

26

Testowanie

„Ja tylko udowodniłem, że kod jest poprawny,
ale go nie przetestowałem.”

— Donald Knuth

W tym rozdziale opiszemy techniki testowania programów i ich projektowania w taki sposób, aby zawierały jak najmniej błędów. Jest to bardzo obszerna tematyka, dlatego my opiszemy tylko podstawy. Chcemy przede wszystkim pokazać kilka praktycznych technik i podejść do jednostek testowych programu, takich jak funkcje i klasy. Opiszemy stosowanie interfejsów oraz sposoby dobierania dla nich testów. Podkreślimy, jak ważne jest projektowanie systemów w taki sposób, aby ułatwiać testowanie już na najwcześniejszych etapach powstawania programu. Napiszemy też kilka słów o udowadnianiu poprawności programów i rozwiązywaniu problemów z wydajnością.

26.1. Czego chcemy

26.1.1. Zastrzeżenie

26.2. Dowody

26.3. Testowanie

26.3.1. Testowanie regresyjne

26.3.2. Testowanie jednostkowe

26.3.3. Algorytmy i nie-algorytmy

26.3.4. Testy systemowe

26.3.5. Znajdowanie założeń, które się nie potwierdzają

26.4. Projektowanie pod kątem testowania

26.5. Debugowanie

26.6. Wydajność

26.6.1. Kontrolowanie czasu

26.7. Źródła

26.1. Czego chcemy

Przeprowadźmy proste doświadczenie. Napisz funkcję wyszukiwania binarnego. Zrób to teraz. Nie czekaj na zakończenie rozdziału. Nie czekaj, aż przeczytasz ten podrozdział. Ważne jest, abyś spróbował to zrobić teraz. Teraz! Wyszukiwanie binarne w posortowanej sekwencji zaczyna się od środka:

- Jeśli środkowy element sekwencji jest równy z poszukiwanym, szukanie zakończone.
- Jeśli środkowy element jest mniejszy od szukanego, należy przeszukać binarnie prawą połowę sekwencji.
- Jeśli środkowy element jest większy od szukanego, należy przeszukać binarnie lewą połowę sekwencji.
- Wynikiem jest informacja, czy znaleziono szukany element, oraz coś, co pozwala go zmodyfikować, jeśli zostanie znaleziony, np. indeks, wskaźnik lub iterator.

Niech kryterium porównywania (sortowania) będzie operator mniejszości ($<$). Możesz użyć dowolnej struktury danych oraz zwrócić wynik, jak chcesz, ale musisz cały kod wyszukiwania napisać samodzielnie. W tym nietypowym przypadku skorzystanie z funkcji napisanej przez kogoś innego jest bezproduktywne, nawet jeśli odpowiednio się o tym poinformuje. W szczególności nie można użyć algorytmów z biblioteki standardowej (`binary_search` lub `equal_range`), które w większości przypadków stanowią pierwszą myśl programisty. Poświęć na to tyle czasu, ile chcesz.

Zakładamy, że masz już swoją funkcję wyszukiwania binarnego. Jeśli nie, wróć do poprzedniego akapitu. Skąd wiesz, że Twoja funkcja jest poprawna? Jeśli jeszcze tego nie zrobiłeś, napisz, dlaczego uważasz, że Twoja funkcja jest poprawna. W jakim stopniu jesteś pewien swojej argumentacji? Czy można w niej znaleźć słabe punkty?



To był bardzo prosty kod służący do zaimplementowania zwykłego i dobrze znanego algorytmu. Twój kompilator składa się z liczby rzędu 200 tysięcy wierszy kodu, system operacyjny z około 10 do 50 milionów wierszy kodu, a kod, w którym krytyczne znaczenie ma bezpieczeństwo w samolocie, którym polecisz na wakacje lub konferencję, składa się z około 500 tysięcy do 2 milionów wierszy kodu. Czy te informacje nie sprawiają, że czujesz się gorzej? Jak techniki, których użyłeś do zaimplementowania swojej funkcji wyszukiwania binarnego, mają się do rozmiarów realnych programów?

Co ciekawe, mimo tej skali złożoności większość oprogramowania działa bezawaryjnie przez większość czasu. Nie zaliczamy wszystkiego, co znajduje się w zawałonym grami komputerze PC do programów o „krytycznym” znaczeniu. Co ważniejsze, oprogramowanie o krytycznym znaczeniu dla bezpieczeństwa działa poprawnie praktycznie cały czas. Nie przypominamy sobie, aby w ciągu ostatnich dziesięciu lat miała miejsce jakaś katastrofa lotnicza lub drogowa spowodowana błędem oprogramowania. Opowieści o oprogramowaniu bankowym mającym poważne problemy z przetwarzaniem czeków na sumę 0,00 zł mają już długą brodę. Takie rzeczy zasadniczo już się nie zdarzają. A jednak oprogramowanie piszą ludzie tacy jak Ty. Wiesz, że robisz błędy. Wszyscy je robimy. Jak w takim razie oni to robią bezbłędnie?



Odpowiedź jest taka, że „my” nauczyliśmy się budować niezawodne systemy z zawodnych części. Bardzo się staramy, aby każdy program, każda klasa i każda funkcja były poprawne, ale zwykle za pierwszym razem się nam to nie udaje. Później wyszukujemy błędy, testujemy

i modyfikujemy projekt, aby zlokalizować i usunąć jak najwięcej błędów. Jednak w każdym niebanalnym systemie jakieś błędy pozostaną. Wiemy o tym, ale mimo to nie potrafimy ich znaleźć — a raczej nie potrafimy ich znaleźć w czasie i przy wysiłku, które jesteśmy gotowi poświęcić. Później jeszcze raz zmieniamy projekt systemu, aby odzyskiwał sprawność po wystąpieniu niespodzianych i „niemożliwych” zdarzeń. W wyniku tego powstają czasami systemy o wielkiej niezawodności. Należy zauważyć, że systemy te nadal mogą zawierać błędy — i zazwyczaj zawierają — i od czasu do czasu działać gorzej, niż byśmy sobie życzyli. Jednak nie ulegają poważnym awariom i zawsze świadczą minimalny zestaw usług. Na przykład system telefoniczny może przy bardzo dużym obciążeniu nie obsłużyć wszystkich rozmów, ale zawsze będzie w stanie obsłużyć ich dużą liczbę.

Teraz możemy zacząć rozważania filozoficzne na temat, czy niespodziewany błąd, który wykryliśmy, jest naprawdę błędem, ale nie zrobimy tego. Osoby budujące systemy odnoszą znacznie większe korzyści, jeśli zajmą się „tylko” myśleniem, co zrobić, aby poprawić niezawodność systemów.

26.1.1. Zastrzeżenie

Testowanie to temat rzeka. Istnieje kilka szkół, jak należy testować, a w różnych dziedzinach wypracowano różne zwyczaje i standardy dotyczące testowania. To naturalne — inne są standardy niezawodności dla gier komputerowych, a inne dla oprogramowania samolotów — ale prowadzi do nieporozumień terminologicznych i dotyczących stosowania narzędzi. Treść tego rozdziału należy traktować jako źródło pomysłów do testowania własnych projektów i ideałów, jeśli przyjdzie testować jakiś duży system. Testowanie dużych systemów wymaga stosowania rozmaitych narzędzi i struktur organizacyjnych, których opis w tym rozdziale mijałby się z celem.

26.2. Dowody

Chwileczkę! Dlaczego nie możemy po prostu udowodnić poprawności naszych programów zamiast bawić się w to całe testowanie? Jak zwykł mawiać Edsger Dijkstra: „Drogą testowania można tylko odkryć obecność błędów, ale nie ich brak”. To oczywiście powoduje, że kusząca jest możliwość dowodzenia poprawności programów w taki sam sposób, jak matematycy dowodzą poprawności swoich twierdzeń.

Niestety dowodzenie poprawności niebanalnych programów to więcej niż sztuka (wykracza poza bardzo ograniczone ramy dziedzin). Dowody same mogą zawierać błędy (tak jak te opracowywane przez matematyków). W ogóle cała dziedzina dowodzenia poprawności programów jest zaawansowanym zagadnieniem. Dlatego staramy się, jak możemy, aby tak opracować strukturę programów, by można było o nich rozumować i przekonywać samych siebie, że są poprawne. Niemniej jednak przeprowadzamy też testy (podrozdział 26.3) i staramy się tak organizować kod, aby był odporny na pozostające w nim błędy (podrozdział 26.4).

26.3. Testowanie

W podrozdziale 5.11 zdefiniowaliśmy testowanie jako „systematyczne wyszukiwanie błędów”. Przeglądamy techniki, które do tego służą.





Wyróżnia się **testowanie jednostkowe** (ang. *unit testing*) i **systemowe** (ang. *system testing*). Jednostka to jakaś część programu, np. klasa lub funkcja. Jeśli takie jednostki testuje się osobno, wiadomo, gdzie szukać źródła problemu, gdy wystąpi błąd — błąd ten musi być w testowanej jednostce (lub kodzie, za pomocą którego przeprowadzany jest test). Natomiast testowanie systemowe polega na testowaniu całego systemu i wówczas wiadomo tylko, że błąd jest gdzieś w systemie. Zwykle błędy wykrywane w czasie testów systemowych — jeśli dobrze zostały przeprowadzone testy jednostkowe — są spowodowane nieodpowiednimi interakcjami między jednostkami. Trudniej je znaleźć niż błędy w poszczególnych jednostkach i zwykle ich naprawa jest kosztowniejsza.

Oczywiście jednostki (niech będą klasy) mogą składać się z innych jednostek (np. funkcji i innych klas), a systemy (np. handlu elektronicznego) mogą składać się z innych systemów (np. baz danych, GUI, systemu sieciowego i innych). Dlatego granica między testowaniem jednostkowym a systemowym nie jest tak ostra, jak może się wydawać. Ogólna zasada jest taka, że dzięki dobremu przetestowaniu jednostek w programie zaoszczędzamy sobie pracy, a użytkownikom nerwów.

Jednym ze sposobów patrzenia na testowanie jest przyjęcie faktu, że każdy niebanalny system składa się z jednostek, które też składają się z mniejszych jednostek. Zaczyna się od testowania tych najmniejszych składników, później przechodzi się do większych, które są z nich złożone, i tak do przetestowania całego systemu. To znaczy, system jest tylko największą jednostką (dopóki nie zostanie użyty jako jednostka jeszcze większego systemu).

Najpierw przejrzymy techniki testowania jednostek (takich jak funkcje, klasy, hierarchie klas czy szablony). Testerzy wyróżniają testowanie białej skrzynki (ang. *white-box testing* — można obejrzeć szczegóły implementacji testowanej jednostki) i testowanie czarnej skrzynki (ang. *black-box testing* — znany jest tylko interfejs testowanej jednostki). Nie będziemy zwracać zbyt dużej uwagi na to rozróżnienie. Za wszelką cenę staraj się przeczytać dokumentację tego, co testujesz. Pamiętaj jednak, że implementacja może zostać zmieniona, dlatego nie należy uzależniać się od niczego, co nie jest częścią interfejsu. Zasadniczo testowanie cegółkolwiek polega na rzuceniu interfejsowi, co tylko się da, i sprawdzeniu, czy reaguje w odpowiedni sposób.



Jeśli ktoś (może Ty sam) zmieni kod po przetestowaniu, trzeba przeprowadzić testowanie regresyjne (ang. *regression testing*). W zasadzie po każdej zmianie należy ponownie przeprowadzić testy, aby sprawdzić, czy nic się nie zepsuło. Jeśli więc poprawiło się coś w jednostce, trzeba ją ponownie przetestować, a przed przekazaniem systemu komuś innemu (lub przed realnym użyciem go we własnym zakresie) należy przeprowadzić pełne testy systemowe.

Przeprowadzanie pełnych testów systemu często nazywa się **testowaniem regresyjnym**, ponieważ zwykle polega ono na przeprowadzeniu testów, które wcześniej pozwoliły wykryć jakieś błędy, aby sprawdzić, czy już nie występują. Jeśli tak, oznacza to, że program „się cofnął” i trzeba go znowu naprawić.

26.3.1. Testowanie regresyjne



Efektywną metodą tworzenia dobrego zestawu testów dla systemu jest zgromadzenie pokaźnej kolekcji testów, które pozwoliły znaleźć błędy w przeszłości. Wyobraź sobie, że z Twojego systemu korzystają użytkownicy, którzy przysyłają Ci informacje o błędach. Nigdy nie wyrzucaj informacji o błędzie! Zawodowcy pilnują tego za pomocą systemów śledzenia błędów. Niemniej informacja o błędzie oznacza albo rzeczywisty błąd w systemie, albo błędne zrozumienie systemu przez użytkownika. Oba rodzaje informacji są przydatne.

Zwykle raport o błędzie zawiera wiele niepotrzebnych informacji. Dlatego najpierw trzeba utworzyć najmniejszy możliwy program, który ujawnia zgłoszony problem. To często oznacza konieczność usunięcia większości przysłanego kodu. W szczególności trzeba spróbować wyrzucić kod bibliotek i aplikacji, który nie ma wpływu na błąd. Znalezienie takiego minimalnego testowego programu często pomaga zlokalizować błąd w kodzie systemu. Ten minimalny program dodaje się do zestawu testów regresyjnych. Aby znaleźć taki minimalny program, należy usuwać po kawałku kod, aż błąd przestanie występować, i przywrócić ostatnio usunięty kawałek. Ta metoda jest skuteczna, jeśli nie wyczerpie się lista kandydatów do usunięcia.

Przeprowadzanie setek (albo dziesiątków tysięcy) testów utworzonych na bazie starych raportów o błędach nie wydaje się zbyt systematycznym podejściem, ale w rzeczywistości jest to systematyczne wykorzystywanie doświadczenia użytkowników i programistów. Zestaw testów regresyjnych stanowi największą część zbiorowej pamięci grupy programistów. W przypadku dużych systemów nie można oczekiwać, że pierwsi programiści będą w stanie objaśnić wszystkie szczegóły dotyczące projektu i implementacji. Zestaw testów regresyjnych zapobiega odaleniu się systemu od tego, co programiści i użytkownicy uznali za poprawne działanie.

26.3.2. Testowanie jednostkowe

Wystarczy już tego gadania! Wypróbujemy konkretny przykład. Będzie to wyszukiwanie binarne. Poniżej znajduje się specyfikacja ze standardu ISO (punkt 25.3.3.4):

```
template<class ForwardIterator, class T>  
bool binary_search(ForwardIterator first, ForwardIterator last,  
const T& value);
```

```
template<class ForwardIterator, class T, class Compare>  
bool binary_search(ForwardIterator first, ForwardIterator last,  
const T& value, Compare comp);
```

Wymagania: elementy e w przedziale $[first, last)$ są dzielone według wyrażeń $e < value$ i $!(value < e)$ lub $comp(e, value)$ i $!comp(value, e)$. Ponadto dla wszystkich elementów e przedziału $[first, last)$ $e < value$ implikuje $!(value < e)$ lub $comp(e, value)$ implikuje $!comp(value, e)$.

Zwraca: wartość `true`, jeśli w przedziale $[first, last)$ istnieje iterator i , który spełnia odpowiednie warunki: $!(i < value) \ \&\& \ !(value < i)$ lub $comp(*i, value) == false \ \&\& \ comp(value, *i) == false$.

Złożoność: najwyżej $\log(last - first) + 2$ porównań.

Nikt nie twierdził, że czytanie formalnej specyfikacji (może półformalnej) jest łatwe dla niewprawnego oka. Niemniej jeśli naprawdę wykonałeś zalecane przez nas na początku zadanie i napisałeś funkcję wyszukiwania binarnego, dobrze wiesz, na czym to polega i jak to przetłumaczyć. Ta standardowa wersja pobiera jako argumenty parę iteratorów przechodzących do przodu (punkt 20.10.1) oraz wartość i zwraca `true`, jeśli wartość ta należy do zdefiniowanego przez iteratory przedziału. Iteratory muszą definiować posortowaną sekwencję. Kryterium sortowania jest $<$. Drugą wersję funkcji `binary_search()`, która kryterium porównywania pobiera jako argument, pozostawiamy jako pracę domową.

Będziemy zajmować się tylko takimi błędami, których nie potrafi znaleźć kompilator, a więc taki kod, jak na następnej stronie, jest problemem osób, które go piszą:

```
binary_search(1,4,5); // Błąd: liczba typu int nie jest iteratorem przechodzącym do przodu
vector<int> v(10);
binary_search(v.begin(),v.end(),"7"); // Błąd: nie można szukać łańcucha
// w wektorze liczb całkowitych
binary_search(v.begin(),v.end()); // Błąd: brak wartości
```



Jak przeprowadzić systematyczne testowanie funkcji `binary_search()`? Oczywiście nie da się wypróbować jej na każdym możliwym argumentcie, ponieważ trzeba by było przekazać jej każdą możliwą sekwencję każdego możliwego typu danych — mogłaby powstać nieskończona liczba testów! Dlatego należy dokonać selekcji testów na podstawie jakichś ogólnych zasad:

- Testowanie w celu znalezienia **prawdopodobnych błędów** (pozwala znaleźć najwięcej błędów).
- Testowanie w celu znalezienia **poważnych błędów** (pozwala znaleźć potencjalnie najbardziej szkodliwe błędy).

Pisząc „poważne błędy”, mamy na myśli usterki wywołujące najgorsze konsekwencje. Znaczenie tego słowa jest trochę nieprecyzyjne, ale można je doprecyzować dla każdego konkretnego programu. Jeśli na przykład weźmie się pod uwagę samo wyszukiwanie binarne, wszystkie błędy będą równie poważne. Jeśli jednak takie wyszukiwanie zostanie zastosowane w programie, w którym wszystkie wyniki są bardzo skrupulatnie sprawdzane, zwrócenie niepoprawnej odpowiedzi przez funkcję `binary_search()` byłoby znacznie mniej poważne niż niezwrócenie wartości w ogóle z powodu wpadnięcia w nieskończoną pętlę. W takim przypadku znacznie więcej czasu poświęcilibyśmy na oszukanie funkcji `binary_search()`, aby wpadła w nieskończoną (lub bardzo długą) pętlę niż na spowodowanie zwrócenia przez nią nieprawidłowej wartości. Zwróć uwagę na użyte przez nas słowo „oszukać”. Testowanie to między innymi ćwiczenie kreatywnego myślenia na temat: „jak zmusić ten kod do niepoprawnego działania”. Dobrzy testerzy są nie tylko systematyczni, lecz również są dobrymi krętaczami (oczywiście w pozytywnym tego słowa znaczeniu).

26.3.2.1. Strategia testowania

Od czego zacząć próby złamania funkcji `binary_search()`? Należy zacząć od sprawdzenia jej wymagań, a więc co pobiera na wejściu. Niestety, z punktu widzenia testera jest jasno napisane, że funkcja ta pobiera posortowaną sekwencję `[first,last)`. Oznacza to, że obowiązkiem wywołującego jest zapewnić, że takie będą właśnie przekazywane jej dane. Testujący nie może próbować jej złamać, przekazując nieposortowaną sekwencję lub taką, w której `last < first`. Należy zauważyć, że w wymaganiach funkcji `binary_search()` nie napisano nic na temat tego, co robi, gdy odbierze niepoprawne dane. Gdzieś indziej w standardzie jest napisane, że może zgłosić wyjątek, ale nie jest to wymóg. Fakty te warto zapamiętać na czas testowania funkcji, ponieważ jest możliwe, że wywołujący zapomni sprawdzić te wymagania, co może być źródłem błędów.

Oto lista błędów, które mogą wystąpić w funkcji `binary_search()`:

- Nigdy nie wraca (np. nieskończona pętla).
- Awaria (np. nieprawidłowa dereferencja lub nieskończona rekurencja).
- Nie znajduje wartości, mimo że ta znajduje się w przeszukiwanej sekwencji.
- Znajduje wartość, mimo że jej nie ma w sekwencji.

Ponadto pamiętamy o następujących możliwościach popełnienia błędów przez użytkowników:

- Przekazanie nieposortowanej sekwencji (np. {2,1,5,-7,2,10}).
- Przekazanie czegoś, co nie jest prawidłową sekwencją (np. `binary_search(&a[100], ↪&a[50], 77)`).

Jaki błąd mógłby popełnić implementujący w prostym wywołaniu `binary_search(p1,p2,v)`? Błędy często występują w tzw. „specjalnych przypadkach”. Jeśli chodzi o wszelkiego rodzaju sekwencje, zawsze sprawdza się początek i koniec. Zwłaszcza należy pamiętać o sprawdzeniu przypadku pustej sekwencji. Przystudiujemy kilka poprawnie posortowanych tablic liczb całkowitych:

```
{ 1,2,3,5,8,13,21 }      // „zwykła sekwencja”
{ }                      // pusta sekwencja
{ 1 }                    // jeden element
{ 1,2,3,4 }              // parzysta liczba elementów
{ 1,2,3,4,5 }            // nieparzysta liczba elementów
{ 1, 1, 1, 1, 1, 1, 1 }  // wszystkie elementy takie same
{ 0,1,1,1,1,1,1,1,1,1,1 } // inny element na początku
{ 0,0,0,0,0,0,0,0,0,0,0,1 } // inny element na końcu
```

Niektóre sekwencje testowe najlepiej wygenerować za pomocą programu:

- `vector<int> v1;`
 `for (int i=0; i<100000000; ++i) v1.push_back(i); // bardzo długa sekwencja`
- Kilka sekwencji z losową liczbą elementów.
- Kilka sekwencji z losowymi elementami.

Nie jest to tak systematyczne podejście, jakbyśmy chcieli. W zasadzie po prostu „wybraliśmy” kilka sekwencji. Niemniej zastosowaliśmy kilka podstawowych zasad, które często warto stosować przy pracy ze zbiorami wartości. Wzięliśmy pod uwagę:

- pusty zbiór;
- małe zbiory;
- duże zbiory;
- zbiory z ekstremalnym rozkładem;
- zbiory, w których „to co interesujące” znajduje się na końcu;
- zbiory z duplikatami elementów;
- zbiory z parzystą i nieparzystą liczbą elementów;
- zbiory generowane przy użyciu liczb losowych.

Losowych sekwencji używamy, licząc, że będziemy mieli szczęście (tzn. znajdziemy błąd) i znajdziemy coś, o czym nie pomyśleliśmy. Jest to prymitywna technika, ale względnie tania, jeśli chodzi o nasz czas.

Po co nam „parzyste i nieparzyste” sekwencje? Wiele algorytmów dzieli swoje dane wejściowe na dwie połowy i liczymy na to, że może programista wziął pod uwagę tylko przypadek parzystej lub nieparzystej liczby elementów. Mówiąc bardziej ogólnie, gdy dzieli się sekwencję na połowy, punkt, w którym następuje podział, staje się ostatnim elementem podsekwencji, a my wiemy, że tam mogą występować błędy.

Ogólnie mówiąc, szukamy:

- ekstremalnych przypadków (duży, mały, nietypowy rozkład na wejściu itp.);
- warunków granicznych (wszystko, co jest bliskie limitowi).

Co to dokładnie oznacza, zależy od testowanego programu.

26.3.2.2. Prosta uprząż testowa

Wyróżnia się dwie kategorie testów — takie, które powinny zakończyć się powodzeniem (np. szukanie wartości, która jest w sekwencji), oraz takie, które powinny zakończyć się niepowodzeniem (np. szukanie wartości w pustej sekwencji). Utworzymy po kilka testów każdego z tych rodzajów dla każdej naszej sekwencji testowej. Zaczniemy od czegoś prostego i oczywistego i będziemy nanosić poprawki, aż uzyskamy coś, co będzie dobre dla naszej przykładowej funkcji `binary_srearch()`:

```
vector<int> v { 1,2,3,5,8,13,21 };
if (binary_search(v.begin(),v.end(),1) == false) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),5) == false) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),8) == false) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),21) == false) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),-7) == true) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),4) == true) cout << "niepowodzenie";
if (binary_search(v.begin(),v.end(),22) == true) cout << "niepowodzenie";
```

Jest tu dużo powtarzalnej i żmudnej pracy, ale na początek może być. W istocie wiele prostych testów to długie listy podobnych do powyższego wywołań. Zaletą tej naiwnej metody jest jej prostota. Nawet nowy członek zespołu testerów może dodać do takiego zbioru własny test. Niemniej zwykle można zrobić coś znacznie lepszego. Jeśli na przykład coś się tu nie powiedzie, nie będziemy wiedzieli, który to był test. To niedopuszczalne. Ponadto pisanie testów nie stanowi cofnięcia się do „wycinania i wklejania”. Kod testów również musi być dobrze zaprojektowany i przemyślany. Dlatego zmieniamy:

```
vector<int> v { 1,2,3,5,8,13,21 };
for (int x : {1,5,8,21,-7,2,44})
if (binary_search(v.begin(),v.end(),x) == false) cout << "niepowodzenie " << x;
```

Zakładając, że z czasem lista ta urośnie do kilkudziesięciu testów, ta poprawka ma ogromne znaczenie. W realnych systemach zwykle są tysiące testów, a więc precyzyjne określenie, który test się nie powiódł, jest konieczne.

Zanim przejdziemy dalej, warto zwrócić uwagę na jeszcze jedną metodę (półsystematycznego) testowania — do testowania użyliśmy poprawnych wartości, część wybierając z końców, a część ze środka sekwencji. W tym przypadku moglibyśmy sprawdzić wszystkie elementy,

ale zwykle jest to nierealne. W przypadku niepowodzenia znajdowania wartości wybieramy po jednej z każdego końca i jedną ze środka. Nie jest to znowu idealnie systematyczne podejście, ale zaczynamy dostrzegać jakąś regułę, którą można zawsze wykorzystać, gdy ma się do czynienia z sekwencjami wartości.

Jakie wady mają te nasze pierwsze testy:

- Początkowo wielokrotnie piszemy ten sam kod.
- Początkowo ręcznie numerujemy testy.
- Dane wyjściowe są mało pomocne.

Po chwili namysłu postanowiliśmy zapisać nasze testy jako dane w pliku. Każdy test będzie miał etykietę, wartość do znalezienia, sekwencję oraz spodziewany wynik. Na przykład:

```
{ 27 7 { 1 2 3 5 8 13 21} 0 }
```

To jest test numer 27. Szuka wartości 7 w sekwencji { 1 2 3 5 8 13 21} i spodziewana wartość zwrotna to 0 (oznaczająca false). Dlaczego dane wejściowe testów umieściliśmy w pliku zamiast bezpośrednio w tekście programu? Cóż, w tym przypadku moglibyśmy tak zrobić, ale zapisywanie dużych ilości danych w pliku z kodem źródłowym zwykle powoduje bałagan, a poza tym często wykorzystuje się programy do generowania przypadków testowych. Przypadki wygenerowane przez maszynę są zazwyczaj zapisywane w plikach. Poza tym można napisać program, który będzie testowany na wielu różnych plikach z przypadkami testowymi:

```
struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

istream& operator>>(istream& is, Test& t); // Używa opisanego wyżej formatu

int test_all(istream& is)
{
    int error_count = 0;
    for (Test t; is>>t; ) {
        bool r = binary_search( t.seq.begin(), t.seq.end(), t.val);
        if (r !=t.res) {
            cout << "Niepowodzenie: test " << t.label
                << " binary_search: "
                << t.seq.size() << " elementów, val==" << t.val
                << " -> " << t.res << '\n';
            ++error_count;
        }
    }
    return error_count;
}

int main()
```

```
{
    int errors = test_all(ifstream("moje_testy.txt"));
    cout << "Liczba błędów: " << errors << "\n";
}
```

Poniżej znajduje się trochę testów wykorzystujących wypisane wyżej sekwencje:

```
{ 1.1 1 { 1 2 3 5 8 13 21 } 1 }
{ 1.2 5 { 1 2 3 5 8 13 21 } 1 }
{ 1.3 8 { 1 2 3 5 8 13 21 } 1 }
{ 1.4 21 { 1 2 3 5 8 13 21 } 1 }
{ 1.5 -7 { 1 2 3 5 8 13 21 } 0 }
{ 1.6 4 { 1 2 3 5 8 13 21 } 0 }
{ 1.7 22 { 1 2 3 5 8 13 21 } 0 }

{ 2 1 { } 0 }

{ 3.1 1 { 1 } 1 }
{ 3.2 0 { 1 } 0 }
{ 3.3 2 { 1 } 0 }
```

Teraz wiadomo, czemu zastosowaliśmy etykiety tekstowe zamiast liczb. Dzięki temu nasz system numerowania jest bardziej elastyczny — tutaj użyliśmy systemu dziesiętnego do numerowania poszczególnych testów dla jednej sekwencji. W bardziej wyszukany sposób można by było wyeliminować konieczność powtarzania sekwencji w pliku z danymi testowymi.

26.3.2.3. Sekwencje losowe

Wybieranie wartości do wykorzystania w testach to próba przechytrzenia implementujących (którymi często sami jesteśmy). Należy szukać takich wartości, które są związane z obszarami, w których jest największa szansa na znalezienie błędów (np. skomplikowane sekwencje warunków, końce sekwencji, pętle itp.). To samo jednak robiliśmy, gdy pisaliśmy i debugowaliśmy kod. Możliwe więc jest, że popełnimy ten sam logiczny błąd, co wcześniej i kompletnie ominiemy problem. Jest to jeden z powodów, dla których warto zaangażować w testowanie kogoś innego niż programista, który napisał program. Dysponujemy jedną techniką, która pomaga rozwiązać ten problem — wygenerowanie mnóstwa losowych wartości. Poniżej znajduje się na przykład funkcja drukująca na wyjściu opis testu za pomocą funkcji `randint()` z podrödziału 24.7 i pliku `std_lib_facilities.h`:

```
void make_test(const string& lab, int n, int base, int spread)
    // Drukuj opis testu z etykietą lab na wyjściu
    // Generuj sekwencję n elementów, zaczynając od base
    // Średnia odległość między elementami jest jednostajnie rozłożona
    // w przedziale [0:spread)
{
    cout << "{ " << lab << " " << n << " { ";
    vector<int> v;
    int elem = base;
    for (int i = 0; i < n; ++i) { // Tworzy elementy
        elem += randint(spread);
        v.push_back(elem);
    }
```



```

    }

    int val = base + randint(elem-base); // Tworzy wartość do szukania
    bool found = false;
    for (int i = 0; i<n; ++i) { // Drukuje elementy i sprawdza, czy val został znaleziony
        if (v[i]==val) found = true;
        cout << v[i] << " ";
    }
    cout << "}" << found << " }\n";
}

```

Należy zauważyć, że do sprawdzenia, czy losowa wartość `val` znajduje się w losowej sekwencji, nie została użyta funkcja `binary_search()`. Nie można za pomocą testowanej funkcji określać poprawnej wartości testu.

W istocie funkcja `binary_search()` nie jest zbyt dobrym przykładem takiego prymitywnego podejścia do testowania. Wątpliwe, czy uda się w ten sposób znaleźć jakiegokolwiek błędy, których nie znajdą nasze „domowej roboty” testy, ale technika ta często bywa przydatna. Utworzymy kilka losowych testów:

```

int no_of_tests = randint(100); // Tworzy około 50 testów
for (int i = 0; i<no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab+to_string(i), // to_string z podrozdziału 23.2
        randint(500),           // liczba elementów
        0,                      // podstawa
        randint(50));           // rozproszenie
}

```

Generowanie testów z liczb losowych jest szczególnie użyteczną techniką przy testowaniu zbiorczych efektów operacji, których wynik zależy od tego, jaki był wynik poprzednich operacji, tzn. gdy system ma określony stan (podrozdział 5.2).

Powodem, dla którego liczby losowe nie są tak przydatne w testowaniu funkcji `binary_search()`, jest to, że każde przeszukiwanie sekwencji jest niezależne od innych jej przeszukiwań. Oczywiście zakładamy, że funkcja `binary_search()` nie zrobi nic kompletnie głupiego, jak zmodyfikowanie przeszukiwanej przez siebie sekwencji. Do tego potrzebujemy lepszego testu (5. zadanie pracy domowej).

26.3.3. Algorytmy i niealgorytmy

Funkcji `binary_search()` użyliśmy jako przykładu. Jest to poprawny algorytm, który:

- Ma dobrze określone wymagania dotyczące danych wejściowych.
- Ma dobrze określony efekt, jaki wywołuje na danych wejściowych (w tym przypadku brak efektu).
- Nie jest zależny od obiektów niebędących jego jawnymi danymi wejściowymi.
- Nie ma poważnych ograniczeń nałożonych przez środowisko (np. nie musi zmieścić się w określonym czasie lub określonej przestrzeni czy współdzielić zasobów).




Ma oczywiste i wyraźnie zdefiniowane warunki wstępne i końcowe (podrozdział 5.10). Innymi słowy, jest to marzenie każdego testera. Często nie ma się tyle szczęścia i trzeba testować niechlujny kod, który w najlepszym przypadku jest opisany trochę dziwnym angielskim językiem i kilkoma schematami.

Chwileczkę! Czy my nie pozwalamy sobie tutaj na powierzchowną logikę? Jak możemy mówić o poprawności i testowaniu, gdy nie wiemy dokładnie, co ma robić testowany kod? Problem polega na tym, że wiele rzeczy, które muszą zostać zrobione w oprogramowaniu, trudno opisać przy użyciu czysto matematycznych narzędzi. Poza tym w wielu przypadkach, w których teoretycznie da się to zrobić, taka matematyka jest poza umiejętnościami programistów piszących i testujących kod. Zostajemy więc z ideałem perfekcyjnie precyzyjnych specyfikacji i rzeczywistością dotyczącą tego, co ktoś może zrobić w określonym czasie i dysponując określonymi umiejętnościami.


Żałujemy, że mamy do przetestowania niechlujnie napisaną funkcję. Piszac „niechlujnie napisaną”, mamy na myśli:

- *Dane wejściowe* — wymagania dotyczące (jawnych lub niejawnych) danych wejściowych nie są tak dobrze opisane, jakbyśmy chcieli.
- *Dane wyjściowe* — (jawne lub niejawne) dane wyjściowe nie są tak dobrze opisane, jakbyśmy chcieli.
- *Zasoby* — sposoby korzystania przez nią z zasobów (czas, pamięć, pliki itp.) nie są tak dobrze opisane, jakbyśmy chcieli.

Słowa „jawne” i „niejawne” oznaczają, że nie wystarczy tylko sprawdzić formalnych parametrów i wartości zwrotnej, lecz także wiele skutków działania na zmienne globalne, strumienie wejścia i wyjścia, alokację w pamięci wolnej itp. Co można w takim razie zrobić? Po pierwsze taka funkcja prawie na pewno jest za długa — albo dałoby się lepiej określić jej wymagania i efekty. Może mamy do czynienia z funkcją zajmującą pięć stron lub wykorzystującą na skomplikowane i nieoczywiste sposoby funkcje pomocnicze. Może się wydawać, że pięć stron to bardzo dużo jak na funkcję. To prawda, ale widzieliśmy funkcje, które były jeszcze dużo dłuższe. Niestety nie należą one do rzadkości.



Gdyby to był nasz kod i gdybyśmy mieli czas, moglibyśmy spróbować rozbić taką długą funkcję na mniejsze i bliższe naszym ideałom i najpierw przetestować te mniejsze jednostki. Jednak teraz naszym celem jest testowanie oprogramowania, a więc systematyczne znajdowanie jak największej liczby błędów, a nie tylko poprawianie błędów, gdy uda się je znaleźć.



Czego w takim razie szukamy? Jesteśmy testerami, a więc naszym zadaniem jest znajdowanie błędów. Gdzie mogą ukrywać się błędy? Jakie cechy ma kod, który może zawierać błędy?

- Niewyraźne zależności od innego kodu — szukaj użycia zmiennych globalnych, argumentów w postaci niestałych referencji, wskaźników itp.
- Zarządzanie zasobami — szukaj instrukcji zarządzania pamięcią (operatory `new` i `delete`), używających plików, blokad itp.
- Pętle — sprawdź warunki końcowe (jak w funkcji `binary_search()`).
- Instrukcje `if` i przełączniki (często nazywane rozgałęzieniami) — szukaj błędów w ich logice.

Poszukajmy przykładów każdego z nich.

26.3.3.1. Zależności

Rozważmy poniższą bezsensowną funkcję:

```
int do_dependent(int a, int& b) // marna funkcja
    // niedyscyplinowane zależności
{
    int val;
    cin >> val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

Aby przetestować powyższą funkcję `do_dependent()`, nie można po prostu zgromadzić zbioru argumentów i sprawdzić, co z nimi zrobi. Trzeba wziąć pod uwagę, że używa zmiennych globalnych `cin`, `cout` i `vec`. W tej małej funkcji to jest oczywiste, ale w realnym kodzie to wszystko może być ukryte pod górą kodu. Na szczęście istnieje oprogramowanie, które może nam pomóc znaleźć takie zależności. Niestety nie zawsze łatwo je dostać. Jeśli nie ma się dostępu do oprogramowania analizującego, trzeba samodzielnie przejrzeć kod funkcji wiersz po wierszu i wypisać wszystkie zależności.

Przy testowaniu funkcji `do_dependent()` musimy wziąć pod uwagę:

- Wejście:
 - wartość parametru `a`;
 - wartość parametru `b` i wartość liczby typu `int`, którą wskazuje;
 - dane ze strumienia `cin` (zapisywane w zmiennej `val`) oraz jego stan;
 - stan strumienia `cout`;
 - wartość wektora `vec`, zwłaszcza wartość `vec[val]`.
- Wyjście:
 - wartość zwrótna;
 - wartość liczby typu `int`, którą wskazuje parametr `b` (zwiększyliśmy ją);
 - stan strumienia `cin` (uwaga na stan strumienia i stan formatu);
 - stan strumienia `cout` (uwaga na stan strumienia i stan formatu);
 - stan wektora `vec` (było przypisanie do `vec[val]`);
 - wszelkie wyjątki, które może zgłosić `vec` (element `vec[val]` może być poza zakresem).

To długa lista. W istocie jest dłuższa niż sama funkcja. To dobrze wyjaśnia naszą niechęć do zmiennych globalnych i nasze zastrzeżenia do stosowania niestałych referencji (i wskaźników). Jest coś naprawdę przyjemnego w funkcji, która po prostu wczytuje swoje argumenty i wytwarza wynik jako wartość zwrótną — łatwo ją zrozumieć i przetestować.



Po zidentyfikowaniu wejść i wyjść zasadniczo wracamy do przypadku funkcji `binary_search()`. Generujemy testy z wartościami wejściowymi (jawnymi i niejawnymi) i sprawdzamy, czy wyniki są poprawne (zarówno jawne, jak i niejawne). Testowanie funkcji `do_dependent()` zaczęlibyśmy pewnie od bardzo dużej i ujemnej wartości `val`. Wygląda na to, że lepiej by było, gdyby wektor `vec` miał sprawdzanie zakresu (albo będzie łatwo wywołać bardzo proste błędy). Oczywiście moglibyśmy sprawdzić, co jest na temat tych wejść i wyjść napisane w dokumentacji. Jednak biorąc pod uwagę, że funkcja została napisana tak niedbale, trudno oczekiwać, że jej specyfikacja będzie precyzyjna i kompletna. Dlatego zaczniemy od łamania funkcji (tzn. znajdowania błędów) i pytania, co jest poprawne. Takie zadawanie pytań i testowanie często prowadzi do przeprojektowania funkcji.

26.3.3.2. Zarządzanie zasobami

Rozważmy poniższą bzdurną funkcję:

```
void do_resources1(int a, int b, const char* s) // marna funkcja
// niezdyscyplinowane wykorzystanie zasobów
{
    FILE* f = fopen(s, "r"); // Otwiera plik (styl języka C)
    int* p = new int[a];      // Alokuje trochę pamięci
    if (b <= 0) throw Bad_arg(); // Może zgłosić wyjątek
    int* q = new int[b];      // Alokuje trochę więcej pamięci
    delete[] p;              // Dealokuje pamięć wskazywaną przez p
}
```


Aby przetestować funkcję `do_resources1()`, musimy pomyśleć, czy wszystkie pobrane zasoby zostały poprawnie rozdysponowane, a więc czy każdy zasób został zwolniony lub przekazany innej funkcji.

Tutaj jest oczywiste, że:

- Plik o nazwie `s` nie zostaje zamknięty.
- Pamięć zarezerwowana dla `p` wycieknie, jeśli `b <= 0` lub druga instrukcja `new` zgłosi wyjątek.
- Pamięć zarezerwowana dla `q` wycieknie, jeśli `0 < b`.

Ponadto zawsze trzeba wziąć pod uwagę możliwość, że nie uda się otworzyć pliku. Aby uzyskać ten marny wynik, celowo zastosowaliśmy przestarzały styl programowania (funkcja `fopen()` jest standardową metodą otwierania plików w języku C). Znacznie ułatwilibyśmy testerom pracę, gdybyśmy napisali taki kod:

```
void do_resources2(int a, int b, const char* s) // trochę lepsza funkcja
{
    ifstream is(s); // Otwiera plik
    vector<int> v1(a); // Tworzy wektor (posiadający pamięć)
    if (b <= 0) throw Bad_arg(); // Może zgłosić wyjątek
    vector<int> v2(b); // Tworzy inny wektor (posiadający pamięć)
}
```

 Teraz każdy zasób jest w posiadaniu obiektu mającego destruktor, który go zwolni. Czasami zastanawianie się nad tym, jak bardziej przejrzyste napisać funkcję, może pomóc w wymyśleniu

dobrych testów. Opisana w punkcie 19.5.2 technika RAII (ang. *Resource Acquisition Is Initialization*) proponuje ogólną strategię postępowania z takim problemem z zarządzaniem zasobami.

Należy zauważyć, że zarządzanie zasobami nie polega tylko na sprawdzaniu, czy każdy fragment alokowanej pamięci zostaje zwolniony. Czasami pobiera się zasoby z innych miejsc (np. jako argumenty), a czasami przekazuje się je poza funkcję (np. jako wartość zwrótną). Czasami trudno określić, co jest w takich przypadkach właściwe. Rozważmy:



```
FILE* do_resources3(int a, int* p, const char* s) // marna funkcja
// niezdypliniowane wykorzystanie zasobów
{
    FILE* f = fopen(s, "r");
    delete p;
    delete var;
    var = new int[27];
    return f;
}
```

Czy dobrze, że funkcja `do_resources3()` przekazuje (rzekomo) otwarty plik jako wartość zwrótną? Czy powinna usuwać pamięć przekazaną do niej jako argument `p`? Dodaliśmy jeszcze podstępne użycie globalnej zmiennej `var` (oczywiście wskaźnik). Zasadniczo przekazywanie zasobów z i do funkcji jest często stosowaną i przydatną techniką, ale aby stwierdzić, czy jest właściwie stosowana, trzeba znać strategię zarządzania zasobami. Kto jest właścicielem danego zasobu? Kto powinien go usunąć lub zwolnić? Jasne i proste odpowiedzi na te pytania powinny znajdować się w dokumentacji (możemy sobie pomarzyć). Bez względu na wszystko przekazywanie zasobów jest bogatym źródłem błędów i kuszącym obszarem do testowania.

Zwróć uwagę jak (celowo) skomplikowaliśmy zarządzanie zasobami poprzez użycie globalnej zmiennej. Sprawa może się skomplikować, gdy zacznie się mieszać takie źródła potencjalnych błędów. Jako programiści staramy się ich unikać. Jako testerzy szukamy ich jako przykładów łatwych punktów zaczepienia.



26.3.3.3. Pętle

Była mowa o pętlach przy omawianiu funkcji `binary_search()`. Zasadniczo większość błędów występuje na końcach:

- Czy wszystko jest odpowiednio zainicjowane na początku pętli?
- Czy poprawnie kończymy pracę z ostatnim przypadkiem (często ostatnim elementem)?



Oto przykład, jaki błąd można popełnić:

```
int do_loop(const vector<int>& v) // marna funkcja
// niezdypliniowana petla
{
    int i;
    int sum;
    while(i<=vec.size()) sum+=v[i];
    return sum;
}
```

W powyższym kodzie znajdują się trzy oczywiste błędy. Potrafisz je wskazać? Ponadto dobry tester od razu zauważy możliwość wystąpienia przepełnienia przy dodawaniu do wartości `sum`:



- Wiele pętli operuje na danych i może spowodować przepełnienie, jeśli dostanie duże wartości na wejściu.

Słynny i wredny błąd związany z pętlami — przepełnienie bufora — należy do kategorii błędów, które można wyłapywać poprzez systematyczne zadawanie dwóch podstawowych pytań dotyczących pętli:

```
char buf[MAX]; // bufor o stałym rozmiarze

char* read_line() // niebezpieczne niedbalstwo
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch!='\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}
```

Oczywiście **Ty** nie napisałbyś czegoś takiego! Czemu nie, co jest takiego złego w tej funkcji `read_line()`? Jest to jednak bardzo rozpowszechniony rodzaj kodu, który występuje na dodatek w wielu wersjach, jak poniższa:

```
// niebezpieczne niedbalstwo:
gets(buf); // Wczytuje wiersz danych do buf
scanf("%s", buf); // Wczytuje wiersz danych do buf
```



Poszukaj w dokumentacji informacji o funkcjach `gets()` i `scanf()` i unikaj ich jak ognia. Pisząc „niebezpieczne”, mamy na myśli, że takie przepełnienia są okazją do włamania do komputera. Wiele implementacji przestrzega przed stosowaniem funkcji `gets()` i jej podobnych z tych właśnie powodów.

26.3.3.4. Rozgałęzianie

Oczywiście, gdy musimy dokonać wyboru, istnieje niebezpieczeństwo, że dokonamy złego wyboru. Z tego powodu dobrym celem dla testera są instrukcje `if` i `switch`. Są dwa najważniejsze rodzaje problemów, których można poszukać:



- Czy wszystkie możliwości zostały przewidziane?
- Czy z poszczególnymi możliwościami powiązано odpowiednie działania?

Rozważmy poniższą bzdurną funkcję:

```
void do_branch1(int x, int y) // marna funkcja
// niedyscyplinowane stosowanie instrukcji if
{
    if (x<0) {
        if (y<0)
            cout << "Bardzo ujemne.\n";
    }
}
```



```

        else
            cout << "Coś ujemne.\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "Bardzo dodatnie.\n";
        else
            cout << "Coś dodatnie.\n";
    }
}

```

Oczywistym błędem w tym przypadku jest sytuacja, gdy x ma wartość 0. Przy przeprowadzaniu testów dotyczących zera (lub wartości dodatnich i ujemnych) często wykrywa się, że o nim zapomniano lub zostało ono zaliczone do niewłaściwego przypadku (np. uznane za wartość ujemną). Jest tu jeszcze jeden, mniej wyrazisty błąd — działania dla przypadków $(x>0 \ \&\& \ y<0)$ i $(x>0 \ \&\& \ y \geq 0)$ „jakoś się poprzestawiały”. To często się zdarza przy kopiowaniu i wklejaniu.

Im bardziej skomplikowane są instrukcje warunkowe, tym większa szansa na wystąpienie błędu. Jako testerzy przeglądaliśmy taki kod i robimy wszystko, aby każde rozgałęzienie zostało przetestowane. Oczwistym zestawem testów dla funkcji `do_branch1()` jest poniższy:

```

do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);

```

Zasadniczo jest to prymitywna metoda typu „wypróbuj wszystkie możliwości”. Jej zastosowanie pozwoliło wykryć, że funkcja `do_branch1()` porównuje wartości z 0 tylko za pomocą operatorów `<` i `>`. Aby znaleźć nieprawidłowe działania dla wartości dodatnich x , musimy połączyć wywołania z ich pożądanym wynikiem.

Postępowanie z instrukcjami `switch` zasadniczo nie różni się od instrukcji `if`.

```

void do_branch1(int x, int y) // marna funkcja
// niezdyktynowane używanie instrukcji switch
{
    if (y<0 && y<=3)
        switch (x) {
            case 1:
                cout << "jeden\n";
                break;
            case 2:
                cout << "dwa\n";
            case 3:
                cout << "trzy\n";
        }
}

```

W powyższym kodzie popełnieliśmy cztery klasyczne błędy:

- Sprawdziliśmy zakres nie tej co trzeba zmiennej (y zamiast x).
- Zapomnieliśmy o instrukcji `break`, przez co dla $x==2$ zostanie wykonane nieprawidłowe działanie.
- Zapomnieliśmy o domyślnym przypadku (myśląc, że zajęliśmy się nim w instrukcji `if`).
- Napisaliśmy $y<0$, gdy mieliśmy na myśli $0<y$.



Jako testerzy zawsze szukamy nieobsłużonych przypadków. Należy zauważyć, że samo naprawienie błędu nie wystarcza. Może on pojawić się jeszcze raz, gdy nie będziemy patrzeć. Musimy pisać testy, które będą systematycznie znajdować błędy. Gdybyśmy po prostu poprawili ten kod, moglibyśmy równie dobrze wprowadzić nowy błąd. Celem takiego przeglądania kodu nie jest tak naprawdę zauważenie błędów (mimo że to bardzo przydatne), lecz zaprojektowanie odpowiedniego zestawu testów do znalezienia wszystkich błędów (albo mówiąc bardziej realistycznie — znalezienia wielu błędów).

Należy zauważyć, że pętle mają niejawną instrukcję `if` — sprawdzają, czy osiągnięto koniec. W związku z tym pętle są zarazem instrukcjami warunkowymi. Widząc program z rozgałęzieniami, zawsze zadajemy sobie następujące pytanie: „Czy pokryliśmy (przetestowaliśmy) wszystkie rozgałęzienia?”. Co zaskakujące, to nie zawsze jest możliwe w realnych programach (ponieważ w realnym kodzie funkcje są wywoływane zgodnie z zapotrzebowaniem przez inne funkcje i niekoniecznie we wszystkie możliwe sposoby). W związku z tym testerom często zadaje się następujące pytanie: „Jakie uzyskałeś pokrycie kodu?”. Lepiej, żeby odpowiedź na nie brzmiała: „przetestowałem większość rozgałęzień” i była uzupełniona objaśnieniem, czemu trudno sięgnąć do pozostałych rozgałęzień. Pokrycie w 100% jest ideałem.



26.3.4. Testy systemowe

Testowanie jakiegokolwiek większego systemu to praca dla wykwalifikowanych testerów. Na przykład testowanie komputerów zarządzających systemami telefonicznymi odbywa się w specjalnych pomieszczeniach wypełnionych półkami zastawionymi komputerami, które symulują duże ilości połączeń telefonicznych. Takie systemy kosztują miliony i służą jako narzędzie pracy zespołów bardzo dobrze wyszkolonych inżynierów. Główna centrala telefoniczna powinna działać 20 lat i w tym czasie może pozwolić sobie na nie więcej niż 20 minut niedziałania (z jakiegokolwiek powodu, wliczając przerwy w dopływie energii, trzęsienia ziemi i powódzie). Nie będziemy zagłębiać się w szczegóły. Łatwiej by było początkującego fizyka nauczyć obliczania korekacji kursu marsjańskiego łazika. Spróbujemy jednak przekazać pewne informacje, które mogą przydać się w mniejszych projektach lub do zrozumienia procesu testowania większych systemów.



Przed wszystkim należy sobie uświadomić, że testy przeprowadza się po to, aby znaleźć błędy, zwłaszcza takie, które mogą często występować i są potencjalnie niebezpieczne. Nie polega to na prostym napisaniu jak największej liczby testów. Bardzo ważne jest zrozumienie działania całego testowanego systemu. Efektywne testowanie systemu jest jeszcze bardziej zależne od znajomości dziedziny zastosowań niż testowanie jednostkowe. Do utworzenia systemu trzeba więcej niż tylko znajomości języka programowania i informatyki. Potrzebna jest jeszcze wiedza z zakresu obszaru zastosowań oraz na temat ludzi, którzy będą danego programu używać.



Wydaje nam się, że jest to ważny czynnik motywujący nas do pracy z kodem — możemy poznać tyle ciekawych zastosowań i interesujących ludzi.

Aby można było przetestować kompletny system, musi on zostać w całości złożony. To może zajmować dużo czasu, przez co wiele systemów testuje się tylko raz dziennie (często w nocy, gdy programiści śpią) po wykonaniu wszystkich testów jednostkowych. Kluczowe znaczenie mają tu testy regresyjne. Największe szanse na znalezienie błędów są w nowym kodzie i miejscach, w których wcześniej zostały znalezione błędy. Dlatego przeprowadzenie starych testów (regresyjnych) jest podstawą. Bez tego nigdy nie uda się ustabilizować dużego systemu. Nowe błędy byłyby wprowadzane z taką samą prędkością, z jaką byłyby usuwane.

Należy zauważyć, że uznajemy za coś oczywistego, iż poprawienie kilku błędów powoduje automatyczne pojawienie się paru nowych. Mamy nadzieję, że ta liczba nowych błędów będzie mniejsza od liczby usuniętych oraz że te nowe błędy będą mniej poważne. Dopóki nie przeprowadzi się testów regresyjnych i nie doda nowych testów dla nowego kodu, system należy uważać za uszkodzony (przez nasze poprawki błędów).



26.3.5. Znajdowanie założeń, które się nie potwierdzają

W specyfikacji algorytmu `binary_search()` jest wyraźnie napisane, że sekwencja do przeszukania musi być posortowana. To pozbawiło nas okazji do przeprowadzenia wielu podstępnych testów. Ale oczywiście są okazje do napisania złego kodu, dla którego nie przygotowaliśmy testów (z wyjątkiem testów systemowych). Czy możemy wykorzystać nasze rozumienie jednostek systemu (funkcji, klas itp.), aby projektować lepsze testy?

Niestety najprostszą odpowiedzią brzmi: nie. Jako testerzy nie możemy zmieniać kodu. Aby jednak było możliwe wykrywanie przypadków łamania wymagań interfejsu (warunków wstępnych), ktoś musi to sprawdzać albo przed każdym wywołaniem, albo jako część implementacji każdego wywołania (podrozdział 5.5). Jeśli testujemy własny kod, możemy wpisać do niego takie testy. Jeśli jesteśmy testerami i ludzie, którzy piszą kod, chcą nas słuchać (nie zawsze tak jest), możemy im powiedzieć o niesprawdzonych warunkach, aby mogli to poprawić.



Wróćmy jeszcze do algorytmu `binary_search()`. Nie moglibyśmy sprawdzić, że `[first,last)` jest rzeczywiście sekwencją i na dodatek posortowaną (punkt 26.3.2.2). Moglibyśmy natomiast napisać funkcję, która to robi:

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // Sprawdza, czy [first,last) to sekwencja:
    if (last<first) throw Bad_sequence();

    // Sprawdza, czy sekwencja jest posortowana:
    if (2 <= last-first)
        for (Iter p = first+1; p<last; ++p)
            if (*p<*(p-1)) throw Not_ordered();

    // Wszystko jest w porządku, wywołujemy funkcję binary_search():
    return binary_search(first,last,value);
}
```

Są powody, dla których w funkcji `binary_search()` nie umieszczono takich testów. Oto niektóre z nich:

- Testu `last < first` nie można wykonać dla iteratora przechodzącego w przód. Na przykład iterator kontenera `std::list` nie ma operatora `<` (punkt B.3.2). Zasadniczo nie ma dobrego sposobu na sprawdzenie, czy para iteratorów definiuje sekwencję (rozpoznanie iteracji od `first` w nadziei na dojście do `last` nie jest dobrym pomysłem).
- Skanowanie sekwencji, aby przekonać się, czy jej wartości są uporządkowane, jest znacznie kosztowniejsze niż wykonanie samej funkcji `binary_search()` (zadaniem funkcji `binary_search()`, w przeciwieństwie do `std::find()`, nie jest ślepe przeszukiwanie wszystkich elementów sekwencji po kolei, aż do napotkania szukanego).

Co można zrobić? Można zamienić `binary_search` na `b2` na czas testowania (ale tylko dla wywołań funkcji `binary_search()` z iteratorami dostępu wolnego). Alternatywnie można by było nakłonić implementatora funkcji `binary_search()` do dodania kodu, który tester mógłby włączać:


```
template<class Iter, class T> // Ostrzeżenie: zawiera pseudokod
bool binary_search (Iter first, Iter last, const T& value)
{
    if (włączony test) {
        if (Iter jest iteratorem dostępu swobodnego) {
            // Sprawdź, czy [first,last) to sekwencja:
            if (last < first) throw Bad_sequence();
        }

        // Sprawdź, czy sekwencja jest uporządkowana:
        if (first != last) {
            Iter prev = first;
            for (Iter p = ++first; p != last; ++p, ++prev)
                if (*p < *prev) throw Not_ordered();
        }
    }

    // Wyszukiwanie binarne
}
```

Ponieważ znaczenie `włączony test` zależy od aranżacji testowania kodu (dla specyficznego systemu w specyficznej organizacji), pozostawiliśmy to jako pseudokod. Przy testowaniu własnego kodu możesz po prostu utworzyć zmienną o nazwie np. `test_enabled`. Zostawiliśmy też test `Iter` jest iteratorem dostępu swobodnego jako pseudokod, ponieważ nie objaśniliśmy „cech iteratorów” (ang. *iterator traits*). Jeśli potrzebujesz więcej informacji na ich temat, zajrzyj do jakiejś książki o C++ dla zaawansowanych.

26.4. Projektowanie pod kątem testowania



Kiedy zaczynamy pisać program, chcielibyśmy kiedyś go ukończyć i żeby nie miał błędów. Wiemy też, że aby to osiągnąć, trzeba przeprowadzić testy. Dlatego już od samego początku projektujemy, mając na uwadze poprawność i testowanie. W istocie wielu programistów działa

według sloganu: „Zaczynaj testowanie wcześniej i rób to często” i nie zaczynają pisać kodu, dopóki nie przemyślą, jak go będą testować. Myślenie o testowaniu już na wczesnym etapie przede wszystkim pomaga uniknąć błędów (oraz ułatwia je znajdować później). Podpisujemy się pod tą filozofią. Niektórzy programiści wręcz piszą testy jednostkowe, zanim zaimplementują te jednostki.

Przykład przedstawiony w punkcie 26.3.2.1 oraz przykłady w punkcie 26.3.3 ilustrują poniższe kluczowe zasady:

- Używaj dobrze zdefiniowanych interfejsów, aby móc pisać dla nich testy.
- Zapewnij sobie sposób zapisywania operacji w postaci tekstu, aby można było je zapisywać, analizować i powtarzać. Dotyczy to także operacji wyjściowych.
- Wbuduj testy niesprawdzonych założeń (asercji) w kodzie wywołującym, aby znaleźć złe argumenty przed testowaniem systemowym.
- Zminimalizuj zależności i postaraj się, aby były jawne.
- Przyjmij przejrzystą strategię zarządzania zasobami.

Mówiąc bardziej filozoficznie, można te zasady podsumować jako włączenie technik testowania jednostkowego dla podsystemów i kompletnych systemów.

Jeśli wydajność nie ma wielkiego znaczenia, można testy niesprawdzanych założeń (wymagań, warunków wstępnych) pozostawić włączone cały czas. Zwykle jednak są jakieś powody do niesprawdzania systematycznie tych warunków. Widzieliśmy na przykład, że sprawdzenie, czy sekwencja jest posortowana, jest nie tylko skomplikowane, lecz także bardziej kosztowne niż wywołanie funkcji `binary_sort()`. Dlatego dobrze jest tak projektować system, aby można było włączać i wyłączać wybrane testy. W wielu systemach dobrze jest zostawić pewną liczbę mniej pożerających zasoby testów nawet w finalnej wersji. Czasami dzieją się „niemożliwe” rzeczy i lepiej uzyskać konkretny komunikat na ich temat niż dowiedzieć się tylko, że wystąpiła jakaś awaria.



26.5. Debugowanie

Debugowanie to kwestia techniki i podejścia. Z tych dwóch ważniejsze jest podejście. Przeczytaj jeszcze raz rozdział 5. Zwróć uwagę na różnice między testowaniem a debugowaniem. Celem wykonywania obu tych rodzajów czynności jest znajdowanie błędów. Ale debugowanie jest bardziej chaotyczne i zwykle skupia się na usuwaniu znanych błędów i implementowaniu czegoś. Jeśli da się coś zrobić, aby debugowanie bardziej przypominało testowanie, należy to zrobić. Przesadzilibyśmy, mówiąc, że kochamy testowanie, ale z pewnością nie cierpimy debugowania. Dobre testowanie od wczesnych etapów pracy i projektowanie z myślą o testowaniu pomagają zminimalizować konieczność debugowania.



26.6. Wydajność


Aby program był przydatny do użytku, nie wystarczy tylko, aby nie zawierał błędów. Nawet jeśli przyjmie się, że posiada wystarczającą funkcjonalność, musi jeszcze być odpowiednio wydajny. Dobry program jest „wystarczająco wydajny”, tzn. działa z akceptowalną szybkością,



mając do dyspozycji określone zasoby. Należy zauważyć, że bezwzględna wydajność jest nieinteresująca, a obsesyjne optymalizowanie prędkości działania programu może doprowadzić do utworzenia skomplikowanego kodu (w którym może wystąpić więcej błędów, co implikuje więcej debugowania), przez co trudniej go utrzymać (także dostosować do innych platform i zoptymalizować wydajność) i jest to bardziej kosztowne.

Skąd wiadomo, że program (lub jednostka programu) jest „wystarczająco wydajny”? Jeśli nie ma żadnego punktu odniesienia, nie wiadomo. W przypadku wielu programów sprzęt jest na tyle szybki, że pytanie to nie ma nawet znaczenia. Znamy przypadki dostarczania użytkownikom programów skompilowanych w trybie debugowania (tzn. działających około 25 razy wolniej, niż trzeba). Robi się tak, aby uzyskać lepsze informacje diagnostyczne na temat błędów po wdrożeniu (to może zdarzyć się nawet najlepszym programom, gdy muszą współistnieć z kodem napisanym „gdzieś indziej”).

W związku z tym odpowiedź na pytanie: „Czy ten program jest wystarczająco wydajny” brzmi: „Sprawdź, ile czasu zajmują interesujące Cię przypadki użycia”. Oczywiście aby to było możliwe, trzeba bardzo dobrze znać potencjalnych użytkowników oraz mieć pojęcie, co uznaliby oni za „interesujący przypadek” i ile czasu byłoby w stanie na niego poświęcić. Logicznym posunięciem jest zmierzenie stoperem czasu wykonywania testów, aby dowiedzieć się, czy któryś z nich nie zajmuje zbyt dużo czasu. Jest to praktyczne, jeśli do mierzenia czasu używa się funkcji typu `system_clock()` (punkt 26.6.1) oraz gdy można automatycznie porównać czas trwania testów z czasem, który szacunkowo został uznany za rozsądny. Ewentualnie (lub dodatkowo) można zapisać czas trwania testów i porównać wyniki z czasem trwania wcześniejszych testów. Jest to swego rodzaju regresyjny rodzaj testowania wydajności.



Niektóre najgorsze błędy odbijające się na wydajności są wynikiem zastosowania słabych algorytmów i można je wykryć poprzez testowanie. Jednym z powodów, dla których przeprowadza się testy na dużych ilościach danych, jest wykrycie niewydajnych algorytmów. Przeanalizujemy przykład programu sumującego elementy w wierszach macierzy (używającego opisaney w rozdziale 24. biblioteki `Matrix`). Ktoś napisał odpowiednią funkcję:

```
double row_sum(Matrix<double,2> m, int n); // Suma elementów w wierszu m[n]
```

Teraz ktoś przy użyciu tej funkcji tworzy wektor `sum`, gdzie `v[n]` jest sumą elementów `n` pierwszych wierszy:

```
double row_accum(Matrix<double,2> m, int n) // Suma elementów w m<0,n>
{
    double s = 0;
    for (int i=0; i<n; ++i) s+=row_sum(m,i);
    return s;
}
// Oblicza sumę sum wierszy macierzy m:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

Można sobie wyobrazić, że jest to część testu jednostkowego lub kod wykonywany jako część aplikacji badanej przez test systemowy. W obu przypadkach, jeśli macierz będzie naprawdę duża, stanie się coś dziwnego — zasadniczo czas wykonywania rośnie wraz z podnoszeniem do kwadratu rozmiaru macierzy `m`. Dlaczego? Zsumowaliśmy elementy pierwszego wiersza, następnie dodaliśmy elementy z drugiego wiersza (ponownie odwiedzając wszystkie elementy

pierwszego wiersza), później dodaliśmy elementy trzeciego wiersza (ponownie odwiedzając wszystkie elementy pierwszego i drugiego wiersza) itd.

Jeśli uważasz, że to był zły przykład, pomyśl, co by było, gdyby funkcja `row_sum()` musiała pobierać dane z bazy danych. Odczyt danych z dysku jest wiele tysięcy razy wolniejszy niż z pamięci głównej.

Teraz możesz się zżymać, że nikt by nie napisał czegoś tak głupiego. Niestety widzieliśmy dużo gorsze rzeczy, a poza tym nie jest wcale łatwo zauważyć w gąszczu kodu, czy dany algorytm jest dobry czy słaby (z punktu widzenia wydajności). Czy zauważyłeś ten problem z wydajnością, pierwszy raz patrząc na ten kod? Często trudno jest zauważyć problem, jeśli nie szuka się konkretnie tego rodzaju błędu. Poniżej znajduje się prosty przykład realnego kodu znalezioneego w serwerze:



```
for (int i=0; i<strlen(s); ++i) { /*jakiś działania przy użyciu s[i] */ }
```

Często s był łańcuchem o długości nawet 20 000 znaków.

Nie wszystkie problemy związane z wydajnością mają coś wspólnego ze słabymi algorytmami. W istocie (na co zwracaliśmy uwagę w punkcie 26.3.3) znaczna ilość pisanego przez nas kodu nie kwalifikuje się jako właściwe algorytmy. Problemy niezwiązane z algorytmami zwykle klasyfikowane są jako wynikające ze „słabego projektowania”. Zaliczają się do nich:

- Wielokrotne powtarzanie obliczeń (np. przedstawiony powyżej problem z sumowaniem).
- Wielokrotne sprawdzanie tej samej rzeczy (np. sprawdzanie, czy indeks należy do zakresu przy każdym użyciu go w pętli, lub wielokrotne sprawdzanie argumentu, który bez żadnych zmian jest przekazywany do lub z funkcji).
- Wielokrotne wracanie do dysku (lub internetu).

Zwróć uwagę na częste użycie słowa **wielokrotne**. Oczywiście mamy tu na myśli „niepotrzebne powtarzanie”, ale jeśli nie robi się czegoś wielokrotnie, nie będzie to miało wpływu na wydajność. Wszyscy jesteśmy za skrupulatnym sprawdzaniem argumentów funkcji i zmiennych pętlowych, ale jeśli ten sam test wykona się milion razy, to może na tym ucierpieć wydajność. Jeśli pomiary wykażą, że ucierpiała wydajność, sprawdźmy, czy da się pozbyć jakichś powtarzalnych czynności. Nie należy tego robić, dopóki nie zdobędzie się pewności, że problemem jest rzeczywiście wydajność. Zbyt wczesna optymalizacja jest źródłem wielu błędów i powodem marnowania czasu.

26.6.1. Kontrolowanie czasu

Skąd wiadomo, czy dany fragment kodu działa wystarczająco szybko? Skąd wiadomo, ile czasu zajmuje dana operacja? W wielu przypadkach można to sprawdzić, patrząc na zegarek (stoper albo zwykły zegarek na rękę). Nie jest to naukowa ani precyzyjna metoda, ale jeśli nie da się inaczej, pozwala zorientować się, czy program działa wystarczająco szybko. Nie jest dobrze mieć obsesję na punkcie wydajności.




Jeśli trzeba zmierzyć czas w mniejszych jednostkach lub nie można użyć zegarka, trzeba skorzystać z pomocy komputera, który wie, ile czasu zajmują różne operacje i może nas o tym poinformować. W systemie Unix wystarczy przed poleceniem postawić przedrostek `t ime`, aby został wydrukowany czas potrzebny na jego wykonanie. Można tą metodą sprawdzić, ile czasu zajmie kompilacja pliku z kodem źródłowym w języku C++. Zwykle plik o nazwie `x.cpp` kompiluje się tak:

```
g++ x.cpp
```

Aby zmierzyć czas trwania tej kompilacji, należy dodać przedrostek `time`:

```
time g++ x.cpp
```

Powyższe polecenie spowoduje skompilowanie pliku `x.cpp` i wydrukowanie czasu trwania tej operacji na ekranie. Jest to prosta i efektywna metoda mierzenia czasu działania małych programów. Pamiętaj, aby zawsze wykonać kilka pomiarów, ponieważ wynik może zostać zniekształcony przez inne działania komputera. Jeśli trzy razy uzyska się podobny wynik, można mu zaufać.

 Jak zmierzyć czas w milisekundach? Jak wykonać własne, bardziej szczegółowe pomiary części programu? Należy użyć standardowych narzędzi z nagłówka `<chrono>`. Aby na przykład zmierzyć czas działania funkcji `do_something()`, można napisać taki kod:

```
#include <chrono>
#include <iostream>
using namespace std;

int main()
{
    int n = 10000000;                                // Funkcję do_something() wykonamy n razy

    auto t1 = system_clock::now();                    // czas rozpoczęcia

    for (int i = 0; i<n; i++) do_something();           // pętla mierząca czas

    auto t2 = system_clock::now();                    // czas zakończenia

    cout << "Wykonanie do_something() " << n << " razy zajęło "
         << duration_cast<milliseconds>(t2-t1).count() << "milisekund\n";

}
```

Zegar `system_clock` to jeden ze standardowych zegarów, a funkcja `system_clock::now()` zwraca punkt w czasie (`time_point`), w którym została wywołana. Wystarczy odjąć jeden `time_point` od drugiego (tutaj `t2-t1`), aby otrzymać czas wykonywania (`duration`). Słowo kluczowe `auto` oszczędza nam konieczności bawienia się z zawiłościami typów `duration` i `time_point`, które są zaskakująco skomplikowane dla kogoś, dla kogo czas to tylko liczby pokazywane przez zegarek na ręku. W rzeczywistości narzędzia do pracy z czasem biblioteki standardowej początkowo były projektowane z myślą o zaawansowanych zastosowaniach w fizyce, dzięki czemu są znacznie bardziej ogólne i elastyczne niż potrzebuje większość użytkowników.

Aby uzyskać czas trwania w wybranej jednostce, np. sekundach, milisekundach lub nanosekundach, należy przekonwertować (rzutować) daną wartość za pomocą funkcji konwersji `duration_cast`. Konstrukcja typu `duration_cast` jest potrzebna, ponieważ różne systemy i różne zegary mierzą czas w różnych jednostkach. Nie zapomnij o funkcji `.count()`, która pobiera liczbę jednostek (tyknięć zegara) z obiektu `duration` zawierającego zarówno tyknięcia zegara, jak i jednostkę.

Zegar `system_clock` jest przeznaczony do mierzenia przedziałów czasu trwających od ułamka sekundy do kilku sekund. Nie należy za jego pomocą odmierzać godzin.

Pamiętaj, aby nie ufać żadnym wynikom takich pomiarów, jeśli nie da się uzyskać podobnego wyniku przynajmniej trzy razy. Co oznacza „podobny wynik?”. Rozsądnie jest akceptować odchylenie w granicach 10%. Pamiętaj, że nowoczesne komputery są bardzo **szybkie** — nie jest niczym nadzwyczajnym wykonywanie 1 000 000 000 instrukcji na sekundę. To oznacza, że nie da się nic zmierzyć, jeśli nie powtórzy się tego kilkadziesiąt tysięcy razy lub nie wykonuje się jakiejś bardzo wolnej operacji, jak zapis danych na dysku lub dostęp do zasobów internetowych. W tym drugim przypadku wystarczy powtórzenie rzędu kilkuset razy, ale trzeba mieć na uwadze, że dzieje się tyle rzeczy, iż można nie zrozumieć wyniku.



26.7. Źródła

Stone Debbie, Jarrett Caroline, Woodroffe Mark, Minocha Shailey, *User Interface Design and Evaluation*, Morgan Kaufmann, 2005.

Whittaker James A., *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2003.

Ćwiczenia

Przygotuj funkcję `binary_search()` do działania:

1. Zaimplementuj operator wejściowy w strukturze `Test` z punktu 26.3.2.2.
2. Uzupełnij plik testów dla sekwencji z podrozdziału 26.3:
 - a. { 1 2 3 5 8 13 21 } // zwykła sekwencja
 - b. { }
 - c. { 1 }
 - d. { 1 2 3 4 } // parzysta liczba elementów
 - e. { 1 2 3 4 5 } // nieparzysta liczba elementów
 - f. { 1 1 1 1 1 1 1 } // wszystkie elementy takie same
 - g. { 0 1 1 1 1 1 1 1 1 1 1 1 } // inny element na początku
 - h. { 0 0 0 0 0 0 0 0 0 0 0 1 } // inny element na końcu
3. Bazując na punkcie 26.3.1.3, skompletuj program, który generuje:
 - a. bardzo dużą sekwencję (jaką sekwencję uznasz za dużą i dlaczego?);
 - b. dziesięć sekwencji z losowymi liczbami elementów;
 - c. dziesięć sekwencji z 0, 1, 2, 3...9 losowymi elementami (ale uporządkowanych).
4. Powtórz te testy dla sekwencji łańcuchów, np. {Bohr Darwin Einstein Lavoisier Newton Turing}.

Powtórzenie

1. Sporządź listę aplikacji i opisz najgorsze możliwe skutki, jakie mógłby spowodować w nich błąd, np. sterowanie samolotem — katastrofa lotnicza, śmierć 231 osób, straty materialne szacowane na 500 milionów dolarów.
2. Dlaczego po prostu nie dowodzi się poprawności programów?
3. Jaka jest różnica między testowaniem jednostkowym a systemowym?
4. Co to jest testowanie regresyjne i czemu jest ważne?
5. Jaki jest cel testowania?
6. Czemu funkcja `binary_search()` nie sprawdza po prostu swoich wymagań?
7. Jeśli nie można poszukać wszystkich możliwych błędów, jakiego rodzaju błędów należy szukać w pierwszej kolejności?
8. W których miejscach kodu manipulującego sekwencjami elementów istnieje największe prawdopodobieństwo wystąpienia błędów?
9. Czemu dobrze jest testować duże wartości?
10. Czemu często reprezentuje się testy jako dane, a nie kod?
11. Dlaczego i kiedy należy przeprowadzić dużo testów bazujących na wartościach losowych?
12. Dlaczego testowanie programów z GUI jest trudne?
13. Co jest potrzebne do testowania jednostki w „odosobnieniu”?
14. Co łączy „testowalność” z przenośnością?
15. Co sprawia, że testowanie klas jest trudniejsze od testowania funkcji?
16. Czemu ważne jest, aby testy były powtarzalne?
17. Co może zrobić tester, który odkryje, że „jednostka” wykorzystuje nieweryfikowane założenia (warunki wstępne)?
18. Co może zrobić projektant lub implementator, aby ułatwić testowanie?
19. Czym różni się testowanie od debugowania?
20. Kiedy wydajność ma znaczenie?
21. Podaj przynajmniej dwa przykłady tego, jak łatwo można stworzyć problemy słabej wydajności.

Terminologia

<code>clock()</code>	stan	warunek końcowy
dowód	test jednostkowy	warunek wstępny
odmierzanie czasu	test systemowy	wejścia
pokrycie testami	testowanie	wyjścia
projektowanie z myślą o testowaniu	testowanie białej skrzynki	wykorzystanie zasobów
regresja	testowanie czarnej skrzynki	założenia
rozgałęzianie	uprząż testowa	

Praca domowa

1. Przeprowadź na swojej funkcji `binary_search()` z podrozdziału 26.1 testy przedstawione w punkcie 26.3.2.1.
2. Zmodyfikuj testowanie funkcji `binary_search()`, aby można było zastosować dowolne typy elementów. Następnie przetestuj ją na sekwencjach łańcuchów i liczb zmiennoprzecinkowych.
3. Powtórz ćwiczenie z punktu 1. z wersją funkcji `binary_search()` przyjmującą kryterium porównywania jako argument. Sporządź listę nowych okazji do wystąpienia błędów stworzonych przez ten argument.
4. Opracuj taki format danych testowych, aby można było zdefiniować sekwencję raz i uruchomić na niej kilka testów.
5. Dodaj do zbioru testów funkcji `binary_search()` testy pozwalające wykryć (mało prawdopodobny) błąd modyfikowania przez tę funkcję przeszukiwanej sekwencji.
6. Zmodyfikuj kalkulator z rozdziału 7., aby pobierał dane wejściowe z pliku i wysyłał wyniki do pliku (albo użyj narzędzi swojego systemu operacyjnego do przekierowywania wejścia i wyjścia). Następnie utwórz sensownie pełny test.
7. Przetestuj edytor tekstu z podrozdziału 20.6.
8. Dodaj interfejs tekstowy do interfejsowej biblioteki graficznej z rozdziałów 12. – 15. Na przykład łańcuch `Circle(Point(0,1),15)` powinien powodować wygenerowanie wywołania `Circle(Point(0,1),15)`. Przy użyciu tego interfejsu tekstowego narysuj dwuwymiarowy domek z dachem, dwoma oknami i drzwiami.
9. Dodaj tekstowy format wyjściowy dla interfejsowej biblioteki graficznej. Jeśli np. jest wykonywane wywołanie `Circle(Point(0,1),15)`, w strumieniu wyjściowym powinien być drukowany łańcuch typu `Circle(Point(0,1),15)`.
10. Wykorzystaj interfejs tekstowy z zadania 9. do napisania lepszego testu dla interfejsowej biblioteki graficznej.
11. Zmierz czas działania programu sumującego z podrozdziału 26.6 dla kwadratowych macierzy m o wymiarach 100, 10 000, 1 000 000 oraz 10 000 000. Zastosuj losowe wartości dla elementów z przedziału $[-10,10]$. Jeszcze raz napisz kod obliczający v , używając bardziej wydajnego (nie $O(n^2)$) algorytmu i porównaj czasy.
12. Napisz program generujący losowe liczby zmiennoprzecinkowe i sortujący je za pomocą funkcji `std::sort()`. Zmierz czasy sortowania 500 000 i 5 000 000 liczb typu `double`.
13. Powtórz poprzednie zadanie, ale przy użyciu losowych łańcuchów o długościach z przedziału $[0,100)$.
14. Powtórz poprzednie zadanie, tylko zamiast wektora użyj słownika, aby wykluczyć konieczność sortowania.

Podsumowanie

Jako programiści marzymy o pisaniu pięknych programów, które po prostu działają — najlepiej już przy pierwszej próbie. Rzeczywistość jest jednak inna — trudno jest dobrze napisać program i nie łatwiej utrzymać jego dobry stan przy jego ulepszaniu. Testowanie (wliczając projektowanie z myślą o testowaniu) jest najlepszym sposobem na zapewnienie działania systemu. Pod koniec każdego naszego technicznego dnia powinniśmy zawsze poświęcić chwilę refleksji dla (często zapominanych) testerów.

