CSE 220: Systems Fundamentals I

Homework #5

Spring 2017

Assignment Due: Friday, April 21, 2017 by 11:59 pm

Assignment Overview

The focus of this homework assignment is writing recursive functions and its application to string functions commonly used for studies like DNA sequencing. This assignment also reinforces MIPS function calling and register conventions.

Please read the assignment completely before implementing any of the functions.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in the `hw5.asm` file.

⚠ **You MUST follow the efficient MIPS calling and register conventions. Do not brute-force save registers! You WILL lose points.**

⚠ Do **NOT** rely on changes you make in the main files! We will test your functions with our own testing mains. Functions will not be called in the same order and will be called independent of each other.

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

ℹ If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

ℹ When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

Getting started

Download `hw5.zip` from Piazza in the Homework section of Resources. This file contains `hw5.asm` and 4 different main files (one for testing each of your functions), which you need for the assignment.

At the top of your `hw5.asm` program in comments put your name and SBU ID number.

```
# Homework #5
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to Test Your Functions

To test your functions, simply open the provided `hw5.asm` and one of the `main` files in MARS. Assemble the `main` file and run the file. MARS will take the contents of the file referenced with the `.include hw5.asm` at the bottom of the file and add the contents of your file to the main file before assembling it. Once the contents have been substituted into the file, MARS will then assemble it as normal.

⚠ Your assignment will not be graded using the provided main. Any modifications to `main` files will not be graded. You will only submit your `hw5.asm` file via Sparky. Make sure that all code require for implementing your functions (`.text` and `.data`) are included in the `hw5.asm` file!

⚠ Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

Part 1 - Matching a globbed DNA pattern

DNA sequences are usually very long. The human genome has over 3 billion base pairs! Searching for exact patterns in such a long sequence can be very tedious and hence sometimes searching for approximate patterns is useful to narrow down the search space. In this part you will be writing a function check if a sequence matches with a globbed sequence.

Globbing

A glob pattern is a pattern that includes wildcard characters. A wildcard character is a single character, such as an asterisk ('*'), used to represent a number of characters or an empty string. It is often used in file searches so the full name need not be typed. You might have seen `*.txt` used to specify all text files in UNIX systems. If you have not yet, you will in CSE320 ;) For this homework you will only support the * wildcard. It matches any number of any characters, including none.

`match_glob` - Recursive function

This function compares a DNA sequence to a globbed pattern and tells you if there is an exact match, along with the length of all the globbed section(s) that was matched.

```
(boolean match, int glob_len) match_glob(char[] seq , char[] pat)
```

- `seq` : A string containing only letters representing DNA nucleotides (A,C,G or T). You can assume that `seq` will always be valid. Note, the characters are CASE INSENSITIVE!

- `pat`: A string specifying the globbed pattern. You can assume the string contains only letters representing DNA nucleotides (A,C,G or T). It may contain 0 or more * characters. You can assume that `pat` will always be valid, and there will be 0 or 1 exact matches of `pat` in `seq`.
- *returns*: `match` as true (1) if `pat` is match with `seq`, false (0) if not. `glob_len` is the total length of the string(s) that was matched by the wildcard(s). `glob_len` is irrelevant if `match` is false. All matches are case insensitive.

Examples:

Code	Return Value
`match_glob("","*")`	true (1), 0
`match_glob("ACGTTCAAGAGTACC","")`	false (0), X (don't care)
`match_glob("acgttcaagagtacc","ACG*")`	1, 12
`match_glob("ACGTTCAAGAGTACC","AcG*AcC")`	1, 9
`match_glob("acgttcaagagtacc","*tACc")`	1, 11
`match_glob("ACGTTCAAGAGTACC","ACG")`	0, X (don't care)
`match_glob("acgttcaagagtacc","*GTA*")`	1, 12
`match_glob("ACGTTCAAGAGTACC","*")`	1, 15

⚠ Don't care values will not be checked when grading. Any value may be returned.

⚠ Note: We will not test with two or more wildcard (*) characters sequentially.

The logic for this recursive function can be confusing or difficult. We have provided the pseudo code for the algorithm below.

```
(boolean, int) match_glob(char[]  seq, char[] pat) {
    /* CHECK BASE CASES */
    // Wildcard is the only character left
    if(pat.equals("*"))
        return (true, seq.length());

    // Seq and pat are identical
    if(seq.equalsIgnoreCase(pat))
        return (true, 0);

    // If seq or pat is empty and the other is not, then no match possible.
    if((seq.length() XOR pat.length()) != 0)
        return (false, 0);
    /* RECURSIVE CALLS for remaining sequence */
    // Check if the characters are equal
    if(seq[0].toLower() == pat[0].toLower())
        return match_glob(seq.substring(1), pat.substring(1));

    //the current char is a glob match
    if(pat[0] == '*') {
        // recursively call on the remaining pattern
        (match1, glob_len1) = match_glob(seq, pat.substring(1));

        if (match1) {
           // match is found!
           return (match1, glob_len1);
        } else {
           // no match found. Continue searching from the next character
           (match2, glob_len2) = match_glob(seq.substring(1), pat);
           return (match2, glob_len2 + 1);
        }
    }
    return (false, 0);
}
```

Part 2 - DNA Sequence Permutation Generation

A nucleotide is the basic structural unit of DNA. There are 4 different kinds of nucleotides in DNA: Adenine (A), Guanine (G), Cytosine (C) and Thymine (T). A strand of DNA has a double-helix structure, which looks like a spiraling ladder. To form such a structure, the nucleotides are paired using complementary base pairing. Without delving into the details, this means that A can only connect with T (and vice versa) and C can only connect to G (and vice versa).

In this part, you will be writing functions to generate and display all permutations of a DNA sequence of desired length.

`save_perm` - Helper function

This function saves `seq` in character pairs separated by a hyphen (-) to the address specified by `dst`. A newline character is saved at the end of the permutation.

`char[] save_perm(char[] dst, char[] seq)`

- `seq`: A string containing only letters representing DNA nucleotides (A,C,G or T). You can assume that `seq` will always be valid, be of even length, and contain only CAPITAL letters.

- `dst`: The starting destination address of where to save the permutation characters with hyphens.
- *returns*: The address of the next byte after the newline char.

Examples:

Code	
`save_perm("@@@@@@@@@@@@1234", "ATGCCGTA")`	
`dst` After Call	**Return Value**
`"AT-GC-CG-TA\n1234"`	Address of byte after \n

Code	
`save_perm("@@@@@@@@@@@@1234", "AT")`	
`dst` After Call	**Return Value**
`"AT\n@@@@@@@@@1234"`	Address of byte after \n

`construct_candidates` - Helper function

This function constructs the possible candidates for the next character in the permutation. If the next character is the start of a new pair it can be any nucleotide, however, if it is a continuation of the same pair then the next character has to contain the complementary base pair.

```
int construct_candidates(char[] cand, char[] seq, int n)
```

- `cand`: Space to store the candidates for the next character of the permutation. This space is declared in the caller function.
- `seq`: A character array representing the current state of the permutation. This will only contain valid DNA nucleotides but will not be the completed permutation. You can assume that `seq` will always be valid but **not necessarily null-terminated**.
- `n`: A number describing the location of the character-to-be-filled in the `seq` array.
- *returns*: Returns the number of candidates that are present (Maximum is 4).

Examples:

Code	
`construct_candidates("@@@@!!!!", "AT", 2)`	
cand After Call	**Return Value**
`"ACGT!!!!"`	4

Code	
`construct_candidates("@@@@!!!!", "ATC", 3)`	
cand After Call	**Return Value**
`"G@@@!!!!"`	1

Code	
`construct_candidates("@@@@!!!!", "ATCGG", 5)`	
cand After Call	**Return Value**
`"C@@@!!!!"`	1

Code	
`construct_candidates("@@@@!!!!", "ATCGGCT", 7)`	
cand After Call	**Return Value**
`"A@@@!!!!"`	1

Code	
`construct_candidates("@@@@!!!!", "ATCGGCTAA", 9)`	
cand After Call	**Return Value**
`"T@@@!!!!"`	1

Code	
`construct_candidates("@@@@!!!!", "ATCGGCTAAB@!PQYRSHR.,@TAAG", 9)`	
cand After Call	**Return Value**
`"T@@@!!!!"`	1

We have provided the pseudo code for the algorithm below.

```
int construct_candidates(char[] candidates, char[] seq, int n) {
    //If the next candidate is part of a pair, pick complement
    if(n%2 != 0) {
        if(seq[n-1] == ‘A’) {
            candidates[0] = ‘T’;
            return 1;
        } else if(seq[n-1] == ‘T’) {
            candidates[0] = ‘A’;
            return 1;
        } else if(seq[n-1] == ‘C’) {
            candidates[0] = ‘G’;
            return 1;
        } else {
            candidates[0] = ‘C’;
            return 1;
        }
    } else {
        candidates[0] = ‘A’;
        candidates[1] = ‘C’;
        candidates[2] = ‘G’;
        candidates[3] = ‘T’;
        return 4;
    }
}
```

`permutations` - Recursive function

This is the recursive function that will generate all the possible permutations of desired length and store them into memory using the `save_perm` function.

```
(int rv, char[] next) permutations(char[] seq, int n, char[] res, int length)
```

- `seq` : Buffer character array of size at least `length+1` , which is not necessarily initialized, to build up the permutation.
- `n` : Continue creating permutation starting at the `n` th character (`seq[n-1]`).
- `res` : Pointer of location to store the result when the permutation is complete (size == `length`). The value of this pointer will be updated as permutations are stored.
- `length` : The total number of characters for each permutation.
- *returns*: Error (-1,0) if `length` is invalid, otherwise return (0, pointer), where `pointer` is the address of the next location for storing the next permutation (i.e the address that is returned by `save`).

Return (-1, 0) in any of the following cases:

- `length` is less than or equal to 0
- `length` is an odd number

Examples:

Code	**Return Value**
`permutations(buf, 0, 0x400, 3)`	-1, 0
`permutations(buf, 0, 0x400, 0)`	-1, 0
`permutations(buf, 0, 0x400, 4)`	0, 0x460
`permutations(buf, 3, 0x400, 6)`	0, 0x424

⚠ `0x400` is an arbitrary address for `res` . This address will change when each full permutation is completed and is decided by the value returned by `save_perm` .

Here is what is stored for `permutations(buf, 0, 0x400, 4)`

AT-AT\nAT-CG\nAT-GC\nAT-TA\nCG-AT\nCG-CG\nCG-GC\nCG-TA\nGC-AT\nGC-CG\nGC-GC\nGC-TA\n TA-AT\nTA-CG\nTA-GC\nTA-TA\n

The last example is an example of a partial recursive call. In this recursive call the first 3 elements of buf are part of a solution and has been set (e.g: ATC@@@). A full example of such a call is below.

Here is what is stored for a partial permutation where `buf` already contains "ATC"
`permutations(buf, 3, 0x400, 6)`

AT-CG-AT\nAT-CG-CG\nAT-CG-GC\nAT-CG-TA\n

The logic for this recursive function can be confusing or difficult. We have provided the pseudo code for the algorithm below.

```
(int, char[]) permutations(char[] seq, char[] res, int n, int length) {

    // Validation: length cannot be 0 and it has to be an even number
    if(length%2 == 0 || length == 0) {
        return (-1, 0);
    }

    // A Permutation of the final length has been reached. Save it to res.
    if (n == length) {
        add_null(seq, length+1)
        char[] next = save_perm(res, seq);
        return (0, next);
    }
    else {
        // Create candidates for the next character and repeat
        // Declare space on the stack and you can use the frame pointer
        // for reference.
        char[] candidates = new char[4];

        int ncand = construct_candidates(n, seq, candidates);

        // Iterate through the candidates, creating permutations starting
        // with each candidate
        for(int i = 0; i < ncand; i++) {

            seq[n] = candidates[i];
            // Recursive call to select candidates for next character
            (ret, res) = permutations(seq, n + 1, res, length);
    }
    return 0, res;
}
```

Hand-in Instructions

See Sparky Submission Instructions on Piazza for hand-in instructions.

There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.