# Pracownia z Algorytmów i struktur danych

## Zasady zaliczania i szczegóły techniczne oceniania

## (semestr letni 2024)

## 1 Zasady zaliczania

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: https://aisd.ii.uni.wroc.pl/. Prowadzący pracownię będzie też informować o nowych zadaniach na kanale *general* w zespole związanym z wykładem z AiSD w MS Teams. Na rozwiązanie każdego zadanie będzie co najmniej 2 tygodnie, zazwyczaj około 3–4 tygodni. Podstawą zaliczenia pracowni jest liczba punktów zdobytych za rozwiązywanie powyższych zadań.

1. 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*.

2. 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 dodatkowymi warunkami podanymi w treści (takimi jak zabronienie używania konkretnych bibliotek lub plików nagłówkowych).

3. Niech $P_M \in [0, 60]$ będzie sumą punktów, które student(ka) uzyskał(a) za wszystkie zadania, natomiast $P_L \in [0, 40]$ będzie sumą punktów z pominięciem dwóch zadań, za które student(ka) uzyskał(a) najmniejszą liczbę punktów. Następnie wyliczamy:

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

   Warto zauważyć, że $-15 \leqslant Y_M \leqslant Y_L \leqslant 15$.

   a) Ocena za pracownię wystawiana jest na podstawie poniższej tabeli, gdzie przypadki graniczne rozstrzygane są na korzyść studenta(-ki).

| $Y_L$ | -15–0 | 0–3 | 3–6 | 6–9 | 9–12 | 12–15 |
|---|---|---|---|---|---|---|
| ocena | 2,0 | 3,0 | 3,5 | 4,0 | 4,5 | 5,0 |

   b) Jeśli $Y_L \geqslant 0$, student(ka) ma prawo podejścia do egzaminu na poziomie licencjackim, otrzymując $Y_L$ punktów egzaminacyjnych wliczanych do tego egzaminu.

   c) Jeśli $Y_M \geqslant 0$, student(ka) ma prawo podejścia do egzaminu na poziomie magisterskim, otrzymując $Y_M$ punktów egzaminacyjnych wliczanych do tego egzaminu.

4. Kod oddawanego programu musi być tworzony w całości samodzielnie.

   (a) W przypadku niesamodzielnego rozwiązania (różne osoby oddają takie same lub podobne rozwiązania), 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.

   (b) W przypadku oddania cudzego rozwiązania (również przerobionego), minimalną karą jest anulowanie punktów za to zadanie i dodatkowe zmniejszenie wartości $Y_L$ i $Y_M$ o 6.

   Warto wiedzieć, że wszystkie programy będą automatycznie porównywane ze sobą a także z rozwiązaniami dostępnymi w Internecie.

5. Osoby, które zaliczyły pracownię w latach ubiegłych mogą brać w niej udział bez zapisywania się na nią w USOS-ie. W takim wypadku do egzaminu na poziomie licencjackim wlicza się maksimum wartości $Y_L$ uzyskanych w roku bieżącym i latach poprzednich, zaś do egzaminu na poziomie magisterskim analogiczne maksimum wartości $Y_M$.

## 2 Techniczne szczegóły oceniania

1. Dla każdego zadania zdefiniowane zostanie 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 zostanie skompilowany i w przypadku błędu kompilacji zostanie on odrzucony. W przeciwnym przypadku program zostanie sprawdzony na testach jawnych i pół-jawnych. Student(ka) zostaje powiadomiony(-a) o wynikach tych testów.

3. Do upływu terminu oddawania zadania, można wysyłać rozwiązanie dowolną liczbę razy. Po upływie terminu oddawania zadania, sprawdzaczka przestaje akceptować kolejne rozwiązania.[1] **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 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.

   Za każdy pomyślnie wykonany test pół-jawny lub tajny student(ka) otrzymuje 1 punkt. Za testy jawne student(ka) nie otrzymuje żadnych punktów. Niekompilujący się program otrzyma zero punktów.

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

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, kompilator `gcc 12.2.0`
     `gcc -std=gnu18 -Wall -Wextra -Wshadow -O2 -static -DSPRAWDZACZKA -lm`
   - C++, kompilator `gcc 12.2.0`
     `g++ -std=gnu++17 -Wall -Wextra -Wshadow -O2 -static -DSPRAWDZACZKA`
   - OCaml, kompilator `OCaml native-code compiler 4.13.1`
     `ocamlopt -ccopt -static -w A`
   - Rust, kompilator `rustc 1.63.0`
     `rustc -edition=2021 -C opt-level=2 -C target-feature=+crt-static`

# 3 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 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[]`).

### 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 chcemy wczytać cały wiersz, to wykorzystanie funkcji `fgets()` będzie znacznie szybsze niż wczytywanie po znaku funkcjami `scanf("%c", &znak)` czy `cin >> znak`.
- Dane wejściowe będą spełniać specyfikację zadania i nie trzeba tego sprawdzać.
- Można 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.

- Skompiluj swój program z opcjami `-fsanitize=address,undefined` i uruchom go. Jeśli skompilujesz swój program z opcją `-g`, dostaniesz precyzyjniejsze informacje na temat miejsca potencjalnych błędów.
- 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`.

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?

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*