Q1) You are required to implement a suitable string-matching algorithm to offer the user the ability to find a pattern in a text when the pattern contains "wild-cards". A wild-card is a special "symbol" which is meant to match any letter. For this assignment only, the underscore is considered as a wildcard. For concreteness, it is assumed that the text does not contain the "_"-symbol (the underscore). For illustration, the "pattern" **c_g** can be found in the following "text" **cogwrgaccag** two times (at the beginning and at the end).

1. The pattern should be allowed to contain more than one wild-card (i.e., more than one "_" and for example c_ _g, c_g_, c_ _g_).

2. Your algorithm should find all positions where the *pattern* is found.

How to Deliver

- Your solution should contain
  - Compilable (and afterwards runnable) source file(s) of your implementation.

  - You should convince the examiner, best in a concise and clear manner, that your solution does the job. That should be based on two things:

    ➢ **An explanation:** justifying the selection of the string-matching algorithm in a separate *Readme* file**.** However, a lengthy discussion of the principles of the selected string-matching algorithm is not needed. Concentrate on your solution and highlight the special points of the code. Salient comments in the source code may also help. However, a general explanation should be in the *Readme* file, not part of the comments.

    ➢ **Test-data**: Convince the examiner by preparing some test runs. Provide instructions as to how to execute the tests. As a suggestion: make a small number of tests. Therefore, provide **n** pairs of *Pattern* and *Text*, for instance pattern1.txt, text1.txt, pattern2.txt, text2.txt etc. and produce files patternmatch1.output, patternmatch2.output. Make meaningful test cases (for instance, try also the empty pattern/text).

Q2) Write a program to search a database of DNA sequences, represented as strings of characters, to find matches of other DNA subsequences. Thus, the program should take as input two sets of strings: the Database strings and the Query strings. These two sets of strings are referred to as the database and the query base respectively. Both the DNA database and the query strings are made up of only four characters: A, C, G and T and the query strings are of different lengths. The output must report the location of an exact match within any given input DNA sequence for each input search query string. If the query string matches within multiple sequences within the database, each result must be reported; and if the query string matches multiple locations within the same database sequence, the earliest position that matches exactly must be reported.

The input database and query files will have the same format. Each sequence will be prefixed by a line starting with a greater than character (“>”) followed by a short description of the origin of the sequence. The DNA sequence will then begin on the next line for some number of lines. Each line will contain exactly 70 characters from the set A, C, G and T, except the last line, which may hold fewer than 70 characters. Following this last line will be the descriptor line of the next DNA sequence. The end of the file will be signified by the descriptor (“>EOF”). For each query string contained within the second input file, the output file should print the descriptor of the query sequence and the descriptors of any database sequences that contain a match as well as the position within the database sequence of that exact match. If the query sequence string is not found within any database sequences, a message to that effect should be printed after the query descriptor.

Example DNA Database:

```
>DB description string 1
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACT
GGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAG
ACAGATAAAAATTACA
>DB description string 2
AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGACAGTGCGGGCTTTTTTTTTCGACCAAAG
GTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGC
CGATATTCTGGAAAGCAATGC
>EOF
```

Example query base:

```
>Query description string 1
CATTCTGACTGCAA
>Query description string 2
AAAAAAG
>Query description string 3
GTAA
>Query description string 4
AGAGAGAGAGAGAGAGAGAGAGAGAGAGAG
>EOF
```

Example output:

```
Query description string 1
[DB description string 1] at offset 7

Query description string 2
[DB description string 1] at offset 47
[DB description string 2] at offset 33

Query description string 3
[DB description string 1] at offset 94
[DB description string 2] at offset 79
```

```
Query description string 4
NOT FOUND
```

How to Deliver

The instructions are as given in the first question. In addition to the explanation of the string-matching algorithm selected for this purpose you should also explain whether any heuristics were applied to make the algorithm efficient. Test data should consist of *querybase.txt* file and the *output.txt* file. *querybase.txt* file should contain the query strings that were used for test runs and the corresponding output should be given in *output.txt* file.

It is possible for you to produce one *Readme* file for both questions.