

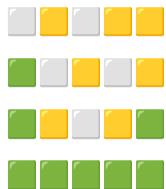
P2

Wordle is a word game where the player attempts to guess a 5-letter English word. Each incorrect guess receives feedback in the form of colored tiles indicating how closely the letter matches the target word.

This image shows a game with 4 guesses (*arise*, *route*, *rules*, *rebus*) for the target word, *rebus*. Guessed letters that exactly match the target word are marked green while letters that are in the target word (but not in the right position) are marked yellow. Letters that aren't in the target word are marked gray.

A	R	I	S	E
R	O	U	T	E
R	U	L	E	S
R	E	B	U	S

This result is typically expressed using a **pattern** of square emojis where each square corresponds to a letter.



If you're having issues viewing these emojis (e.g., they all appear as white boxes), or if emojis are not accessible to you, please make a post on the message board so that staff can help you get it fixed!

Absurdle is a variant of Wordle developed by [gntm](#):

Wordle picks a single secret word at the beginning of the game, and then you have to guess it. Absurdle gives the impression of picking a single secret word, but instead what it actually does is consider the entire list of all possible secret words

which conform to your guesses so far

By completing this assignment, students will be able to:

- Write a complete Java program to match a specification.
- Use sets and maps to create and manipulate nested collections of data.
- Follow prescribed conventions for code quality, documentation, and readability.

1 A game of Absurdle






Suppose the Absurdle manager only knows the following 4-letter words (the contents of the provided `small_dict.txt`).


- *ally, beta, cool, deal, else, flew, good, hope, ibex*

In Absurdle, instead of beginning by choosing a word, the manager narrows down its set of possible answers as the player makes guesses.

Turn 1



Step A: If the player guesses "argh" as the first word, the Absurdle manager considers all the possible patterns corresponding to the guess.

-  — *cool, else, flew, ibex*
-  — *hope*
-  — *good*
-  — *beta, deal*
-  — *ally*

Step B: The manager picks the pattern that contains the largest number of target words (so that it can do as little pruning of the dictionary as possible). In this case, it would pick the pattern  corresponding to the target words *cool, else, flew, ibex*.

Turn 2

Step A: If the player then guesses "beta", the manager chooses between the following possible patterns.




-  — *cool*
-  — *else, flew*
-  — *ibex*

Step B: The manager would pick  corresponding to the target words *else, flew*.

Turn 3


Step A: If the player then guesses "flew", the manager chooses between the following possible patterns.

-  — *else*
-  — *flew*

Step B: In this case, there's a tie between the possible patterns because both patterns include only 1 target word. The manager chooses the pattern  *not because it would prolong the game*, but **because**  **appears before**  **when considering the patterns in sorted order.**

You don't have to do anything special to handle "alphabetical" ordering for the emojis. Write your code so that the patterns are considered in alphabetical order and you should be fine!

Turn 4

After this, there's only a single target word, *else*. The game ends when the player guesses the target word and the manager is left with no other option but to return the pattern .

You can see a sample execution of the game described above by clicking "Expand" below.

Expand

2 Provided Scaffolding

The final version of this program will be longer and more complicated than the programs we've asked you to implement up until now. Because of this, we are providing more scaffold code than we have in previous assignments. Our scaffold code will manage most of the file scanning, user input, and the functionality of the game continuing to prompt the user for guesses and ending when the user finally guesses the correct word.

3 Implementation Requirements

You are responsible for implementing 3 specific behaviors, outlined below.

Pruning the Dictionary

public static Set<String> pruneDictionary(List<String> dictionary, int wordLength)

The provided scaffold code will handle asking the user for the dictionary file to be used, and reading the words from the provided dictionary into a `List<String>`.

For example, if the example dictionary file from above was used, `pruneDictionary` would be passed a list that would contain

[ally, beta, cool, deal, else, flew, good, hope, ibex]

At the beginning of the game, the user is also asked for the length of the word they would like to guess.

You should write a method `pruneDictionary` that takes the `List<String>` containing the contents of the dictionary file as a parameter and the user's chosen word length, and return a `Set<String>` that contains only the words from the dictionary that are the length specified by the user and eliminates any duplicates.

Your method should not modify the given dictionary - it should remain unchanged after your method has finished executing.

This method should also throw an `IllegalArgumentException` if the given `wordLength` is less than 1. You may assume that the given dictionary contains only non-empty `String` composed entirely of lowercase letters.

Handling a User's Guess

public static String record(String guess, Set<String> words, int wordLength)

Our provided scaffold code will handle repeatedly asking the user for a guess, but you will need to write the code to record or "process" the user's guess. Using the given guess, your method should *determine the next set of words that will be under consideration*, and return the pattern for the guess.

Recall the description of Absurdle (as opposed to its more benevolent counterpart, Wordle):

Absurdle gives the impression of picking a single secret word, but instead what it actually does is consider the entire list of all possible secret words

| *which conform to your guesses so far*




To that end, this method should

1. Consider the possible patterns given the user's guess and the possible answers it still has left to choose from.
2. Choose the pattern that corresponds to the largest set of words still remaining. In other words, choose the pattern that results in as little pruning of the dictionary as possible.

This method should throw an `IllegalArgumentException` if the set of words is empty or if the guess does not have the correct length. You may assume that the guess is made up only of lowercase letters.

Creating Patterns for Guesses

`public static String patternFor(String word, String guess)`

As the user makes guesses, the game will need to produce a pattern for the given guess (made up of the    blocks) as described above. You should write a helper method called `patternFor` that will be used to generate the pattern for a given target word and guess.

The process and algorithm for generating this pattern is nontrivial, so we've dedicated an entire slide (named "patternFor Development") to help you understand how we are asking you to approach this problem and to test your `patternFor` method in isolation.

We recommend you test and be confident in your `patternFor` implementation before moving on to the rest of the program.

While you're not required to follow the algorithm we outline in the "patternFor Development" slide, we strongly recommend that you do so. But, regardless of what algorithm you use in `patternFor`, **you may use at most 2 data structures in your implementation, and if you use 2 data structures then one of them must be a `Map`.**

4 Development Strategy

As in previous assignments, we *strongly* encourage you to write and test your code in stages rather than attempting to write it all at once before trying to run or test your work. You may recall that this is referred to as "iterative enhancement" or "stepwise refinement."

We suggest that you work on your assessment in three stages:

- **Part 1** - Write the `patternFor` method. The following slide (named "patternFor Development") should help you write your `patternFor` method in isolation before moving on to the rest of the program. Once you're passing all tests in the `patternFor` slide, copy and paste your completed method into the "Absurdle" slide to include it in your full program.
- **Part 2** - Write the `pruneDictionary` method. This is called by the scaffold code at the very beginning of the program, so it will be helpful to have it in place before you move on to actually accepting and recording the user's guesses.
- **Part 3** - Write the `record` method. This will involve calling the `patternFor` helper method.

5 Development Notes

Maps and Sets

As explained above, your program will need to consider patterns in *alphabetical* order. Given this, think carefully about which implementation of `Map` and `Set` is most appropriate for this problem.

`patternFor` Method

See the following slide (titled "patternFor Development") for detail on the specifics of building up a pattern for a given word and guess.

Because emojis can be challenging to work with, the scaffold code includes class constants `GREEN`, `YELLOW`, and `GRAY` that you can use in your code rather than using the emoji itself!

`record` Method

For each call to `record`, you should construct a `Map` to associate **patterns** with **target word sets** and use it to *pick the pattern associated with the largest number of target words*.

When there are multiple patterns that have the same largest number of target words, pick the pattern that appears first in the sorted order, i.e. the pattern that appears earliest when iterating over the `TreeMap`. The associated target word set becomes the dictionary for the next call to `record`.

6 Assignment Requirements

- We are telling you to write a few specific methods, but we still want you to practice *good functional decomposition*. Some of the tasks that we're asking you to write methods for are complicated and have significant sub-tasks. **Your finished program must include 2 "helper" methods.**
 - One of your required methods `patternFor` is counted as one of the helper methods, so you need to write *at least one additional helper method*.
- You should use interface types where appropriate and properly use type parameters when specifying the types of data structures in your solution.
- The spec describes some Exceptions which your methods must throw correctly in the specified cases. Exceptions should be thrown as soon as possible, so that no extra work is done before the Exception gets thrown.
 - Exceptions should also be documented in the comments of any method where an Exception is thrown. These comments should include the type of exception thrown and under what conditions.
- More generally, your method comments should follow the [Commenting Guide](#). You should write comments with basic info (a header comment at the top of your file), a class comment, and a comment for every method other than `main`.
 - Note: You do not need to write comments for the methods that we provide for you (so you do not need to write comments for `printPatterns`, `isFinished`, or `loadFile`) but should write comments for all other methods.

- As a general rule, you have learned everything necessary to implement a solution to an assignment by the time it is released. If you are tempted to use any concepts or tools that have *not* been covered in class, check the **Forbidden Features** section of the Code Quality Guide first to make sure they do not appear there!