Summary: This project explores pattern matching techniques to find a pattern in a DNA sequence containing letters in the DNA alphabet {A, C, G, T}. For example, suppose we have a DNA sequence as follows:

ATGACGATCTACGTATGGCAGCCACGCTTTTGATGTTAAGTCACACAGCCAAGTCA
ACAAGGGCGACTTCATGATCTTTCCGCTCCGTTGGTGTAGGCCCGTGTTCAAATTC
AATGGCTGATTGGAATTACCTTTGAAATACTCCAACCGACCGCCACGGCCAGGGT
CCCGCTCGCTCTCTGTGGCCCTCCCACAAAACTCCGGTGAAAGTTGATTTGGACAC
GGACCCAAAGCAGCGTAGATTATTCGAGCGTATTCGGTAGTCATTGAGGCCCCAA

The pattern "AATGG" can be found at the beginning of the third line. Note that overlapping matches are counted individually. For example, if the sequence is 'AAAAAA' and the pattern is 'AAA', there are 4 occurrences of the pattern.

Part 1: A shell program `DNA-shell.c` is provided that randomly initializes a text string (10240 characters) with the DNA alphabet. A pattern of 3 to 7 characters is also randomly generated using the DNA alphabet. You must insert code into the shell to implement the function 'match' which takes four input parameters, the pointer to the text string, the length of the text string, a pointer to the pattern, and the length of the pattern. The 'match' function must return the array of indices of occurrences of the pattern in the text string. For example, if the sequence is 'AACAAC' and the pattern is 'AAC', the return value is '03' where '0' indicates the index of the first occurrence of 'AAC' and '3' indicates the index of the second occurrence.

The shell program includes a print statement for reporting the frequency of occurrences computed by 'match', so it can be properly graded. Its output format should not be modified.

You should design, implement, and test your own code. Otherwise you won't learn the things you need to know for later parts of the projects. **Any submitted project containing code not fully created and debugged by the student constitutes academic misconduct.**

You should use `gcc` under Ubuntu to develop your program (type `man gcc` for compiler usage). Normally, you should compile and run your program using the Linux command line:

```
> gcc DNA-shell.c -g -Wall -o DNAsearch
> ./DNAsearch
```

In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named **<your last name>-p1-1.c**.
2. You name and the date should be included in the header comment.
3. The starting *shell* program should not be modified except for the replacement of the comment `/* your program goes here */` and the addition of declared local variables. It is especially important not to remove or modify the print statement since that will be used in the automatic grading process.
4. Your solution must be properly uploaded to the submission site before the scheduled due date **Wednesday, April 2 at 11:55 p.m.** It will be accepted late without penalty through **Friday, April 4 at 10 p.m.**

Part 2: In this project, you will write an assembly program that performs the DNA search task. You will call a software interrupt that will generate a random test DNA sequence and a random pattern to search for. Your MIPS assembly code must compute the indices of each occurrence of the pattern in the DNA sequence. As with P1-1, this is returned in a null terminated array of

integers. The null character is -1. A shell program `DNA-shell.asm` is provided to get you started. As in part 1, there may be code available from other sources (e.g., on the web, from friends, etc.); don't use it! You should figure this out on your own.

Library Routines: There are three software interrupt routines (accessible via the `swi` instruction).

SWI 548: Create DNA Sequence and Pattern: This routine initializes memory beginning at the specified base address, given in register $1, with the randomly generated DNA sequence. In particular, the DNA sequence has a fixed length of 4800 characters from the DNA alphabet. The 4800 characters are encoded into a 600-word array starting at the memory address given in $1. Each character is represented by 2 bits, packed in little endian order in the lower 16 bits of each of the 600 words. The upper 16 bits of all the 600 words are not used.

Therefore, bits 0-1 of the first word denotes the first character of the text; bits 2-3 of the same word represents the second character; bits 0-1 of the second word represents the 9th character of the text. The alphabet is encoded as follows:

A: 00
T: 01
G: 10
C: 11

For example, if we have binary number 0000 0000 0000 0000 0000 0000 1110 0100, it represent ATGCAAAA .

Your MIPS code must decode the input word array and unpack the 4800 characters. The pattern is returned in $2 by swi 548. The pattern itself is packed in the lower half word of $2, the length of the pattern (between 3 and 7) is encoded in the upper half word of $2.

In addition, swi 548 will display the DNA sequence as a 80x60 matrix.

INPUTS: $1 should contain the base address of the 600 word (2400 byte) 80x60 array already allocated in memory. OUTPUTS: Register $2 contains a randomly generated pattern in its lower half word and the length of the pattern (between 3 and 7) in the upper half word.

SWI 549: Highlight Position: This routine allows you to specify a position in the 80x60 DNA sequence array and it marks the position in the DNA sequence visualization. INPUTS: $1 should contain the position (i.e., the offset into the DNA sequence) to be highlighted. OUTPUTS: none.

SWI 580: Matches: This routine (580) allows you to report the indices of occurrences of the pattern your program found in the DNA sequence. INPUTS: Register $2 must contain the base address of the array storing the indices of the occurrences of the pattern found in the DNA sequence. OUTPUTS: none.

Evaluation: In this version, correct operation and efficient performance are evaluated. The code size, dynamic execution length, and operand storage requirements are scored empirically, relative to a baseline solution. The baseline numbers for this project are static code size: 53 instructions, dynamic instruction length: 40,970 instructions (avg.), storage required: 650 words Your score will be determined through the following equation:

Percent Credit = 2 – (your program metric / baseline program metric)

Percent Credit is then used to determine the number of points for the corresponding points category. Important note: while the total score for each part can exceed 100%, especially bad performance can earn *negative credit*. The sum of the combined performance metrics scores (code size, execution length, and storage) will not be less than zero points. **Finally, the performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn performance credit if your implementation fails ten or more of the 100 trials.**

In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named **`<your last name>-p1-2.asm`**.
2. Your name and the date should be included in the beginning of the file.
3. Your program must produce and store your final answer in $2 when it returns. This answer is used by an automatic grader to check the correctness of your code.
4. Your program must return to the operating system via the **`jr`** instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
5. Your solution must be properly uploaded to the submission site before the scheduled due date **Friday, April 11 at 11:54 p.m.** It will be accepted late without penalty until **Sunday, April 13 at 10 p.m.**

Project Grading: The project grade will be determined as follows:

part	***description***	***percent***
P1-1	Functional DNA search	
	Algorithm	20
	Style	10
	Compiles	10
	Runs without crashing	20
	Correctness	40
	total	100
P1-2	Performance DNA search	
	correct operation, proper technique and style	45
	static code size	15
	dynamic execution length	35
	operand storage requirements	5
	total	100