Assignment 2

General

General

You must read fully and carefully the assignment specification and instructions.

- **Course:** COMP20007 Design of Algorithms @ Semester 1, 2024
- **Deadline Submission:** Thursday 23rd May 2024 @ 11:59 pm
- **Course Weight:** 20%
- **Assignment type:** individual
- **ILOs covered:** 1, 2, 3, 4
- **Submission method:** via ED

Purpose

The purpose of this assignment is for you to:

- Design efficient algorithms in pseudocode.
- Improve your proficiency in C programming and your dexterity with dynamic memory allocation.
- Demonstrate understanding of data structures and designing and implementing a set of algorithms.

Task 1: Dynamic Time Warping Background

Dynamic Time Warping (DTW)

What is Dynamic Time Warping?

Dynamic Time Warping (DTW) is an algorithm primarily used in time series analysis, where the objective is to measure the similarity between two temporal sequences which may vary in time or speed. Imagine you have two audio recordings of the same speech; one is spoken quickly, and the other slowly. DTW helps to align these varying sequences to find the optimal alignment despite their differences in timing.

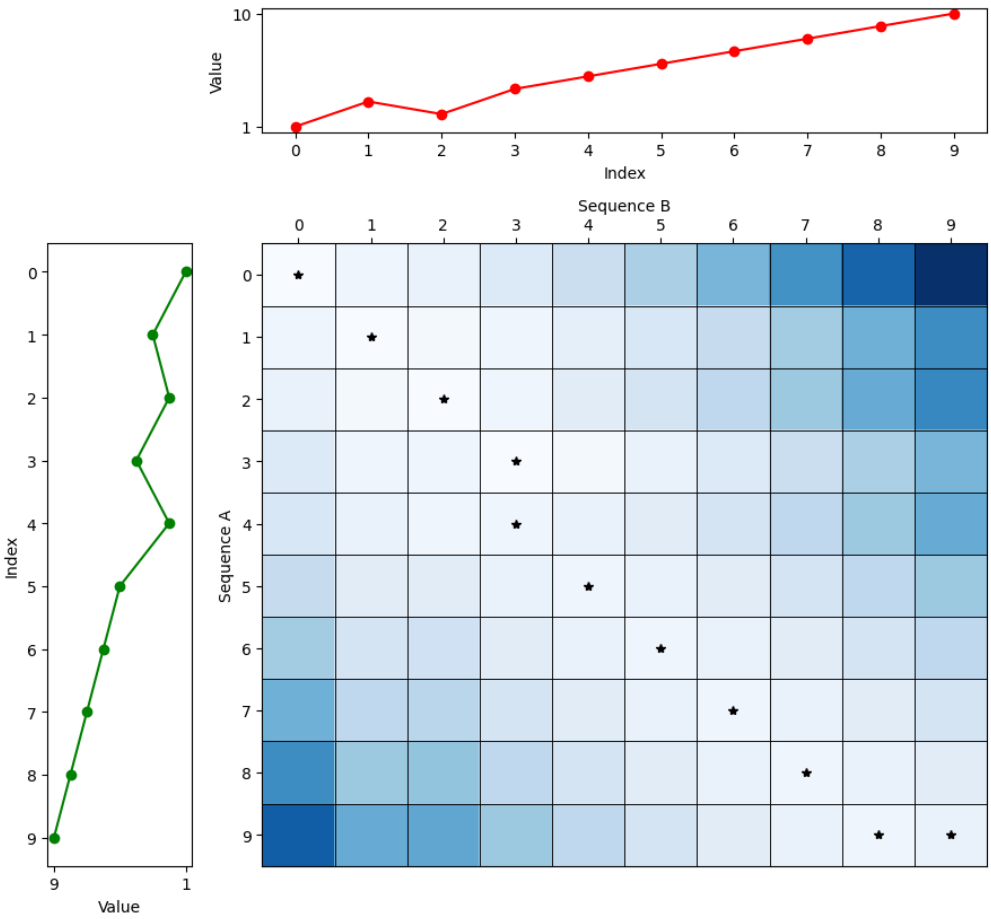
An optimal alignment in the context of DTW refers to the best possible matching between two sequences that minimises the total distance (or cost) between them. For example, in comparing two audio recordings of the same speech spoken at different speeds, an optimal alignment would adjust for these differences by stretching or compressing the time axes of the sequences. This means aligning elements from one sequence with one or more elements of the other sequence, where the sequence points that are matched may not necessarily be equidistant or ordered consecutively. The goal is to ensure that similar sounds or features that occur at different times due to variations in speed are aligned in a way that accurately reflects their underlying similarity.

More formally, DTW is a technique that calculates an optimal alignment between two given sequences with certain restrictions. The sequences are "warped" non-linearly in the time dimension to determine a measure of their similarity independent of certain non-linear variations in the time dimension. This warping allows the algorithm to find the best possible alignment by "stretching" or "compressing" segments of the sequences.

The figure below illustrates the computed DTW matrix after finding the optimal alignment between a pair of sequences. The cells in the matrix represent the cumulative distances or costs calculated for aligning Sequence A (in green colour) and Sequence B (in red colour). For the example in the figure below we use a colour map with different shades of blue colour to visually represent the magnitude of the costs. The different shades indicate the following:

- Lighter shades of blue indicate lower costs, representing closer matches or smaller differences between the two sequence elements being compared.
- Darker shades of blue indicate higher costs, indicating greater differences or less optimal alignments between the sequence elements.

The optimal alignment or "warping path" is depicted with black stars starting at position (0,0) and ending at position (9,9). The total alignment cost or distance is the value at position (9,9).



Applications of Dynamic Time Warping

Dynamic Time Warping has a variety of real-world applications across different fields. Some of the typical applications include:

- **Speech Recognition**: DTW is extensively used in speech processing to align and compare different speech patterns for recognition and verification.
- **Finance**: In financial time series analysis, DTW can compare stock market trends or economic indicators over time, aligning them despite temporal discrepancies for better correlation analysis.
- **Healthcare**: In medical signal processing, DTW helps in analyzing irregularities in heartbeat or other medical signals by comparing them against normal patterns.
- **Activity Recognition**: In wearable computing, DTW can help recognize patterns of activity based on sensor data, aligning sequences of movement to identify specific activities like walking, running, or gestures.
- **Video Processing**: DTW is used in video analysis to align sequences of video frames for motion analysis, surveillance, and content retrieval.

Dynamic Time Warping remains a powerful tool in data analysis, providing robust methods for comparing sequences that vary in time, which is a common challenge in many real-world applications.

Limitations of Dynamic Time Warping

While Dynamic Time Warping is a versatile tool for analysing sequences, it has limitations, particularly when dealing with missing values. DTW requires that each point in one sequence must be matched with one or more points in the other sequence. However, the algorithm does not inherently handle gaps or missing values in the sequences. Therefore, before applying DTW, it may be necessary to preprocess the data to impute missing values or to remove gaps, ensuring that the sequences are complete. This limitation is significant in practical applications where data might be incomplete or irregular, requiring careful preparation and validation of data before analysis.

DTW Pseudocode

```
DynamicTimeWarping(sequenceA, sequenceB):
    // Initialise the lengths of sequences
    n <- length(sequenceA)
    m <- length(sequenceB)

    // Create and initialise the DTW matrix with infinity
    dtwMatrix <- matrix[n+1][m+1] filled with infinity
    dtwMatrix[0][0] <- 0

    // Populate the DTW matrix
    for i from 1 to n:
        for j from 1 to m:
            cost <- abs(sequenceA[i-1] - sequenceB[j-1])         // Calculate cost using the absolu
            dtwMatrix[i][j] <- cost + min(dtwMatrix[i-1][j],    // Insertion
                                          dtwMatrix[i][j-1],    // Deletion
                                          dtwMatrix[i-1][j-1])  // Match

    // The DTW distance is in the bottom-right corner of the matrix
    return dtwMatrix[n][m]
```

Task 1: Dynamic Time Warping

Part A (Code)

Implement DTW: Write a program that implements the DTW algorithm. The algorithm should return the DTW matrix and the alignment cost (i.e., bottom-right cell in DTW matrix).

Part B (Written)

Describe DTW: In your own words, explain what DTW is.

Explain how DTW differs from simple sequence matching: Highlight the unique aspects of DTW that allow it to handle variations in speed and timing.

Detail the initial setup of the cost matrix: Describe what each cell represents and how the initial values are set.

Describe the process of calculating the optimal path through the cost matrix: Explain the criteria used to determine the path (e.g., minimising total cost).

Part C (Written)

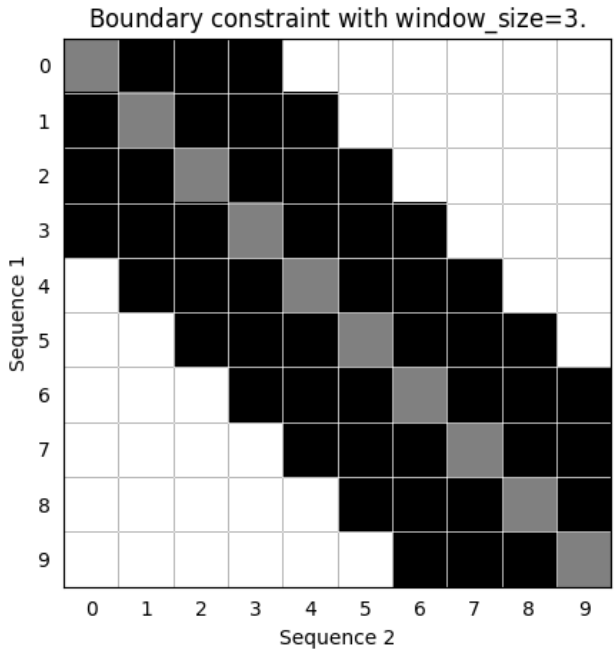
Recurrence Relation: Explain the recurrence relation used in DTW, describing how it facilitates the dynamic programming approach to find the minimal distance.

Part D (Code)

Dynamic Time Warping (DTW) is an effective method for aligning temporal sequences that vary in speed or timing. However, without constraints, DTW can produce alignments that stretch or compress the sequences excessively, potentially leading to unrealistic results. To address these issues, in Part D we introduce boundary constraints.

As shown in the figure below, a boundary constraint limits the alignment path to a window around the diagonal, with a `window_size = 3`. This means that only cells up to a distance of 3 to the right and left of the diagonal are considered in the alignment path. Both the diagonal cells (in grey) and those to the left and right of the diagonal (in black) are included in the computation. Cells outside this window (in white) are excluded, ensuring that DTW only aligns points within a reasonable distance,

maintaining plausible alignments.



Implement DTW with Boundary Constraints

Modify the standard DTW algorithm to include boundary constraints that limit the alignment to a specified window around the diagonal of the cost matrix. This window will define the maximum allowable distance in time that corresponding points on each sequence can have.

- **Implementation Details:**
  - **Window Size:** Introduce a parameter `window_size` that controls the width of the boundary around the diagonal. For each cell `(i, j)` in the DTW matrix, limit the alignment path to remain within this window band to maintain plausible matches.
  - **Matrix Calculation:** Adjust the DTW matrix computation to consider only the elements within the window constraints for each matrix cell. This involves modifying the loops that fill in the DTW matrix to skip calculations for elements outside the specified window.
  - **Output**: The algorithm should return the DTW matrix and the alignment cost (i.e., bottom-right cell in DTW matrix).

Part E (Written)

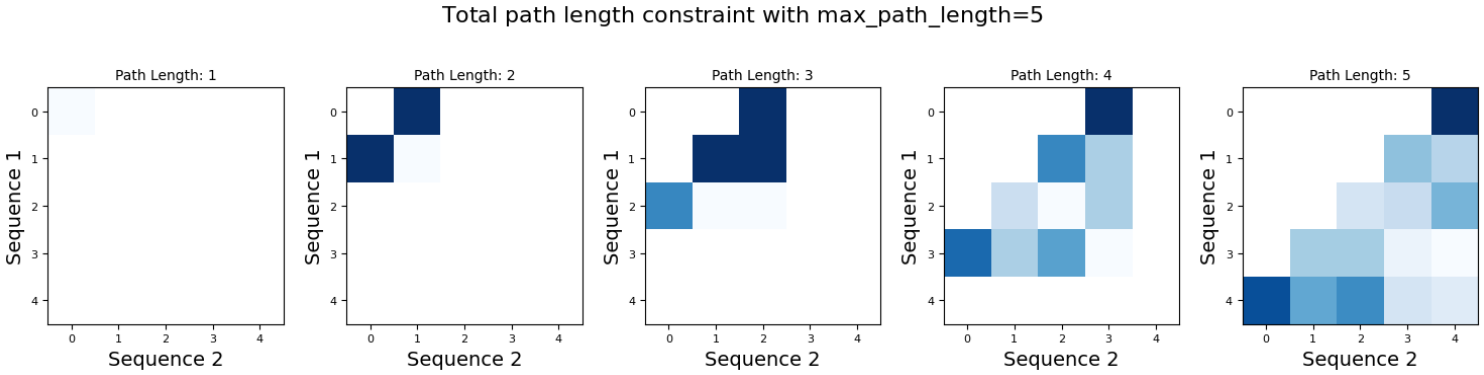
Computational complexity: Discuss the computational complexity of the algorithm implemented in Part D (i.e., DTW with boundary constraints). Is it different from the algorithm in Part A (i.e., standard DTW)? State and explain the recurrence relation.

Part F (Code)

While boundary constraints introduced in Part D prevent unrealistic alignments, they may also restrict the algorithm's ability to find the most accurate alignment under certain conditions. Rigid constraints can potentially exclude optimal paths, particularly in sequences with varying densities or irregular pacing.

Part F aims to overcome this limitation by capping the total length of the warping path (i.e., the total number of steps (insertions, deletions or matches) in the alignment path) instead of imposing rigid constraints between individual points. This approach allows the DTW algorithm to adapt more naturally to the given data (i.e., sequences), providing flexibility in alignment while maintaining control over the path complexity.

To better understand how this constraint affects alignments, the figure below illustrates how cumulative alignment costs are calculated across varying path lengths (within a defined max_path_length). Each subplot represents the alignment costs at a specific path length, starting from shorter paths (left) to longer paths (right). The alignment costs are computed for each cell by considering cumulative costs from the previous path length. For instance, a cell at position (i, j) in the matrix for a longer path length could have its cost updated based on the closest neighbours to the top, left, and top-left diagonal (i.e., insertion, deletion, match) of cell at position (i, j) from the matrix of the previous path length. This cumulative process allows DTW to track alignment paths step-by-step, capturing variations in data while preventing the warping path from exceeding the allowed total path length.



Implement DTW with Total Path Length Constraint

Modify the standard DTW algorithm to include a constraint that limits the total number of steps in the warping path. This constraint ensures a balance between maintaining sequence integrity and allowing enough flexibility for natural variations in the data.

Implementation Details:

- **Path Length Parameter:** Introduce a parameter `max_path_length` that controls the maximum allowable total steps in the alignment.
- **Output:** The algorithm should return the alignment cost. See *Path Extraction Hint* below.

Implementation Hints:

- **Iterative Logic Hint:** Consider how to evaluate alignment paths of varying lengths, from 1 up to the maximum path length allowed.
    - *How can an iterative approach capture cumulative costs effectively for each possible path length?* By iterating over incremental path lengths starting from 1, the algorithm can build solutions step-by-step and capture cumulative costs effectively for each possible path length.

- **Matrix Calculation Adjustments Hint:** Implement the DTW calculation so that it only accumulates paths within the allowed total number of steps. Use checks at each matrix cell to ensure that the path count remains within the valid indices of both sequences and does not exceed the current path length (see Figure above).

- **Data Structure Adjustment Hint:** So far, we have been tracking minimum alignment costs. Now, consider that you also need to ensure that the total steps in the alignment path do not exceed a given maximum.
    - *Is a 2D matrix still sufficient to keep track of the alignment costs while maintaining a constraint on the total number of steps?* Probably not. It seems like we need an additional dimension for the path length. What about a 3D matrix to explicitly capture alignment costs across different path lengths, or a 2D matrix where each cell is a dictionary to store cumulative costs for multiple path lengths?

- **Path Extraction Hint:** Consider how to identify the minimum cumulative cost among all feasible path lengths. If using a 3D matrix, compute the minimum cost across the third dimension at the last indices of the two sequences i.e., `(n, m)`. If using a 2D matrix with a dictionary structure, ensure that the cumulative cost is extracted across all possible path lengths.

Part G (Written)

Computational complexity: Discuss the computational complexity of the algorithm implemented in Part F (i.e., DTW with total path length constraint). Is it different from the algorithm in Part A (i.e., standard DTW)? Explain the recurrence relation.

Task 1: Assignment Submission

Upload your solution to Task 1!

Input and Output Formats

The test cases for each problem are present in the `test_cases` folder, and the expected answers for each of these test cases are present in the `test_case_answers` folder.

Part A

Part A takes two filenames at the command line:

- The first filename is the name of the file containing the first sequence,
- The second filename is the name of the file containing the second sequence.

The format of the file with the first given filename will be similar to this example:

```
-3.68E-01, -6.22E-01, -6.01E-01, -6.28E-01, -6.11E-01, -3.93E-01, -3.23E-01, -5.79E-01, -8.96E-01
```

Where all files follow the format:

- A single line listing a series of numbers specified in scientific notation, with each number separated by a comma and space.

The format of file with the second given filename will follow the same format.

These values correspond to the value of an ECG reading at a series of consecutive points in time.

The output must consist of two components:

- The first line of output must be the cost calculated for the bottom-right matrix value (e.g. the total cost of optimal alignment).
- The remaining lines must be an $n \times m$ grid (where n is the number of values in the original first sequence and m is the number of values in the second original sequence), where each row contains the optimal alignment cost found through `DynamicTimeWarping`. The $(i+1)^{th}$ row must show all values calculated for $dtwMatrix[i][j]$ separated by a space and printed to two decimal places. The first row ($i = 0$) and first column ($j = 0$) should be omitted from the output.

For the given example, if the second file contained the sequence:

```
-7.09E-01, -9.01E-01, -9.27E-01, -1.21E+00, -1.20E+00, -1.29E+00, -1.49E+00, -1.47E+00, -1.62E+00,
```

the output would be:

```
5.55
0.34 0.87 1.43 2.28 3.11 4.03 5.15 6.25 7.51 9.10
0.43 0.62 0.93 1.51 2.09 2.76 3.63 4.48 5.47 6.81
0.54 0.73 0.95 1.53 2.11 2.78 3.65 4.50 5.49 6.83
0.62 0.81 1.03 1.53 2.10 2.76 3.62 4.47 5.46 6.79
0.72 0.91 1.13 1.63 2.12 2.78 3.64 4.48 5.48 6.81
1.03 1.22 1.44 1.94 2.43 3.01 3.88 4.72 5.71 7.04
1.42 1.61 1.83 2.33 2.82 3.40 4.18 5.02 6.02 7.35
1.55 1.74 1.96 2.46 2.95 3.53 4.31 5.07 6.06 7.40
1.73 1.55 1.58 1.90 2.20 2.60 3.19 3.76 4.49 5.55
```

Part B

Add your answers to Part B to a pdf called `written-tasks.pdf`.

Part C

Add your answers to Part C to a pdf called `written-tasks.pdf`.

Part D

Part D takes two filenames at the command line which have the same meaning as in Part A, it also must take a third argument describing the `window_size` value.

- The first filename is the name of the file containing the first sequence,
- The second filename is the name of the file containing the second sequence,
- The number in the third argument is a numerical value that will be used as the `window_size`.

The output will be the final optimal alignment cost to two decimal places, with the `window_size` used to modify the algorithm as described in the previous slide.

Part E

Add your answers to Part E to a pdf called `written-tasks.pdf`.

Part F

Part F takes two filenames at the command line which have the same meaning as in Part A, it also must take a third argument describing the `max_path_length` value.

- The first filename is the name of the file containing the first sequence,
- The second filename is the name of the file containing the second sequence,
- The number in the third argument is a numerical value that will be used as the

`max_path_length`.

The output will be the final optimal alignment cost to two decimal places, with the `max_path_length` used to modify the algorithm as described in the previous slide.

Part G

Add your answers to Part G to a pdf called `written-tasks.pdf`.

You can check your submission using the "Test" button and then click the "Submit" button when you're happy with your answers. You may submit as many times as you like.

Task 2: Boggle Buddy

The game of Boggle is normally played on a 4x4 grid. 16 dice, each die printed with six characters from the letters A to Z, are placed in a closed container and the container is shaken. The dice land in the 4x4 grid and the letter on the top face is the letter for that grid square. A three-minute timer is typically started and players try and make as many words from the letters on the grid as they can before time runs out. These words have to follow certain rules - for this task we will assume the rules followed are:

- Each letter (except for the first letter) must be adjacent (horizontally, vertically or diagonally) to its preceding letter.
- A grid square's letter can only be used once (if there are two copies of the same letter appearing in different grid squares - both can be used).
- The word must be one of the allowed words.

Though there are additional rules when played as a group of players, but points are assigned for the total number of letters in all valid words made.

Part A (Code)

A regular board-game playing group have invited some tourists to play and brought Boggle. The tourists wanted to try playing but were not confident in their foreign language skills and were worried the words they'd make might not be valid for play. The group asked online if anyone would be willing to put together a system to quickly retrieve all valid words so that the tourists can quickly check the list to see that they did not make a mistake.

Part B (Code)

Hooked on the board game, the tourists purchased a copy and brought it home as a souvenir. Their children wanted to play the game with their parents, but because they are still practicing the language, they sometimes get stuck trying to find a word. Seeing a good opportunity for their children to practice a foreign language, the parents reached out again to see if anyone could put together a tool that gives a hint for the next letter that can be played to make a word.

Part C (Written)

The parents notice the program often takes time to think of solutions, but only on boggle games where the same letter appears lots of times. Given the sand timer for thinking of words gives such a short time, they ask if there's a way to speed it up, for example, if each letter can only be used once in each word (even if it appears more than once on the board). How might their rule modification change the worst case complexity of finding all words?

Part D (Code)

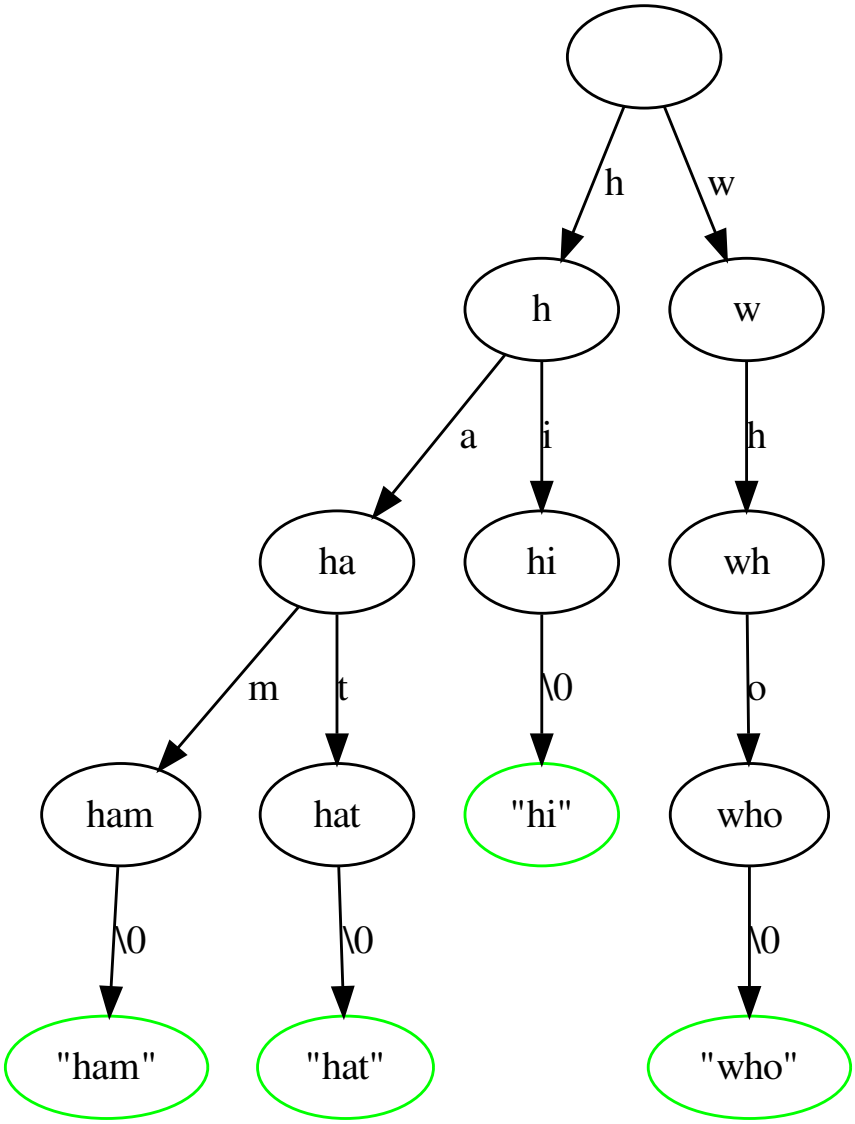
Hearing that the improvement might be promising, they ask you to put together the tool giving all valid words with the added rule.

Task 2: Prefix Tries and Boggle Graphs

Background - Prefix Tries

For Task 2, you will need to build a prefix trie. This is a data structure where, rather than containing a full key in each node, each edge in the trie represents a particular letter. Since each edge is associated with a character, both checking whether a given string is in the tree and finding *all* strings that match a given prefix are efficient operations.

For example, to store the strings "ham", "hat", "hi" and "who", our prefix tree might look like:



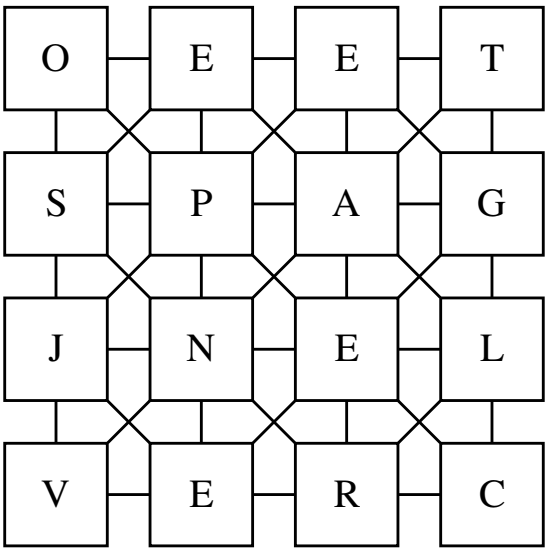
For simplicity, the tree will:

- Contain a pointer for each possible following character - even if that character is unlikely to appear.
- Also store the `\0` character used as the delimiter when storing the string in the tree.

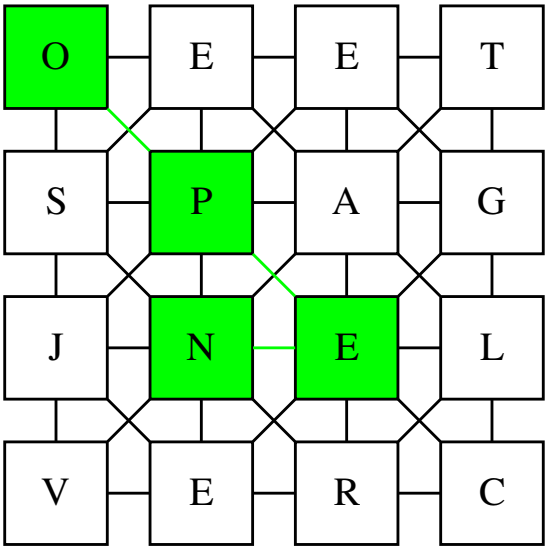
This means every node will contain 256 pointers - the majority of most of which are typically set to `NULL`. This also means each leaf of the tree will correspond to the completion of an inserted string - with completion of strings occurring nowhere else in the tree.

Background - Boggle Graphs

For a given set of dice, a graph can be constructed showing the connections.



This graph can then be used to work out what are valid words to play. For example, the word "open" could be played, starting from `0`:



Part A (Code)

To support checking you will search simultaneously through the prefix tree and the boggle graph. Returning the list of all words starting from each position. You must use the prefix tree to limit the available locations to travel to next. You will likely find it useful to temporarily mark seen words as you travel through the prefix tree (and then unmark these at the end of the search).

Part A will take two filenames at the command line:

- The first filename is the name of the dictionary of words which are allowed for that game.
- The second filename is the name of the board used for boggle.

The format of the file with the first given filename will be similar to this example:

```
10
cleaner
clean
lean
opera
open
pants
panel
pager
speak
team
```

Where all files follow the format:

- The first line specifies the number of words in the dictionary (10 in this example)
- All following lines specify words in the dictionary.

The format of the file with the second given file name will be similar to this example:

```
O E E T
S P A G
J N E L
V E R C
```

Where the board this file represents matches the example board above. The format of the file will always be 16 capital letters, arranged in a 4-by-4 grid. When determining whether letters on the board can be used to make words, you must ignore capitalisation.

The output must be the list of words (alphabetically ordered (breaking ties by length)) that can be made on the board which follow the boggle rules stated earlier. For the given example this would be:

```
clean
cleaner
lean
open
pager
panel
```

Part B (Code)

In Part B, the file inputs are the same, but an additional input is given on the command line through `stdin`. This is the word constructed so far - you must output the list of letters following the given input that can follow on the boggle board as a hint.

For example, for the same input filenames as in Part A, if the input `pa` were given through `stdin`, the output printed to `stdout` must be:

```
g
n
```

If the word can be terminated, a blank space ( ) character should also be printed on its own line. For example, if the input `clean` were given through `stdin`, the output printed to `stdout` must be:

```

e
```

Part B Notes

- Where the same letter is present as the following letter (e.g. if `cle` were given), it must be output only once.
- Any non-letter character must be printed as a space.
- If the stem given is not a prefix in the dictionary, you must output nothing to `stdout` (e.g. if `opa` were given, `stdout` must not be written to, as no words in the dictionary match that starting stem).
- If the word cannot be made on the board, its next letter must not be printed (e.g. if `spe` were given, `a` should not be output, as `speak` cannot be made on the boggle board).

Part C (Written)

In Part C, you must create a pdf format document called `written-tasks.pdf`, which explains the impact of only allowing each letter to appear once in each word (regardless of how many times the letter appears on the board). Your answer must state an upper-bound on the time complexity reflecting the impact of this change, with each term used explained clearly.

In order to avoid trivial answers, you must assume the board could be extended arbitrarily to higher dimensions (e.g. 5x5 and beyond) and that the alphabet used could increase in size (e.g. the maximum length of a word is not 26 letters).

Part D (Code)

In Part D, you must put together code that takes input in the same format as Part A, but which takes advantage of allowing each letter to appear only once in the word.

For the inputs given in Part A, the output for Part D would be:

```
clean
lean
open
panel
```

```
pager
```

Task 2: Assignment Submission

Upload your solution to Task 2!

Input Format

The test cases for each problem are present in the `test_cases` folder, and the expected answers for each of these test cases are present in the `test_case_answers` folder.

Part A

The input for Part A comprises a board and dictionary.

Example

Board:

```
O E E T (row of first four letters)
S P A G (row of second four letters)
J N E L (row of third four letters)
V E R C (row of fourth four letters)
```

Dictionary:

```
10 (number of words in the dictionary)
cleaner (word 0)
clean (word 1)
lean (word 2)
opera (word 3)
open (word 4)
pants (word 5)
panel (word 6)
pager (word 7)
speak (word 8)
team (word 9)
```

Part B

The input for Part B on the command line is the partial string for which completions that can be made on the board (that are in the dictionary) are returned for. The other inputs are the same as Part A.

Example

```
pa
```

Part C

Add your answers to Part D as a pdf called `written-tasks.pdf`.

Part D

The input for Part D is the same as Part A.

You can check your submission using the "Test" button and then click the "Submit" button when you're happy with your answers. You may submit as many times as you like.

Academic Honesty

This is an individual assignment. The work must be your own work.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as doing this without proper attribution is considered plagiarism.

If you have borrowed ideas or taken inspiration from code and you are in doubt about whether it is plagiarism, provide a comment highlighting where you got that inspiration.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that it will be necessary to write pseudocode in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late Policy

Late Policy

The late penalty is 20% of the available marks for that project for each working day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! Frequently asked questions about the project will be answered on Ed.

Requirements: C Programming

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- Your code should be easily extensible to multiple data structure instances. This means that the functions for interacting with your data structures should take as arguments not only the values required to perform the operation required, but also a pointer to a particular data structure, e.g. `search(dictionary, value)`.
- Your implementation must read the input file once only.
- Your program should store strings in a space-efficient manner. If you are using `malloc()` to create the space for a string, remember to allow space for the final end of string character, '`\0`' (`NULL`).
- Your approach should be reasonably *time efficient*.
- Your solution should begin from the provided scaffold.

Hints:
• If you haven't used `make` before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces.
• It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule — if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Mark Breakdown

There are a total of 20 marks given for this assignment.

Your C programs for both tasks should be accurate, readable, and observe good C programming structure, safety and style, including documentation. Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names.

Note that marks related to the correctness of your code will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your code marks will otherwise be determined by test cases.

Mark Breakdown

Task	Mark Distribution
Task 1	2 Marks (Part A) + 2 Marks (Part B) + 1 Marks (Part C) + 2 Marks (Part D) + 1 Mark (Part E) + 2 Marks (Part F) + 1 Mark (Part G)
Task 2	2 Marks (Part A) + 2 Marks (Part B) + 1 Mark (Part C) + 4 Marks (Part D)

Note that not all marks will represent the same amount of work, you may find some marks are easier to obtain than others. Note also that the test cases are similarly not always sorted in order of difficulty, some may be easier to pass than others.

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.