# CPS506 Project Description

**Preamble**

You will write a program that plays the game of Boggle (with some minor modifications, described below). You are given an N*N grid of letters, and your task is to find connected chains of letters that form legal words. The basic rules of the game can be found at this link:

**https://en.wikipedia.org/wiki/Boggle**

In addition to the general requirements of the project, each language comes with its own language-specific constraints. These specify the format of the input and output, as well as submission instructions for each language.

Aside from these requirements, anything you do inside your program is up to you. Use as many helper functions or methods as you want, use any syntax you find useful whether we covered it in class or not.

**Game description**

The first input to your program will be an N*N grid of letters. The second input will be a list of legal words. Your task is to search for unique, legal words in the letter grid according to the rules of Boggle (see Wikipedia link above for more info):

- Each letter after the first must be a horizontal, vertical, or diagonal neighbor of the one before it.
- No individual letter cube may be used more than once in a word.

Your program will return this list of words, along with the indexes of each letter. Type details for inputs and output are language-specific and will be described below. The list of words returned by your program will be validated, and assigned a score:

| **Word length** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8+ |
|---|---|---|---|---|---|---|---|---|
| **Points** | 1 | 2 | 4 | 6 | 9 | 12 | 16 | 20 |

Variations from the official game

- Traditional Boggle has a minimum word length of 3, but for this project it will be 1.
- Word scoring also differs. See the table above.
- A traditional game of Boggle uses a 4x4 grid. The input to your program can be as small as 2x2 and grow arbitrarily large to NxN. The input will always be square.
- Traditional boggle includes a letter tile that combines “Qu”. There will be no such block in the project. Every letter tile will contain exactly one character.
- Boggle has a 3-minute time limit. Your program must complete the unit tests for each language in 30 seconds or less. We will test on a reasonably powerful computer.

**Game Example #1**

Below is a sample 4x4 board, as well as some (but not all!) words that are in it. Of course, the words you can find depend also on the list of legal words passed in but assume for this example that all standard English words are legal.

| | | | |
|---|---|---|---|
| **i** | **s** | **u** | **o** |
| **o** | **s** | **v** | **e** |
| **n** | **e** | **p** | **a** |
| **n** | **t** | **s** | **u** |

| | | | | | |
|---|---|---|---|---|---|
| **is** | **son** | **spa** | **sent** | **vast** | **issue** |
| **so** | **sap** | **ape** | **oven** | **nice** | **events** |
| **us** | **ten** | **east** | **nose** | **save** | **sonnet** |
| **up** | **vet** | **nest** | **tens** | **pause** | **session** |

The words found in the board above would be scored as follows:

is, so, us, up = 8 points (2 each)
son, sap, ten, vet, spa, ape = 24 points (4 each)
east, nest, sent, oven, nose, tens, vast, nice, save = 54 points (6 each)
pause, issue = 18 points (9 each)
events, sonnet = 24 points (12 each)
session = 16 points (16 each)

**8 + 24 + 54 + 18 + 24 + 16 = <u>144 points total</u>**

**Game Example #2**

Here is a 2x2 board, which is **much** simpler to solve. Immediately, we can eliminate all words longer than 4 letters. Further, a brute force approach that simply generates and tests every possible sequence of letters found in the game board is tractable. This is not the case in the 4x4 example.

| | |
|---|---|
| **e** | **a** |
| **s** | **t** |

| | | | | |
|---|---|---|---|---|
| **as** | **ate** | **set** | **eats** | **seat** |
| **at** | **eat** | **sat** | **east** | **sate** |

This solution scores **2*2 + 4*4 + 4*6 = <u>44 points total</u>**

**Technical details & language requirements**

Below are described the type constraints for input arguments and return values for each language. Note that you are not stuck with the types you are given. The game board enters your function in one form, but there's nothing stopping you from converting it to some other form internally. Same goes for the return value. Build your list of words and coordinates however you like as long as you convert it to the expected type before returning. Represent and operate on things however you like internally.

**Smalltalk:**

Create a class named **`Boggle`** that implements a class method called **`search:for:`**

Input: The first argument to this method will be the game board, represented as a static array of arrays whose elements are characters. For example, the 2x2 example above would be represented as **`#(#($e $a) #($s $t))`**

The second argument will be an array of strings representing the list of legal words, e.g. **`#('word1' 'word2' 'word3' … )`** and so on.

Output: Your search:for: method will return a dictionary. The keys will be the words that you've found, and the value at each key will be an array of points (see the Squeak terse guide for info on Dictionary and Point objects). Each point corresponds to the 2D index of the corresponding character in the game board. Remember that Smalltalk indexing starts at 1, so the location of $e would be 1@1.

You'll have to use the dynamic Array syntax for creating an array of points. Alternatively, if you create an OrderedCollection of Points and then convert to an Array using the asArray message, that will work too. For example, the word 'seat' would be represented by the following dictionary entry: **`'seat'->{(2@1).(1@1).(1@2).(2@2)}`**

**Elixir:**

Implement a function called **`boggle`** that accepts two arguments.

Input: The first argument, representing the game board, is a tuple of tuples whose elements are strings. The second argument is a list of strings containing the legal words for that game. For example, the 2x2 example would be: **`{{"e", "a"}, {"s", "t"}}`** and the legal word list would be: **`["word1", "word2", … ]`** and so on.

Output: The output will be a map, which is Elixir's dictionary equivalent:

**https://hexdocs.pm/elixir/1.12/Map.html**

Keys will be strings, and the values will be a list of coordinates. Each coordinate will be a pair tuple. For example, a map containing only the word 'seat' would be represented as follows: **`%{ "seat"=>[ {1, 0}, {0, 0}, {0, 1}, {1, 1} ] }`**

**Haskell:**

Implement a function called **`boggle`** that accepts two arguments.

Input: Haskell's argument types differ a bit from Elixir's. The board will be a list of Strings, where each string represents a row. The legal word list will also be a list of strings.

Output: Your boggle function should return a list of tuples. Each tuple contains two elements. The first element is a string (the word), and the second element is a list of pair tuples, where each pair tuple contains the coordinates of the corresponding letter in the string. The word 'seat' from the 2x2 board would be represented as follows:

```
[ ("seat", [ (1,0), (0,0), (0,1), (1,1) ]) ]
```

Haskell also has a Map type that you may find useful, but you are not required to use it:
https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html

**Rust:**

Implement a function called **`boggle`** that accepts two arguments.

Input: The game board in Rust will be passed as a reference to an immutable array of string slices – each slice is a row of the board. The legal word list will be passed as a reference to an immutable Vector of Strings.

Output: Your boggle function will return a HashMap, whose keys are the words you've found (String) and whose values are Vectors of pair tuples representing character coordinates. Documentation for Vector and HashMap is below.

https://doc.rust-lang.org/std/vec/struct.Vec.html
https://doc.rust-lang.org/std/collections/struct.HashMap.html

## Testing & Evaluation

Your code will be evaluated using an automated tester written by me. Therefore, it is of ***utmost importance*** that your code compile properly, handle input properly, and return results in the specified format for each language. Do not deviate from the requirements or your code will fail the tester outright. Your code must ***compile and run*** as a baseline. Half-finished code that doesn't compile or is riddled with syntax errors will not be accepted.

To help you achieve the baseline of "code that works", you are provided with mix/cabal/cargo projects with a handful of simple test cases for Elixir/Haskell/Rust. For Smalltalk, you are provided a Pharo image containing unit tests. The simple tests are to get you started, but your final evaluation will be based on larger boards and larger word lists.

Your grade for each project will be based primarily on how many points you achieve on each test game. For example, if finding all the words in a given game board would achieve 80 points, then your score for that test will be based on how many points you score out of 80. Finding short words in small boards is easy. Finding all the long words on a large board is more difficult, requiring a clever and efficient approach (not just brute force).

## Marking Rubric

**5 marks – Code runs, does *something* without crashing (test_1_mass_4x4)**

Your code compiles (without warnings!) and runs without crashing, entering an infinite loop, etc. Your boggle function (or search:for: method) must accept the input correctly and return a result of the proper type **with at least one word found**.

This test is not based on word scores. It will generate many randomized 4x4 boards and test your code with a small word list (500 words). The idea here is to tease out any edge cases that cause runtime issues.

**8 marks – 2x2 game board, 3000 words (test_2a_given_2x2)**

Code achieves maximum points on the 2x2 board shown above. Marks are awarded proportionally based on how close you get to the maximum score. For example, if the maximum score is 50, and you score 40, you get 4/5 for this board. Do NOT find the words manually and hard-code them. You will get a zero on any test that you do this.

**12 marks – 4x4 game board, 3000 words (test_2b_given_4x4)**

Same as above, but with a larger board! Marks are awarded proportionally based on how close you get to the maximum score.

**8 marks – 2x2 game board, ~280,000 words (test_3a_small_2x2)**

Same board as test_2a_given_2x2, but with a much larger word list.

**6 marks – 4x4 game board, ~280,000 words (test_3b_medium_4x4)**

Same board as test_2b_given_4x4, but with a much larger word list.

**6 marks – 8x8 game board, ~280,000 words (test_3c_large_8x8)**

Same as above, but with a larger board!

**5 marks – 16x16 game board, ~280,000 words (test_3d_mega_16x16)**

The largest board with the largest word list. This test, when run independently, must complete in 30 seconds or less.

**50 marks total**

## Submission

Projects must be submitted ***individually***.

**Smalltalk submission:** Export your Boggle class the same way you did for Lab #2. If you wrote additional helper classes, submit those also.

**Elixir/Haskell/Rust submissions:** Submit your boggle.ex, boggle.hs, or main.rs file on D2L. You do not need to submit the entire mix/cabal/cargo project.