Testowanie rozwiązań zadań algorytmicznych za pomocą narzędzi napisanych w językach Bash i C++

Grzegorz Guśpiel

Nie chcę, ale muszę.
– Lech Wałęsa

Jeżeli na Olimpiadzie Informatycznej przetestowałeś/-aś swoje rozwiązanie wyłącznie za pomocą testów `ocen` i kilku „z palca” (i nie mówimy tu o brucie, ale o rozwiązaniu o przeciętnym stopniu skomplikowania), to duża szansa, że Twój kod jest zbugowany i prawie nic nie warty. Testowanie automatyczne wystawia zaś Twój program na próbę tysiące razy na sekundę.

W typowym zadaniu z Olimpiady Informatycznej dla ustalonych danych wejściowych jest tylko jedna poprawna odpowiedź (z dokładnością do białych znaków). Zdarzają się też zadania, w których poprawnych odpowiedzi jest wiele – testowanie takich zadań jest odrobinę trudniejsze. Z kolei same testerki ludzie piszą na różne sposoby – na przykład w Bashu, w Pythonie, a nawet w C++. Niniejszy poradnik składa się z dwóch części – w pierwszej poznasz podstawy Basha i napiszesz testerkę w Bashu dla zadania z unikalną poprawną odpowiedzią, a w drugiej dowiesz się, jak napisać testerkę w C++ dla zadania z poprawną odpowiedzią nieunikalną.

Spis treści

Do rozwiązywania ćwiczeń zawartych w tym poradniku niezbędne są pewne materiały. Jeżeli czytasz ten PDF na komputerze i masz dostęp do Internetu, uzyskasz je, klikając na linki. W przeciwnym razie potrzebna Ci jest paczka `http://grzegorzguspiel.staff.tcs.uj.edu.pl/generatory.zip` **.**

Pewne niezbędne podstawy, takie jak przekierowywanie wejścia/wyjścia i niektóre użyteczne polecenia, zostały omówione w poradniku „Techniki przydatne przy rozwiązywaniu zadań algorytmicznych” (`debug.pdf`).

1 Testerka w Bashu, zadanie z unikalną poprawną odpowiedzią

Przypomnijmy, że korzystając z linuksowego terminala, komunikujesz się ze specjalnym programem nazywanym powłoką (ang. *shell*). Działanie powłoki sprowadza się do wczytywania słów, które wpisujesz, i wywoływania odpowiednich programów. Najczęściej używaną powłoką jest Bash.

Bash to również nazwa języka programowania. Jest to język skryptowy, czyli taki, który nie wymaga kompilacji. Dzięki możliwości używania w terminalu języka głęboko zintegrowanego z powłoką, automatyzowanie czynności, które pracują na plikach i przekazują argumenty różnym linuksowym poleceniom, jest o niebo przyjemniejsze.

Gdy uruchamiasz polecenie w terminalu, powłoka dzieli wpisaną przez Ciebie linię na maksymalne spójne kawałki bez spacji. Pierwszy kawałek to wywoływane polecenie, a kolejne to jego argumenty. Liczba spacji między argumentami nie ma znaczenia – ważne, by była choć jedna.

```
POLECENIE ARGUMENT1 ARGUMENT2 ARGUMENT3
```

Ponieważ będziesz intensywnie edytować polecenie wpisywanie w terminalu, zachęcam Cię do zapoznania się z następującymi przyspieszającymi pracę klawiszami i skrótami: `↑` i `↓` (przywoływanie poprzednich poleceń), `CTRL+R` (przeszukiwanie historii poleceń), `HOME` (skok na początek polecenia) oraz `CTRL+←` i `CTRL+→` (przeskakiwanie o jedno słowo w lewo / w prawo w obrębie polecenia).

W wielu sytuacjach, aby przerwać działanie programu lub wpisywanie polecenia, wygodnie Ci będzie użyć `CTRL-C` (czasem raz nie wystarczy).

Często będziesz potrzebował(a) sprawdzać, jak użyć danego polecenia. Robi się to w następujący sposób: `man NAZWA_POLECENIA`. Do poruszania się przydają się strzałki (lub skróty `vim`owe), zaś do wyszukiwania klawisze `/` i `n`. Wychodzi się za pomocą `q`.

1.1 Polecenie `cat`

Aby zapisać coś do pliku `X`, wygodnie jest wywołać `cat > X`, wpisać zawartość, a następnie zatwierdzić tak, jak to było omówione w poradniku `debug.pdf`. Aby odczytać zawartość pliku `X`, wystarczy uruchomić `cat X`.

Ćwiczenie 1 *Zapisz tekst „Ala ma kota” do pliku `ala`. Następnie wypisz zawartość pliku `ala` i usuń go za pomocą polecenia `rm`.*

1.2 Zmienne oraz jak sobie nie zrobić krzywdy

Polecenie `ZMIENNA=WARTOŚĆ` tworzy zmienną o nazwie `ZMIENNA` i przypisuje jej wartość `WARTOŚĆ` – pod warunkiem, że `WARTOŚĆ` nie ma znaków specjalnych ani spacji. Jeżeli kiedykolwiek w Bashu potrzebujesz, by wpisany przez Ciebie ciąg znaków zawierający spacje był traktowany jako jeden spójny kawałek, zawrzyj ten ciąg w podwójnym cudzysłowie, np. `ZMIENNA="ala ma kota"`. **Uwaga! W instrukcji przypisania nie wolno mieć spacji ani po lewej, ani po prawej stronie znaku `=`.**

Użycie wartości zmiennej polega na poprzedzeniu jej nazwy znakiem dolara: `$ZMIENNA`. Do pisania po ekranie służy komenda `echo`, która wypisuje po kolei wszystkie swoje argumenty.

Ćwiczenie 2 *Utwórz zmienną, przypisz do niej wartość i za pomocą `echo` sprawdź, czy się zapisało:) Następnie powtórz, tym razem zapisując wartość zawierającą spację.*

Ćwiczenie 3 *Sprawdź, czy w Bashu nazwy zmiennych są czułe na wielkość liter.*

Uwaga! Niektóre nazwy zmiennych są już zajęte – przez tzw. zmienne środowiskowe. Gdy je nadpiszesz, zazwyczaj problem się pojawi dopiero, gdy uruchomisz program, który danej zmiennej potrzebuje, i to zdarzy Ci się rzadko. Jest jednak kilka wyjątków – nadpisanie zmiennych takich jak `HOME` czy `PATH` ma zabawne skutki. Zazwyczaj wystarczy uruchomić ponownie terminal, by naprawić problem – modyfikacja zmiennej środowiskowej ma efekt tylko w obrębie danego terminala.

Ćwiczenie 4 *W osobnym terminalu przypisz do zmiennej `PATH` pustą wartość (`""`). Następnie spróbuj użyć jakiegokolwiek polecenia konsolowego.*

Ponieważ wszystkie zmienne środowiskowe są pisane wielkimi literami, sugeruję po prostu używać małych liter w nazwach zmiennych:) Przykłady zmiennych środowiskowych, które masz szansę nadpisać:

```
DISPLAY, HOME, JOB, LANG, LANGUAGE, LOGNAME, PATH, PWD, SESSION, SHELL, TERM, USER
```

W Bashu wiele czynności realizuje się nie swojsko brzmiącymi słowami, lecz symbolami. Powoduje to, że programowanie w Bashu wygląda bardziej brzydko niż literacko. Poniższa linia, na przykład, to legalny program (nie uruchamiaj go):

```
:(){ :|:& };:
```

Powyższy kod to znana ciekawostka, tak zwana fork-bomba – kod tworzący wykładniczo samoreplikujący się proces, który ma szanse zawiesić komputer.

Jeżeli w którymś miejscu swojego skryptu potrzebujesz użyć symbolu specjalnego tak, by nie był traktowany jako specjalny, poprzedź go znakiem `\`.

Ćwiczenie 5 *Przypisz do zmiennej `x` wartość `"`. Upewnij się, że zadziałało.*

1.3 Pętla `for`

Składnia pętli `for` wygląda następująco: `for ((i = 0; i < n; i++)); do POLECENIE; done`. Przykład:

```
for ((i = 0; i < 10; i++)); do echo AAAAA; done
```

Pojedyncze polecenie można zastąpić kilkoma – wystarczy każde dwa kolejne rozdzielić średnikiem.

```
for ((i = 0; i < 10; i++)); do echo A; echo B; echo C; done
```

Ćwiczenie 6 *Wykonaj pętlę, która wypisuje liczby od 10 do 20 w kolejnych liniach, za każdym razem pauzując na sekundę (`sleep 1`). Aby przerwać jej działanie, wciśnij `CTRL-C`.*

1.4 Pętla `while`

Warto znać również pętlę `while`:

```
while (( 1 )); do date +%H:%M:%S:%N; done
```

Działanie pętli możesz przerwać Ty (`CTRL-C`) i instrukcja `break`.

1.5 Kod wyjścia i instrukcja `if`

W naszym przykładzie posłużymy się zadaniem „Sortowanie”. Do Twojej dyspozycji są następujące pliki:

- `sortowanie.pdf` – treść zadania,
- `sortowanie-example.in` – test przykładowy,
- `sortowanie-slow.cpp` – brut,
- `sortowanie-sol.cpp` – wzorcówka do przetestowania, z jednym błędem.

Zapisz te pliki w osobnym katalogu. Przy okazji usuń prefiks `sortowanie-` z ich nazw.

Każdy program uruchamiany w powłoce Bash w momencie zakończenia działania zwraca kod wyjścia, czyli liczbę całkowitą. Wartość 0 oznacza sukces (to dlatego w C++ standardowo funkcja `main` zwraca 0), a inne wartości błąd. Czasem różne niezerowe wartości odpowiadają różnym błędom. Kod błędu ostatnio wykonywanego polecenia przechowywany jest w zmiennej `?` (pojedynczy pytajnik) i można go podejrzeć na przykład tak: `echo $?`.

Ćwiczenie 7 *Uruchom polecenie* `ls` *na istniejącym katalogu i zobacz, że kod wyjścia wynosi 0. Następnie uruchom to polecenie, podając mu nieistniejący katalog jako argument, i zobacz, że kod wyjścia jest niezerowy.*

Poniższe ćwiczenia dotyczą polecenia `cmp` – oto krótkie przypomnienie, jak z niego korzystać. Polecenie służy do porównywania ze sobą dwóch tekstów. Poniższa instrukcja porównuje wyjście programu `program` z plikiem `test.out`:

```
./program < test.in | cmp test.out
```

Do porównania ze sobą dwóch plików służy instrukcja:

```
cmp plik1 plik2
```

W razie wykrycia różnic, `cmp` wypisuje informacje o tych różnicach na ekran[1], a w przeciwnym wypadku jego wyjście jest puste. To, o czym nie mówiliśmy jeszcze, to kod wyjścia tego programu: 0 oznacza brak różnic, 1 oznacza wykrycie różnicy, 2 oznacza błąd.

Pamiętaj, że aby odczytać kod wyjścia, między poleceniem, którego kod wyjścia Cię interesuje, a zajrzeniem do zmiennej `?`, nie możesz umieścić żadnego innego polecenia, bo wtedy kod wyjścia tego drugiego polecenia przysłoniłby kod wyjścia polecenia, który chcesz sprawdzić.

Ćwiczenie 8 *Uruchom* `cmp` *tak, aby wygenerować kod wyjścia 0, i potwierdź, że zmienna* `?` *ma wartość 0.*

Ćwiczenie 9 *Uruchom* `cmp` *tak, aby wygenerować kod wyjścia 1, i potwierdź, że zmienna* `?` *ma wartość 1.*

Poniższe przykłady pokazują, jak wygląda instrukcja warunkowa sprawdzająca kod wyjścia.

```
if (($? == 0)); then POLECENIE_JESLI_SUKCES; fi

if (($? != 0)); then POLECENIE_JESLI_PORAZKA; fi

if (($? == 0)); then POLECENIE_JESLI_SUKCES; else POLECENIE_JESLI_PORAZKA; fi
```

Ćwiczenie 10 *Posługując się brutem* `slow.cpp`*, zapisz poprawną odpowiedź dla przykładowego testu* `example.in` *w pliku* `example.out`*. Następnie skonstruuj* ***jednolinijkowe*** *polecenie, które uruchamia wzorcówkę* `sol.cpp` *na teście przykładowym, porównuje output z plikiem* `example.out`*, a następnie (do oddzielenia poleceń możesz użyć średnika) wypisuje na ekran słowo „IDENTYCZNE" lub „RÓŻNE" w zależności od tego, czy program przeszedł test przykładowy. Upewnij się, że Twój one-liner działa, wprowadzając na chwilę błąd do wzorcówki.*

1.6 Zapisywanie skryptów w pliku

We wszystkich dotychczasowych przykładach nasze skrypty były jednolinijkowcami pisanymi bezpośrednio w terminalu. Każdy taki jednolinijkowiec możesz równie dobrze umieścić w pliku i kazać *Bash*owi go uruchomić. Reguły pisania skryptów w pliku są takie same, jak w konsoli – tylko już nie musisz wszystkiego pisać w jednej linii:)

Są co najmniej dwa sposoby na uruchomienie skryptu z pliku. Załóżmy, że zapisałaś/-eś go w pliku `SKRYPT`. Pierwszy sposób polega na ustawieniu praw wykonywalnych do pliku (`chmod +x SKRYPT`) i umieszczeniu `#!/bin/bash` (tzw. shebang) w pierwszej linii. Wtedy można uruchamiać skrypt poleceniem `./SKRYPT`. Jest to sposób lepszy, gdyż Twój edytor odczyta shebang i będzie kolorował składnię. Jeśli zapomnisz powyższej metody, zawsze możesz uruchomić skrypt niezależnie od shebang i praw wykonywalnych komendą: `bash SKRYPT`.

Dla obu wymienionych sposobów rozszerzenie pliku nie ma znaczenia. Jeśli bardzo chcesz mieć jakieś rozszerzenie, `.sh` jest częstym wyborem.

[1] Czasem zdarza się, że `cmp` zgłasza różnicę, której gołym okiem nie widać. Wtedy może chodzić o białe znaki, np. gdy w brucie wypisujesz spację na końcu wyjścia, a we wzorcówce nie. Narzędzia typu `cmp` i `diff` podają dokładną lokalizację pierwszej znalezionej różnicy, co bywa w takich sytuacjach pomocne.

Ćwiczenie 11 *Zapisz swój one-liner z poprzedniego ćwiczenia do pliku za pomocą `CTRL-SHIFT-C`, `CTRL-SHIFT-V` (pozostaw go w jednej linii) i upewnij się, że działa.*

Średnik to tak naprawdę alias dla końca linii. Teraz, gdy przeniosłeś/-aś wszystko do pliku, możesz ładniej sformatowac swój skrypt.

Ćwiczenie 12 *Zastępując średniki końcami linii i wyrównując kod klawiszem `TAB`, spraw, by Twój skrypt wyglądał ładnie, po czym upewnij się, że działa.*

1.7 Przerwa na przygotowanie generatora

Teraz Twoim zadaniem będzie napisanie generatora testów do zadania „Sortowanie".

Losowanie w C++ polega na wołaniu funkcji `rand()`. Funkcja `rand()` generuje losową liczbę całkowitą z zakresu $[0, \texttt{RAND\_MAX}]$, gdzie `RAND_MAX` jest stałą, zazwyczaj wystarczająco dużą[2], zdefiniowaną w `<cstdlib>`. Przed pierwszym losowaniem należy ustawić tzw. seed, podając go do funkcji `srand`. Robi się to raz w trakcie działania programu. Seed determinuje ciąg wyników wywołań `rand()` – a tym samym test, jaki wyprodukuje Twój generator. Będziesz uruchamiać swój generator wielokrotnie, za każdym razem z innym seedem. Seed należy traktować jako numer testu – ponownie uruchamiając generator z tym samym seedem, uzyskasz ten sam test, co ma szansę się przydać.

Twój generator powinien wczytywać poprzez standardowe wejście następujące liczby:

- `SEED`[3]– seed, używany zgodnie z opisem powyżej,
- `Z` – liczbę zestawów danych do wygenerowania,
- `MAXN` – wspólne ograniczenie górne na długość ciągu i jego elementy.

Oto, jak losować liczbę z zakresu $[\texttt{minv}, \texttt{maxv}]$[4]:

```
int randint(int minv, int maxv) { return rand() % (maxv - minv + 1) + minv; }
```

Ważne, by nie trzymać się jednej wartości n (długość ciągu), lecz za każdym razem losować długość ciągu z zakresu między 1 **a `MAXN`.** W przeciwnym razie pomijasz przypadek brzegowy $n = 1$.

Ćwiczenie 13 *Napisz program `gen.cpp` generujący losowe testy do zadania `sortowanie.pdf` zgodnie z powyższą specyfikacją.*

Można przekazywać wartości z linii poleceń wprost na standardowe wejście generatora:

```
echo 42 1 5 | ./gen
```

1.8 Pętla testująca

W tej sekcji napiszesz kompletną testerkę. Możesz to zrobić zarówno jako skrypt w pliku, jak i jako one-liner w konsoli. Niektórzy wolą to pierwsze, bo zbyt długi one-liner jest mało czytelny. Z drugiej strony, edytowanie parametrów generatora bezpośrednio w linii poleceń bywa wygodniejsze, zwłaszcza z pomocą wyjaśnionych na początku poradnika klawiszy/skrótów ↑, ↓, `CTRL+R`, `HOME`, `CTRL+←` i `CTRL+→`. Sam(a) oceń, jaki tryb pracy wolisz.

Niech Twoja testerka ma postać pętli `for` testującej wartości `i` od 1 do, powiedzmy, 1000, w środku której kolejno:

1. Generujesz test `t.in`, używając `SEED=$i`, `Z=1`[5] i `MAXN=4` (na dobry początek).
2. Generujesz poprawną odpowiedź `t.out` za pomocą bruta.
3. Sprawdzasz wynik działania wzorcówki za pomocą `./sol < t.in | cmp t.out`.
4. W razie różnicy, przerywasz pętlę poleceniem `break`. Nie musisz nic pisać na ekran, `cmp` to za Ciebie zrobi.

[2] Dokumentacja gwarantuje tylko $\texttt{RAND\_MAX} \geq 32767$, ale na Twoim komputerze najprawdopodobniej `RAND_MAX` będzie wynosić $2\,147\,483\,647 = 2^{31} - 1$.

[3] Niektórzy zamiast wczytywania seeda podają do funkcji `srand` wartość `time(NULL)`. Jest to fatalne rozwiązanie, jeżeli uruchamiasz generator dużo częściej niż raz na sekundę, gdyż `time(NULL)` zmienia się raz na sekundę.

[4] Oczywiście taki sposób losowania nie daje idealnie równomiernego rozkładu, ale taki wystarcza do większości zastosowań.

[5] Zwróć uwagę, że gdy używasz $Z = 1$, system musi uruchomić nowy proces dla każdego kolejnego zestawu danych, co spowalnia testowanie. Gdy już Twój program przechodzi testy z $Z = 1$, warto zwiększyć Z, tak aby uruchamianie nowych procesów nie było wąskim gardłem – zwiekszysz wtedy szansę na to, że w ustalonym czasie testerka znajdzie ewentualny błąd.

Ćwiczenie 14 *Napisz jednolinijkową testerkę zgodnie z powyższą instrukcją.*

Ćwiczenie 15 *Eksperymentując z różnymi wartościami* `MAXN`*, znajdź (możliwie najmniejszy) test, na którym* `sol.cpp` *działa niepoprawnie. Wypisz go na ekran poleceniem* `cat`*. Następnie napraw błąd w programie* `sol.cpp`*.*

Więcej błędów w programie `sol.cpp` nie ma. W tym momencie powinieneś/-nnaś móc odpalać testerkę z dużą wartością `Z` i różnymi wartościami `MAXN`, i za każdym razem widzieć, że program przechodzi (zrób to). Aby mieć spokój ducha, zróbmy jeszcze ostatnie sanity checki.

Po pierwsze, musimy się upewnić, że żaden element tandemu generator+brut+wzorcówka się nie zapętla ani nie działa zbyt wolno. Nasza testerka nie wypisuje nic na ekran, więc gdyby coś się zapętlało, nawet byśmy tego nie zauważyli. Celujemy w setki, może nawet setki tysięcy zestawów danych na sekundę, inaczej całe nasze przedsięwzięcie ma mały sens.

Ćwiczenie 16 *Tymczasowo dopisz do testerki polecenie* `echo`*, tak aby wypisywała coś na ekran za każdym razem, gdy tworzy nowy test. Uruchom testerkę najpierw z* `MAXN` *rzędu 5 i zobacz, że testów generuje się mnóstwo. Następnie stopniowo zwiększaj* `MAXN` *i zobacz, w którym momencie wykładniczy brut staje się zbyt wolny.*

Chcemy jeszcze upewnić się, że nie generujemy w kółko tego samego testu (przez błąd w mechanizmie seedowania). Ponadto fajnie by było sprawdzić, że to nie jest tak, że losujemy wyłącznie bezsensowne testy (uszkodzone / zbyt małe / w inny sposób trywialne), bo np. mamy błąd w generatorze lub przekazaliśmy mu złe argumenty.

Ćwiczenie 17 *Uruchom testerkę tak, aby chodziła przez dłużej niż kilka sekund. W drugim terminalu kilka razy wpisz* `cat t.in`*, aby upewnić się, że losowane testy 1) są za każdym razem różne i 2) wizualnie wyglądają sensownie.*

Parę ćwiczeń temu napisana przez Ciebie testerka wyłapała błąd w `sol.cpp`, więc jesteśmy spokojni, że w razie błędu we wzorcówce testerka się zatrzyma. Ale gdybyś, już podczas prawdziwych zawodów, miał(a) od początku bezbłędne rozwiązanie, to obowiązkowo powinieneś/-nnaś wprowadzić sztuczny błąd do wzorcówki, by sprawdzić, czy testerka w ogóle wyłapuje błędy.

To by było na tyle – kurs testowania w Bashu możesz uznać za ukończony:)

2 Testerka w C++, zadanie z nieunikalną poprawną odpowiedzią

W tej sekcji napiszemy testerkę w C++, dla odmiany dla zadania, w którym dla jednego testu istnieje wiele poprawnych odpowiedzi. Oczywiście moglibyśmy to zrobić w Bashu, ale robimy to w C++, by poznać alternatywny sposób pisania testerek. Oto nasze zasoby:

- `podciag.pdf` – treść zadania;
- `podciag-gen.cpp` – generator, który wczytuje kolejno seed, liczbę zestawów do wygenerowania i wspólne ograniczenie górne na długość i elementy ciągu, inicjuje generator liczb losowych wartością seed, a następnie wypisuje na standardowe wyjście odpowiedni losowy test;
- `podciag-slow.cpp` – brut wykładniczy, który liczy jedynie długość najdłuższego podciągu rosnącego;
- `podciag-sol.cpp` – rozwiązanie do przetestowania, o złożoności $O(n^2)$[6], z błędami;
- `podciag-checker.cpp` – checkerka, czyli program, który wczytuje kolejno nazwę pliku z testem, nazwę pliku z outputem naszego bruta i nazwę pliku z outputem testowanego programu, a następnie produkuje kod wyjścia 0 wtedy i tylko wtedy, gdy wynik się zgadza.

[6] Jak sugeruje treść zadania, da się lepiej.

Ćwiczenie 18 *Przejdź do nowego katalogu i umieść w nim powyższe pliki, usuwając prefiks* `podciag-` *z ich nazw. Przyjrzyj się implementacji checkerki (nie musisz wnikać w implementację funkcji* `is_subseq`*), w szczególności sposobowi wykorzystania strumienia* `ifstream` *do wczytywania danych z pliku. Bez tego trudno napisać checkerkę, gdyż checkerka musi wczytywać i dane wejściowe, i weryfikowany output (a czasem i output bruta, jak w naszym przykładzie).*

O strumieniu `ifstream` zapamiętaj, że:

- zadeklarowany jest w nagłówku `<fstream>` (oczywiście `<bits/stdc++.h>` też działa),
- należy go stworzyć: `ifstream s;`,
- otworzyć: `s.open(NAZWA_PLIKU);`,
- a następnie używać tak, jak `cin`.

W C++ można uruchamiać polecenia tak, jak w konsoli, za pomocą funkcji `system`. Funkcja jest zadeklarowana w `<cstdlib>`, ale – jak poprzednio – równie dobrze można zainkludować `<bits/stdc++.h>`.

Ćwiczenie 19 *Utwórz program* `tester.cpp`*. Sprawdź, czy wywołania systemowe działają – dopisz* `system("ls");` *Skompiluj, uruchom – powinna się wypisać zawartość bieżącego katalogu.*

Argumentem funkcji `system` jest pojedynczy łańcuch znaków (`const char*`). Jeżeli chcesz ją wywołać na konkatenacji tekstu i liczby, musisz:

- skonwertować liczbę na typ `string` za pomocą funkcji `to_string`,
- skleić oba napisy, korzystając z operatora dodawania dla reprezentantów klasy `string`,
- skonwertować obiekt typu `string` na łańcuch znaków za pomocą funkcji `string::c_str()`.

Ćwiczenie 20 *Z poziomu kodu* `tester.cpp`*, wywołaj polecenie* `echo` *10 razy, podając mu jako argumenty kolejno liczby od 1 do 10.*

Ponieważ wywołania systemowe będziemy wołać wiele razy, warto opakować funkcję `system` w funkcję:

```
void run(string command)
```

Ma to dwa cele. Pierwszy to uniknięcie konieczności otaczania sumy `string`ów nawiasami i wołania na niej `c_str()` za każdym razem, gdy wołamy polecenie systemowe. Drugi to dopisanie asercji, która upewnia się, czy kod wyjścia jest 0. Niemądre byłoby ignorowanie kodów mówiących, że coś się nie powiodło.

Ćwiczenie 21 *W kodzie z poprzedniego ćwiczenia opakuj funkcję* `system` *w funkcję* `run(string)` *i umieść w niej wyżej opisaną asercję. Następnie wywołaj* `run("ls asdfghjkqwerty")` *by upewnić się, że asercja zatrzyma działanie programu w razie niezerowego kodu błędu.*

Ćwiczenie 22 *Uruchom z poziomu testerki polecenie* `echo 0 2 10 | ./gen > t.in`*. Upewnij się, że w pliku* `t.in` *znajduje się test z dwoma zestawami danych.*

Twoja testerka powinna mieć pętlę, która testuje kolejne wartości seed i za każdym razem:

1. za pomocą `echo` przekazuje seed, oczekiwaną liczbę zestawów i ograniczenie wielkości testu do generatora, by stworzyć test `t.in`,
2. uruchamia bruta na teście `t.in` i wynik jego działania zapisuje w pliku `t.out`,
3. uruchamia program `sol` na teście `t.in` i wynik zapisuje w pliku `t.sol`,
4. uruchamia checkerkę, podając jej na wejściu lokalizacje testu, outputu bruta i outputu do weryfikacji.

Jeśli checkerka stwierdzi niepoprawność outputu, funkcja `assert` wyświetli komunikat i zakończy program.

Ćwiczenie 23 *Zaimplementuj testerkę zgodnie z powyższymi wytycznymi i znajdź (możliwie najmniejszy) test, na którym program nie działa. Następnie przejrzyj kod* `sol.cpp` *i napraw linijkę, która odpowiada za ten błąd. Powinno Ci ją być łatwo znaleźć, głębokie zrozumienie kodu nie będzie konieczne:)*

3 Zakończenie i źródła

Gratulacje – ćwiczenia ukończone:) Powodzenia na Olimpiadzie!

Podziękowania za uwagi dla Jana Gwinnera, Adama Polaka, Jacka Salaty i Jakuba Siuty. Odsyłam również do materiałów:

1. `http://grzegorzguspiel.staff.tcs.uj.edu.pl/debug/wikol-guide.zip` – *Poradnik testowania rozwiązań na Olimpiadzie Informatycznej* autorstwa Wiktora Kuropatwy.
2. `http://github.com/glapul/testowanie_advanced` – warsztaty z testowania przygotowane przez Michała Glapę.