

TEORETYCZNE PODSTAWY INFORMATYKI

22/01/2018

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Wykład 14b: Repetytorium

2

Złożoność obliczeniowa

- Notacja „*wielkie O*”
- Notacja Ω i Θ
- Rozwiązywanie rekurencji
- Złożoność obliczeniowa zamortyzowana

Złożoność obliczeniowa

3

- **Złożoność obliczeniowa:**
 - Jest to miara służąca do porównywania efektywności algorytmów.
 - Mamy dwa kryteria efektywności:
 - Czas,
 - Pamięć
- Do oceny efektywności stosujemy jednostki logiczne wyrażające związek między rozmiarem danych N (wielkość pliku lub tablicy) a ilością czasu T potrzebną na ich przetworzenie.

Złożoność asymptotyczna

4

- Funkcja wyrażająca zależność między N a T jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych
- Przybliżona miara efektywności to tzw. **złożoność asymptotyczna**.

Notacja „wielkie O”

5

Definicja:

$f(n)$ jest $O(g(n))$, jeśli istnieją liczby dodatnie c i n_0 takie że:
 $f(n) < c \cdot g(n)$ dla wszystkich $n \geq n_0$.

Przykład:

□ $f(n) = n^2 + 100n + \log_{10} n + 1000$

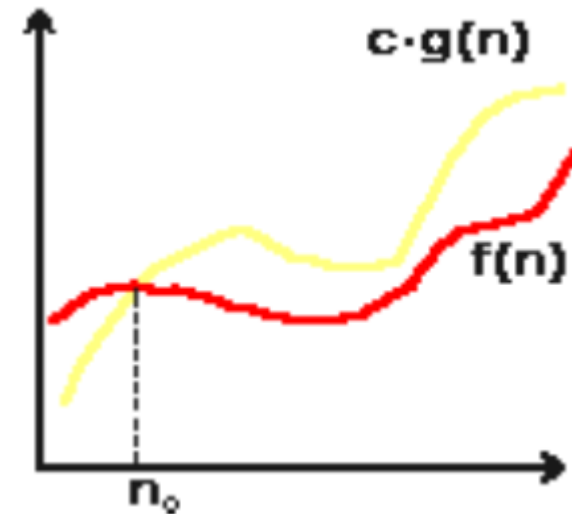
możemy przybliżyć jako:

$$f(n) \approx n^2 + 100n + O(\log_{10} n)$$

albo jako:

$$f(n) \approx O(n^2)$$

- Notacja „wielkie O” ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów.



Własności notacji „wielkie O”

6

- **Własność 1** (przechodność):
 - Jeśli $f(n)$ jest $O(g(n))$ i $g(n)$ jest $O(h(n))$, to $f(n)$ jest $O(h(n))$
- **Własność 2:**
 - Jeśli $f(n)$ jest $O(h(n))$ i $g(n)$ jest $O(h(n))$, to $f(n)+g(n)$ jest $O(h(n))$
- **Własność 3:**
 - Funkcja an^k jest $O(n^k)$
- **Własność 4:**
 - Funkcja n^k jest $O(n^{k+i})$ dla dowolnego dodatniego i

Własności notacji „wielkie O”

7

- Z tych wszystkich własności wynika, że dowolny wielomian jest „wielkie O” dla n podniesionego do najwyższej w nim potęgi, czyli :

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \text{ jest } O(n^k)$$

(jest też oczywiście $O(n^{k+i})$ dla dowolnego dodatniego i)

Własności notacji „wielkie O”

8

□ Własność 5:

□ Jeśli $f(n) = c g(n)$, to $f(n)$ jest $O(g(n))$

□ Własność 6:

□ Funkcja $\log_a n$ jest $O(\log_b n)$ dla dowolnych a i b większych niż 1

□ Własność 7:

□ $\log_a n$ jest $O(\log_2 n)$ dla dowolnego dodatniego a

Własności notacji „wielkie O”

9

- Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest **funkcja logarytmiczna**.
- Jeżeli można wykazać że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry.
- Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak **$O(\log_2 \log_2 n)$** czy **$O(1)$** ma praktyczne znaczenie.

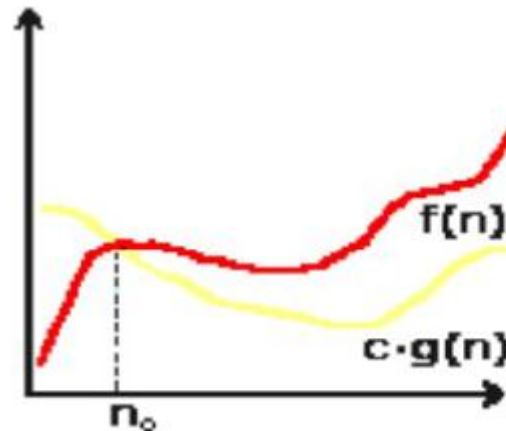
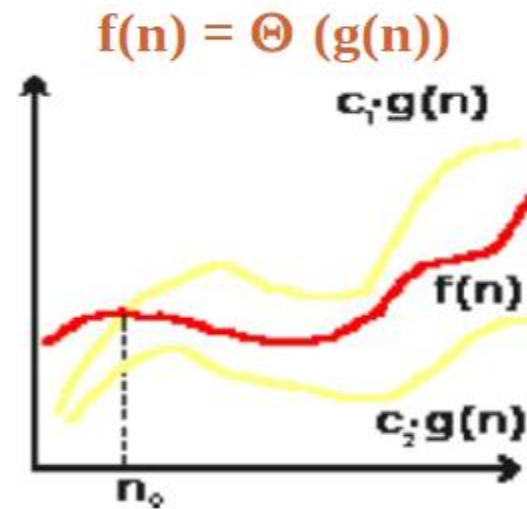
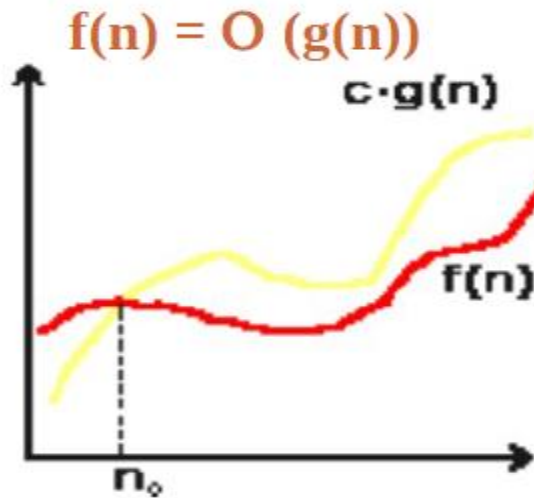
Notacja Ω i Θ

10

- Notacja „**wielkie O**” odnosi się do górnych ograniczeń funkcji. Istnieje symetryczna definicja dotycząca dolnych ograniczeń
- Definicja
 - ▣ $f(n)$ jest **$\Omega(g(n))$** , jeśli istnieją liczby dodatnie c i n_0 takie że, $f(n) \geq c g(n)$ dla wszystkich $n \geq n_0$.
- Równoważność
 - ▣ $f(n)$ jest **$\Omega(g(n))$** wtedy i tylko wtedy, gdy $g(n)$ jest $O(f(n))$
- Definicja
 - ▣ $f(n)$ jest **$\Theta(g(n))$** , jeśli istnieją takie liczby dodatnie c_1 , c_2 i n_0 takie że, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla wszystkich $n \geq n_0$.

Notacja O , Ω i Θ

11



$f(n) = \Omega(g(n))$

Czas działania programu

12

- Mając do dyspozycji definicję „**wielkie O**” oraz własności (1)-(7) będziemy mogli, wg. kilku prostych zasad, skutecznie analizować czasy działania większości programów spotykanych w praktyce.
- Efektywność algorytmów ocenia się przez szacowanie ilości czasu i pamięci potrzebnych do wykonania zadania, dla którego algorytm został zaprojektowany.
- Najczęściej jesteśmy zainteresowani **złożonością czasową**, mierzoną zazwyczaj liczbą przypisań i porównań realizowanych podczas wykonywania programu.
- Bardzo często interesuje nas tylko **złożoność asymptotyczna**, czyli czas działania dla dużej ilości analizowanych zmiennych.

Czas działania programu

13

- Dla konkretnych danych wejściowych jest wyrażony liczba wykonanych prostych (elementarnych) operacji lub “kroków”. Jest dogodne założenie że operacja elementarna jest maszynowo niezależna.
- Każde wykonanie i -tego wiersza programu jest równe c_i , przy czym c_i jest stałą.
- Kiedy algorytm zawiera rekurencyjne wywołanie samego siebie, jego czas działania można często opisać zależnością rekurencyjną (rekurencja) wyrażającą czas dla problemu rozmiaru n za pomocą czasu dla podproblemów mniejszych rozmiarów.
- Możemy więc użyć narzędzi matematycznych aby rozwiązać rekurencje i w ten sposób otrzymać oszacowania czasu działania algorytmu.

Rekurencja dla algorytmu typu „dziel i zwyciężaj”

14

- Rekurencja odpowiadającą czasowi działania algorytmu typu “dziel i zwyciężaj” opiera się na **podziale jednego poziomu rekursji na trzy etapy**.
 - Niech $T(n)$ będzie czasem działania dla jednego problemu rozmiaru n .
 - Jeśli rozmiar problemu jest odpowiednio mały, powiedzmy $n \leq c$ dla pewnej stałej c , to jego rozwiązanie zajmuje stały czas, co zapiszemy jako $\Theta(1)$.
 - Załóżmy że dzielimy problem na a podproblemów, każdy rozmiaru n/b . Jeśli $D(n)$ jest czasem dzielenia problemu na podproblemy, a $C(n)$ czasem scalania rozwiązań podproblemów w pełne rozwiązanie dla oryginalnego problemu, to otrzymujemy rekurencje
 - $T(n) = \Theta(1)$ jeśli $n \leq c$
 - $T(n) = a T(n/b) + D(n) + C(n)$ w przeciwnym przypadku

Rekurencja dla algorytmu typu „dziel i zwyciężaj”

15

- **Przykład: algorytm sortowania przez scalanie**
 - **dziel:** znajdujemy środek przedziału, zajmuje to czas stały $D(n) = \theta(1)$,
 - **zwyciężaj:** rozwiązujemy rekurencyjnie dwa podproblemy, każdy rozmiaru $n/2$, co daje czas działania $2 T(n/2)$,
 - **połącz:** działa w czasie $\theta(n)$, a więc $C(n) = \theta(n)$.
- **Ostatecznie:**
 - $T(n) = \theta(1)$ jeśli $n=1$
 - $T(n) = 2 T(n/2) + \theta(1) + \theta(n)$ jeśli $n>1$
- **Rozwiązaniem tej rekurencji jest $T(n) = \theta(n \log n)$.**

Metody rozwiązywania rekurencji

16

□ Metoda podstawiania:

- zgadujemy oszacowanie, a następnie dowodzimy przez indukcję jego poprawność.

□ Metoda iteracyjna:

- przekształcamy rekurencję na sumę, korzystamy z technik ograniczania sum.

□ Metoda uniwersalna:

- stosujemy oszacowanie na rekurencję mające postać $T(n) = a T(n/b) + f(n)$, gdzie $a \geq 1$, $b > 1$, a $f(n)$ jest daną funkcją.

Metoda podstawiania

17

- Polega na **zgadnięciu postaci rozwiązania**, a następnie wykazaniu przez indukcję, że jest ono poprawne.
- Trzeba też znaleźć odpowiednie stałe.
- Bardzo skuteczna, stosowana tylko w przypadkach kiedy łatwo jest przewidzieć postać rozwiązania.

Metoda podstawiania

18

□ Przykład:

□ Postać rekurencji:

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

□ Zgadnięte rozwiązanie:

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

□ Podstawa:

$$n=2; T(1)=1; T(2)=4;$$

□ Indukcja:

$$T(n) \leq 2 (c(n/2)\log(n/2)) + n \leq c n \log(n/2) + n$$

$$T(n) \leq c n \log(n/2) + n = c n \log(n) - c n \log(2) + n$$

$$T(n) \leq c n \log(n) - c n \log(2) + n = c n \log(n) - c n + n$$

$$T(n) \leq c n \log(n) - c n + n \leq c n \log(n)$$

spełnione dla $c \geq 1$;

Metoda iteracyjna

19

- Polega na **rozwijaniu (iterowaniu) rekurencji i wyrażanie jej jako sumy składników zależnych tylko od n warunków brzegowych. Następnie mogą być użyte techniki sumowania do oszacowania rozwiązania.**

Metoda iteracyjna

20

□ Przykład:

□ Postać rekurencji:

$$T(n) = 3T(n/4) + n$$

□ Iterujemy:

$$T(n) = n + 3T(n/4) = n + 3((n/4) + 3T(n/16)) = n + 3(n/4) + 9T(n/16)$$

$$T(n) = n + 3(n/4) + 9T(n/16) = n + 3n/4 + 9n/16 + 27T(n/64)$$

□ Iterujemy tak długo aż osiągniemy warunki brzegowe.

Składnik i -ty w ciągu wynosi $3^i n/4^i$.

Iterowanie kończymy, gdy $n=1$ lub $n/4^i = 1$ (czyli $i > \log_4(n)$).

$$T(n) \leq n + 3n/4 + 9n/16 + 27n/64 + \dots + 3 \log_4 n \theta(1)$$

$$T(n) \leq 4n + 3 \log_4 n \theta(1) = \theta(n)$$

Metoda iteracyjna

21

- **Metoda iteracyjna jest zazwyczaj związana z dużą ilością przekształceń algebraicznych, więc zachowanie prostoty nie jest łatwe.**
- **Punkt kluczowy to skoncentrowanie się na dwóch parametrach:**
 - ▣ **liczbie iteracji koniecznych do osiągnięcia warunku brzegowego**
 - ▣ **oraz sumie składników pojawiających się w każdej iteracji.**

Drzewa rekursji

22

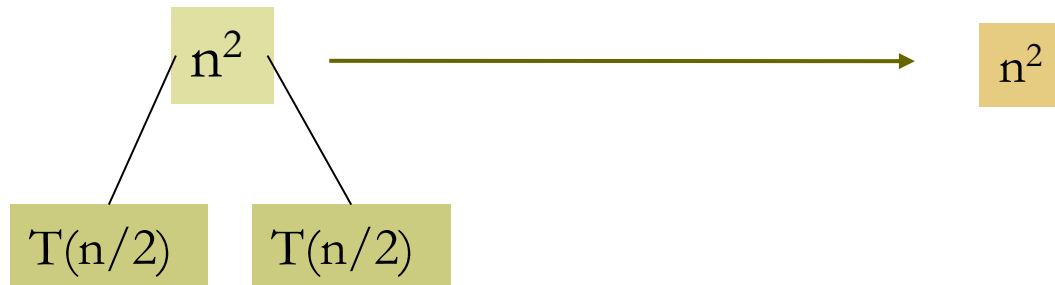
- **Pozwalają w dogodny sposób zilustrować rozwijanie rekurencji, jak również ułatwiają stosowanie aparatu algebraicznego służącego do rozwiązywania tej rekurencji.**
- **Szczególnie użyteczne gdy rekurencja opisuje algorytm typu “dziel i zwyciężaj”.**

Drzewa rekursji dla algorytmu „dziel i zwyciężaj”

„dziel

23

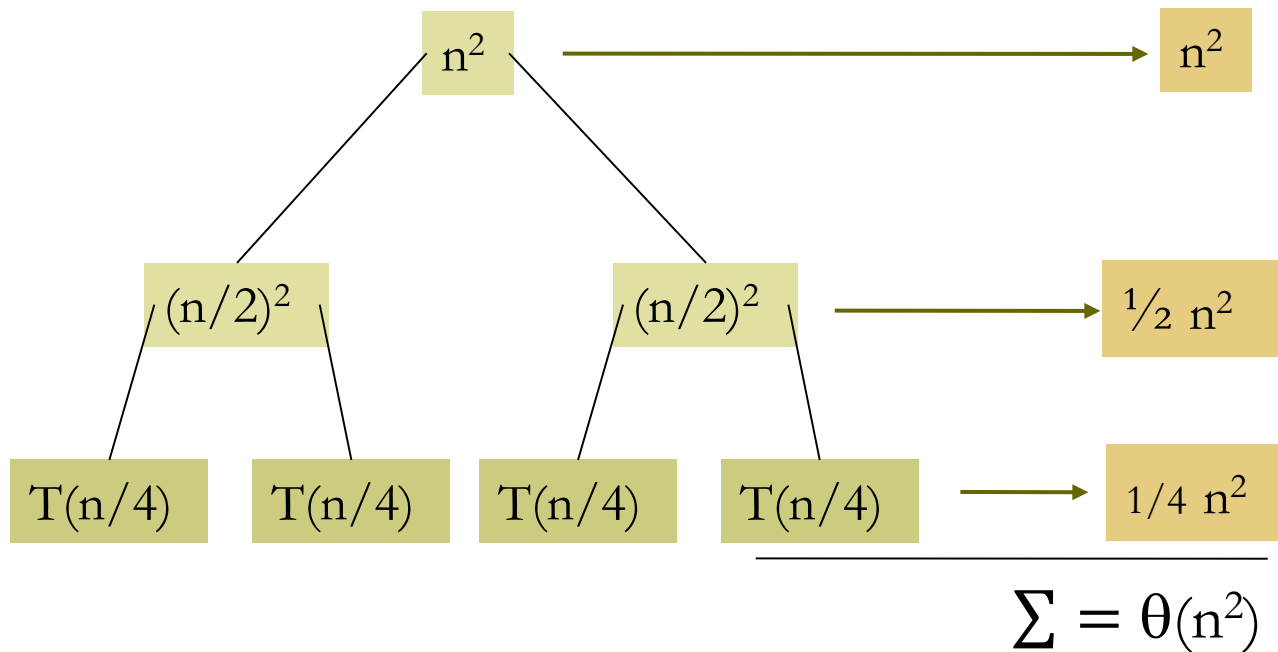
□ $T(n) = 2 T(n/2) + n^2$



Drzewa rekursji dla algorytmu „dziel i zwyciężaj”

24

□ $T(n) = 2 T(n/2) + n^2$

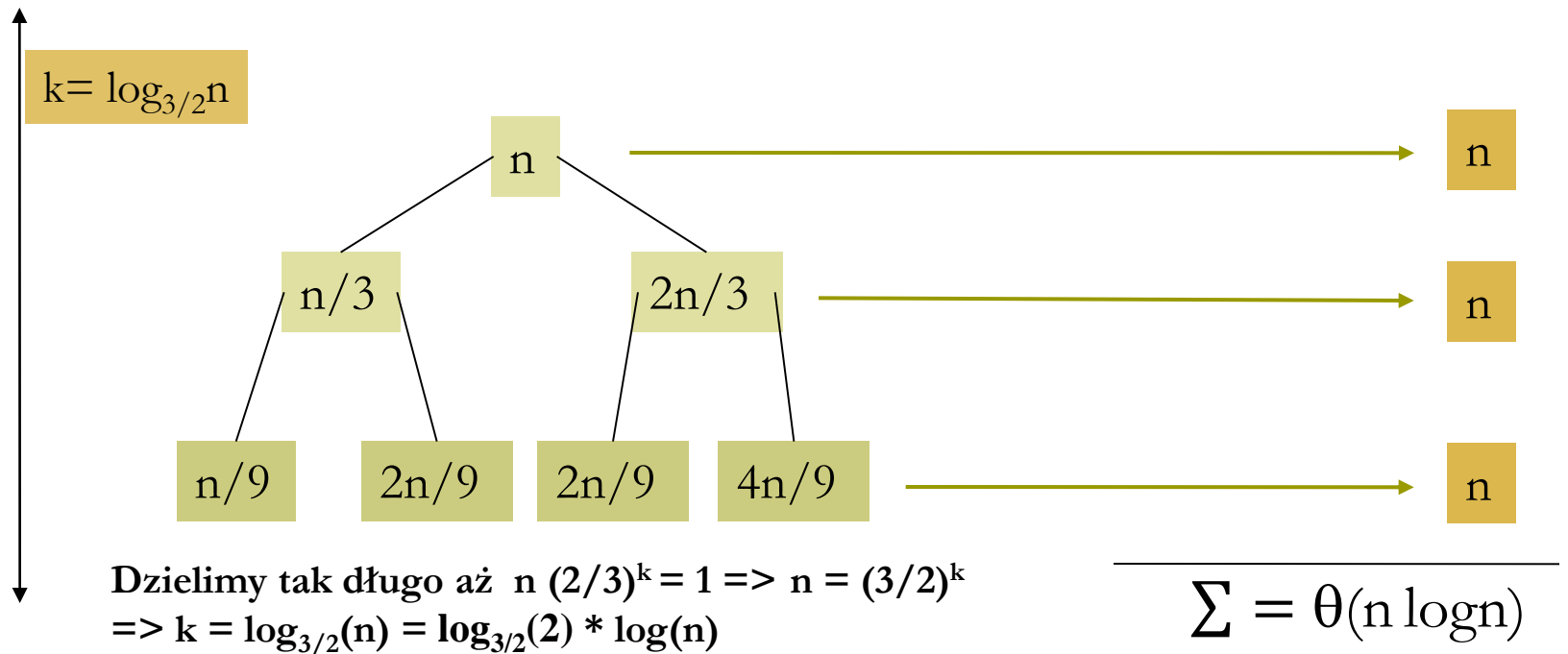


□ Rozwiązaniem tego równania jest $T(n) = \Theta(n^2)$

Drzewa rekursji

25

□ $T(n) = T(n/3) + T(2n/3) + n$



□ Rozwiązaniem tego równania jest $T(n) = \theta(n \log n)$

Funkcje tworzące

26

- **Rekurencje były badane już od 1202 roku przez L. Fibonacciego**
- **A. de Moivre w 1730 roku wprowadził pojęcie funkcji tworzących do rozwiązywania rekurencji.**

Metoda rekurencji uniwersalnej

27

- **Metoda rekurencji uniwersalnej podaje “uniwersalny przepis” rozwiązywania równania rekurencyjnego postaci:**
 - $T(n) = a T(n/b) + f(n)$
gdzie $a \geq 1$ i $b > 1$ są stałymi, a $f(n)$ jest funkcja asymptotycznie dodatnia.
- **Za wartość (n/b) przyjmujemy najbliższą liczbę całkowitą (mniejsza lub większą od wartości dokładnej).**

Metoda rekurencji universalnej

28

- Rekurencja opisuje czas działania algorytmu, który dzieli **problem rozmiaru n** na **a problemów, każdy rozmiaru n/b** , gdzie a i b są dodatnimi stałymi.
- Każdy z a problemów jest rozwiązywany rekurencyjnie w czasie $T(n/b)$.
- Koszt dzielenia problemu oraz łączenia rezultatów częściowych jest opisany funkcja $f(n)$.

Twierdzenie o rekurencji uniwersalnej

29

- Niech $a \geq 1$ i $b > 1$ będą stałymi, niech $f(n)$ będzie pewną funkcją i niech $T(n)$ będzie zdefiniowane dla nieujemnych liczb całkowitych przez rekurencje

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

gdzie (n/b) oznacza najbliższą liczbę całkowitą do wartości dokładnej n/b .

- Wtedy funkcja $T(n)$ może być ograniczona asymptotycznie w następujący sposób:
 - Jeśli $f(n) = O(n^{\log_b a - \epsilon})$ dla pewnej stałej $\epsilon > 0$, to $T(n) = \Theta(n^{\log_b a})$.
 - Jeśli $f(n) = \Theta(n^{\log_b a})$ to $T(n) = \Theta(n^{\log_b a} \log n)$.
 - Jeśli $f(n) = n^{\log_b a + \epsilon}$ dla pewnej stałej $\epsilon > 0$ i jeśli $af(n/b) \leq cf(n)$ dla pewnej stałej $c < 1$ i wszystkich dostatecznie dużych n , to $T(n) = \Theta(f(n))$.

Twierdzenie o rekurencji uniwersalnej

30

□ “Intuicyjnie...”:

- W każdym z trzech przypadków porównujemy funkcje $f(n)$ z funkcją $n^{\log_b a}$. Rozwiązanie rekurencji zależy od większej z dwóch funkcji.
- Jeśli funkcja $n^{\log_b a}$ jest większa, to rozwiązaniem rekurencji jest:
 - $T(n) = \Theta(n^{\log_b a})$.
- Jeśli funkcje są tego samego rzędu, to mnożymy przez $\log n$ i rozwiązaniem jest:
 - $T(n) = \Theta(n^{\log_b a} \log n) = T(n) = \Theta(f(n) \log n)$.
- Jeśli $f(n)$ jest większa, to rozwiązaniem jest:
 - $T(n) = \Theta(f(n))$.

Przykład

31

$$\square T(n) = 9 T(n/3) + n$$

$$a=9,$$

$$b=3,$$

$$f(n)=n,$$

a zatem $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.

- \square Ponieważ $f(n) = O(n^{\log_3 9 - \epsilon})$, gdzie $\epsilon = 1$, możemy zastosować przypadek 1 z twierdzenia i wnioskować że rozwiązaniem jest $T(n) = \Theta(n^2)$.

Przykład

32

$$\square T(n) = T(2n/3) + 1$$

$$a=1,$$

$$b=3/2,$$

$$f(n)=1,$$

$$\text{a zatem } n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1.$$

- \square Stosujemy przypadek 2, gdyż $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$,
a zatem rozwiązaniem rekurencji jest $T(n) = \Theta(\log n)$.

Przykład

33

□ $T(n) = 3T(n/4) + n \log n$

$$a=3,$$

$$b=4,$$

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

a zatem $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$.

- Ponieważ $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, gdzie $\epsilon \approx 0.2$, więc stosuje się tutaj przypadek 3, jeśli możemy pokazać że dla $f(n)$ zachodzi warunek regularności.

Dla dostatecznie dużych n :

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n\log(n) = c f(n)$$

dla $c=3/4$.

- Warunek jest spełniony i możemy napisać że rozwiązaniem rekurencji jest $T(n) = \Theta(n \log n)$.

Przykład

34

□ $T(n) = 2T(n/2) + n \log n$

$a=2,$

$b=2,$

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

a zatem $n^{\log_b a} = n.$

- Wydaje się że powinien to być przypadek 3, gdyż $f(n)=n \log n$ jest asymptotycznie większe niż $n^{\log_b a} = n$, ale nie wielomianowo większy.
- Stosunek $f(n)/ n^{\log_b a} = (n \log n)/n = \log n$ jest asymptotycznie mniejszy niż n^ϵ dla każdej dodatniej stałej ϵ .
- W konsekwencji rekurencja ta “wpada” w lukę między przypadkiem 2 i 3.

Złożoność zamortyzowana

35

- W wielu sytuacjach na strukturach danych działają nie pojedyncze operacje ale ich sekwencje.
- Jedna z operacji takiej sekwencji może wpływać na dane w sposób powodujący modyfikacje czasu wykonania innej operacji.
Jednym ze sposobów określania czasu wykonania w przypadku pesymistycznym dla całej sekwencji jest dodanie składników odpowiadających wykonywaniu poszczególnych operacji.
- Jednak wynik tak uzyskany może być zbyt duży w stosunku do rzeczywistego czasu wykonania. Analiza amortyzacji pozwala znaleźć bliższą rzeczywistej złożoność średnią.

Złożoność zamortyzowana

36

- Analiza z amortyzacją polega na analizowaniu kosztów operacji, zaś pojedyncze operacje są analizowane właśnie jako elementy tego ciągu. Koszt wykonania operacji w sekwencji może być różny niż w przypadku pojedynczej operacji, ale ważna jest też częstość wykonywania operacji.
- Jeśli dana jest sekwencja operacji op_1, op_2, op_3, \dots to analiza złożoności pesymistycznej daje złożoność obliczeniową równą:
$$C(op_1, op_2, op_3, \dots) = C_{\text{pes}}(op_1) + C_{\text{pes}}(op_2) + C_{\text{pes}}(op_3) + \dots$$
dla złożoności średniej uzyskujemy
$$C(op_1, op_2, op_3, \dots) = C_{\text{śre}}(op_1) + C_{\text{śre}}(op_2) + C_{\text{śre}}(op_3) + \dots$$
- Nie jest analizowana kolejność operacji, “sekwencja” to po prostu “zbiór” operacji.

Złożoność amortyzowana

37

- Przy analizie z amortyzacją zmienia się sposób patrzenia, gdyż sprawdza się co się stało w danym momencie sekwencji i dopiero potem wyznacza się złożoność następnej operacji:

$$C(\text{op}_1, \text{op}_2, \text{op}_3, \dots) = C(\text{op}_1) + C(\text{op}_2) + C(\text{op}_3) + \dots$$

gdzie **C** może być złożonością optymistyczną, średnią, pesymistyczną lub jeszcze inną - w zależności od tego co działo się wcześniej.

- Znajdowanie złożoności amortyzowanej tą metoda może być zanadto skomplikowane.

Znajomość natury poszczególnych procesów oraz możliwych zmian struktur danych używane są do określenia funkcji **C**, którą można zastosować do każdej operacji w sekwencji. Funkcja jest tak wybierana aby szybkie operacje były traktowane jak wolniejsze niż w rzeczywistości, zaś wolne jako szybsze.

- Sztuka robienia analizy amortyzacji polega na znalezieniu funkcji **C**; takiej która dociąży tanie operacje dostatecznie aby pokryć niedoszacowanie operacji kosztownych.

Przykład

38

□ Przykład:

- dodawanie elementu do wektora zaimplementowanego jako elastyczna tablica

□ Przypadek optymistyczny: wielkość wektora jest mniejsza od jego pojemności, dodanie elementu ogranicza się do wstawienia go do pierwszej wolnej komórki.

Koszt dodania nowego elementu to $O(1)$.

□ Przypadek pesymistyczny: rozmiar jest równy pojemności, nie ma miejsca na nowe elementy. Konieczne jest zaalokowanie nowego obszaru pamięci, skopiowanie do niego dotychczasowych elementów i dopiero dodanie nowego.

Koszt wynosi wówczas $O(\text{rozmiar (wektor)})$.

Pojawia się nowy parametr, bo pojemność można zwiększać o więcej niż jedną komórkę wtedy przepełnienie pojawia się tylko “od czasu do czasu”.

Przykład

39

- **Analiza z amortyzacją:** badane jest jaka jest oczekiwana wydajność szeregu kolejnych wstawień. Wiadomo, że we przypadku optymistycznym jest to $O(1)$, a w przypadku pesymistycznym $O(\text{rozmiar})$, ale przypadek pesymistyczny zdarza się rzadko.
- Należy przyjąć pewną hipotezę:
 - a) $\text{kosztAmort}(\text{push}(x)) = 1$
niczego nie zyskujemy, łatwe wstawienia nie wymagają poprawek, nie pojawia się jednak zapas na kosztowne wstawienia
 - b) $\text{kosztAmort}(\text{push}(x)) = 2$
zyskujemy zapas na łatwych wstawieniach, ale czy wystarczający...?
Zależy to od rozmiaru wektora...

Przykład dla kosztu amortyzacji = 2

40

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	2	0+1	1
2	2	2	1+1	1
3	4	2	2+1	0
4	4	2	1	1
5	8	2	4+1	-2
6	8	2	1	-1
7	8	2	1	0
8	8	2	1	1
9	16	2	8+1	-6
10	16	2	1	-5
...
16	16	2	1	1
17	32	2	16+1	-14
18	32	2	1	-13

Przykład dla kosztu amortyzacji = 3

41

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	3	0+1	2
2	2	3	1+1	3
3	4	3	2+1	3
4	4	3	1	5
5	8	3	4+1	3
6	8	3	1	5
7	8	3	1	7
...
16	16	3	1	17
17	32	3	16+1	3
18	32	3	1	5

c) $\text{kosztAmort}(\text{push}(x)) = 3$

- Nigdy nie pojawia się “debet”, zaoszczędzone jednostki są niemal w całości zużywane gdy pojawi się kosztowna operacja.

Złożoność amortyzowana

42

- W przedstawionym przykładzie wybór funkcji stałej był słuszny, ale zwykle tak nie jest.
- Niech funkcja przypisująca liczbę do konkretnego stanu struktury danych ds będzie nazywana **funkcją potencjału**. Koszt amortyzowany definiuje się następująco:

$$\text{koszAmort}(op_i) = \text{koszt}(op_i) + f.\text{potencjału}(ds_i) - f.\text{potencjału}(ds_{i-1})$$

- Jest to faktyczny koszt wykonania operacji op_i powiększony o zmianę potencjału struktury danych ds po wykonaniu tej operacji. Definicja ta obowiązuje dla pojedynczej operacji.

$$\text{koszAmort}(op_1, op_2, op_3, \dots, op_m) = \sum_{i=1}^m (\text{koszt}(op_i) + f.\text{potencjału}(ds_i) - f.\text{potencjału}(ds_{i-1}))$$

- W większości przypadków funkcja potencjału początkowo jest zerem, nigdzie nie jest ujemna, tak że czas amortyzowany stanowi kres górny czasu rzeczywistego.

Kontynuacja przykładu

43

□ f.potencjału (vector_i) =

= 0 (jeśli $\text{rozmiar}_i = \text{pojemność}_i$ czyli vector jest pełny)

= $2 \cdot \text{rozmiar}_i - \text{pojemność}_i$ (w każdym innym przypadku)

□ Można sprawdzić że przy tak zdefiniowanej

f.potencjału, kosztAmort(op_i) jest faktycznie równy **3** w każdej konfiguracji (tanie wstawianie, kosztowne, tanie po kosztownym)

Dane zewnętrzne

44

- Podstawowy czynnik rzutujący na różnice między obróbką danych wewnętrznych (czyli przechowywanych w pamięci operacyjnej) a obróbką danych zewnętrznych (czyli przechowywanych w pamięciach masowych) jest specyfika dostępu do informacji.
- Mechaniczna struktura dysków sprawia, że korzystnie jest odczytywać dane nie pojedynczymi bajtami, lecz w większych **blokach**. Zawartość pliku dyskowego można traktować jako listę łączy poszczególnych bloków, bądź też jako drzewo którego liście reprezentują właściwe dane, a węzły zawierają informacje pomocniczą ułatwiającą zarządzanie tymi danymi.

Dane zewnętrzne

45

- Załóżmy że:
 - adres bloku = 4 bajty
 - długość bloku = 4096 bajtów
- Czyli w jednym bloku można zapamiętać adresy do 1024 innych bloków.
- Czyli informacja pomocnicza do 4 194 304 bajtów będzie zajmować 1 blok.
- Możemy też budować strukturę wielopoziomową, w strukturze dwupoziomowej blok najwyższy zawiera adresy do 1024 bloków pośrednich, z których każdy zawiera adresy do 1024 bloków danych. Maksymalna wielkość pliku w tej strukturze $1024 * 1024 * 1024 = 4\ 294\ 967\ 296$ bajtów = 4GB, informacja pomocnicza zajmuje 1025 bloków.

Dane zewnętrzne

46

- Nieodłącznym elementem współpracy pamięci zewnętrznej z pamięcią operacyjną są **bufory**, czyli zarezerwowany fragment pamięci operacyjnej, w której system operacyjny umieszcza odczytany z dysku blok danych lub z którego pobiera blok danych do zapisania na dysku.
- **Miara kosztu dla operacji na danych zewnętrznych.**
 - Głównym składnikiem czasu jest czekanie na pojawienie się właściwego sektora pod głowicami. To może być nawet kilkanaście milisekund... Co jest ogromnie długo dla procesora taktowanego kilku-gigahercowym zegarem.
 - Zatem “merytoryczna jakość” algorytmu operującego na danych zewnętrznych będzie zależna od liczby wykonywanych **dostępów blokowych** (odczyt lub zapis pojedynczego bloku nazywamy dostępem blokowym).

Sortowanie

47

- ❑ **Sortowanie zewnętrzne** to sortowanie danych przechowywanych na plikach zewnętrznych.
- ❑ Sortowanie przez łączenie pozwoli na posortowanie pliku zawierającego n rekordów, przeglądając go jedynie $O(\log n)$ razy.
- ❑ Wykorzystanie pewnych mechanizmów systemu operacyjnego – dokonywanie odczytów i zapisów we właściwych momentach – może znacząco usprawnić sortowanie dzięki zrównoległowieniu obliczeń z transmisją danych.

Sortowanie przez łączenie

48

- Polega na organizowaniu sortowanego pliku w pewną liczbę serii, czyli uporządkowanych ciągów rekordów. W kolejnych przebiegach rozmiary serii wzrastają a ich liczba maleje, ostatecznie (posortowany) plik staje się pojedynczą serią.
- Podstawowym krokiem sortowania przez łączenie dwóch plików, f_1 i f_2 , jest zorganizowanie tych plików w serie o długości k , tak że:
 - liczby serii w plikach f_1, f_2 , z uwzględnieniem “ogonów” różnią się co najwyżej o jeden
 - co najwyżej w jednym z plików f_1, f_2 , może się znajdować ogon
 - plik zawierający “ogon” ma poza nim co najmniej tyle serii ile jego partner.
- Prosty proces polega na odczytywaniu po jednej serii (o długości k) z plików f_1, f_2 , łączenia tych serii w dwukrotnie dłuższą i zapisywania tak połączonych serii na przemian do plików g_1, g_2 .
- Całkowita liczba dostępow blokowych w całym procesie sortowania jest rzędu $O((n \log n)/b)$ gdzie b jest liczba rekordów w jednym bloku.

Przykład

49

Pliki oryginalne:

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

Pliki zorganizowane w serie
o długości 2:

28 31	93 96	54 85	30 39	8 10	8 10
3 5	10 40	9 65	13 90	69 77	22

Pliki zorganizowane w serie
o długości 4

3 5 28 31	9 54 65 85	8 10 69 77
10 40 93 96	13 30 39 90	8 10 22

Pliki zorganizowane w serie
o długości 8:

3 5 10 28 31 40 93 96	8 8 10 10 22 69 77
9 13 30 39 54 65 85 90	

Pliki zorganizowane w serie
o długości 16:

3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96
8 8 10 10 22 69 77

Pliki zorganizowane w serie
o długości 32:

3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

Podsumowanie

50

- „*Nie przejmuj się efektywnością algorytmu... wystarczy poczekać kilka lat.*”
- Taki pogląd funkcjonuje w środowisku programistów, nie określono przecież granicy rozwoju mocy obliczeniowych komputerów. Nie należy się jednak z nim zgadzać w ogólności. Należy zdecydowanie przeciwstawiać się przekonaniu o tym, że ulepszenia sprzętowe uczynią prace nad efektywnymi algorytmami zbyt ciężką.
- Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych jest teoretycznie możliwe, ale praktycznie przekracza możliwości istniejących technologii. Przykładem takiego problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy “inteligentna” komunikacja.
- Pomiedzy komputerami a ludźmi na rozmaitych poziomach.
- Kiedy pewne problemy stają się “proste”... Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrazić, wytyczy nowe granice możliwości wykorzystania komputerów.