Introductory Programming 2022
General Setting & Project Description
ITU Copenhagen

Mahsa Varshosaz*
mahv@itu.dk

1 Introduction

The goal of the final programming project is to implement a large piece of software; specifically, you will develop a web-based search engine. This project consists of two parts:

1. A mandatory part where you create the core of the project. A satisfactory solution to this part is required to qualify for the exam. Each member of the group should contribute to most of the mandatory parts, including committing code to the repository.
2. An optional part where you can do extensions to the basic core. When the mandatory tasks are done, you can pick from proposed extensions, or work on your own extensions. Though doing this part is optional you are encouraged to try, since it will help prepare you for the exam as well as later courses.

The project must be carried out in the Java programming language, in groups of 3–4 students. You should already know who your group members are by the time you read this. If for any reason some group members are not able to complete the project, the remaining members are still expected to hand in solutions to the mandatory tasks. Each group is required to document its work in a written report, handed in at the end of the project.

The final product (a zip file with all source code) and report (a single PDF file) must be handed in via LearnIT by **Monday December 19 at 12:00 noon** (hard deadline). It suffices that one member of each group submits. Please use the file names **groupXX.zip** and **groupXX.pdf**, where XX is your assigned group number.

*Based on project description by Martin Aumüller and Troels Bjerre Lund

1.1 Covered Programming Tools and Techniques

The project makes use of the following techniques and tools that have been covered in class:

1. Version control using git and ITU's github.
2. Unit testing with JUnit.
3. Debugging (in the programming language/with IDE support).
4. Simple web services.
5. Code documentation using Javadoc.
6. Build tools such as Gradle.
7. Benchmarking code (optional).

Becoming familiar with these tools and techniques through practical application is a central goal of the final project. The goal is *not* to develop a search engine — that is only a by-product. In particular, you should not make use of external *libraries* that implement parts of the functionality; the only allowed libraries for the mandatory parts are Java's standard libraries. The code you commit must be written by yourselves (copying code is no different from other kinds of plagiarism), with the exception of code in the project template. You are welcome to learn from example code that you find (e.g. to learn how to use some standard library), but then use this understanding to implement your own solution.

2 Background

Search engines are basic tools in our everyday lives. Their task sounds simple: Given a query string, retrieve all web pages that "match" the query string (from a database of web pages), and rank them according to some measure of "importance" with respect to the query string. Achieving this goal seems straightforward, but there are plenty of challenges to overcome. The mandatory parts of this project will allow you to provide basic solutions to these challenges in a guided way. More advanced features can be developed as extensions. This section provides some background on the general structure of the search engine program that you are going to develop.

2.1 The Database of Web pages

The database of web pages is represented as a "flat" file. Consequently, the search engine program takes a filename as an argument. Each file lists a number of web pages including their URL, their title, and the words that appear on a web page. In particular, a web page starts with a line "*PAGE:" that is followed by the URL of the web page. The next line represents the title of the web page in natural language. This line is followed by a list of words that occur on the page. The following example illustrates the file format:

Example File

```
*PAGE:http://www.exampleA.com/
Web page A's title
Here
is
a
word
and
one
word
more
*PAGE:http://www.exampleB.com/
Web page B's title
Here
is
more
and
yet
more
```

Note

If a web page entry contains less than two lines after the "*PAGE" line, i.e., it has either no title or no words, the entry should be **omitted**.
(We have to assume that the entry for this web page is erroneous.)

We provide you with datasets extracted from the English wikipedia (`http://en.wikipedia.org`). Each dataset has different sizes, as detailed below:

Wikipedia

- `enwiki-tiny.txt`
  `(4kb, 6 web pages)`
- `enwiki-small.txt`
  `(544kb, 1033 web pages)`
- `enwiki-medium.txt`
  `(11.6mb, 25011 web pages)`
- `enwiki-large.txt`
  `(328mb, 25033 web pages)`

We have tested the build script with Gradle (7.4.2) — if you have an older version installed you may have to upgrade.

2.2 General Recipe To Answer a Query

Assume the user of your program wants to query the collection of web pages with a query string. From an abstract point of view, the search engine has to do the following:

Recipe to process a query

1. **Check** the query. ("Does the query make sense?")
2. Retrieve the list of web pages that **match** the query.
3. **Rank** the list of matching web pages according to their importance with regard to the query.
4. **Return** the list of ranked web pages.

Your project starts with a very basic setup: A query consists of a single word, everything else is invalid. A web page matches a query when the query word is contained in the list of words found on the web page. No ranking is going to take place.

2.3 The "Working" Prototype

Your git repository initially contains the source code for a simple search engine with minimal functionality. It is implemented in a way that mostly ignores the principles of good

software design, so an ongoing task throughout the project will be to refactor the code to introduce useful abstractions. The program takes the name of a data file and tries to read this file, almost according to the rules stated above. The file name is written in the `config.txt` file, to make launching of the webserver simpler. `config.txt` should not be under version control, since it would cause too many needless changes and merge conflicts.

In the initial prototype, each web page is internally stored simply as a list of the relevant lines from the input data file. These lists are stored in another list, as a field of the WebServer. Another responsibility of the WebServer class is to launch an HTTP service on port 8080. This can be accessed through the url `http://localhost:8080`. This will show a simple user interface, allowing the user to query the search engine: Given a word, it outputs the names and URLs of all web pages that contain this word. The client is written using HTML/CSS/Javascript and is provided to you. You will only have to make very few changes to the client, but you are free to do any modification you desire. The client sends specific messages to your server application and expects the query result to be provided in a certain way. A schematic representation is given in Figure 1.

3 Mandatory Tasks

The mandatory tasks consists of solving five tasks, which will be explained in the project kick-off lecture. You should consider these tasks consecutively, i.e., in the second task you will expand on the work you did for the first task, and so forth.

You are encouraged to use functionality that is already implemented in Java. In particular, data structures provided in `java.util`[1] and the methods for String objects in `java.String`[2] will turn out to be useful. **If you think the solution to a task involves a very complicated piece of code, consult with your TA first!**

Note

Some tasks contain challenges, clearly marked as "Challenges" in a green box. These tasks are **not** part of the mandatory tasks, but recommended for students who want to be better prepared for the exam and beyond.

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html`

[2] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html`

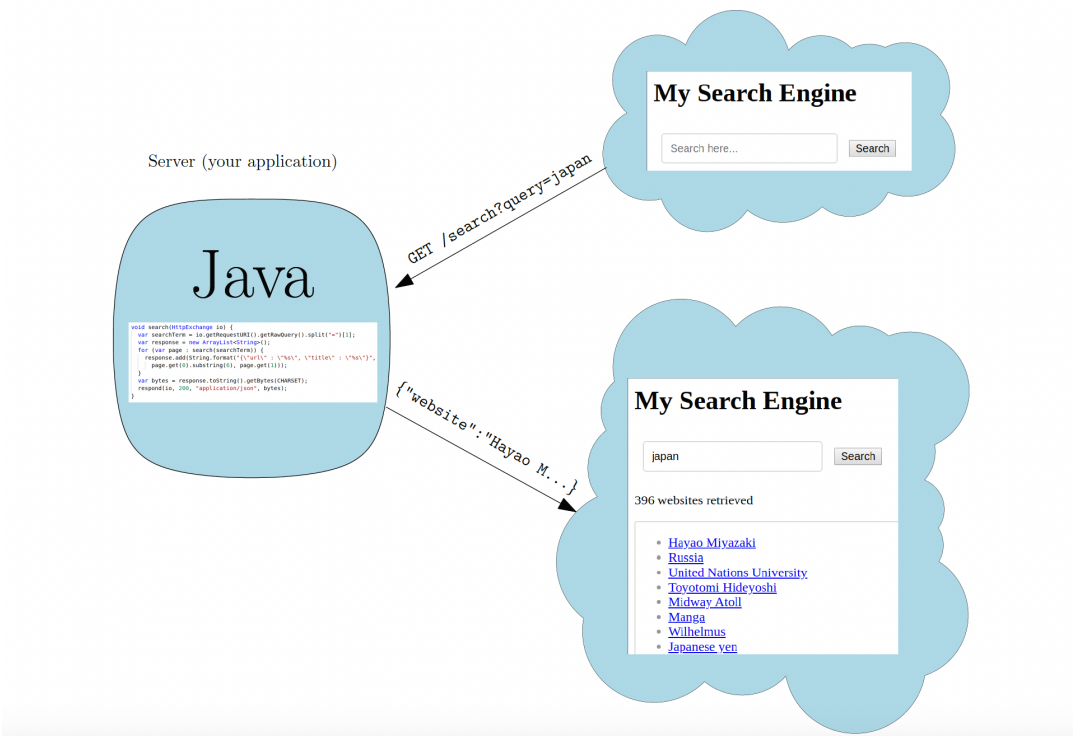

Figure 1: Schematic overview of the client-server architecture. The client sends the query to the web server (using HTTP GET) and receives an answer (in JSON) that is subsequently displayed.

Applying Software Development Techniques

In this course, we have learned about different software development techniques, in particular **unit testing** and **technical documentation** using Javadoc. **Your final product has to provide unit tests and documentation for all mandatory tasks.**

3.1 Task 0: Getting the Source & Compilation

This task makes sure that you can access the source code and are able to compile and run it. Every member of the group must be able to modify, compile, run, and contribute to the project. This is done through GitHub at `github.itu.dk`.

Getting the Source Code using Git.

- You should already have git installed on your system. If not, do not delay any further!
- Clone the prototype source code that has been shared with you at `https://github.itu.dk/`. The name of the repository is **mahv/GP22-groupXX**, where XX is your group number. This is your collaboration repository that all group members as well as your TA has access to.
- Compile and run the application by running
  `gradlew run`

> **Note**
>
> Do not consider Task 0 in your report!

3.2 Task 1: Refactoring

This task will ensure that the code base gets cleaned up to a more malleable state, where new functionality can be added more safely.

- This will be an ongoing task that you should return to again and again after each of the remaining tasks. Make sure to document your choices in the report.
- Consider if the test suite properly covers the current functionality. If not, add additional tests or rewrite existing ones. To help with this, run
  `gradlew test jacocoTestReport`
  , which generates a coverage report in
  `build/reports/jacoco/test/html/index.html`
- With proper test coverage in place, refactor the code, making the desired changes until you have a cleaner code base which passes the same tests as before.
- This step might require adding additional classes to introduce new abstractions. This is definitely the case for the initial prototype.

3.3 Task 2: Modifying the Basic Search Engine

This is a warmup task to get your familiar with the code.

- Modify the program such that if no web pages match the query, the program outputs "*No web page contains the query word.*"
- Modify the reading of the data files such that it only creates a web page if both the title is set and the word list contains at least one word.

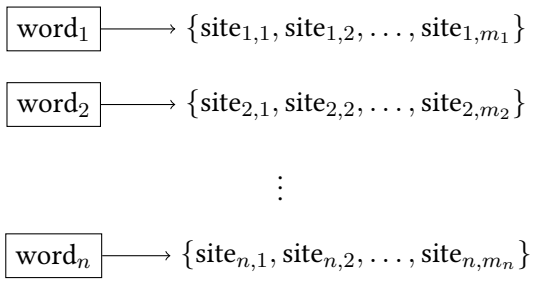

Figure 2: Abstract view of an inverted index for a database of web pages containing n words in total. Each word is associated with a list (represented as $\{\ldots\}$) of web pages that contain the word. In our notation, word "word$_i$" is contained in exactly m_i web pages.

3.4 Task 3: Faster Queries using an InvertedIndex

This task will make the query algorithm more efficient by providing a better way to search for web pages containing a certain word.

The prototype searches very inefficiently: It iterates through the list of web pages and collects those that contain the query word. An inverted index[3] allows for better performance by maintaining a mapping between words and a list of web page objects that contain these words. See Figure 2 for an example.

- Implement an inverted index, e.g., using a `java.util.Map`.
- It must be easy to specify which data file the index is to be created from, either through a constructor or a separate helper method. This should "populate" the index step-by-step to end up with a representation that has the structure depicted in Figure 2.
- Your class must have a separate method to lookup a given search term, so that the index can be tested (and benchmarked).
- Add the implementation to your search engine, to answer queries in the `WebServer`. That means, instead of looping over the list of web pages, build the inverted index at the start of the program, and only search the index to find the web pages that contain the input word.

[3] `https://en.wikipedia.org/wiki/Inverted_index`

3.5 Task 4: Refined Queries

This task will allow your search engine to work with more complex queries, as it is common in a search engine.

Note

This task will be much easier when you make yourself familiar with the methods provided by Java for String objects, including methods for Sets. And do not forget regular expressions!

First, look at the following example:

Example

The query

President USA OR Queen Denmark OR Chancellor Germany

should return all web pages "containing" at least one of the following word sets:

(i) President USA

(ii) Queen Denmark

(iii) Chancellor Germany

Here, "containing" means that all of these words can be found on the web page (but they do not have to occur adjacent to each other).

More specifically, your search engine has to provide the following functionality for a query string `str`:

Multiple Words If `str` has the form[4]

$$w_1 w_2 \dots w_n,$$

i.e., it contains n words split by a blank "␣", a web page matches the query string if and only if it contains these n words. (However, we do not require that they are adjacent to each other or in order.)

[4] We assume that no word equals the word "OR", see the next item.

Merging via OR Suppose that `str` can be decomposed into parts split by the keyword "OR" as follows:[5]

$$\mathrm{w}_{1,1} \ldots \mathrm{w}_{1,m_1} \mathrm{OR} \mathrm{w}_{2,1} \ldots \mathrm{w}_{2,m_2} \mathrm{OR} \ldots \mathrm{OR} \mathrm{w}_{n,1} \ldots \mathrm{w}_{n,m_n}$$

A web page matches the query if there exists an index i between 1 and n such that the text on the web page contains each word $\mathrm{w}_{i,1}$, $\mathrm{w}_{i,2}$, …, w_{i,m_i}. This means that at least one of these word sequences has the "Multiple Words" property from above.

Advice: Start slowly and extend the functionality step-wise. Try to keep your code extendable. For example, you could implement these features in the following steps:

(i) Support queries "$\mathrm{w}_1\mathrm{w}_2 \ldots$", without the "OR".

(ii) Support a single "OR" in the query.

(iii) Support multiple "OR" words in the query.

If you have not gone back to Task 1 yet, you should probably do this by now. Think about a good architecture for your query engine. It is recommended to put all the logic in its own class, for example `QueryHandler` with one public method `getMatchingWebPages(String query)`. In there, first, decompose the query into its components. For a single word, the query still retrieves lists of web pages from the inverted index. For the multiple words feature, a collection of lists must be checked for web pages that appear in all lists. The "OR" feature is easiest implemented by merging the results of multiple word queries. Make sure to remove duplicates, i.e., a web page is only allowed to be part of the result at most once!

Note that the query handling should be done in a separate class. It uses the InvertedIndex, but does not change it!

[5] This means that there exists no word $w_{i,j} = \mathrm{OR}$.

Challenges

1. Add one or both of the following two features to your query engine:

   URL Filter If `str` contains a word starting with "site:", let `url` be the characters following "site:" until the next blank "␣". A web page matches the query if and only if its URL contains `url` as a substring. All words starting with "site:" have to be removed from the actual query string.

   Prefix Search If a word ends with a "*" character, this word matches all words that start with the string before the "*" character. For example, the word "ho*" matches "hole", "home", "house", ...

2. Implement the query engine using an implementation of `java.util.Set`. Report on performance differences you observe via a benchmark (using JMH is optional).

3. Argue about the performance of your query engine. How long does it take to answer a multiple-words query with n words, assuming that word w_i yields a result list of size S_i? Assuming that the input contains $m - 1$ "OR" keywords (and thus m multiple-word queries), how long does it take to merge the m result lists of size $M_1, \ldots, M_m$?

What you need to know to solve the task

- Good understanding of the methods provided by the `java.util.*` classes covered in the course.
- This task usually requires more "thinking about how to do it" than "actual coding". Before starting to solve the task, make up some examples and discuss how the process of answering the examples should look like.

3.6 Task 5: Ranking Algorithms

In this task you will improve the results of your search engine by providing ways to rank a list of web pages with regard to their importance for a specific query.

The idea is simple: If a web page S contains a word w, we compute a number $s(w, S)$ (the "score") that shows how important this word is on the web page; the higher the number,

the more important the word. Have a look at Wikipedia[6] to see the descriptions of some well-known approaches.

Example

Assume the query is

President USA OR Queen Denmark OR Chancellor Germany

Further assume we come up with the following scores of the words on one web pages (they are made up here):

President	USA	Queen	Denmark	Chancellor	Germany
12	12	1	26	2	3

Computation of score:

- each part (split by **OR**): add scores together

President USA	Queen Denmark	Chancellor Germany
24	27	5

- final score for web page with respect to query: take the maximum!
- score for this web page for the query is **27**

1. You will need to come up with a suitable abstraction over the concept of a scoring method, so you can swap between different scoring methods, without changing too much code. Do not worry if you do not get it right on the first try; that's what refactoring is for.
2. Implement the term frequency score to fit your abstraction.
3. Implement the term frequency-inverse document score to fit your abstraction.
4. Modify your program such that each web page that matches the query is assigned a score. For a multiple-word query, the score of the web page is the sum of its score values for all words in the query. For an OR between two multiple-word queries, the score of the web page is the **maximum** of the scores in these two queries.

[6] `https://en.wikipedia.org/wiki/Tf-idf`

5. Modify your program such that the list of web pages that match a query is sorted by their score to the query in descending order.

6. Provide queries and their results that show differences in the result quality that is obtained by using these two score functions. Give a short description of how they differ.

Challenge

Implement the Okapi BM25 ranking algorithm.
(`https://en.wikipedia.org/wiki/Okapi_BM25`)

4 Extensions

This section introduces ideas for extensions you can work on after finishing the mandatory part of the task. You are encouraged to work on your own ideas, as long as they do not interfere with the mandatory parts. The mandatory parts are the meat of the project and all mandatory parts must be completed. Extensions are for those students who want to challenge themselves to learn more.

Benchmarking. In task 3 we introduced an inverted index as good way to answer single query words. But how much faster is it than the solution we had in our prototype? And how do different implementations influence the performance? If you are using maps, how does the choice between the different kinds of maps influence the performance? In this extension, you will benchmark these different approaches.

- You will need to come up with an abstraction over the concept "index", so you can more easily compare different implementations of it.
- Reimplement the simplistic index (from the initial prototype) to fit your chosen abstraction.
- Refactor your inverted index to fit the abstraction, and create one or more variants for comparison.
- You should now have at least three interchangeable indices: the prototype index, an inverted index using TreeMap, and an inverted index using HashMap. Design a small experiment that evaluates these data structures with respect to the running time of their `lookup` method.

You are encouraged to use the Java Microbenchmarking Harness as it takes care of most of the requirements below, but a thorough separate benchmark is also acceptable.

Solutions to Challenges in Mandatory Assignments. It might be difficult to do these, without the support of the whole group, as these typically modify the core elements.

Improve the Client GUI. Change the client code such that the result of searches are displayed in a nicer way. (E.g., the title of the page is provided, a short summary of the text displaying an occurrence of the search string, add pagination to show only a certain number of results on the front page.)

Add an Autocompletion Feature. Add an autocompletion feature that proposes search words to the user. This should be solved by a client/server interaction as in the normal query procedure. You should make yourself familiar with JQuery UI to solve this task with only very few lines of Javascript.

Personalized Queries. Add a feature to the server such that it stores search queries in a file and uses previous searches in the autocompletion feature.

Make Your Own Web Crawler. Provide a program that starts with a given URL and builds a collection of web pages by following hyperlinks on the web page. Parse this into a flat file format, but modify the format such that it also allows you to store the hyperlinks that are present on a web page.

> **Note**
>
> To avoid problems with your provider and/or being blacklisted from webservers, make sure to connect to only few (e.g., 30) web pages per minute.

Implement PageRank. After writing your own web crawler, you can implement the PageRank algorithm to rank web pages. (That is the algorithm which is the base of Google's web search.) See `https://en.wikipedia.org/wiki/PageRank` to get an overview of the algorithm. Compare the pagerank ranking to other rankings obtained by methods you implemented in Task 5.

Provide a "Fuzzy-Search" feature. If a search produces no result, it might be because the user performed an erroneous keystroke, so that, e.g., an "a" became an "s". In such cases you can choose to return pages which contain words almost matching the word searched for. A general technique to measure the similarity of two words is the Levenshtein distance (`https://en.wikipedia.org/wiki/Levenshtein_distance`). To build an an efficient index, you may consider to use a "q-Gram index", see, e.g., the lecture notes of Prof. Hannah Bast at `https://daphne.informatik.uni-freiburg.de/ws1516/InformationRetrieval/svn-public/public/slides/lecture-05.pdf`.

Finding Similar Web pages. Design a tool that, given a web page A, will search for web pages similar to A. Here, a clustering algorithm such as K-Means[7] using the vector space model[8] can be the algorithm of choice.

5 Checklist for Mandatory Assignments

Use the following checklist to make sure your project fulfills all mandatory requirements.

- ☐ Tasks 1–5 have corresponding Java code that solve them, in particular,
  - ☐ you have implemented an inverted index,
  - ☐ your search engine understands complex queries,
  - ☐ your search engine is able to rank results to a query, using different scores,
  - ☐ your search engine is based on a web service.
- ☐ All code in the project either comes from the project template, or was implemented by the group members,
- ☐ Each class and each method is documented using Javadoc
- ☐ Every class is accompanied by unit tests, in which you test the public methods exposed by the class (no tests need to be provided for main and "getter" methods)

[7] `https://en.wikipedia.org/wiki/K-means_clustering`
[8] `https://en.wikipedia.org/wiki/Vector_space_model`

6 The Report

Your group should hand in a report covering the mandatory tasks. We provide you with a LaTeXtemplate[9] for the report that you are encouraged to use (with high probability you will need to use this professional typesetting system later in your studies).

The report should start with a cover page with the information stated in the guidelines mentioned above, followed by a table of contents and an introduction. The introduction should briefly (in 1 page or less) give an overview of what is covered in the report, and **state what were the contributions of individual group members**.

The following structure of the report is advised:

1. Cover page
2. Introduction
3. Walk-through for Mandatory Task 1
4. Walk-through for Mandatory Task 2
5. Walk-through for Mandatory Task 3
6. Walk-through for Mandatory Task 4
7. Walk-through for Mandatory Task 5
8. Conclusion

For each of the mandatory tasks you should concisely, in at most 2 pages of text:

1. Describe how the task was solved, which data structures were used and how they were used (when applicable). This description must be in natural language and also explain the architectural choices that were made in order to solve the task. Java code may only be used to support the description. You are encouraged to provide figures, tables, and examples.
2. Reflect on the choices made when in solving the task, e.g. class design, data structure or algorithm. Mention any known bugs or shortcomings of your solution, and alternatives considered.

[9] `https://www.overleaf.com/read/cnfmvgsjdtfw`

Any tables, figures and illustrations do not count towards the maximum of two pages. The conclusion (1 page of text) should discuss general lessons learned, and what might have been done different if a new, similar project was started from scratch.

In addition to the project report you should hand in the source code for all relevant versions of your search engine as a **single** zip file containing the whole git repository (it is not needed to include the data folder). To avoid including generated files, it is advised to make a fresh clone of the repository and zip that for hand in. Make sure that your code compiles, is well-documented, and contains unit tests for all non-trivial methods.

A Javadoc

Gradle has built-in support for compiling javadoc comments to html. If you run

```
gradlew javadoc
```

the html will be written to `build/docs/javadoc/index.html` Use this to verify that your documentation of your public API is sufficient.

B Unit testing

Gradle has built-in support for jUnit. If you run

```
gradlew test
```

all the tests in the `src/test` subtree will be run. To measure the coverage of your test suite, you can use jacoco by running:

```
gradlew test jacocoTestReport
```

which generates a coverage report in

```
build/reports/jacoco/test/html/index.html
```