Pracownia z Algorytmów i struktur danych

Zasady zaliczania i szczegóły techniczne oceniania (semestr letni 2020)

1 Zasady zaliczania

Podstawą zaliczenia pracowni jest liczba punktów zdobytych za rozwiązywanie zadań programistycznych. W przypadku zaliczenia pracowni student otrzymuje od 0 do 15 *punktów egzaminacyjnych*. Jest to wkład pracowni w ocenę końcową; liczba ta zależy od tego, czy student zdaje egzamin na poziomie licencjackim czy magisterskim. Niezaliczenie pracowni implikuje niezaliczenie całego przedmiotu. Student otrzymuje również „zwykłą" ocenę za pracownię w skali 2,0 – 5,0. Jest to ocena na poziomie licencjackim, która zostaje automatycznie zmieniona na ocenę z poziomu magisterskiego, jeśli student zda egzamin na poziomie magisterskim.

W semestrze ukaże się 6 zadań programistycznych, sprawdzających zdolność algorytmicznego myślenia, a także umiejętność zakodowania różnych algorytmów i struktur danych. Zadania będą publikowane na stronie pracowni: `http://www.ii.uni.wroc.pl/~mbi/dyd/aisd_20s/`. Prowadzący pracownię będzie też informować o nowych zadaniach mailem wysyłanym do zapisanych na wykład w Systemie Zapisów. Jednocześnie ostateczną wykładnią jest powyższa strona: nieprzeczytanie maila nie jest okolicznością łagodzącą i usprawiedliwieniem tego, że nie zrobiło się zadania.

1. Na każde zadanie będzie co najmniej 2 tygodnie, zazwyczaj około 3–4 tygodni.

2. Każde zadanie polega na wczytaniu danych wejściowych ze standardowego wejścia, wykonaniu obliczeń i wypisaniu wyniku na standardowe wyjście. Za każde zadanie można otrzymać od 0 do 10 punktów. Dokładne zasady oceniania znajdują się w części *Techniczne szczegóły oceniania*.

3. Zadania oceniane są automatycznie; odnośnik do sprawdzaczki znajduje się na stronie pracowni. Punkty przyznane automatycznie przez program sprawdzający mogą być anulowane przez prowadzącego pracownię, jeśli stwierdzi on niezgodność rozwiązania z warunkami podanymi w treści. Przykładowo jeśli w zadaniu zabronione jest używanie losowości, student, który jej użyje, otrzyma zero punktów.

4. W przypadku zdawania egzaminu na poziomie magisterskim punkty egzaminacyjne zależą od wszystkich zadań, zaś w przypadku poziomu licencjackiego pomijane są dwa najgorzej rozwiązane przez studenta zadania. Dokładny algorytm obliczania oceny i tych punktów jest następujący.

a) Niech X_M będzie sumą punktów, które student uzyskał za wszystkie zadania, natomiast X_L będzie sumą punktów z pominięciem dwóch zadań, za które student uzyskał najmniejszą liczbę punktów. Zatem

$$X_M \in [0, 60] \qquad \text{oraz} \qquad X_L \in [0, 40].$$

b) Niech

$$Y_M = 30 \cdot \left(\frac{X_M}{60} - \frac{1}{2} \right) \qquad \text{oraz} \qquad Y_L = 30 \cdot \left(\frac{X_L}{40} - \frac{1}{2} \right).$$

Warto zauważyć, że $Y_M \leq Y_L \leq 15$.

c) W zależności od wartości Y_L możliwe są następujące dwa przypadki:

- Jeśli $Y_L < 0$, student nie zalicza pracowni.
- Jeśli $Y_L \geq 0$, student zalicza pracownię otrzymując z niej ocenę na podstawie poniższej tabeli.

Y_L	ocena
12 – 15	5.0
9 – 12	4.5
6 – 9	4.0
3 – 6	3.5
0 – 3	3.0

Przypadki graniczne rozstrzygane są na korzyść studenta. Jeśli $Y_L \geq 0$, student ma prawo podejścia do egzaminu na poziomie licencjackim, otrzymując Y_L punktów egzaminacyjnych wliczanych do tego egzaminu.

d) Jeśli $Y_M \geq 0$, student ma prawo zdawać egzamin również na poziomie magisterskim i otrzymuje Y_M punktów egzaminacyjnych wliczanych do tego egzaminu.

5. Wszystkie programy będą automatycznie porównywane ze sobą a także z rozwiązaniami dostępnymi w Internecie. W przypadku wykrycia identycznych lub podobnych rozwiązań grożą sankcje włącznie z niezaliczeniem całego przedmiotu i skreśleniem z listy studentów. Minimalną karą jest anulowanie punktów za to zadanie i dodatkowe zmniejszenie wartości Y_L i Y_M o 3. Kara dotyczy *wszystkich* osób, u których wykryto izomorficzne rozwiązania.

2 Dodatkowe zasady dla powtarzających przedmiot

Do egzaminu w roku R wlicza się *maksimum* punktów egzaminacyjnych zdobytych w latach $R-2$, $R-1$ oraz R. Maksimum liczone jest osobno dla wartości Y_M i Y_L.

Dla wpisywania oceny w roku 2020 obowiązują dotychczasowe przepisy; ulegną one zmianie za rok (patrz niżej).

Wpisywanie oceny w roku 2020

Ocena w roku 2020 jest maksymalną oceną wyliczoną (patrz punkt 4 z poprzedniej sekcji) na podstawie zadań rozwiązanych w latach 2018, 2019 i 2020. Do podejścia do egzaminu konieczne jest posiadanie pozytywnej oceny z pracowni wpisanej do USOS-a w roku 2019 lub 2020. Dokładniej mówiąc można wyróżnić dwa przypadki:

1. Student ma pozytywną ocenę w USOS-ie z pracowni z roku 2019 (i być może pozytywną ocenę z roku 2020). Wtedy warto nie zapisywać się na pracownię w Systemie Zapisów; nie otrzymuje się oceny w USOS-ie w roku 2020, lecz ważna jest ocena z roku 2019. Można (i warto) bez negatywnych konsekwencji rozwiązywać zadania i w ten sposób poprawić sobie punkty egzaminacyjne (nie można ich pogorszyć).

2. Student ma w USOS-ie pozytywną ocenę z pracowni z roku 2018 i nie ma pozytywnej oceny z roku 2019. W takim przypadku trzeba zapisać się na pracownię w Systemie Zapisów (i prawdopodobnie zapłacić za jej powtarzanie). Można (i warto) bez negatywnych konsekwencji rozwiązywać zadania i w ten sposób poprawić sobie punkty egzaminacyjne i wpisywaną w roku R ocenę (nie mozna ich pogorszyć). [1]

Wpisywanie oceny od roku 2021

Ocena w roku $R \geq 2021$ jest oceną otrzymaną na podstawie zadań rozwiązanych w roku R i upoważnia do podejścia do egzaminu tylko w roku R.

3 Techniczne szczegóły oceniania

1. Dla każdego zadania istnieje 13 testów: 3 testy *jawne*, 5 testy *pół-jawne* i 5 testów *tajnych*. Testy jawne dołączone są do treści zadania wraz z poprawnymi odpowiedziami. Zawartość testów pół-jawnych i tajnych jest nieznana.

2. Po wysłaniu zadania do sprawdzaczki:
   - Program zostaje skompilowany. Jeśli wystąpi błąd kompilacji program zostanie odrzucony.
   - Program jest sprawdzany na testach jawnych i pół-jawnych. Student zostaje powiadomiony o wynikach tych testów.

3. Do upływu terminu oddawania zadania, można wysyłać rozwiązanie dowolną liczbę razy. Nie ma kar za dużą liczbę wysłanych rozwiązań. Po upływie terminu oddawania zadania, sprawdzaczka przestaje akceptować kolejne rozwiązania.[2] **Ocenie podlega ostatnia wersja, którą udało się zgłosić do sprawdzaczki.** Zostanie ona przetestowana na testach jawnych, pół-jawnych i tajnych.

4. Test zostanie zaliczony, jeśli program udzieli poprawnej odpowiedzi, udzieli jej w określonym w warunkach zadania czasie i nie zużyje więcej pamięci niż podane w treści zadania. Rozmiar kodu źródłowego programu nie może przekraczać 100 KB. Za każdy test można dostać albo komunikat `OK` oznaczający, że test został wykonany pomyślnie i zostaną za niego przyznane punkty albo jeden z następujących komunikatów o błędzie:
   - `(WA) Wrong answer`: wygenerowana przez program odpowiedź jest nieprawidłowa.
   - `(TLE) Time limit exceeded`: program przekroczył limit czasu przewidziany na test.
   - `(RE) Runtime error`: program zakończył się z błędem wykonania.
   - `(IO) Illegal operation`: program próbował wykonać niedozwoloną operację, taką jak otwarcie pliku czy uruchomienie dodatkowego procesu lub wątku.

[1] Można zatem mieć automatycznie przepisaną ocenę z roku $R-2$ pod warunkiem, została ona otrzymana na podstawie zadań rozwiązywanych w roku $R-2$ a nie już przepisana.

[2] Z różnych przyczyn sprawdzaczka kończy przyjmować rozwiązania ok. 3 min. po zapowiedzianym terminie.

Za każdy zaliczony test pół-jawny lub tajny student otrzymuje 1 punkt. Za testy jawne student nie otrzymuje żadnych punktów. Niekompilujący się program otrzyma zero punktów.

5. Programy będą kompilowane i uruchamiane w 64-bitowym środowisku Linux na komputerze PC. Pamięć cache procesora sprawdzaczki to 3 MB.

   Programy będą zapisywane w pliku `prog.`*`rozszerzenie`* i kompilowane do pliku `prog`. Wersje kompilatorów i opcje kompilacji dla poszczególnych języków są następujące:

   - C i C++, kompilator `gcc 8.3.0`
     ```
     gcc -std=gnu18 -Wall -Wextra -Wshadow -O2 -static -DSPRAWDZACZKA -lm
     g++ -std=gnu++17 -Wall -Wextra -Wshadow -O2 -static -DSPRAWDZACZKA
     ```
   - Haskell, kompilator `Glasgow Haskell Compilation System 8.4.4`
     ```
     ghc -Wall -O2 -optl-static -optl-pthread
     ```
   - OCaml, kompilator `OCaml native-code compiler 4.05.0`
     ```
     ocamlopt -ccopt -static -w A
     ```
   - Pascal, kompilator `Free Pascal Compiler 3.0.4`
     ```
     fpc -O2 -Xt -dSPRAWDZACZKA
     ```

4 Często zadawane pytania (z odpowiedziami)

Co to jest standardowe wejście i wyjście?

Najprostsza odpowiedź brzmi: to klawiatura i monitor. Oczywiście nie należy zakładać, że po drugiej stronie siedzi żywa osoba i nawiązywać z nią interakcji (wypisywać „ludzkich" komunikatów na ekran czy czekać na naciśnięcie klawisza).

W szczególności do odczytania czegoś ze standardowego wejścia w języku C można wykorzystać funkcje `scanf()`, `getchar()` lub `read(0,...)`, zaś do wypisania czegoś na standardowe wyjście funkcje `printf()`, `putchar()` lub `write(1,...)`. Nie należy mylić standardowego wejścia z parametrami przekazywanymi do programu (w języku C jest to tablica `argv[]`).

Bardziej precyzyjna odpowiedź na to pytanie brzmi: standardowe wejście/wyjście to strumienie skojarzone z zerowym i pierwszym deskryptorem plików.

Jak najlepiej wczytywać dane?

To wbrew pozorom bardzo istotna kwestia, bo w przypadku niektórych problemów czytanie danych może zajmować znaczący procent czasu wykonania całego programu.

- Mniej doświadczonym programistom piszącym w C++ radzę korzystać z funkcji `scanf()` i `printf()` zamiast ze strumieni `cin` / `cout`. *Umiejętne* korzystanie ze strumieni *może* przyspieszyć czytanie wejścia o 10–20% w stosunku do `scanf()` / `printf()`, ale korzystanie z nich bez doświadczenia często kończy się na programie, który działa 10 razy wolniej. W przypadku korzystania ze strumieni `cin` / `cout`, należy zwrócić uwagę na następujące rzeczy.
  - Nie należy mieszać strumienia `cin` z wywołaniami `scanf()` ani strumienia `cout` z wywołaniami `printf()`. Dodatkowo jeśli tego nie robimy, należy wyłączyć synchronizację strumieni ze `stdio` poleceniem `std::ios::sync_with_stdio(false)`.
  - Należy w pełni wykorzystać buforowanie i nie wypisywać za często zawartości buforów związanych z `cout` i `stdio`. Nawet jeśli w programie nigdzie nie mamy polecenia

`flush(cout)`, to taka operacja jest wykonywana niejawnie za każdym razem kiedy wykonujemy `cout « endl`. Takie wywołania warto zamienić na `cout « "\n"`.

  * Próba odczytu `cin` powoduje również wypisanie buforów związanych z `cout`; można temu zapobiec umieszczając na początku programu polecenie `cin.tie(NULL)`.
  * Jeśli nie wyłączyliśmy synchronizacji strumieni ze `stdio`, to próba odczytu `cin` spowoduje również wypisanie buforów związanych ze `stdio`.

- Jeśli w pojedynczym wierszu mamy do wczytania dużo znaków, to wczytywanie każdego osobno przez `scanf("%c", &znak)` czy `cin » znak` jest również bardzo nieefektywne. Znacznie szybciej jest wczytać ten wiersz funkcją `fgets()`.
- Dane wejściowe będą spełniać specyfikację zadania i nie trzeba tego sprawdzać.
- Wolno wypisywać część wyniku przed przeczytaniem całości danych wejściowych. W przypadku, w którym dane wejściowe nie mieszczą się w całości w pamięci, jest to nawet konieczne.

Ile pamięci zużywa mój program?

Obliczenie zajmowanej przez program pamięci i dbanie o to, żeby nie została ona przekroczona dla żadnego testu jest częścią zadania. Poniżej zamieszczam trochę informacji na temat tego jak ją oszacować. Ustalany przez sprawdzaczkę limit dotyczy *całej* dostępnej dla procesu pamięci wirtualnej, w skład której wchodzą:

- Binarny kod programu. Ze względu na to, że kod jest statycznie skonsolidowany z bibliotekami, pusty program w C/C++ zajmuje ok. 1050 KB; w przypadku innych języków jest to zazwyczaj więcej.
- Zmienne statyczne (te zdefiniowane poza funkcjami w C/C++). Zazwyczaj proste sumowanie zajętości poszczególnych zmiennych wystarcza; wyjątkiem są struktury, przy których należy wziąć pod uwagę efekty związane z wyrównywaniem danych (http://en.wikipedia.org/wiki/Data_structure_alignment).
- Stos. Zapisywane są na nim między innymi zmienne lokalne wywoływanych funkcji, parametry wywołania funkcji i adres powrotu z funkcji (http://en.wikipedia.org/wiki/Call_stack). Na jego objętość należy uważać zwłaszcza w przypadku funkcji rekurencyjnych. Nie ma dodatkowego limitu na stos (standardowy linuksowy limit wynoszący 8 MB jest wyłączany przez sprawdzaczkę).
- Sterta. Pamięć rezerwowana dynamicznie (zazwyczaj za pomocą funkcji `new` i `malloc`). Warto pamiętać, że funkcje te potrzebują dodatkowej pamięci na swoje struktury. W szczególności ich obecna implementacja w systemie 64-bitowym powoduje, że wywołanie `malloc(`n`)` rezerwuje $n + 8$ bajtów zaokrąglane w górę do wielokrotności 16 bajtów, przy czym minimalną liczbą rezerwowanych bajtów jest 32. Zaokrąglanie w przypadku operatora `new` jest takie samo.

Mój program dostał odpowiedź Runtime Error (RE)

Oznacza to, że na konkretnym teście program zakończył się z kodem wyjścia różnym od zera. Są dwa możliwe powody takiej sytuacji.

- Program został zakończony przez sygnał. Statystycznie najczęściej przytrafiające się sygnały to:

  - `SIGKILL`: otrzymuje go program, którego zmienne statyczne (w C/C++ te definiowane poza funkcjami) wraz z kodem programu nie mieszczą się w pamięciowym limicie zadania.
  - `SIGSEGV`: naruszenie ochrony pamięci; wysyłany kiedy program zapisuje (lub odczytuje) cudzą (lub niezarezerwowaną) pamięć. Zazwyczaj występuje, jeśli w C/C++ zaczniemy zapisywać dane poza zakresem tablicy. Inną częstą możliwością jest przekroczenie limitu pamięci na stercie. W tym przypadku próba rezerwowania pamięci funkcją `malloc` (lub operatorem `new`) zwraca pusty wskaźnik i jeśli tego nie sprawdzimy, to zaczniemy zapisywać dane pod tym pustym wskaźnikiem.
  - `SIGABRT`: najczęściej występuje jeśli program w C++ nie przechwycił powstałego wyjątku (przykładowo jeśli w STL nie udało się zarezerwować pamięci).

  Innymi słowy: praktycznie we wszystkich przypadkach taki błąd oznacza, że albo zabrakło nam pamięci albo piszemy po pamięci, która nie należy do naszego procesu.

- Program explicite zwrócił niezerową wartość, np. w programie w C/C++ wykonanie funkcji `main()` zakończyło się instrukcją `return 1`.

Mój program dostał odpowiedź Illegal Operation (IO)

IO oznacza próbę wykonania niedozwolonej funkcji jądra. Oznacza to, że albo usiłujesz celowo przechytrzyć sprawdzaczkę albo robisz to nieświadomie. W tym drugim przypadku prawdopodobnie jest to spowodowane wywołaniem `system("pause")`, co usiłuje stworzyć nowy proces.

Jeśli chcesz korzystać z wywołania `system("pause")` na windowsowym kompilatorze w domu i nie kasować go przy każdym wysyłaniu rozwiązania do sprawdzaczki, użyj następującej konstrukcji:

```
#ifndef SPRAWDZACZKA
system("pause");
#endif
```

Jeśli nie próbujesz zrobić niczego dziwnego i nie wywołujesz dodatkowego procesu (patrz wyżej) a mimo to otrzymujesz odpowiedź `IO`, zgłoś to prowadzącemu pracownię.

Mój program działa w domu, a na sprawdzaczce nie działa dla testów jawnych

Wykonaj następujące kroki:

- Sprawdź, czy kompilacja na sprawdzaczce generuje jakiekolwiek ostrzeżenia i pozbądź się ich.
- Sprawdź, czy program działa poprawnie na jakimś komputerze z zainstalowanym Linuksem. Być może w domu używasz kompilatora niezgodnego ze standardem, np. starego Visual C++. Koniecznie użyj do kompilacji tych samych opcji, ze szczególnym uwzględnieniem opcji `-O2`. Spróbuj uruchomić program na tej samej wersji kompilatora co na sprawdzaczce.
- Uruchom program parokrotnie i sprawdź, czy zawsze otrzymujesz takie same odpowiedzi. Być może Twój program zachowuje się niedeterministycznie, gdyż np. korzysta z niezainicjowanych zmiennych.
- Przed uruchomieniem programu ogranicz pamięć dla procesu za pomocą polecenia powłoki `ulimit -v` *`limit_w_KB`* i uruchom program na możliwie największych (dopuszczalnych przez treść) danych.

- Uruchom program za pomocą `valgrind -tool=memcheck ./twoj_program < input` i przeczytaj uważnie wszystkie ostrzeżenia ze szczególnym uwzględnieniem komunikatów dotyczących korzystania z niezarezerwowanej pamięci lub niezainicjowanych zmiennych. Ten sposób znacząco spowalnia wykonanie programu, więc warto stosować go na małych plikach wejściowych. Żeby uniknąć fałszywych ostrzeżeń związanych z biblioteką standardową warto skompilować program bez opcji `-static`. Dodatkowo jeśli skompilujesz swój program z opcją `-g` dostaniesz precyzyjniejsze informacje na temat miejsca potencjalnych błędów.

Jeśli żadna z powyższych rad nie pozwala na wyeliminowanie błędu, skontaktuj się z prowadzącym pracownię.

Jak ustalane są limity czasowe dla poszczególnych testów?

Czas zostaje tak dobrany, żeby dopuścić rozwiązania o nieco gorszych stałych niż optymalne, ale — przynajmniej na niektórych testach — odrzucić rozwiązania, które korzystają z asymptotycznie nieoptymalnych algorytmów.

W tym celu na sprawdzaczce uruchamiany jest program wzorcowy. Program wzorcowy jest napisany w C/C++ i zazwyczaj ma asymptotycznie optymalną złożoność. Jeśli wejście jest ciągiem liczb, to wczytywane są one funkcją `scanf()`. Długie wiersze napisów wczytywane są funkcją `fgets()`. Limity to zazwyczaj czasy uzyskane przez program wzorcowy przemnożone przez ok. 3–4.

Jak ściśle należy trzymać się warunków określonych w zadaniu?

Przede wszystkim należy pamiętać, że programy będą sprawdzane przez bezduszną maszynę. Niedopuszczalne jest na przykład oczekiwanie na naciśnięcie klawisza czy wypisywanie zbędnych informacji na ekran. W przypadku, kiedy istnieje tylko jedna dobra odpowiedź, dopuszczalne jest drobne odchylenie w niektórych białych znakach: w takim przypadku wynik działania programu będzie porównywany z wzorcowym za pomocą polecenia `diff -b -B`.

Marcin Bieńkowski