Algorytmy

TABLICE INFORMATYCZNE • Piotr Wróblewski

Helion

WPROWADZENIE

Tablice pomogą Ci szybko przypomnieć sobie podstawowe zagadnienia dotyczące algorytmów i ich zastosowania. Opracowania tego możesz użyć jako przydatnej ściągi na wykładach lub laboratoriach. Tam, gdzie istniała taka możliwość, pokazałem także reprezentatywne fragmenty kodu w C++, a jeśli nie było to możliwe z uwagi na rozmiar listingu, posłużyłem się pseudokodem. Tablice napisane są w oparciu o książkę ***Algorytmy, struktury danych i techniki programowania***.

PODSTAWOWE POJĘCIA

Algorytm

- Skończony ciąg reguł, który stosuje się na skończonej liczbie danych, aby rozwiązywać zbliżone do siebie klasy problemów.
- Zespół reguł charakterystycznych dla pewnych obliczeń lub czynności informatycznych.

Pochodzenie

Termin *algorytm* pochodzi od nazwiska perskiego matematyka Muhammada ibn Musy al-Chuwarizmiego, który żył w IX wieku n.e. Jego zasługą jest dostarczenie klarownych reguł wyjaśniających krok po kroku zasady operacji arytmetycznych wykonywanych na liczbach dziesiętnych.

Algorytmy deterministyczne i niedeterministyczne

- Algorytm jest *deterministyczny*, gdy wynik działania jest jednoznacznie określony przez warunki początkowe (parametry) niezależnie od liczby jego wykonań.
- Algorytm *niedeterministyczny* można uzyskać, wprowadzając czynnik losowości (np. generator liczb losowych), przetwarzanie równoległe lub tzw. logikę kwantową (stany pośrednie między 0 i 1).

Budowa algorytmu informatycznego

- *Dane wejściowe* (w ilości większej lub równej 0) pochodzą z dobrze zdefiniowanego zbioru, który tworzą liczby całkowite, napisy lub ciągi znaków, złożone struktury wejściowe (np. grafy, zbiory).
- *Wynik* produkowany przez algorytm (rezultat wykonania) nie zawsze jest numeryczny (np. napisy, zbiory, listy).
- *Warunki wejściowe* mają na celu m.in. wyeliminowanie danych, które nie zawierają się w domenie obsługiwanej przez algorytm (np. pewien algorytm może akceptować wyłącznie dodatnie liczby całkowite).
- *Struktury danych* umożliwiają przechowywanie i obsługę danych przetwarzanych przez algorytm (np. tablice, listy, drzewa).

Cechy algorytmu

- *Skończony* — wynik algorytmu musi zostać kiedyś dostarczony (dla algorytmu A i danych wejściowych D powinno być możliwe precyzyjne określenie czasu wykonania $T(A, D)$).
- *Precyzyjny* — każdy krok algorytmu musi być jednoznacznie zdefiniowany.
- *Uniwersalny* — dany algorytm można zastosować do rozwiązywania całej klasy zadań, a nie tylko jednego konkretnego zadania (np. algorytm sortowania powinien dobrze działać zarówno na tablicy liczb całkowitych, jak i na tablicy obiektów złożonych).
- *Efektywny* — algorytm powinien wykonywać swoje zadanie w jak najkrótszym czasie i wykorzystywać do tego celu jak najmniejszą ilość pamięci (komputery mają ograniczoną pamięć i moc obliczeniową).

Schemat algorytmu

Algorytm Euklidesa (NWD)

Pojęcie algorytmu bywa ilustrowane przepisem greckiego matematyka Euklidesa (365 – 300 rok p.n.e.) na obliczanie największego wspólnego dzielnika (NWD) dwóch liczb `:a` i `b`. Algorytm NWD można opisać na wiele sposobów prowadzących do tego samego wyniku końcowego.

```
NWD (dane wejściowe: a i b;
   zmienna pomocnicza: r) {
jeśli b równa się zero, to
  wynikiem jest a;
w przeciwnym wypadku
   podstaw za r resztę
   z dzielenia a przez b;
rezultat: NWD(q, r);
}
```

```
#include <iostream>
using namespace std;
int nwd (int a, int b){
  if (b==0)
   return a;
  else
   return nwd (b, a % b); // Funkcja modulo
                          // w C++
}
```

Wynik działania algorytmu będzie taki sam zarówno w przypadku zastosowania pseudonotacji, jak i języka programowania.

ZŁOŻONOŚĆ OBLICZENIOWA

- *Który z dwóch programów wykonujących to samo zadanie (ale odmiennymi metodami) jest efektywniejszy?* Efektywność analizuje się zazwyczaj pod kątem czasu wykonania lub zajętość pamięci.
- Informacja typu *„program jest szybki, bo wykonał się w 1 minutę" nie daje* żadnej miarodajnej informacji!
- Miara *złożoności obliczeniowej* musi być reprezentatywna, aby użytkownicy na przykład małego komputera osobistego i potężnej stacji roboczej — obaj korzystający z tego samego algorytmu — mogli się ze sobą porozumieć co do sprawności obliczeniowej bez wyszczególniania, jakich używają komputerów, wersji kompilatora, rodzajów i architektury procesora.
- Parametrem najczęściej decydującym o czasie wykonania określonego algorytmu jest rozmiar danych n, z którymi ma on do czynienia. Rozmiar danych ma wiele znaczeń: dla funkcji sortującej tablicę będzie to jej rozmiar, dla programu obliczającego wartość funkcji silnia — wielkość danej wejściowej.

Notacja dużego O

Funkcja oznaczana jako T ukazuje rezultat dokładnych obliczeń działania algorytmu i jest nazywana *złożonością praktyczną*.

Gdy szukamy funkcji O, interesuje nas typ funkcji matematycznej występującej w T, który odgrywa w niej najważniejszą rolę, wpływając najsilniej na czas wykonywania programu. Przykłady poniżej.

T(n)	O(n)
6	O(1)
3n+1	O(n)
n^2-n+1	O(n^2)
2^n+n^2+4	O(2^n)

$O(1)$ Liczba operacji wykonywanych przez algorytm jest niezależna od rozmiarów problemu (co nie oznacza, że będzie mała!).

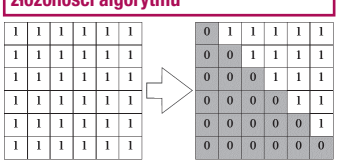
$O(n)$ Algorytm wykonuje się w czasie proporcjonalnym do rozmiaru problemu, np. przetwarzanie sekwencyjne ciągu znaków, obsługa kolejki itp. Jest to zależność liniowa, gdzie każdej z n danych wejściowych algorytm musi poświęcić pewien czas na wykonanie swoich obliczeń.

$O(\log n)$ Lepsza od liniowej. Logarytm liczby $x > 0$ o podstawie $b \neq 1$, oznaczony jako $u = \log_b x$, jest to liczba u spełniająca zależność $b^u = x$, np. $3 = \log_2 8$. Jeśli klasa problemu rośnie geometrycznie (np. o rząd wielkości ze 100 na 1000), to wzrost złożoności będzie arytmetyczny (tutaj dwukrotny). Jeśli n rośnie niezbyt szybko, to algorytm zwalnia, ale nie drastycznie. Ze złożonością logarytmiczną spotkamy się na przykład w algorytmie przeszukiwania posortowanej tablicy, gdzie w każdym kroku algorytmu będziemy pomijali część danych.

$O(n^2)$ Klasa *ta* często spotykana jest w rozważaniach kombinatorycznych, gdzie mamy do czynienia z regułą „każdy z każdym", np. dodawanie matryc o rozmiarach $n \times n$.

$O(2^n)$ Często przytaczana jest jako swego rodzaju straszak, choć w praktyce algorytmy tej klasy mogą być też używane, oczywiście jeśli zwracają wyniki w sensownym dla użytkownika czasie.

Przykład wyliczania złożoności algorytmu



A — czas wykonania instrukcji przypisania
C — czas wykonania instrukcji porównania

```
int tab[n][n];
void zerowanie(){
  int i,j;
  i=0;                    //A
  while (i<n){            //C
    j=0;                  //A
    while (j<=i){         //C
      tab[i][j]=0;        //A
      j=j+1;              //A
    }
    i=i+1;                //A
  }
}
```

Pętla `while` wykonuje n razy instrukcje zawarte w nawiasach klamrowych, warunek natomiast jest sprawdzany $n+1$ razy. Korzystając z tej uwagi i informacji zawartych w liniach komentarza, możemy napisać:

$$T(n) = C + A + \sum_{i=1}^{N}\left(2A + 2C + \sum_{j=1}^{i}(C + 2A)\right).$$

Po usunięciu sumy z wewnętrznego nawiasu otrzymamy:

$$T(n) = C + A + \sum_{i=1}^{N}(2A + 2C + i(C + 2A)). \qquad (*)$$

Przypomnijmy jeszcze użyteczny wzór na sumę szeregu liczb naturalnych od 1 do N:

$$1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2}.$$

Po jego zastosowaniu w równaniu (*) otrzymamy:

$$T(n) = C + A + 2N(A + C) + \frac{N(N+1)}{2}(C + 2A).$$

Ostateczne uproszczenie wyrażenia powinno nam dać:

$$T(n) = A\left(1 + 3N + N^2\right) + C\left(1 + 2{,}5C + \frac{N^2}{2}\right);$$

co sugeruje od razu, że analizowany program jest klasy $O(n^2)$.

Porady dotyczące optymalizacji

- Jeśli algorytm ma za zadanie coś obliczyć kilka razy, to jego optymalizacja może mijać się z celem. Taniej będzie spróbować go uruchomić lub ekstrapolować czas wykonania.
- Prostota: czasem optymalne algorytmy są zupełnie niezrozumiałe na pierwszy rzut oka i stopień ich skomplikowania łatwo prowadzi do błędów logicznych.
- Precyzja, tak potrzebna na przykład w algorytmach numerycznych, może być ważniejsza niż klasa algorytmu lub jego fragmentu.
- Notacja dużego O tak naprawdę odgrywa rolę dla takich danych wejściowych, dla których czas wykonania może zbliżać się do bardzo dużych liczb. W praktyce może to oznaczać, że algorytm „kwadratowy" będzie w pewnych konfiguracjach lepszy od „logarytmicznego"!

REKURENCJA

- Algorytmy *iteracyjne* polegają na n-krotnym wykonywaniu instrukcji w taki sposób, aby wyniki uzyskane podczas poprzednich iteracji (przebiegów) mogły służyć jako dane wejściowe do kolejnych (sterowanie przez instrukcje pętli, np. `for` lub `while`).
- Algorytmy *rekurencyjne* realizują zapętlenie nieco inaczej, a mianowicie poprzez wywoływanie tej samej procedury (funkcji) przez siebie samą z innymi parametrami.
- Matematycy stosują często zapis rekurencyjny, np. funkcja silnia: `0!=1, n!=n*(n-1)!`

Typy rekurencji

- Notacja matematyczna pokazana na przykładzie funkcji silnia może być określona jako *rekurencja naturalna*.
- Informatycy, w celu optymalizacji czasu wykonania programów, używają tzw. *rekurencji z parametrem dodatkowym*, gdzie parametry dodatkowe służą do przekazywania częściowo obliczonych wyników.

```
// Przykład rekurencji z parametrem dodatkowym
unsigned long int silnia (unsigned long
int x,
   unsigned long int tmp=1){
   if (x==0)
      return tmp;
   else
      return silnia(x-1, x*tmp);
} // Przykładowe wywołanie: silnia(5)
```

Funkcja Fibonacciego zapisana w formie rekurencyjnej

W ciągu Fibonacciego dwa wyrazy ciągu są równe 1, a każdy następny powstaje jako suma dwóch poprzednich: 1, 1, 2, 3, 5, 8, 13…

Jeżeli podzielimy przez siebie dowolne kolejne dwa wyrazy ciągu Fibonacciego, to ich stosunek będzie równy zawsze tej samej złotej liczbie ϕ (fi) równej w przybliżeniu 1,618.

Liczby ciągu Fibonacciego i złota liczba pojawiają się często w przyrodzie (np. rozmnażanie się zwierząt, wzrost roślin), muzyce (prawa harmonii), sztuce i naukach ścisłych.

$$fib(0) = 0,$$
$$fib(1) = 1,$$
$$fib(n) = fib(n-1) + fib(n-2), \text{ gdzie } n \geq 2$$

```
unsigned long int fib(int x){
   if (x<2)
      return x;
   else
      return fib(x-1)+fib(x-2);
}
```

Pułapki rekurencji

Rekurencja umożliwia proste opisanie skomplikowanych algorytmów, ale gdy jest realizowana przez fizyczne komputery, ukazuje istotne wady.

Niska efektywność wskutek wielokrotnego wywoływania tych samych fragmentów kodu.

Zobacz drzewko wywołań funkcji `fib(4)`, cała gałąź `fib(2)` jest zdublowana!

Nieskończona liczba wywołań rekurencyjnych dla pewnych konfiguracji danych wejściowych, np. `StadDoWiecznosci(2)`, co szybko prowadzi do zawieszenia się programu z powodu przekroczenia dostępnej pamięci.