Katedra
Informatyki i Automatyki
Politechniki Rzeszowskiej

Informatyka

Notatka odnośnie C++ i Hackerrank

Opracował: Maciej Penar

Spis treści

Hackerrank

Hackerrank (link) to taki SPOJ (link), tylko ma ładniejszy interfejs użytkownika.

Ogólna idea jest taka że wpisujecie kod online i wysyłacie na serwer. Serwer wykonuje kod i wyświetla Wam czy udało się wam rozwiązać problem.

JAK TO DZIAŁA?

Wybierzmy pierwszy problem algorytmiczny (poniżej) link. Gdy wybierzemy język C++ - okno **Current Buffer** zostanie uzupełnione 1) kodem który wczytuje dane w pseudo-magiczny sposób 2) funkcją którą musimy uzupełnić zgodnie z instrukcją.

Całe zadanie polega na zapoznaniu się z problemem (w zakładce **PROBLEM**). Dla pierwszego problemu algorytmicznego (Sumowania liczba) zadanie sprowadza się do uzupełnienia linii 10-15:

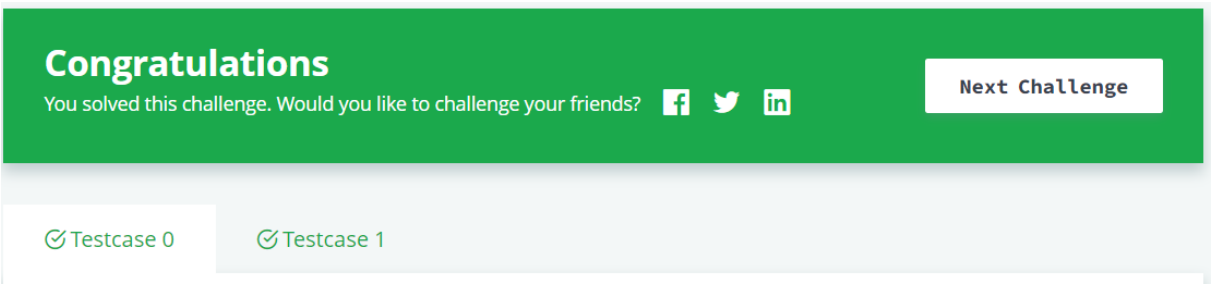
```
int simpleArraySum(vector<int> ar) {
   /*
    * Write your code here.
    */

}
```

Można się przestraszyć typu **vector<int> -** komentarz odnośnie tego typu jest w sekcji komentarzy co C++. Wybrane problemy których wykonanie przygotowuje do laboratoriów 3 są poniżej.

Jak uzupełnicie kod, to możecie go uruchomić via: **Run Code –** na łatwych przypadkach testowych.

Jak jesteście pewni swojego rozwiązania możecie uruchomić kod via: **Submit Code –** co spowoduje wybranie bardzo trudnych przypadków testowych (Akurat nie dla problemów które ja wybrałem) – dostaniecie informacje zwrotną ile przypadków udało się zaliczyć.

Gdy zobaczycie:



Wasze serca napełnią się dumą i satysfakcją – po czym wybieracie kolejne zadanko.

PROBLEMY "TUTORIALOWE" (OBOWIĄZYWAŁY DO DRUGIEGO LABORATORIUM):

1. Hello world: https://www.hackerrank.com/challenges/cpp-hello-world/problem
2. Typy danych: https://www.hackerrank.com/challenges/c-tutorial-basic-data-types/problem
3. Instrukcje warunkowe: https://www.hackerrank.com/challenges/c-tutorial-conditional-if-else/problem
4. Pętla for: https://www.hackerrank.com/challenges/c-tutorial-for-loop/problem
5. Funkcje: https://www.hackerrank.com/challenges/c-tutorial-functions/problem
6. Wskaźniki: https://www.hackerrank.com/challenges/c-tutorial-pointer/problem
7. Tablice: https://www.hackerrank.com/challenges/arrays-introduction/problem
8. Ciągi znaków: https://www.hackerrank.com/challenges/c-tutorial-strings/problem
9. Struktury: https://www.hackerrank.com/challenges/c-tutorial-struct/problem

PROBLEMY TUTORIALOWE (OBOWIĄZUJĄ NA TRZECIM LABORATORIUM):

1. Klasy: https://www.hackerrank.com/challenges/c-tutorial-class/problem
2. Zadanie odwrotne do 10): https://www.hackerrank.com/challenges/classes-objects/problem
3. Vector: https://www.hackerrank.com/challenges/variable-sized-arrays/problem

PROBLEMY ALGORYTMICZNE:

1. Sumowanie (Easy): https://www.hackerrank.com/challenges/simple-array-sum/problem
   Sumowanie 2 (Very Easy): https://www.hackerrank.com/challenges/a-very-big-sum/problem
2. Porównywanie trójek (Easy): https://www.hackerrank.com/challenges/compare-the-triplets/problem
3. Świeczki (Easy): https://www.hackerrank.com/challenges/birthday-cake-candles/problem
4. Parzyste pary (Easy): https://www.hackerrank.com/challenges/divisible-sum-pairs/problem
5. Koty i Myszy (Easy+): https://www.hackerrank.com/challenges/cats-and-a-mouse/problem
6. Leaderboard (Medium): https://www.hackerrank.com/challenges/climbing-the-leaderboard/problem

C++

Tu podaje klika przydatnych informacji, jeszcze więcej nowej składni, ale mam nadzieję że uprości Wam pisanie kodu.

NOWA SKŁADOWA STRUKTUR

Do tej pory struktury przechowywały jedynie dane np.: *struct Student { int age; string name; }*

Okazuje się że istnieje wiele innych elementów które możemy zdefiniować wewnątrz struktur. Wszystkie nowe elementy będą wpływać na zachowanie obiektów. Nowym elementem składowym są:

Metody – czyli funkcje wewnątrz definicji struktury. Metody są wywoływane na rzecz obiektów struktury/klasy. (Zaawansowany komentarz: Oznacza to że przekazywany jest niejawnie dodatkowy parametr – referencja na obiekt który wywołuje metodę)

Przykład: załóżmy że nasz student potrafi się przywitać za pomocą metody **void *hello*().** Metodę wywołujemy na rzecz zmiennej typu Student za pomocą notacji kropkowej.

```
struct Student {
	int age;
	std::string name;
	void hello() { std::cout << "Hai! My name is: " << name << std::endl; }
};

int main()
{
	Student s = { 18, "MP" };
	Student s2 = { 15, "GB" };

	s.hello();
	s2.hello();
}
```

Ten program wypisze:

```
Hai! My name is: MP
Hai! My name is: GB
```

Ponieważ wywołania metod są na rzecz obiektów (tutaj reprezentowanych przez zmienne **s** i **s2**). Metody mają swobodny dostęp do składowych struktur (tu age i name), dzięki czemu podobnie wyglądające wywołania nie mają takiego samego rezultatu.

KLASA VECTOR

Załóżmy że mamy strukturę **Student**. Załóżmy też że w naszym programie chcemy wczytać nieznaną liczbę studentów. Myślimy sobie: „Wczytam najpierw ile studentów będzie użytkownik potrzebował i utworzę tablicę o takiej długości”. Brzmi jak dobry pomysł. Piszemy kod:

```
struct Student {};
int main()
{
    int length;
    std::cin >> length;
    Student arr[length];
    return 0;
}
```

I dostajemy po twarzy od środowiska czerwonym podkreśleniem i komunikatem *„wymaga stała, a nie zmienna”*.

I tu z pomocą przychodzi klasa **vector** (wymagająca #include <vector>) która stanowi implementację tablicy o **dowolnej** długości. Istotne elementy tej klasy to:

- Typ **T** przechowywany w wektorze – specyfikujemy go w nawiasach kwadratowych: vector<**T**> np.
  vector<int> - przechowuje liczby całkowite
  vector<Student> przechowuje studentów
- Metoda **push_back(T)** umieszczająca element typu T na koniec tablicy
- Indeksator [numer] pobierający element jak w tablicy
- Metoda size() – pobierająca ile elementów jest w wektorze

Przykład zapisu i odczytu z wektora:

```
#include <vector>

struct Student {};

int main()
{
	std::vector<Student> vec;
	for (int i = 0; i < 100; ++i) {
		vec.push_back(Student()); // Dodanie elementu na koniec
	}

	Student s = vec[50]; // Pobranie za pomocą indeksatora
	return 0;
}
```

MAŁY HAX DOT. PRZETWARZANIA ILUŚTAM ELEMENTÓW W PĘTLI

Zazwyczaj kod przetwarzający wszystkie elementy tablicy i wektora wygląda podobnie – wykorzystywana jest pętla for.

Dla tablicy:

```
int main()
{
	int arr[] = { 1, 5, 12, 23, 2 };
	for (int i = 0; i < sizeof(arr)/sizeof(int); ++i) {

		std::cout << arr[i] << std::endl;
	}
	return 0;
}
```

Dla wektora:

```
int main()
{
	std::vector<int> vec = { 1, 5, 12, 23, 2 };
	for (int i = 0; i < vec.size(); ++i) {
		std::cout << vec[i] << std::endl;
	}
	return 0;
}
```

Z tego względu istnieje specjalna wersja petli for. Tzw. **pętla zakresowa** – wykorzystywana tam gdzie chcemy wczytać wszystkie elementy pewnego **zakresu (**np. tablicy/wektora).

Dla tablicy:

```
int main()
{
	int arr[] = { 1, 5, 12, 23, 2 };
	for (int item : arr) {
		std::cout << item << std::endl;
	}
	return 0;
}
```

Dla wektora:

```
int main()
{
	std::vector<int> vec = { 1, 5, 12, 23, 2 };
	for (int item : vec) {
		std::cout << item << std::endl;
	}
	return 0;
}
```

Ultra mega hax: wiadomo jakie obiekty/wartości przechowywane są w wektorze lub tablicy. Wynika to z deklaracji tablicy np. **int** arr[]; lub z typu przy wektorze np. vector<**Student**> Dlatego można skorzystać z słowa kluczowego auto:

```
int main()
{
	std::vector<int> vec = { 1, 5, 12, 23, 2 };
	for (auto item : vec) {
		std::cout << item << std::endl;
	}
	return 0;
}
```