University of Exeter

COURSEWORK SPECIFICATION
ECM2433 **The C Family**
Module leader: Richard Everson
Academic year: 2024/25

Submission deadline: **Tuesday 11th March 2025**

This continuous assessment (CA) comprises 30% of the overall module assessment.

This is an individual assessment and you are reminded of the University's regulations on collaboration and plagiarism. You must avoid plagiarism, collusion, and any academic misconduct behaviours. Further details about academic honesty and plagiarism can be found at https://ele.exeter.ac.uk/course/view.php?id=1957.

Use of GenAI tools.

The University of Exeter is committed to the ethical and responsible use of Generative AI (GenAI) tools in teaching and learning, in line with our academic integrity policies where the direct copying of AI-generated content is included under plagiarism, misrepresentation and contract cheating under definitions and offences in TQA Manual Chapter 12.3. To support students in their use of GenAI tools as part of their assessments, we have developed a category tool that enables staff to identify where use of Gen AI is integrated, supported or prohibited in each assessment. This assessment falls under the category of **AI-prohibited**. This is because the use of GenAI tools in developing this assessment prevents achievement of the module learning outcomes.

You can find further guidance on using GenAI critically, and how to use GenAI to enhance your learning, on Study Zone digital.

When submitting your assessment, you must include the following declaration, ticking all that apply:

- ☑ I have not used any GenAI tools in preparing this assessment.
- ☑ I certify that all material in this dissertation which is not my own has been identified.

Please note that submitting your work without an accompanying declaration, or one with no ticked boxes, will be considered a declaration that you have not used GenAI tools in preparing your work.

This CA tests your knowledge of the programming in C and/or C++, particularly the writing and testing of more complex programs.

Make sure that you lay your code out so that it is readable and you comment the code appropriately. All your functions should at least include a comment describing the arguments, what they return and what they do. You may like to adhere to one of the Doxygen conventions for this.

Your programs should compile and run using the `gcc` compiler on either the Azure Lab VMs or the `emps-ugcs[12].ex.ac.uk` machines.

1 Word lengths

Write a function `void histogram(int *x, double *y, int n, int width)` which draws a simple histogram of the percentages given in the array `y` versus the numbers in the array `x`. Both arrays have length `n`. So, for example,

```
int *x = {0, 1, 2, 3, 4, 5};
double *H = {12.5, 6.4, 10, 7.6, 8, 13};
histogram(x, H, 6, 30);
```

should produce the output

```
0 ****************************      12.5
1 ***************      6.4
2 ************************      10
3 ******************      7.6
4 ******************      8
5 ****************************      13
```

Here the `width` argument controls the width (in characters) of the longest bar of stars; in the above example the bar corresponding to `H[5] = 13` comprises 30 stars.

Your histogram function should be in a separate file named `histogram.c`. Write an additional test program `demo_histogram.c` that prints an example like the one above.

Write another function `int *histogram_lengths(char **strings, int n)` that, given an array of `n` strings `strings`, returns an array of integers, `H`, where `H[i]` is the number of times that strings of length `i` occurred in `strings`.

The file `dracula.txt`, available from the ELE page, contains the text of the first 10 chapters of Bram Stoker's *Dracula* with one word per line and all punctuation removed. Use your functions to write a program `wordlengths.c` that draws a histogram of the percentages of times that words of different lengths occur in `dracula.txt`. Your program should take the name of the file containing the words as a command line argument, thus:

```
$ wordlengths dracula.txt
```

(`$` is the shell prompt.)

Your code should be organised so that the functions `histogram` and `histogram_lengths` are in the file `histogram.c`.

[20 marks]

2 Anagrams

Two words are anagrams of each other if they contain precisely the same letters, for example *orchestra* and *carthorse* are anagrams of each other, as are *damned*, *demand* and *madden*. The aim of this exercise is to write a program that will find all the anagrams in an array of words read from `words.txt` on ELE.

Anagrams can be identified by sorting the letters in the words into alphabetical order. Words that are anagrams of each other will have the same sorted "key"; for example "orchestra" and "carthorse" both sort to "acehorrst". Anagrams can be identified by constructing a ("primary") linked list in which each node stores a sorted "key" and all the words that correspond to that key in another secondary list. To facilitate rapid searching, the primary linked list should be constructed in sorted order of the keys.

Write a function `make_anagram_list(char **words, int n)` that returns a linked list as described above given an array of `n` strings in `words`. Use your function to write a program `anagram.c` that uses `words.txt` to find and print the anagram with the largest number of variants (e.g., *damned*, *demand* and *madden* have three variants) and also the longest pair of words that are anagrams of each other.

Let $V(A)$ be the number of variants corresponding to anagram A, so $V(orchestra) = V(carthorse) = 2$ and $V(damned) = V(demand) = V(madden) = 3$. It would be interesting to know how many anagrams there are for a given value of V. Use your `histogram` function to plot a histogram of $\log_{10}$ of the number of anagrams there are corresponding to each V for $V = 2, \ldots, 12$. Logarithms are used here because there are vastly many more anagrams with 2 or 3 variants than those with more variants.

Finally, write a program `anaquery.c` that repeatedly asks the user for a word and prints its anagrams (if any). Show your program working for *orchestra* and *damned*.

[30 marks]

3 Patience

A variation of the card game *patience* is played as follows.[1] Starting with a shuffled deck of cards, the top two cards are placed face up so that their numbers are visible. Then, if their values sum to 11 (i.e., Ace and 10; 2 and 9; 3 and 8; 4 and 7; 5 and 6) then they are covered by two new cards drawn from the deck. If their values do not sum to 11 then a new card is drawn from the deck and placed face up to form a third visible card. If any pair of these sum to 11, then they are covered by new cards drawn from the deck; if not a new card is drawn from the deck to form a fourth visible card. Play continues this way until either the player wins by having no more cards in the deck or their are more than nine piles of visible cards. In addition to the "two cards summing to 11" rule, if a Jack and a Queen and a King are **all** visible then, each of them is covered by a new card from the deck.

Here is an example in which the player wins. The cards are represented by the numbers 1, ..., 13, with 1, 11, 12 and 13 meaning Ace, Jack, Queen and King respectively. The deck, written as an array of `int`s, is:

```
int deck[] = 10, 4, 9, 8, 5, 1, 2, 12, 9, 11, 2, 12, 1, 3, 12, 10, 6, 13, 7, 6, 10,
4, 7, 5, 8, 2, 4, 1, 3, 9, 5, 4, 6, 9, 11, 10, 13, 11, 11, 1, 12, 3, 13,
2, 5, 13, 7, 3, 7, 6, 8, 8;
```

The game, which starts with the first two elements in the deck (10 and 4), is:

[1] We use the standard British deck of 52 cards in four suits: Ace (=1), 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King.

10	4								*Cards don't add to 11; start a new pile*
10	4	9							*Cards don't add to 11; start a new pile*
10	4	9	8						*Cards don't add to 11; start a new pile*
10	4	9	8	5					*Cards don't add to 11; start a new pile*
10	4	9	8	5	1				*10 and 1 add to 11; cover with new cards*
2	4	9	8	5	12				*2 and 9 add to 11; cover with new cards*
9	4	11	8	5	12				*Cards don't add to 11; start a new pile*
9	4	11	8	5	12	2			*9 and 2 add to 11; cover with new cards*
12	4	11	8	5	12	1			*Cards don't add to 11; start a new pile*
12	4	11	8	5	12	1	3		*8 and 3 add to 11*
12	4	11	12	5	12	1	10		*1 and 10 add to 11*
12	4	11	12	5	12	6	13		*J, Q, K (11, 12, 13) visible; cover with new cards*
6	4	7	12	5	12	6	10		*6 and 5 add to 11*
4	4	7	12	7	12	6	10		*4 and 7 add to 11*
5	4	8	12	7	12	6	10		*5 and 6 add to 11*
2	4	8	12	7	12	4	10		*7 and 4 add to 11*
2	1	8	12	3	12	4	10		*1 and 10 add to 11*
2	9	8	12	3	12	4	5		*2 and 9 add to 11*
4	6	8	12	3	12	4	5		*6 and 5 and to 11*
4	9	8	12	3	12	4	11		*8 and 3 add to 11*
4	9	10	12	13	12	4	11		*Q (12), K (13), J (11) visible; new cards: 11, 1 and 11*
4	9	10	11	1	12	4	11		*10 and 1 add to 11*
4	9	12	11	3	12	4	11		*Start a, 9th and last, new pile*
4	9	12	11	3	12	4	11	13	*Q, J, K visible; new cards: 5, 2, 13*
4	9	5	2	3	12	4	11	13	*9 and 2 add to 11*
4	7	5	3	3	12	4	11	13	*Q, K, K visible; new cards: 7, 6, 8*
4	7	5	3	3	6	4	7	8	*4 and 7 add to 11, but 8 is the only card left in deck*

Note that in some cases more than one pair adds to 11 or a pair adds to 11 and the Jack, Queen and King are all visible. In these cases it depends on the player (or program!) to determine the order in which the rules are applied. Most games end with the player losing because they need to add a tenth pile of cards. A short, unannotated example is:

3	2								
3	2	1							
3	2	1	5						
3	2	1	5	13					
3	2	1	5	13	5				
3	2	1	5	13	5	3			
3	2	1	5	13	5	3	2		
3	2	1	5	13	5	3	2	6	
3	2	1	7	13	5	3	2	12	
3	2	1	7	13	5	3	2	12	6

At the end there are 9 piles of cards, but no pair adds to 11 and, although the Queen and King are visible, the Jack is not.

The majority of games end with some cards left in the deck. The object of this question is to estimate the probability of winning a game. To do that you are asked to write a program that simulates many games and thus to calculate the fraction of games that are won.

Represent the cards as the integers 1 to 13. You should choose suitable data structures to represent the deck of cards and the visible cards.

You should write two functions `add_to_11(visible)` and `jqk(visible)` that take as their arguments the data structures representing the visible cards and return data indicating which cards either add to 11 or are the Jack, Queen and King. Depending on how you choose the represent the visible cards you may need to supply additional arguments to these functions.

Write a function `int play(deck, verbose)` that plays a single game of patience with a deck `deck`. The `int` argument `verbose` should control whether the visible cards are printed at each stage of the game (as above, but just the cards without the annotations); if `verbose` is false (0), the progress of the game is not printed. Your `play` function should return the number of cards left in the deck at the end of a game; thus 0 means the player has won and a positive number means that the player has lost. Test your function by writing a program `patience.c` that plays and prints a single game of patience with either a randomly chosen seed for shuffling the deck or the seed supplied as a command line argument. Save the output of a game in which the player wins (in `win.txt`) and another in which they lose (in `lose.txt`).

Write a function `many_plays(int N)` which plays `N` games of patience. The function should return an array of length 53 showing the number of times a game ended with 0, ..., 52 cards left in the deck. Thus if the returned array is called `remaining`, and if 20 games end with 6 cards remaining, then `remaining[6] = 20`.

Use your `histogram` function (from the first question) to plot a histogram of the percentage of times that games ended with 0, ..., 52 cards left in the deck. That is, plot a histogram of the array returned by your `many_plays` function; this should be saved in a file `phistogram.txt`. Your program, `pstatistics.c`, should be fast enough that you can choose `N = 10000` and therefore obtain a reasonable estimate for the probability of any number of cards remaining in the deck. In particular, state what the probability of winning a game is.

Hint:

- You can use the `shuffle` function available from the ELE page to shuffle the array of integers representing the deck of cards. This function wraps the GNU Scientific Library to shuffle an array of integers in place. To use this you will need to link your code with the GNU Scientific Library; see the comments in the code and the accompanying `Makefile`. The zip file contains the `shuffle.c` function and code demonstrating how to use it.

You should submit:

- A copy of your programs `patience.c` and `pstatistics.c` together with any additional source files needed to compile them. You should also provide a `Makfile` that can be used to compile the `patience` and `pstatistics` programes.
- Output of your programs in `win.txt`, `lose.txt` and `phistogram.txt`.

[50 marks]

[Total 100 marks]

Marking criteria

Work will be marked against the following criteria. Although it varies a bit from question to question they all have approximately equal weight.

- **Does your algorithm correctly solve the problem?**
  In most of these exercises the algorithm has been described in the question, but not always in complete detail and some decisions are left to you.
- **Does the code correctly implement the algorithm?**
  Have you written correct code? Do your functions meet the specification? Is the code you have written syntactically correct?

- **Is the code beautiful?**
  Is the implementation clear and efficient or is it unclear and inefficient? Have you used appropriate data structures? Is the code well structured? Have you made good use of functions?
- **Is the code well laid out and commented?**
  Is there a comment describing what the code does? Are there comments describing the major portions of the code or particularly tricky bits? Do functions have a comment describing what they do and how they are used? Although C doesn't care about indentation, have you used indentation and space to make the code clear to human readers?
- **Does the code correctly allocate and free memory?**
  Make sure your code doesn't leak memory.

There is a **10% penalty for not naming files** as instructed in the questions.