Efektywność algorytmów

Algorytmika

- **Algorytmika** to dział informatyki zajmujący się poszukiwaniem, konstruowaniem i badaniem własności algorytmów, w kontekście ich przydatności do rozwiązywania problemów obliczeniowych za pomocą komputerów.
- Nazwa ta została wprowadzona przez izraelskiego informatyka Davida Harela w tytule jego książki o algorytmach: *Algorithmics. The Spirit of Computing.*
- Zadania algorytmiki:
  - projektowanie algorytmów i szacowanie ich złożoności obliczeniowej;
  - tworzenie ogólnych metod rozwiązujących pewne grupy zadań obliczeniowych;
  - badanie właściwości pewnych klas problemów obliczeniowych;
  - projektowanie struktur danych i operacji na tych strukturach;
  - analizowanie stanu po modyfikacjach struktur danych;
  - badanie złożoności obliczeniowej operacji na strukturach danych.

Od zadania do rozwiązania

- Analiza zadania
- Zaprojektowanie algorytmu
- Szacowanie złożoności
- Optymalizacja
- Implementacja i kompilacja
- Przygotowanie danych
- Uruchomienie i odczytanie wyników

Zadanie obliczeniowe

- **Problem obliczeniowy** to zadanie, które może być rozwiązane za pomocą komputera lub innej maszyny liczącej.
- **Specyfikacja** zadania:
  - zbiór **danych** wyjściowych
  - warunki jakie ma spełniać **wynik**
  - **zasoby** komputerowe do dyspozycji obliczeń (czas, pamięć, komunikacja, itp.)
- Problem obliczeniowy to funkcja, która przekształca zbiór danych wejściowych na zbiór danych wyjściowych.

Algorytm

- **Algorytm** to sposób na rozwiązanie zadania obliczeniowego (precyzyjnie zdefiniowany ciąg czynności, koniecznych do przeprowadzenia systemu z określonego stanu początkowego do zadanego stanu końcowego).
- Słowo "algorytm" pochodzi od nazwy "Algoritmi" – zlatynizowanej wersji nazwiska Abu Abdullaha Muhammada ibn Musy al-Chuwarizmiego, matematyka perskiego z IX wieku.
- Może istnieć wiele algorytmów rozwiązujących jeden problem obliczeniowy.
- Zapis algorytmu:
  - wskazówka
  - lista kroków
  - schemat blokowy
  - pesudokod

Złożoność obliczeniowa algorytmu

- **Złożoność obliczeniowa** algorytmu to jego zapotrzebowanie na zasoby komputerowe (czas, pamięć, komunikacja, itp.) potrzebne realizacji obliczeń.
- Złożoność obliczeniowa jest funkcją rozmiaru danych wejściowych.
- Złożoność obliczeniową algorytmu szacujemy od góry: ile zasobów z najgorszym przypadku wystarczy do przeprowadzenia obliczeń?
- Złożoność obliczeniową problemu szacujemy z dołu: ile zasobów zużyje dowolny algorytm rozwiązujący określony problem w najgorszym przypadku?

Złożoność obliczeniowa algorytmu

- **Złożoność pamięciowa** to zapotrzebowanie algorytmu na pamięć, którą liczymy w **komórkach**:
  - **dane wejściowe** dostarczone do algorytmu **nie** wchodzą do kosztów pamięciowych;
  - wywołania funkcji generują koszty pamięciowe – w szczególności **wywołania rekurencyjne**.
- **Złożoność czasowa** to zapotrzebowanie algorytmu na czas, którą liczymy za pomocą **operacji dominujących**:
  - operacja dominująca to proste obliczenie, działające w czasie stałym, które ma najistotniejszy wpływ na liczbę kroków algorytmu.

Złożoność obliczeniowa algorytmu

- Złożoność obliczeniowa zwykle nie zależy wyłącznie od rozmiaru danych, ale może się znacznie różnić dla danych wejściowych o identycznym rozmiarze. Często stosowanymi kryteriami są:
  - **złożoność pesymistyczna** – rozpatrywanie przypadków najgorszych;
  - **złożoność oczekiwana** – zastosowanie określonego sposobu uśrednienia wszystkich możliwych przypadków;
  - złożoność optymistyczna – rozpatrywanie przypadków najprostszych (rzadko stosowane kryterium).

Asymptotyka

- Złożoność obliczeniowa jest zwykle bardzo skomplikowaną funkcją, dlatego szacuje się ją z dokładnością do pewnego stałego czynnika (czasem z dokładnością do nieistotnego składnika) za pomocą operatorów asymptotycznych.
- Założenia:
  - rozpatrujemy funkcje $\mathbb{N} \to \mathbb{R}_+$
  - funkcja graniczna to g(n)
- Notacja dużego **O**, **Ω** i **Θ** została zaproponowana po raz pierwszy w roku 1894 przez niemieckiego matematyka Paula Bachmanna. W późniejszych latach spopularyzował ją w swoich pracach Edmund Landau, niemiecki matematyk, stąd czasem nazywana jest notacją Landaua.

Asymptotyka

- Ograniczenie górne wyznaczone przez funkcję g(n):
  O(g(n)) = { f(n) : ∃ c>0 ∃ n_0 ∀ n ≥ n_0 : f(n) ≤ c·g(n) }
- Ograniczenie dolne wyznaczone przez funkcję g(n):
  Ω (g(n)) = { f(n) : ∃ c>0 ∃ n_0 ∀ n ≥ n_0 : c·g(n) ≤ f(n) }
- Ograniczenie dokładne wyznaczone przez funkcję g(n):
  Θ (g(n)) = O(g(n)) ∩ Ω (g(n))
  Θ (g(n)) = { f(n) : ∃ 0<c_1<c_2 ∃ n_0 ∀ n ≥ n_0 :
  c_1·g(n) ≤ f(n) ≤ c_2·g(n)}
- Asymptotyczna równość funkcji f(n) ~ g(n):
  lim[n→∞] f(n)/g(n) = 1

Potęgowanie

- Dane: $x \in \mathbb{R}$, $n \in \mathbb{N}$
- Zadanie: obliczyć x^n
- Rozwiązanie naiwne: $x \cdot x \cdot x \cdot \ldots \cdot x$ (n razy)
- Algorytm:

```
function potega(x, n) {
        if (n = 0) then return 1;
        r := x;
        for i = 2…n do r := r · x;
        return r;
}
```

- Złożoność czasowa (liczba mnożeń): O(n)
- Złożoność pamięciowa O(1)

Szybkie potęgowanie

- Idea: najpierw policzymy $x^{\lfloor n/2 \rfloor}$ a potem podniesiemy wynik do kwadratu
- Algorytm:

```
function qpow (x, n) {
        if (n = 0) then return 1;
        c := qpow(x, ⌊n/2⌋);
        if (n jest parzyste) then return c · c;
        else return c · c · x;
}
```

- Złożoność czasowa (liczba mnożeń): O(log n)
- Złożoność pamięciowa (głębokość rekurencji): O(log n)

Potęgowanie binarne

- Idea: przedstawienie wykładnika n w zapisie binarnym $n = (n_k, n_{k-1}, \ldots n_1, n_0)$
- Algorytm:

```
function bpow (x, n) {
        y := x;
        r := 1;
        while (n > 0) {
                if (n jest nieparzyste) then r := r · y;
                y := y ^2;
                n := ⌊n/2⌋;
        }
        return r;
}
```

- Złożoność czasowa (liczba mnożeń): O(log n)
- Złożoność pamięciowa: O(1)

Liczby Fibonacciego

- Liczby Fibonacciego to ciąg liczb zdefiniowany rekurencyjnie:
  $F_0 = 0$
  $F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
- Algorytm:

```
function fib (n) {
        if (n < 2) then return n;
        else return fib(n-1) + fib(n-2);
}
```

- Funkcja rekurencyjna działa w czasie wykładniczym!
  Niech R(n) oznacza liczbę wywołań rekurencyjnych funkcji fib(n). Można indukcyjnie wykazać, że
  $R(n) \in \Omega(2^{n/2})$

Liczby Fibonacciego

- Algorytm iteracyjny wpisujący kolejne liczby Fibonacciego do tablicy.
- Algorytm:

```
function fib (n) {
        if (n < 2) then return n;
        new f[n+1];
        f[0] := 0;
        f[1] := 1;
        for i = 2…n do f[i] := f[i-1] + f[i-2];
        return f[n];
}
```

- Złożoność czasowa: O(n)
- Złożoność pamięciowa: O(n)

Liczby Fibonacciego

- Algorytm iteracyjny wyliczający kolejne liczby Fibonacciego i pamiętający tylko dwie ostatnie liczby.
- Algorytm:

```
function fib (n) {
        new f[3];
        f[0] := 0;
        f[1] := 1;
        f[2] := 1;
        if (n ≤ 2) then return f[n];
        for i = 3…n do { f[0] := f[1]; f[1] := f[2]; f[2] := f[1]+f[0]; }
        return f[2];
}
```

- Złożoność czasowa: O(n)
- Złożoność pamięciowa: O(1)

Liczby Fibonacciego

- Idea: macierz transformacji M przekształcająca wektor $[F_{n-1}, F_{n-2}]$ w $[F_n, F_{n-1}]$ za pomocą macierzy przekształcenia $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.
- Wyjaśnienie na Wikipedii: https://pl.wikipedia.org/wiki/Ci%C4%85g_Fibonaccieggo#Obliczanie_liczb_Fibonacciego
- Złożoność czasowa: O(log n), bo potrafimy potęgować w czasie O(log n)
- Złożoność pamięciowa: O(1), bo potęgowanie binarne wymaga jedynie pamięci O(1)