Laboratorium AISDE – wskazówki

Niniejszy skrypt powstał po przeanalizowaniu kilku tysięcy sprawozdań z laboratoriów i zajęć projektowych z przedmiotu *Algorytmy i Struktury Danych*. Zawiera garść porad oraz sporą dawkę przykładów typowych błędów popełnianych na etapie badania algorytmów, prezentacji danych i interpretacji wyników. Wiele z tych błędów sprawia wrażenie absurdalnych, jednak każdy z nich przytrafia się dostatecznie często, by poświęcić mu chwilę uwagi.

Poniższe zalecenia nie mają oczywiście na celu krępowania inwencji studentów wykonujących ćwiczenie. Wręcz przeciwnie – własne pomysły są wysoko punktowane. Chodzi przede wszystkim o to, by przed wykonaniem jakiegokolwiek eksperymentu zastanowić się, na jakie pytanie ma on odpowiedzieć.

Planowanie eksperymentu

Oszacowanie jakiejkolwiek cechy ilościowej algorytmu, np. złożoności czasowej, jest tym bardziej wiarygodne, im szerszy przyjmiemy zakres wielkości zbiorów wejściowych. Eksperyment przeprowadzony np. na tablicy o liczbie elementów zmieniającej się od 4.000 do 6.000 trudno uznać za udany – prowadzi z reguły do absurdalnego wniosku, że „czas działania algorytmu jest niemal niewrażliwy na rozmiar problemu". W prawidłowym eksperymencie zakres zmienności rozmiaru problemu obejmuje kilka rzędów wielkości. Niestety, zakres ten trudno zdefiniować a priori – na przykład pewne algorytmy analizy grafów radzą sobie w czasie kilkudziesięciu milisekund z grafami o kilkuset tysiącach krawędzi, podczas gdy dla innych analiza grafu o stu krawędziach jest praktycznie niewykonalna. Z reguły w ćwiczeniu porównywanych jest kilka algorytmów realizujących podobne funkcje. Jeśli któryś z nich jest zdecydowanie szybszy od innych, warto dodatkowo uruchomić go dla bardzo dużych zbiorów, niemożliwych do obróbki pozostałymi procedurami. Podobnie, jeśli któraś metoda jest wyraźnie wolniejsza od innych, trzeba dla niej – i **tylko** dla niej – pominąć analizę największych zbiorów.

Z tego względu „formalny" eksperyment powinien być poprzedzony kilkoma próbami, które pozwolą określić zakres wielkości zbiorów danych. Górne ograniczenie jest wyłącznie kwestią cierpliwości osoby wykonującej ćwiczenie. Natomiast na dolne ograniczenie wpływa kilka czynników. Jednym z nich jest rozdzielczość rejestracji czasu. Jeśli wynosi ona jedną milisekundę, to wynik 2 ms jest potencjalnie obarczony dwudziestopięcioprocentowym błędem i sugeruje, żeby zwiększyć zbiór wejściowy przynajmniej kilkukrotnie. Ponadto na szybkość wykonania algorytmu mają wpływ inne procesy uruchomione na komputerze, w tym procesy systemowe. Im większy czas działania, tym bardziej uśredniony zostaje wpływ tych procesów na czas wykonania algorytmu i tym bardziej wiarygodne są wyniki. Należy wreszcie pamiętać, że gdy dane wejściowe traktujemy jako losowe, to wiarygodne wnioski można wyciągać dopiero dla dostatecznie dużych zbiorów. Jeśli np. dwukrotnie rzucimy monetą, to uzyskanie dwóch orłów nie daje nam podstaw do stwierdzenia, że moneta jest oszukana – eksperyment musimy kontynuować. Podobnie ciąg liczb o kilku lub kilkunastu elementach, choć uzyskany w drodze losowania, może niewiele różnić się od ciągu posortowanego (w pożądanym kierunku lub odwrotnie), co może prowadzić do fałszywych wniosków, jeśli ciągu tego użyjemy następnie do szacowania wydajności algorytmu sortowania.

Krok, z jakim modyfikujemy liczność zbioru danych, nie musi być stały. Załóżmy, że oczekujemy, że w wyrażeniu opisującym złożoność obliczeniową algorytmu wystąpi logarytm. Jeśli chcemy oszacować czas działania tego algorytmu dla kilku tablic o rozmiarach od 100 do 1.000.000 elementów, to dobrym pomysłem będzie powiększanie tablicy o stały czynnik, czyli np. użycie sekwencji 100, 1.000, 10.000 itd.

Realizacja eksperymentu

Każde z ćwiczeń wymaga uruchomienia dostępnych w laboratorium programów dla wielu – kilkunastu a czasem kilkudziesięciu – różnych zbiorów danych, wartości parametrów i innych ustawień. Oczywiście takie zadanie można sobie ułatwić, automatyzując proces modyfikacji parametrów, kompilacji programu, jego uruchomienia i obróbki wyników. Można w tym celu zmodyfikować oryginalny kod lub posłużyć się zewnętrznym skryptem napisanym w dowolnym języku programowania, np. języku powłoki systemu Unix, Perlu, Pythonie itp. Należy jednak pamiętać o dwóch sprawach.

- Ćwiczenie ma na celu naukę analizy algorytmów, a nie wykazanie się znajomością języków skryptowych, programów graficznych itp. Dlatego o zastosowaniu tego typu udogodnień można w sprawozda-

niu jedynie **krótko** napomknąć.

- Przeniesienie części pracy na komputer nie zwalnia użytkownika od odpowiedzialności za sensowność otrzymanych wyników. W szczególności należy mieć na uwadze zalecenia podane w punkcie *Planowanie eksperymentu*.

Nieco inną kwestią jest wykorzystanie bardziej zaawansowanych metod lub/i narzędzi analizy danych, np. do aproksymacji uzyskanych wyników czy ich analizy statystycznej. Tutaj jak najbardziej wskazane jest podanie przyjętej strategii działania, poczynionych założeń, liczbowych miar jakości rozwiązania i innych istotnych informacji.

Teoretyczna analiza właściwości algorytmów

Kluczowym elementem większości ćwiczeń jest oszacowanie właściwości analizowanych algorytmów – chodzi głównie o złożoność obliczeniową, lecz także np. o stabilność sortowania. Po wykonaniu eksperymentów **konieczne** jest porównanie wyników otrzymanych z przewidywanymi. Te szacowania teoretyczne powinny być oparte na analizie kodu programów użytych w ćwiczeniu. Choć programy te mogą na pierwszy rzut oka wydawać się bardzo rozbudowane, to badane algorytmy są w nich zaimplementowane w postaci funkcji liczących kilka, rzadziej kilkanaście, wierszy kodu.

Analiza uruchamianego kodu jest **niezbędna** i należy opierać się pokusie porównywania wyników eksperymentu wyłącznie z informacjami o algorytmach uzyskanymi z literatury. Zachowanie algorytmu zależy bowiem silnie od szczegółów implementacji. Można wśród nich wymienić:

- Struktury danych, na których operuje algorytm.
- Warunek zakończenia całego algorytmu lub pewnych pętli wewnętrznych – np. nawet bardzo prosty algorytm sortowania bąbelkowego można ulepszyć tak, by kończył działanie „przed czasem”, o ile ostatni przebieg nie przyniósł żadnej zmiany.
- Sposób porównywania elementów (np. przy sortowaniu). Wybór pomiędzy nierównością ostrą a nieostrą często decyduje o ważnych cechach algorytmu.

Źródła literaturowe często podają złożoność obliczeniową algorytmu zaimplementowanego w sposób „optymalny” lub „typowy”, zwykle bez wytłumaczenia, co kryje się za tymi określeniami. Im bardziej złożony algorytm, tym większa liczba sposobów, w jakie można go zaimplementować. Dlatego analiza wyników musi być oparta na konkretnym kodzie, który pozwolił te wyniki uzyskać. Oczywiście warto potem przeprowadzić dyskusję, czy implementacja zaproponowana w ćwiczeniu jest optymalna i zaproponować własne udoskonalenia.

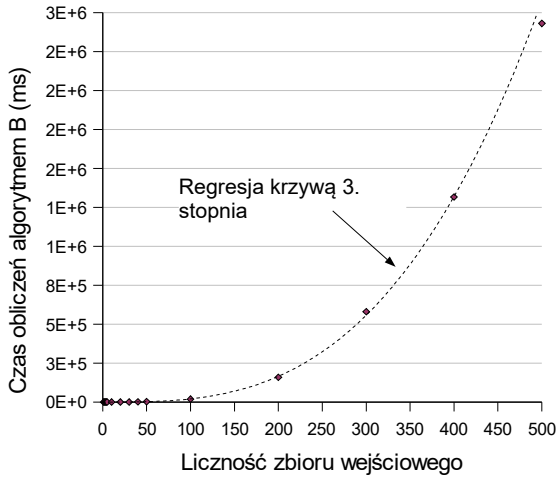
Za wszelką cenę należy wystrzegać się ogólników w rodzaju: „Powyższy algorytm cechuje się małą zajętością stosu oraz w miarę dobrą wydajnością”, zwłaszcza jeżeli żadna z tych wielkości nie jest wyrażona w sprawozdaniu bardziej formalnie, np. na wykresie czy w tabeli. Również stwierdzenie, że „czas wykonania algorytmu był pomijalnie mały” nie świadczy dobrze o rzetelności przeprowadzonych badań. To, czy czas może być traktowany jako pomijalny, zależy wyłącznie od kontekstu.

Analiza wyników eksperymentów

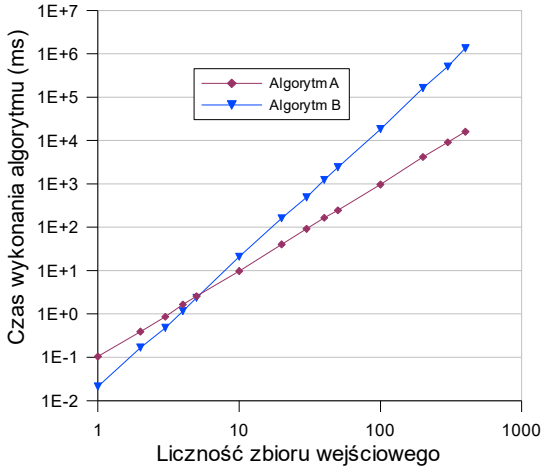
Nieocenionym narzędziem, pozwalającym oszacować charakter zależności między dwiema zmiennymi, jest wykres jednej z nich w funkcji drugiej. W sprawozdaniu przedstawiany jest z reguły czas obliczeń w funkcji wielkości zbioru wejściowego. Najczęściej spotykane w sprawozdaniach zdanie zaczyna się od słów *Jak widać na wykresie, złożoność wynosi...* I tu zaczyna się problem, bo w większości przypadków autor widzi to, co chce zobaczyć, zaś dla osób trzecich (w tym oceniającego) wykres przedstawia jedynie nieokreśloną krzywą rosnącą. Dlatego przy sporządzaniu wykresów należy zwrócić uwagę na kilka rzeczy.

Dobrym sposobem demonstrowania, do jakiej klasy złożoności należy algorytm, jest wykreślenie czasu pracy wraz z taką krzywą, która zapewnia najlepsze dopasowanie. Rodzaj krzywej (potęgowa, logarytmiczna itp.) dobieramy na podstawie znajomości teoretycznej złożoności algorytmu. Parametry krzywej można dobrać własnoręcznie lub dokonać aproksymacji średniokwadratowej za pomocą arkusza kalkulacyjnego lub dowolnego pakietu do obróbki i prezentacji danych. Przykładowy wykres tego typu został pokazany na Rys. 1a.

a)



b)



Rys. 1. Przykłady **dobrych** wykresów: czytelny format liczb na osiach, porównanie danych eksperymentalnych z wynikiem aproksymacji (a), odpowiednia skala (tu – podwójnie logarytmiczna), jasno wskazująca na potęgowy charakter zależności (b)

Inną, bardzo elegancką formą przedstawienia charakteru zależności między dwiema wielkościami jest taki dobór skal na osiach, by punkty ułożyły się wzdłuż linii prostej. Skala na jednej lub obu osiach może być więc liniowa lub logarytmiczna. Zamiast zmiennej – zależnej lub niezależnej – można również użyć pewnej jej funkcji. Gdy spodziewamy się złożoności $O(n \log_2 n)$ – gdzie n jest rozmiarem problemu – warto wykreślić czas obliczeń podzielony przez n w funkcji $\log_2 n$. W doborze właściwego układu współrzędnych pomocna jest teoretyczna analiza złożoności algorytmu. Jeśli np. spodziewamy się potęgowej zależności czasu obliczeń t od liczności n zbioru danych:

$$t = b\, n^a,$$

to warto użyć skali logarytmicznej na obu osiach. Po zlogarytmowaniu obu stron powyższej zależności otrzymujemy:

$$\log_{10} t = a \log_{10} n + \log_{10} b.$$

Po wprowadzeniu zmiennych będących logarytmami zmiennych pierwotnych i oznaczeniu ich według konwencji $X \equiv \log_{10} x$, otrzymujemy:

$$T = a\, N + B.$$

Na wykresie w skali podwójnie logarytmicznej otrzymamy zatem prostą o współczynniku kierunkowym a dekad na dekadę. Przykładem może być Rys. 1b, na którym przedstawiono wpływ liczności zbioru danych na czas wykonywania dwóch algorytmów: o złożoności kwadratowej i sześciennej. Złożoności te można łatwo odczytać z nachylenia otrzymanych prostych.

W opisany sposób można – i warto – analizować kilka algorytmów na wspólnym wykresie. Jest to jednak niemożliwe, gdy algorytmy należą do różnych klas złożoności – np. $O(n^2)$ i $O(n \log_2 n)$. Należy wtedy sporządzić **osobne** wykresy – każdy w odpowiedniej skali – a oprócz tego porównać wszystkie algorytmy na dodatkowym, wspólnym wykresie. Jeśli wartości w seriach danych różnią się o kilka rzędów wielkości, porównanie takie wymaga oczywiście użycia skali logarytmicznej.

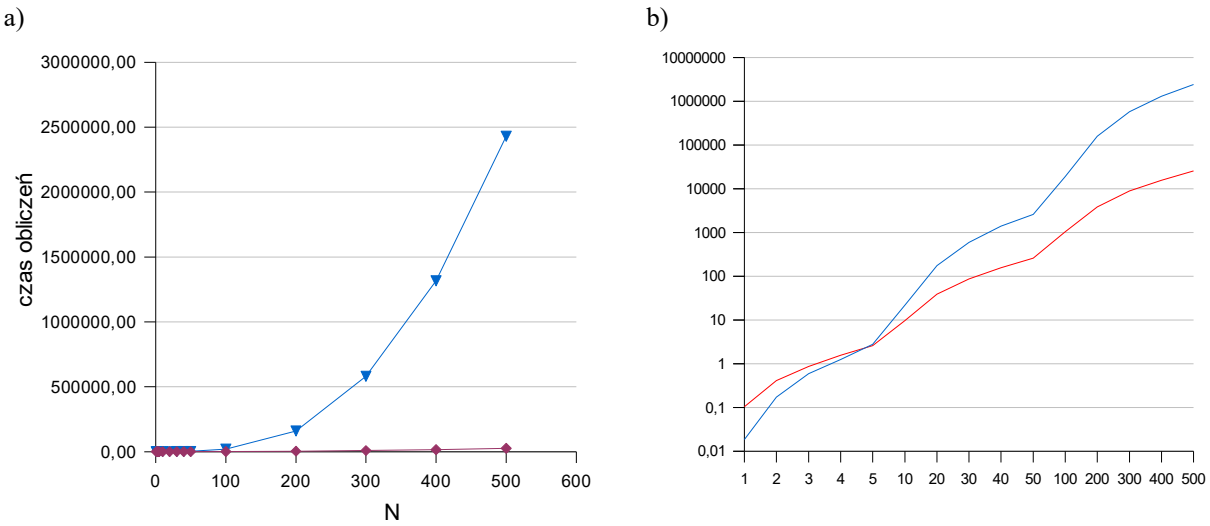
Tutaj **ważna uwaga**. Przy sporządzaniu wykresu mamy do wyboru (między innymi) skale lin-lin, lin-log, log-lin i log-log. W zależności od klasy złożoności problemu użycie jednej z tych skal pozwala często (choć nie zawsze) uzyskać linię zbliżoną do prostej, natomiast pozostałe skale dają w efekcie bliżej nieokreślone krzywe. Niestety często zdarza się, że kompletnie nieczytelnym wykresom towarzyszy magiczna formuła *Wykreślenie czasu w skali logarytmicznej pozwoliło stwierdzić, że…* – i tu różne osoby stwierdzają różne rzeczy w zależności od tego, co chciały na tych wykresach zobaczyć.

W rzeczywistości obserwacje rzadko układają się dokładnie wzdłuż oczekiwanej prostej lub krzywej. Szczególnie dużym odchyleniom podlegają obserwacje bardzo małych czasów (patrz punkt *Planowanie eksperymentu*). Dlatego do analizy współczynnika nachylenia należy dobrać taki zakres danych, w jakim zależność jest rzeczywiście dobrze przybliżona np. prostą. Jeśli nie można znaleźć wystarczająco dużego zakresu liniowych zmian, należy poważnie zastanowić się nad wykreśleniem serii danych w innym układzie współrzędnych. Natomiast samą wartość współczynnika nachylenia w znalezionym zakresie lepiej obliczyć bezpośrednio z zanotowanych danych, niż „na oko" oszacować na wykresie.

Nawet najlepszy wykres jest bezużyteczny bez objaśnienia, jakie wartości zostały wykreślone ani w jakich jednostkach zostały wyrażone. Taki opis może znaleźć się na osiach wykresu, w jego legendzie lub – ostatecznie – w tekście sprawozdania.

Na zakończenie podajemy ku przestrodze dwa przykłady wyjątkowo nieczytelnych wykresów (Rys. 2). Przedstawiono na nich te same serie danych, co na Rys. 1b (!). Typowe „wnioski" wyciągane z wykresu Rys. 2a brzmią następująco: *Jak widać, seria danych oznaczona trójkątami układa się wzdłuż paraboli/krzywej wykładniczej. Natomiast dla serii danych oznaczonej rombami liczność zbioru danych ma pomijalny/liniowy/logarytmiczny wpływ na czas obliczeń*. W rzeczywistości serię danych oznaczoną trójkątami można najlepiej przybliżyć wielomianem trzeciego stopnia, ale należałoby to udowodnić, używając którejś z technik opisanych powyżej. Stwierdzenie, że coś „widać" na pewno nie jest wystarczające. Jeszcze gorzej jest z wnioskami dotyczącymi serii danych oznaczonej trójkątami: w rzeczywistości zależność jest kwadratowa, a więc relatywnie silna, lecz skala przyjęta na Rys. 2a uniemożliwia jakiekolwiek szacowanie.

Innym, **zdumiewająco częstym** błędem jest wykonywanie wykresów takich jak na Rys. 2b, gdzie każdemu kolejnemu punktowi danych jest przydzielony jednakowy przedział na osi odciętych. Niestety, w większości arkuszy kalkulacyjnych ten format wykresu – określany jako liniowy – jest formatem domyślnym i przez to jest on bardzo często wykorzystywany odruchowo. Wykres taki może być użyteczny wyłącznie w przypadku, gdy wartość zmiennej niezależnej przyrasta ze stałym krokiem. W przypadku zbiorów danych przedstawionych na Rys. 2b jest oczywiście inaczej. Obserwując krzywe trudno się domyślić, że są opisane funkcjami potęgowymi. Inne, mniej istotne błędy popełnione na Rys. 2 to brak legend, brakujące lub niejednoznaczne opisy osi (co to jest *N*?), brak jednostek oraz nieczytelny format liczb na osi rzędnych – format „naukowy" pozwoliłby uniknąć dużej liczby zer.



Rys. 2. Przykłady **złych** wykresów, przedstawiających te same serie danych, co Rys. 1b: niejasny opis osi odciętych, brak jednostek, brak legendy, seria danych „zlewa się" z osią, nieczytelny format liczb na osi rzędnych (a), nieprawidłowa skala na osi odciętych – kolejne wartości zmiennej niezależnej 3, 4, 5, 10 itp. są oddzielone jednakowymi odstępami, choć nie tworzą żadnego sensownego ciągu (b)

Własne implementacje algorytmów

Elementem niemal każdego z ćwiczeń jest własnoręczna implementacja jakiegoś algorytmu lub modyfikacja istniejącego. Zadanie to nie wymaga dużych umiejętności programistycznych, a liczba wierszy kodu koniecznych do dopisania nie przekracza kilkunastu. Najważniejsze jest wykazanie się zrozumieniem, jak napisany kod działa, toteż nieodzownym elementem takiego zadania jest krótki opis.

Każda napisana samodzielnie funkcja – lub modyfikacja istniejącej – powinna być poddana testowaniu, którego rezultat znajdzie się w sprawozdaniu. Podczas testowania sprawdzamy kolejno, czy po wprowadzeniu zmian program:

- kompiluje się,
- zwraca prawidłowe rezultaty,
- działa lepiej od oryginału pod jakimś względem, np. szybkości.

Intuicyjny test prawidłowości działania polega na próbie rozwiązania problemu na tyle niewielkiego, by wyniki działania programu przed modyfikacją i po niej można było łatwo porównać wzrokowo. Przykładem może być próba posortowania tablicy o dziesięciu elementach i zaobserwowanie wyników na ekranie. Warto przy tym uwzględnić przypadki „skrajne". Implementując algorytm wykorzystujący zasadę „dziel i zwyciężaj" warto się zastanowić, co jest dla niego elementarnym przypadkiem (np. fragment tablicy zawierający jeden lub dwa rekordy) i jak powinien się on zachować po napotkaniu takiego przypadku. Jeśli np. algorytm dzieli pewną tablicę na pół, warto sprawdzić, czy zachowuje się poprawnie dla tablicy o nieparzystej liczbie rekordów. Błędne działanie programu wynika często z użycia nierówności ostrych tam, gdzie konieczne są nieostre (lub na odwrót). W rezultacie wykonywana jest jedna iteracja za mało lub przeciwnie – program próbuje sięgnąć poza strukturę danych, na której operuje.

Niestety wiele błędów programistycznych objawia się dopiero przy próbie rozwiązania odpowiednio dużych problemów. Oczywiście prawidłowość posortowania tablicy o tysiącu elementów trudno ocenić wzrokowo. W tym konkretnym przypadku trzeba będzie napisać dodatkową funkcję sprawdzającą uporządkowanie kolejnych par elementów.

Nawet jeśli zmodyfikowany program nie działa do końca prawidłowo, warto zamieścić go w sprawozdaniu, przy czym uzyskane wyniki należy koniecznie skomentować i zastanowić się nad przyczyną błędnego działania. Natomiast **wysoce niewskazane** jest zamieszczanie w sprawozdaniu kodu, który już na pierwszy rzut oka nie ma szans nawet na pomyślną kompilację.

Opis kodu – własnego lub dostarczonego przez prowadzącego zajęcia – ma wyjaśniać, w jaki sposób realizuje on algorytm i w jakich przypadkach wykonywane są jego poszczególne sekcje (pętle czy instrukcje warunkowe). Interpretacja kodu **nie** polega natomiast na tłumaczeniu „wiersz po wierszu" wykonywanych operacji – do tego służą komentarze w kodzie, których liczbę można zresztą znacznie ograniczyć, jeśli dobierze się odpowiednie nazwy zmiennych.

Osobną kwestią jest estetyka i czytelność prezentowanego kodu. Oczywiście nie jest ona najważniejsza, ale warto trzymać się następujących zaleceń:

- Kod powinien być prosty i zwięzły.
- Czasem zadanie polega nie na napisaniu kompletnej funkcji, lecz na modyfikacji gotowego fragmentu kodu, zwykle przez dodanie kilku poleceń w różnych miejscach. W takiej sytuacji nieuniknione jest zamieszczenie w sprawozdaniu części oryginalnego kodu, lecz własne elementy muszą być wyraźnie wyróżnione, najlepiej poprzez wytłuszczenie lub zmianę koloru.

Poniżej prezentujemy dwa skrajnie różne przypadki kodu obliczającego silnię argumentu. Kod z lewej strony jest zwięzły, a opis dobrze wyjaśnia jego działanie. Kod z prawej strony jest nadmiernie rozbudowany, co pogarsza jego wydajność i czytelność. Komentarze są trywialne, zaś opis świadczy o tym, że piszący „widzi drzewa, lecz nie widzi lasu". Brak wcięć i użycie pochylonej czcionki o zmiennej szerokości znacznie utrudniają analizę.

DOBRZE

Funkcja oblicza silnię argumentu wywołania *n*. Na początku zmienna lokalna *silnia* jest inicjowana jedynką. Jeżeli *n* jest dodatnie, zmienna *silnia* jest mnożona przez kolejne liczby naturalne począwszy od *n* a zakończywszy na 1 (pętla w wierszach 3 i 4). W przeciwnym przypadku, tj. dla zerowego argumentu wywołania, zmienna *silnia* zachowuje wartość jednostkową.

```
1   long silnia_dobra (int n) {
2      long silnia = 1;
3      while (n > 0)
4         silnia *= n--;
5      return silnia;
6   }
```

ŹLE

Na początku deklarowany jest int, long i stos intów. Potem w pętli for kolejne liczby aż do n umieszczane są na stosie (funkcja push()). Potem zmienna s jest ustawiana na jeden. Później zmienna s jest mnożona przez wartość odczytaną ze stosu. Wartość odczytana jest zdejmowana ze stosu. To wszystko jest wykonywane, aż spełniony będzie warunek st.empty==0. W końcu wartość s jest zwracana.

```
long silnia_niedobra (int n)
{
int i;  // Zmienna tymczasowa
long s;          // Zmienna tymczasowa
stack<int> st;  // Stos
for (i = 1; i <= n; i++)
{
st.push(i);      // Wloz i na stos
}
s = 1;           // Nadaj zmiennej s wartosc „jeden”
while (st.empty() == 0)
{
s = s * st.top(); // Pomnoz zmienna s przez wierzcholek stosu
st.pop();                    // Zdejmij ze stosu wierzcholek
}
return s;                    // Zwroc zmienna s
}
```

Wnioski

Celem ćwiczeń jest wykształcenie umiejętności analizowania algorytmów, zadawania sobie nietrywialnych pytań i odpowiadania na nie. Dlatego koniecznym – a może najważniejszym – elementem sprawozdania są spostrzeżenia i wnioski. Podobnie jak przy teoretycznej analizie złożoności algorytmów, także i tu **często popełnianym błędem jest usilna próba dopasowania wniosków do teorii przy całkowitym zignorowaniu wyników eksperymentu, które tej teorii ewidentnie przeczą. Dzieje się tak, gdy „teoria” nie wynika z analizy wykonywanego kodu, lecz jest przepisana z internetu i dotyczy algorytmu, który z ćwiczeniem ma niewiele wspólnego poza nazwą.** Na drugim biegunie znajdują się wnioski tak oczywiste, że nic nie wnoszące. Bardzo częste są spostrzeżenia typu *Algorytm A jest szybszy od B, co wynika z jego natury/konstrukcji/implementacji/użytych struktur danych.* Takie zdanie bardzo dobrze nadaje się na wprowadzenie – problem w tym, że często jest to jedyny wniosek na temat porównywanych algorytmów.

Dominik Kasprowicz
marzec 2019