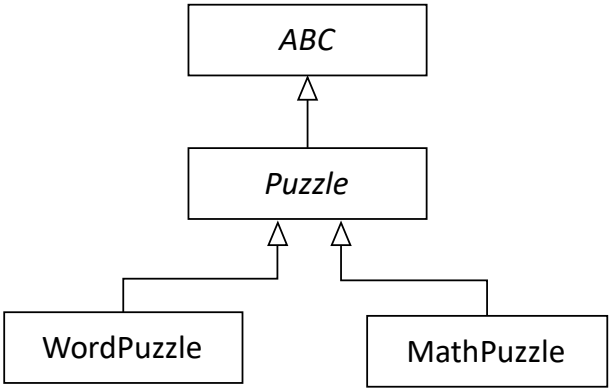
Assignment: Word and Math Puzzles

Learning Outcomes

- String manipulation, slicing, nested lists and list comprehensions, list membership test
- OOP: constructors, instance methods, class variables, abstract methods, single inheritance
- File input, exception handling

Introduction

In this assignment, you are required to implement a single program (puzzle.py) for solving both word puzzles and math puzzles on a 2D game board. The program must define the following class hierarchy:



The superclass `Puzzle` is an *abstract base class (ABC)* which defines an *abstract method* called `solve()`. Two concrete subclasses named `WordPuzzle` and `MathPuzzle` are derived from `Puzzle`. They override the parent `solve()` method with their own versions for solving the puzzle with different logics – one is to locate all correct English words on the game board while another is to find the maximum product of a fixed-size list of adjacent numbers on the board.

We will divide the discussion into Part A and Part B.

Part A: Word Puzzles

Suppose that you are given a word puzzle represented by a 2D grid or array of alphabets. Solving the puzzle means finding all correct words in the grid by checking against an English dictionary (a list of words loaded from a text file). For example, given the following array in Figure 1(a), your program is to find words from the dictionary that can be spelled out in the puzzle by starting at any character, then moving in a straight line right or down. In other words, the program performs two scanning phases: (1) **horizontal scanning** from left to right on every row, starting from row 0; and (2) **vertical scanning** from top to bottom on every column, starting from column 0. Horizontal scanning is done before vertical scanning. For example, in phase (1), the word “walk” is found at row 3 as Figure 1(b) depicts; in phase (2), the same word “walk” is found in a vertical sense at column 1 in Figure 1(c). Word matching in diagonal senses is not required in this program.

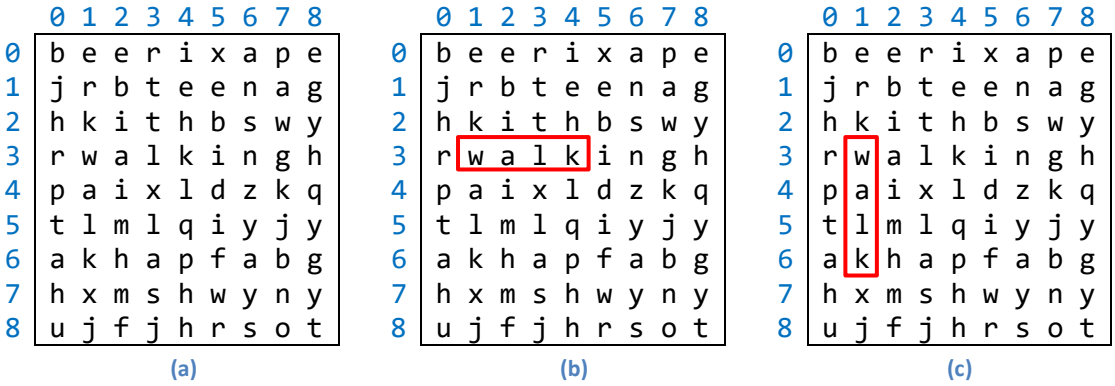


Figure 1: An example word search puzzle

Figure 2(a) and (b) show all the words found in phases (1) and (2) respectively. Figure 2(c) shows the combined result of the two phases (words found highlighted in red). Since our output is displayed on a colourless console screen, we will capitalize letters of each word found in the program output.

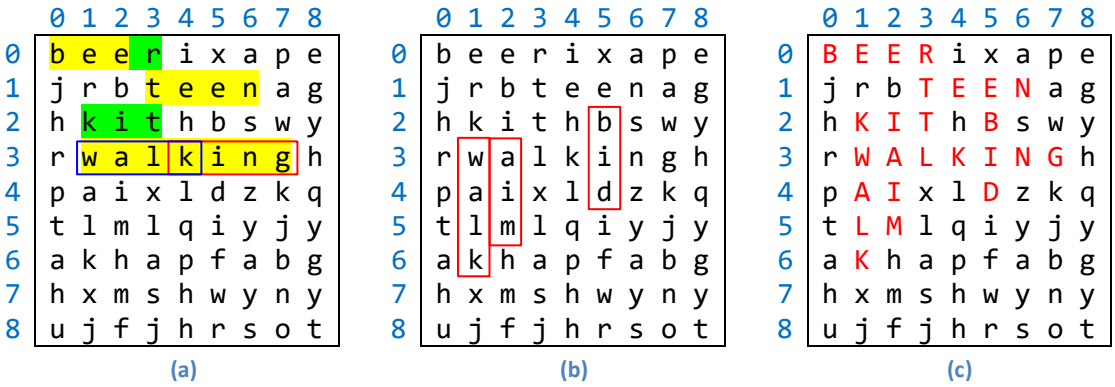


Figure 2: An example solved word puzzle

The program will also list every word found and the row or column it is located as follows:

```
row 0: bee
row 0: beer
row 1: teen
row 2: kit
row 3: walk
row 3: walking
row 3: king
col 1: walk
col 2: aim
col 5: bid
```

For the exact format of the output required, please refer to the Sample Runs section. What's worth noting is that some word may be comprised of other words. For example, the word "walking" contains two other words "walk" and "king". These should be counted as three matches.

Your task in this part is to define a class `WordPuzzle` implementing an instance method called `solve()` to search for all these valid words by scanning the grid and checking if each candidate word is in the dictionary.

Part B: Math Puzzles

Apart from word-search puzzles, this program is to support another type of puzzles which is to solve a math problem.

For example, consider the following 20 × 20 grid:

```
73 24 11 28 77 12 18 28 14 56 49 02 20 64 51 88 11 43 15 50
22 71 11 17 91 36 27 02 41 97 26 25 90 92 95 22 72 53 63 29
87 88 52 50 91 45 35 53 95 23 92 23 28 11 57 54 93 71 49 29
81 46 56 50 40 12 61 20 67 19 09 94 34 76 04 40 48 75 91 76
38 42 28 19 04 07 08 59 90 93 18 82 17 88 43 18 84 46 32 82
38 59 94 75 61 32 70 75 82 23 87 82 59 86 08 70 49 47 54 03
15 77 54 88 71 31 21 04 60 36 91 52 05 04 77 05 02 24 39 25
35 49 13 39 90 48 46 84 91 25 33 12 35 29 50 92 68 58 65 44
02 26 16 82 18 85 10 50 20 56 08 09 61 94 75 82 05 32 65 43
70 33 88 35 71 03 76 95 13 67 67 70 05 41 26 89 64 04 78 43
64 15 98 36 65 41 37 36 48 11 30 84 49 42 49 61 87 88 03 83
40 11 93 02 45 87 14 95 13 49 36 48 80 66 52 43 87 95 21 79
35 12 66 93 11 57 57 28 83 32 84 48 49 29 04 30 52 67 23 07
59 92 08 85 03 23 82 79 51 60 20 61 99 56 34 54 31 21 20 57
42 65 68 88 37 90 09 57 93 76 68 24 48 18 62 02 55 69 04 33
80 67 68 92 64 62 07 78 84 14 30 41 31 81 92 32 53 56 48 22
86 58 12 14 87 67 52 21 31 10 48 40 17 16 05 32 36 28 49 85
44 84 35 17 20 67 48 47 33 34 64 69 10 45 64 37 57 86 47 40
89 69 55 67 07 08 21 90 63 09 78 73 13 81 76 03 30 87 98 76
46 73 69 09 84 51 76 60 68 82 68 98 88 30 80 51 14 49 34 44
```

What is the greatest product of any four adjacent numbers in the same direction (i.e., horizontally, vertically, and diagonally) in the grid? And where it happens in the grid?

The solution to this grid is highlighted in blue: 93 × 85 × 88 × 92 = 63998880. In zero-based indexing, the first number (93) composing this maximum product is located at the point (12, 3) – row index 12, column index 3.

Your task in this part is to define a class `MathPuzzle` implementing an instance method called `solve()` which scans the 2D array (a nested in Python) to compute and record the maximum product of every *k* adjacent numbers (*k* = 4 for the above example). The method returns a 3-tuple holding the maximum product value, the row index and column index of the starting element which forms the product.

Hint: to solve this puzzle, you may scan every point in the grid (say, in a left to right, and top to bottom manner); taking it as the starting point, compute the product of *k* numbers in adjacency of it in four directions. Take the numbers in red as an example:

- horizontal: 75 × 82 × 23 × 87 = 12306150 (current max.)
- vertical: 75 × 4 × 84 × 50 = 1260000
- diagonal: 75 × 60 × 25 × 8 = 900000
- diagonal (reversed): 75 × 21 × 48 × 18 = 1360800

Record the current maximum product and the array indexes of the starting element corresponding to the maximum product.

Data Files

In this assignment, the program needs to read some text files to obtain the input data. There are two types of files needed:

Dictionaries

The dictionary used for checking words in this program is contained in a text file of the format below. On the first line (the header line), the two numbers denote the total number of words and the length of the longest word in the dictionary. All words are stored on second line through the last line.

```
3964 10
act
add
age
...
bad
bag
...
able
acid
...
baby
back
...
zone
zoom
about
above
...
youth
zebra
zippy
abroad
absorb
...
xenophobia
xylography
```

You may assume that the list of dictionary words has always been sorted in (1) ascending word length, and (2) then in alphabetic order. So, shorter words always appear earlier than longer words. For words of the same length, they are sorted in lexicographical order, e.g. “act” < “add”, “youth” < “zebra”, etc.

There are two given sample dictionary files, namely `dict.txt` and `dictbig.txt`, for your program testing. The first file contains 3964 words; the longest word in it has 10 letters. The second file has 370103 words and 31 letters in its longest word. Your program should prompt the user to choose a dictionary file at the very beginning of the runtime. You need NOT test the program against dictionary files other than these two provided dictionary files.

Puzzles

A puzzle in form of a 2D board is contained in a data file in the format presented as follows. Word and math puzzles share the same file format. On the first line (we called it the *header line*), there are two numbers denoting the dimensions (i.e., # rows and columns) of the "puzzle game board".

Word Puzzle Files (wordnn.txt)

The box below shows the content of a sample word puzzle file. The first line stores 8 and 12, meaning that there are 8 rows (or in other words, 8 letters on each column) and 12 columns (or in other words, 12 letters per row). All the space characters in this file will be ignored when the data is being loaded into a 2D array in your program (you may split the data using the space as separator).

```
8 12
b e a r i h a p p y e n
j v s t e e n a g e r s
x i t t h n s w y d x c
r r u l y e n g h o s t
p t d x l d z k q b t e
t u i l q i y j x f w r
a e o a p f a i t h e r
h x m s h w y n o t e a
```

Math Puzzle Files (mathnn.txt)

The box below shows the content of a sample math puzzle file. In this example, there are 12 rows and 17 columns in the puzzle.

```
12 17
92 41 30 88 70 67 98 78 64 52 75 17 38 30 59 89 27
49 43 76 68 62 56 94 89 94 98 89 32 65 71 70 51 27
69 49 90 32 41 62 61 48 09 33 57 63 95 07 42 61 04
37 75 75 39 36 19 35 72 77 49 04 09 35 02 26 81 88
81 89 45 99 07 84 41 89 60 35 53 26 99 28 87 48 95
47 94 77 17 97 93 52 97 34 46 05 60 31 92 11 95 05
52 22 27 61 27 54 14 50 67 53 11 57 94 65 34 88 92
15 86 59 16 99 19 24 81 63 69 85 13 26 15 72 58 81
62 03 67 65 86 26 84 45 89 74 48 84 67 78 39 74 64
11 12 76 80 51 23 19 09 82 97 27 10 39 73 71 04 58
42 64 27 93 71 40 45 02 46 49 88 79 66 81 10 91 24
13 15 16 83 64 10 96 97 63 73 76 47 53 77 38 85 28
```

Notes:

The header line is provided for your convenience of programming. Whether to make use of the header line to get the number of rows and columns of the puzzle is up to your program design. It is easy to count them yourself by other ways such as calling the `len()` and `max()` functions.

Program Specifications

This section describes the requirements, assumptions, hints and suggested program flow.

Requirements

1. Your program has to repeat prompting the user for the input file name of a new puzzle to solve until the user responds with a character 'n' or 'N' (implying not to continue playing the game).
2. Your program must be able to support puzzles of different dimensions (variable across puzzle files) instead of just one hardcoded dimension.
3. Exceptions like I/O errors must be handled properly. For example, if the specified input file cannot be found, a custom error message is printed instead of letting the program crash.
4. Your program should follow the inheritance hierarchy described on p.1. Create an abstract base class named `Puzzle` defining an abstract method `solve()`, and subclasses named `WordPuzzle` and `MathPuzzle` that override `solve()` with their own versions for solving the puzzle with their logics. Note that in Python, the parent method can be overridden by a subclass's method of the same name but with a different parameter list.
5. *Your program output should be exactly the same as that produced by our given sample executable program* (same text, letter case, spacings, line breaks, etc.) except for possible trailing space(s) on each line. See the Sample Runs section for more details.

Assumptions

1. This assumption on file naming rules always holds: all word puzzle files are named "wordnn.txt" and all math puzzle files "mathnn.txt" where nn = 01, 02, 03, … You may make use of this fact to distinguish word puzzles from math puzzles when handling them in your program.

For word puzzles:

2. The dictionary file will be loaded once only at the program start. The same dictionary will be used for solving all word puzzles until the program ends.
3. Word puzzle and dictionary files consist of only lowercase letters.
4. All elements in the 2D array representing a word puzzle are alphabet strings of one character only.
5. All valid words must consist of at least 3 letters, so we won't match words like "a", "an", "in", "at", "of" that are considered too short. In other words, the word to search at a time is always 3 to *cols* letters long where *cols* refers to the number of columns of the word puzzle. You may define a global constant like below near the top of your program to allow easy scaling in the future:

```
MIN_WORD = 3    # length of the shortest word to search
```

For math puzzles:

6. For math puzzles, all elements in the array are of 2 digits at most, ranging between 1 to 99.
7. There is only one solution for each puzzle (the largest product happens at one location only).
8. Recap that the goal is to find the maximum product of every *k* adjacent numbers in the 2D array. By default, set *k* = 4. Again, you may define a global constant like below for future tuning:

```
FACTORS = 4     # number of factors in the product
```

Hints

Recap that for a word puzzle, we need to (1) scan horizontally from left to right on every row; and (2) scan **vertically** from top to bottom on every column. Each of these two scanning processes is a deeply nested loop. However, their logic is pretty similar. Do we need to write two sets of highly redundant code to handle the similar word searching processes? The answer is no. We can write such a nested loop once only in a method for doing the horizontal scanning on the puzzle array. Then for the vertical scanning, we can just reuse the same method but on a *transposed* version of the puzzle array.

In linear algebra, transpose is an operation that flips a matrix by switching its rows with its columns. Suppose that `a` is a `rows`-by-`cols` array. Transposing the array `a` will create a `cols`-by-`rows` array `b`, and transpose means that the rows of the source array become columns of the destination array as depicted in Figure 3 below.

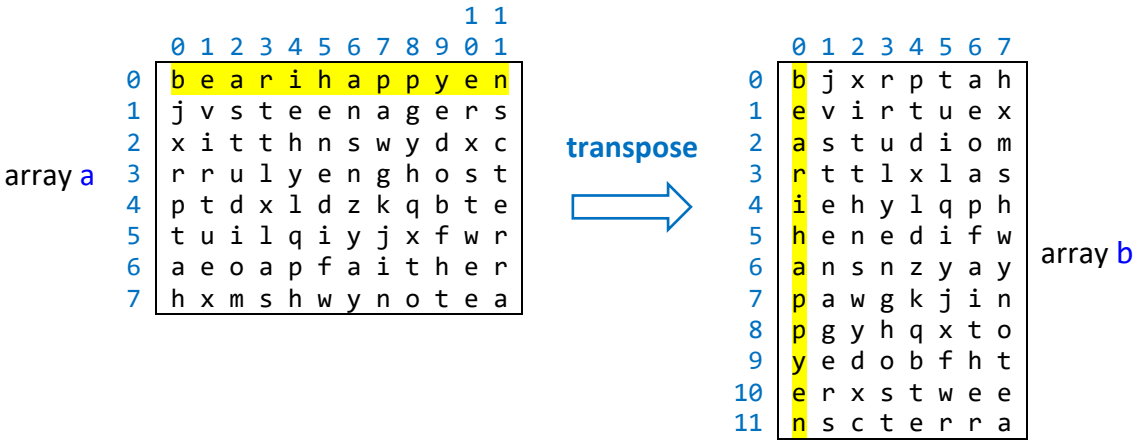


Figure 3: Transpose of an 8-by-12 array

There are many ways to implement a transpose function. To save your coding effort, we have provided below perhaps the most concise way to do so in Python:

```
def transpose(a):
    return list(map(list, zip(*a)))
```

While you are encouraged to do more self-study for why it works, you can simply use it like a black box; you may also implement your own one based on more intuitive looping techniques.

Why we need this function is that we don't want to implement another search method to do vertical scanning. Suppose that we have implemented a method for horizontal scanning. We can simply reuse it to scan vertically by passing to it a transposed puzzle array. This method may need to define an additional parameter, say `vertical` (may be given a default value `False`), which is used to allow the caller to signify (by passing `True`) that the search is now in a vertical scanning phase, and some behavior may need to change, e.g., replace "row" by "col" in the printout, if necessary.

Note also that transposing an array twice just returns the original array unchanged.

As another hint, think about how to reuse code. The logics for reading different types of files in this assignment should be very similar. You may create one function for file reading. Likewise, printing the two types of puzzles is more of less the same code except for subtle difference in the formatting requirement like zero padding for numbers. You may define one method for printing the puzzle array in the base Puzzle class, and control subtle formatting via parameter passing.

Suggested Program Flow

The program flow is described as follows. You may define whatever functions or methods as you see fit to ease your implementation.

1. The program starts asking the user to enter the file name of a dictionary file.
2. Load the data of the dictionary file into the dictionary array (a Python list).
3. Ask the user to enter the file name of a puzzle.
4. Check the file name to see if it is a word puzzle or a math puzzle.
5. Load the data of the puzzle file into a puzzle array (a nested Python list).
6. Print the puzzle array.
7. If it is a word puzzle, create a WordPuzzle object. You may pass the puzzle array (of letters) and dictionary array as arguments to its constructor. Similarly, create a MathPuzzle object for a math puzzle file and pass the puzzle array (of integers) to its constructor.
8. Call the solve() method.
   For a word puzzle:
   8.1 Perform horizontal scanning on the puzzle array for each possible word (from shortest to longest) by looking up the word from the dictionary array.
      a. for each search word **w** at a location in the puzzle array,
         i. check if **w** is in the dictionary array (the membership test operator `in` may be used).
         ii. if found, print **w** with its location (which row it lies) and capitalize its letters in the puzzle array.
      b. increment the search word length and repeat (a) until it hits the puzzle's width.
      c. move on to the next position in the same row, and repeat (a)-(b).
      d. move on to the next row, and repeat (a)-(c).
   8.2 Transpose the puzzle array.
   8.3 Perform step 8.1 again, but printing "col" instead of "row" for the location of each word found. This in effect achieves the vertical scanning phase.
   8.4 Transpose the puzzle array again.
   For a math puzzle,
   8.1 For every value at index `[y][x]` in the puzzle array, extend to the adjacent points `[y][x+s]`, `[y+s][x]`, `[y+s][x+s]` and `[y+s][x-s]`, where s is a shift value between 0 and $k - 1$ (recall that $k = 4$ by default) being controlled by a loop, to collect four products in the four directions (horizontal, vertical, diagonal, diagonal reversed).
   8.2 Record the maximum of these four products and the x, y values.
9. Print the solved puzzle array for a word puzzle or the solution tuple for a math puzzle.
10. Keep asking the user whether to continue on another puzzle if the reply is not 'y' or 'n' (case-insensitive).
11. Repeat steps 3-10 if the user answers 'y' (yes). Or quit the program if answering 'n' (no).

Note: The above is just a suggested program flow. Your steps and the detailed implementation can deviate from it as long as your program generates correct output.

Sample Runs

In the following sample runs, the blue text is user input and the other text is the program printout. A sample program executable has been provided on Blackboard. You can use it to test for other inputs. Assume that files xyz and abc below are not existent. It is normal that accidental mixing up of dictionary files and puzzle data files may crash the program. You need not test against this issue.

```
Enter dictionary file name: xyz↵
Failed to open file xyz
(Program ends normally)
```

```
Enter dictionary file name: dictbig.txt↵
Dictionary loaded with 370103 words
Enter puzzle file name: abc↵
Failed to open file abc
(Program ends normally)
```

```
Enter dictionary file name: dict.txt↵
Dictionary loaded with 3964 words
Enter puzzle file name: word01.txt↵

Puzzle:
b e e r i x a p e
j r b t e e n a g
h k i t h b s w y
r w a l k i n g h
p a i x l d z k q
t l m l q i y j y
a k h a p f a b g
h x m s h w y n y
u j f j h r s o t

Words found:
row 0: bee
row 0: beer
row 1: teen
row 2: kit
row 3: walk
row 3: walking
row 3: king
col 1: walk
col 2: aim
col 5: bid

Puzzle solved:
B E E R i x a p e
j r b T E E N a g
h K I T h B s w y
r W A L K I N G h
p A I x l D z k q
t L M l q i y j y
a K h a p f a b g
h x m s h w y n y
u j f j h r s o t

Continue with next puzzle? (y/n): y↵
Enter puzzle file name: word08.txt↵
```

```
Puzzle:
r t t e t g g o s
a o n m d p t m n
i f y e h e r e l
o r d e o o d e n
o o d e t e o d e
i r d t t a l r n
c i c t b t t a e
o v e e d e n c p

Words found:
row 2: her
row 2: here
col 4: hot
col 5: eat

Puzzle solved:
r t t e t g g o s
a o n m d p t m n
i f y e H E R E l
o r d e O o d e n
o o d e T E o d e
i r d t t A l r n
c i c t b T t a e
o v e e d e n c p

Continue with next puzzle? (y/n): y↵
Enter puzzle file name: math01.txt↵

Puzzle:
73 24 11 28 77 12 18 28 14 56 49 02 20 64 51 88 11 43 15 50
22 71 11 17 91 36 27 02 41 97 26 25 90 92 95 22 72 53 63 29
87 88 52 50 91 45 35 53 95 23 92 23 28 11 57 54 93 71 49 29
81 46 56 50 40 12 61 20 67 19 09 94 34 76 04 40 48 75 91 76
38 42 28 19 04 07 08 59 90 93 18 82 17 88 43 18 84 46 32 82
38 59 94 75 61 32 70 75 82 23 87 82 59 86 08 70 49 47 54 03
15 77 54 88 71 31 21 04 60 36 91 52 05 04 77 05 02 24 39 25
35 49 13 39 90 48 46 84 91 25 33 12 35 29 50 92 68 58 65 44
02 26 16 82 18 85 10 50 20 56 08 09 61 94 75 82 05 32 65 43
70 33 88 35 71 03 76 95 13 67 67 70 05 41 26 89 64 04 78 43
64 15 98 36 65 41 37 36 48 11 30 84 49 42 49 61 87 88 03 83
40 11 93 02 45 87 14 95 13 49 36 48 80 66 52 43 87 95 21 79
35 12 66 93 11 57 57 28 83 32 84 48 49 29 04 30 52 67 23 07
59 92 08 85 03 23 82 79 51 60 20 61 99 56 34 54 31 21 20 57
42 65 68 88 37 90 09 57 93 76 68 24 48 18 62 02 55 69 04 33
80 67 68 92 64 62 07 78 84 14 30 41 31 81 92 32 53 56 48 22
86 58 12 14 87 67 52 21 31 10 48 40 17 16 05 32 36 28 49 85
44 84 35 17 20 67 48 47 33 34 64 69 10 45 64 37 57 86 47 40
89 69 55 67 07 08 21 90 63 09 78 73 13 81 76 03 30 87 98 76
46 73 69 09 84 51 76 60 68 82 68 98 88 30 80 51 14 49 34 44

Puzzle solved:
(63998880, 12, 3)

Continue with next puzzle? (y/n): y↵
Enter puzzle file name: math05.txt↵

Puzzle:
```

```
92 41 30 88 70 67 98 78 64 52 75 17 38 30 59 89 27
49 43 76 68 62 56 94 89 94 98 89 32 65 71 70 51 27
69 49 90 32 41 62 61 48 09 33 57 63 95 07 42 61 04
37 75 75 39 36 19 35 72 77 49 04 09 35 02 26 81 88
81 89 45 99 07 84 41 89 60 35 53 26 99 28 87 48 95
47 94 77 17 97 93 52 97 34 46 05 60 31 92 11 95 05
52 22 27 61 27 54 14 50 67 53 11 57 94 65 34 88 92
15 86 59 16 99 19 24 81 63 69 85 13 26 15 72 58 81
62 03 67 65 86 26 84 45 89 74 48 84 67 78 39 74 64
11 12 76 80 51 23 19 09 82 97 27 10 39 73 71 04 58
42 64 27 93 71 40 45 02 46 49 88 79 66 81 10 91 24
13 15 16 83 64 10 96 97 63 73 76 47 53 77 38 85 28

Puzzle solved:
(77067592, 1, 6)

Continue with next puzzle? (y/n): not sure↵
Continue with next puzzle? (y/n): yes↵
Continue with next puzzle? (y/n): Y↵
Enter puzzle file name: word02.txt↵

Puzzle:
b e a r i h a p p y e n
j v s t e e n a g e r s
x i t t h n s w y d x c
r r u l y e n g h o s t
p t d x l d z k q b t e
t u i l q i y j x f w r
a e o a p f a i t h e r
h x m s h w y n o t e a

Words found:
row 0 : bear
row 0 : ear
row 0 : happy
row 1 : teen
row 1 : teenage
row 1 : teenager
row 1 : age
row 3 : ghost
row 3 : host
row 6 : faith
row 6 : the
row 7 : not
row 7 : note
col 1 : virtue
col 2 : studio

Puzzle solved:
B E A R i H A P P Y e n
j V S T E E N A G E R s
x I T t h n s w y d x c
r R U l y e n G H O S T
p T D x l d z k q b t e
t U I l q i y j x f w r
a E O a p F A I T H E r
h x m s h w y N O T E a

Continue with next puzzle? (y/n): no↵
```

```
Continue with next puzzle? (y/n): n↵

(Program ends normally.)
```

Program Testing

We have provided some puzzle data files for your program testing. You may also create your own ones based on our specified format for further testing. For dictionary files, you may simply use either `dict.txt` or `dictbig.txt` that we provided. The above sample output assumes that the files are in a right folder that your program executable can locate and open.

Submission and Marking

- Your program file name should be `puzzle.py`. Submit this file only (skip all those data files) in Blackboard (https://blackboard.cuhk.edu.hk/).
- Insert *your name*, *student ID*, and *e-mail* as comments at the beginning of your source file.
- You can submit your assignment multiple times. Only the latest submission counts.
- Your program should *include suitable comments as documentation*.
- ***Do NOT plagiarize.*** Sending your work to others is subject to the same penalty as the copying student will.
- Detailed marking scheme is not available. But the mark weightings on Part A and Part B are roughly in a proportion of 60% (Part A) and 40% (Part B). In this assignment, we will not just look at program correctness, but also the OOP structure of your code, and coding style (naming convention and program documentation). So, certain deductions would be made on poor code structure (e.g., without the mentioned inheritance hierarchy), heavily redundant code, incorrect naming conventions and missing of helpful comments in your code.