Project One

Wikipedia Indexer – Version 3.1

Table of Contents

Version history

S.no	Version #	Date	Author	Comments
1.	1.0	2 Sep 13	Nikhil L	Initial version
2.	1.1	4 Sep 13	Nikhil L	Added appendix
3.	2.0	7 Sep 13	Nikhil L	Added tokenization details
4.	3.0	15 Sep 13	Nikhil L	Added indexing details
5.	3.1	17 Sep 13	Nikhil L	Added grading guidelines

1. Project Description

This project aims to build a Wikipedia indexer with the following goals:

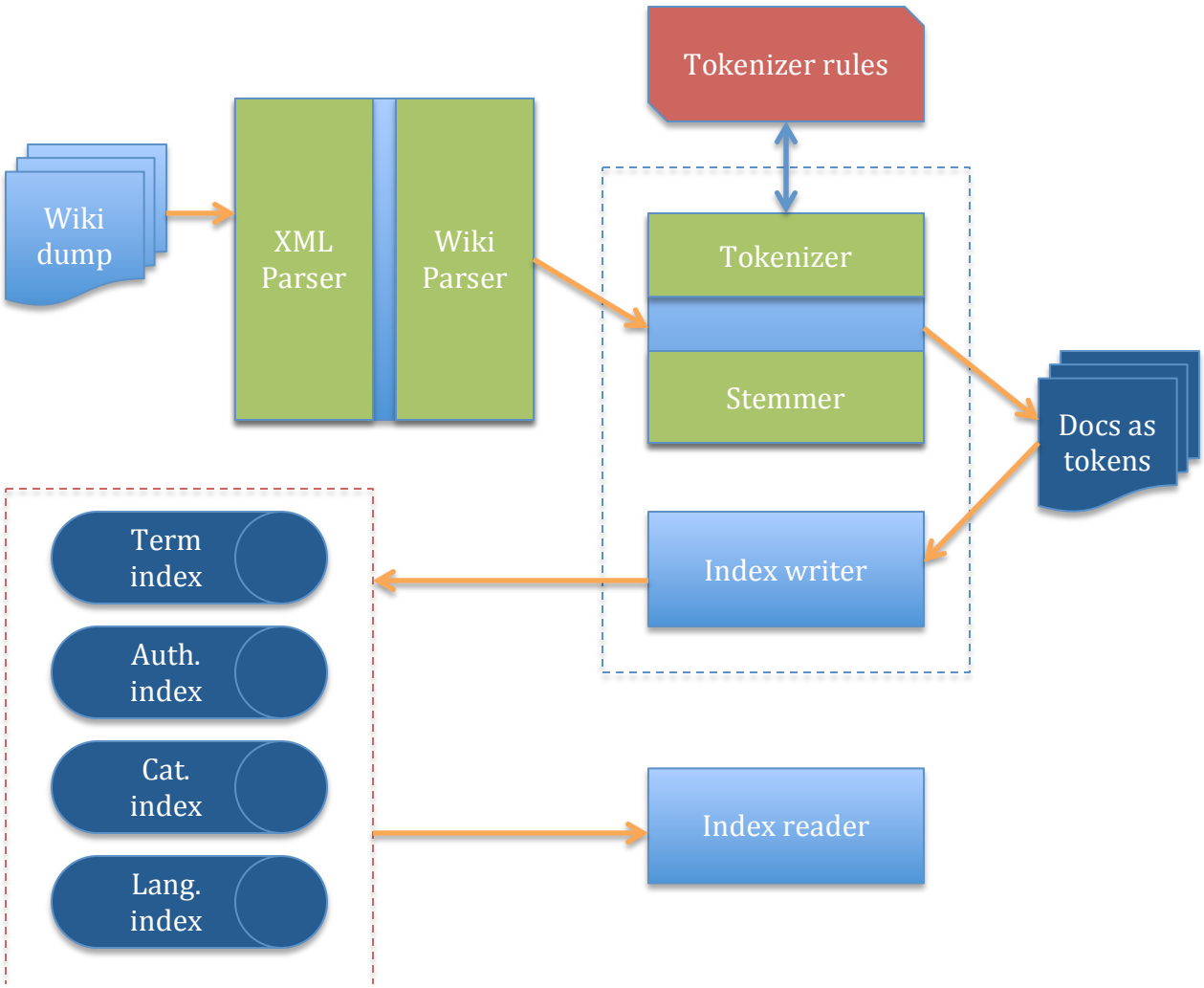
- Parse fairly involved Wikipedia markup.
- Index a decent sized subset of the Wikipedia corpus.
- Create multiple indexes on the page data as well as metadata.
- Provide an index introspection mechanism that can later be built upon to support queries.

You will be implementing this project purely in Java and starter code will be provided with detailed description of each component, method etc.

All deliverables are due by **29th September 2013, 23:59 EST/EDT**

2. System components

The system consists of two main components: a parser and an indexer. An overall system diagram is shown below.



The following subsections describe the functions of each component in more detail.

2.1. Parser

This component is responsible for converting the Wikipedia dump xml into a collection of documents. We have two document views as tabulated below:

S. no.	Functional view	Technical counterpart	Description
a.	Document with pre-defined fields	WikipediaDocument	The page XML is parsed as-is and represented as a collection of pre-defined fields. The only transformation is removal of Wikipedia markup
b.	Document as a collection of terms	IndexableDocument	The underlying text is tokenized, normalized, etc. and represented as a collection of token streams and thereby, tokens.

More technical details can be obtained by clicking on the class links listed above. For now, the distinction to understand is the demarcation between the two views. The first is an intermediate step that is closest to how the page appears in a browser. The second is where the structural details of the document are lost and the document is essentially nothing but a collection of tokens. As is evident, the first stage would focus on removal of Wikipedia markup and the second does the heavy lifting: tokenization, stemming and other linguistic normalizations. More details on this follow.

2.2. Indexer

Once the documents have been converted into a bag of tokens, the indexer then writes them onto multiple index files. Each index would also have its corresponding dictionary where the term to id mappings would be stored. You are expected to implement the following indexes. Representative snapshots of the indexes and dictionaries follow.

- **Term index**: An index that maps different terms to documents. This is the standard index on which you would perform retrieval.
- **Author index**: An author to document index, stores the different documents written by a given author.
- **Category index**: A category to document index, stores the different documents classified by a given category.
- **Link index**: A forward index that maps the different Wikipedia pages referenced by a given page.

a. **Document dictionary**: This contains a document, i.e., Wikipedia page to document id mapping. This should contain mappings for all base English pages as well as referenced language pages. See the table below:

Document or page	**Doc id**
Page_1	1
Page_2	2
…..	
ar:Page_k	5001
….	
zh:Page_m	9001
……	

This dictionary will be referenced by **all** the indexes.

b. **Term dictionary**: Like above it contains a term to term id mapping like illustrated below

Term	**Term id**
…	…
apple	12
applicat	13
…	…

c. **Term index**: It is a simple inverted index storing the document postings list for each term as follows

1	10,22,65,….
2	17,31,88,…
3	12,16,82,112,….
……	

d. **Author dictionary & author index**: Analogous to term dictionary and index; create a dictionary and index for authors.
e. **Category dictionary & category index**: Similar to term and author indexes, create a category dictionary and index.
f. **Language dictionary**: Create a language dictionary mapping each unique language found to a language id.
g. **Language indexes**: The inverted language index is a little tricky. Consider a page in English with a document id of say 100. It is referenced in say Arabic and German and the respective document ids for those pages are 7128 and 9432 respectively. There is another document in English with an id of 500 referenced in Arabic and Russian by ids 7832 and 8523 respectively. Then, the two indexes would look as follows if the language ids for Arabic, German and Russian are 7, 12 and 15 respectively:

Inverted index

Language id	Postings list
7	…,100, 500,…
12	…,100,…
15	…,500,…

Forward index

Document id	Postings list
100	7128,9432,…
500	7832,8532,…

h. **Link index**: This is a forward index that simply indexes all the linked pages by their document ids from a given page like:

Document id	Postings list
…	…
12	71,99,112,…
13	16,51,88,…
…	……

We leave it to the students to decide whether they want to create one large merged index or multiple smaller indexes. However, for each index there should not be more than 27 buckets: one for each character and one more for symbols, special characters and numbers.

We would use two approaches to indexing: a distributed approach for the term index and single threaded mechanisms for other indexes.

- We would have three different threads: one each for the Author, Category and Link indexes. These would be fed the corresponding key, value entries as read from the token maps of IndexableDocument instances. We expect to have only one index file per field type.
- We would use a distributed mechanism for the term index. We would delegate the responsibility to partition the term to different indexes based upon the term to a partitioner class. We would create one thread per partition and as above pass the terms to the respective partitions. The only hitch is that all partitions would share a document dictionary. This shared object would be used for all applicable indexes.

The entry point for each index is an instance of the IndexWriter class as described in the following section. As long as the corresponding TODO methods are implemented fine for each of the classes, you should be fine as we would take care of instantiating the classes as needed. We would also take care of multithreading as before.

3. Technical details

This section provides details about the starter code, and corresponding methods that need to be implemented.

We provide an entry point class called the Runner to trigger the indexing process. The main method needs two arguments:

- The fully qualified properties filename. A reference properties file has been provided. You are expected to modify the entries as required to make your code run.
- Mode: One amongst i (index only), t (run tests only) and b (do both).

3.1. Parsing

This stage consists of mainly three classes:

3.1.1. Parser

This class is responsible for parsing the given dump file into a collection of WikipediaDocument instances. It should internally invoke methods from WIkipediaParser class to parse Wikipedia markup. This class should primarily parse XML.

Methods to be implemented:

void	parse	This method is the entry point into this class. This method should do all the parsing and populate the collection with the parsed documents.
1.	String filename	This is the name of the XML file to be parsed. It is a fully qualified filename.
2.	Collection <WikipediaDocument> docs	This is a reference to the collection to which the parsed documents must be added. See below.

Utility methods:

void	add	This method is provided to add the parsed WikipediaDocument objects into the specified collection. The method does not do much but merely provides a synchronized way to do the adding.
1.	WikipediaDocument doc	The parsed document that needs to be added to the collection.
2.	Collection <WikipediaDocument> documents	The collection to which the document needs to be added

3.1.2. WikipediaParser

This class is responsible for removing Wikipedia mark-up. It contains a bunch of static methods, each with a specific purpose. Refer to the Javadoc for more details on each method.

3.1.3. WIkipediaDocument

This class represents a structured view of the Wikipedia page. The different fields are listed below:

- publishDate: The ***timestamp*** tag within the revision tag from the XML file.
- author: The ***username*** or ***ip*** tag within the contributor tag from the XML
- id: The ***id*** tag within the top level page tag.
- Links: A set of all pages referenced by this page. We are only bothered about other Wikipedia pages here and external URLs, file links, etc. can be ignored.
- categories: A list of all categories attached to this page.
- langLinks: A map containing entries for language to the corresponding page.
- Sections: A given page is broken into various sections. For pages with no sections or orphaned content, a section title of "Default" should be used.

3.2. Tokenizer

This stage is responsible for transforming the given WikipediaDocument class instances to IndexableDocument instances. We present this in two phases, first by presenting how tokenization occurs and second, on how the transformation is invoked.

3.2.1. Tokenization

We have created a pluggable tokenization interface that allows adding custom tokenization rules and configuring the flow based upon the context. The main classes are as follows:

3.2.1.1. Tokenizer

This class acts as an aggregator for different rules of tokenization. This class is pre-implemented and the only method of interest is the constructor. It expects a array of TokenizerRule implementations to be passed to it, each annotated with the name of the rule. The order is important as it determines in what order the different rules will be invoked.

To perform the tokenization, call the **tokenize** method.

3.2.1.2. TokenizerRule

This is an interface that must be implemented by a class that implements a given tokenization rule. Refer to the table in the following section for the different expected rules.

Each implementing class must be annotated with the ***RuleClass*** annotation. The ***Tokenizer*** will call the **apply** method on the given ***TokenStream*** to trigger this rule. You must implement this method and modify the stream as dictated by the given rule.

Refer to the ***EnglishStemmer*** for an example of how to annotate and use the apply method.

3.2.1.3. TokenizerFactory

This class is used to instantiate the ***Tokenizer*** for a given field type. The idea is that based upon the field being tokenized, you would need to modify what rules to apply and what order you would invoke them in. Implement this in the **getTokenizer** method. Given a field, you should figure out which rules to instantiate and in which order. Once you figure that out, simply create instances of the rule classes in order and pass them to the constructor of ***Tokenizer*** class and return this ***Tokenizer***.

3.2.1.4. TokenStream

This is the main class that encapsulates and provides utility APIs to handle a token stream. The table below summarizes the different methods.

void	append	This method is used to append new tokens at the end of the stream.
	String[] tokens	The tokens to be appended
Map <String, Integer>	getTokenMap	This method returns a token and count view of the stream. While the actual stream is imagined to be linear, this view is useful for the indexing stage.
Collection <String>	getAllTokens	This method returns the underlying stream as a collection of ordered tokens. It should return the view of the stream as-is.
int	query	A method to query for a given token from the stream. The method is expected to return the number of occurrences of the given token in the stream. If the token does not exist,

		obviously 0 should be returned.
	String	The token to be queried for
boolean	hasNext	Method to check if the stream has a next token The stream is assumed to have an internal iterator that allows moving token by token through the stream. This method is associated with the forward iterator. This method should return false when you reach the end of the stream.
boolean	hasPrevious	Method to check if the stream has a next token The stream is assumed to have an internal iterator that allows moving token by token through the stream. This method is associated with the reverse iterator. This method should return false when you reach the start of the stream.
String	next	Return the next token from the forward iteration of the stream. It should return null when the iterator reaches the end of the stream.
String	previous	Return the next token from the reverse iteration of the stream. It should return null when the iterator reaches the start of the stream.
void	remove	Method to remove the current token from the stream. The iterator should point to the next token in forward iteration.
boolean	mergeWithPrevious	Method to merge the current token with the previous token in the stream. It should return true if the merge succeeded, false otherwise. After the merge, the iterator should point to the newly merged token.
boolean	mergeWithNext	Method to merge the current token with the previous token in the stream. It should return true if the merge

		succeeded, false otherwise. After the merge, the iterator should point to the newly merged token.
void	set	Method to set the current token value to the specified token(s). If the number of tokens added is more than one, the stream should be manipulated to account for that. The iterator should point to the token corresponding to the last element in the array. Also, it is expected that remove will be called to delete a token instead of passing an empty string or null to this method.
	String… tokens	The tokens to be set.
void	reset	Method to reset the iterator to the start of the stream.
void	seekEnd	Method to reset the iterator to the end of the stream.
void	merge	Method to merge two token streams. The passed stream is assumed to be appended to the end of the current stream. The iterator should not be modified.
	TokenStream other	The stream to be merged into this stream.

3.2.1.5. TokenizerException

This is just a wrapper exception class for any Tokenizer related exceptions.

3.2.1.6. EnglishStemmer

This is the implementation of the Porter stemmer for English as a ***TokenizerRule***. The Stemmer code is from the website of the author of the algorithm and has not been written by us. This class should provide you some insight on how to implement the rules by modifying the token stream. Also note the required annotation at the top of the class declaration.

Further, we enumerate the different expected tokenization rules here. These are the rules we would test. You are free to add more rules if you want. The mechanism to inject new rules is fairly simple. All that a new rule needs is a new RuleClass name and annotation to that class.

S.no	RuleClass	Description
1.	PUNCTUATION	Any punctuation marks that possibly mark the end of a sentence (. ! ?) should be removed. Obviously if the symbol appears within a token it should be retained (a.out for example).
2.	APOSTROPHE	Any possessive apostrophes should be removed ('s s' or just ' at the end of a word). Common contractions should be replaced with expanded forms but treated as one token. (e.g. should've => should have). All other apostrophes should be removed.
3.	HYPHEN	If a hyphen occurs within a alpha-numeric token it should be retained (B-52, at least one of the two tokens must have a number). If both contracted tokens are alphabets, it should be replaced with a whitespace and retained as a single token (week-day => week day). Any other hyphens padded by spaces on either or both sides should be removed.
4.	SPECIALCHARS	Any character that is not a alphabet or a number and does not fit the above rules should be removed.
5.	DATES	Any date occurrence should be converted to yyyymmdd format for dates and HH:mm:ss for time stamps (yyyymmdd HH:mm:ss for both combined). Time zones can be ignored. The following defaults should be used if any field is absent: • Year should be set as 1900. • Month should be January • Date should be 1st. • Hour, minute or second should be 00.
6.	NUMBERS	Any number that is not a date should be removed.
7.	CAPITALIZATION	All tokens should be lowercased unless: • The whole word is in caps (AIDS etc.) • The word is camel cased and is not the first word in a sentence. • If adjacent tokens satisfy the above rule, they should be combined into a single token (San Francisco, Brad Pitt, etc.)
8.	ACCENTS	All accents and diacritics must be removed by folding into the corresponding English characters.
9.	WHITESPACE	A tokenizer that splits tokens based on whitespace as the delimiter. The splits should occur on one to any number of spaces.
10.	DELIM	A more generalized tokenizer than above that splits on any specified delimiter.

11.	STEMMER	A stemmer that replaces words with their stemmed versions. EnglishStemmer is already provided for this.
12.	STOPWORDS	A stopword removal rule. It removes tokens that occur in a standard stop list.

3.2.2. DocumentTransformer

This class is a threaded implementation of the conversion process. It has a constructor that takes a map of inderfield to tokenizer as one parameter and the WikipediaDocument to be converted as another. The call method must take care of reading fields from the WikipediaDocument instance and tokenize them using the corresponding Tokenizer and set the final token streams on the IndexableDocument object and return it.

The additional enum of interest here is INDEXFIELD that simply lists the different indexable fields within the document. You should think about which tokenizer rules to use for which field and in what order as explained before.

3.3. Indexing

The indexing is implemented by a series of different classes as explained below.

3.3.1. Writeable

This is an interface that provides marker methods that any object that can be written to the disk must implement. The different methods are listed below:

- writeToDisk: A method that when called should flush the contents of its current buffers onto the disk. This will be called once by the Runner on all the Writeable objects visible to it. The idea is to let the IndexWriter instances decide when to flush objects onto the disk. The only guarantee we provide is that we will call it once, so even if no partial flushing is done, the indexes would still be written to the disk.
- cleanUp: A method that should trigger clean up of all resources opened by this object. This also means that if partial writes have been done to the disk, a merge should be done here. As again the Runner would call this on all writeable objects visible to it.

3.3.2. IndexerException

This is a generic exception class to raise any exceptions or error conditions encountered during indexing.

3.3.3. Dictionary and subclasses

This is a set of three classes, an abstract class called Dictionary and two implementations: LocalDictiionary and SharedDictionary. The first is a thread safe dictionary that will always be called by one thread. The latter is

shared between different threads and must guarantee thread safe access. The different methods are enumerated below:

- Constructor: It takes two parameters, the Properties file and the name of the field on which this dictionary is being created.
- exists: Checks whether an given entry exists in the dictionary or not.
- getTotalTerms: Method returns the size of the dictionary.
- query (BONUS): Method to query the entries in the dictionary using a wild card search. It supports two wildcards: ? and *.
- lookup: The method to look up the given entry from the dictionary. If the entry already exists, it should return the id for the entry and if not, an entry should be created and the id returned.

3.3.4. IndexWriter

This class does all the heavy lifting of maintaining and creating an index. The different methods to be implemented are as given below:

- Overloaded constructor: It takes four arguments: the properties file, the two corresponding INDEXFIELD values and a Boolean specifying whether the index is forward or not. The two INDEXFIELD values are used to define an index. Every index is a collection of map / value entries. For term index for example, the key is the terms and the values are the documents and so on and so forth.
- addToIndex methods: There are four overloaded implementations. The variants are purely based on how the key and value fields are provided. They can be given in one of two ways: either as the converted id after doing a dictionary lookup or as the raw value before the lookup. Based on how the value is provided, the writer would have to do its own lookup. The caveat is as long as documents are one of the terms, they would always be provided as translated values.

3.3.5. Partioner

This class merely assigns a partition to the given term. It is left to the students to figure out how the partitions are created. The only two restrictions are:

- All methods are static, so the implementations must not be temporally defined. Same arguments should return same values despite in what order or when they are called.
- The final term index should not have more than 27 partitions. But when implementing this class, remember that the number of partitions would determine the number of threads allotted to the term indexer. Creating too many would give synchronization overheads and hamper performance. Too little would be a bad idea as well.

The two methods to be implemented are pretty straightforward and enough details can be found in the javadocs.

3.4. Index reader

This is the final component. All this class does is give utility methods to read an index (which means all including referenced dictionaries). The different methods are listed below:

- Constructor: It takes two arguments: the properties file and the field on which the index was constructed.
- getTotalKeyTerms and getTotalValueTerms: This is just the size of the underlying dictionaries, the former for the key field and latter is for the value field.
- getPostings: Method to retrieve the postings list for a given term. Apart from the corresponding reverse lookups (for both keys and values), the expected result is only a map with the value field as the key of the map and the number of occurrences as the value of the map.
- getTopK: This returns the key dictionary terms that have the k largest postings list. The return type is expected to be an ordered string in the descending order of result postings, i.e., largest in the first position, and so on.
- query: This emulates an evaluation of a multiterm Boolean AND query on the index. Implement this only for the bonus. The order of the entries in the map is again defined by the cumulative sum of the number of occurrences. What that means is, when evaluating the queries, for every retained postings entry, add its local occurrences to its running count. The value with the maximum occurrences should be at the top.

4. Code testing and evaluation

The project would be evaluated by running unit tests. A majority of the tests will be shipped with the starter code. We would run some additional tests as well as run the code against a larger corpus. Refer grading guidelines for more details.

4.1. Running local tests

For every method that you are expected to implement, corresponding tests will be provided. For a Class C in package P, its corresponding test class would be named CTest and can be found in the package P.test. Each method M would have its test method named as MTest.

We encourage you to write your own tests for code you add as well as any additional tests you think pertinent. We would be evaluating your code with our own test classes, so don't be afraid to experiment.

4.2. Remote evaluation

Once your code has been submitted, we would only use your code files. We do not expect any external libraries to be required for this project. We will ignore any extraneous code submitted and you stand to lose points or worse, not get any if your code does not compile or run.

Reiterating the testing methodology and restrictions:

1. **We will overwrite all classes within the *.test directories , i.e all *Test files. We will also overwrite the Runner class. Once this is done, we will compile and run your code and tests. If your code does NOT compile, we would not evaluate any further.**
2. **You should not modify the signatures of any TODO methods. We would have corresponding test methods for most of these methods. If we're unable to compile the code (as the test cases would reference this), we will not evaluate any further.**
3. **Do NOT modify any methods that are called by the Runner class (again as we will use our own Runner). Beyond this, you can add code as you please by creating classes, methods, fields as required.**

4.3. Grading guidelines

We summarize the scoring methodology as a table below:

S.no	Module	Sub module	Points
Level 1: We evaluate the correctness and completeness of the different modules. The idea is to verify that methods and classes have been implemented as per the requirements and does not focus on functionality.			**50**
1.	Parsing		***15***
		XML parser	5
		Wikipedia parser	10
2.	Tokenizing		***25***
		TokenStream	3
		11 rules * 2	22
3.	Indexing		***10***
		Dictionary classes	4
		IndexWriter	6
	Bonus 1	*Wildcard queries*	*5*
	Bonus 2	*Boolean queries (only AND)*	*5*
	Bonus 3	*Dictionary compression**	*5*
	Bonus 4	*Postings compression**	*5*
Level 2: We evaluate the functional completeness here by verifying the indexes against a internally generated gold standard. The evaluation strategy would involve verifying both the dictionaries and postings. One set would evaluate the sizes, comparing against expected size ranges and the other would evaluate content, random queries to			**30**

validate correct indexing.			
1.	Sizes		***12***
		Term index	6
		Other indexes * 2 each	6
2.	Correctness		***18***
		Term index	9
		Other indexes * 3 each	9
Level 3: We evaluate the performance of the system against different parameters. We are concerned with largely three parameters: speed, memory and scalability. We would evaluate the system by changing system parameters and the corpus used.			**20**
1.	Speed		***8***
2.	Memory		***6***
3.	Scalability		***6***
4.	*Bonus 5*	*Top 10 speed performers*	*5*

We would use automated testing for the entire evaluation (except items marked *, see below). The final score would then also be automatically generated based on the test results. The curve grading would be done at the end of the semester on the total marks,

5. Submission guidelines

We expect ONLY the source code to be submitted. Please zip your entire source code (the src directory, edu should be the top directory) and name it as cse535_<team name>.zip. Please convert team name to all lowercase on your submission.

To submit, from any cse unix machine (timberlake / mettalica / etc.) invoke:

submit_cse535 <zip file name>

and press enter. You would receive a confirmation message. You can make multiple submissions; any new submissions would overwrite the old ones. In case of name confusions, the latest file will be used.

Late days: The following late day penalties exist –

- 1 day: Lose 10 points
- 2 days: Lose 20 points
- 3 days: Lose 30 points

These will be applied on top of your total score. If you make any submissions after the deadline, it would be considered as a late day is being used.

If you choose to implement the compression techniques, please email Nikhil and set up demo timing. Any points for compression would only be given after a successful demo.

6. Appendix

6.1. Wikipedia structure

The Wikipedia document structure can be roughly broken down into two levels. The first is the XML markup that encapsulates the page and has additional metadata fields. The second is the structure within the page that uses Wikipedia markup to create structure.

6.1.1. XML markup

A skeleton Wikipedia XML page dump is as shown below with added comments:

```
<!—Top level tag, the full XML is enclosed within this
tag -->
<mediawiki …>
  <!—Contains details about the mediawiki site from
which the XML was exported -->
  <siteinfo>
    <!—Name of the site -->
    <sitename>Wikipedia</sitename>
    <!—Base webpage URL for the site -->

<base>http://en.wikipedia.org/wiki/Main_Page</base>
    <!—The program that generated this XML -->
    <generator>MediaWiki 1.22wmf14</generator>
     <!—Information about the used XML namespace -->
    <namespaces>
     …
    </namespaces>
  </siteinfo>
  <!—Start of a page -->
  <page>
    ….
  <!—End of a page -->
  </page>
  <!—More pages follow -->
<!—End of XML -->
</mediawiki>
```

For the purposes of this project, we are only concerned about the contents within the <page> tags.

The details for the page markup are given below. Note that the given list is not exhaustive and you may encounter additional tags. However, all necessary tags have been mentioned.

```
<!—Top level tag -->
<page>
    <!—The page title -->
    <title>…</title>
     <!—Namespace identifier, related to the
namespaces tag above -->
    <ns>…</ns>
    <!—ID of the page, unique within a language wiki -
->
    <id>…</id>
    <!—Information about the last revision -->
    <revision>
      <!—Revision ID , distinct from page ID -->
      <id>…</id>
      <!—Parent revision ID →
      <parentid>…</parentid>
      <!—Timestamp of when the revision was made -->
      <timestamp>….</timestamp>
      <!—Details about who made the revision -->
      <contributor>
        <!—User name of the author -->
        <username>…</username>
         <!—Author’s ID -->
        <id>….</id>
      </contributor>
      <!—Comment made with the revision -->
      <comment….</comment>
       <!—The text tag, contains all the page text -->
      <text ...>…</text>
      <!—Unique SHA key for the page -->
      <sha1>….</sha1>
     </revision>
  </page>
```

Refer the WIkipediaDocument class to figure out which tags are important and what fields they map to.

6.1.2. Wikipedia markup

Extensive markup information is provided here: http://en.wikipedia.org/wiki/Help:Wiki_markup. We believe the documentation is extensive enough and does not need additional explanation.

6.2. JUnit

JUnit is a simple testing framework for Java. The power of JUnit lies in being simple but allowing extensive tests to be run. It takes very little to set it up. Let's dive right in and take a simple example: we're trying to build a function that prints the factorial of a given number. The class and method skeleton is as given below:

```
public class Factorial {

   /**
    * Method to compute the factorial
    * @param number: The number whose factorial is to be
computed
    * @return the computed value
    */
   public int compute (int number) {
        return 0;
   }
}
```

Now we want to test that this method acts as we expect and that's where JUnit comes in. Let's write the skeleton test code. If you are using Eclipse IDE, you would get a similar skeleton code:

```
public class FactorialTest {
   @Test
   public void testCompute() {
        fail("Not yet implemented");
   }
}
```

We see two interesting things here:

1. We see a "Test" annotation. This was added in JUnit 4.x. What this tells the compiler is that this is a test method. Any method in any class marked with this would be treated as a test method and executed as a test case.
2. The fail(…) method. As the name suggests, if you execute these method, the test case will fail with a message: "Not yet implemented".

So far so good but how do we write tests? Well the library provides the following overloaded functions:

- assertTrue(…): Assert that the given Boolean expression is true
- assertFalse(…): Assert that the given Boolean expression is false.
- assertEquals(…): Assert that the given two objects are equal.
- assertNotEquals(…): Assert that the given two objects are not equal.

Let's see an example and incrementally build our tests. What do we know about the factorial function? We can build a truth table as follows:

Input	Expected Output
0	1
1	1
2	2
3	6
4	24
…	…

These become our test cases and the modified FactorialTest class looks as:

```
public class FactorialTest {
   @Test
   public void testCompute() {
        assertEquals(1, Factorial.compute(0));
        assertEquals(1, Factorial.compute(1));
        assertEquals(2, Factorial.compute(2));
        assertEquals(6, Factorial.compute(3));
        assertEquals(24, Factorial.compute(4));
   }
}
```

Notice how the first input is the expected value and the second is the value returned by the method under test. You just wrote your first test cases within two minutes. It is really as simple as that.

The power of JUnit lies in its flexibility. Almost all kinds of input and output conditions for a method can be tested. JMock, a mocking library to deal with dependent objects, further extends it but you don't need to worry about that.

We have some advanced concepts used as part of the framework:

- TestSuite: Allows grouping test cases into logical groups. We might use this to create groups of test cases based on functionality.
- Paramaterized and Parameters: Allows passing parameters into the test methods without having to hardcode all tests in code. We use this to inject the Properties file into the test classes that need them.

You are encouraged to look at the code and read up on these features. Though you are not required to work with these features for this project, they could be useful to you in the future.