Implementacja algorytmu Insertion Sort w podejściu TDD

Grzegorz Rozdzialik

22 maja 2021

Streszczenie

Sprawozdanie zadania 1. zawiera pierwsze 3 cykle podejścia TDD (ang. Test-Driven Development) dla implementacji algorytmu Insertion Sort w języku Rust.

Jest to rozwiązanie zadania 1. 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

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/).

Ustalono, że docelowa funkcja powinna zawsze sortować liczby typu `i32` niemalejąco.

Do uruchomienia testów użyto standardowej komendy dla języka Rust:

```
cargo test
```

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.

W kolejnych sekcjach zostaną zaprezentowane kroki wykonane dla pierwszych 3 cykli implementacji algorytmu w podejściu TDD.

Początkowa struktura

Na początku przygotowano strukturę kodu, która umożliwi uruchomienie testów w przyszłości. Początkowy kod źródłowy zawierający pustą implementację funkcji `insertion_sort` oraz pusty moduł z testami został przedstawiony poniżej:

```rust
/// Sorts the array in-place to be in the ascending order.
pub fn insertion_sort(_arr: &mut Vec<i32>) {}

#[cfg(test)]
mod tests {}
```

Ten kod będzie rozszerzany w kolejnych cyklach.

Cykl 1

Pierwszy cykl TDD rozpoczęto od dodania testu weryfikującego czy algorytm poprawnie posortuje wektor (tablicę) składającą się z 2 elementów `vec![2, 1]`.

Dodanie testu

Moduł z testami wyglądał następująco:

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_should_swap_two_unsorted_elements() {
        let mut arr = vec![2, 1];

        insertion_sort(&mut arr);
```

```
        assert_eq!(arr, vec![1, 2]);
    }
}
```

Wynik uruchomienia testów komendą `cargo test`:

```
20:06 $ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 1 test
test tests::it_should_swap_two_unsorted_elements ... FAILED

failures:

---- tests::it_should_swap_two_unsorted_elements stdout ----
thread 'tests::it_should_swap_two_unsorted_elements' panicked at 'assertion failed: `(left == right)`
  left: `[2, 1]`,
 right: `[1, 2]`', src/lib.rs:14:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_should_swap_two_unsorted_elements

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Z powodu braku działań wykonanych przez funkcję `insertion_sort`, tablica pozostała nieposortowana, a test oczekiwanie zakończył się porażką.

Implementacja funkcjonalności

W celu spełnienia warunków oczekiwanych przez test zmieniono implementację funkcji `insertion_sort` na następującą (linie zaczynające się od minusa to linie usunięte, a linie zaczynające się od plusa to linie dodane, jak w formacie komendy `git diff`):

```
-pub fn insertion_sort(_arr: &mut Vec<i32>) {}
+pub fn insertion_sort(arr: &mut Vec<i32>) {
+    arr[0] = 1;
+    arr[1] = 2;
+}
```

Ta implementacja, mimo że nieprawidłowa z punktu widzenia implementacji algorytmu sprawiła, że napisany dotychczas test zakończył się sukcesem:

```
20:16 $ cargo test
   Compiling tiwo-sorting-tdd v0.1.0 (/home/grzegorz/projects/personal/tiwo-sorting-tdd/cycles)
    Finished test [unoptimized + debuginfo] target(s) in 0.45s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 1 test
test tests::it_should_swap_two_unsorted_elements ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Kod funkcji `insertion_sort` wydaje się zrozumiały i nie był potrzebny krok związany z wprowadzaniem poprawek.

Cykl 2

W drugim cyklu dodano test sprawdzający zachowanie algorytmu dla pustej tablicy – tablica powinna zostać niezmieniona oraz nie powinno być żadnych błędów (`panic` w języku Rust).

Dodanie testu

Do modułu z testmi dodano następujący test:

```
#[test]
fn it_should_work_for_empty_vectors() {
    let mut arr = vec![];

    insertion_sort(&mut arr);

    assert_eq!(arr, vec![]);
}
```

Wynik uruchomienia `cargo test` po dodaniu testu:

```
20:27 $ cargo test
   Compiling tiwo-sorting-tdd v0.1.0 (/home/grzegorz/projects/personal/tiwo-sorting-tdd/cycles)
    Finished test [unoptimized + debuginfo] target(s) in 0.46s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 2 tests
test tests::it_should_swap_two_unsorted_elements ... ok
test tests::it_should_work_for_empty_vectors ... FAILED

failures:

---- tests::it_should_work_for_empty_vectors stdout ----
thread 'tests::it_should_work_for_empty_vectors' panicked at 'index out of bounds:
            the len is 0 but the index is 0', src/lib.rs:3:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_should_work_for_empty_vectors

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Oczekiwanie, nowy test kończy się niepowodzeniem, ponieważ implementacja `insertion_sort` stara się dostać do indeksu 0 tablicy, a tablica ma długość 0, co kończy się paniką programu (`panic`).

Implementacja funkcjonalności

Prostym zabiegiem, który sprawi, że drugi dodany test będzie kończył się powodzeniem jest dodanie specjalnego warunku dla przypadku gdy tablica ma długość 0. Wtedy algorytm nic nie musi robić. Zmienione linie:

```
 /// Sorts the array in-place to be in the ascending order.
 pub fn insertion_sort(arr: &mut Vec<i32>) {
+    if arr.is_empty() {
+        return;
+    }
+
     arr[0] = 1;
     arr[1] = 2;
 }
```

Po dodaniu tego warunku brzegowego, wszystkie testy ponownie przechodzą z powodzeniem:

```
20:30 $ cargo test
   Compiling tiwo-sorting-tdd v0.1.0 (/home/grzegorz/projects/personal/tiwo-sorting-tdd/cycles)
    Finished test [unoptimized + debuginfo] target(s) in 0.45s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 2 tests
test tests::it_should_swap_two_unsorted_elements ... ok
test tests::it_should_work_for_empty_vectors ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Kod funkcji `insertion_sort` wydaje się zrozumiały i nie był potrzebny krok związany z wprowadzaniem poprawek.

Cykl 3

W tym cyklu postanowiono dodać bardziej skomplikowany przypadek, który wymusiłby pełne zaimplementowanie algorytmu Insertion Sort.

Dodanie testu

Test dodany w trzecim cyklu to weryfikacja czy tablica składająca się z 7 nieposortowanych liczb zostanie prawidłowo posortowana:

```rust
#[test]
fn it_should_sort_a_few_unsorted_elements() {
    let mut arr = vec![1, 8, 3, 2, 5, 10, -1];

    insertion_sort(&mut arr);

    assert_eq!(arr, vec![-1, 1, 2, 3, 5, 8, 10]);
}
```

Po dodaniu testu i uruchomieniu `cargo test`, nowo dodany test zakończył się niepowodzeniem, ponieważ aktualna implementacja zawsze wstawia bardzo konkretne wartości (1, 2) jako pierwsze dwa elementy tablicy:

```
20:34 $ cargo test
   Compiling tiwo-sorting-tdd v0.1.0 (/home/grzegorz/projects/personal/tiwo-sorting-tdd/cycles)
    Finished test [unoptimized + debuginfo] target(s) in 0.45s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 3 tests
test tests::it_should_swap_two_unsorted_elements ... ok
test tests::it_should_work_for_empty_vectors ... ok
test tests::it_should_sort_a_few_unsorted_elements ... FAILED

failures:

---- tests::it_should_sort_a_few_unsorted_elements stdout ----
thread 'tests::it_should_sort_a_few_unsorted_elements' panicked at 'assertion failed: `(left == right)`
  left: `[1, 2, 3, 2, 5, 10, -1]`,
 right: `[-1, 1, 2, 3, 5, 8, 10]`', src/lib.rs:39:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_should_sort_a_few_unsorted_elements

test result: FAILED. 2 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Jak widać, wartość `left` zawiera wejściową tablicę z niepoprawnie zmienionymi dwoma początkowymi elementami. Należy to w tym cyklu poprawić.

Implementacja funkcjonalności

Sprawienie, żeby test dodany w tym cyklu kończył się powodzeniem wymagało zaimplementowania pełnego algorytmu Insertion Sort.

```
 /// Sorts the array in-place to be in the ascending order.
 pub fn insertion_sort(arr: &mut Vec<i32>) {
-    if arr.is_empty() {
-        return;
-    }
-
-    arr[0] = 1;
-    arr[1] = 2;
+    for index_to_sort in 1..arr.len() {
+        let v = arr[index_to_sort];
+
+        (0..index_to_sort)
+            .find(|&i| arr[i] >= v)
+            .and_then(|index_to_insert_at| {
+                arr[index_to_insert_at..=index_to_sort].rotate_right(1);
+
+                Some(())
+            });
+    }
 }
```

Zatem ostateczna wersja funkcji `insertion_sort` wygląda następująco:

```
/// Sorts the array in-place to be in the ascending order.
pub fn insertion_sort(arr: &mut Vec<i32>) {
    for index_to_sort in 1..arr.len() {
        let v = arr[index_to_sort];

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

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

Taka implementacja to pełna, poprawna implementacja algorytmu. Po uruchomieniu testów wszystkie 3 testy kończą się powodzeniem:

```
20:39 $ cargo test
   Compiling tiwo-sorting-tdd v0.1.0 (/home/grzegorz/projects/personal/tiwo-sorting-tdd/cycles)
    Finished test [unoptimized + debuginfo] target(s) in 0.48s
     Running unittests (target/debug/deps/tiwo_sorting_tdd-69ffa5c2f7f2f198)

running 3 tests
test tests::it_should_work_for_empty_vectors ... ok
test tests::it_should_sort_a_few_unsorted_elements ... ok
test tests::it_should_swap_two_unsorted_elements ... ok
```

```
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Kolejne kroki

W kolejnych krokach (a więc w zadaniu 2.) należy dodać więcej przykładów testowych aby zweryfikować czy implementacja algorytmu jest poprawna.

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://www.youtube.com/watch?v=58jGpV2Cg50