Implementacja algorytmu Insertion Sort w podejściu TDD

Grzegorz Rozdzialik

23 maja 2021

Streszczenie

Sprawozdanie zadania 2. zawiera omówienie zaimplementowanego algorytmu Insertion Sort w języku Rust w podejściu TDD (ang. Test-Driven Development). Omówiony zostaje algorytm, utworzone testy automatyczne w kolejnych cyklach TDD, a także sposób uruchomienia testów.

Jest to rozwiązanie zadania 2. z przedmiotu *Testowanie i Weryfikacja Oprogramowania* studiach magisterskich OKNO 2020/2021.

Spis treści

Opis algorytmu Insertion Sort

Algorytm Insertion Sort ma na celu posortowanie listy elementów. Realizuje to poprzez podzielenie listy na 2 podlisty:

1. Lista elementów już posortowanych
2. Lista elementów oczekujących na posortowanie

A następnie wybieraniu kolejnych elementów z listy elementów oczekujących na posortowanie i umieszczaniu ich w odpowiednim miejscu w liście elementów posortowanych.

Początkowo lista elementów posortowanych zawiera wyłącznie pierwszy element listy, a pozostałe są uznawane jako nieposortowane.

Szerszy opis oraz przykład działania algorytmu został zaprezentowany na stronie internetowej https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm

Stabilność algorytmu sortowania

W testach sprawdzana będzie również *stabilność* algorytmu sortowania.

Algorytm sortowania jest stabilny jeżeli elementy o tym samym kluczu sortowania są w tej samej kolejności względem siebie w danych wejściowych oraz w posortowanym wyniku.

Przykładowo, dla następujących par liczb `(x, y)`, gdzie `x` jest kluczem sortowania:

`(0, 1), (-1, 1), (0, 5)`

Posortowaniem stabilnym względem klucza `x` będzie jedynie:

`(-1, 1), (0, 1), (0, 5)`

Natomiast poniższy wynik nie będzie wynikiem stabilnego algorytmu sortowania:

`(-1, 1), (0, 5), (0, 1)`

Pomimo, że elementy są posortowane względem klucza `x` (pierwszego elementu z każdej pary), to elementy o kluczu 0 występują w wynikowej liście w odwrotnej kolejności niż w liście wejściowej.

Podejście Test-Driven Development

W podejściu Test-Driven Development (TDD) implementacja algorytmu lub funkcjonalności następuje w kolejno następujących po sobie cyklach składających się z następujących kroków:

1. Napisanie testu dla niezrealizowanej funkcjonalności. Wykonanie tego testu powinno zakończyć się porażką.
2. Implementacja funkcjonalności. Po zakończeniu implementacji nowo napisany test, jak i testy napisane w poprzednich cyklach powinny zakończyć się powodzeniem.
3. Czyszczenie, ulepszenie implementacji (ang. refactoring). Po tym kroku wszystkie testy nadal powinny kończyć się powodzeniem.

Implementacja algorytmu z zadania

Algorytm z zadania zaimplementowano w języku Rust (https://www.rust-lang.org/).

Zaimplementowana funkcja obsługuje nie tylko liczby typu `i32` (jak było to w pierwszym zadaniu), ale dowolne elementy, które można ze sobą porównywać. W praktyce oznacza to, że funkcja `insertion_sort` przyjmuje wektor elementów, które implementują cechę (ang. trait) `Ord`, która pozwala je ze sobą porównać.

Ponadto, zaimplementowana funkcja wykonuje sortowanie stabilne, bo zostało potwierdzone w jednym z testów.

Stworzony projekt jest biblioteką (ang. library) i nie dostarcza aplikacji wykonywalnej (ang. executable binary). Udostępniona została jedynie funkcja `insertion_sort`, która wykonuje sortowanie. Użytkownicy tej biblioteki mogą użyć ją do sortowania w swoich projektach w języku Rust (przykładowym zastosowaniem byłoby stworzenie programu będącego odpowiednikiem komendy `sort` znanej z systemów UNIX).

Do uruchomienia testów użyto standardowej komendy dla języka Rust:

```
cargo test
```

Komenda ta jest dostępna po instalacji narzędzi dla języka Rust: https://www.rust-lang.org/tools/install.

Kod źródłowy projektu został udostępniony w serwisie GitHub: https://github.com/Gelio/tiwo-sorting-tdd. Korzystając z historii zmian można przejrzeć kolejne cykle bezpośrednio w serwisie (nazwy zmian zaczynają się od `(cycle X)`, gdzie `X` jest liczbą całkowitą), a także pobrać repozytorium i uruchomić testy we własnym środowisku.

Interfejs algorytmu sortowania

Funkcja `insertion_sort` zaimplementowana w tym zadaniu posiada następującą sygnaturę:

```rust
/// Sorts the array in-place to be in the ascending order.
pub fn insertion_sort(arr: &mut Vec<impl Ord>) {}
```

Funkcja będzie zmieniała przekazany przez referencję wektor elementów, które implementują cechę `Ord`.

Cykle TDD dla implementacji algorytmu

W ramach implementacji algorytmu Insertion Sort wykonano cykle wymienione poniżej. Pierwsze 3 z nich zostały szczegółowo omówione w sprawozdaniu do pierwszej części zadania.

1. Poprawne posortowanie wektora 2 elementów `[2, 1]`

   Najprostsza możliwa implementacja wykonywała stałe podstawienie pierwszych dwóch elementów wektora aby otrzymać wektor wynikowy `[1, 2]`.

   Funkcja `insertion_sort` dopuszczała sortowanie wyłącznie liczb typu `i32`.

   Nazwa testu: `it_should_swap_two_unsorted_elements`

2. Brak błędów dla pustego wektora `[]`

   Do implementacji dopisano warunek brzegowy dla pustego wektora. W tym przypadku funkcja nie musiała wykonywać żadnych operacji.

   Nazwa testu: `it_should_work_for_empty_vectors`

3. Sprawdzenie wyniku sortowania wektora z kilkoma nieposortowanymi elementami

   Sprawdzono wynik sortowania dla `1, 8, 3, 2, 5, 10, -1`. W tym cyklu zaimplementowano niestabilny algorytm Insertion Sort.

   Nazwa testu: `it_should_sort_a_few_unsorted_elements`

4. Sortowanie dowolnych elementów spełniających cechę `Ord`

   Dodano test dla nowego typu `MyNode` składającego się z 2 elementów:

   ```
   struct MyNode {
       /// The key to sort by
       key: i32,
       metadata: i32,
   }
   ```

   Sortowanie odbywało się na podstawie wartości właściwości `key`.

   W tym cyklu zmodyfikowano implementację i sygnaturę funkcji aby obsługiwała ona elementy z cechą `Ord`. Poprzednie testy nie wymagały zmian, ponieważ typ `i32` implementuję cechę `Ord`.

   W tym cyklu nie była sprawdzana stabilność sortowania. Weryfikowano jedynie czy po posortowaniu właściwości `key` są uszeregowane niemalejąco.

   Nazwa testu: `it_should_sort_any_vector_which_implements_ord`

5. Sprawdzenie stabilności sortowania

   Dodano test weryfikujący kolejność całych obiektów `MyNode`. Należało także zmodyfikować implementację algorytmu aby była stabilna (zmiana operatora `>=` na `>` podczas porównywania elementów w celu wyboru indeksu do wstawienia elementu.

   Nazwa testu: `it_should_perform_stable_sorting`

6. Sprawdzenie wyniku sortowania dla dużego wektora z elementami w kolejności malejącej

   Sprawdzono czy wektor o elementach `100000, 99999, 99998, ..., 1, 0` po wywołaniu funkcji `insertion_sort` będzie posortowany rosnąco.

   Test przechodził bez konieczności modyfikacji funkcji.

   Nazwa testu: `it_should_work_for_a_large_vector_of_reverse_sorted_numbers`

7. Sprawdzenie wyniku sortowania dla wektora z jednym elementem `[1]`

   Test przechodził bez konieczności modyfikacji funkcji.

   Nazwa testu: `it_should_work_for_a_single_element`

8. Sprawdzenie wyniku sortowania dla już posortowanego wektora

   Sprawdzono czy wektor o elementach `0, 1, ..., 999, 1000` po wywołaniu funkcji `insertion_sort` będzie nadal posortowany rosnąco.

   Test przechodził bez konieczności modyfikacji funkcji.

   Nazwa testu: `it_should_not_do_anything_with_an_already_sorted_vector`

9. Sprawdzenie wyniku sortowania dla tego samego elementu powtórzonego wiele razy

   Sprawdzono czy wektor składający się z liczby 1 powtórzonej 1000 razy po wywołaniu funkcji `insertion_sort` będzie niezmieniony.

   Test przechodził bez konieczności modyfikacji funkcji.

   Nazwa testu: `it_should_not_do_anything_with_a_vector_of_the_same_number`

Implementacja funkcji `insertion_sort`

Wynikowa kod funkcji `insertion_sort` po przeprowadzeniu wszystkich wymienionych cykli TDD jest przedstawiony poniżej:

```rust
/// Sorts the array in-place to be in the ascending order.
pub fn insertion_sort(arr: &mut Vec<impl Ord>) {
    for index_to_sort in 1..arr.len() {
        let v = &arr[index_to_sort];

        (0..index_to_sort)
            .find(|&i| arr[i].gt(v))
            .and_then(|index_to_insert_at| {
                arr[index_to_insert_at..=index_to_sort].rotate_right(1);

                Some(())
            });
    }
}
```

Wynik wykonania testów

Wynik uruchomienia komendy `cargo test` po przeprowadzeniu wszystkich wymienionych cykli TDD:

```
10:54 $ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 9 tests
test tests::it_should_perform_stable_sorting ... ok
test tests::it_should_sort_any_vector_which_implements_ord ... ok
test tests::it_should_sort_a_few_unsorted_elements ... ok
test tests::it_should_swap_two_unsorted_elements ... ok
test tests::it_should_work_for_a_single_element ... ok
test tests::it_should_work_for_empty_vectors ... ok
test tests::it_should_not_do_anything_with_a_vector_of_the_same_number ... ok
test tests::it_should_not_do_anything_with_an_already_sorted_vector ... ok
test tests::it_should_work_for_a_large_vector_of_reverse_sorted_numbers ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.94s
```

Wszystkie 9 testów zakończyło się powodzeniem, a ich wykonanie trwało 0.94s.

Odniesienie wymaganych funkcji z treści zadania

W treści zadania 2. widnieje informacja, że jednymi z funkcji przygotowanego programu mają być 2 przykłady działania algorytmu: prosty oraz bardziej skomplikowany.

W przypadku tego projektu przykładem prostym może być ten z cyklu TDD numer 3 (*Sprawdzenie wyniku sortowania wektora z kilkoma nieposortowanymi elementami*), który sprawdza wynikowe posortowanie dla wektora początkowego `1, 8, 3, 2, 5, 10, -1`.

Przykładami bardziej zaawansowanymi mogą być w tym przypadku:

- cykl numer 4 (*Sortowanie dowolnych elementów spełniających cechę* `Ord`)
- cykl numer 5 (*Sprawdzenie stabilności sortowania*)
- cykl numer 6 (*Sprawdzenie wyniku sortowania dla dużego wektora z elementami w kolejności malejącej*)

Pierwsze dwa z nich sprawdzają dodatkowe cechy, które nie były wymagane w podstawowej wersji algorytmu, gdy sortował on tylko liczby.

Ostatni przykład działa na dużej liczbie danych (100000 elementów) oraz jest to przykład, w którym każdy element musi zmienić swoje miejsce.

Testy oprogramowania wraz z ich tekstowym podsumowaniem uruchamiane są komendą `cargo test`.

Funkcja "*Powrót do systemu*" nie jest wymagana, ponieważ projekt nie jest aplikacją konsolową, tylko biblioteką.

Bibliografia

1. Opis algorytmu Insertion Sort
   https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm
2. Slajdy do przedmiotu *Testowanie i Weryfikacja Oprogramowania* na studiach magisterskich OKNO
3. Dokumentacja pisania testów w języku Rust https://doc.rust-lang.org/book/ch11-01-writing-tests.html
4. *Clean Code - Uncle Bob / Lesson 4* (sekcja o TDD rozpoczyna się około 21:41) https://youtu.be/58jGpV2Cg50?t=1300
5. *Klasyfikacja algorytmów sortowania* - Wikipedia https://pl.wikipedia.org/wiki/Sortowanie#Klasyfikacja