tej metody jest bardzo niewielkie zużycie pamięci, ponieważ wykorzystuje ona jedynie kilka zmiennych pomocniczych do przechowywania bieżącego zakresu oraz pozycji środka. Nie ma potrzeby tworzenia dodatkowych struktur danych ani kopiowania elementów, co znacząco ogranicza narzut pamięciowy. Dzięki temu algorytm jest lekki i łatwy do zastosowania nawet w prostych środowiskach.

Takie podejście sprawia, że metoda jest szczególnie przydatna w urządzeniach o ograniczonych zasobach, takich jak systemy wbudowane czy starszy sprzęt. W takich przypadkach każdy dodatkowy fragment pamięci ma znaczenie, dlatego oszczędność jest dużą zaletą. Co więcej, brak dodatkowych struktur upraszcza implementację i zmniejsza ryzyko błędów.¹⁰⁷

Wyszukiwanie binarne – opis

Wyszukiwanie binarne to przykład algorytmu, który wykorzystuje dodatkową wiedzę o danych, a mianowicie fakt, że są on posortowane, aby radykalnie zmniejszyć ilość pracy potrzebnej do odnalezienia elementu.

Zamiast zaczynać od pierwszego elementu, wybieramy środek zbioru. Następnie jedno porównanie pozwala nam odrzucić połowę przestrzeni poszukiwań: jeśli element szukany jest mniejszy, to z pewnością znajduje się w lewej części tablicy, a jeśli większy, to w prawej.

108

Wyszukiwanie binarne – opis

Ten proces powtarza się rekurencyjnie lub iteracyjnie, za każdym razem zmniejszając zakres o połowę. Właśnie dlatego czas działania wyszukiwania binarnego jest logarytmiczny.

Jest to jedna z najbardziej eleganckich i efektywnych metod wyszukiwania wykorzystywana w większości struktur danych umożliwiających szybki dostęp.

109

Wyszukiwanie binarne – pseudokod

BinarySearch(A, x):

left = 1

right = n

while left <= right:

mid = (left + right) div 2

if A[mid] == x:

return mid

else if A[mid] < x:

left = mid + 1

else:

right = mid - 1

return "brak"

110

Wyszukiwanie binarne – pseudokod

Pseudokod ten jest przykładem prostoty, która jednocześnie kryje bardzo dużą efektywność działania algorytmu. Choć sam zapis jest krótki i przejrzysty, opiera się na bardzo silnej idei dzielenia problemu na mniejsze części. Kluczową rolę odgrywa tutaj zmienna *mid*, która wskazuje środkowy element aktualnego zakresu poszukiwań i stanowi punkt odniesienia dla dalszych decyzji.

To właśnie dzięki niej możliwe jest szybkie określenie, w której połowie zbioru może znajdować się szukana wartość. Jeśli element środkowy jest większy lub mniejszy od szukanego, od razu odrzucamy połowę danych, która na pewno nie zawiera wyniku. Takie podejście znacząco redukuje liczbę potrzebnych porównań.

Wyszukiwanie binarne – pseudokod

Warto zauważyć, że najdroższą operacją w algorytmie jest samo porównanie wartości, dlatego tak istotne jest ograniczenie ich liczby.

Warunek zakończenia pętli $\text{left} \leq \text{right}$ zapewnia, że w momencie przecięcia się wskaźników lewego i prawego przedziału, poszukiwania mają logiczne zakończenie, bowiem nie ma już elementów do sprawdzenia.

112

Wyszukiwanie binarne – analiza

Analiza wyszukiwania binarnego wskazuje na ogromny zysk wynikający z uporządkowania danych. Każde porównanie zmniejsza liczbę możliwych pozycji do sprawdzenia o połowę, co daje złożoność $O(\log n)$. To oznacza, że przy milionie rekordów wykonamy zaledwie około 20 porównań, zamiast milion. Różnica jest astronomiczna.

Nawet w najgorszym przypadku, gdy element nie istnieje, algorytm wciąż działa logarytmicznie. Jest to fundament wielu technik wyszukiwania, od przeszukiwania drzew binarnych po optymalizację z użyciem funkcji unimodalnych.

Porównanie metod wyszukiwania

- Wyszukiwanie liniowe i binarne reprezentują dwa zupełnie inne światy.
- To pierwsze jest odporne na brak porządku danych, ale rośnie liniowo wraz z ich liczbą.
- To drugie jest niezwykle szybkie, ale wymaga uprzedniego sortowania, które samo w sobie ma koszt obliczeniowy.

114

Porównanie metod wyszukiwania

Dlatego wybór metody zależy od kontekstu:

- Jeśli mamy pojedyncze wyszukiwanie w nieposortowanej tablicy — linearne.
- Jeśli wykonujemy setki tysięcy zapytań — binarne jest bezkonkurencyjne.

W praktyce często najpierw sortujemy dane, a następnie wykonujemy wiele operacji wyszukiwania, co daje gigantyczne oszczędności czasu.

115

Czym jest sortowanie?

Sortowanie to proces uporządkowywania danych według określonego kryterium. Może to być porządek rosnący, malejący, alfabetyczny, chronologiczny, a nawet leksykograficzny dla złożonych obiektów.

Znaczenie sortowania wykracza jednak daleko poza samo ułożenie danych. Posortowane dane stają się łatwiejsze do przeszukiwania, grupowania, analizowania czy kompresowania.

Wielu algorytmów nie dałoby się efektywnie zrealizować bez wcześniejszego uporządkowania danych, dlatego sortowanie jest integralnym elementem niemal wszystkich systemów informatycznych.

116

Dlaczego sortowanie jest kluczowe?

Sortowanie jest jedną z najczęściej wykonywanych operacji w całej informatyce. Znajdziemy je w bazach danych, wyszukiwarkach internetowych, aplikacjach finansowych, kryptografii, a nawet w systemach operacyjnych zarządzających procesami i pamięcią.

Dane uporządkowane są po prostu łatwiejsze w użyciu. Sortowanie umożliwia stosowanie szybszych metod wyszukiwania, ułatwia tworzenie indeksów, minimalizuje koszty agregacji i analizy. W rezultacie jest absolutnie fundamentalne w projektowaniu efektywnych systemów przetwarzania informacji. 117

Klasyfikacje algorytmów sortowania

Algorytmy sortowania możemy klasyfikować na wiele sposobów i każda z tych klasyfikacji pozwala lepiej zrozumieć ich zastosowanie. Główny podział dotyczy metody działania: sortowania *przez porównania* oraz *bez porównań*.

Sortowania oparte na porównaniach analizują relacje między elementami, ich złożoność ma dolne ograniczenie $\Omega(n \log n)$. Sortowania nieporównawcze, takie jak radix sort, wykorzystują dodatkową wiedzę o strukturze danych, dzięki czemu mogą działać w czasie liniowym.

Dolne ograniczenie sortowania

Dlaczego wiele algorytmów sortowania ma tę samą złożoność $O(n \log n)$? Odpowiedź jest prosta: istnieje matematyczny dowód, że **nie da się posortować danych szybciej**, jeśli porównujemy elementy.

Wynika to z faktu, że każde porównanie zmniejsza liczbę możliwych permutacji danych i dopiero po wykonaniu $\log_2(n!)$ porównań możemy być pewni poprawnej kolejności.

To prowadzi do ścisłego dolnego ograniczenia $\Omega(n \log n)$. Dlatego mergesort i heapsort są optymalne, a inne algorytmy nie mogą być asymptotycznie szybsze, chyba że wykorzystują dodatkowe informacje o danych, jak robi to radix sort.

119

Bubble Sort

Bubble sort to klasyczny przykład algorytmu, który jest łatwy do zrozumienia, ale nieefektywny dla większych zbiorów danych. Działa poprzez wielokrotne „przechodzenie” przez tablicę i wymienianie par elementów, które są w złej kolejności. Można powiedzieć, że największe elementy „wypływają na powierzchnię” tablicy jak pęcherzyki powietrza — stąd nazwa.

120

Bubble Sort

Chociaż bubble sort jest bardzo prosty, jego główną wadą jest złożoność $O(n^2)$, która czyni go niepraktycznym dla dużych zbiorów. W celach dydaktycznych świetnie pokazuje podstawy porównywania i zamiany elementów, lecz w praktyce ustępuje wszystkim innym metodom.

121

Bubble Sort

BubbleSort(A):

for i = 1 to n-1:

for j = 1 to n-i:

if A[j] > A[j+1]:

zamień A[j] z A[j+1]

122

Bubble Sort

Pseudokod ten przedstawia klasyczną wersję bubble sorta, w której przy każdej iteracji największy element wędruje na swoje właściwe miejsce z prawej strony. Warto zauważyć, że liczba iteracji jest zmniejszana w każdej kolejnej rundzie, ponieważ ostatnie elementy tablicy są już uporządkowane.

Pomimo tego optymalnego mechanizmu, algorytm nadal wykonuje znaczną liczbę operacji porównań i zamian, co powoduje jego słabą efektywność.

123

Bubble Sort

Analiza bubble sort potwierdza jego nieefektywność: nawet w wersji „ulepszonej” z dodatkową flagą wykrywającą brak zamian, najgorszy przypadek pozostaje kwadratowy.

Zaletą jest jednak stabilność, elementy o takich samych wartościach zachowują swoją kolejność.

Bubble sort nadal pojawia się w literaturze i dydaktyce, ponieważ jest bardzo łatwy do wizualizacji oraz rozumienia, ale w zastosowaniach praktycznych jest niemal zawsze zastępowany przez szybsze metody.

124

Sortowanie przez wstawianie

Insertion sort jest algorytmem, który w naturalny sposób naśladuje sposób, w jaki człowiek sortuje karty trzymane w ręku. Zaczynamy od niewielkiej, uporządkowanej części zbioru, a następnie każdy nowy element „wstawiamy” w odpowiednie miejsce.

Chociaż w najgorszym przypadku zawiera pętlę zagnieżdżoną i osiąga $O(n^2)$, to jednak jest niezwykle wydajne dla danych częściowo uporządkowanych.

W wielu systemach operacyjnych i bibliotekach standardowych insertion sort jest używany jako część bardziej złożonych algorytmów, np. quicksort dla bardzo krótkich podtablic.

125

Sortowanie przez wstawianie

InsertionSort(A):

for i = 2 to n:

key = A[i]

j = i - 1

while j > 0 and A[j] > key:

A[j+1] = A[j]

j--

A[j+1] = key

126

Sortowanie przez wstawianie

Ten pseudokod prezentuje działanie algorytmu krok po kroku: wybieramy element, przesuwamy większe wartości w prawo, a następnie wstawiamy element we właściwe miejsce.

Główną siłą algorytmu jest jego zdolność do błyskawicznego działania na danych prawie posortowanych, każde porównanie od razu prowadzi do pozytywnego wyniku.

127

Sortowanie przez wstawianie

Insertion sort posiada tę wyjątkową cechę, że dla danych prawie posortowanych działa niemal liniowo. To czyni go algorytmem idealnym wtedy, gdy spodziewamy się elementów dodawanych w kolejności rosnącej lub lekko zaburzonej.

W najgorszym przypadku, gdy dane są uporządkowane odwrotnie, konieczne jest przesuwanie każdego elementu przez cały zbiór, wówczas złożoność rośnie do $O(n^2)$.

Algorytm jest stabilny i wykonuje minimalną liczbę zbędnych operacji, co czyni go bardzo „miękkim” dla danych.

128

Sortowanie przez wybór

Selection sort to jedna z najprostszych metod sortowania, która koncentruje się na znajdowaniu najmniejszego elementu i umieszczaniu go na początkowej pozycji.

W przeciwieństwie do bubble sort minimalizuje liczbę zamian, bowiem wykonuje dokładnie $n-1$ zamian, co może być przydatne w systemach, w których operacje zamiany są kosztowne.

Niestety liczba porównań pozostaje kwadratowa, niezależnie od struktury danych.

129

Sortowanie przez wybór

SelectionSort(A):

for i = 1 to n:

minIndex = i

for j = i+1 to n:

if A[j] < A[minIndex]:

minIndex = j

zamień A[i] z A[minIndex]

130

Sortowanie przez wybór

Algorytm zawsze przegląda wszystkie pozostałe elementy w poszukiwaniu najmniejszego z nich, niezależnie od tego, czy dane są już częściowo uporządkowane. Oznacza to, że w każdej iteracji wykonuje pełne przejście po nieposortowanej części tablicy. Dopiero po zakończeniu tej analizy następuje pojedyncza zamiana elementów, co odróżnia go od innych metod wykonujących wiele zamian w trakcie działania.

Zaletą takiego podejścia jest jego duża przewidywalność oraz prostota analizy. Bez względu na początkowy układ danych algorytm wykonuje zawsze tę samą liczbę iteracji i porównań, co pozwala łatwo określić jego złożoność czasową. Dzięki temu jego działanie jest stabilne pod względem liczby operacji, co bywa korzystne w sytuacjach wymagających deterministycznego czasu wykonania.

131

Sortowanie przez wybór

Selection sort jest niestabilny i posiada złożoność $O(n^2)$, co czyni go nieopłacalnym w większości praktycznych scenariuszy.

Jest jednak bardzo prosty, a minimalna liczba zamian sprawia, że jest dobrym przykładem algorytmu pokazującego kompromisy projektowe, a mianowicie oszczędność na zamianach kosztem większej liczby porównań.

132

Sortowanie przez scalanie

Sortowanie przez scalanie (Merge Sort) jest jednym z najważniejszych algorytmów sortowania, stanowiących klasyczny przykład strategii *dziel i zwyciężaj*. Cała jego siła wynika z tego, że ma bardzo dobrze zdefiniowaną strukturę rekurencyjną: tablica jest dzielona na dwie połowy, obie części są sortowane niezależnie, a następnie łączone w jedną całość.

Takie podejście gwarantuje stabilny czas działania, bez względu na to, czy dane są posortowane, odwrócone, czy losowe, zawsze wykonanych zostanie $n \log n$ operacji.

133

Sortowanie przez scalanie

Scalanie dwóch posortowanych list jest szczególnie efektywne: wystarczy w każdej iteracji porównać pierwsze elementy obu list i wybrać mniejszy. Z tego powodu Merge Sort doskonale radzi sobie w strukturach, w których łatwo uzyskać dostęp sekwencyjny.

Wadą jest konieczność użycia dodatkowej pamięci. Algorytm musi przechowywać tymczasowe kopie podtablic, co bywa istotnym ograniczeniem w systemach o ograniczonej pamięci lub przetwarzających bardzo duże ilości danych.

134

Sortowanie przez scalanie — pseudokod

MERGE-SORT(tablica)

jeśli długość(tablica) ≤ 1

zwróć tablica

środek $\leftarrow$ długość(tablica) / 2

lewa $\leftarrow$ podtablica(tablica, 0, środek)

prawa $\leftarrow$ podtablica(tablica, środek, długość(tablica))

lewa_posortowana $\leftarrow$ MERGE-SORT(lewa)

prawa_posortowana $\leftarrow$ MERGE-SORT(prawa)

zwróć MERGE(lewa_posortowana, prawa_posortowana)

135

Sortowanie przez scalanie

Pseudokod Merge Sort dzieli algorytm na dwa zasadnicze etapy: rekurencyjne dzielenie tablicy na coraz mniejsze fragmenty oraz scalanie ich w jedną, posortowaną całość.

Najważniejszą częścią algorytmu jest funkcja *Merge*, która bierze dwie posortowane sekwencje i „zszywa” je w jedną większą sekwencję, zachowując porządek.

Proces scalania jest wyjątkowo elegancki: porównujemy elementy dwóch list i wybieramy ten mniejszy, przesuwając odpowiedni wskaźnik. Dzięki temu scalanie dwóch części o łącznej długości n trwa dokładnie $O(n)$.

136

Sortowanie przez scalanie

Cała idea Merge Sort opiera się na tym, że skomplikowany problem sortowania rozbijamy na wiele prostych, lokalnych decyzji scalania – a te można wykonywać bardzo efektywnie.

Analiza Merge Sort jest jedną z najprostszych, ale zarazem najładniejszych analiz w algorytmice. Dzielimy tablicę $\log n$ razy, ponieważ każda rekurencja dzieli ją na pół. Na każdym poziomie wykonujemy scalanie, które obejmuje wszystkie n elementów. Łączna liczba operacji daje więc:

$$T(n) = n \log n,$$

co stanowi teoretycznie optymalny wynik dla sortowania przez porównania. 137

Sortowanie przez scalanie

Merge Sort jest stabilny, co oznacza, że nie zmienia kolejności elementów równych. Własność ta jest szczególnie przydatna przy sortowaniu danych rekordowych, gdzie kolejność może mieć znaczenie semantyczne.

Jedyną poważną wadą jest konieczność wykorzystania pamięci pomocniczej rzędu $O(n)$. To sprawia, że Merge Sort rzadziej stosuje się tam, gdzie pamięć jest krytyczna, mimo że jest to algorytm niezwykle przewidywalny i niezawodny.

138

Quicksort

Quicksort jest prawdopodobnie najbardziej znanym i najczęściej stosowanym algorytmem sortowania. Jego niezwykła popularność wynika z doskonałego kompromisu między efektywnością, prostotą a praktyczną wydajnością. Podstawą działania Quicksorta jest wybór elementu zwanego pivotem.

Pivot dzieli tablicę na dwie części: te elementy, które są mniejsze, oraz te, które są większe. Następnie każda z części jest sortowana rekurencyjnie. Najważniejszą cechą tego algorytmu jest to, że działa *in-place*, czyli nie wymaga dużych dodatkowych struktur pamięciowych. Jest to istotna przewaga nad Merge Sortem. 139

Quicksort

W praktyce Quicksort okazuje się niezwykle szybki — większość bibliotek standardowych języków programowania stosuje właśnie jego odmiany, zwłaszcza w wersjach optymalizowanych i hybrydowych.

Pseudokod Quicksorta opiera się na funkcji *Partition*, która odpowiada za podział tablicy wokół pivotu. W zależności od tego, jakie elementy znajdują się przed pivotem i za nim, tworzą się dwie podtablice, które są następnie sortowane.

140

Quicksort

QUICKSORT(A, lewy, prawy)

jeśli lewy < prawy

pivot_index $\leftarrow$ PARTITION(A, lewy, prawy)

QUICKSORT(A, lewy, pivot_index - 1)

QUICKSORT(A, pivot_index + 1, prawy)

141

Quicksort

PARTITION(A, lewy, prawy)

```
pivot ← A[prawy]    for j ← lewy to prawy - 1 do
                      if A[j] ≤ pivot then
                        i ← i + 1
                        swap A[i] with A[j]

swap A[i + 1] with A[prawy]
return i + 1
```

142

Quicksort

Kluczowe jest to, że nie sortujemy elementów bezpośrednio, a jedynie umieszczamy pivot na swojej poprawnej pozycji, zaś pozostałe elementy reorganizujemy tak, aby znajdowały się po właściwych stronach.

Dzięki temu Quicksort sortuje dane lokalnie i pozostawia je w stanie bliskim uporządkowania już po jednym przejściu Partition. Następne rekurencje jedynie „dopieczętowują” porządek.

Funkcja Partition decyduje o wydajności Quicksorta, ponieważ jej zadaniem jest takie przestawienie elementów, by stanowiły dwa logiczne przedziały: mniejszych i większych od pivotu.

143

Quicksort

Najbardziej klasyczna wersja Partition wybiera pivot z prawego końca tablicy i przesuwą elementy mniejsze na lewo, a większe na prawo, używając wskaźników i oraz j .

Działanie Partition jest bardzo lekkie obliczeniowo — wykonuje jedynie prostą iterację i kilka zamian. To sprawia, że średnia wydajność Quicksorta jest tak wysoka.

Jednocześnie, jeśli pivot zostanie wybrany niefortunnie np. jeśli tablica jest już posortowana, a pivot wybieramy jako ostatni element, to algorytm może się zdegradować do $O(n^2)$. Dlatego w praktycznych implementacjach stosuje się bardziej przemyślane strategie wyboru pivotu, np. metodę mediany trzech.

144

Quicksort

W analizie Quicksorta wyróżniamy dwa przypadki:

- w **średnim przypadku**, który jest najbardziej istotny w praktyce, pivot zazwyczaj dzieli tablicę na względnie równe części. Wówczas rekurencyjne wywołania mają rozmiar $n/2$, a całkowity koszt daje $O(n \log n)$.
- w **najgorszym przypadku**, pivot dzieli tablicę bardzo nierówno np. 1 element + reszta. Wtedy powstaje głęboka rekurencja i koszt rośnie do $O(n^2)$.

145

Quicksort

Pomimo ryzyka najgorszego przypadku, Quicksort pozostaje najszybszym sortowaniem w praktyce, zwłaszcza dzięki optymalizacjom zmniejszającym jego ryzyko.

Złożoność pamięciowa jest bardzo dobra: $O(\log n)$ wynikające z głębokości stosu rekurencyjnego, co czyni go niezwykle atrakcyjnym w środowiskach o ograniczonych zasobach.

146

Heap Sort

Heap Sort wykorzystuje kopiec, specjalną strukturę danych, która pozwala w czasie $O(\log n)$ znajdować największy element.

Jego działanie polega na:

1. Zbudowaniu kopca z danych wejściowych,
2. Powtarzaniem „wyciąganiu” największego elementu i umieszczaniu go na końcu tablicy,
3. Naprawie kopca (heapify) po każdej takiej operacji.

147

Heap Sort

Wynikiem jest tablica posortowana rosnąco. Heap Sort ma tę przewagę, że zawsze działa w czasie $O(n \log n)$, niezależnie od struktury danych, jest zatem bardzo przewidywalny. Jednocześnie jest algorytmem niestabilnym i zwykle wolniejszym od Quicksorta pod względem stałych ukrytych w notacji O . Dlatego w praktyce jest stosowany głównie wtedy, gdy potrzebujemy gwarancji najgorszego przypadku.

148

Heap Sort

Proces Heap Sortu można podzielić na dwie fazy:

- budowę kopca
- właściwe sortowanie.

Budowanie kopca odbywa się w czasie liniowym, ponieważ operacje „naprawiania” zachodzą w sposób zoptymalizowany od najniższych poziomów struktury.

149

Heap Sort

W drugiej fazie kopiec pełni rolę struktury wyciągającej największy element. Każde wyciągnięcie zaburza strukturę, więc trzeba ją przywrócić, ale ponieważ kopiec ma logarytmiczną wysokość, koszt pojedynczej naprawy wynosi $O(\log n)$. Powtarzamy tę operację n razy, co daje łączną złożoność $O(n \log n)$.

Ten mechanizm łączenia struktury danych z procesem sortowania czyni Heap Sort eleganckim, choć nieco bardziej złożonym koncepcyjnie.

150

HEAP-SORT(tablica)

$n \leftarrow \text{długość}(\text{tablica})$

// 1. Zbuduj kopiec (max-heap)

dla i od $\lfloor n/2 \rfloor - 1$ do 0 w dół

 HEAPIFY(tablica, n , i)

// 2. Pobierz elementy z kopca

dla i od $n - 1$ do 1 w dół

 zamień $\text{tablica}[0]$ z $\text{tablica}[i]$

 HEAPIFY(tablica, i , 0)

151

```
HEAPIFY(tablica, rozmiar, i)
```

```
    największy ← i
```

```
    lewy ← 2*i + 1
```

```
    prawy ← 2*i + 2
```

```
    jeśli lewy < rozmiar oraz  
    tablica[lewy] > tablica[największy]
```

```
        największy ← lewy
```

```
    jeśli prawy < rozmiar oraz
```

```
        tablica[prawy] > tablica[największy]
```

```
            największy ← prawy
```

```
    jeśli największy ≠ i
```

```
        zamień tablica[i] z
```

```
        tablica[największy]
```

```
        HEAPIFY(tablica, rozmiar,  
        największy)
```

152

Z punktu widzenia teoretycznego Heap Sort jest jednym z najbardziej stabilnych algorytmów, nie w sensie stabilności względem kolejności elementów równych, ale w sensie przewidywalności czasu działania.

Bez względu na dane wejściowe, jego czas działania zawsze wynosi $O(n \log n)$, co daje pewność w systemach krytycznych czasowo. Niestety stabilizacja struktury kopca w każdej iteracji powoduje, że algorytm często przegrywa wydajnościowo z Quicksortem.

Ponadto sortowanie odbywa się w miejscu, ale narusza wewnętrzną strukturę, co powoduje, że nie jest stabilne. Mimo to Heap Sort jest niezwykle ważnym algorytmem z punktu widzenia analizy teoretycznej.

153

Sortowanie kubełkowe

Sortowanie kubełkowe (Bucket Sort) jest algorytmem, który zakłada, że dane są rozłożone równomiernie w pewnym zakresie.

Polega na podziale tego zakresu na tzw. kubełki (ang. buckets), a następnie umieszczaniu elementów w odpowiednich kubełkach.

Po wypełnieniu kubełków każdy z nich sortujemy, najczęściej prostą metodą, np. insertion sort, ponieważ kubełki są niewielkie.

154

Sortowanie kubełkowe

Na koniec łączymy zawartość kubełków w jedną listę. Technika ta wykorzystuje strukturę danych do przyspieszenia działania i może osiągać złożoność liniową, jeśli rozkład danych jest sprzyjający.

Bucket Sort jest chętnie stosowany w zadaniach statystycznych, obliczeniach naukowych i wszędzie tam, gdzie dane mają przewidywalny rozkład.

155

Radix Sort

Radix Sort jest jednym z nielicznych algorytmów sortujących, które mogą działać w czasie rzeczywistie liniowym, pod warunkiem, że liczby mają ograniczoną długość.

Algorytm sortuje dane „cyfra po cyfrze”, najpierw według najmniej znaczącej, a następnie według kolejnych. Sortowanie na każdej pozycji cyfr odbywa się za pomocą stabilnego algorytmu sortującego, np. counting sort. Dzięki temu wartości są idealnie „układane” według cyfr, co sprawia, że końcowy wynik jest całkowicie posortowany.

156

Radix Sort

Radix Sort jest intensywnie wykorzystywany w systemach przetwarzających duże liczby, w przetwarzaniu tekstów, a także w bazach danych.

Analiza Radix Sortu pokazuje, że złożoność algorytmu zależy bardziej od długości wartości (np. liczby cyfr, bajtów, znaków) niż od liczby elementów.

157

Radix Sort

Jeśli mamy n elementów o d cyfrach, oraz każdą cyfrę możemy sortować w czasie k , to łączny koszt wynosi $O(d \cdot (n + k))$.

W przypadku stałej liczby cyfr — np. 32-bitowych liczb całkowitych — d i k są stałe, więc całość upraszcza się do $O(n)$.

To czyni Radix Sort jedynym praktycznym algorytmem o złożoności liniowej w wielu realnych zastosowaniach.

158

Porównanie sortowań

Porównując algorytmy sortowania, możemy zauważyć, że żaden z nich nie jest idealny we wszystkich sytuacjach i zastosowaniach. Wybór odpowiedniego algorytmu zależy od wielu czynników, takich jak rozmiar danych, ich początkowe uporządkowanie, dostępna pamięć czy wymagania dotyczące stabilności sortowania.

Algorytmy proste, takie jak sortowanie bąbelkowe czy przez wstawianie, są bardzo intuicyjne i świetnie sprawdzają się w celach edukacyjnych, ponieważ łatwo zrozumieć ich działanie krok po kroku.

159

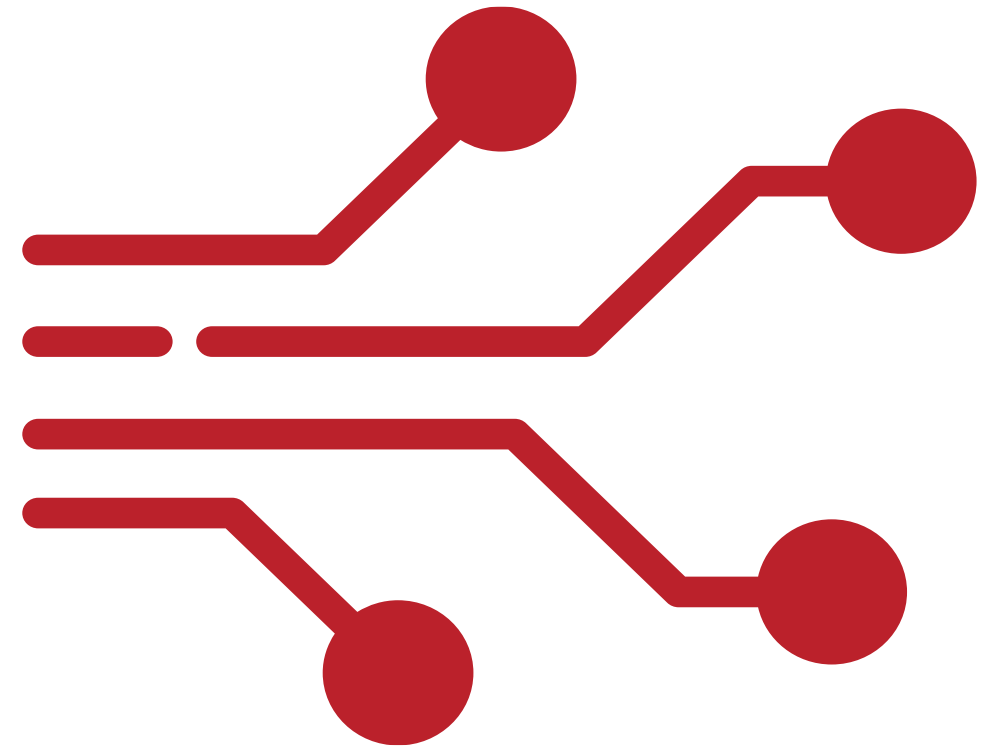
Porównanie sortowań

Algorytomy zaawansowane, takie jak Merge Sort i Quicksort, oferują świetne kompromisy między wydajnością, pamięcią i stabilnością. Sortowania liniowe, takie jak Radix Sort, są nieocenione wtedy, gdy dane mają bardzo szczególne właściwości.

Wybór metody często zależy od trzech kluczowych pytań: jak duże są dane, jaki mają charakter i czy stabilność sortowania jest wymagana.

160

Algorytmy rekurencyjne. Rozwiązywanie niektórych równań rekurencyjnych



161

Wstęp

W ramach tego wykładu wprowadzamy pełną perspektywę na algorytmy rekurencyjne oraz metody rozwiązywania równań rekurencyjnych pojawiających się w analizie algorytmów. Najpierw przypominamy ideę rekurencji, jej strukturę i sposób myślenia o problemach definiowanych poprzez odwołania do wersji uproszczonych. Następnie przechodzimy przez przykłady klasycznych funkcji rekurencyjnych i analizujemy ich działanie, zarówno od strony intuicyjnej, jak i formalnej.

162

Wstęp

W drugiej części wykładu wprowadzamy równania rekurencyjne opisujące czas wykonywania algorytmów opartych na strategii dziel i zwyciężaj.

Wprowadzamy techniki ich rozwiązywania:

- rozwijanie rekurencji,
- metodę podstawień
- twierdzenie Mastera, które umożliwia szybkie i ogólne określenie asymptotycznego zachowania $T(n)$.

163

Wprowadzenie do rekurencji

Rekurencja jest sposobem definiowania rozwiązania poprzez odwołanie do tego samego problemu, ale o mniejszej skali. Polega na tym, że funkcja, zamiast rozwiązywać problem w całości, redukuje go do prostszej wersji i deleguje dalsze obliczenia do swojego własnego wywołania. Dzięki temu możemy pracować nad zadaniami, które mają wyraźną strukturę samopodobną, typową dla drzew, przeszukiwań, podziałów zbiorów i wielu zadań matematycznych.

Fundamentem poprawności jest przypadek bazowy, który przerywa dalszą redukcję i dostarcza minimalnej odpowiedzi. Bez niego rekurencja prowadziłaby do nieskończonego łańcucha wywołań. 164

Intuicja działania funkcji rekurencyjnych

Działanie funkcji rekurencyjnej można zobrazować jako stopniowe rozbijanie problemu na coraz mniejsze części, aż osiągnięta zostanie struktura tak prosta, że jej rozwiązanie nie wymaga dalszych wywołań.

Każde wywołanie jest w istocie węzłem w drzewie wywołań, w którym potomkami są kolejne, coraz mniejsze podproblemy.

165

Intuicja działania funkcji rekurencyjnych

Drzewo to rozwija się w dół, aż wszystkie gałęzie trafią do przypadków bazowych. Następnie rozwiązania są „odwijane”, czyli zwracane w górę w kierunku pierwotnego wywołania.

To właśnie dlatego proces rekurencyjny bywa intuicyjny: *zamiast opisywać każdy krok po kolei, określamy jedynie sposób redukcji problemu i sposób odbudowy rozwiązania na końcu.*

166

Rekurencja vs iteracja

Wiele problemów można zapisać zarówno w formie rekurencyjnej, jak i iteracyjnej, jednak oba podejścia reprezentują odmienne sposoby myślenia. Iteracja opisuje rozwiązanie przez wskazanie kolejnych operacji wykonywanych w pętli.

Rekurencja natomiast koncentruje się na zdefiniowaniu zależności między rozwiązaniem dla n oraz rozwiązaniami dla mniejszych wartości. Rekurencyjna definicja bywa często prostsza, bardziej matematyczna i elegancka, ale wymaga pamięci stosu, co wpływa na efektywność.

Z kolei iteracja eliminuje narzut wywołań funkcji, lecz bywa mniej przejrzysta. Dobór podejścia zależy więc od natury problemu oraz wymagań pamięciowych i wydajnościowych.

Struktura funkcji rekurencyjnej

Każda funkcja rekurencyjna posiada dwa elementy konieczne dla poprawnego działania. Pierwszym z nich jest przypadek bazowy, czyli najmniejsza wersja problemu, której rozwiązanie znamy od razu, na przykład warunki $n = 0$ lub $n = 1$ w klasycznych funkcjach matematycznych.

Drugim elementem jest przypadek ogólny, w którym funkcja wyraża rozwiązanie dla argumentu n przez rozwiązania dla argumentów mniejszych, takich jak $n - 1$ lub $n/2$. Tę zależność zapisujemy zwykle jako równanie rekurencyjne.

168

Struktura funkcji rekurencyjnej

Przykładowo, dla funkcji wykonującej jedną operację i następnie wywołującej samą siebie na argumencie $n - 1$ zapisujemy

$$T(n) = T(n - 1) + 1,$$

a dla algorytmów dzielących dane na połowy równanie przyjmuje postać

$$T(n) = 2T(n/2) + n.$$

To właśnie te równania będą analizowane w drugiej części wykładu.

Warunek bazowy jako fundament poprawności

Warunek bazowy jest najważniejszym elementem każdej funkcji rekurencyjnej, ponieważ to on zatrzymuje proces redukowania problemu. Jest to najmniejsza, nieredukowalna postać zadania, która może zostać rozwiązana bez dalszych wywołań.

Aby rekurencja była poprawna, przypadek bazowy musi być osiągalny po skończonej liczbie kroków. Klasycznym przykładem jest funkcja silni, gdzie dla $n = 0$ wynik jest znany i równy 1.

Jeśli przypadek bazowy jest źle zdefiniowany lub niedostępny, funkcja prowadzi do nieskończonej rekurencji. W praktyce warunek bazowy pełni tę samą rolę, co dolna granica w dowodach indukcyjnych.

Wywołanie rekurencyjne jako proces redukcji problemu

Wywołanie rekurencyjne jest częścią funkcji, w której rozwiązanie dla argumentu n zostaje wyrażone poprzez rozwiązanie dla wartości mniejszej lub prostszej.

Redukcja musi prowadzić w stronę przypadku bazowego, co oznacza, że każdy kolejny krok upraszcza problem. Jeśli rekurencja dotyczy struktur drzewiastych, pojedyncze wywołanie może generować wiele kolejnych.

171

Wywołanie rekurencyjne jako proces redukcji problemu

W algorytmach liniowych, takich jak obliczanie n -tego elementu prostego ciągu, redukcja ma formę

$$n \rightarrow n - 1.$$

Kluczową ideą jest to, aby każda redukcja miała sens logiczny i algebraiczny, a jej efekt pozwalał na odbudowanie finalnego wyniku.

172

Przykład: obliczanie silni i drzewo wywołań

Dobrym przykładem ilustrującym działanie rekurencji jest silnia. Definiujemy ją wzorem:

$$n! = n \cdot (n - 1)! \\ 0! = 1$$

Jeśli wywołamy `factorial(4)`, proces rozwija się w drzewo o postaci [\(następny slajd\)](#)

173

Przykład: obliczanie silni i drzewo wywołań

factorial(4) odwołuje się do factorial(3)
factorial(3) odwołuje się do factorial(2)
factorial(2) odwołuje się do factorial(1)
factorial(1) odwołuje się do factorial(0)
factorial(0) zwraca 1

Po osiągnięciu 1 funkcja zaczyna się odwijać i kolejne wyniki są mnożone:

$1 \rightarrow 1 \rightarrow 2 \rightarrow 6 \rightarrow 24$

174

Przykład: Fibonacciego i rekurencja drzewiasta

Fibonacci jest klasycznym przykładem rekurencji, która tworzy drzewo o bardzo szybko rosnącej liczbie węzłów. Definicja jest następująca:

$$F(0) = 0 ;$$

$$F(1) = 1;$$

$$F(n) = F(n - 1) + F(n - 2)$$

175

Przykład: Fibonacci i rekurencja drzewiasta

Wywołanie funkcji $\text{fib}(n)$ prowadzi do dwóch kolejnych wywołań, a każde z nich generuje kolejne dwie gałęzie. Drzewo rośnie wykładniczo, ponieważ wartości $\text{fib}(n - 1)$ i $\text{fib}(n - 2)$ są obliczane wielokrotnie. Dlatego czas wykonania funkcji spełnia równanie rekurencyjne

$$T(n) = T(n - 1) + T(n - 2) + 1$$

którego rozwiązaniem jest funkcja wykładnicza. Ten przykład stanowi podstawę do późniejszej dyskusji o memoizacji oraz o tym, jak wykładnicze rekurencje można przekształcić w liniowe dzięki zapamiętywaniu wyników.

176

Pseudokod prostych funkcji rekurencyjnych

Pseudokod rekurencyjny jest zwykle bardzo krótki i prosty, ponieważ duża część złożoności ukrywa się za wywołaniem samej funkcji. Przykład silni:

```
function factorial(n):  
    jeśli  $n = 0$   
        zwróć 1  
    w przeciwnym razie  
        zwróć  $n \times \text{factorial}(n - 1)$ 
```

177

Pseudokod prostych funkcji rekurencyjnych

Przykład ciągu Fibonacciego:

```
function fib(n):  
    jeśli  $n < 2$   
        zwróć  $n$   
    zwróć  $\text{fib}(n - 1) + \text{fib}(n - 2)$ 
```

Choć kod wygląda elegancko, jego złożoność wykładnicza pokazuje, że rekurencja nie zawsze jest efektywna. W kolejnych slajdach przejdziemy od pisania funkcji do interpretowania ich kosztów obliczeniowych.

178

Wprowadzenie do analizy równań rekurencyjnych

Równania rekurencyjne opisują zależność czasu działania algorytmu od czasu wymaganych dla mniejszych podproblemów. Stanowią one formalny sposób analizy funkcji rekurencyjnych i są podstawą oceny algorytmów opartych na strategii dziel i zwyciężaj.

Dla funkcji takich jak factorial czy prosty fibonacci równania te przyjmują postać

$$T(n) = T(n - 1) + 1$$

lub

179

Wprowadzenie do analizy równań rekurencyjnych

$$T(n) = T(n - 1) + T(n - 2),$$

natomiast w algorytmach sortowania przez scalanie lub przeszukiwania binarnego pojawiają się zależności typu

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

czy

$$T(n) = T(n / 2) + 1.$$

Zrozumienie sposobu rozwiązywania takich równań pozwala z automatu ocenić czas działania całych klas algorytmów, nawet bez znajomości szczegółów implementacji. 180

Przykład równania liniowego: $T(n) = T(n - 1) + 1$

Jednym z najprostszych równań jest zależność pokazująca, że czas działania dla n zależy od czasu działania dla $n - 1$ powiększonego o jedną jednostkę. Rozwinięcie tej rekurencji prowadzi do następującej postaci:

$$T(n) = T(n - 1) + 1$$

$$T(n - 1) = T(n - 2) + 1$$

...

$$T(1) = T(0) + 1$$

181

Przykład równania liniowego: $T(n) = T(n - 1) + 1$

Po zsumowaniu wszystkich równań otrzymujemy

$$T(n) = T(0) + n.$$

Oznacza to, że złożoność rośnie liniowo wraz z wielkością problemu. Takie zależności są charakterystyczne dla funkcji rekurencyjnych realizujących proste przejścia przez kolejne wartości n , jak w przypadku silni lub najprostszego wyszukiwania elementu w strukturze rekurencyjnej.

182

Przykład równania: $T(n) = T(n - 1) + n$

W tym przykładzie oprócz redukcji problemu pojawia się dodatkowy koszt równy n . Rozwinięcie rekurencji prowadzi do wyrażenia:

$$T(n) = T(n - 1) + n$$

$$T(n - 1) = T(n - 2) + (n - 1)$$

...

$$T(1) = T(0) + 1$$

183

Przykład równania: $T(n) = T(n - 1) + n$

Po zsumowaniu otrzymujemy

$$T(n) = T(0) + 1 + 2 + \dots + n,$$

a suma ta wynosi $n(n + 1) / 2$. Oznacza to złożoność kwadratową, typową dla algorytmów wykonujących narastający koszt w każdym kroku redukcji. W praktyce pojawia się to przy prostych algorytmach przeglądających wszystkie pary elementów poprzez rekurencyjne skracanie zbioru.

184

Metoda podstawień w rozwiązywaniu rekurencji

Metoda podstawień polega na zgadywaniu przypuszczalnego rozwiązania równania i udowodnieniu jego poprawności przez indukcję. Przykładowo, dla równania $T(n) = T(n - 1) + n$ możemy zaproponować, że $T(n) = \Theta(n^2)$. Następnie sprawdzamy:

$$T(n - 1) \approx (n - 1)^2 = n^2 - 2n + 1$$

$$T(n) = T(n - 1) + n \approx n^2 - 2n + 1 + n = n^2 - n + 1$$

co wskazuje, że $T(n)$ rzeczywiście rośnie jak n^2 . Ta metoda wymaga intuicji, ale jest niezastąpiona w bardziej skomplikowanych równaniach, które trudno rozwijać ręcznie.

185

Metoda rozwijania rekurencji

Metoda rozwijania polega na wielokrotnym podstawianiu rekurencji, aż osiągniemy przypadek bazowy. W przykładzie równania $T(n) = 2T(n/2) + n$ kolejne rozwinięcia dają:

$$\begin{aligned}T(n) &= 2T(n/2) + n \\&= 2(2T(n/4) + n/2) + n \\&= 4T(n/4) + 2n \\&= 8T(n/8) + 3n \\&\dots \\&= 2^k T(n / 2^k) + k n\end{aligned}$$

186

Metoda rozwijania rekurencji

Gdy

$$n / 2^k = 1$$

otrzymujemy $2^k = n$, a więc $k = \log_2 n$. Wstawiając:

$$T(n) = n T(1) + n \log_2 n$$

co prowadzi do złożoności $\Theta(n \log n)$.

Ta metoda jest podstawą przy analizie algorytmów takich jak mergesort.

187

Wprowadzenie do równań typu divide & conquer

Równania typu divide and conquer opisują algorytmy, które dzielą problem na mniejsze części, rozwiązują je rekurencyjnie i następnie łączą wyniki. Najczęściej przyjmują postać:

$$T(n) = a T(n / b) + f(n)$$

gdzie a oznacza liczbę podproblemów, b sposób ich zmniejszenia, a $f(n)$ koszt operacji niezależnych od rekurencji. W zależności od relacji między $f(n)$ a $n^{\log_b a}$ otrzymujemy różne zachowania funkcji $T(n)$. Te trzy przypadki analizuje twierdzenie Mastera.

188

Równanie $T(n) = 2T(n/2) + 1$

Jest to rekurencja odpowiadająca algorytmom, w których po podzieleniu problemu na dwie części ich scalanie zajmuje stały czas. Rozwinięcie pokazuje:

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 4T(n/4) + 1 + 2 \\ &= 8T(n/8) + 1 + 2 + 4 \\ &\dots \\ &= 2^k T(n / 2^k) + (2^k - 1) \end{aligned}$$

189

Równanie $T(n) = 2T(n/2) + 1$

Jak wcześniej, po osiągnięciu

$$n / 2^k = 1$$

otrzymujemy $2^k = n$.

Podstawienie daje tożsamość:

$$T(n) = nT(1) + (n - 1)$$

co oznacza $T(n) = \Theta(n)$.

190

Równanie $T(n) = 2T(n/2) + n$

Rekurencja ta opisuje algorytmy, w których podział jest niedrogi, lecz scalanie wymaga czasu proporcjonalnego do n . Po rozwinięciu:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\dots \\ &= 2^k T(n/2^k) + k n \end{aligned}$$

191

Równanie $T(n) = 2T(n/2) + n$

Ponownie $\log_2 n = k$ daje:

$$T(n) = n T(1) + n \log_2 n$$

Zatem $T(n) = \Theta(n \log n)$.

Jest to rekurencja charakterystyczna dla sortowania przez scalanie.

192

Równanie $T(n) = T(n/2) + 1$

W tej rekurencji problem zmniejsza się o połowę, a koszt dodatkowy jest stały. Rozwinięcie prowadzi do:

$$T(n) = T(n/2) + 1 = T(n/4) + 2 = T(n/8) + 3$$

po k krokach: $T(n) = T(n/2^k) + k$

gdy $n/2^k = 1$, otrzymujemy $k = \log_2 n$. Zatem:

$$T(n) = T(1) + \log_2 n$$

co oznacza $T(n) = \Theta(\log n)$. Jest to fundament wyszukiwania binarnego.

193

Intuicyjne wprowadzenie do twierdzenia Mastera

Twierdzenie Mastera jest narzędziem pozwalającym ocenić asymptotykę równania

$$T(n) = aT(n/b) + f(n)$$

bez konieczności rozwijania rekurencji. Zasada opiera się na porównaniu kosztu rekurencyjnego, czyli $n^{\log_b a}$, z kosztem $f(n)$, który wynika z pracy wykonywanej poza wywołaniami rekurencyjnymi. Jeśli $f(n)$ jest mniejsze, większe lub równoważne temu wyrażeniu, otrzymujemy trzy różne przypadki końcowe. Siła twierdzenia Mastera polega na jego uniwersalności i prostocie stosowania.

194

Master: przypadek pierwszy

W pierwszym przypadku funkcja $f(n)$ rośnie wolniej niż $n^{\log_b a}$.
Formalnie:

$$f(n) = O(n^{\log_b a - \varepsilon}) \text{ dla pewnego } \varepsilon > 0.$$

Wówczas koszt rekurencji dominuje i $T(n)$ ma asymptotykę:

$$T(n) = \Theta(n^{\log_b a}).$$

Oznacza to, że większość pracy wykonują wywołania rekurencyjne, a dodatkowy koszt nie zmienia charakteru wzrostu. Przykładem jest $T(n) = 2T(n/2) + n^{0,4}$.

195

Master: przypadek drugi

Jeśli $f(n)$ rośnie w tym samym tempie co $n^{\log_b a}$, czyli:

$$f(n) = \Theta(n^{\log_b a}),$$

wówczas całość jest równoważna:

$$T(n) = \Theta(n^{\log_b a} \log n).$$

Pojawia się dodatkowy czynnik logarytmiczny, wynikający z równoważnego udziału kosztów rekurencji i kosztów scalania. Przykładem jest mergesort, gdzie $f(n) = n$ oraz $a = 2$, $b = 2$, więc $n \log_2 n$.

196

Master: przypadek trzeci

W trzecim przypadku $f(n)$ rośnie szybciej niż $n^{\log_b a}$. Jeśli:

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ dla pewnego $\varepsilon > 0$
oraz spełniony jest warunek regularności:
 $a f(n/b) \leq c f(n)$ dla pewnego $c < 1$,

to koszt dodatkowy dominuje i:

$$T(n) = \Theta(f(n)).$$

Ta sytuacja pojawia się rzadziej, lecz obejmuje algorytmy, w których scalanie jest kosztowniejsze niż rozwiązywanie podproblemów.

197

Przykład zastosowania twierdzenia Mastera

Weźmy równanie $T(n) = 3T(n/3) + n$. Mamy $a = 3$ i $b = 3$, więc $n^{\log^3 3} = n$. Koszt rekurencji i koszt $f(n)$ są zgodne co do rzędu, co prowadzi do przypadku drugiego twierdzenia Mastera:

$$T(n) = \Theta(n \log n).$$

Jeśli natomiast $T(n) = 4T(n/2) + n^2$, wówczas $n^{\log^2 4} = n^2$, a $f(n) = n^2$, więc również przypadek drugi. Gdybyśmy mieli $T(n) = 2T(n/2) + n^2$, wówczas $n^{\log^2 2} = n$, a n^2 dominuje, więc $T(n) = \Theta(n^2)$.

198

Rekurencje nieliniowe

Nie wszystkie rekurencje mają postać liniową w $T(n)$.
Przykładowo:

$$T(n) = T(n - 1) T(n - 2)$$
$$T(n) = [T(n/2)]^2$$

Takie zależności nie opisują typowych algorytmów, ale pojawiają się w analizach probabilistycznych lub nietypowych problemach. Ich rozwiązywanie wymaga innych technik, czasem zaawansowanej analizy asymptotycznej. W praktyce rzadko spotyka się je w klasycznej algorytmice.

199

Analiza przestrzeni rekurencyjnej

Oprócz czasu działania analizujemy także pamięć, szczególnie głębokość stosu rekurencyjnego. Dla rekurencji liniowej, takiej jak $T(n) = T(n - 1) + 1$, głębokość stosu jest równa n .

Natomiast w rekurencjach typu divide and conquer stos rośnie logarytmicznie, ponieważ każda redukcja zmniejsza problem dwukrotnie.

Te różnice mają duże znaczenie w środowiskach o ograniczonej pamięci.

200

Rekurencja ogonowa i optymalizacja pamięci

Rekurencja ogonowa to taka, w której wywołanie rekurencyjne jest ostatnią instrukcją funkcji. Przykład:

```
function f(n, acc):  
    jeśli n = 0  
        zwróć acc  
    zwróć f(n - 1, acc + n)
```

Taka rekurencja może być optymalizowana przez kompilator poprzez „spłaszczenie” i eliminację stosu. Dzięki temu złożoność pamięciowa redukuje się z $O(n)$ do $O(1)$, co czyni rekurencję równie efektywną jak iterację. 201

Rekurencja wzajemna

Rekurencja wzajemna występuje wtedy, gdy dwie lub więcej funkcji wywołuje się nawzajem. Przykładowo:

$f(n)$ wywołuje $g(n - 1)$
 $g(n)$ wywołuje $f(n - 2)$

Takie rekurencje wymagają szczególnej uwagi przy analizie, ponieważ prowadzą do bardziej złożonych układów równań. Ich złożoność może być liniowa, kwadratowa lub wykładnicza w zależności od wzoru połączeń.

202

Rekurencja drzewiasta

Rekurencja drzewiasta generuje strukturę, w której każde wywołanie tworzy wiele kolejnych. Algorytm fib(n) jest typowym przykładem.

Złożoność rośnie wykładniczo, ponieważ liczba węzłów w drzewie jest proporcjonalna do fib(n), czyli około $\varphi^n / \sqrt{5}$.

Ten typ rekurencji pokazuje, jak ważne jest rozpoznanie nadmiarowych obliczeń i wprowadzenie memoizacji.

203

Memoizacja i dynamiczne przekształcanie rekurencji

Memoizacja polega na zapisywaniu wyników wcześniejszych obliczeń funkcji rekurencyjnej, aby uniknąć powtórnych obliczeń. W przypadku $fib(n)$ zmienia to rekurencję wykładniczą w liniową. Prowadzi to do równania $T(n) = T(n - 1) + 1$, ponieważ każde wywołanie wykonywane jest tylko raz. Takie techniki są fundamentem programowania dynamicznego.

204

Rekurencja w strukturach drzewiastych

Algorytmy działające na drzewach, takie jak obliczanie wysokości drzewa, liczby węzłów lub przeszukiwanie DFS, naturalnie korzystają z rekurencji. Przykład:

```
height(node):  
    jeśli node jest pusty  
        zwróć 0  
    lewa = height(node.left)  
    prawa = height(node.right)  
    zwróć 1 + max(lewa, prawa)
```

205

Rekurencja w strukturach drzewiastych

Rekurencja tej funkcji ma postać

$$T(n) = T(n - 1) + T(n - 2) + 1,$$

gdy drzewo jest niezbalansowane, lub

$$T(n) = 2T(n/2) + 1,$$

gdy drzewo jest idealnie zrównoważone.

Rekurencja w sortowaniu Quicksort

Quicksort wykorzystuje rekurencję w następujący sposób:

$$T(n) = T(k) + T(n - k - 1) + n$$

gdzie k zależy od wyboru pivotu. W najlepszym przypadku

$$k \approx \frac{n}{2} \text{ i } T(n) = \Theta(n \log n).$$

W najgorszym przypadku $k = 0$ i

$$T(n) = T(n - 1) + n = \Theta(n^2).$$

Ta analiza pokazuje, że rekurencja może mieć różne oblicza w zależności od rozkładu danych.

207

Rekurencja w sortowaniu Mergesort

Mergesort dzieli problem idealnie na pół, a scalanie wymaga liniowego czasu. Rekurencja:

$$T(n) = 2T(n/2) + n$$

jak wcześniej pokazano, prowadzi do $T(n) = \Theta(n \log n)$. To jeden z najbardziej stabilnych przykładów divide and conquer.

208

Rekurencja jako model obliczeń

Wiele algorytmów opisuje się rekurencyjnie, nawet jeśli implementacja jest iteracyjna. Rekurencja pozwala opisać złożoność algorytmu, wprowadzając relacje między wielkościami podproblemów. Nawet iteracyjne algorytmy sortowania można zapisać w formie rekurencyjnych zależności czasu.

209

Wpływ struktury danych na rekurencję

Struktura danych może narzucać sposób rekurencji. Tablice często prowadzą do rekurencji binarnych lub liniowych, natomiast drzewa w naturalny sposób wymuszają rekurencje drzewiaste. Dobór struktury danych wpływa na stopień rozgałęzienia i głębokość stosu.

210

Rekurencje w analizie algorytmów geometrycznych

Algorytmy takie jak najbliższa para punktów, triangulacja czy obliczanie otoczki wypukłej opierają się na zaawansowanych rekurencjach divide and conquer. Często mają postać:

$$T(n) = 2T(n/2) + O(n)$$

lub

$$T(n) = T(n/2) + T(n/3) + O(n)$$

co wymaga bardziej złożonej analizy.

211

Dlaczego czasem usuwamy rekurencję

Choć rekurencja jest naturalnym i eleganckim sposobem opisu wielu algorytmów, w praktyce pojawia się konieczność jej eliminacji. Powodem są ograniczenia związane ze stosowaniem stosu wywołań, które prowadzą do przepełnienia przy dużych wartościach n , szczególnie w rekurencjach liniowych.

Dodatkowo wywołanie funkcji niesie koszt administracyjny, który bywa istotny w algorytmach bardzo często wykonywanych. Usunięcie rekurencji pozwala na optymalizację pamięci, zwiększenie kontroli nad przebiegiem obliczeń oraz dostosowanie algorytmu do warunków, w których rekurencja jest niewskazana. 212

Eliminacja rekurencji liniowej poprzez pętlę iteracyjną

Najprostszym przypadkiem jest rekurencja liniowa, w której funkcja odwołuje się do argumentu $n - 1$. Silnia stanowi tu klasyczny przykład. Funkcja $\text{factorial}(n)$ może zostać zapisana iteracyjnie poprzez wprowadzenie zmiennej akumulującej wynik.

Zamiast wielokrotnych wywołań $\text{factorial}(n - 1)$, algorytm przechodzi kolejno przez wartości od 1 do n . W efekcie otrzymujemy tę samą logikę, lecz bez budowania stosu. Dokładnie ten sam proces można zastosować do ciągów definiowanych rekurencyjnie.

213

Przykład konwersji rekurencyjnej silni na wersję iteracyjną

Rekurencyjna postać silni:

```
function factorial(n):
```

```
    jeśli  $n = 0$ 
```

```
        zwróć 1
```

```
    zwróć  $n \times \text{factorial}(n - 1)$ 
```

Można ją przekształcić do wersji iteracyjnej:

```
function factorial_iter(n):
```

```
    wynik = 1
```

```
    dla i od 1 do n
```

```
        wynik = wynik  $\times$  i
```

```
    zwróć wynik
```

Obie funkcje mają identyczną złożoność czasową, ale różnią się zachowaniem względem pamięci. Wersja iteracyjna korzysta z pamięci stałej, natomiast rekurencyjna wymaga stosu o głębokości n .

Usuwanie rekurencji w strukturach drzewiastych

Rekurencja drzewiasta, taka jak ta używana przy obliczaniu Fibonacciego, stanowi większe wyzwanie. Eliminacja rekurencji polega na zapisaniu procesu w sposób iteracyjny lub dynamiczny, dzięki czemu drzewo obliczeń przestaje być generowane. Fibonacciego można obliczać przy użyciu dwóch zmiennych przechowujących wartości $F(n - 1)$ i $F(n - 2)$. Zamiast obliczać te wartości powtórnie, pętla przechodzi kolejne n , a każdy wynik jest wyliczany na podstawie poprzednich. Dzięki temu usuwamy nie tylko rekurencję, ale także wielokrotność obliczeń.

215

Eliminacja rekurencji poprzez własny stos danych

Specjalną techniką usuwania rekurencji jest wykorzystanie jawnego stosu zamiast stosu systemowego. Idea polega na tym, że za każdym razem, gdy rekurencja wywoływałaby funkcję na mniejszym podproblemie, program iteracyjny odkłada te podproblemy na własną strukturę danych. Po dojściu do najprostszych elementów algorytm zaczyna zdejmować zadania ze stosu i wykonywać operacje wymagane do odbudowy wyniku. Taka technika jest szczególnie użyteczna w algorytmach przetwarzających drzewa lub grafy i pozwala precyzyjnie kontrolować zużycie pamięci.

216

Przykład: iteracyjny wariant DFS

Typowy DFS zapisany rekurencyjnie wykorzystuje stos wywołań funkcji. Jednak w systemach, w których głębokość drzewa może być duża, stos rekurencyjny staje się niebezpieczny. Iteracyjny DFS zapisuje się wprowadzając jawny stos:

(następny slajd)

217

Przykład: iteracyjny wariant DFS

```
DFS_iter(root):  
    stos.push(root)  
    dopóki stos nie jest pusty  
        v = stos.pop()  
        przetwórz v  
        dla każdego sąsiada v  
            stos.push(sąsiad)
```

Ta wersja eliminuje rekurencję całkowicie, przejmując pełną kontrolę nad porządkiem odwiedzania elementów.

218

Usuwanie rekurencji poprzez akumulator

Często rekurencję można zastąpić wersją iteracyjną wykorzystującą akumulator, czyli zmienną przechowującą część wyniku, która rośnie wraz z kolejnymi krokami. Wersja z akumulatorem przypomina rekurencję ogonową, lecz zapisana zostaje jako zwykła pętla. Przykładowo funkcja sumująca liczby od 1 do n może być zapisana rekurencyjnie lub jako pętla wykorzystująca aktualną sumę. Taki sposób jest szczególnie przydatny w obliczeniach liniowych.

219

Usuwanie rekurencji w algorytmach divide and conquer

Usunięcie rekurencji z algorytmów dziel i zwyciężaj jest bardziej skomplikowane, ponieważ wymaga ręcznego sterowania kolejnością częściowych podproblemów. Na przykład w sortowaniu przez scalanie iteracyjny wariant rozpoczyna nie od dzielenia tablicy, lecz od scalania bloków długości jeden, następnie dwóch, później czterech i tak dalej. W efekcie drzewo rekurencji zostaje przekształcone w proces iteracyjny przechodzący kolejne poziomy podziału od dołu do góry.

220

Przykład: iteracyjny mergesort

W wersji iteracyjnej mergesortu blok o długości jednego elementu uważa się za już posortowany. Następnie program łączy sąsiednie bloki o długości dwa, później o długości cztery, aż do osiągnięcia pełnej długości tablicy. W przeciwieństwie do klasycznej rekurencji, która dzieli tablicę od góry do dołu, iteracyjny mergesort odwraca kolejność procesów, eliminując wszystkie wywołania rekurencyjne. Jego złożoność pozostaje identyczna, lecz stos nie jest wykorzystywany.

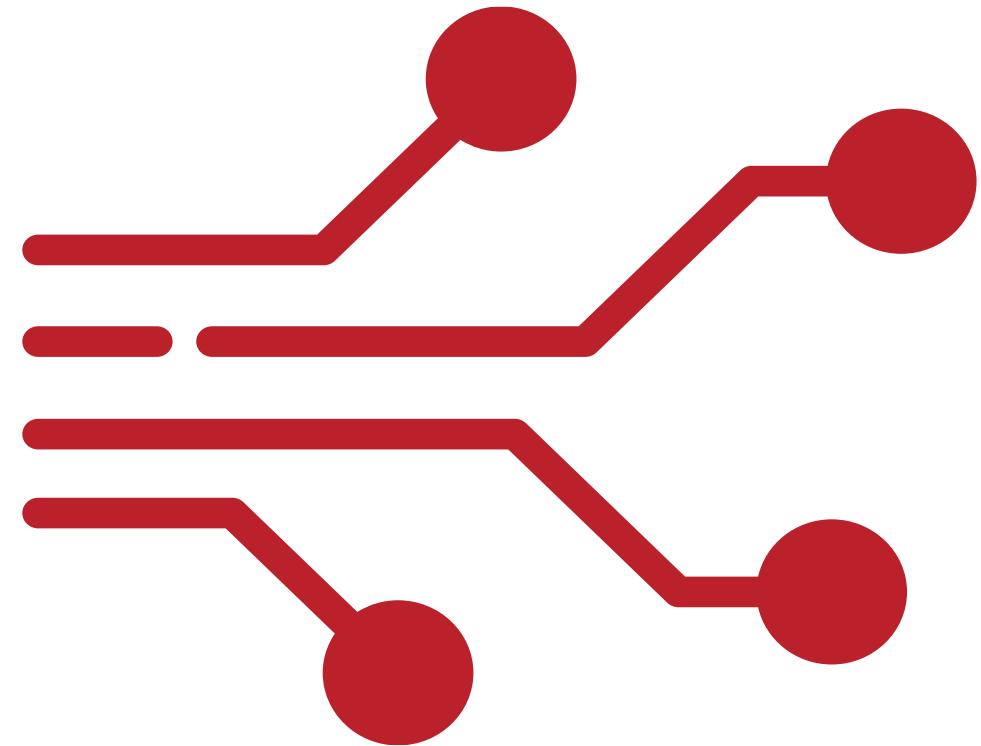
221

Podsumowanie technik eliminacji rekurencji

Eliminacja rekurencji jest działaniem mającym na celu poprawę efektywności pamięciowej i zwiększenie stabilności obliczeń. Można tego dokonać poprzez zamianę rekurencji liniowej na pętle, zastąpienie rekurencji drzewiastej programowaniem dynamicznym, wykorzystanie jawnego stosu oraz sterowanie porządkiem scalania w algorytmach typu divide and conquer. Usuwanie rekurencji nie zmienia samej logiki algorytmu, lecz przekształca jego reprezentację w bardziej bezpieczną i kontrolowaną formę.

222

Podstawowe struktury danych i abstrakcyjne struktury danych



223

Wprowadzenie do struktur danych

Struktury danych stanowią podstawę projektowania każdego algorytmu. Organizują informacje w taki sposób, aby możliwe było ich efektywne przetwarzanie, modyfikowanie i wyszukiwanie. W praktyce programistycznej wybór odpowiedniej struktury danych decyduje o ostatecznej złożoności czasowej programu. W tym wykładzie przejdziemy przez najważniejsze klasy struktur danych, ich właściwości, ograniczenia oraz scenariusze użycia. Towarzyszą temu przykłady, które pokazują wpływ struktury danych na efektywność algorytmów.

224

Struktura danych jako fundament algorytmu

Każdy algorytm operuje na danych, a to jak dane są przechowywane określa możliwości algorytmu. Struktura danych jest narzędziem, które determinuje czas dostępu, liczbę operacji dodatkowych i wygodę implementacji. Tablica umożliwia szybkie adresowanie, lecz jest sztywna pamięciowo. Lista daje elastyczność, ale ogranicza dostęp losowy. Zrozumienie tych kompromisów jest jednym z głównych celów tego wykładu, ponieważ to od wyboru struktury zależy zarówno praktyczna efektywność, jak i teoretyczny koszt algorytmu.

225

Abstrakcyjny typ danych

Abstrakcyjny typ danych (ADT) jest opisem zachowania struktury danych bez wskazania sposobu jej implementacji. ADT definiuje operacje oraz gwarantowane własności, pozostawiając wybór sposobu realizacji projektantowi algorytmu. Przykładowo ADT „stos” określa, że elementy są zdejmowane w odwrotnej kolejności niż zostały dodane. Nie mówi jednak czy stos ma być realizowany tablicą, listą czy bardziej złożonym mechanizmem. To oddzielenie logiki od implementacji pozwala konstruować algorytmy niezależne od niskopoziomowych detali.

226

Tablica: podstawowe własności

Tablica przechowuje elementy w sposób ciągły w pamięci, zapewniając stały czas dostępu do dowolnego indeksu. Jest to ogromna przewaga w algorytmach sortowania, przeszukiwania i generowania wartości. Wadą tablicy jest niemożność dynamicznego zwiększania rozmiaru bez kosztownej operacji kopiowania. Tablice świetnie sprawdzają się w sytuacjach, w których z góry znamy wielkość danych oraz gdy wymagany jest szybki dostęp losowy.

227

Operacje na tablicy i ich koszt

Dostęp do elementu jest stały, ponieważ indeks jest natychmiast przeliczany na adres pamięci. Wstawienie elementu na środek tablicy wymaga przesunięcia wszystkich elementów po prawej stronie, co prowadzi do złożoności liniowej. Usunięcie elementu z początku lub środka ma analogiczny koszt. Te ograniczenia sprawiają, że tablice są doskonałe do przechowywania danych statycznych, ale mniej przydatne przy intensywnych modyfikacjach struktury.

228

Przykład użycia tablicy w pseudokodzie

Poniższy pseudokod przedstawia prosty algorytm znajdowania średniej wartości elementów tablicy:

Average(A, n):

suma $\leftarrow$ 0

for i $\leftarrow$ 1 to n:

 suma $\leftarrow$ suma + A[i]

return suma / n

Operacja ta wykorzystuje fakt, że dostęp do elementu tablicy jest stały i nie wymaga przechodzenia po wskaźnikach.

229

Tablica dynamiczna

Tablica dynamiczna rozszerza tablicę tradycyjną o możliwość automatycznego zwiększania rozmiaru. Gdy tablica jest pełna, jej rozmiar zostaje podwojony, a elementy kopiowane są do nowej pamięci.

Dzięki temu tablice dynamiczne są powszechnie stosowane w językach programowania jako wewnętrzne struktury list.

230

Lista jednokierunkowa

Lista jednokierunkowa składa się z węzłów, z których każdy wskazuje na kolejny element. Struktura ta nie wymaga przechowywania elementów w jednym bloku pamięci. Możliwość wstawiania i usuwania elementów w czasie stałym jest ogromną zaletą list. Wadą natomiast jest brak bezpośredniego dostępu do elementów po indeksie, co wymaga sekwencyjnego przejścia od początku listy.

231

Pseudokod dodawania elementu do listy

InsertFront(L, x):

nowy $\leftarrow$ Node(x)

nowy.next $\leftarrow$ L.head

L.head $\leftarrow$ nowy

Ta operacja działa w czasie stałym, niezależnie od liczby elementów. Jest przykładem efektywnego wstawiania, niemożliwego do osiągnięcia w tablicy bez przesuwania danych.

232

Lista dwukierunkowa

Lista dwukierunkowa dodaje możliwość przemieszczania się także w przeciwnym kierunku, dzięki czemu operacje usuwania i przemieszczania się po liście mogą być znacznie efektywniejsze. Każdy element posiada dwa wskaźniki: na następny i poprzedni element. Jest to struktura często wykorzystywana w implementacjach deque oraz buforów edycyjnych, gdzie kluczowa jest swoboda poruszania się po strukturze.

233

Stos: zasada LIFO

Stos jest strukturą w której dane przetwarzane są zgodnie z zasadą ostatni wszedł, pierwszy wyszedł. Ta kolejność operacji jest naturalna dla algorytmów rekurencyjnych, obsługi wyrażeń matematycznych oraz mechanizmów cofania działań. Stos przechowuje jedynie elementy, do których dostęp jest możliwy z jednej strony — od jego szczytu.

234

Pseudokod użycia stosu w analizie nawiasów

CheckBrackets(S):

```
stos ← empty
for każdy znak c w S:
    if c jest '(':
        push(stos, c)
    else if c jest ')':
        if stos jest pusty:
            return false
        pop(stos)
return stos jest pusty
```

Algorytm ten pokazuje
naturalne zastosowanie stosu
w analizie poprawności
nawiasowania.

235

Kolejka: zasada FIFO

Kolejka przetwarza elementy w kolejności ich dołączenia. Jest to struktura niezwykle istotna dla algorytmów przeszukiwania grafów, systemów kolejkowych i symulacji. Wiele systemów operacyjnych wykorzystuje kolejkę do harmonogramowania zadań, ponieważ gwarantuje ona sprawiedliwe traktowanie przychodzących procesów.

236

Pseudokod BFS wykorzystujący kolejkę

BFS(G , start):

```
odwiedzone ← pusty zbiór  
Q ← pusta kolejka  
enqueue(Q, start)  
while Q nie jest pusta:  
    v ← dequeue(Q)  
    jeśli v nie był odwiedzony:  
        odwiedzone ← odwiedzone  $\cup$  {v}  
        dla każdego sąsiada u w  $G[v]$ :  
            enqueue(Q, u)
```

return odwiedzone

Kolejka umożliwia przetwarzanie węzłów w kolejności rosnącej odległości od punktu startowego.

Kolejka dwustronna

Deque pozwala na dodawanie i usuwanie elementów z obu końców struktury. Jest niezwykle użyteczna w algorytmach optymalizacyjnych takich jak obliczanie maksimum z przesuwającego się okna danych.

Struktura ta jest także podstawą wielu implementacji BFS z wagami zero.

238

Tablice mieszające i funkcje hashujące

Tablica mieszająca przekształca klucz za pomocą funkcji mieszającej na indeks w tablicy. Jej główną zaletą jest możliwość wykonywania operacji w czasie średnio stałym. Jako że zderzenia są nieuniknione, konieczne są mechanizmy ich rozwiązywania, takie jak łańcuchowanie lub adresowanie otwarte. Hash table jest jedną z najczęściej używanych struktur w praktyce programistycznej.

239

Pseudokod wyszukiwania w tablicy mieszajcej

Find(T, key):

$i \leftarrow \text{hash}(\text{key})$

dla każdego elementu e na liście $T[i]$:

jeśli $e.\text{key} = \text{key}$:

return $e.\text{value}$

return nie znaleziono

Wyszukiwanie wymaga przeglądnięcia jedynie listy przypisanej do danego indeksy.

240

Drzewa jako struktury hierarchiczne

Drzewo jest strukturą o węzłach połączonych relacjami rodzic–dziecko. Drzewa pozwalają reprezentować dane hierarchicznie, co umożliwia szybkie wyszukiwanie, klasyfikację i sortowanie. Struktury takie jak BST, AVL czy drzewa czerwono-czarne są kluczowe dla implementacji dynamicznych słowników i zestawów.

241

Drzewo BST i własności porządku

Drzewo BST utrzymuje zależność, że elementy mniejsze znajdują się w lewym poddrzewie, a większe w prawym. Pozwala to na wyszukiwanie w czasie proporcjonalnym do wysokości drzewa. Problem pojawia się, gdy drzewo staje się niezrównoważone, co zmniejsza jego efektywność.

242

Pseudokod wyszukiwania w BST

SearchBST(T, x):

jeśli T jest pusty:

return nie znaleziono

jeśli $x = T.key$:

return T

jeśli $x < T.key$:

return SearchBST(T.left, x)

return SearchBST(T.right, x)

Rekurencyjne przeszukiwanie drzewa odbywa się zgodnie z jego naturalną strukturą.

Drzewa samobalansujące

Aby utrzymać efektywność BST, stosuje się drzewa samobalansujące, takie jak AVL lub czerwono-czarne. Automatycznie pilnują one, aby wysokość drzewa była logarytmiczna. Dzięki temu operacje wyszukiwania, dodawania i usuwania mają gwarantowaną złożoność $O(\log n)$.

244

Graf jako uogólnienie drzewa

Graf jest strukturą przechowującą węzły oraz dowolne relacje między nimi. Grafy stanowią podstawę algorytmów sieciowych, nawigacyjnych, analizy połączeń w sieciach społecznościowych oraz systemów rekomendacji. W grafie nie istnieje konieczność istnienia struktury hierarchicznej, co czyni go znacznie bardziej elastycznym niż drzewo.

245

Reprezentacja grafu: lista sąsiedztwa

Lista sąsiedztwa jest najbardziej efektywną pamięciowo reprezentacją grafu rzadkiego. Dla każdego wężła przechowywana jest lista jego sąsiadów. Pozwala to na szybkie przechodzenie po połączeniach, co jest kluczowe dla algorytmów takich jak BFS czy DFS.

246

Reprezentacja grafu: macierz sąsiedztwa

Macierz sąsiedztwa jest tablicą, która przechowuje informację o połączeniach między każdą parą węzłów. Struktura ta pozwala na bardzo szybkie sprawdzenie, czy dwa węzły są połączone, lecz wymaga dużo pamięci dla grafów rzadkich. Jest często używana w grafach gęstych, gdzie liczba połączeń jest bliska liczbie możliwych par.

247

DFS jako przykład głębokiej eksploracji

DFS przegląda graf w głąb, wykorzystując stos do przechowywania ścieżki eksploracji. Algorytm jest naturalnie definiowany rekurencyjnie, choć możliwe jest także jego zapisanie w formie iteracyjnej poprzez jawne użycie stosu. DFS jest wykorzystywany przy wykrywaniu cykli, analizie spójności i porządkowaniu topologicznym.

248

Pseudokod DFS

DFS(v):

 oznacz v jako odwiedzony

 dla każdego sąsiada u w $G[v]$:

 jeśli u nieodwiedzony:

 DFS(u)

Ta forma podkreśla naturalną rekurencyjność przeglądania grafu.

249

Zbiory rozłączne

Struktura zbiorów rozłącznych pozwala określać, które elementy należą do tego samego zbioru, oraz łączyć zbiory w sposób szybki. Wykorzystywana jest w algorytmach grafowych takich jak Kruskal przy budowie minimalnego drzewa rozpinającego. Optymalizacje takie jak spłaszczanie ścieżki i union by rank prowadzą do niemal stałej złożoności operacji.

250

Pseudokod Union–Find

Find(x):

jeśli $\text{parent}[x] \neq x$:

$\text{parent}[x] \leftarrow \text{Find}(\text{parent}[x])$

return $\text{parent}[x]$

Union(x, y):

$rx \leftarrow \text{Find}(x)$

$ry \leftarrow \text{Find}(y)$

jeśli $rx \neq ry$:

$\text{parent}[ry] \leftarrow rx$

Struktura ta jest dowodem na to, że odpowiedni projekt danych potrafi drastycznie poprawić wydajność algorytmu.

251

Kopiec jako struktura priorytetowa

Kopiec jest strukturą danych zapewniającą szybki dostęp do elementu o najwyższym priorytecie. Kopiec binarny jest reprezentowany tablicą i używany jako podstawa kolejki priorytetowej. Jego właściwości pozwalają wykonywać wstawianie i usuwanie w czasie $O(\log n)$, co jest kluczowe dla algorytmu Dijkstry oraz dla sortowania heapsort.

252

Pseudokod operacji insert w kopcu

Insert(H, x):

dodaj x na koniec tablicy H

$i \leftarrow$ indeks x

while $i > 1$ i $H[i] > H[i/2]$:

zamień $H[i]$ z $H[i/2]$

$i \leftarrow i/2$

Operacja ta przywraca własność kopca poprzez przesuwanie elementu w górę.

253

Kolejka priorytetowa i jej zastosowania

Kolejka priorytetowa pozwala na przetwarzanie elementów w kolejności ich znaczenia, a nie momentu pojawienia się. Jest to struktura naturalna dla algorytmów planowania, przydzielania zasobów i obliczania najkrótszych ścieżek. W przeciwieństwie do zwykłej kolejki umożliwia efektywne wybieranie najważniejszego zadania w danym momencie.

254

Rola struktur danych w efektywności algorytmów

Wybór struktury danych rzutuje na wszystkie aspekty algorytmu. Źle dobrana struktura może spowodować wzrost złożoności z liniowej do kwadratowej. Na przykład operacja wyszukiwania wykonywana tysiące razy na liście będzie dużo wolniejsza niż na tablicy haszującej. Projektowanie algorytmów jest więc nierozzerwalnie związane z analizą struktur danych.

255

Struktury danych a złożoność pamięciowa

Oprócz czasu działania analizujemy także ilość pamięci potrzebnej do działania struktury. Tablice wymagają ciągłego kawałka pamięci, co może powodować problemy przy dużych rozmiarach. Listy wykorzystują pamięć bardziej elastycznie, ale płacą koszt narzutu wskaźników. Kopce i grafy wymagają struktury pośredniej, co zwiększa ich złożoność pamięciową.

256

Struktury specjalistyczne

Oprócz podstawowych struktur istnieją bardziej zaawansowane, takie jak drzewa przedziałowe, drzewa Fenwicka czy skiplisty. Są one stosowane w zadaniach wymagających dynamicznego wykonywania zapytań na przedziałach lub szybkiej agregacji danych. Ich projektowanie jest bardziej złożone, ale oferują ogromne przyspieszenie w wyspecjalizowanych zastosowaniach.

257

Skiplisty jako probabilistyczna struktura danych

Skiplista wykorzystuje losowość do konstruowania wielopoziomowej listy pozwalającej na szybkie wyszukiwanie. Mimo że jest strukturą probabilistyczną, w praktyce osiąga wydajność porównywalną z drzewami samobalansującymi. Skiplisty są ciekawym przykładem tego, jak elegancko można zastąpić złożone struktury deterministyczne poprzez losowość.

258

Przykład wyszukiwania w skipliście

Wyszukiwanie przebiega od najwyższego poziomu listy i stopniowo schodzi w dół, aż znajdzie odpowiednią pozycję. Dzięki losowej wysokości elementów średnia złożoność wyszukiwania, dodawania i usuwania wynosi $O(\log n)$, a implementacja jest znacznie prostsza niż w drzewach AVL lub RB.

259

Struktury danych a architektura procesora

W praktyce efektywność struktur danych zależy również od lokalności pamięci i organizacji cache. Tablice są zwykle znacznie szybsze od list, ponieważ ich elementy znajdują się w sąsiadujących komórkach pamięci, co ułatwia ich pobieranie przez procesor. Dla wielu algorytmów różnica ta jest bardziej znacząca niż różnica w złożoności teoretycznej.

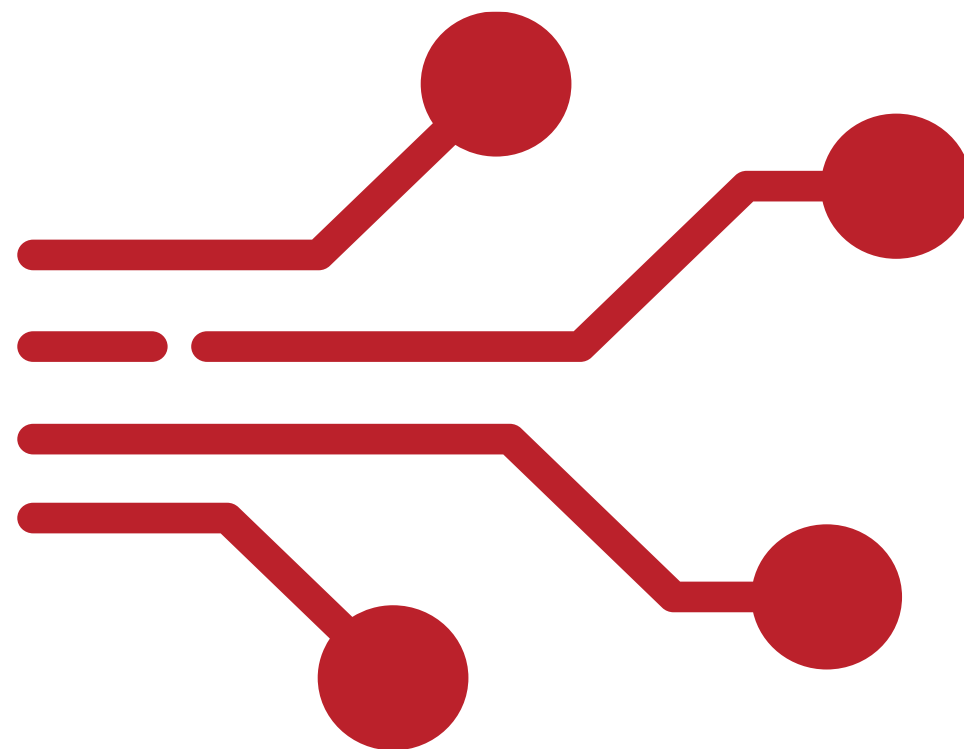
260

Podsumowanie wykładu

Przedstawiliśmy szeroki przegląd struktur danych oraz pokazaliśmy ich zastosowania w algorytmach. Struktury takie jak tablice, listy, stosy, kolejki, kopce, drzewa i grafy stanowią fundament współczesnej informatyki. Wybór odpowiedniej struktury danych jest jednym z kluczowych elementów projektowania algorytmów i decyduje o ich efektywności. Zrozumienie ich właściwości jest niezbędne do dalszej pracy nad bardziej zaawansowanymi technikami algorytmicznymi.

261

Kolejka priorytetowa, słownik



262

Wprowadzenie do struktur abstrakcyjnych

Kolejka priorytetowa oraz słownik należą do fundamentalnych abstrakcyjnych struktur danych, które odgrywają kluczową rolę zarówno w projektowaniu algorytmów, jak i w implementacji systemów informatycznych. Kolejka priorytetowa dostarcza mechanizmu pobierania elementów o najwyższym priorytecie, co okazuje się niezbędne w harmonogramowaniu zadań, algorytmach grafowych czy systemach operacyjnych.

263

Wprowadzenie do struktur abstrakcyjnych

Natomiast słownik, zwany też mapą, dostarcza elastycznego modelu przechowywania par klucz–wartość, który stanowi fundament dla baz danych, kompilatorów, indeksów i wyszukiwarek. W niniejszym wykładzie przedstawiamy oba te abstrakcyjne typy danych, ich modele, operacje, a także implementacje oparte na drzewach i tablicach mieszających.

264

Kolejka priorytetowa: definicja

Kolejka priorytetowa jest strukturą, która przechowuje elementy wraz z ich priorytetami i pozwala użytkownikowi na pobranie elementu, który ma najwyższy priorytet. Można ją traktować jako uogólnienie zwykłej kolejki FIFO, z tą różnicą, że kolejność usuwania elementów nie zależy od czasu ich dodania, lecz od ich znaczenia, które wyrażone jest priorytetem.

Operacje podstawowe obejmują wstawianie nowego elementu z priorytetem oraz usunięcie elementu o najwyższym priorytecie. W zależności od implementacji priorytetem może być liczba, wartość logiczna, a nawet bardziej złożona struktura podlegająca porównaniom.

265

Operacje kolejki priorytetowej

Kolejka priorytetowa definiuje trzy fundamentalne operacje: wstawienie nowego elementu wraz z priorytetem, podejrzenie elementu o najwyższym priorytecie bez usuwania go oraz usunięcie tego elementu. Wszystkie te operacje muszą zachować poprawną strukturę kolejki, tak aby w każdej chwili można było szybko określić, który element ma największy priorytet.

Intuicyjnie można traktować kolejkę priorytetową jako dynamiczny zbiór uporządkowany według priorytetu, który stale podlega zmianom, lecz zawsze pozwala na natychmiastowy dostęp do elementu najlepszej jakości.

Pseudokod prostego modelu

Najprostsza implementacja kolejki priorytetowej może wykorzystywać zwykłą listę. Wówczas wstawienie elementu jest operacją łatwą, natomiast pobranie elementu o najwyższym priorytecie wymaga przeszukania całej listy. Poniżej przedstawiono prosty pseudokod:

Insert(Q , x):
dodaj x na koniec listy Q

267

Pseudokod prostego modelu

```
ExtractMax(Q):  
    najlepszy ← Q[1]  
    indeks ← 1  
    dla i = 2..n:  
        jeśli Q[i].priorytet > najlepszy.priorytet:  
            najlepszy ← Q[i]  
            indeks ← i  
    usuń Q[indeks]  
    zwróć najlepszy
```

268

Wykorzystanie kopca: intuicja

Kopiec binarny jest najczęściej stosowaną strukturą do implementacji kolejki priorytetowej, ponieważ zapewnia logarytmiczny czas działania dla wstawiania elementów oraz usuwania elementu o najwyższym priorytecie.

Kopiec można traktować jako drzewo binarne, które utrzymuje porządek priorytetów oraz jest reprezentowane w tablicy. Każdy węzeł ma priorytet co najmniej tak duży jak jego dzieci, dzięki czemu element o najwyższym priorytecie znajduje się zawsze na szczycie kopca i można go pobrać w czasie stałym. Reszta drzewa jest przechowywana w taki sposób, aby operacje naprawy kopca po wstawieniu lub usunięciu były maksymalnie efektywne.

269

Struktura kopca binarnego

Kopiec binarny jest kompletnym drzewem binarnym, które można przechowywać w jednowymiarowej tablicy. Węzeł o indeksie i ma dzieci o indeksach $2i$ i $2i+1$, a rodzica w indeksie $\lfloor i/2 \rfloor$. Ta reprezentacja zapewnia dużą efektywność pamięciową, ponieważ nie wymaga wskaźników. Aby utrzymać własność kopca, każda operacja zmieniająca strukturę drzewiastą wykonuje tzw. przesiewanie w górę lub w dół. Dzięki temu kopiec zawsze zachowuje strukturę drzewa kompletnego, jednocześnie gwarantując poprawność relacji priorytetów pomiędzy węzłami.

270

Wstawianie do kopca: pseudokod

Wstawianie elementu polega na umieszczeniu go na końcu tablicy reprezentującej kopiec, a następnie podciągnięciu go w górę, aż znajdzie właściwe miejsce zgodnie z porządkiem priorytetów. Mechanizm ten nosi nazwę „heapify-up”:

271

Wstawianie do kopca: pseudokod

Insert(H, x):

$n \leftarrow n + 1$

$H[n] \leftarrow x$

$i \leftarrow n$

dopóki $i > 1$ i $H[i].\text{priorytet} > H[i/2].\text{priorytet}$:

zamień $H[i]$ z $H[i/2]$

$i \leftarrow i/2$

Dzięki temu podciąganiu wystarczy wykonać co najwyżej $\log_2(n)$ zamian, co daje bardzo efektywną operację, szczególnie przy dużych zbiorach danych.

272

Usuwanie maksimum z kopca

Usunięcie elementu o najwyższym priorytecie wymaga zastąpienia korzenia ostatnim elementem z tablicy, zmniejszenia rozmiaru kopca oraz poprawienia jego struktury przez przesiewanie w dół. Proces ten nosi nazwę „heapify-down” i wygląda następująco:

(porównaj następny slajd)

ExtractMax(H):

$\text{max} \leftarrow H[1]$

$H[1] \leftarrow H[n]$

$n \leftarrow n - 1$

$i \leftarrow 1$

273

Usuwanie maksimum z kopca

```
dopóki istnieje dziecko H[i]:  
    wybierz dziecko o większym  
    priorytecie  
    jeśli H[i].priorytet <  
    H[dziecko].priorytet:  
        zamień H[i] z H[dziecko]  
        i ← dziecko  
    inaczej:  
        przerwij  
zwróć max
```

274

Budowa kopca w czasie liniowym

Zbudowanie kopca z nieuporządkowanej tablicy można wykonać w czasie liniowym, stosując procedurę „heapify” od połowy tablicy w dół. Mimo że operacje lokalne mają charakter logarytmiczny, łączna praca na poziomach drzewa daje zaskakująco efektywne $\Theta(n)$. Jest to jeden z najciekawszych wyników z teorii struktur danych, ponieważ pozwala natychmiast przekształcić dowolny zbiór elementów w kolejkę priorytetową o optymalnych właściwościach.

275

Budowa kopca w czasie liniowym

Podczas budowania kopca kluczowe znaczenie ma sposób, w jaki elementy są przesiewane w dół drzewa. Koszt tej operacji nie jest jednakowy dla wszystkich węzłów, lecz zależy od ich położenia w strukturze kopca. Elementy znajdujące się bliżej korzenia wymagają więcej pracy, ponieważ mogą być przesuwane przez większą liczbę poziomów. Z kolei elementy położone bliżej liści przemieszczają się na krótsze odległości, co znacząco zmniejsza koszt ich przetwarzania.

276

Kolejka priorytetowa w algorytmach grafowych

Kolejka priorytetowa stanowi centralny element wielu algorytmów na grafach, takich jak Dijkstra, Prim czy A*. W każdym z tych algorytmów konieczne jest szybkie pobieranie wierzchołka, który w danym momencie ma najlepsze właściwości – czy to najmniejszy dystans, czy najniższy koszt heurystyczny.

Kopiec binarny pozwala wykonać tę operację w czasie logarytmicznym, co jest kluczowe przy wielokrotnym powtarzaniu jej dla setek tysięcy elementów. Dzięki temu struktura ta umożliwia praktyczne wykonywanie obliczeń na ogromnych grafach, które pojawiają się w logistyce, nawigacji i analizie sieci.

277

Kolejka priorytetowa a sortowanie

Sortowanie przez kopcowanie, znane jako Heapsort, jest bezpośrednią konsekwencją właściwości kolejki priorytetowej. Umieszczając elementy w kopcu, a następnie wielokrotnie pobierając maksimum, otrzymujemy tablicę w porządku malejącym. Cała operacja wykorzystuje dokładnie tę samą strukturę danych i jest przykładem praktycznego wykorzystania kopca do celów sortowania. Heapsort działa deterministycznie i zawsze w czasie $O(n \log n)$, co czyni go ważnym algorytmem z punktu widzenia teoretycznej analizy algorytmów. Skuteczność tej metody zależy od stabilności operacji „heapify”, które powtarzają się wielokrotnie podczas opróżniania kopca.

278

Złożoność kolejki priorytetowej

Najważniejsze właściwości złożoności kolejki priorytetowej są następujące: operacja wstawiania ma złożoność logarytmiczną, tak samo jak operacja usuwania maksimum. Z kolei podejrzenie maksimum wymaga jedynie stałego czasu, ponieważ element o największym priorytecie znajduje się zawsze na szczycie kopca. W praktyce oznacza to, że kolejka priorytetowa jest jedną z najbardziej użytecznych struktur danych w sytuacjach, w których dostęp do „najlepszego” elementu jest kluczowy. Z tego powodu stanowi fundament nie tylko algorytmów grafowych, ale także systemów operacyjnych zarządzających procesami.

279

Wprowadzenie do słownika

Słownik, zwany również mapą lub tablicą asocjacyjną, jest strukturą przechowującą pary klucz–wartość, co pozwala na szybkie przechowywanie i odnajdywanie informacji na podstawie klucza. W przeciwieństwie do tablic indeksowanych liczbowo, kluczami mogą być niemal dowolne obiekty, takie jak napisy, liczby, symbole lub bardziej skomplikowane struktury. Słownik jest fundamentem współczesnych języków programowania, w których stanowi podstawę implementacji rekordów, obiektów, tablic mieszających, baz danych w pamięci oraz wielu innych struktur.

280

Abstrakcyjne operacje na słowniku

Słownik udostępnia trzy podstawowe operacje: wstawienie nowej pary klucz–wartość, pobranie wartości na podstawie klucza oraz usunięcie istniejącego wpisu. W trakcie działania słownika konieczne jest utrzymanie spójności między kluczami a wartościami, a także efektywne zarządzanie pamięcią. Ważne jest, aby operacje słownika działały szybko, nawet jeśli kluczy jest bardzo dużo.

W praktyce słowniki są używane w kompilatorach, analizie semantycznej, zarządzaniu symbolami, a nawet w implementacjach skomplikowanych indeksów tekstowych.

281

Model matematyczny słownika

Można formalnie zapisać słownik jako zbiór par (k, v) , w którym każdy klucz k jest unikalny. Operacja wyszukiwania zwraca wartość v , jeśli klucz istnieje w strukturze, albo sygnalizuje brak wpisu. Z punktu widzenia abstrakcyjnej struktury danych nie definiujemy sposobu przechowywania kluczy ani ich rozmieszczenia w pamięci. Liczy się jedynie gwarancja poprawności oraz odpowiednia złożoność. To właśnie implementacja decyduje o tym, czy wyszukiwanie zajmuje $\Theta(\log n)$, czy może $\Theta(1)$ w przypadku tablic mieszających.

282

Implementacja słownika za pomocą drzewa BST

Najprostszą implementacją słownika jest zastosowanie binarnego drzewa poszukiwań, w którym klucze są uporządkowane według relacji porównywania. Wyszukiwanie odbywa się poprzez przechodzenie w dół drzewa, wybierając lewe lub prawe poddrzewo w zależności od tego, czy klucz jest mniejszy, czy większy od klucza w aktualnym węźle. W idealnych warunkach wysokość drzewa wynosi $\log_2(n)$, co pozwala osiągnąć złożoność wyszukiwania i wstawiania na poziomie logarytmicznym. Niestety, w niekorzystnych przypadkach drzewo może stać się niezbalansowane, co wydłuża operacje do czasu liniowego.

283

Wstawianie w drzewie BST: pseudokod

Wstawianie nowej pary (k, v) polega na znalezieniu odpowiedniej pozycji w drzewie i umieszczeniu tam nowego węzła:

Ta definicja zapewnia unikalność kluczy i zachowuje uporządkowanie drzewa, ale nie gwarantuje optymalnej wysokości, co prowadzi do konieczności stosowania struktur samobalansujących.

$\text{Insert}(T, k, v)$:

jeśli T jest puste:

$T \leftarrow \text{nowy\_węzeł}(k, v)$

zwróć

jeśli $k < T.\text{klucz}$:

$\text{Insert}(T.\text{lewe}, k, v)$

inaczej jeśli $k > T.\text{klucz}$:

$\text{Insert}(T.\text{prawe}, k, v)$

inaczej:

$T.\text{wartość} \leftarrow v$

284

Drzewa samobalansujące

Aby zapobiec degeneracji drzewa BST, w praktyce stosuje się struktury samobalansujące, takie jak drzewa AVL, czerwono-czarne, splay czy B-drzewa. Każde z nich posiada mechanizmy rotacji i korekcji drzewa, które gwarantują, że jego wysokość pozostaje logarytmiczna, nawet przy niekorzystnej kolejności wstawiania.

Dzięki temu operacje słownika można realizować konsekwentnie w czasie $O(\log n)$, co jest niezbędne w systemach bazodanowych i indeksach plikowych. Samobalansowanie nie zmienia abstrakcyjnego modelu słownika, lecz wpływa na jego efektywność.

285

Tablice mieszające: idea

Najbardziej efektywną i najczęściej stosowaną implementacją słownika są tablice mieszające, które wykorzystują funkcje hashujące do przekształcania kluczy na indeksy w tablicy. Dzięki temu wyszukiwanie może odbywać się w czasie średnim $O(1)$, co jest nieporównywalnie szybsze niż struktury drzewiaste. Funkcja mieszająca powinna równomiernie rozkładać klucze, aby ograniczyć liczbę kolizji. W praktyce tablice mieszające są fundamentem słowników w językach Python, Java, C++ oraz wielu innych środowiskach programistycznych.

286

Funkcja mieszająca

Funkcja mieszająca powinna przekształcać klucz w liczbę całkowitą, tak aby niewielkie zmiany w kluczu powodowały duże zmiany w wyniku funkcji. Dzięki temu klucze są rozpraszane równomiernie w tablicy. Idealna funkcja hashująca przypomina funkcję losową, choć musi działać deterministycznie i efektywnie. Klucze tekstowe zwykle przekształca się, iterując po znakach i akumulując wartość według pewnego równania, jak np. mieszanie wielomianowe. Niewłaściwy dobór funkcji mieszającej prowadzi do wielu kolizji, co znacząco obniża wydajność słownika.

287

Kolizje w tablicy mieszającej

Kolizja powstaje wtedy, gdy dwa różne klucze dają ten sam wynik funkcji hashującej. Ponieważ tablica mieszająca musi obsługiwać takie przypadki, stosuje się techniki rozwiązywania kolizji. Najpopularniejsze metody to łańcuchowanie oraz adresowanie otwarte. W łańcuchowaniu każdy element tablicy przechowuje listę kluczy, które przyporządkowano temu samemu indeksowi.

288

Kolizje w tablicy mieszającej

W adresowaniu otwartym elementy są umieszczane w alternatywnych miejscach tablicy według z góry określonego schematu.

Oba podejścia mają swoje zalety i ograniczenia, ale umożliwiają utrzymanie średniej złożoności wyszukiwania w czasie stałym.

289

Pseudokod słownika z łańcuchowaniem

Insert(T , k , v):

$i \leftarrow \text{hash}(k)$

jeśli istnieje para(k , v') w
 $T[i]$:

zaktualizuj v'

inaczej:

dodaj (k , v) do listy $T[i]$

Find(T , k):

$i \leftarrow \text{hash}(k)$

dla każdej pary (k' , v) w
 $T[i]$:

jeśli $k' = k$:

zwróć v

zwróć "brak"

290

Adresowanie otwarte: idea

Adresowanie otwarte polega na tym, że wszystkie elementy znajdują się bezpośrednio w tablicy, a w przypadku kolizji stosuje się pewien schemat sondowania, aby znaleźć następny wolny indeks. Najprostszy wariant to sondowanie liniowe, w którym elementy przesuwają się o jeden indeks naprzód, aż znajdą wolne miejsce.

W innych wersjach stosuje się sondowanie kwadratowe lub podwójne mieszanie. Podejście to eliminuje potrzebę stosowania list, lecz wymaga bardziej skomplikowanej analizy i może prowadzić do zjawiska klastrów.

291

Rehashing i współczynnik zapełnienia

Tablice mieszające muszą być dynamiczne, aby zachować wysoką efektywność. Kiedy liczba elementów przekracza pewien próg, zwykle równy 70–80% pojemności tablicy, konieczna jest jej przebudowa. Polega ona na utworzeniu większej tablicy oraz ponownym mieszaniu wszystkich kluczy według funkcji haszującej.

Proces ten nosi nazwę rehashingu i ma na celu utrzymanie niskiego współczynnika zapełnienia. Choć rehashing jest kosztowny, jego koszt amortyzowany na pojedynczą operację pozostaje stały, co zapewnia wysoką efektywność słownika w długim okresie.

292

Złożoność operacji w słowniku

Główną zaletą tablic mieszających jest to, że wyszukiwanie, wstawianie i usuwanie działają w czasie średnim $O(1)$, co czyni słownik jedną z najskuteczniejszych struktur danych. W najgorszym przypadku czas działania może spaść do $O(n)$, lecz przy dobrej funkcji mieszającej i właściwym współczynniku zapełnienia przypadek ten występuje niezwykle rzadko.

293

Złożoność operacji w słowniku

Drzewa BST (Binary Search Trees) zapewniają w typowym przypadku złożoność operacji takich jak wyszukiwanie, wstawianie czy usuwanie na poziomie $O(\log n)$, pod warunkiem że drzewo pozostaje zrównoważone. W odróżnieniu od tablic mieszających (hash tables), których wydajność silnie zależy od jakości funkcji haszującej oraz rozkładu kluczy, drzewa BST oferują bardziej przewidywalne działanie. Nawet jeśli dane wejściowe mają niekorzystny rozkład, odpowiednio zbalansowane struktury (np. drzewa AVL czy czerwono-czarne) utrzymują gwarancję logarytmicznej złożoności.

294

Słownik w praktyce programistycznej

Współczesne języki programowania wyposażają słownik w dodatkowe funkcje, takie jak iteracja po elementach, sprawdzanie istnienia kluczy, usuwanie wielu elementów naraz czy dynamiczne powiększanie struktury. Python stosuje zaawansowaną tablicę mieszającą z adresowaniem otwartym, Java wykorzystuje HashMap oraz TreeMap, natomiast C++ udostępnia unordered_map oraz map.

295

Słownik w praktyce programistycznej

Każda z tych struktur korzysta z opisanych wcześniej mechanizmów, dopasowując ich realizację do wymagań platformy. Słowniki stanowią niezastąpione narzędzie w większości aplikacji, począwszy od systemów operacyjnych aż po narzędzia analityczne.

296

Wybór implementacji słownika

Decyzja o wyborze konkretnej implementacji słownika powinna zależeć od sposobu wykorzystania i oczekiwanych operacji. Jeśli klucze mają stabilny rozkład, tablica mieszająca będzie najbardziej efektywna. Jeżeli natomiast konieczne jest przechowywanie kluczy w porządku uporządkowanym lub wykonywanie operacji zakresowych, lepiej wybrać drzewo samobalansujące.

297

Wybór implementacji słownika

W przypadku systemów o dużej liczbie kolizji lub niejednorodnych kluczach funkcja mieszająca musi zostać starannie zaprojektowana. Kluczowy jest również koszt pamięci, ponieważ tablice mieszające zwykle wymagają dodatkowej przestrzeni w celu utrzymania odpowiedniego zapełnienia.

298

Kolejka priorytetowa vs słownik

Kolejka priorytetowa i słownik różnią się zasadniczo w sposobie zarządzania danymi. Kolejka priorytetowa pozwala na szybkie pobieranie elementu o najwyższym priorytecie, lecz nie umożliwia efektywnego wyszukiwania po kluczu. Słownik natomiast pozwala na wyszukiwanie na podstawie klucza, lecz nie dostarcza możliwości pobierania elementu o najwyższym priorytecie w czasie stałym lub logarytmicznym.

W praktycznych systemach często stosuje się obie struktury równocześnie, aby uzyskać pełnię funkcjonalności, na przykład w algorytmach grafowych czy systemach bazodanowych.

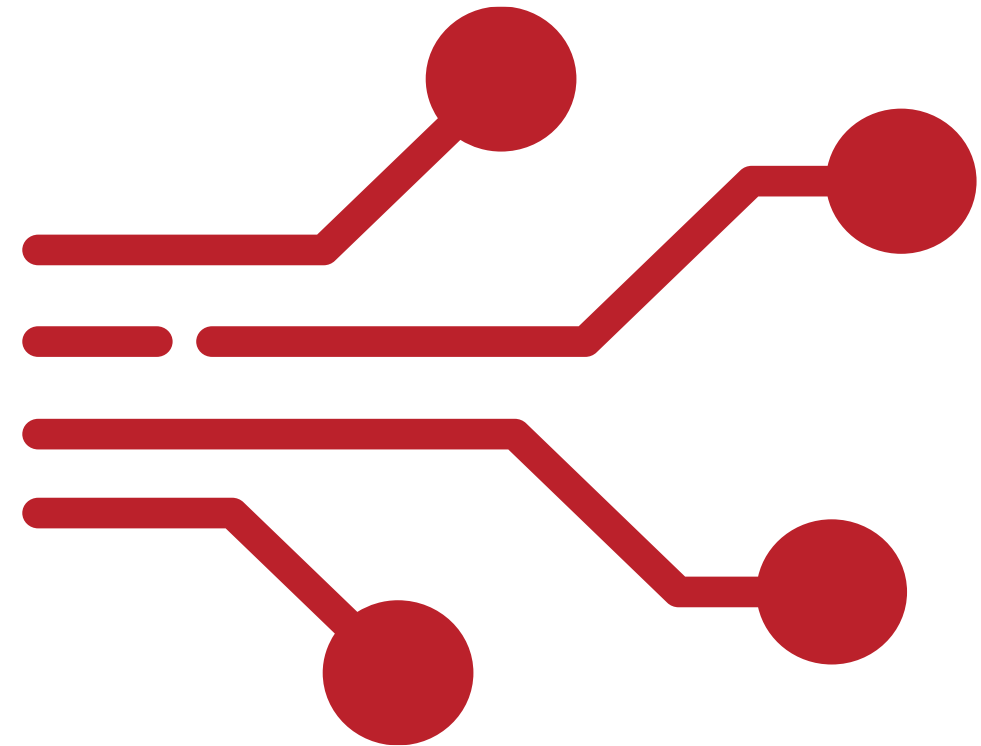
299

Kolejka priorytetowa oraz słownik są kluczowymi abstrakcyjnymi strukturami danych, które znajdują zastosowanie w niemal wszystkich obszarach informatyki. Kolejka priorytetowa wykorzystująca kopiec binarny zapewnia szybkie operacje pobierania elementu najlepszej jakości, co jest fundamentem wielu algorytmów i systemów.

Słownik natomiast umożliwia szybkie wyszukiwanie po kluczu dzięki tablicom mieszającym lub drzewom samobalansującym. Zrozumienie budowy, działania i złożoności tych struktur jest niezbędne do projektowania efektywnych rozwiązań programistycznych.

300

Podstawowe algorytmy grafowe



301

Plan wykładu

W ramach tego wykładu przedstawimy kluczowe pojęcia związane z grafami, ich formalną definicję oraz różne sposoby reprezentacji. Następnie omówimy fundamentalne algorytmy przeszukiwania grafów, takie jak Breadth-First Search (BFS, przeszukiwanie wszerz) oraz Depth-First Search (DFS, przeszukiwanie w głąb), ich własności, przykłady działania oraz pseudokod

302

Plan wykładu

W dalszej części przejdziemy do algorytmów znajdowania najkrótszych ścieżek: algorytmu Dijkstry, algorytmu Bellmana-Forda oraz algorytmu Floyd-Warshalla dla wszystkich par wierzchołków. Na koniec omówione zostaną algorytmy budowania Minimalnego Drzewa Rozpinającego (Minimum Spanning Tree, MST), czyli algorytm Prima oraz algorytm Kruskala. Każda sekcja zawiera analizę złożoności oraz interpretację wyników.

303

Wprowadzenie do grafów

Graf jest modelem relacji między obiektami, w którym obiekty przedstawione są jako wierzchołki, a połączenia między nimi jako krawędzie. Grafy stanowią podstawę modelowania sieci społecznych, struktur komunikacyjnych, połączeń drogowych, zależności między zadaniami, a także wielu modeli sztucznej inteligencji. W informatyce grafy są jedną z najczęściej używanych struktur danych, a ich analiza wymaga zrozumienia zarówno formalnych definicji, jak i praktycznych algorytmów przetwarzania.

304

Definicja formalna grafu

Graf definiujemy jako $G = (V, E)$, gdzie V to zbiór wierzchołków, natomiast E to zbiór krawędzi. Krawędź jest parą wierzchołków (u, v) . Jeżeli krawędzie nie mają kierunku, mówimy o grafie nieskierowanym. Jeżeli kierunek ma znaczenie — o grafie skierowanym (digrafie). Możemy wyróżnić także grafy ważone, w których każda krawędź ma wagę reprezentującą koszt lub odległość. Takie grafy stanowią podstawę dla algorytmów najkrótszych ścieżek.

305

Podstawowe pojęcia grafowe

Ścieżka w grafie to ciąg wierzchołków połączonych kolejnymi krawędziami. Długość ścieżki to liczba lub suma wag krawędzi.

Cykl to ścieżka rozpoczynająca się i kończąca w tym samym wierzchołku. Graf nazywamy spójnym, jeżeli z każdego wierzchołka możemy dotrzeć do każdego innego.

W grafach skierowanych stosujemy pojęcia osiągalności oraz silnej spójności. Węzły mogą mieć różny stopień, czyli liczbę krawędzi incydentnych — w grafach skierowanych rozróżniamy stopień wejściowy i wyjściowy.

306

Reprezentacje grafów — wprowadzenie

Algorytmy grafowe operują na wewnętrznej strukturze przechowującej informacje o sąsiadach. W praktyce stosuje się trzy podstawowe formy: macierz sąsiedztwa, listę sąsiedztwa oraz listę krawędzi. Każda z nich ma inne zastosowania i wpływa na złożoność czasu oraz pamięci. Wybór reprezentacji zależy od liczby krawędzi w stosunku do liczby wierzchołków oraz od charakteru zastosowanego algorytmu.

307

Macierz sąsiedztwa

Macierz sąsiedztwa to kwadratowa tablica o wymiarach $|V| \times |V|$,

w której wpis 1 lub waga oznacza istnienie krawędzi między wierzchołkami. Reprezentacja ta jest szczególnie użyteczna w grafach gęstych, gdzie liczba krawędzi jest bliska maksymalnej. Zaletą macierzy są operacje sprawdzania połączeń w czasie stałym $O(1)$. Wadą jest koszt pamięci $O(|V|^2)$, który czyni ją nieoptymalną dla grafów rzadkich.

308

Lista sąsiedztwa

Lista sąsiedztwa jest strukturą, w której dla każdego wierzchołka przechowywana jest lista jego sąsiadów. Taka reprezentacja jest bardzo pamięciooszczędna i pozwala na szybkie przeglądanie wszystkich krawędzi wychodzących z wierzchołka. Jej złożoność pamięciowa wynosi $O(|V| + |E|)$. To właśnie ta forma reprezentacji wykorzystywana jest w większości algorytmów grafowych z uwagi na jej praktyczność i wystarczającą wydajność.

309

Lista krawędzi

Lista krawędzi przechowuje jedynie uporządkowaną listę wszystkich krawędzi grafu. Każda krawędź zapisana jest jako para wierzchołków oraz opcjonalnie waga. Taka forma jest idealna dla algorytmów operujących bezpośrednio na krawędziach, takich jak algorytm Kruskala wyznaczający Minimalne Drzewo Rozpinające (Minimum Spanning Tree, MST). Lista krawędzi jest wyjątkowo kompaktowa i prosta w obsłudze.

310

Przeszukiwanie grafu — idea ogólna

Aby analizować strukturę grafu, musimy umieć go przeszukiwać. Przeszukiwanie pozwala określić, które wierzchołki są osiągalne, w jakiej kolejności odwiedzamy elementy oraz jak odtworzyć relacje między nimi.

Najpopularniejsze strategie to Breadth-First Search (BFS) — przeszukiwanie wszerz oraz Depth-First Search (DFS) — przeszukiwanie w głąb. Obie techniki stanowią fundament niemal wszystkich bardziej zaawansowanych algorytmów grafowych.

311

Breadth-First Search (BFS) — opis działania

Breadth-First Search (BFS) to metoda przeszukiwania grafu warstwami. Rozpoczynamy od wierzchołka startowego i odwiedzamy wszystkich jego bezpośrednich sąsiadów, następnie sąsiadów tych sąsiadów i tak dalej. BFS gwarantuje znalezienie najkrótszej ścieżki w grafach nieważonych. Wykorzystuje kolejkę (Queue) jako strukturę pomocniczą, ponieważ odwiedza wierzchołki dokładnie w takiej kolejności, w jakiej zostały odkryte.

312

BFS — pseudokod

BFS(G, s):

dla każdego wierzchołka v w G :

$visited[v] \leftarrow false$

Queue Q

$visited[s] \leftarrow true$

$Q.enqueue(s)$

dopóki Q nie jest pusta:

$u \leftarrow Q.dequeue()$

przetwórz(u)

dla każdego sąsiada

wierzchołka u :

jeśli $visited[w] =$

false:

$visited[w] \leftarrow$

true

$Q.enqueue(w)$

313

BFS — interpretacja i złożoność

Breadth-First Search odwiedza każdy wierzchołek co najwyżej raz, a każdą krawędź dokładnie dwa razy w grafie nieskierowanym, co daje złożoność $O(|V| + |E|)$. Algorytm wykorzystuje pamięć na kolejkę, która może w najgorszym scenariuszu zawierać nawet całą warstwę poziomą grafu. BFS gwarantuje najkrótsze ścieżki w grafach nieważonych, co czyni go niezbędnym w zadaniach planowania ruchu, mapowania lub analizy odległości.

314

Depth-First Search (DFS) – opis działania

Depth-First Search (DFS) to metoda przeszukiwania polegająca na schodzeniu jak najgłębiej wzdłuż dostępnych krawędzi. Dopiero gdy wierzchołek nie ma nieodwiedzonych sąsiadów, algorytm wraca wstecz. DFS może być realizowany zarówno rekurencyjnie, jak i iteracyjnie z użyciem stosu. Metoda ta jest szczególnie ważna przy wykrywaniu cykli, analizie komponentów silnie spójnych oraz budowie drzew rozpinających.

315

DFS — pseudokod (wersja rekurencyjna)

DFS(G):

dla każdego wierzchołka v w G :

$visited[v] \leftarrow false$

dla każdego wierzchołka v w G :

jeśli $visited[v] = false$:

DFS-Visit(v)

DFS-Visit(v):

$visited[v] \leftarrow true$

przetwórz(v)

dla każdego sąsiada w :

jeśli $visited[w] = false$:

DFS-Visit(w)

316

DFS — złożoność i własności

Podobnie jak BFS, Depth-First Search działa w czasie $O(|V| + |E|)$. Rekurencyjna wersja DFS tworzy drzewo przeszukiwania, które odzwierciedla strukturę połączeń. Zastosowanie DFS obejmuje wykrywanie cykli, sortowanie topologiczne w grafach skierowanych, znajdowanie mostów oraz punktów artykulacji w grafach, a także analizę spójności silnej. DFS działa głęboko, odkrywając zagnieżdżoną strukturę grafu.

317

Zastosowania BFS i DFS — porównanie

Breadth-First Search jest idealny do znajdowania najkrótszych ścieżek w grafach bez wag, natomiast Depth-First Search pozwala rozpoznać hierarchię i strukturę cykli. BFS jest metodą poziomą, a DFS metodą pionową; obie uzupełniają się nawzajem. W praktyce wybór algorytmu zależy od tego, czy bardziej interesuje nas odległość, czy struktura wnętrza grafu.

318

Algorytmy najkrótszych ścieżek — przegląd

Znalezienie najkrótszej ścieżki między wierzchołkami jest jednym z najważniejszych zadań grafowych. W zależności od rodzaju wag stosuje się różne algorytmy: algorytm Dijkstry dla grafów o nieujemnych wagach, algorytm Bellmana-Forda dla grafów z wagami ujemnymi oraz algorytm Floyda-Warshalla dla obliczania najkrótszych ścieżek między wszystkimi parami wierzchołków. Każdy algorytm ma swoje zastosowanie, ograniczenia i złożoność.

319

Algorytm Dijkstry – opis

Algorytm Dijkstry, znany jako Single Source Shortest Path (SSSP) dla nieujemnych wag, iteracyjnie wyznacza najkrótszą ścieżkę od jednego źródła do wszystkich innych wierzchołków. Wykorzystuje strukturę kolejki priorytetowej (kolejka priorytetowa była omówiona w wykładzie W6), która pozwala w każdym kroku wybierać wierzchołek o najmniejszej znanej odległości. Algorytm jest deterministyczny i niezwykle szybki w grafach rzadkich.

320

Algorytm Dijkstry – pseudokod

Dijkstra(G, s):

dla każdego wierzchołka v :

$\text{dist}[v] \leftarrow \infty$

$\text{dist}[s] \leftarrow 0$

$PQ \leftarrow$ kolejka priorytetowa
zawierająca $(s, 0)$

dopóki PQ nie jest pusta:

$(u, d) \leftarrow PQ.\text{extract-min}()$

jeśli $d > \text{dist}[u]$:

kontynuuj

dla każdej krawędzi (u, w) o

wadze c :

jeśli $\text{dist}[w] > \text{dist}[u] + c$:

$\text{dist}[w] \leftarrow \text{dist}[u] + c$

$PQ.\text{insert}(w, \text{dist}[w])$

Dijkstra – interpretacja

Algorytm wybiera najbliższy wierzchołek spośród tych jeszcze nieprzetworzonych i relaksuje wszystkie wychodzące z niego krawędzie. Relaksacja oznacza aktualizację odległości, jeśli nowa ścieżka jest krótsza. Dzięki wykorzystaniu kolejki priorytetowej operacja wyboru najbliższego wierzchołka jest niezwykle efektywna. Warunkiem poprawności jest brak ujemnych wag.

322

Dijkstra — złożoność

Złożoność algorytmu Dijkstry zależy od użytej struktury kolejki priorytetowej. Przy zastosowaniu kolejki binarnej z dołączania i pobierania minimum otrzymujemy złożoność $O((|V| + |E|) \log |V|)$. W grafach rzadkich dominuje składnik $|E| \log |V|$. Istnieją warianty szybsze z wykorzystaniem kopców Fibonacciego, ale rzadko stosuje się je w praktyce.

323

Algorytm Bellmana-Forda – opis

Algorytm Bellmana-Forda to metoda znajdowania najkrótszych ścieżek z jednego źródła w grafach z wagami ujemnymi. W przeciwieństwie do algorytmu Dijkstry, algorytm ten wykonuje relaksację wszystkich krawędzi wielokrotnie – dokładnie $|V|-1$ razy. Dzięki temu potrafi poprawnie wykryć i zgłosić istnienie cykli o ujemnej sumie wag, co czyni go bardziej ogólnym, choć wolniejszym.

324

Bellman-Ford — pseudokod

BellmanFord(G, s):

dla każdego wierzchołka v :

$\text{dist}[v] \leftarrow \infty$ dla i od 1 do $|V|-1$:

$\text{dist}[s] \leftarrow 0$

dla każdej krawędzi (u, w) o wadze c :

jeśli $\text{dist}[w] > \text{dist}[u] + c$:

$\text{dist}[w] \leftarrow \text{dist}[u] + c$

dla każdej krawędzi (u, w) o wadze c :

jeśli $\text{dist}[w] > \text{dist}[u] + c$:

zgłoś cykl ujemny

325

Bellman-Ford — interpretacja i złożoność

Algorytm działa w czasie $O(|V| \cdot |E|)$, co jest znacząco wolniejsze niż algorytm Dijkstry. Mimo to jest niezastąpiony w sytuacjach, gdy wagi krawędzi mogą być ujemne lub gdy analizujemy stabilność sieci i zależności. Wykrywanie cykli ujemnych stanowi kluczowy krok w systemach finansowych i algorytmach arbitrażowych.

326

Algorytm Floyda-Warshalla – opis

Algorytm Floyda-Warshalla rozwiązuje problem znajdowania najkrótszych ścieżek między wszystkimi parami wierzchołków (All-Pairs Shortest Path, APSP). Polega na dynamicznym porównywaniu wszystkich ścieżek przechodzących przez kolejne wierzchołki. Algorytm jest oparty na metodzie Programowania Dynamicznego (Dynamic Programming, DP) i tworzy tablicę odległości aktualizowaną dla każdego wierzchołka pośredniego.

327

Floyd-Warshall – pseudokod

FloydWarshall(G):

dla wszystkich par (i, j):

jeśli istnieje krawędź $i \rightarrow j$ o wadze c :

$\text{dist}[i][j] \leftarrow c$

inaczej jeśli $i = j$:

$\text{dist}[i][i] \leftarrow 0$

inaczej:

$\text{dist}[i][j] \leftarrow \infty$

dla k od 1 do $|V|$:

dla i od 1 do $|V|$:

dla j od 1 do $|V|$:

jeśli $\text{dist}[i][j] >$

$\text{dist}[i][k] + \text{dist}[k][j]$:

$\text{dist}[i][j] \leftarrow$

$\text{dist}[i][k] + \text{dist}[k][j]$

328

Floyd-Warshall – złożoność

Algorytm działa w czasie $O(|V|^3)$, co czyni go odpowiednim dla grafów małych lub średnich. Jego implementacja jest prosta i niezwykle regularna, co sprawia, że łatwo go zoptymalizować. Floyd-Warshall może również wykrywać cykle ujemne — jeśli $\text{dist}[i][i] < 0$ dla jakiegoś i , w grafie występuje cykl o ujemnej wadze.

329

Minimalne Drzewo Rozpinające (MST) – wprowadzenie

Minimalne Drzewo Rozpinające (Minimum Spanning Tree, MST) to zbiór krawędzi łączący wszystkie wierzchołki grafu w sposób spójny, przy minimalnej sumie wag krawędzi. Stosuje się je w planowaniu sieci, optymalizacji połączeń, minimalizacji kosztów oraz projektowaniu infrastruktury. Dwa najbardziej znane algorytmy to algorytm Kruskala i algorytm Prima.

330

Algorytm Kruskala – idea

Algorytm Kruskala operuje na liście krawędzi posortowanych według wag. Zaczyna od pustego zbioru i dodaje kolejne najtańsze krawędzie, o ile nie tworzą cyklu. Operacja wykrywania cykli realizowana jest za pomocą struktury Union-Find, znanej również jako Disjoint Set Union (DSU). Algorytm działa efektywnie w grafach rzadkich.

331

Kruskal — pseudokod

Kruskal(G):

sortuj wszystkie krawędzie według wag

DSU.init(|V|)

MST $\leftarrow \emptyset$

dla każdej krawędzi (u, w) o wadze c:

jeśli DSU.find(u) $\neq$ DSU.find(w):

DSU.union(u, w)

MST $\leftarrow$ MST $\cup \{(u, w)\}$

zwróć MST

Kruskal — złożoność i interpretacja

Dominującą operacją jest sortowanie krawędzi, które wymaga $O(|E| \log |E|)$. Operacje Union-Find dzięki optymalizacjom takim jak spłaszczanie ścieżek wykonywane są praktycznie w czasie stałym amortyzowanym. Algorytm jest bardzo prosty i niezwykle skuteczny, szczególnie w reprezentacji listy krawędzi.

333

Algorytm Prima – idea

Algorytm Prima rozpoczyna budowę Minimalnego Drzewa Rozpinającego (MST) od losowego wierzchołka i w każdym kroku dodaje najtańszą krawędź wychodzącą z już utworzonego drzewa. Do obsługi wyboru minimalnej krawędzi wykorzystywana jest kolejka priorytetowa, podobnie jak w algorytmie Dijkstry. Algorytm działa szczególnie dobrze w grafach gęstych.

334

Prim – pseudokod

Prim(G, s):

dla każdego wierzchołka v :

$\text{key}[v] \leftarrow \infty$

$\text{parent}[v] \leftarrow \text{null}$

$\text{key}[s] \leftarrow 0$

$PQ \leftarrow$ kolejka priorytetowa zawierająca $(s, 0)$

335

Prim – pseudokod

dopóki PQ nie jest pusta:

$(u, k) \leftarrow \text{PQ.extract-min}()$

dla każdej krawędzi (u, w) o wadze c :

jeśli $c < \text{key}[w]$:

$\text{key}[w] \leftarrow c$

$\text{parent}[w] \leftarrow u$

$\text{PQ.insert}(w, c)$

zwróć zbiór $\text{parent}[]$ reprezentujący MST

Prim – złożoność

Zależnie od implementacji kolejki priorytetowej złożoność wynosi $O((|V| + |E|) \log |V|)$. Algorytm jest idealny dla grafów prezentowanych w postaci macierzy sąsiedztwa, gdzie łatwo jest odczytać wszystkie krawędzie. Wynikiem algorytmu jest drzewo o minimalnej sumie wag, które zachowuje pełną spójność grafu.

337

Porównanie algorytmów Kruskala i Prima

Kruskal jest bardziej naturalny dla grafów rzadkich i reprezentacji w postaci listy krawędzi, natomiast Prim lepiej sprawdza się w grafach gęstych lub reprezentowanych macierzą sąsiedztwa. Oba algorytmy gwarantują optymalność i tworzą minimalną strukturę obejmującą wszystkie wierzchołki. Wybór konkretnego algorytmu zależy od liczby krawędzi i rodzaju reprezentacji.

338

Podsumowanie wykładu

Wykład przedstawił najważniejsze techniki analizy grafów: reprezentacje danych, przeszukiwanie wszerz i w głąb, algorytmy znajdowania najkrótszych ścieżek oraz konstrukcji Minimalnego Drzewa Rozpinającego (MST). Algorytmy BFS i DFS stanowią podstawę większości metod grafowych. Algorytm Dijkstry jest efektywny dla wag nieujemnych, Bellman-Ford dla ujemnych, a Floyd-Warshall dla wszystkich par wierzchołków.

339

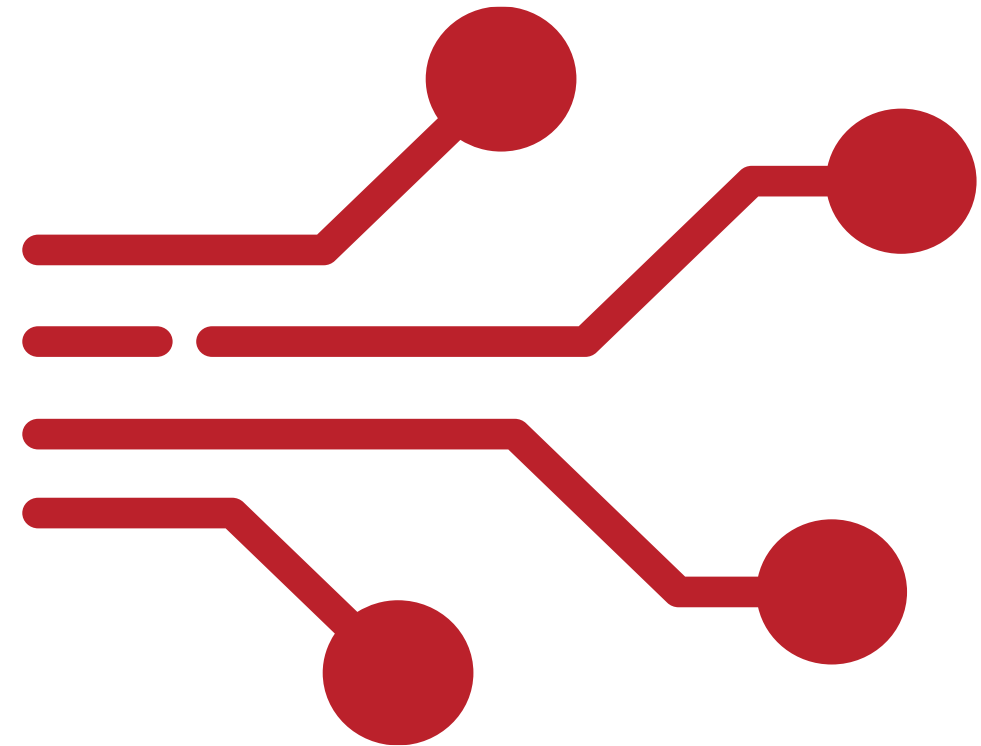
Podsumowanie wykładu

Algorytmy Kruskala i Prima umożliwiają budowanie tanich struktur łączących wszystkie węzły. Poznanie tych technik pozwala zrozumieć strukturę złożonych sieci oraz projektować efektywne rozwiązania oparte na grafach.

Algorytmy grafowe są fundamentem współczesnej informatyki. Omówione metody tworzą trzon większości rozbudowanych systemów przetwarzania danych. Zrozumienie ich działania i analizy pozwala utrwalić podstawy algorytmiki i przygotować się do bardziej zaawansowanych tematów, takich jak algorytmy przepływu, analiza sieci czy teoria grafów losowych.

340

Wybrane algorytmy tekstowe



341

Plan wykładu

W ramach niniejszego wykładu omówimy następujące zagadnienia:

- Wprowadzenie do problemów tekstowych i modeli danych tekstowych.
- Podstawowe definicje: tekst, wzorzec, dopasowanie, niezgodności.
- Wyszukiwanie naiwne i właściwości jego działania.
- Algorytm Knutha–Morrisa–Pratta oraz funkcja prefiksowa.
- Algorytm Boyera–Moore’a i heurystyki przesunięcia wzorca.
- Algorytmy oparte na automatach skończonych i tablicach przejść.
- Struktury sufiksowe: tablica sufiksowa, LCP, algorytm Kasai.
- Wyszukiwanie w tablicy sufiksowej.
- Metody kompresji oparte na analizie tekstu: RLE, LZ77, LZ78, Huffman.

342

Wprowadzenie

Algorytmy tekstowe stanowią fundament współczesnego przetwarzania informacji. Każde wyszukiwanie w dokumentach, filtrowanie logów, detekcja znaków w sekwencjach DNA, analizowanie kodu źródłowego czy praca wyszukiwarek internetowych opiera się na metodach efektywnego dopasowywania wzorców w dużych zbiorach tekstu. W odróżnieniu od wielu innych obszarów algorytmiki, tutaj kluczową rolę odgrywa struktura danych, ponieważ tekst może mieć długość milionów znaków.

343

Wprowadzenie

Z tego powodu celem algorytmów tekstowych jest minimalizacja niepotrzebnych porównań oraz wykorzystanie informacji o wcześniejszych dopasowaniach do przyspieszenia działania. Wprowadzone zostaną narzędzia zarówno proste, jak i wysoko zaawansowane, które stanowią podstawę do zrozumienia wyszukiwania pełnotekstowego, analiz lingwistycznych oraz systemów indeksowania.

344

Model danych tekstowych

Przetwarzanie tekstu rozpoczyna się od przyjęcia formalnego modelu danych. Tekst traktujemy jako ciąg znaków $T[1..n]$, natomiast wzorzec jako krótszy ciąg $P[1..m]$, przy czym zazwyczaj $m \ll n$. Celem algorytmu jest odnalezienie wszystkich pozycji i , dla których fragment $T[i..i + m - 1]$ jest równy P . W niektórych wariantach dopuszcza się pewne niezgodności, jednak w tym wykładzie koncentrujemy się głównie na dopasowaniu dokładnym.

345

Model danych tekstowych

Model ten jest wspólny dla wszystkich prezentowanych algorytmów, lecz każdy z nich wykorzystuje go w odmienny sposób, od prostego porównywania znaków po zaawansowane indeksy. Analiza efektywności polega zazwyczaj na oszacowaniu liczby porównań lub operacji przetwarzania znaków, ponieważ operacje te są najbardziej kosztowne przy dużych tekstach.

346

Problem wyszukiwania wzorca

Wyszukiwanie wzorca polega na znalezieniu wszystkich miejsc w tekście, w których pojawia się sekwencja znaków będąca wzorcem. Problem ten wydaje się prosty, lecz jego bezpośrednia implementacja prowadzi do poważnych ograniczeń efektywności. Wysoka długość tekstu sprawia, że nie każdy algorytm może stosować się do realnych zastosowań.

347

Problem wyszukiwania wzorca

Celem metod, które poznamy, jest redukcja liczby niepotrzebnych porównań, wykorzystywanie faktu wcześniejszych niezgodności oraz budowanie struktur pozwalających na skakanie przez duże fragmenty tekstu. To właśnie różne sposoby redukcji liczby porównań stanowią podstawę odróżniającą algorytmy naiwne od bardziej zaawansowanych metod, takich jak KMP czy Boyer–Moore.

348

Naiwne wyszukiwanie wzorca: wprowadzenie

Najprostszą metodą wyszukiwania wzorca jest algorytm naiwny, który sprawdza każdą możliwą pozycję tekstu. Dla każdego i od 1 do $n - m + 1$ porównuje kolejne znaki wzorca z odpowiadającymi im znakami tekstu, przerywając dopiero przy niezgodności lub po znalezieniu pełnego dopasowania. Choć intuicyjny i niezwykle prosty, algorytm ten może wykonywać nawet m porównań dla każdej z n pozycji, co prowadzi do złożoności $O(nm)$. Jego główną zaletą jest prostota implementacji oraz brak przygotowania dodatkowych struktur, jednak w przypadku dużych tekstów staje się on zupełnie niepraktyczny. W kolejnych slajdach wprowadzimy jego formalny pseudokod oraz analizę własności.

349

Naiwne wyszukiwanie: pseudokod

Pseudokod algorytmu naiwnego w sposób bezpośredni odzwierciedla jego ideę. Dla każdej możliwej pozycji i w tekście rozpoczynamy porównywanie wzorca z tekstem od pierwszego znaku. Jeśli w jakimś momencie napotkamy niezgodność, przerwijmy i przejdźmy do kolejnej pozycji $i+1$. Jeśli natomiast uda się porównać wszystkie m znaków, wówczas odnaleziono pełne dopasowanie. Mechanizm ten jest prosty, lecz jego słabością jest wykonywanie wielu zbędnych porównań, których można uniknąć, jeśli wykorzysta się wiedzę o strukturze wzorca.

350

Naiwne wyszukiwanie: pseudokod

NaiwneWyszukiwanie(T, P):

n = długość(T)

m = długość(P)

dla i od 1 do n - m + 1

j = 1

dopóki j ≤ m oraz T[i + j - 1] = P[j]

j = j + 1

jeśli j > m

zwróć i

zwróć „brak dopasowania”

351

Naiwne wyszukiwanie: analiza

Analiza algorytmu naiwnego opiera się na oszacowaniu liczby porównań znaków. W najgorszym przypadku każde potencjalne przesunięcie prowadzi do m porównań, a liczba takich przesunięć wynosi $n - m + 1$. Daje to złożoność $O(nm)$. W najlepszym przypadku, gdy pierwsze znaki wzorca różnią się od znaków tekstu, potrzebne jest tylko jedno porównanie, co prowadzi do złożoności $O(n)$. Jednak algorytm nie wykorzystuje informacji o wcześniejszych dopasowaniach, przez co może powtarzać pracę wykonaną wcześniej. Ta jego cecha jest punktem wyjścia dla bardziej zaawansowanych metod.

352

Od intuicji do optymalizacji

Wyszukiwanie naiwne działa, lecz jego efektywność jest uzależniona od struktury tekstu i wzorca. Jeśli wzorzec ma fragmenty powtarzalne, niektóre porównania będą wykonywane wielokrotnie. Efektywne algorytmy tekstowe starają się wykorzystać informację o wcześniej dopasowanych fragmentach, tak aby w momencie niezgodności nie wracać do początku wzorca. W praktyce oznacza to przesuwanie wzorca o więcej niż jeden znak, co zapewnia istotne przyspieszenie działania. W tej części wykładu poznamy sposoby wykorzystywania struktury wzorca, począwszy od funkcji prefiksowej w algorytmie KMP.

353

Algorytm Knutha–Morrisa–Pratta: idea

Algorytm KMP stanowi kluczowy przykład techniki unikania powtórnych porównań. Jego siła wynika z faktu, że analiza struktury samego wzorca pozwala określić, jak daleko można przesunąć wzorzec po wystąpieniu niezgodności. Zamiast cofania się i ponownego porównywania znaków, KMP wykorzystuje informację o najdłuższym właściwym prefiksie wzorca, który jest jednocześnie jego sufiksem. Dzięki temu możliwe jest przesunięcie wzorca, zachowując zgodność z dotychczas dopasowaną częścią. Kluczowym narzędziem algorytmu jest funkcja prefiksowa π , która dla każdego j określa długość najdłuższego prefiksu P zgodnego z sufiksem $P[1..j]$.

354

Funkcja prefiksowa: definicja

Funkcja prefiksowa π opisuje strukturę powtórzeń we wzorcu. Dla każdej pozycji j w wzorcu obliczamy, jaka jest największa długość prefiksu $P[1..k]$, który jednocześnie stanowi sufix $P[1..j]$. Wartość $\pi[j]$ mówi, ile początkowych znaków wzorca jest zgodnych z końcowym fragmentem $P[1..j]$. Funkcja ta nie analizuje tekstu — dotyczy tylko struktury wzorca i jest obliczana w czasie liniowym. Pozwala na skrócenie pracy algorytmu wyszukiującego poprzez wskazanie właściwego przesunięcia wzorca po niezgodności.

355

Obliczanie funkcji prefiksowej: intuicja

Aby obliczyć π , analizujemy wzorzec od lewej do prawej, stopniowo wydłużając część rozpatrywaną. Dla każdej pozycji j staramy się wydłużyć dopasowanie do maksymalnej możliwej długości. Jeśli znak $P[j]$ zgadza się z $P[k + 1]$, wydłużamy zgodność, natomiast w przeciwnym przypadku cofamy się do wartości $\pi[k]$, próbując ponownie. Proces ten przypomina dopasowanie samego wzorca do siebie, lecz wykonywane jest w jednym przebiegu, gwarantując złożoność $O(m)$.

356

Funkcja prefiksowa: pseudokod

ObliczPrefiks(P):

$m = \text{długość}(P)$

$\pi[1] = 0$

$k = 0$

dla j od 2 do m

dopóki $k > 0$ oraz $P[k+1] \neq P[j]$

$k = \pi[k]$

jeśli $P[k+1] = P[j]$

$k = k + 1$

$\pi[j] = k$

zwróć π

Funkcja prefiksowa: pseudokod

Pseudokod ten obrazuje, że długość zgodności k jest wydłużana jedynie wtedy, gdy znaki odpowiadają.

W przypadku niezgodności wracamy do wartości $\pi[k]$, co minimalizuje liczbę porównań.

358

Algorytm KMP: dopasowanie

Po obliczeniu funkcji prefiksowej przechodzimy do właściwego dopasowywania wzorca do tekstu. Przechodzimy przez tekst znak po znaku, starając się dopasowywać wzorzec w sposób ciągły. Jeśli pojawia się niezgodność, nie wracamy do początku wzorca, lecz korzystamy z π , aby ustalić, jaką część dopasowania można zachować. W ten sposób gwarantujemy, że każdy znak tekstu jest przetwarzany najwyżej dwukrotnie, co daje złożoność $O(n + m)$.

359

Algorytm KMP: pseudokod

KMP(T, P):

π = ObliczPrefiks(P)

n = długość(T)

m = długość(P)

q = 0

dla i od 1 do n

dopóki q > 0 oraz P[q+1] ≠ T[i]

q = π [q]

jeśli P[q+1] = T[i]

q = q + 1

jeśli q = m

zwróć i - m + 1

q = π [q]

zwróć „brak dopasowania”

360

KMP: analiza

Złożoność czasowa KMP wynika z faktu, że wskaźnik q cofa się tylko zgodnie z wartością funkcji prefiksowej. Oznacza to, że zarówno w krokach dopasowania, jak i w krokach cofania, każda pozycja wzorca jest odwiedzana jedynie kilka razy. Sumarycznie daje to złożoność $O(n + m)$. Algorytm ten usuwa redundantne porównania, co w praktyce stanowi ogromny skok efektywności względem metody naiwnej. Wyszukiwanie wzorca w bardzo długich tekstach, nawet liczących setki milionów znaków, jest możliwe dzięki takiej konstrukcji.

361

Kiedy KMP jest szczególnie użyteczny?

KMP sprawdza się najlepiej tam, gdzie wzorzec ma złożoną strukturę wewnętrzną, a tekst zawiera wiele powtórzeń znaków. W przypadku wzorców o wielu powtarzających się fragmentach funkcja prefiksowa umożliwia natychmiastową reakcję na niezgodność. Metoda ta jest niezastąpiona, gdy poszukujemy wielu dopasowań w ogromnych strumieniach danych lub w systemach, które nie mogą pozwolić sobie na powtarzanie pracy. W praktyce algorytm znajduje zastosowanie w edytorach tekstu, narzędziach przetwarzania logów czy mechanizmach skanowania sekwencji biologicznych.

362

Algorytm Boyera–Moore’a: motywacja

Boyer–Moore stanowi jeden z najbardziej efektywnych i praktycznych algorytmów wyszukiwania wzorca. Zasadniczą różnicą względem KMP jest fakt, że porównania wykonywane są od końca wzorca. Dzięki temu niezgodności wykrywane są jak najwcześniej, a przesunięcia wzorca mogą być bardzo duże. Algorytm wykorzystuje dwie heurystyki: złego znaku i dobrego sufiksu, które razem pozwalają na pomijanie dużych fragmentów tekstu. W wielu praktycznych zastosowaniach Boyer–Moore jest znacznie szybszy od metod liniowych, mimo że jego najgorszy przypadek pozostaje kwadratowy.

363

Heurystyka złego znaku: idea

Heurystyka złego znaku polega na analizie znaków w tekście, które prowadzą do niezgodności. Jeśli ostatni znak wzorca różni się od aktualnego znaku w tekście, możemy przesunąć wzorzec tak, aby ostatnie wystąpienie tego znaku w P znalazło się pod aktualnym znakiem tekstu. Jeśli taki znak nie istnieje we wzorcu, wzorzec można przesunąć o całą jego długość. Kluczem jest wstępne przygotowanie tablicy zawierającej ostatnie pozycje wystąpienia każdego znaku.

364

Heurystyka złego znaku: przykład

Założmy, że dopasowujemy wzorzec ABAC do tekstu i znajdujemy niezgodność pod literą X. Jeśli X nie pojawia się we wzorcu, wzorzec przesuwamy o 4 znaki. Jeśli natomiast X jest obecne na pozycji 2 we wzorcu, przesunięcie ustawiamy tak, aby wzorzec został wyrównany do tej pozycji. Dzięki temu redukujemy liczbę niepotrzebnych porównań, szczególnie gdy tekst zawiera znaki rzadkie lub nieobecne we wzorcu.

365

Heurystyka złego znaku: tablica

Tablicę złego znaku tworzymy w czasie proporcjonalnym do alfabetu. Dla każdego znaku alfabetu obliczamy jego ostatnie wystąpienie we wzorcu lub wartość -1 , jeśli znak nie występuje. Tablica ta jest wykorzystywana przy każdej niezgodności, aby określić minimalne sensowne przesunięcie wzorca. W praktyce efektywność tej heurystyki zależy od rozkładu znaków w tekście.

366

Heurystyka dobrego sufiksu: idea

Heurystyka dobrego sufiksu wykorzystuje fakt, że jeśli końcowa część wzorca została dopasowana, lecz wcześniej pojawiła się niezgodność, część ta może wystąpić również w innej części wzorca. W takim przypadku wzorzec można przesunąć tak, aby kolejne dopasowanie do tego sufiksu znalazło się na odpowiedniej pozycji. Jeżeli taki sufiks nie występuje w innej części wzorca, przesuwamy wzorzec tak, aby najdłuższy możliwy prefiks był zgodny z sufiksem dopasowanego fragmentu.

367

Boyer–Moore: pseudokod

```
BoyerMoore(T, P):  
    przygotuj tablicę złego znaku  
    przygotuj tablicę dobrego sufiksu  
    i = m  
    dopóki i ≤ n  
        j = m  
        dopóki j ≥ 1 i P[j] = T[i - m + j]  
            j = j - 1  
        jeśli j = 0  
            zwróć i - m + 1  
        i = i + maks(przesunięcie_złego_znaku,  
przesunięcie_dobrego_sufiksu)
```


Boyer–Moore: analiza

Choć najgorszy przypadek Boyera–Moore’a to $O(nm)$, w praktyce algorytm ten jest jednym z najszybszych. Jego efektywność wynika z faktu, że porównuje znaki od końca i potrafi przesuwac wzorzec o duże wartości. Na typowych danych tekstowych często działa w czasie zbliżonym do liniowego, szczególnie gdy alfabet jest duży, a wzorzec zawiera wiele różnych znaków.

369

Algorytm automatu skończonego: idea

Wyszukiwanie wzorca można również zrealizować za pomocą deterministycznego automatu skończonego (DFA). Automat analizuje tekst znak po znaku, przechodząc do kolejnych stanów na podstawie funkcji przejścia. Stan odpowiada długości dopasowanej części wzorca. Jeśli automat osiąga stan m , oznacza to znalezienie dopasowania. Kluczem jest przygotowanie tablicy przejść δ , która wskazuje, do jakiego stanu przechodzi automat po odczytaniu znaku.

370

Automat skończony: konstrukcja

Aby zbudować automat, analizujemy wzorzec i określamy dla każdego stanu q i każdego znaku alfabetu a , jaką jest największa długość prefiksu P , która zgadza się z sufiksem $P[1..q] \cdot a$. Procesy te są zbliżone do obliczania funkcji prefiksowej, lecz wymagają analizy dla całego alfabetu. Złożoność przygotowania automatu wynosi $O(m \cdot |\Sigma|)$, natomiast wyszukiwanie odbywa się już w czasie $O(n)$.

371

Automat skończony: pseudokod

AutomatWzorcowy(T , δ , m):

$q = \emptyset$

dla i od 1 do n

$q = \delta[q][T[i]]$

jeśli $q = m$

zwróć $i - m + 1$

zwróć „brak dopasowania”

372

Porównanie automatu i KMP

Choć automat skończony oraz KMP wykorzystują różne struktury danych, oba realizują tę samą ideę: wyszukiwanie liniowe bez powtarzania pracy. Różnica polega na tym, że automat wymaga analizy całego alfabetu, natomiast KMP ogranicza się do wzorca. W praktyce automat jest stosowany rzadziej, ze względu na większe koszty przygotowania, lecz jest szczególnie efektywny przy wielokrotnych wyszukiwaniach.

373

Tablice sufiksowe: wprowadzenie

Tablica sufiksowa to struktura indeksująca tekst, która pozwala na wyszukiwanie wzorców w czasie logarytmicznym. Zawiera wszystkie sufiksy tekstu T uporządkowane leksykograficznie. Jeśli wzorzec występuje w T , to pojawi się jako prefiks jednego z sufiksów. Wyszukiwanie polega na wykonaniu binarnego przeszukiwania tablicy sufiksów. Najważniejszą cechą tablicy sufiksowej jest jej kompaktowość w porównaniu do drzewa sufiksowego.

374

Budowa tablicy sufiksowej (metody klasyczne)

Najprostsza metoda budowy tablicy sufiksowej polega na posortowaniu wszystkich sufiksów tekstu. Jest to jednak kosztowne, ponieważ porównywanie sufiksów może być długie. Takie podejście prowadzi do złożoności $O(n^2 \log n)$. Aby poprawić efektywność, stosuje się technikę sortowania podwójnych rang, dzięki której można uzyskać złożoność $O(n \log n)$. W późniejszych badaniach opracowano metody liniowe, takie jak SA-IS.

375

Budowa tablicy sufiksowej (metoda rang)

Metoda rang polega na przypisywaniu każdemu sufiksowi podwójnej wartości: rangi bieżącej oraz rangi fragmentu położonego o k znaków dalej. Na początku rangi odpowiadają wartościom znaków. Następnie sukcesywnie podwajamy długość analizowanych fragmentów, sortując sufiksy według par rang. Po $\log n$ krokach uzyskujemy pełne posortowanie sufiksów.

376

Tablica sufiksowa: przykład

Przykładowo dla tekstu *BANANA* sufiksy to:

BANANA, ANANA, NANA, ANA, NA, A.

Posortowane leksykograficznie dają kolejność:

A, ANA, ANANA, BANANA, NA, NANA.

Tablica sufiksowa zawiera indeksy: 6, 4, 2, 1, 5, 3. Dzięki tej strukturze wyszukiwanie wzorca odbywa się poprzez binarne przeszukiwanie tekstu uporządkowanego po sufiksach.

377

Tablica LCP: idea

LCP (Longest Common Prefix) to tablica przechowująca długość najdłuższego wspólnego prefiksu sąsiednich sufiksów w tablicy sufiksowej. Dzięki niej można analizować podobieństwa między fragmentami tekstu, redukować liczbę porównań przy wyszukiwaniu i przyspieszać budowę dodatkowych struktur indeksujących. Algorytm Kasai pozwala obliczyć LCP w czasie liniowym, korzystając z odwrotnej tablicy sufiksowej.

378

Algorytm Kasai: pseudokod

```
ObliczLCP(T, SA):  
  n = długość(T)  
  rank[SA[i]] = i dla  
  każdego i  
  k = 0  
  dla i od 1 do n  
    jeśli rank[i] = n  
      k = 0  
      kontynuuj  
  j = SA[rank[i] + 1]
```

```
  dopóki i + k ≤ n oraz j  
  + k ≤ n oraz T[i+k] =  
  T[j+k]  
    k = k + 1  
  LCP[rank[i]] = k  
  jeśli k > 0  
    k = k - 1  
  zwróć LCP
```

379

Wyszukiwanie wzorca w tablicy sufiksowej

Aby wyszukać wzorzec P w tablicy sufiksowej, wykonujemy binarne wyszukiwanie na uporządkowanych sufiksach. W każdym kroku porównujemy wzorzec z sufiksem zaczynającym się od $SA[mid]$. Porównania te mogą być przyspieszone dzięki tablicy LCP. Jeśli wzorzec jest prefiksem sufiksu, odnaleziono dopasowanie. W przeciwnym przypadku analizujemy stronę przedziału, która może zawierać potencjalne dopasowania.

380

Złożoność wyszukiwania w SA

Najważniejszą zaletą tablicy sufiksowej jest koszt wyszukiwania: $O(m \log n)$, gdzie m to długość wzorca, a n długość tekstu. Jeśli wykorzystujemy dodatkowe struktury lub tablicę LCP, koszt może zbliżyć się do $O(m + \log n)$. Dlatego tablice sufiksowe są podstawą indeksów tekstowych wykorzystywanych przez wyszukiwarki internetowe i systemy analizy genomów.

381

Kompresja tekstu: wprowadzenie

Algorytmy tekstowe obejmują również techniki kompresji, które wykorzystują analizę struktury danych tekstowych. Kompresja polega na redukcji liczby bitów potrzebnych do przechowania tekstu. W tej części wykładu przedstawimy metody wykorzystujące powtarzalność danych, analizę częstotliwości znaków oraz dynamiczne odnajdywanie reguł w tekście. Skupimy się na czterech najważniejszych metodach: RLE, LZ77, LZ78 oraz kodowaniu Huffmana.

382

RLE: kompresja powtórzeń

RLE (Run-Length Encoding) to najprostsza metoda kompresji oparta na powtarzalności znaków. Polega na zapisie ciągów identycznych znaków jako par (znak, długość serii). Na przykład ciąg AAAAABBB zapisywany jest jako A5B3. Metoda ta jest niezwykle efektywna dla danych zawierających długie serie, lecz nie nadaje się do kompresji ogólnych tekstów o dużej zmienności.

383

Kodowanie Huffmana: idea

Kodowanie Huffmana polega na przypisaniu krótkich kodów binarnych znakom o dużej częstotliwości występowania oraz dłuższych kodów znakom rzadkim. Tworzony jest pełny binarny trie, w którym każdy znak odpowiada liściowi, a długość ścieżki odpowiada długości kodu. Konstrukcja drzewa Huffmana wykorzystuje kolejkę priorytetową, w której przechowuje się węzły uporządkowane według częstotliwości.

384

Kodowanie Huffmana: pseudokod

Huffman(znaki, częstotliwości):
utwórz kolejkę priorytetową Q
dla każdego znaku z częstotliwością
utwórz węzeł i dodaj do Q
dopóki Q zawiera więcej niż jeden węzeł
usuń dwa węzły o najmniejszej częstotliwości
połącz je tworząc nowy węzeł
dodaj nowy węzeł do Q
zwróć korzeń drzewa Huffmana

385

LZ77: idea kompresji słownikowej

LZ77 jest metodą opartą na przesuwającym się oknie, w którym zapisujemy przeszłą część tekstu. Kompresja polega na odnajdowaniu najdłuższego prefiksu bieżącego tekstu, który wystąpił już wcześniej. Wynikiem jest para (długość, odległość), która opisuje, jak daleko wstecz znajduje się powtarzający się fragment. Jeśli znak nie występuje wcześniej, zapisujemy go bezpośrednio. Metoda ta stanowi fundament nowoczesnych algorytmów kompresji, takich jak DEFLATE.

386

LZ77: przykład i zastosowania

Jeśli tekst zawiera wiele powtarzających się sekwencji, LZ77 potrafi kompresować go bardzo efektywnie. Dzięki analizie lokalnego słownika metoda ta doskonale radzi sobie z danymi, które mają charakter naturalny, jak dokumenty tekstowe. W praktycznych implementacjach stosuje się dodatkowe ograniczenia okna oraz mechanizmy efektywnego wyszukiwania najdłuższego dopasowania.

387

LZ78: idea

LZ78 rozszerza ideę LZ77 o budowę słownika globalnego. Tekst analizowany jest od początku, a wszelkie nowe napotkane sekwencje są dodawane do słownika. Bieżący fragment tekstu jest reprezentowany jako para [indeks hasła słownikowego, kolejny znak]. Słownik rośnie w trakcie przetwarzania, a jego struktura stanowi wstęp do bardziej zaawansowanych metod, takich jak LZW.

388

LZ78: przykład

Przykładowo, dla tekstu *ABAABABA*, LZ78 zbuduje słownik:

A, B, AA, AB, ABA.

Tekst zostaje zamieniony na sekwencję indeksów, które odnoszą się do tych haseł. Idea polega na wykorzystywaniu wzrastającej pamięci o wcześniejszych sekwencjach, co poprawia efektywność kompresji.

389

Podsumowanie algorytmów kompresji

Metody kompresji opierają się na dwóch kluczowych ideach: analizie powtórzeń w tekście oraz analizie częstotliwości występowania znaków. RLE i Huffman wykorzystują lokalne powtarzalności i częstotliwości, natomiast LZ77 i LZ78 tworzą dynamiczne słowniki. Każda metoda ma inne zastosowanie, zależnie od charakteru danych.

390

Zestawienie metod wyszukiwania wzorca

Na tym wykładzie poznaliśmy różnorodne metody wyszukiwania wzorca: naiwną, KMP, Boyera–Moore’a, automat skończony oraz metody indeksowania oparte na tablicy sufiksowej. Każda z nich wykorzystuje inną obserwację dotyczącą przetwarzania tekstu. KMP eliminuje powtarzanie pracy dzięki funkcji prefiksowej, Boyer–Moore wykorzystuje analizę końca wzorca, a tablica sufiksowa jest globalnym indeksem tekstu.

391

Główne różnice między metodami

Metody te różnią się nie tylko złożonością, ale także strukturą danych, które wymagają przygotowania. Algorytmy liniowe, takie jak KMP, działają bez dodatkowej pamięci po wstępnym przetworzeniu wzorca. Algorytmy indeksujące wymagają jednak budowy dużych struktur, co jest kosztowne, lecz pozwala na wykonywanie wielokrotnych wyszukiwań bardzo szybko. W praktyce dobór metody zależy od tego, czy priorytetem jest szybka odpowiedź jednorazowa, czy wielokrotne zapytania.

392

Algorytmy tekstowe we współczesnych systemach

Współczesne systemy informatyczne niemal w każdym aspekcie wykorzystują algorytmy tekstowe. Wyszukiwarki internetowe korzystają z indeksów sufiksowych, edytory tekstów wykorzystują KMP i Boyera–Moore’a przy wyszukiwaniu w dokumentach, a kompresory oparte na LZ77 znajdują zastosowanie w przeglądarkach, archiwizatorach oraz transmisji danych. Zrozumienie tych algorytmów jest kluczowe dla budowy efektywnych narzędzi informatycznych.

393

Podsumowanie wykładu

Algorytmy tekstowe obejmują zarówno proste metody polegające na porównywaniu znaków, jak i zaawansowane techniki indeksowania i kompresji. Ich wspólnym celem jest redukcja kosztów przetwarzania danych tekstowych. Wykład przedstawił szeroką grupę metod, począwszy od wyszukiwania wzorca, poprzez automaty skończone, aż po tablice sufiksowe i algorytmy kompresji. Dzięki zrozumieniu tych narzędzi możliwe jest projektowanie nowoczesnych systemów, które radzą sobie z ogromnymi ilościami danych tekstowych.

394

Dziękuję za uwagę