Techniki przydatne przy rozwiązywaniu zadań algorytmicznych

Grzegorz Guśpiel

Celem początkowych ćwiczeń z niniejszego poradnika jest zagwarantowanie, by Twoja nauka algorytmiki nie została spowolniona przez brak pewnych całkiem prostych do opanowania umiejętności technicznych. Następne ćwiczenia uczą umiejętności zaawansowanych, przekładających się na dalsze usprawnianie procesu debugowania kodu i zwiększających Twoją samodzielność.

Podczas rozwiązywania kolejnych zadań nie wahaj się prosić o pomoc / dodatkowe wyjaśnienia!

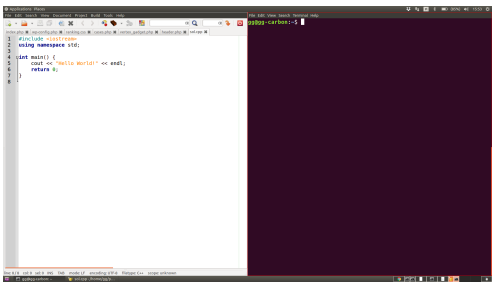
Spis treści

Pobierz plik `http://grzegorzguspiel.staff.tcs.uj.edu.pl/debug.zip` . Zawiera on materiały niezbędne do rozwiązywania ćwiczeń zawartych w tym poradniku.

1 Środowisko pracy

Poradnik zakłada, że opanowałaś/-eś już poruszanie się po katalogach w linuksowym terminalu za pomocą `ls` i `cd`. Jeśli tak nie jest, poświęć chwilę na poćwiczenie tych poleceń.

Oczywiście możesz kompilować i uruchamiać program tak, jak Ci się żywnie podoba, ale chciałbym, abyś wypróbował(a) moją propozycję. Po wykonaniu pierwszego ćwiczenia możesz wrócić do swojego ulubionego sposobu pracy (choć rozważ korzystanie z mojej propozycji w pozostałych ćwiczeniach, pamiętaj, że to nie są ćwiczenia na czas, więc możesz sobie pozwolić na eksperymenty:)).



Otwórz terminal tak, aby zajmował prawą połowę ekranu. Utwórz osobny katalog na zadanie: `mkdir NAZWA_ZADANIA` . Unikaj używania spacji w nazwach plików, w terminalu najwygodniej pracuje się z plikami, których nazwy ich nie mają. Przejdź do niego: `cd NAZWA_ZADANIA` . Otwórz plik z kodem w swoim ulubionym edytorze, np.: `geany sol.cpp &` . Przesuń okno tak, aby wypełniało lewą połowę ekranu. Aby skompilować, wpisz `make sol` . Aby uruchomić, wpisz `./sol` .

Przeglądarkę otwórz na osobnym pulpicie. Między pulpitami przełączaj się za pomocą `CTRL+ALT+strzałki`.

Zapewne wiesz, że polecenia *kopiuj* i *wklej* mają skróty `CTRL+C`, `CTRL+V`. W terminalu prawdopodobnie będziesz musiał(a) używać `CTRL+SHIFT+C` i `CTRL+SHIFT+V`.

Pamiętaj, że w terminalu strzałki góra/dół przywołują poprzednio wpisane polecenia.

Gdy testujesz rozwiązanie, test zapisz do pliku, uruchamiając `cat > test.in` , wpisując test i zatwierdzając za pomocą `CTRL-D` (czasem trzeba wcisnąć `CTRL-D` dwa razy[1]). Gdy chcesz uruchomić swój program na tym teście, użyj `./sol < test.in` . To w połączeniu ze strzałką w górę znaczne przyspiesza testowanie.

Jeszcze wygodniej Ci będzie, używając polecenia `make sol && ./sol < test.in` . Zbitka `&&` łączy dwa polecenia: najpierw wykonuje pierwsze, a następnie drugie, przy czym drugie zostanie wykonane tylko wtedy, gdy pierwsze się powiedzie.

Pamiętaj, że można wpisać tylko część polecenia / nazwy pliku i wcisnąć `TAB` (jak nic się nie dzieje, to kilka razy), aby terminal spróbował dokończyć nazwę za Ciebie.

Podczas pracy nad zadaniem prawdopodobnie wyprodukujesz więcej niż jeden test. Staraj się nowe testy zapisywać do nowych plików `.in`, aby móc do nich zawsze wrócić. Przecież gdy program zacznie działać na nowym teście, może przestać działać na starym.

Ćwiczenie 1.1 *Zgodnie z powyższymi instrukcjami otwórz terminal, utwórz nowy katalog dla zadania „Dodawanie” (`dodawanie.pdf`), otwórz plik z kodem i napisz rozwiązanie,* ***wystarczy, że będzie działać dla liczb z zakresu do miliarda – czyli wczytujesz 2 inty i je sumujesz****. Następnie przeklej test przykładowy i zapisz do pliku `0.in` za pomocą polecenia `cat`, po czym zapisz do plików `1.in` i `2.in` testy `1 99` i `99 1`. Przetestuj swój program na tych trzech testach. Przy okazji wypróbuj uzupełnianie nazw za pomocą `TAB`.*

W poniższych ćwiczeniach zachęcam Cię do rozwiązywania każdego w osobnym katalogu. Ma to dwa cele. Po pierwsze, podczas pracy nad kodem często pojawia się więcej plików niż tylko kod i binarka. Gdy utworzysz kilka testów, bez osobnego katalogu miał(a)byś bałagan. Po drugie, wiem, że nie wszyscy wygodnie poruszacie się po katalogach za pomocą poleceń `cd` i `ls`. Warto się do nich przyzwyczajać.

2 Debugowanie poprzez czytanie kodu

W szerokim znaczeniu, słowo debugowanie oznacza usuwanie błędów z programów, jakkolwiek, choćby zanosząc kod do wróżki. W wąskim znaczeniu, słowo odnosi się do uruchamiania programu na konkretnych danych i śledzenia przebiegu jego działania.

[1] Jeżeli w terminalu są jakieś znaki wpisane, ale nie wysłane do aplikacji (tak się dzieje zawsze, gdy coś wpiszesz bez zatwierdzania `ENTER`-em), to `CTRL-D` jedynie wysyła te znaki do programu. Dopiero w sytuacji, gdy każdy wpisany w terminalu znak dotarł do programu, wciśnięcie `CTRL-D` informuje program, że wejście się skończyło – na taki sygnał program `cat` reaguje zamknięciem pliku i skończeniem działania.

Niektóre błędy najprościej znaleźć, porządnie czytając kod linijka po linijce. Należy zwracać szczególną uwagę na wszystkie przypadki brzegowe (0 czy 1, `<` czy `<=`, warunki logiczne w pętlach i ifach), nazwy zmiennych itd.

Ćwiczenie 2.1 *Przeczytaj treść zadania „Patyczki" (`patyczki.pdf`) i zrozum test przykładowy. Następnie, nie uruchamiając programu, znajdź błąd w następującym rozwiązaniu: `patyczki.cpp`. Jeśli masz dostęp do tego zadania na Satori, zasubmituj i uzyskaj OK.*

3 Kompilowanie na odpowiednim systemie i z odpowiednimi flagami

Jedyne, co robi polecenie `make` (w najprostszym zastosowaniu), to sprawdza, czy odpowiedni plik z kodem się zmienił, i jeśli tak, to uruchamia kompilator, by stworzyć nową binarkę (tak nazywamy plik wykonywalny). Zwróć uwagę na to, że `make` wyświetla, jakim poleceniem ten kompilator został uruchomiony. W razie potrzeby zawsze możesz to polecenie zmodyfikować i uruchomić samemu.

Jeśli chcesz uruchomić kompilator bez pośrednictwa `make`, użyj polecenia `g++ program.cpp -o program`. Wszystko to, co dopiszesz do tego polecenia, nazywamy flagami. Flagi to ustawienia dla kompilatora. Przykładowe użycie flag: `g++ program.cpp -o program -O2 -std=c++0x`.

Polecenie `make` należy skonfigurować, tak aby używało odpowiednich flag. Aby to zrobić, dopisuje się linię

```
export CXXFLAGS="TU LISTA TWOICH FLAG"
```

do pliku `.bashrc` w katalogu domowym. Kropka oznacza pod Linuksem plik ukryty, więc aby się do niego dostać, najwygodniej użyć terminala. Flagi te zaczną być stosowane przy ponownym uruchomieniu terminala. Aby się upewnić, że udało Ci się ustawić nowe domyślne flagi, po prostu popatrz, czy pojawiły się one w outpucie `make`'a.

Unikniesz wielu kłopotów, kompilując w odpowiedni sposób. Satori kompiluje program pod 64-bitowym Linuksem, za pomocą określonej wersji kompilatora GCC i z określonymi flagami. Jeżeli Ty robisz inaczej, zazwyczaj nie ma to znaczenia, ale czasem te różnice wychodzą na jaw i powodują, że na Satori jest błąd, a na Twoim komputerze wszystko działa. Wtedy ważne jest jak najwierniejsze zreprodukowanie warunków z Satori. Przypominam polecane domyślne flagi kompilacji[2]:

```
-Wall -Wextra -Wshadow -Wunused -std=c++0x -g -O2 -static
```

W kolejnych zadaniach, o ile nie jest powiedziane inaczej, należy kompilować w ten sposób.

Flagi `-W` to warningi. Możesz na własną odpowiedzialność zrezygnować z warningów, zwłaszcza że powyższe są nieco paranoiczne, ale na Twoim miejscu nauczyłbym się raczej pisać kod tak, by te warningi nigdy nie wyskakiwały, chyba że tylko na chwilę[3] (taka jest ogólna zasada pracy z warningami – czyścić je od razu, bo przecież jak człowiek nauczy się je ignorować i zrobi się ich kilkanaście, to ich potem nie przeczyta). Raz na jakiś czas popełnisz błąd, który warning wytknie natychmiast, a gdybyś warninga nie miał(a), debugowanie zajęłoby dużo czasu. Niezainicjowane zmienne, kolizje zmiennych o tej samej nazwie, niezauważone konwersje typów zmiennych[4], brak instrukcji `return`...

Nie wszystkie komunikaty kompilatora będą dla Ciebie zrozumiałe. Jeżeli któregoś nie rozumiesz, pisz do prowadzącego.

Na Satori masz RTE na teście 0 (czyli pewnie przykładowym), a u Ciebie działa? Użyj Linuksa i sprawdź flagi, zwłaszcza `-O2`. Niektóre błędy typu RTE wychodzą dopiero z tą flagą. W razie braku dostępu do systemu Linux polecam stronę ideone.com.

Na koniec prosta rada: output kompilatora czyta się **od góry**. Interesuje Cię pierwszy błąd, a nie jego konsekwencje. Jeżeli dopadła Cię plaga `g++`-a, koszmarnie długa lista błędów, to przytrzymaj klawisz `ENTER`, by zrobić dużo pustego miejsca w terminalu, i za pomocą strzałki do góry skompiluj kod ponownie, a następnie za pomocą rolki przejedź na początek wyniku kompilacji.

[2] Na Olimpiadzie Informatycznej używana jest już flaga `-O3`.

[3] Uwaga do osób startujących w konkursach: zawodników/-czek ta rada dotyczy wręcz podwójnie:) Warningi mogą uratować Ci skórę.

[4] Tu może pomóc flaga kompilacji `-Wconversion`. Nie została ona wymieniona w polecanych flagach, gdyż produkuje sporo warningów w poprawnych kodach, co bywa dla wielu osób irytujące. Za każdym razem, gdy wyskoczy taki niepotrzebny warning, trzeba w dane miejsce w kodzie wpisać odpowiednie rzutowanie – np. gdy przypisujesz rozmiar wektora (typu unsigned) do zmiennej typu int, należy napisać `(int)w.size()`. Ale ta flaga czasem uratuję Ci skórę, gdy zapomnisz o long longach, i działa przy innych błędach na styku różnych typów. Zachęcam do jej stosowania.

4 Testy „z palca”

W większości zadań powinnaś/powinieneś być w stanie wymyślić kilka prostych testów, na których możesz uruchomić swój program.

Ćwiczenie 4.1 *Nie czytając kodu, znajdź test, na którym nie działa następujące rozwiązanie zadania „Algorytm Euklidesa” (`euklides.pdf`): `gcd.cpp`. Na razie nie zajmuj się naprawą tego kodu, jedynie zapisz test w pliku `test.in`. Hint: jeśli nie umiesz znaleźć testu, spójrz na treść zadania i zastanów się, jakie są przypadki brzegowe.*

Jeżeli w zadaniu są zestawy, warto podwoić test przykładowy, tzn. skopiować wszystkie zestawy i wkleić je na końcu, tworząc test mający 2 razy więcej zestawów.

Aby z poziomu terminala edytować jakiś plik, możesz użyć edytora graficznego: `gedit 0.in &` lub tekstowego: `mcedit 0.in` (wariant hardcore: `vim 0.in`). Znak `&` powoduje, że terminal się nie zblokuje (co jest bardzo przydatne przy uruchamianu aplikacji graficznych i kompletnie bez sensu, jeśli uruchamiasz edytor działający w trybie tekstowym), ale uważaj – zamknięcie terminala zamknie edytor bez ostrzeżenia. Nie wszystkie aplikacje graficzne uruchamia się wygodnie w ten sposób. Czasem taka aplikacja wysyła tekst na standardowe wyjście, co zaśmieca terminal. Jeżeli np. uruchomiony przez Ciebie Google Chrome tak robi, to wystarczy uruchomić go w następujący sposób: `google-chrome > /dev/null 2> /dev/null &` i problem znika.

Ćwiczenie 4.2 *(Oczywiście w osobnym katalogu) zapisz kod `dwojkowy.cpp` w pliku `sol.cpp`. Jest to zepsute rozwiązanie zadania „System dwójkowy” (`dwojkowy.pdf`). Zapisz test przykładowy w pliku `0.in`, używając polecenia `cat`. Następnie w pliku `00.in` zapisz podwojenie testu przykładowego. Uruchom, znajdź błąd i jeśli masz dostęp do sprawdzaczki, uzyskaj OK.*

Zanim zaczniesz śledzić działanie programu na konkretnym teście, zmniejsz go maksymalnie, ile się da, by potem nie przekopywać się przez masę śmieciowych komunikatów. Np. jeśli jest wiele zapytań, warto zostawić tylko to, na którym program nie działa.

Ćwiczenie 4.3 *(Oczywiście w osobnym katalogu) zapisz kod `binsearch_ans.cpp` w pliku `sol.cpp` i test `binsearch.in` w pliku `1.in`. Jest to rozwiązanie zadania “Naczelny Statystyk” (`naczelny.pdf`). Program `sol.cpp` na tym teście nie działa. Zrób kopię testu: `cp 1.in 1b.in` i przerób ten test tak, by był jak najmniejszy i by wciąż program `sol` na nim nie działał.*

5 Debugowanie za pomocą `cout`

Gdy masz test, na którym program nie działa, wstawiasz w kluczowe miejsca programu polecenia wypisujące na ekran, co program zrobił i jaki jest stan zmiennych.

Metoda ta wydaje Ci się banalna i w sam raz dla ludzi z epoki kamienia łupanego? Uważasz, że prawdziwi programiści korzystają z dużo bardziej zaawansowanych technik? Niektórzy może tak, ale debugowanie za pomocą `cout` jest niezawodną i najważniejszą metodą debugowania. Wszystko można w ten sposób zdebugować. Nie potrzebujesz umieć prawie nic więcej.

Każde użycie `cout` do debugowania kończ znakiem `endl`! (Nie dotyczy to sytuacji, gdy komunikat jest częścią większego komunikatu, np. wypisujesz na ekran tabelkę z zawartością tablicy.) Powody są dwa: czytelność i automatyczne flushowanie wyjścia.

5.1 Flushowanie

Obiekt `cout` nie pisze prosto na ekran, lecz do specjalnej tablicy zwanej buforem, która raz na jakiś czas jest przenoszona na ekran. Przenoszenie to nazywa się „flushowaniem”. Jeżeli nastąpi błąd wykonania, program może nie zdążyć zflushować bufora...

Ćwiczenie 5.1 *(Oczywiście w osobnym katalogu) zapisz kod `flush.cpp` pod nazwą `flush1.cpp`. Uruchom: `./flush1`. Czy rozumiesz, dlaczego nic się nie wypisało?*

Ćwiczenie 5.2 *Ten sam kod zapisz w `flush2.cpp` i zmień w nim `" "` na `"\n"`. Uruchom: `./flush2`*

Znak `'\n'` wymusza flushowanie (przynajmniej na komputerze piszącego te słowa). Czy jest to jednak pewna metoda?

Ćwiczenie 5.3 *Uruchom program, przekierowując jego wyjście do pliku* `2.out`*:* `./flush2 > 2.out`*. Obejrzyj zawartość pliku* `2.out`*. Następnie zrób kolejną kopię programu, nazwij ją* `flush3.cpp` *i zmień w niej* `"\n"` *na* `endl`*. Uruchom program, najpierw normalnie, a potem przekierowując wyjście do pliku* `3.out`*. Czy tym razem zadziałało?*

Na komputerze autora poradnika `"\n"` flushuje wyjście, jeżeli piszemy na ekran, i nie flushuje w razie pisania do pliku. Na podstawie tego, co zostało powiedziane do tej pory, nie wiemy, czy tak się dzieje zawsze. Stąd rada, by w komunikatach debugujących korzystać z `endl`, gdyż w tym przypadku gwarancje mamy w dokumentacji, zgodnie z którą `endl` **flushuje zawsze**.

Podsumowując, morał tej części poradnika jest następujący: często patrzymy, które komunikaty debugujące się wypisały, i jeśli jakiś komunikat się nie wypisał, to znaczy, że program nie doszedł do tego miejsca. Powyższe ćwiczenia pokazują Ci, że to rozumowanie jest uprawnione tylko wtedy, jeśli zadbałeś/-aś o flush po każdym swoim komunikacie debugującym. Dlatego właśnie należy używać `endl` po każdym debugującym `cout`-cie.

5.2 Sposób wypisywania tablic i inne sugestie dotyczące czytelności

Uczysz się programowania dynamicznego i właśnie wyprodukowałaś/-eś dwuwymiarową tablicę z wynikami dla podzadań? Wypisz ją całą na ekran i sprawdź, czy policzyła się zgodnie z definicją, którą na pewno z wielką starannością postawiłeś/-aś jednoznacznie:)

Miło, gdy wypisana przez Ciebie tabelka ma prawdziwe kolumny. Aby to osiągnąć, trzeba zadbać, by kolejne pozycje miały po tyle samo znaków. Można to osiągnąć, oddzielając je symbolem tabulacji `'\t'` zamiast spacji lub, jeśli `'\t'` zajmuje za dużo znaków i wiersz tabelki się nie mieści, używając `cout` z odpowiednim manipulatorem:
`cout << setw(MAKSYMALNA_LICZBA_CYFR) << t[i][j] << " ".`

Ćwiczenie 5.4 *W zadaniu „Skarb faraona” (*`skarb.pdf`*) należy wypełnić tablicę* $t[][]$*, gdzie dla* $i \in \{0, ..., n\}, j \in \{0, ..., B\}$*, pozycja* $t[i][j]$ *oznacza maksymalną wartość plecaka o pojemności* j*, zapakowanego przedmiotami wybranymi ze zbioru* $\{1, ..., n\}$*. W swoim rozwiązaniu tego zadania wypisz tę tablicę na ekran tak, aby dla testu przykładowego wyglądało to następująco:*

```
     0  1  2  3  4  5  6  7  8  9 10
 0:  0  0  0  0  0  0  0  0  0  0  0
 1:  0  0  0  5  5  5  5  5  5  5  5
 2:  0  0  0  5  5  5  5  5 16 16 16
 3:  0  0  0  5  5  5 10 10 16 16 16
 4:  0  0  0  5  7  7 10 12 16 16 17
```

Mała uwaga na koniec: jeżeli wyjście w zadaniu to ciąg znaków lub liczb w jednej linii, to należy tę linię wypisać za jednym zamachem, bez skomplikowanej logiki między wypisywaniem kolejnych fragmentów linii. W przeciwnym razie trudno Ci będzie dodać komunikaty debugujące do tej logiki. Łatwo ten cel osiągnąć. Jeżeli w zadaniu odpowiedź to ciąg znaków produkowany przez złożoną logikę, to zamiast wypisywać wyjście znak po znaku, należy stworzyć `string`-a i dopisywać znaki do niego, a pod koniec wywołać `cout` raz.

5.3 Triki

W C++ istnieje narzędzie zwane preprocesorem. Jest to program uruchamiany przed właściwą kompilacją, który wstępnie obrabia kod. Interesują go wyłącznie linie zaczynające się znakiem `#`. To preprocesor wkleja w miejsce wszystkich dyrektyw `#include` kolejne pliki. Ale polecenie `#include` to tylko mały fragment możliwości tego narzędzia. Niektórzy korzystają z dyrektywy `#define`. Umieszczenie w kodzie

```
#define COŚ COŚ INNEGO
```

powoduje, że wszystkie dalsze wystąpienia pierwszego słowa po `#define` (tutaj: `COŚ`) zostaną zastąpione przez pozostałą zawartość linii (u nas: `COŚ INNEGO`). Twoje podstawienia mogą też przyjmować argumenty, wtedy zwane są makrami (dlaczego tyle nawiasów? wyjaśni się za chwilę):

```
#define KWADRAT(a) ((a) * (a))
// ...
cout << KWADRAT(5) << endl;
```

Makra są niebezpieczne, gdyż są traktowane jak tekst, nie jak wyrażenia języka C++. W makrach nie ma kontroli zgodności typów, co grozi błędami. Jaskrawym przykładem, dlaczego używając preprocesora, należy wiedzieć, co się robi, jest następujące ćwiczenie:

Ćwiczenie 5.5 (*) *Uruchom kod* `preprocesor.cpp`. *Czy rozumiesz, dlaczego program wypisał 6, a nie 8?*

W nowoczesnym programowaniu preprocesor używany jest rzadko. Jeżeli w jakimś zastosowaniu zamiast preprocesora można użyć czegoś bezpieczniejszego, wybiera się to drugie. Używajcie preprocesora z ostrożnością.

Po tym, jak zostaliście ostrzeżeni, przejdźmy do tego, do czego preprocesor może się przydać. W krótkich kodach na algorytmice niektórzy lubią długie konstrukcje językowe zastępować makrami. Przykładem jest makro `#VAR`:

```
#define VAR(v) #v << " " << v << " "
```

Tutaj krzyżyk jest specjalną konstrukcją preprocesora, wypisującą nazwę zmiennej. Dzięki `VAR` można napisać np.

```
cout << VAR(zmienna1) << VAR(zmienna2) << endl;
```

i to zostanie zastąpione przez

```
cout << "zmienna1" << " " << zmienna1 << " " << "zmienna2" << " " << zmienna2 << " " << endl;
```

Wygodne, prawda?

Ćwiczenie 5.6 *Ściągnij następujące rozwiązanie zadania „Zbiory” (*`zbiory.pdf`*):* `zbiory.cpp`. *Za pomocą makra* `VAR` *spraw, aby funkcja* `pole` *wypisywała na ekran swoją nazwę i* ***wszystkie*** *argumenty, na których została wywołana. Zauważ, że nie musisz samodzielnie wstawiać spacji między kolejne wywołania* `VAR`. *Przetestuj na przykładowym.*

Poznaj jedną z przewag `cout`-ów nad `printf`-ami:

Ćwiczenie 5.7 (*) *Ściągnij lepsze rozwiązanie zadania „Zbiory”:* `zbiory_struct.cpp` *i zrób to samo, tylko aplikując makro* `VAR` *bezpośrednio do zmiennych typu* `P`. *Jest to możliwe dzięki operatorowi* `ostream& operator<<()`, *który powoduje, że* `cout`, *otrzymując zmienną typu* `P`, *wie, jak ją wypisać.*

Można mieć generyczny kod do wypisywania par i wektorów!

Ćwiczenie 5.8 (*) *Ściągnij program* `ostream.cpp` *i uzupełnij kod operatora tak, aby wektor* `v1` *został wypisany w sposób następujący:* `[1, 2, 3]`

Wielokrotne zakomentowywanie i odkomentowywanie komunikatów debugujących jest żmudne i prosi się o pomyłkę. Ludzie radzą sobie z tym różnie[5], jednym z rozwiązań jest poniższe makro:

```
#define debug if(1)
```

Najwygodniej umieścić je na początku pliku. Typowe zasosowania:

```
debug cout << "funkcja f " << VAR(zmienna) << endl;

debug funkcja_wypisujaca_na_ekran_wieksza_strukture(struktura);

debug {
  cout << ... ;
  for (...) {
    ...
  }
}
```

Zamiana 1 na 0 w definicji makra wyłącza wszystkie komunikaty debugujące.

[5] Na przykład za pomocą strumienia `cerr` – to, co tam wyślesz, nie jest traktowane jako output programu, a w terminalu się wyświetla. Zważ jednak na to, że wtedy zużywasz czas procesora na policzenie tych wartości. Jeśli te komunikaty są liczone w gorszej złożoności niż rozwiązanie, lub po prostu piszesz zbyt dużo do strumienia `cerr`, to Twoje rozwiązanie straci punkty na czasie. Dlatego odradzam takie rozwiązanie.

Ćwiczenie 5.9 *(Dla osób mających dostęp do zadania na Satori.) W swoim rozwiązaniu zadania „Wojna" wypisz komunikat debugujący za każdym razem, gdy wołasz* `push` *na którejś kolejce. Wypisuj wtedy nazwę kolejki i wstawianą wartość. Uruchom na przykładowym, po czym wyłącz komunikaty debugujące i wyślij rozwiązanie na Satori.*

6 Wczesne wyłapywanie błędów za pomocą asercji

Po dopisaniu do programu `#include <cassert>` (lub `#include <bits/stdc++.h>`, to include'uje całą bibliotekę standardową za jednym zamachem[6]) możesz używać makra `assert`. Przykładowy sposób użycia:

```
assert(!stos.empty());
int wartosc = stos.top();
```

Makro `assert` przyjmuje wartość logiczną i powoduje błąd wykonania, jeżeli wartość to fałsz. Dobrym momentem na użycie tego makra są sytuacje, gdy dla poprawności programu kluczowa jest jakaś własność i choć wydaje Ci się, że ta własność jest spełniona, to po napisaniu stu linii kodu trudno mieć pewność. Jeżeli to, na co liczyłeś/-aś, wcale nie jest prawdziwe, to dowiesz się o tym natychmiast, a nie dostając komunikat ANS na potencjalnie wielkim teście, bez żadnej wskazówki, gdzie zacząć szukać błędu...

Ćwiczenie 6.1 *(Dla osób mających dostęp do zadania na Satori.) Jeżeli dostałeś/-aś kiedyś komunikat ANS w zadaniu „Randka w ciemno", to ściągnij swoje pierwsze zgłoszenie z tym komunikatem i za pomocą asercji sprawdź, czy Twoje sortowanie zadziałało poprawnie na teście przykładowym i na Satori. To mówi Ci, gdzie szukać błędu – w części sortującej, czy w algorytmie z dwoma wskaźnikami. Jeżeli nie miałeś/-aś nigdy ANS-a na tym zadaniu, wygeneruj go sobie wcześniej, psując w jakiś sposób sortowanie.*

Warto wiedzieć: umieszczenie `#define NDEBUG` na początku programu wyłącza wszystkie asercje.

7 Absolutne minimum znajomości `gdb`

Debugger `gdb` jest potężnym narzędziem, umożliwiającym zatrzymanie programu w wybranym punkcie i podejrzenie wszystkich zmiennych. Na tym etapie jego znajomość jest Ci niepotrzebna, za wyjątkiem dwu sytuacji omówionych w niniejszej sekcji.

7.1 Znajdowanie linii powodującej błąd wykonania i inne podstawy

Najpierw zatroszcz się o to, by program był skompilowany z flagą `-g` (czyli upewnij się, że znajduje się ona wśród argumentów polecenia `g++`)[7]. Flaga ta jest konieczna, gdy używasz `gdb`, gdyż bez niej debugger nie poda Ci numerów linii. Generalna zasada jest taka, że program powinien być kompilowany dokładnie tak, jak na Satori. Poza wspomnianym `-g`, które powinno być włączone. Nie przejmuj się też warningami (`-Wcośtam`), które nie mają wpływu na zachowanie programu.

Na osobny komentarz zasługuje mająca ogromne znaczenie dla działania programu flaga `-O2` (alternatywnie `-O3`), czyli włączanie optymalizacji (wyższy numer oznacza mocniejszą, bardziej agresywną optymalizację):

- Na sprawdzaczce program testowany jest z optymalizacją. Oznacza to, że jeśli chcesz dokładnie zreprodukować zachowanie programu ze sprawdzaczki, to musisz użyć tej samej flagi optymalizacji, co sprawdzaczka. Niestety włączenie optymalizacji może spowodować, że `gdb` nie będzie widział części Twojego kodu / zmiennych (w ramach optymalizacji kompilator może zastąpić Twój kod dowolnym kodem, który produkuje ten sam efekt), zaś dodawanie `cout`-ów debugujących do programu kompilowanego z optymalizacją często powoduje, że kod zostanie inaczej zoptymalizowany i błędy ujawnią się w innym miejscu – i debugowanie jest wtedy nieprzyjemne.

- Jeżeli zamiast tego użyjesz flagi `-O0`, czyli wyłączysz optymalizację, to jest spora szansa, że program i tak zatrzyma się z błędem (niekoniecznie tym samym). Możesz wtedy spokojnie zdebugować ten błąd pod flagą `-O0` bez zatruwania sobie życia przykrym wpływem optymalizacji na proces debugowania.

[6] Przez co kompilacja potrwa odrobinę dłużej...

[7] Jest też flaga `-ggdb3`. Nie korzystałem z niej, więc za nią nie ręczę, ale ponoć jest nowocześniejsza, więc możesz z nią poeksperymentować.

Osobiście zaczynam debugowanie z `-O2` – z lenistwa, gdyż nie chce mi się przekompilowywac programu, i dlatego, bo fajnie jest na początku widzieć dokładnie to, co się stało na Satori. Dopiero, jak debugowanie z `-O2` zaczyna boleć (i błąd nie znika przy przejściu na `-O0`), przechodzę na `-O0`. Na początek przygody z debugowaniem być może lepiej jest robić na odwrót: najpierw próbować z `-O0` i przechodzić na `-O2` tylko wtedy, gdy jest to konieczne.

Po skompilowaniu programu, aby uruchomić debugger, należy wpisać `gdb NAZWA_PROGRAMU`. Aby odpalić program, należy wpisać `run`. Aby progam czytał z pliku zamiast z klawiatury, należy wpisać `run < test.in`. W razie błędu wykonania `gdb` wstrzyma działanie programu, pokaże Ci linię, w której jesteś, i da możliwość inspekcji stanu programu. Możesz wypisać stan zmiennej za pomocą `p ZMIENNA` (alias dla `print ZMIENNA`) i obejrzeć tzw. stos wywołań (która funkcja wywołała funkcję bieżącą, która funkcja wywołała funkcję, która wywołała funkcję bieżacą itd.) za pomocą polecenia `bt` (alias dla `backtrace`). Aby wyjść, wciśnij `CTRL+D` (zadziała, jeśli nic nie wpisałaś/-eś w bieżącej linii, w przeciwnym razie trzeba najpierw ją wyczyścić `BACKSPACE`-em lub za pomocą `CTRL+C`).

Ćwiczenie 7.1 *Wróć do katalogu, który stworzyłeś/-aś na potrzeby ćwiczenia 4.1. Jeśli nie zrobiłaś/-eś tego wcześniej, umieść w pliku `test.in` znaleziony wcześniej kontrprzykład. Za pomocą `gdb` znajdź linię, w której nastąpił błąd wykonania. Następnie wypróbuj polecenie `p` i za pomocą `bt` sprawdź, która funkcja wywołała funkcję `gcd`. Zwróć uwagę na to, czy stos wywołań jest wypisywany od najbardziej „zewnętrznych", czy „najnowszych" wywołań funkcji.*

7.2 Znajdowanie miejsca, w którym program się zapętla

Wystarczy uruchomić `gdb` tak, jak poprzednio, i wcisnąć `CTRL+C` w trakcie wykonywania nieskończonej pętli. Debugger natychmiast wstrzyma działanie programu w tym miejscu, w którym akurat uda mu się go przyłapać. Być może złapiesz program głęboko w wywołaniu jakiejś funkcji bibliotecznej, ale to nic – za pomocą `bt` możesz sprawdzić, w środku których swoich funkcji jesteś i w których liniach.

Ćwiczenie 7.2 *Skompiluj `gdb_tle.cpp` z `-O2` i uruchom w gdb. W której linii nastąpiło zapętlenie programu? Prawdopodobnie jesteś głęboko w środku jakiejś funkcji bibliotecznej – **wskaż tę z funkcji zadeklarowanych w pliku `gdb_tle.cpp`, która znajduje się najbliżej szczytu stosu wywołań**. Czy wypisano stan zmiennej `choice`? Jeśli nie, skompiluj bez `-O2` i sprawdź ponownie.*

8 Weryfikacja wyjścia programu za pomocą `cmp`

Aby zapisać wyjście programu do pliku, uruchom `./program < test.in > test.out`. Aby porównać wyjście programu z plikiem: `./program < test.in | cmp test.out`. Aby porównać dwa pliki ze sobą: `cmp plik1 plik2`. W razie wykrycia różnic, program `cmp` wypisuje informację o tym na ekran, a w przeciwnym razie nie wypisuje nic.

Ćwiczenie 8.1 *Uruchom generator dużego testu (`skarb_gen.cpp`) do zadania „Skarb faraona" (`skarb.pdf`), zapisując wygenerowany test w pliku `max.in`. Uruchom swoje rozwiązanie na tym teście i zapisz wynik w pliku `max.out`. Następnie zmień typ głównej tablicy na `short` i uruchom swoje rozwiązanie na teście `max.in`, porównując wyjście z plikiem `max.out`. Czy `cmp` zgłosił różnicę? Teraz napraw rozwiązanie i sprawdź ponownie.*

9 Pomiar czasu działania za pomocą `time`

Czas działania dowolnego polecenia można przetestować w następujący sposób: `time POLECENIE`. Na przykład: `time ./program < test.in`. Jeżeli program wypisuje dużo na standardowe wyjście, aby pomiar był bardziej wiarygodny, przekieruj wyjście do pliku: `time ./program < test.in > test.out` lub w nicość: `time ./program < test.in > /dev/null`.

Ćwiczenie 9.1 *Zmierz czas działania swojego rozwiązania zadania „Skarb faraona" na teście `max.in`. Następnie zamień kolejność pętli w swoim rozwiązaniu: zamiast najpierw liczyć wszystkie $t[1][]$, potem wszystkie*

t[2][] *itd., najpierw policz wszystkie t*[][0]*, potem wszystkie t*[][1] *itd. Jak zmieniła się wydajność Twojego programu?*

10 Kontrola zużycia pamięci

10.1 Ustawianie limitów pamięciowych: polecenie `ulimit`

W terminalu (ściślej, w powłoce, ang. *shell*, czyli w programie, który siedzi w terminalu, rozmawia z Tobą i koordynuje uruchamianie wpisywanych przez Ciebie poleceń) możesz ustawić limit pamięci (zwanej pamięcią wirtualną, ang. *virtual memory*). Jest to maksymalna ilość pamięci, jaką mogą zużyć programy uruchamiane w danym terminalu. Limit ten można wyświetlić za pomocą `ulimit -v` i zmienić za pomocą `ulimit -v NOWY_LIMIT`.

Ćwiczenie 10.1 *Otwórz stronę podręcznika dla używanej przez Ciebie powłoki* `bash`*:* `man bash`*. Wyszukaj polecenie ulimit, wciskając znak* `/`*, wpisując* `ulimit` *i zatwierdzając* `ENTER`*-em. Sprawdź, w jakiej jednostce podawany jest limit pamięci wirtualnej. Wyjdź za pomocą klawisza* `q`*.*

Ćwiczenie 10.2 *Stwórz program z globalną tablicą o wielkości 30 MB i pustym mainem. Ustaw limit pamięci wirtualnej na 35 MB i zobacz, że program działa. Zmień na 29 MB i zobacz, że nie działa. Spróbuj przywrócić limit pamięci 35 MB. Udało się?*

Widzisz więc, że nałożonego za pomocą `ulimit -v` limitu pamięci nie można cofnąć – trzeba zrestartować terminal. Mi to jakoś bardzo nie przeszkadza. Pewnie da się tego uniknąć, ale w tym celu pasowałoby otworzyć `man bash`, wyszukać `ulimit` i poczytać o tym, że limity dzielą się na *hard* i *soft* oraz jakimi flagami obsługuje się to rozróżnienie.

Pamięć programu dzieli się na: fragment wykonywalny (skompilowany kod programu i bibliotek), miejsce dla zmiennych globalnych, stos i stertę. Stos to zmienne deklarowane w funkcjach, a sterta to pamięć alokowana dynamicznie (np. za pomocą `new` i przez STL). Sam fakt wywołania funkcji również zostaje odnotowany na stosie – dzięki temu wiadomo, w które miejsce kodu procesor ma wrócić po wyjściu z funkcji[8]. W systemach desktopowych często obecny jest osobny limit na stos. Oznacza to, że równolegle z kontrolą limitu `ulimit -v`, kontrolowana jest wielkość Twojego stosu i jeżeli program przekroczy dozwoloną wartość, zostanie wywłaszczony. Na Satori nie ma osobnego limitu na stos. To dlatego niektóre programy, zwłaszcza z rekurencją, mogą dostać OK na Satori i nie działać na Twoim komputerze. Do podejrzenia i zmiany limitu na stos służy `ulimit -s`. W szczególności, aby znieść limit na stos w swoim terminalu, użyj `ulimit -s unlimited`.

10.2 Orientacyjny pomiar ilości użytej pamięci

Istnieje kilka metod mierzenia, ile pamięci zużywa program. Nie należy się nimi bezrefleksyjnie kierować. Sprawdzaczka może korzystać z innej metody pomiaru niż Ty. Możesz nie mieć testu, który zmusza Twój program do największego zużycia pamięci. Najlepiej kierować się kalkulatorem i operatorem `sizeof`, i dodatkowo zostawić odpowiedni margines, jeżeli używane są np. `vectory`. Pamiętaj, że to, ile pamięci zużyje program, zależy od platformy, na której go uruchamiasz. Jest duża różnica między systemami 32- a 64-bitowymi, gdyż w pierwszych wskaźnik zajmuje 4, a w drugich 8 bajtów, zmienia się też pojemność typu `long`.

Z tym zastrzeżeniem, przedstawiam jedną z metod szybkiego sprawdzenia zużycia pamięci:
`/usr/bin/time -v ./PROGRAM` (nie mylić z `time`!) – interesuje Cię wartość *maximum resident set size*.

Ćwiczenie 10.3 (*) *Zmierz za pomocą* `/usr/bin/time`*, ile pamięci zużywa Twój program z poprzedniego zadania. Mało, prawda? A przecież program nie dał rady się wykonać, gdy* `ulimit` *ustawiłaś/-eś na 29 MB. Dopisz w* `main`*-ie pętlę, która zeruje całą tablicę, i ponownie użyj polecenia* `/usr/bin/time`*.*

Morał z powyższego ćwiczenia jest taki, że to, że Twój program zużywa mało pamięci wg `/usr/bin/time` nie znaczy, że zmieści się w limicie pamięciowym! Innymi słowy, `/usr/bin/time` nie zawsze mówi prawdę.

[8] Oznacza to również, że wywołania rekurencyjne kosztują pamięć. Jeżeli głębokość wywołań rekurencyjnych sięga miliona, to trzeba się liczyć z tym, że kosztuje to kilka-kilkanaście MB, nawet jeśli funkcja nie przyjmuje argumentów i nie deklaruje nowych zmiennych.

10.3 Ostrzeżenie przed `-O3`

Gdy chcemy oszacować, ile pamięci zużywa wywołanie rekurencyjne, jest na to klasyczny sposób. Koszt pojedynczego wywołania rekurencyjnego w systemie 64-bitowym to około 16 bajtów plus suma wielkości wszystkich zmiennych lokalnych i argumentów funkcji. Czyli gdy funkcja `f(int)` woła samą siebie milion razy i za każdym wywołaniem potrzebuje pamięci tylko na swój argument typu `int`, to należy się spodziewać zużycia $1000000 \cdot (16 + 4)$ bajtów.

Niedawno jeden z uczniów spotkał się z przypadkiem, gdzie korzystanie z `-O3` zamiast `-O2` powodowało, że program zużywał więcej pamięci, niż wynikałoby to z powyższego szacowania. Nie jest to bug, jest to feature `-O3`. Po prostu `-O3` jest na tyle agresywnym ustawieniem, że może powodować, że program będzie zużywał trochę więcej stosu, jeśli to przyspieszy działanie programu. Nie wiemy wiele o tym, co dokładnie ta flaga może spowodować. Zapamiętaj po prostu, by liczyć się z tym, że wywołania rekurencyjne mogą zajmować nieco więcej (np. dwu-/trzykrotnie więcej) pamięci, niż się spodziewasz, jeśli włączona jest flaga `-O3`.

11 Wyłapywanie przyczyn RTE za pomocą `-fsanitize` i `-D_GLIBCXX_DEBUG`

Wyobraź sobie, że Twój kod skorzystał z cudzej pamięci (np. z powodu wyjścia poza tablicę lub nieprawidłowego wskaźnika), ale działa dalej. Nie wolno Ci mu już ufać! Twój program właśnie zamienił się w bombę z opóźnionym zapłonem. Od pierwszego momentu, gdy program wykonał taką nieprawidłową operację, należy się spodziewać wszystkiego: ANS-a, RTE, nieskończonej pętli i tego, że spadnie Ci na głowę fortepian. Analizowanie zachowania programu po tym momencie nie ma najmniejszego sensu. Gdy tylko zorientujesz się, że Twój program zachowuje się dziwnie, musisz przestać przejmować się wynikami generowanymi przez algorytm i przestawić się na szukanie samych błędów wykonania. Dopisywanie `cout`ów nie zawsze wskaże Ci miejsce, gdzie jest błąd, gdyż pierwszy objaw błędu wykonania może pojawić się 100 linii dalej, niż błąd wystąpił. Prawdopodobnie w takiej sytuacji musiał(a)byś czytać kod linia po linii i sprawdzać wszystkie indeksy, wielkości tablic itd. Okazuje się, że da się prościej – istnieją narzędzia, które często są w stanie wskazać Ci miejsce, gdzie program zrobił coś niedozwolonego.

Pierwszym jest flaga kompilacji `-fsanitize=address`. Dodaje ona do programu specjalny kod, który będzie się wykonywał przy każdym odwołaniu do pamięci i sprawdzał, czy przypadkiem nie skorzystano z pamięci cudzej. Drugim jest flaga `-D_GLIBCXX_DEBUG`, która włącza w bibliotece standardowej mnóstwo checków – przydaje się, jeśli niepoprawnie korzystasz z STL-a. **Nie zapomnij, by wraz z nimi używać `-g`, inaczej nie będziesz miał(a) numerów linii.** Tej drugiej flagi warto używać w połączeniu z `gdb`, aby dostać numer linii we własnym kodzie, a nie tylko w środku STL-a.[9]

Sugeruję znać oba narzędzia. Czasem lepiej sprawdzi się jedno, a czasem drugie.

Aby pokazać Ci, że `-O2` potrafi namieszać podczas debugowania, zaczniemy ćwiczenie z `-O2`, a potem przejdziemy na `-O0`.

Ćwiczenie 11.1 *Kod `spa.cpp` w jednym miejscu wychodzi poza tablicę, lecz nie powoduje to przerwania działania programu od razu. Na sprawdzaczce dostaje RTE, zaś na teście `spa.in` co prawda kończy działanie bez błędów, lecz daje złą odpowiedź.*

Najpierw skompiluj rozwiązanie tak, jak zwykle (z `-O2`) i uruchom na teście – powinien produkować output bez objawów błędu wykonania. Następnie skompiluj i uruchom program z `-fsanitize=address` i `-O2`. Powinieneś/powinnaś zaobserwować, że błąd następuje podczas instrukcji przypisania `result[i] = 0;`. Aby to zbadać, wypisz wartość zmiennej `i` chwilę przed błędem – w tym celu odkomentuj polecenie z linii 200, skompiluj (z `-fsanitize` i `-O2`) i uruchom program ponownie na tym teście. Okazuje się, że teraz błąd leci w innej linii!

Powyższe zaskakujące zachowanie jest spowodowane flagą `-O2`, która spowodowała, że początkowy wynik `-fsanitize` był mylący. Skorzystanie ze zmiennej *i* zmieniło sposób, w jaki kompilator zoptymalizował program, i `-fsanitize` zgłosiło coś innego. Aby wynik `-fsanitize` był wiarygodny, powinnaś/powinieneś wyłączyć optymalizację, czyli zastąpić `-O2` przez `-O0`.

Ćwiczenie 11.2 *Z powrotem zakomentuj linię 200, a następnie uruchom program na tym samym teście, ale kompilując z `-O0`. Za pomocą `-fsanitize` ustal, w której linii następuje błąd wykonania. Teraz chcemy potwierdzić, że w tym miejscu wychodzimy poza tablicę/wektor. W podejrzanej linii dwa razy odwołujemy*

[9] Tych narzędzi jest jeszcze więcej, starałem się wybrać najbardziej przydatne. Jeśli chcesz poczytać o reszcie, zajrzyj do sekcji *Źródła*.

się do zawartości pewnego wektora. Wypisz na ekran, jaka jest wielkość tego wektora, i o jakie indeksy pyta program. Ćwiczenie możesz uznać za zaliczone, jeśli za pomocą powyższego wypisywania faktycznie na własne oczy zobaszysz, że któryś z indeksów wyszedł poza tablicę. Na koniec, jeżeli chcesz zobaczyć inną flagę kompilacji w akcji, skompiluj program tym razem bez `-fsanitize` *ale za to z* `-D_GLIBCXX_DEBUG` *i zobacz, jakiego rodzaju komunikat dostaniesz wtedy.*

Korzystając z okazji, że jesteśmy przy błędach wykonania, chciałbym *zniechęcić* Cię do samodzielnego zarządzania pamięcią. Chodzi o operatory `new` i `delete`. Jeżeli po prostu potrzebujesz tablicy o wielkości nieznanej w czasie kompilacji – do tego służy `vector`. Jeżeli chcesz mieć strukturę wskaźnikową, możesz przechowywać indeksy zamiast wskaźników – przy okazji komunikaty debugujące staną się czytelniejsze. Jeżeli potrzebujesz np. alokować nowe węzły listy, wskaźniki w Twoich węzłach mogą wskazywać na elementy pewnej globalnej tablicy. To wszystko minimalizuje ryzyko RTE, gdyż jeśli używasz `new`, musisz dbać o to, by na każde użycie `new` przypadało dokładnie jedno użycie `delete`. Jeżeli mniej – gubisz pamięć. Jeżeli więcej – produkujesz błąd wykonania. Po co się męczyć?

12 Czego nie zdążyliśmy omówić

Kilka tematów, o których można by jeszcze porozmawiać:

- przydatność skrótu `CTRL+R` w terminalu,
- przeglądanie dużych plików z wejściem/wyjściem za pomocą `less` (alternatywnie `vim`, jeśli umiesz się nim posługiwać),
- trik: spacja na początku wpisanego w terminalu polecenia zapobiega zapamiętaniu polecenia w historii – przydatne dla poleceń, które mogą coś usunąć/nadpisać, jak np. `rm`, `cp` i `mv`,
- podstawy Basha,
- podstawy testowania: pętla w Bashu, pisanie generatorów,
- uwaga, że gdy w kodzie obecne są duże stałe, niektóre rodzaje błędów ujawnią się tylko na nietrywialnych dużych testach i nie mamy tego jak przetestować,
- profilowanie, np. `valgrind` wraz z `cg_annotate`.

13 Źródła

Uczynienie niniejszego materiału bardziej kompletnym umożliwiły sugestie kolegów z TCS-u, w szczególności Michała Glapy, Wiktora Kuropatwy, Krzysztofa Maziarza i Adama Polaka. Jeśli chcesz dowiedzieć się więcej o testowaniu rozwiązań, możesz chcieć zerknąć na poniższe materiały:

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.
3. `http://grzegorzguspiel.staff.tcs.uj.edu.pl/debug/propdoc-1.3.pdf` – rady organizatorów Olimpiady Informatycznej.
4. `http://codeforces.com/blog/entry/15547` – jeszcze więcej przydatnych flag kompilacji.