AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm Merge Sort oraz Demonstracja Unit Testów przy użyciu Google Test

Autor:
Mateusz Stanek

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest stworzenie programu pozwalającego sortować tablicę algorytmem Merge Sort oraz kontrolowanie jego wersji za pomocą narzędzia git. Należy także stworzyć szereg unit testów mających na celu sprawdzenie poprawności programu.

Program będzie podzielony na plik klasy, oraz plik zajmujący się testami.

Wynikiem projektu powinno być repozytorium git i działająca klasa sortująca tablicę oraz zbiór testów, które program spełnia.

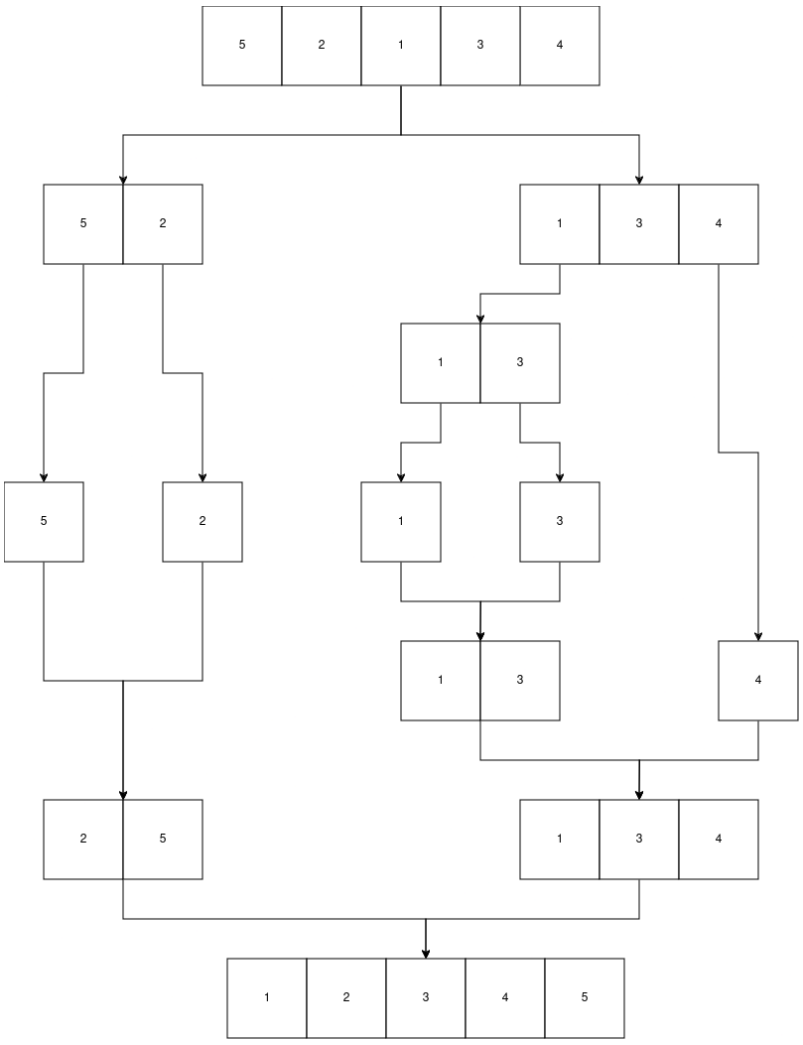
2. Analiza problemu

2.1. Algorytm Merge Sort

Algorytm Merge Sort[**mergewiki**] (Sortowanie przez Scalanie), jest algorytmem sortowania typu dziel i zwyciężaj[**divideandconquerwiki**] o złożoności obliczeniowej najgorszego i średniego przypadku $O(n \log n)$. Złożoność pamięciowa algorytmu wynosi natomiast $O(n)$.

Algorytm polega na początkowym podzieleniu tablicy na jednoelementowe części. Następnie, części te są ze sobą stopniowo scalane. Przy scalaniu element jednej podtablicy porównywany jest z drugą. Mniejszy wynik porównania trafia do następnego "poziomu"scalonej tablicy, a algorytm przechodzi do porównywania następnego elementu danej podtablicy.

Graficzna reprezentacja algorytmu jest zaprezentowana na rysunku **??**



Rys. 2.1. Reprezentacja sortowania Merge Sort

2.2. Git

Kolejnym konceptem, którym zajmuje się projekt jest narzędzie git[2]. Pozwala ono zarządzać poszczególnymi wersjami projektów. Głównym korzeniem gita jest system commitów, czyli zapisania zmian w pliku w stosunku do commita starszego. To, w połączeniu z jego innymi możliwościami pozwala na tworzenie długich i skomplikowanych osi czasu danych projektów.

Użycie gita można zademonstrować na prostym przykładzie. Tworzymy katalog a w nim repozytorium, uzywając komendy `git init`, jak widać na rys. 2.2.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/mattys/skrypty-i-syfy/studia/rok2/progr
amowanie-zaawansowane/p1/git-test/.git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % ls
drwxr-xr-x - mattys  6 Nov 15:54  .git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test %
```

Rys. 2.2. Puste repozytorium git

Stwórzmy jakiś plik i dodajmy go do repozytorium. Plik można dodać do repozytorium komendą `git add`

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit1" > plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git add plik.txt
```

Rys. 2.3. Stworznie pliku w repozytorium

Następnie należy scommitować zmiany.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "utworzenie pl
ik.txt"
[master (root-commit) bb3b898] utworzenie plik.txt
 1 file changed, 1 insertion(+)
 create mode 100644 plik.txt
```

Rys. 2.4. Commit nr. 1

Na rysunku 2.4 użyta komenda `git commit` commituje wszystkie dodane pliki (`-a`) z jakimś komunikatem (`-m`).

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit2" >> plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "uzupelnienie
plik.txt"
[master 40694c2] uzupelnienie plik.txt
 1 file changed, 1 insertion(+)
```

Rys. 2.5. Commit nr. 2

Na rys. 2.5, został utworzony kolejny commit, dodajacy zmiany do `plik.txt`.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git log
commit 40694c2e73758368445cb817f4524ee5c5afbece (HEAD → master)
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:26:07 2024 +0100

    uzupelnienie plik.txt

commit bb3b898df760395b5763575e204c3c43b513d2f6
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:12:31 2024 +0100

    utworzenie plik.txt
```

Rys. 2.6. Log gita

Jak na rys. 2.6 jest pokazane, używając komendy `git log`, można wyświetlić log commitów w repozytorium.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git checkout bb3b898df760395b
5763575e204c3c43b513d2f6
Note: switching to 'bb3b898df760395b5763575e204c3c43b513d2f6'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at bb3b898 utworzenie plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo plik.txt
plik.txt
```

Rys. 2.7. Demonstracja checkout

Jak widać na rys. 2.7, komenda `git checkout`, pozwala na przejście repozytorium w inny stan, w tym przypadku przechodzi się do commita o danym ID, pokazanym na rys. 2.6. Jako, że jest to pierwszy commit, nie ma w nim zmian z drugiego.

2.3. Doxygen

Doxygen[3] jest narzędziem automatycznie generującym dokumentację programu z komentarzy w kodzie źródłowym. Potrafi on generować strony HTML, gdzie można dynamicznie nawigować się miedzy rożnymi częściami kodu oraz pliki LaTeX, które można konwertować na różne, statyczne formaty.

2.4. Google Test

Google Test [**gtestrepo**] jest frameworkiem stworzonym przez Google służącym do prowadzenia Unit Testów w języku `C++`. Narzędzie to pozwala na gwarantowanie poprawności logiki kodu.

3. Projektowanie

3.1. Implementacja Merge Sort

Do zaimplementowania Merge Sorta zostanie użyty Język `C++` z kompilatorem `g++`. Wersja standardu `C++` to `C++20`. Jako, że projekt ma być rozdzielony na dwa pliki, zostanie zastosowany `CMake` w celu automatyzacji procesu budowania. `CMake` pozwala na generowanie plików budujących dany projekt, zgodnie z określoną konfiguracją. Oszczędza to programiście, szczególnie przy większych projektach, manualne pisanie Makefileów. Plik konfiguracyjny `CMakeLists.txt` może wyglądać jak na rysunku

```
cmake_minimum_required(VERSION 3.15)

set(PROJECT_NAME proj1)

project(${PROJECT_NAME} VERSION 0.1 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS True)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/out)

add_subdirectory(src)

```

Listing 1. Plik konfiguracyjny CMake

Edytorem będzie program Neovim. Jest to terminalowy edytor tekstu z możliwością poszerzenia funkcjonalności przy użyciu wszelkiego rodzaju pluginów. Wybrany został, dlatego że jest on już skonfigurowany na moim komputerze zgodnie z moimi preferencjami.

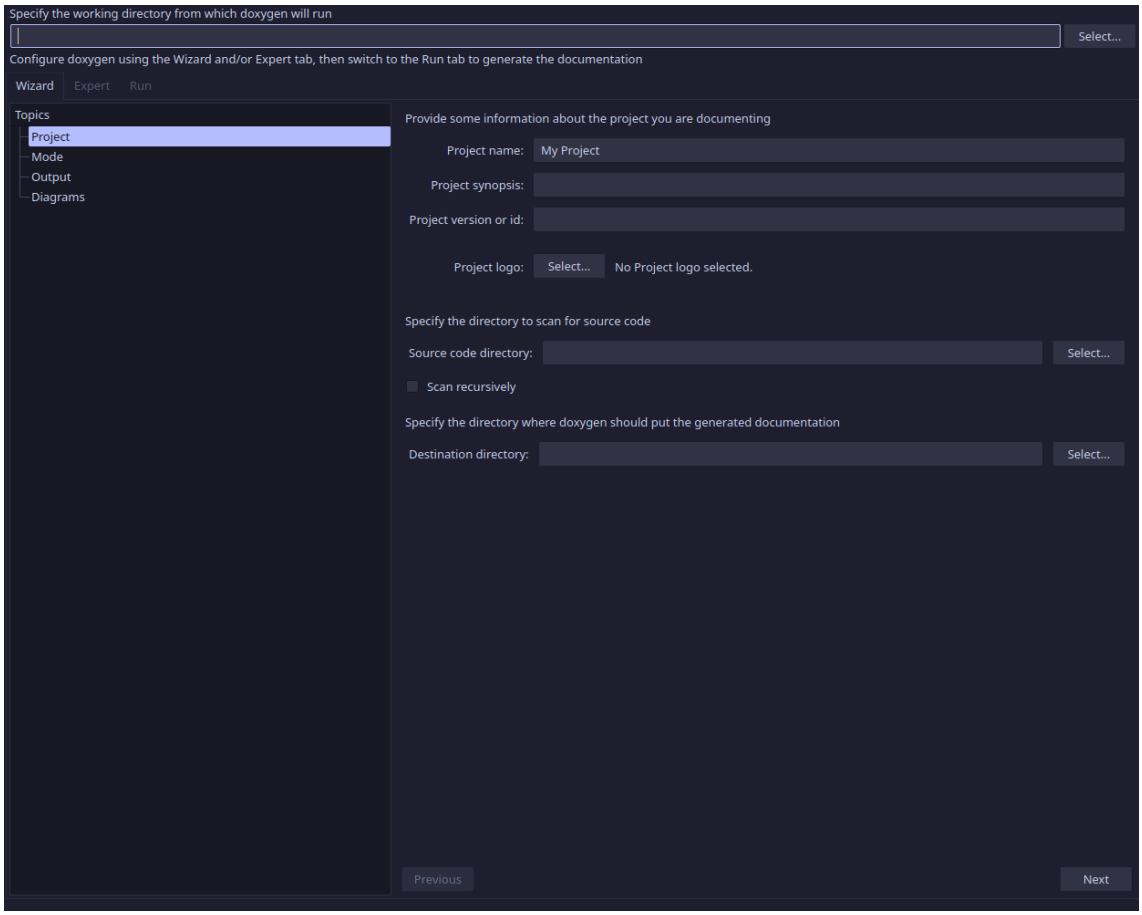
3.2. Git

Dla ułatwienia pracy, zastosowany został front-end dla gita o nazwie lazygit. Jest to terminalowy program, którego główną zaletą jest łatwa nawigacja przy użyciu klawiatury. Ponadto, jest on lekki i szybki.

3.3. Doxygen

Konfiguracja dla Doxygena jest wygenerowana przy użyciu programu doxywizard, pokazany na rys. 3.1, pozwalającego na graficzne zmienianie ustawień. Po wygene-

rowaniu konfiguracji, Doxygen wywoływany jest przy użyciu komendy.



Rys. 3.1. Interfejs programu doxywizard

3.4. Google Test

Google Test został dodany do projektu jako biblioteka w `CMake`, co zostało ukazane na listingu nr. **??**. Framework jest automatycznie pobierany i instalowany przy konfiguracji.

```
include(FetchContent)
  FetchContent_Declare(
      googletest
      URL https://github.com/google/googletest/
   archive03597a01ee50ed33e9dfd640b249b4be3799d395.zip
  )
FetchContent_MakeAvailable(googletest)
```

Listing 2. Dodanie Google Test do projektu

Pliki źródłowe testów znajdują się we własnym folderze, więc też trzeba do niego dodać `CMakeLists`, jak widać na listingu nr. **??**

```
file(GLOB_RECURSE SRC_FILES *.cpp *.h)
set(TESTS_EXECUTABLE ${PROJECT_NAME}_tests)

enable_testing()

add_executable(${TESTS_EXECUTABLE} ${SRC_FILES})

target_link_libraries(
  ${TESTS_EXECUTABLE} PUBLIC
  ${PROJECT_NAME}_lib
  GTest::gtest_main
)
```

Listing 3. Dodanie Google Test do projektu

Plik wykonywalny testów jest linkowany do - oprócz samego frameworka - biblioteki jaka jest wygenerowana z klasy `MergeSorter`, o której więcej w sekcji nr. **??**.

4. Implementacja

4.1. Klasa MergeSorter

Klasa jest zaimplementowana jako jeden plik `.hpp`. Nie jest podzielona na plik implementacji oraz nagłówek, ponieważ jest ona szablonem. Deklaracja klasy oraz prywatne elementy wyglądają tak jak na listingu nr. **??**.

```
template<typename T>
class MergeSorter {
private:
  std::vector<T> mergeSort(const std::vector<T>& toMerge) {
    if(toMerge.size() <= 1) {
      return toMerge;
    }

    auto left = toMerge | std::views::take(toMerge.size() / 2) |
   std::ranges::to<std::vector>();
    auto right = toMerge | std::views::drop(toMerge.size() / 2) |
   std::ranges::to<std::vector>();

    auto sortedLeft = mergeSort(left);
    auto sortedRight = mergeSort(right);

    return merge(sortedLeft, sortedRight);
  }

  std::vector<T> merge(const std::vector<T>& left, const std::
   vector<T>& right) {
    std::vector<T> merged;
    auto leftIt { left.begin() }, rightIt { right.begin() };

    for (; leftIt != left.end() && rightIt != right.end();) {
      if(*leftIt <= *rightIt) {
        merged.push_back(*leftIt);
        leftIt++;
      } else {
        merged.push_back(*rightIt);
        rightIt++;
      }
    }

    merged.insert(merged.end(), leftIt, left.end());
    merged.insert(merged.end(), rightIt, right.end());

```

```
35      return merged;
36    }
37
38  public:
39    void operator()(std::vector<T>& toSort) {
40      if(toSort.size() <= 1) {
41        return;
42      }
43
44      toSort = mergeSort(toSort);
45    }
46  };
```

Listing 4. Klasa `MergeSorter`

Klasa posiada tylko jedną publiczną metodę, którą jest `operator()(std::vector<T>& toSort)`. Parametr `toSort` jest wektorem, który ma być posortowany przez metodę. Jak widać na linijce nr. 40, żadne działanie nie jest wykonywane na wektorze, jeżeli ma jeden element albo jest pusty, ponieważ wtedy nie ma czego sortować. Inaczej do wektora przypisywany jest wynik prywatnej metody `mergeSort()`, zdefiniowanej na linijce nr. 4.

Warunkiem bazowym metody jest sprawdzenie czy parametr `toMerge` ma długość mniejszą lub równą jeden - wtedy jest zwracany. Na linijce nr. 9 i 10 deklarowane są zmienne `left` i `right` będące respektywną połową `toMerge`. Na linijkach nr. 12 i 13 do zmiennych `sortedRight` i `sortedLeft` przypisywane są rekurencyjnie wyniki metody `toSort()`. Wiemy że wyniki te będą posortowane, ponieważ albo zostanie zwrocona tablica pusta lub jedno-elementowa, albo program przejdzie dalej i wykona metodę `merge()`, zadeklarowaną na linijce nr. 18, na podanych zmiennych i zwróci jej wynik.

Metoda `merge()` ma na celu scalanie podtablic we większą, posortowaną tablicę. Tworzy ona ku temu celu wektor `merged` do zwrócenia oraz dwa iteratory `leftIt` i `rightIt`, wskazujące na początek wektorów danej połówki. W pętli na linijce nr. 22 sprawdzane są po kolei elementy z danych połówek. Mniejszy zawsze zostaje dodany do `merged` i nie sprawdza się już go, poprzez inkrementację danego iteratora.

4.2. Testy

Typowy test wygląda jak na listingu nr. **??**

```
1  TEST(ImportantTests, SortNormal) {
2    MergeSorter<int> sorter;
3    std::vector test{5, 2, 4, 3, 1};
```

```
4   sorter(test);
5
6   EXPECT_EQ(test, (std::vector{1, 2, 3, 4, 5}));
7 }
```

Listing 5. Test frameworka Google Test

Na linijce nr. 1 test deklarowany jest makrem `TEST()`. Pierwszy jego parametr to nazwa grupy testów, a drugi to nazwa samego testu. Blok testowy działa tak jak zwykły kod. Jednak, można w nim umieszczać wyrażenia mające testować dane warunki. Użyte w tym przypadku `EXPECT_EQ()` ma na celu przetestować równość dwóch wyrażeń. Zadeklarowany na linijce nr. 3 wektor `test` jest sortowany przez zadeklarowany linijkę wcześniej `sorter`. Jego posortowana wartość powinna wynosić ciąg `(1,2,3,4,5)`. Włączając program, widać z listingu nr. **??** że tak jest.

```
1 [ RUN      ] ImportantTests.SortNormal
2 [       OK ] ImportantTests.SortNormal (0 ms)
```

Listing 6. Poprawny wynik testu

Modyfikując ciąg, z którym ma być porównywany `test`, można też uzyskać niepoprawny wynik jak na listingu nr. **??**. Można zauważyć, że program powiadamia użytkownika w jaki sposób test nie został zaliczony.

```
1 [ RUN      ] ImportantTests.SortNormal
2 /home/mattys/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/
     p3/tests/tests.cpp:11: Failure
3 Expected equality of these values:
4   test
5     Which is: { 1, 2, 3, 4, 5 }
6   (std::vector{1, 2, 3, 5, 5})
7     Which is: { 1, 2, 3, 5, 5 }
8 [  FAILED  ] ImportantTests.SortNormal (0 ms)
```

Listing 7. Niepoprawny wynik testu

5. Wnioski

- Merge Sort jest przydatny przy większych zbiorach danych, gdzie limitowana pamięć nie jest dużym problemem. Dla mniejszych zbiorów quicksort jest często lepszym rozwiązaniem, szczególnie jeśli chodzi o uderzenia w cache.
- Unit testowanie jest koniecznością przy większych projektach, gdzie zmienienie jednej funkcji może skutkować zmianami w dziesięciu innych, o których nawet nie wiemy.

Bibliografia

[1] *Artykuł Wikipedii o drzewie BST*. URL: https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84.

[2] *Strona Gita*. URL: https://git-scm.com/.

[3] *Strona Doxygena*. URL: https://www.doxygen.nl/.

Spis rysunków

Spis tabel

Spis listingów