# 

Съдържание:

[1](#_Toc72869750)

[2](#_Toc72869751)

[1. Увод 3](#_Toc72869752)

[**2.** **Изисквания** 4](#_Toc72869753)

[2.1 Функционални изисквания 4](#_Toc72869754)

[2.2 Нефункционални изисквания 5](#_Toc72869755)

[**3.** **Проектиране** 6](#_Toc72869756)

[3.1 Диаграма на високо ниво 6](#_Toc72869757)

[3.2 Блок схема на действията 7](#_Toc72869758)

[4. Функционално описание на приложението 8](#_Toc72869759)

[Експериментални данни и анализ 9](#_Toc72869760)

[Приложение – source code repo: 9](#_Toc72869761)

[Демо 10](#_Toc72869762)

# Увод

Условие **№ 29** – Бенчмаркинг

Програмите за бенчмаркинг са изключително необходими при оценката на дадена функционалност. За тази цел ни трябва приложение, което да реализира функцията sort(), но зад нея стоят различни реализации. Намерете и използвайте поне четири готови алгоритъма за сортиране (напр: бърза сортировка, сортировка на мехурчето, сортировка на Шел, сортировка чрез сливане). Пазете резултатите на алгоритмите в контейнер и при извикването на sort() го обновявайте. Подгответе файлове с данни, с които да тествате всеки алгоритъм (малко/много на брой сортирани/несортирани числа). Записвайте резултатите (структура по ваш избор, съдържаща основните характеристики на измерването) на всеки тест в друг файл (със структурата tm от time.h засичате колко време е отнело). Този алгоритъм, който се е справил най-добре на най-много тестовете бива отпечатан. Този, който се справи найзле - бива изтрит от контейнера.

За приложението:

|  |  |
| --- | --- |
| **Име на приложението** | **StanMARK** with **graphEV** |
| **Версия на приложението** | v1.0, [beta 1.0] |
| **Състояние на документа** | Released |
| **Собственик на документа** | Стефан Станишев |
| **Дизайнер** | Стефан Станишев |
| **Разработчик** | Стефан Станишев |

**Source code: git repo** - <https://github.com/stanisev/stanMARK>

Основният проблем в този вариант е създаването на логиката по която сървърът ще хендълва изпратените клиентски команди и действа съответно със тях. (като първично е функцията която прави сортирането и бенчмаркинга).

Изграждаме client-server приложение с използването на UNIX системни библиотеки, което може да поддържа повече клиенти едновременно.

Приложението работи:

● Пуска се сървър на даден порт на някоя машина, съдържащ последователност от числа във текстови файлове.

● Клиентите подават адрес и порт на който искат да се свържат.

● С командите - use i sort, подадени от клиента към сървъра, се извършва съответния бенчмаркинг от страна на сървъра и след това сървърът връща отговор към отговарящия клиент.

# **Изисквания**

# Функционални изисквания

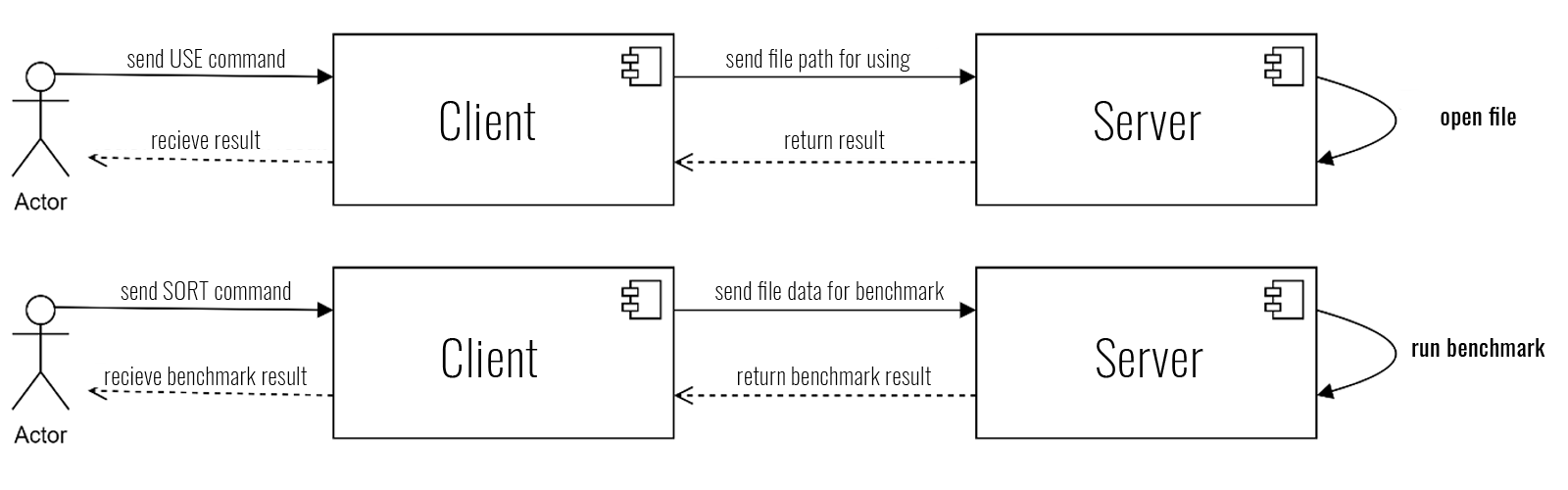
|  |  |  |
| --- | --- | --- |
| Номер | Изисквания | Приоритет |
| 1 | Създаване на сървърно приложение, чакащо за връзка с клиент | Задължително |
| 2 | Създаване на клиентско приложение, което по подаден адрес и порт се свързва със сървър | Задължително |
| 3 | Имплементиране на логика на команда sort <params> който извиква функции за бенчмаркинг | Задължително |
| 4. | Имплементиране на логика за записване на резултатите на всеки тест във файл и изписване на екрана | Задължително |
| 5. | Имплементиране на логика за графично представяне на резултатите, получени от тестовете. | Задължително |

# Нефункционални изисквания

|  |  |  |
| --- | --- | --- |
| Номер | Изисквания | Приоритет |
| 1 | Приложението да  бъде написано на **C**  програмен език | Задължително |
| 2 | Приложението да използва time.h библиотека | Задължително |
| 3 | Помощно приложение за създаване на диаграми, да бъде написано на **python** | По избор |
| 4. | Експортиране на диаграма във формат .**png** | По избор |

# **Проектиране**

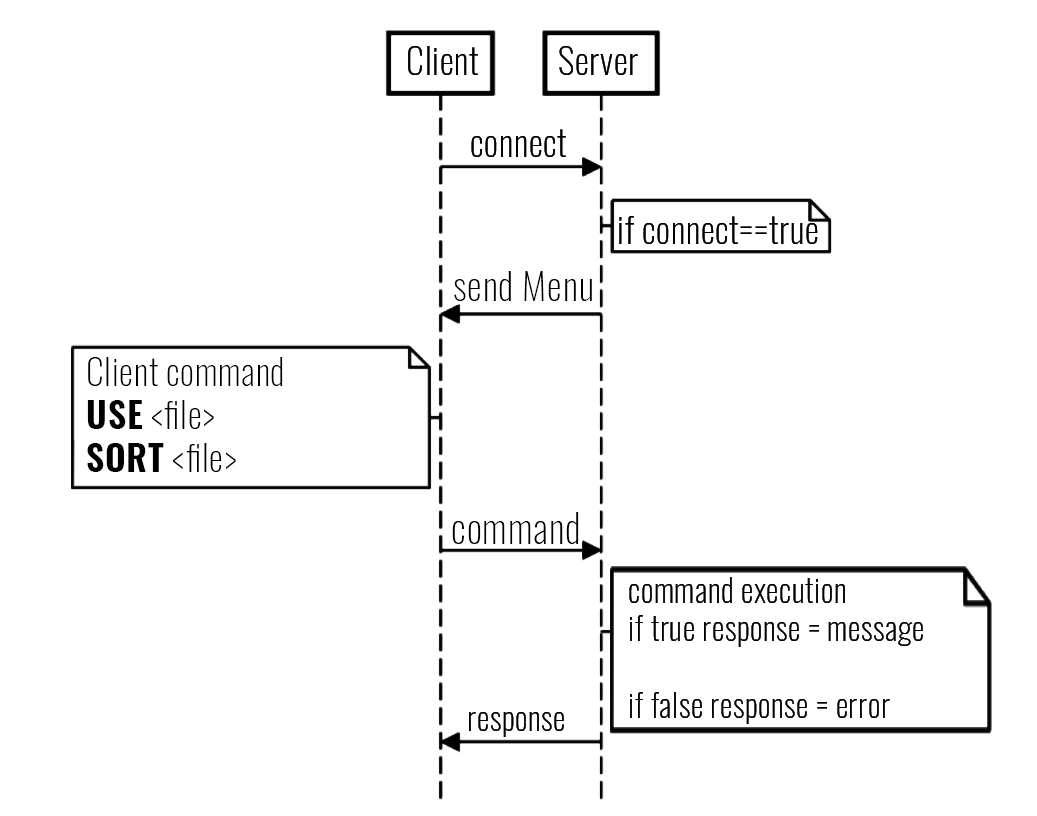
# Диаграма на високо ниво



# Блок схема на действията

Блок схема на логиката за изпълнение на процесите:

*Client – Server flow:*



# Функционално описание на приложението

Приложението започва с пускане на сървърната част и като параметър му се подава **порт**-а на който искаме той да работи. След това се пуска **клиентската** част и като параметри му се подават ***хостът*** на сървъра и ***портът***.

**Сървърната** част съдържа **7** функции (main, connection\_handler, useGivenFile, benchmarkGivenFile, sort, intdup, printArray и error).

**main** - в main функцията създавам файлов дескриптор за сокета с подадените параметри. Също така пускаме нишка за всеки свързан клиент.

**connection\_handler** - в тази функция взимаме подадените от клиента масив от символи и ги трансформирам до желаното състояние с което да работя. От първите 2 байта на масива от символите взимам дължината на акцията, от която потребителя се нуждае. Използвайки тази информация, взимаме акцията и я сравнявам с име на акция която сървъра извършва. В зависимост от акцията, останалата част на низа се използва отговарящо. От connection\_handler функцията се викат ***useGivenFile*** и ***benchmarkGivenFile***.

***useGivenFile*** - в тази функция се проверява дали избрания файл съществува. (За тест имаме 3 файла с последователност от цифри с имена SEQUENCE1.txt, SEQUENCE2.txt, SEQUENCE3.txt)

*пример*: **USE SEQUENCE1.txt**

***benchmarkGivenFile*** - след установяване, че файла съществува, тази функция чете от файла, прехвърля числата от файла в масив и извиква функцията **sort**.

**sort** – в сорта си правим 4 копия на масива който се подава и след това извикваме съответно всичките алгоритми за сортиране като им засичаме времето за изпълнение с помощта на clock() от time.h. След това записваме в структура 3 от 4 алгоритма, които са се справили най-добре, и изпращаме съобщение на клиента кой алгоритъм се е справил най-добре!

*пример*: **SORT SEQUENCE1.txt**

**intdup** – спомагателна функция за правене на копия на масив.

**error** – използваме за хващане на грешки.

**printArray** – спомагателна функция за изписване на масива на екрана.

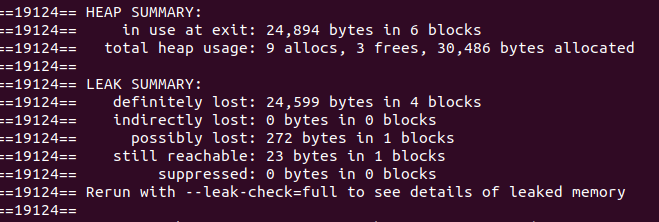
**Клиентската** част съдържа само main - в тази функция се взима акция от потребителя и трансформиран низ се изпраща до сървъра. След изпращане, клиента чака отговор на сървъра и го изписва в съответна форма на екрана.

# Експериментални данни и анализ

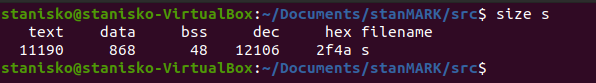
Експерименталните данни ги генерирах от сайта: <https://www.random.org/sequences/>, като три файла (SEQUENCE1.txt, SEQUENCE2.txt и SEQUENCE3.txt). Файловете са с размер от 500, 1000 и 10000 елемента.

Анализ на heap с Valgrind и top:

В сървърната част на приложението, голяма част от паметта се заделя динамично. След изпълнението на програмата се вижда, че част (3) от паметта се освобождава, има данни, които не се освобождават, но не е загубена тяхната референция. С детайлен анализ можем да отстраним тези ‘проблеми‘.



Анализ на Compile-time data:



**text** – размера на кодовия сегмент – това което попада във Flash паметта.

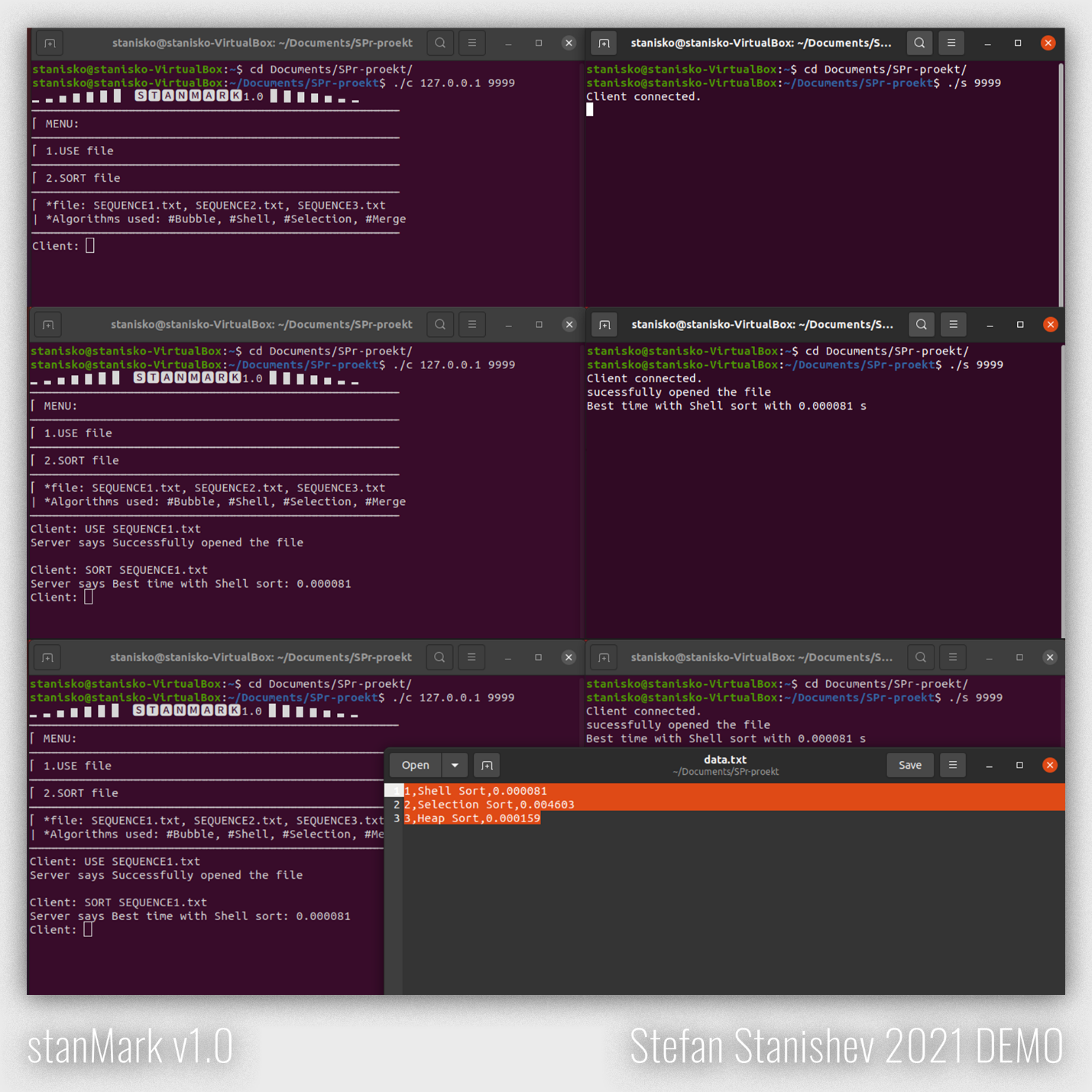
**data** – размера на данните които са инициализирани в кода, пример **buffer[BUFFER\_SIZE] = {0}**, като и int x = 0

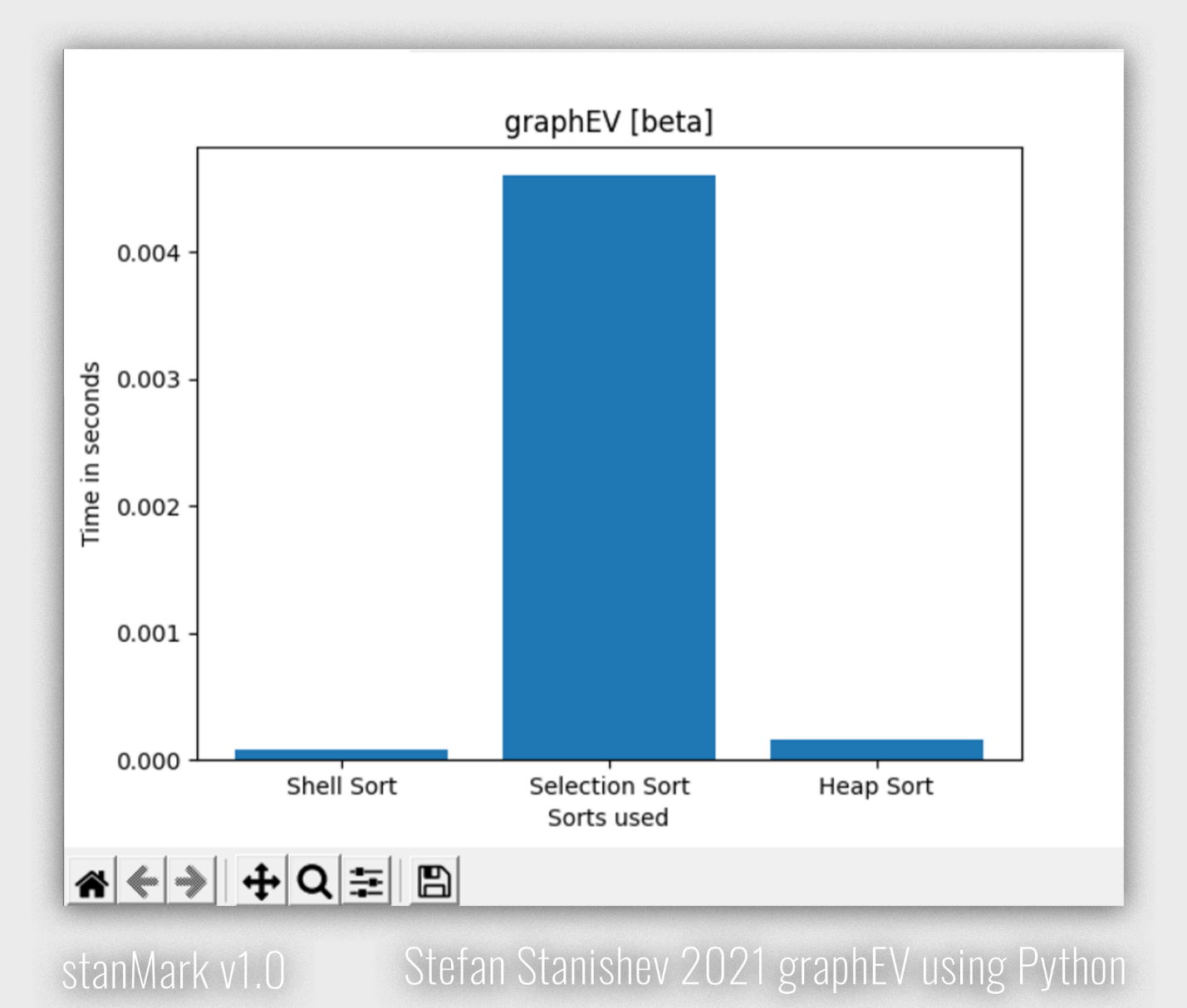
bss – размер на блока стартиран от символен сегмент, което попада в RAM паметта.

**dec** = text + data + bss = 11190+868+48= 12106= 2f4a(hex)

Приложение – source code repo: <https://github.com/stanisev/stanMARK>

# Демо





Литература

[1] - <https://www.tutorialspoint.com/c_standard_library/time_h.htm>

[2] - <https://www.techiedelight.com/find-execution-time-c-program/>

[3] - <https://cboard.cprogramming.com/c-programming/153674-reading-file-storing-values-array-help-please-o.html>

[4] - <https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-write-write-data-file-socket>

[5] - <https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-read-read-from-file-socket>

[6] - <https://www.geeksforgeeks.org/bubble-sort/>

[7] - <https://www.geeksforgeeks.org/shellsort/>

[8] – <https://www.geeksforgeeks.org/selection-sort/>

[9] – <https://www.geeksforgeeks.org/heap-sort/>