Algorytmy i Struktury Danych

Rekurencja

(c) Marcin Sydow

Zawartość wykładu:

- Wprowadzenie: czym jest rekurencja
- ciąg Fibonacciego
- Przykład: liniowe równania rekurencyjne 2. rzędu
- Przykład: zagadka “wieże Hanoi”
- 3 często spotykane rekurencyjne schematy równań
- Twierdzenie o rekurencji uniwersalnej

Rekurencja

Rekurencja ma kilka aspektów, m.in. matematyczny, algorytmiczny i programistyczny

- matematyczny: rekurencyjne definicje (takie, pojęcie definiowane jest przez odwołanie do samego siebie). Niezbędną częścią definicji rekurencyjnej jest tzw. **przypadek bazowy**.
- Algorytmy: rekurencja ma tu odzwierciedlenie w potężnej technice projektowania algorytmów "dziel i zwyciężaj", gdzie rozwiązanie problemu definiuje się w oparciu o rozwiązanie (mniejszych) podproblemów tego samego typu
- Programowanie: rekurencja (zwana też rekursją) jest naturalną techniką implementacji algorytmów rekurencyjnych, czyli funkcji, które wywołują same siebie (dla mniejszych argumentów).

Przykład: matematyczna definicja pojęcia silni

Dla liczby naturalnej $n \in N$ silnia, oznaczana $n!$ zdefiniowana jest jako iloczyn wszystkich kolejnych dodatnich liczb naturalnych niewiększych od n: $n! = n * (n-1) * (n-2) * ... * 1$
(oraz przyjmuje się, że $0! = 1$)

Definicję tę można też podać w **formie rekurencyjnej**:
$n! = (n-1)! \cdot n$
przypadek bazowy:
$0! = 1$

Zauważmy, że bez podania przypadku bazowego, powyższa definicja byłaby wadliwa, rozwijając się w nieskończoność, np.: $2! = 1! \cdot 2 =$ $0! \cdot 1 \cdot 2 =$(-1)!0 $\cdot 1 \cdot 2$, etc...

Przykład 2: ciąg liczb Fibonacciego

Przypadek bazowy:

Fibonacci(0) = 0
Fibonacci(1) = 1

definicja rekurencyjna:

Fibonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)

Pierwsze 10 wartości:
0,1,1,2,3,5,8,13,21,34,...

Rekurencja jako narzędzie projektowania algorytmów

Jest to często używane podejście do projektowania algorytmów, przykłady: MergeSort, QuickSort i wiele innych.

Niektóre algorytmy trudno byłoby inaczej zaprojektować niż rekurencyjnie (np. algorytm rozwiązujący problem "wież Hanoi" przedstawiony na kolejnych slajdach).

Matematyczne definicje rekurencyjne stanowią w zasadzie gotową receptę algorytmu rekurencyjnego.

Stosując rekurencję należy jednak brać pod uwagę wynikający z niej koszt pamięciowy (masowe wywoływanie funkcji). Z tego powodu rekurencji powinno się unikać o ile to możliwe i jeśli nie komplikuje to bardzo algorytmu.

Przykład: liczby Fibonacciego

Korzystając z rekurencyjnej definicji, napiszmy rekurencyjny algorytm obliczający n-tą liczbę Fibonacciego:

```
fibonacci(n){
   if (n < 2) return n;
   else return (fibonacci(n-1) + fibonacci(n-2));
}
```

I gotowe!
Jest jednak problem: jaki?
Co się stanie, gdy zaimplementujemy i wywołamy np. fibonacci(50)?

Przykład: liczby Fibonacciego

Korzystając z rekurencyjnej definicji, napiszmy rekurencyjny algorytm obliczający n-tą liczbę Fibonacciego:

```
fibonacci(n){
   if (n < 2) return n;
   else return (fibonacci(n-1) + fibonacci(n-2));
}
```

I gotowe!

Jest jednak problem: jaki?

Co się stanie, gdy zaimplementujemy i wywołamy np. fibonacci(50)?

Dokładniejsza analiza zachowania tego algorytmu pokazuje, że liczba wywołań rekurencyjnych jest tu **wykładniczą** funkcją liczby n. Algorytm ten będzie więc działał zaskakująco wolno (i może nawet zabraknąć pamięci na wywołania funkcji dla większych wartości n).

Aby zaprojektować bardziej efektywny algorytm, potrzeba **nierekurencyjnego** wzoru na n-tą liczbę Fibonacciego.

Przykład: rekurencyjne równania liniowe 2-rzędu

Istnieje wiele twierdzeń pozwalających rozwiązywać równania rekurencyjne.

Uwaga: Przez rozwiązanie równania rekurencyjnego rozumiemy podanie **nierekurencyjnego** wzoru na n-ty wyraz danego ciągu.

Rekurencyjne równanie na n-ty wyraz ciągu Fibonacciego ma postać *funkcji liniowej* i odwołuje się wstecz do *dwóch poprzednich* wyrazów ciągu:

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Równanie to jest szczególnym przypadkiemy *liniowego równanania rekurencyjnego 2-giego rzędu*.

Twierdzenie o rekurencyjnych liniowych równaniach 2-go rzędu

Jeśli równanie rekurencyjne ma poniższą ogólną postać:

$$s_n = as_{n-1} + bs_{n-2}$$

to rozwiązując pewne pomocnicze tzw **równanie charakterystyczne**: $x^2 - ax - b = 0$,
w zależności od liczby jego pierwiastków otrzymujemy następujące rozwiązanie nierekurencyjne:

1. istnieje 1 pierwiastek r: $s_n = c_1 r^n + c_2 n r^n$
2. istnieją 2 pierwiastki r_1, r_2: $s_n = c_1 r_1^n + c_2 r_2^n$

gdzie stałe c_1, c_2 można znaleść podstawiając wartości bazowe (dla $n = 0$ i $n = 1$)
Nie podamy dowodu tego twierdzenia.
Oczywiście istnieją też inne twierdzenia dla innych typów równań rekurencyjnych.

Przykład zastosowania twierdzenia

Ponieważ równanie Fibonacciego ma właśnie postać rekurencyjnego równania liniowego 2-go rzędu, więc można użyc twierdzenia do jego rozwiązania:

Stosując twierdzenie, zauważamy, że współczynniki wynoszą $a = 1, b = 1$:
Fibonacci(n) = 1· Fibonacci(n-1) + 1· Fibonacci(n-2)

Rozwiązując równanie charakterystyczne otrzymujemy:
$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$

Jest to bezpośredni (nierekurencyjny) wzór na n-ty wyraz ciągu Fibonacciego!
(tzw. “wzór Bineta”)
Korzystając z niego możemy już obliczyć:
Fibonacci(50) =

Przykład zastosowania twierdzenia

Ponieważ równanie Fibonacciego ma właśnie postać rekurencyjnego równania liniowego 2-go rzędu, więc można użyc twierdzenia do jego rozwiązania:

Stosując twierdzenie, zauważamy, że współczynniki wynoszą $a = 1, b = 1$:
Fibonacci(n) = 1· Fibonacci(n-1) + 1· Fibonacci(n-2)

Rozwiązując równanie charakterystyczne otrzymujemy:
$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$

Jest to bezpośredni (nierekurencyjny) wzór na n-ty wyraz ciągu Fibonacciego!
(tzw. “wzór Bineta”)
Korzystając z niego możemy już obliczyć:
Fibonacci(50) = 12 586 269 025
(ciąg Fibonacciego rośnie wykładniczo szybko!)

Przykład: zagadka "wieże Hanoi"

Jak widzieliśmy na przykładzie liczb Fibonacciego, algorytm rekurencyjny często nie jest efektywnym rozwiązaniem równania rekurencyjnego (tu lepsze było rozwiązanie równania rekurencyjnego do wzoru bezpośredniego)

Dla wielu jednak problemów algorytm rekurencyjny jest przynajmniej jedynym naturalnym sposobem ujęcia rozwiązania, i ciężko rozpocząć rozwiązywanie problemu bez sformułowania rekurencyjnego.

Przykładem jest np. zagadka "wieże Hanoi", dla której jedyne łatwe rozwiązanie daje się sformułować w formie równania rekurencyjnego.

Wieże Hanoi

Mamy n krążków, każdy o innym rozmiarze, nawleczonych jeden na drugim na pionowym **początkowym drążku** A, od największego (na dole) do najmniejszego (na górze).

Celem jest przeniesienie wszystkich krążków na **docelowy drążek** C po jednym w każdym ruchu, ale tak, że w każdym ruchu możemy wziąć tylko 1 krążek z wierzchu dowolnego drążka i nigdy nie można kłaść większego krążka nad mniejszym. W całej operacji mamy też do dyspozycji **pomocniczy drążek** B (po każdym ruchu na każdym drążku może być dowolnie wiele krążków, ale znowu, nigdy większe nad mniejszymi)

1) Ile minimalnie ruchów należy wykonać, aby przenieść wszystkie n krążków z A na C? (oznaczmy minimalną liczbę ruchów przez hanoi(n))
2) Jakie dokładnie ruchy należy wykonać?

Wieże Hanoi, c.d.

Ile wynosi:
hanoi(0) =

Wieże Hanoi, c.d.

Ile wynosi:
$hanoi(0) = 0$
$hanoi(1) =$

Wieże Hanoi, c.d.

Ile wynosi:

$hanoi(0) = 0$

$hanoi(1) = 1$

$hanoi(2) =$

Wieże Hanoi, c.d.

Ile wynosi:
$hanoi(0) = 0$
$hanoi(1) = 1$
$hanoi(2) = 3$ (tu musimy już skorzystać z pomocniczego drążka)
$hanoi(10) = ?$

Celem jest znalezienie ogólnego wzoru dla dowolnego n.

Wieże Hanoi: liczba ruchów

Jak zdefiniować rozwiązanie dla n krążków w oparciu o rozwiązanie dla n-1 krążków (czyli rekurencyjnie, wg zasady "dziel i zwyciężaj")?

Wieże Hanoi: liczba ruchów

Jak zdefiniować rozwiązanie dla n krążków w oparciu o rozwiązanie dla n-1 krążków (czyli rekurencyjnie, wg zasady "dziel i zwyciężaj")?

1. Jeśli mamy 1 krążek (n=1), to przenosimy go jednym ruchem z drążka A na drążek C.
2. jeśli na drążku startowym jest n>1 krążków to zauważmy, że jeśli umiemy przenieść (za pomocą pewnej liczby ruchów) wierzchnie n-1 krążków na drążek pomocniczy, to możemy potem przenieść najniższy (odsłonięty) krążek ze startowego na docelowy i na końcu znowu n-1 krążków z pomocniczego na docelowy.

Rozwiązanie rekurencyjne dla hanoi(n)

Daje to w prosty sposów rekurencyjne równanie na liczbę ruchów:

warunek bazowy:

$$\text{hanoi}(1) = 1$$

równanie rekurencyjne:

$$\text{hanoi}(n) = \text{hanoi}(n\text{-}1) + 1 + \text{hanoi}(n\text{-}1) = 2*\text{hanoi}(n\text{-}1) + 1$$

Rozwiązanie równania rekurencyjnego dla wież Hanoi

Zastosujemy prostą metodę tzw. "rozwijania sumy":

$$hanoi(n) = 2 * hanoi(n-1) + 1 =$$
$$2 * (2 * hanoi(n-2) + 1) + 1 = ... = \sum_{i}^{n-1} 2^i = 2^n - 1$$

Mamy więc gotowy nierekurencyjny wzór na liczbę ruchów!

$$\text{hanoi}(10) = 2^{10} - 1 = 1023$$
(funkcja ta rośnie jeszcze szybciej niż liczby Fibonacciego!)

Uwaga: W tym zadaniu wyznaczenie wzoru na liczbę ruchów możliwe było **dzięki rekurencyjnemu ujęciu problemu**.

(pozostawiamy dla zainteresowanych jako nieobowiązkowe, ale ciekawe ćwiczenie, napisanie programu wypisującego listę ruchów, które należy wykonać dla n krążków!)

Zalety i wady stosowania rekurencji

Podsumujmy:

- rekurencja pozwala łatwo napisać algorytm korzystając z rekurencyjnego równania lub rekurencyjnej definicji jakiegoś pojęcia
- rekurencja jest czasami bardzo naturalnym środkiem do rozwiązania pozornie skomplikowanych problemów
- algorytm rekurencyjny może być nieefektywny (zarówno czasowo jak i pamięciowo - stos wywołań rekurencyjnych funkcji)
- jeśli to możliwe, należy zamienić definicję rekurencyjną na nierekurencyjną i w ten sposób na końcu uniknąć rekurencyjnego algorytmu, o ile nie komplikuje to bardzo algorytmu

Często spotykane przypadki rekurencyjne

Przedstawimy teraz pewne 3 często spotykane przypadki równań rekurencyjnych (z rozwiązaniami), które powstają w rezultacie analizy złożoności czasowej algorytmów.

Ich znajomość jest przydatna do obliczenia rzędu złożoności czasowej często spotykanych schematów algorytmów.

W poniższych przykładach wyobraźmy sobie, że t(n) oznacza złożość czasową pewnego algorytmu dla danych o rozmiarze n.

W dowodach 3 poniższych przypadków zakładamy (dla prostoty dowodu), że n jest dokładną potęga 2, ale uzyskane wyniki można uogólnić na dowolne wartości n dla typowo spotykanych funkcji złożoności czasowej t().

Przypadek 1

t(1) = 0
(interpretacja: dla danych 1-elementowych nie ma specjalnie nic do zrobienia)
t(n) = t(n/2) + c; n>1, $c \in N$ jest stałą dodatnią
(interpretacja: dla danych o rozmiarze większych niż 1, można rozwiązac problem przez rozwiązanie podproblemu połowę (n/2) mniejszego przy stałym (c) dodatkowym narzucie pracy)
($n/2$ oznacza $\lfloor (n/2) \rfloor$ lub $\lceil (n/2) \rceil$)
Rozwiązanie:

podstawmy dla wygody rachunków $n = 2^k$ i zastosujmy rozwijanie do sumy:

$$t(2^k) = t(2^{k-1}) + c = t(2^{k-2}) + c + c = t(2^0) + kc = kc = clog(n)$$

rozwiązanie: $t(n) = c(log(n)) = \Theta(log(n))$, czyli logarytmiczna złożoność czasowa

Przykład algorytmu spełniającego ten schemat?
rekurencyjna wersja binSearch

Przypadek 2

$t(1) = 0$
$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + c$; $n>0$, $c \in N$ jest stałą dodatnią.
(interpretacja: rozwiązanie problemu można otrzymać poprzez rozwiązanie dwóch podproblemów połowę mniejszych przy stałym narzucie dodatkowej pracy)

Rozwiązanie (znowu podstawmy dla wygody obliczeń $n = 2^k$):

$$t(2^k) = 2t(2^{k-1}) + c = 2(2t(2^{k-2}) + c) + c = 2^2(t(2^{k-2})) + 2^1 c + 2^0 c = 2^k t(2^0) + c(2^{k-1} + 2^{k-2} + ... + 2^0) = 0 + c(2^k - 1) = c(n-1)$$

rozwiązanie: $t(n) = c(n-1) = \Theta(n)$
(Liniowa złożoność czasowa)

Przykład algorytmu spełniającego ten schemat?

Przypadek 2

$t(1) = 0$
$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + c$; $n>0$, $c \in N$ jest stałą dodatnią.
(interpretacja: rozwiązanie problemu można otrzymać poprzez rozwiązanie dwóch podproblemów połowę mniejszych przy stałym narzucie dodatkowej pracy)

Rozwiązanie (znowu podstawmy dla wygody obliczeń $n = 2^k$):

$$t(2^k) = 2t(2^{k-1}) + c = 2(2t(2^{k-2}) + c) + c = 2^2(t(2^{k-2})) + 2^1 c + 2^0 c = 2^k t(2^0) + c(2^{k-1} + 2^{k-2} + ... + 2^0) = 0 + c(2^k - 1) = c(n-1)$$

rozwiązanie: $t(n) = c(n-1) = \Theta(n)$
(Liniowa złożoność czasowa)

Przykład algorytmu spełniającego ten schemat?

np. rekurencyjne wyszukiwanie maksimum w ciągu n elementów (rekurencyjnie wyszukaj w lewej i prawej połowie ciągu i zwróć większy z wyników)

Przypadek 3

$t(1) = 0$

$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + cn$; $n>0$, $c \in N$ jest stałą

interpretacja: Rozwiązać problem można przez rozwiązanie dwóch podproblemów połowę mniejszych i przy dodatkowym *liniowym* (n) narzucie pracy.

Rozwiązanie (podstawmy $n = 2^k$):

$t(2^k) = 2t(2^{k-1}) + c2^k = 2(2t(2^{k-2}) + c2^{k-1}) + c2^k =$
$2^2 t(2^{k-2}) + c2^k + c2^k = 2^k t(2^0) + kc2^k = 0 + cnlog(n)$

rozwiązanie: $cn(log(n)) = \Theta(nlog(n))$
(Złożoność liniowo-logarytmiczna)

przykład algorytmu spełniającego ten schemat?:

Przypadek 3

$t(1) = 0$

$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + cn$; $n>0$, $c \in N$ jest stałą

interpretacja: Rozwiązać problem można przez rozwiązanie dwóch podproblemów połowę mniejszych i przy dodatkowym *liniowym* (n) narzucie pracy.

Rozwiązanie (podstawmy $n = 2^k$):

$t(2^k) = 2t(2^{k-1}) + c2^k = 2(2t(2^{k-2}) + c2^{k-1}) + c2^k =$
$2^2 t(2^{k-2}) + c2^k + c2^k = 2^k t(2^0) + kc2^k = 0 + cnlog(n)$

rozwiązanie: $cn(log(n)) = \Theta(nlog(n))$
(Złożoność liniowo-logarytmiczna)

przykład algorytmu spełniającego ten schemat?:
mergeSort

Twierdzenie o rekurencji uniwersalnej

Podamy teraz potężne narzędzie opisujące uniwersalną metodę rozwiązywania rekurencyjnych równań, które wyrażają np. funkcje złożoności czasowej T(n) wielu typów rekurencyjnych algorytmów typu "dziel i zwyciężaj", w których rekurencyjnie rozwiązuje się a podproblemów każdy o rozmiarze b razy mniejszym od oryginalnego przy dodatkowym narzucie pracy wyrażonym przez ogólnie daną funkcję $f(n)$.

Dokładniej, twierdzenie mówi jakie jest rozwiązanie równania rekurencyjnego postaci:

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

gdzie $a \geq 1, b > 1$ pewne stałe, $f(n)$ jest asymptotycznie dodatnia

Np. funkcja złożoności mergeSort jest jego szczególnym przypadkiem dla $a = 2, b = 2, f(n) = \Theta(n)$

Twierdzenie o rekurencji uniwersalnej

Załóżmy, że funkcja $T(n) : N \rightarrow R$ jest rekurencyjnie zdefiniowana następująco:

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

gdzie $a \geq 1, b > 1$: stałe, n/b oznacza $\lfloor (n/b) \rfloor$ lub $\lceil (n/b) \rceil$ oraz $f(n) : R \rightarrow R$ jest asymptotycznie dodatnia

Wtedy $T(n)$ może być asymptotycznie wyrażona w zależności od 3 przypadków jako:

1. jeśli $f(n) = O(n^{log_b a - \epsilon})$ dla pewnego $\epsilon > 0$, **wtedy:** $T(n) = \Theta(n^{log_b a})$
2. jeśli $f(n) = \Theta(n^{log_b a})$, **wtedy:** $T(n) = \Theta(n^{log_b a} log(n))$
3. jeśli $f(n) = \Omega(n^{log_b a + \epsilon}$ dla pewnego $\epsilon > 0$, [1], **wtedy:** $T(n) = \Theta(f(n))$

(Dowód: Cormen et al. "Wprowadzenie do algorytmów" rozdział 4.4)

[1] i jeśli dodatkowo założymy, że $af(n/b) \leq cf(n)$ dla pewnego $c < 1$ (tzw. "warunek regularności")

Interpretacja twierdzenia

Twierdzenie porównuje rząd funkcji narzutu $f(n)$ z rzędem pomocniczego wyrażenia $n^{log_b a}$ i mówi, że wyższy z tych rzędów determinuje rząd złożoności funkcji T(n):

1. jeśli rząd $f(n)$ jest (nieco upraszczając) niższego rzędu niż $n^{log_b a}$, to to ostatnie wyrażenie dominuje funkcję T(n)
2. jeśli $f(n)$ i $n^{log_b a}$ są tego samego rzędu, to pojawia się dodatkowy czynnik narzutu rzędu $\Theta(log(n))$
3. jeśli $f(n)$ jest (upraszczając) wyższego rzędu niż $n^{log_b a}$ (i spełnia pewien techniczny warunek), to $f(n)$ dominuje (i wyznacza) rząd całej funkcji T(n)

Uwaga: istnieją bardzo specyficzne funkcje $f(n)$, które nie spełniają powyższych warunków i dla nich twierdzenie nie działa, ale są to bardzo szczególne funkcje raczej niespotykane w analizie podstawowych algorytmów.

Zadania/Problemy:

- wymień pozytywne i negatywne aspekty zastosowania rekurencji w projektowaniu algorytmów
- podaj rekurencyjną definicję liczb Fibonacciego i kilka pierwszych wyrazów
- podaj sekwencję ruchów dla zagadki Hanoi Towers dla n=3 i n=4
- zastosuj twierdzenie o rozwiązywaniu równań rekurencyjnych 2-giego rzędu do równania Fibonacciego
- Wymień 3 podane przypadki schematów równań rekurencyjnych wraz z przykładami algorytmów, które do nich pasują
- napisz rekurencyjną wersję algorytmu binSearch, dokonaj analizy złożoności czasowej i pamięciowej i porównaj z wersją nierekurencyjną. Która jest lepsza? Dlaczego?
- podobnie jak wyżej, tylko dla algorytmu znajdowania minimum w tablicy n liczb
- (*) napisz funkcję wypisującą ciąg ruchów niezbędnych dla rozwiązania problemu wież Hanoi dla n krążków
- zilustruj twierdzenie o rekurencji uniwersalnej do analizy algorytmu mergeSort i do analizy rekurencyjnej wersji algorytmu binSearch

Dziękuję za uwagę.