Wykład:
Złożoność czasowa dla początkujących.

Autorzy:
• v. 1.00: 2010-09-13, Jacek Tomasiewicz

PROSERWY 2010

1 Wstęp

Złożoność czasowa to jeden z najważniejszych parametrów charakteryzujących algorytm. Decyduje on o efektywności całego programu.

2 Czym jest złożoność

Definicja.1. Złożoność czasowa *programu jest to maksymalna liczba operacji, jakie może wykonać program w zależności od wczytanych danych.*

Badając złożoność zawsze wybieramy operację *dominującą* – czyli taką, która będzie wykonywać się najwięcej razy w zależności od wczytanej wartości n.
Spójrzmy na przykład:

```
1. wynik := 0
2. cin >> n
3. for i := 1 to n do
4.    wynik := wynik + i
5. cout << wynik
```

Znajdźmy operację dominującą – jest to z pewnością operacja w linijce 4.:

```
4. wynik := wynik + i
```

Zauważmy, że wykona się ona n razy. I już mamy policzoną złożoność czasową. Wynosi ona $O(n)$ i nazywamy ją *liniową* złożonością.

Jeśli linijka 3. wyglądałaby następująco:

```
3. for i := 1 to 20*n do
```

to liczba operacji wynosi $20n$, jednak wszystkie stałe pomijamy i złożoność pozostaje liniowa. Powiemy więc, że algorytm który wykonuje kn operacji, gdzie k jest stałą (np. $\frac{1}{2}$, 20, 100) i algorytm który wykonuje n operacji mają taką samą złożoność czasową.

Tak więc analizując program będziemy badać jaka jest największa liczba operacji, które może on wykonać przy określonym rozmiarze danych. Często będziemy wyszukiwać specjalne, „złośliwe" przykłady danych, na których dany program będzie działał długo. Czemu tak należy robić? Po pierwsze dzięki temu wiemy, że program będzie działał nie wolniej niż ..., a jak się skończy szybciej, to tylko niespodzianka na plus. Po drugie zaś, okazuje się że pesymistyczne ograniczenie bardzo często jest osiągane – na przykład większość algorytmów wyszukiwania elementu w zbiorze działa najdłużej, gdy szukanego elementu nie ma, co jest w praktyce bardzo częste.

3 Przykłady różnych złożoności

3.1 Złożoność stała

```
1. cin >> n
2. cout << n * n
```

Program wykona zawsze stałą liczbę operacji nie zależnie od wartości n. Złożoność jest więc *stała* i zapisujemy ją jako $O(1)$.

3.2 Złożoność logarytmiczna

```
1. cin >> n
2. wynik := 0
3. while (n > 1) do
4.   n := n / 2
5.   wynik := wynik + 1;
6. cout << wynik
```

W linijce 4. wartość n jest w każdym obrocie pętli zmniejszane o połowę. Liczbę takich operacji oblicza się używając logarytmów. Jeśli $n = 2^x$ to $log(n) = x$. Łatwo więc zauważyć, że złożoność tego rozwiązania wynosić będzie $O(log(n))$, a nazywać będziemy ją złożonością *logarytmiczną*.

3.3 Złożoność liniowa

```
1. cin >> n
2. for i := 1 to n do
3.    cin >> x
4.    if x = 0 then
5.       break
```

Zauważmy, że jeśli pierwszą wczytaną liczbą x byłoby 0 to progam od razu by się zakończył, my jednak szukamy złośliwych przypadków – będzie to taki, w którym x równy 0 nigdy nie występuje. Złożoność jest więc *liniowa*, czyli $O(n)$.

3.4 Złożoność liniowo-logarytmiczna

```
1. cin >> n
2. for i := 1 to n do
3.   cin >> p          // p z przedziału od 1 do n
4.   while (p < n)
5.      p := p * 2
6.   cout << p
```

Znów znajdźmy „złośliwy” przypadek. Będzie on wtedy, gdy $p = 1$. Wówczas w każdym obrocie pętli z linijki 2. instrukcja w linijce 5. wykonywać się będzie $log(n)$ razy. Stąd cała złożoność wynosi $O(n * log(n))$ i nazywamy ją *liniowo-logarytmiczną*.

3.5 Złożoność kwadratowa

```
1. suma_i := suma_j := 0
2. cin >> n
3. for i := 1 to n do
4.    for j := i to n do
5.       suma_i := suma_i + i
6.       suma_j := suma_j + j
7. cout << suma_i + suma_j
```

Zauważmy, że operacja w linijce 5. będzie wykonywana nie rzadziej niż operacja w każdej innej linijce. Policzmy liczbę wykonań tej operacji.

Pętla w linijce 4. będzie się wykonywała n, $n-1$, $n-2$, ... 1 razy. Po zsumowaniu liczba operacji wynosić będzie $\frac{n*(n+1)}{2} = \frac{1}{2} * (n^2 + n) \leqslant n^2$, więc złożoność wynosi $O(n^2)$ i nazywamy ją *kwadratową*.

3.6 Złożoność wykładnicza

```
1. cin >> n
2. p := 1
3. for i := 1 to n
4.    p := p * 2
5. for i := 1 to p
6.    wynik := wynik + i
7. cout << wynik
```

Ile operacji wykona powyższy kod. Pętla w linijce 3. obróci się n razy jednak pętla w linijce 6. obróci się $p = 2^n$ razy. Wybieramy operację dominującą, więc złożoność wynosić będzie $O(2^n)$, która nazywana jest złożonością *wykładniczą*. Złożonościami wykładniczymi są również $O(3^n)$, $O(10^n)$, a także $O(n!)$.

3.7 Wiele zmiennych

Nie zawsze złożoność musi zależeć od jednej zmiennej:

```
1. cin >> n
2. cin >> m
3. for i := 1 to n do
4.    cout << i
5. for j := 1 to m do
6.    cout << j
```

Zauważmy, że czas działania programu zależy od obu wartości n i m, w związku z tym złożoność wynosi $O(n + m)$. Mówimy, że złożoność jest *liniowa* względem n i m.

Spójrzmy na inny przykład:

```
1. cin >> n
2. cin >> m
3. if n > m then
4.    for i := 1 to n do
5.       cout << i
6. else
7.    for j := 1 to m do
8.       cout << j
```

Złożoność zależy od wartości większej z liczb n i m. Wynosi więc ona $O(max(n, m))$.

I jeszcze jeden:

```
1. cin >> n
2. cin >> m
3. wynik := 0
4. for i := 1 to n do
5.    for j := 1 to m do
6.       wynik := wynik + i * j
7. cout << wynik
```

Złożoność zależy od obu wartości n i m. W każym z n obrotów pętli w linii 3. wykonuje się pętla z linii 4. obracająca m razy. Stąd złożonośc wynosi $O(n * m)$.

4 Czy program ma wystraczającą złożoność

Podczas pisania programu, zawsze warto zastanowić się czy istnieje rozwiązanie o lepszej złożoności czasowej. Chyba, że limity wyraźnie wskazują, że wymyślone rozwiązanie, choć może nieoptymalne zostanie w pełni zaakceptowane.

Porównajmy 3 programy obliczające dokładnie to samo:

4.1 Program A

```
1. cin >> n
2. for i := 1 to n do
3.    for j := 1 to i do
4.       wynik := wynik + 1
5. cout << wynik
```

Złożoność: $O(n^2)$

4.2 Program B

```
1. cin >> n
2. for i := 1 to n do
3.    wynik := wynik + i
4. cout << wynik
```

Złożoność: $O(n)$

4.3 Program C

```
1. cin >> n
2. wynik := n * (n + 1) / 2
3. cout << wynik
```

Złożoność: $O(1)$

Chcielibyśmy stwierdzić który program będzie akceptowalny w czasie zawodów. Obecnie można przyjąć, że aby program był akceptowalny w pełni przez sprawdzarkę to może wykonywać około 10^8 operacji – tyle operacji jest wykonywane przez mniej więcej 1 sekundę.

Jeśli mamy zadeklarowane, że $1 \leqslant n \leqslant 10^4$ to wszystkie programy A, B, C powinny być zaakceptowane przez sprawdzarkę.

Jeśli $1 \leqslant n \leqslant 10^6$ to Program A wykona około 10^{12} operacji co niestety wywoła przekroczenie limiu czasu – natomiast program B i C powinien zostać zaakcpetowany.

Jeśli $1 \leqslant n \leqslant 10^{10}$ to tylko program C może zostać zaakceptowany.

Warto zwrócić uwagę, że wczytywanie i wypisywanie danych jest szczególnie wolne. Wczytanie lub wypisanie więcej niż 10^6 liczb może wykonywać się ponad 1 sekundę (w zadaniach raczej nie zdarza się aby potrzebne było wczytanie istotnie wielu liczb)

4.4 Podsumowanie

1. Jeśli $n \leqslant 10^6$ to próbujmy pisać programy o maksymalnej złożoności $O(n)$ lub $O(n * log(n))$.
2. Jeśli $n \leqslant 10^4$ to próbujmy pisać programy o maksymalnej złożoności $O(n^2)$
3. Jeśli $n \leqslant 500$ to próbujmy pisać programy o maksymalnej złożoności $O(n^3)$

Oczywiście nie są to ścisłe limity, choć z pewnością przybliżone. Często zależą one od konkretnego zadania. Na pewno jeśli stwierdzimy, że program wykonuje dużo ponad 10^8 operacji to powinniśmy się zastanowić czy nie istnieje szybsze rozwiązanie naszego problemu.