NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL EXAMINATION II FOR
Semester 1 AY2018/2019

CS1010 Programming Methodology

November 2018 Time Allowed 2 Hours 30 Minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 5 questions and comprises 8 printed pages, including this page.
2. The total marks for this assessment is 45. Answer **ALL** questions.
3. This is an **OPEN BOOK** assessment.
4. You can assume that all the given inputs are valid.
5. You can assume that the types `long` and `double` suffice for storing integer values and real values respectively, for the purpose of this examination.
6. Login to the special account given to you. You should see the following in your home directory:
   - The skeleton code `transpose.c`, `palindrome.c`, `rotate.c`, `marnell.c` and `bracket.c`
   - A file named `Makefile` to automate compilation and testing
   - A file named `test.sh` to invoke the program with its test cases
   - Two directories, `inputs` and `outputs`, within which you can find some sample inputs and outputs
7. You can run the command `make` to automatically compile and (if compiled successfully) run the tests.
8. Solve the given programming tasks by editing the given skeleton code. You can leave the files in your home directory and log off after the examination is over. There is no need to submit your code.
9. Only the code written in `transpose.c`, `palindrome.c`, `rotate.c`, `marnell.c` and `bracket.c` directly under your home directory will be graded. Make sure that you write your solution in the correct file.
10. Marking criteria differ by the question. Please see the end of each question for details.

1 Transpose (6 marks)

A matrix can be transposed by flipping it along its diagonal, a row of the matrix becomes a column in the transposed matrix, and vice versa. For instance, the matrix

$$\begin{pmatrix} 0 & 2 \\ 1 & 1 \\ 0 & 9 \end{pmatrix}$$

when transposed, becomes

$$\begin{pmatrix} 0 & 1 & 0 \\ 2 & 1 & 9 \end{pmatrix}$$

Write a program `transpose` that takes an $m \times n$ matrix M_1 and transpose it into an $n \times m$ matrix M_2. Your program must implement the following function to transpose a matrix `m1` into `m2`:

```
void transpose_matrix(double **m1, double **m2, long nrows, long ncols);
```

where the pointer `m1` points to the matrix M_1 and the pointer `m2` points to the matrix M_2. The parameters `nrows` and `ncols` correspond to m and n (the number of rows and number of columns of M_1) respectively.

Your program should reads the following from the standard input:

- the integers m and n ($m \geq 1, n \geq 1$), followed by
- m rows of real numbers, each row containing n real numbers. Each row corresponds to a row of elements of the matrix M_1.

Your program then prints the transposed matrix M_2 to the standard output, each row of the output corresponds to a row of numbers in M_2.

Sample Run

```
ooiwt@pe101:~$ ./cat inputs/transpose.1.in
2 4
-1.0001 -2.0001 -3.0001 -4.0001
-5.0001 -6.0001 -7.0001 -8.0001
ooiwt@pe101:~$ transpose < inputs/transpose.1.in
-1.0001 -5.0001
-2.0001 -6.0001
-3.0001 -7.0001
-4.0001 -8.0001
```

Grading Criteria

- **Documentation (1 mark)** Write the Doxygen documentation block for the function `transpose_matrix` as well as other functions that you write.
- **Efficiency (1 mark)** Your implementation to transpose the matrix must take no more than $O(mn)$ time.

- **Style (1 mark)** Your code should be neat, readable, and easy to understand. Follow the CS1010 guidelines to avoid getting penalized for the style.
- **Correctness (3 marks)** Your code should not only produce the correct output but also uses the programming constructs provided by C appropriately. Further, you need to ensure that you explicitly free any memory that you allocate in your program.

You will only receive the marks for style, efficiency, and documentation if your code is reasonably correct (as judged by the graders).

Comment:

This is supposed to be a giveaway question. We want to see if you can manipulate 2D arrays, including stepping through the elements, and allocating/deallocating the memory for them.

First, let's write some functions to allocate, read, print, transpose, and free a matrix.

```c
/**
 * Allocated and read a matrix from the standard input.
 * The caller is responsible to free the matrix.
 *
 * @param[in] row The number of rows to read.
 * @param[in] col The number of columns to read.
 * @return    A 2D array containing the matrix.
 */
double** read_matrix(long row, long col)
{
  double **m = malloc(row * sizeof(double *));
  if (m == NULL) {
    return NULL;
  }
  for (long i = 0; i < row; i += 1) {
    m[i] = cs1010_read_double_array(col);
    if (m[i] == NULL) {
      free_matrix(m, i);
      return NULL;
    }
  }
  return m;
}

/**
 * Allocated a 2D matrix of 0s.
 * The caller is responsible to free the matrix.
 *
 * @param[in] row The number of rows to allocate.
 * @param[in] col The number of columns to allocate.
 * @return    A 2D array containing the matrix.
 */
double** alloc_matrix(long row, long col)
{
  double **m = malloc(row * sizeof(double *));
  if (m == NULL) {
    return NULL;
  }
```

```
  for (long i = 0; i < row; i += 1) {
    m[i] = malloc(col * sizeof(double));
    if (m[i] == NULL) {
      free_matrix(m, i);
      return NULL;
    }
  }
  return m;
}

/**
 * Print a given 2D matrix.
 *
 * @param[in] m   The matrix to print.
 * @param[in] row The number of rows to print.
 * @param[in] col The number of columns to print.
 */
void print_matrix(double **m, long row, long col)
{
  for (long i = 0; i < row; i += 1) {
    for (long j = 0; j < col; j += 1) {
      cs1010_print_double(m[i][j]);
      if (j == col - 1) {
        cs1010_println_string("");
      } else {
        cs1010_print_string(" ");
      }
    }
  }
}

/**
 * Free the memory allocated for a 2D matrix.
 *
 * @param[in] m   The matrix to free.
 * @param[in] row The number of rows in the matrix m.
 */
void free_matrix(double **m, long row)
{
  for (long i = 0; i < row; i += 1) {
    free(m[i]);
  }
  free(m);
}

/**
 * Transpose a given matrix m1 and store the output in m2.
 *
 * @param[in]  m1  The given matrix to transpose.
 * @param[out] m2  The transposed matrix.
 * @param[in]  num_of_rows The number of rows in m1.
 * @param[in]  num_of_cols The number of rows in m2.
 */
void transpose(double **m1, double **m2, long num_of_rows, long num_of_cols) {
  for (long i = 0; i < num_of_rows; i += 1) {
```

```
    for (long j = 0; j < num_of_cols; j += 1) {
      m2[j][i] = m1[i][j];
    }
  }
}
```

The main program then looks like this:

```
int main() {
  long num_rows = cs1010_read_long();
  long num_cols = cs1010_read_long();

  double **m1 = read_matrix(num_rows, num_cols);
  if (m1 == NULL) {
    return 1;
  }
  double **m2 = alloc_matrix(num_cols, num_rows);
  if (m2 == NULL) {
    free_matrix(m1, num_rows);
    return 1;
  }

  transpose(m1, m2, num_rows, num_cols);

  print_matrix(m2, num_cols, num_rows);
  free_matrix(m1, num_rows);
  free_matrix(m2, num_cols);
}
```

2 Palindrome (6 marks)

A palindrome is a string that reads the same backwards or forwards if we ignore all characters that are not alphabets (including spaces) and treat upper/lower cases as the same. For instance, the string "Was it a car or a cat I saw?", when converted into lowercase letters without spaces and symbols, becomes the string "wasitacaroracatisaw", which reads the same backward or forward.

Write a program `palindrome` that reads, from the standard input, a string s containing at least one alphabet and prints, to the standard output, the string `yes` if the input string s is a palindrome, or the string `no` if s is not a palindrome.

Here are some tips that may be helpful:

- To check if a char `c` is a lowercase alphabet, you can check if `'a' <= c <= 'z'`,
- To check if a char `c` is a uppercase alphabet, you can check if `'A' <= c <= 'Z'`,
- To convert a char `c` from uppercase to lowercase, you can use the expression `c - 'A' + 'a'`.

You can also choose to use the functions `islower`, `isupper`, or `tolower`, available from `ctype.h` to do the above if you wish, but it is not necessary.

You may solve this problem either iteratively (with loops) or recursively.

Sample Runs

```
ooiwt@pe101:~$ ./palindrome
Was it a car or a cat I saw?
yes
ooiwt@pe101:~$ ./palindrome
it's a car
no
ooiwt@pe101:~$ ./palindrome
gg
yes
```

Grading Criteria

- **Documentation (0 marks)** You do not have to write the documentation blocks for each function. Note that you still need to comment your code, especially to explain your logic that might not be obvious to the readers.
- **Efficiency (1 mark)** Your implementation must take no more than $O(n)$ time, for input string of length n. A solution that is slower than $O(n)$ will receive 0 efficiency marks.
- **Style (1 mark)** Your code should be neat, readable, and easy to understand. Follow the CS1010 guidelines to avoid getting penalized for the style.
- **Correctness (4 marks)** Your code should not only produce the correct output but also uses the programming constructs provided by C appropriately. Further, you need to ensure that you explicitly free any memory that you allocate in your program.

You will only receive the marks for style and efficiency if your code is reasonably correct (as judged by the graders).

Comment:

This is a classic programming question and is also fairly simple. We want to see if you know how to manipulate a string. Here is a loop version:

```
bool is_palindrome(char *s) {
  long start = -1;
  long end = strlen(s);

  do {
    do {
      start += 1;
    } while (!isalpha(s[start]));

    do {
      end -= 1;
    } while (!isalpha(s[end]));

    if (tolower(s[start]) != tolower(s[end])) {
      return false;
    }
  } while (start < end);

  return true;
}
```

The two inner loops skip the non-alphabetical characters at the beginning and at the end of the string. After we ensure that `s[start]` and `s[end]` are both alphabets, we compare the lower-case version. If they are different, then it is not a palindrome and we return false. Otherwise, we continue. If we have exhausted all the characters in the string and we have not returned false, then the string must be a palindrome.

Here is a recursive version:

```
bool is_palindrome(char *s, long start, long end) {
  if (start >= end) {
    return true;
  }

  if (!isalpha(s[start])) {
    return is_palindrome(s, start + 1, end);
  }

  if (!isalpha(s[end])) {
    return is_palindrome(s, start, end - 1);
  }

  if (tolower(s[start]) == tolower(s[end])) {
    return is_palindrome(s, start + 1, end - 1);
  }
  return false;
}
```

3 Rotate (9 marks)

Given an array, a rotation operation does the following: it moves every element to the right by one slot, except the last element, which is moved to the front and becomes the first (indexed 0) element. For example, the array `1 2 -4 0 8` becomes `8 1 2 -4 0` after one rotation.

Suppose we are given an array of n integers that is originally sorted but has been rotated 0 or more times (but we do not know how many times). Write a program that searches for an element in this array in $O(\log n)$ time.

Your program, 'rotate', reads, from the standard input, a positive integer n ($n \geq 1$), followed by n distinct integers that correspond to the elements of a sorted but possibly rotated array, followed by an integer q. Your program then prints, to the standard output, either the string 'not found' if q is not part of the array, or the index of q in the array if q is found. The first element in the array is indexed as 0.

Sample Run 1

```
ooiwt@pe101:~$ cat inputs/rotate.1.in
4
1 2 3 4
5
ooiwt@pe101:~$ ./rotate < inputs/rotate.1.in
not found
```

Sample Run 2

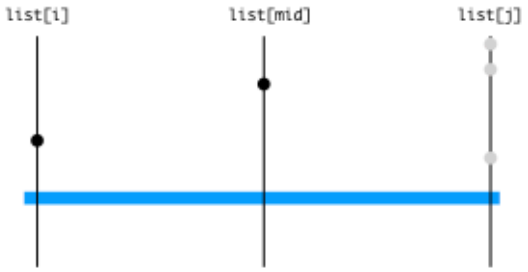
```
ooiwt@pe101:~$ cat inputs/rotate.2.in
8
1 2 3 4 5 -3 -2 0
4
ooiwt@pe101:~$ ./rotate < inputs/rotate.2.in
3
```

Grading Criteria

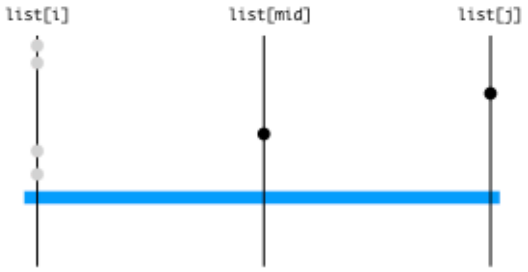
- **Documentation (0 marks)** You do not have to write the documentation blocks for each function. Note that you still need to comment your code, especially to explain your logic that might not be obvious to the readers.
- **Efficiency (5 mark)** Your implementation must take no more than $O(\log n)$ time for an array of size n. A linear solution is trivial and you will not receive any efficiency mark for a $O(n)$ solution.
- **Style (1 mark)** Your code should be neat, readable, and easy to understand. Follow the CS1010 guidelines to avoid getting penalized for the style.
- **Correctness (3 marks)** Your code should not only produce the correct output but also uses the programming constructs provided by C appropriately. Further, you need to ensure that you explicitly free any memory that you allocate in your program.

Comment:

Getting 4 marks for this question is trivial (using a linear search). But trying to get all 9 marks is not so straightforward. It requires a tweak to the binary search to solve in $O(\log n)$ time. The fact that it requires $O(\log n)$ time hinted at the need to halve the array at every recursive call. To decide which half to search, we first have to analyze the properties of the input. The properties can be summarized as follows:



list[i] .. list[mid] must be in increasing order.



list[mid] .. list[j] must be in increasing order.

But once you have the cases above, the code is quite straightforward. If `q` falls under the half that is a sorted array, then we perform a binary search on that half, otherwise, we recursively call the function to search in the rotated and sorted array on the other half.

```
/**
 * Look for q in list[i]..list[j].
 *
 * @pre list is sorted.
 * @return -1 if not found, the position of q in list otherwise.
 */
long search(const long list[], long i, long j, long q) {
  if (j < i) {
    return -1;
  }

  long mid = (i+j)/2;
  if (list[mid] == q) {
    return mid;
  }
```

```
  /*
   * - list[i]..list[mid] is strictly increasing.  So we do binary search
   *   if q falls within this range.  Otherwise, we recursively search
   *   the other half (which is a rotated list).
   */
  if (list[i] < list[mid]) {
    if (list[i] <= q && q <= list[mid]) {
      return binary_search(list, i, mid-1, q);
    }
    return search(list, mid+1, j, q);
  }

  /*
   * - list[i]..list[mid] is rotated.  So we do a binary search on the
   *   other half.
   */
  if (list[mid+1] <= q && q <= list[j]) {
    return binary_search(list, mid+1, j, q);
  }
  return search(list, i, mid-1, q);
}
```

Here, the function `binary_search` is the same one given in the lecture.

4 Marnell (12 marks)

In 2004, Geoffrey R. Marnell proposed the following conjecture: "Every number greater than 10 is the sum of a prime number and a semiprime".

A prime number is an integer larger than 1 that can only be divisible by 1 and itself. A semiprime is a number that is a product of two prime numbers (which may or may not be the same prime number). A square of a prime number is, therefore, a semiprime. The first 10 semiprimes are: 4, 6, 9, 10, 14, 15, 21, 22, 25, 26.

Write a program, `marnell`, that, reads, from the standard input, a positive number n, and prints, to the standard output, the number of pairs of prime number and semiprime that sums up to n. For example, 11 is the sum of the pairs:

- 2 and 9: 2 is prime; 9 = 3 × 3 is a semiprime
- 4 and 7: 7 is prime; 4 = 2 × 2 is a semiprime
- 5 and 6: 5 is prime; 6 = 2 × 3 is a semiprime

None of the pairs that sum up to 10 consists of a prime and a semiprime:

- 2 and 8: 2 is prime; 8 is not a semiprime.
- 3 and 7: 7 is prime; 3 is a prime.
- 4 and 6: Both 4 and 6 are semiprime.
- 5 and 5: 5 is a prime.

So your program `marnell` should print `0` when the input is `10`, and `3` when the input is `11`.

Sample Runs

```
ooiwt@pe101:~$ ./marnell
10
0
ooiwt@pe101:~$ ./marnell
11
3
```

Grading Criteria

- **Documentation (0 marks)** You do not have to write the documentation blocks for each function. Note that you still need to comment your code, especially to explain your logic that might not be obvious to the readers.
- **Efficiency (5 mark)** Your implementation must take no more than $O(n\sqrt{n})$ time, for input string of length n. Furthermore, you should avoid redundant work and repetitive work as much as possible.
- **Style (2 marks)** Your code should be neat, readable, and easy to understand. Follow the CS1010 guidelines to avoid getting penalized for the style.

- **Correctness (5 marks)** Your code should not only produce the correct output but also uses the programming constructs provided by C appropriately. Further, you need to ensure that you explicitly free any memory that you allocate in your program.

Comment:

This question is designed to check if you know how to write efficient code. It is an extension to `goldbach` from PE1. The prerequisite to solving this question well is that you must know how to find a prime number in $O(\sqrt{n})$ time. If you have been paying attention to Assignment 2, PE1 Comments, and the lecture on efficiency, you should have what it takes to solve this.

Everyone's first instinct would be to write a function `is_prime` and a function `is_semiprime` to check if a number is prime or semiprime. However, our second principle of writing efficient code *avoid repetitive work*, applies here. If you wrote these two functions, you will notice that we need to scan through numbers between 2 to $\sqrt{n}$ for an input n for both functions. So we could combine them into one function:

```
long is_prime_or_semiprime(long n) {
  if (n <= 1) {
    return NEITHER;
  }
  for (long i = 2; i*i <= n; i += 1) {
    if (n % i == 0) {
      if (is_prime_or_semiprime(i) == PRIME && is_prime_or_semiprime(n/i) == PRIME) {
        return SEMIPRIME;
      }
      return NEITHER;
    }
  }
  return PRIME;
}
```

With the above function, finding the number of pairs is not much different from `goldbach`:

```
  int counter = 0;

  for (long i = 2; i <= n/2; i += 1) {
    long p1 = is_prime_or_semiprime(i);
    if (p1 == PRIME || p1 == SEMIPRIME) {
      int p2 = is_prime_or_semiprime(n - i);
      if ((p1 == PRIME && p2 == SEMIPRIME) || (p1 == SEMIPRIME && p2 == PRIME)) {
        counter += 1;
      }
    }
  }
```

5 Bracket (12 marks)

Consider a string consisting of only four types of open brackets `(`, `[`, `<` and `{` and the corresponding matching close brackets `)`, `]`, `>` or `}`. We say that such a string is *valid* according to the following rules:

1. An empty string (a string with no character) is always valid;
2. A non-empty string must be either (i) a valid string followed by another valid string, or (ii) starts with an open bracket ends with a matching close bracket, and contain a valid string in between.

Let's look at some examples:

- The string `<>([])` is valid. It consists of two valid strings `<>` and `([])`. `<>` is valid since it starts with an open bracket `<`, ends with a matching close bracket `>`, and contains a valid (empty) string in between. `([])` is also valid since it starts with an open bracket `(`, ends with a matching close bracket `)`, and contains a valid string `[]` in between.
- The string `<(>)` is invalid since we cannot partition it into two valid strings, nor does it contains matching open and close brackets. `<` and `)` do not match.

Write a program `bracket` that reads, from the standard input, a string with at least two characters, consisting of only open and closed brackets, and prints, to the standard output, `yes` if the string is valid, `no` if the string is not valid.

NOTE: You must solve this problem using recursion.

Sample Runs

```
ooiwt@pe101:~$ ./bracket
<>([])
yes
ooiwt@pe101:~$ ./bracket
((())<>[]){}
yes
\begin{Verbatim}
ooiwt@pe101:~$ ./bracket
<(>)
no
```

Grading Criteria

- **Documentation (0 marks)** You do not have to write the documentation blocks for each function. Note that you still need to comment your code, especially to explain your logic that might not be obvious to the readers.
- **Efficiency (5 mark)** Your implementation must take no more than $O(n)$ time, for input string of length n. Furthermore, you should avoid redundant work and repetitive work as much as possible.

- **Style (2 marks)** Your code should be neat, readable, and easy to understand. Follow the CS1010 guidelines to avoid getting penalized for the style.
- **Correctness (5 marks)** Your code should not only produce the correct output but also uses the programming constructs provided by C appropriately. Further, you need to ensure that you explicitly free any memory that you allocate in your program, and you have to solve this problem using recursion.

Comment:

This is meant to be the hardest question that separates A+ from A students. Note that, if we ignore condition 2(i) (a valid string can be a concatenation of two valid strings), a valid string is just a "palindrome" consisting of characters `([<>])`, except that we match the opening brackets to the closing brackets instead of the same characters. If you solve that by modifying the recursive version of the palindrome code above, you should get some partial correctness marks.

The tricky part is to handle condition 2(i). So our recursive function has to tell us if the input string contains a valid string, and where it ends. One solution is to write a recursive function that "consumes" a valid string (and only a valid string) recursively and returns the next character that is unconsumed. If the next character is `0`, then the string must be a valid string, since everything is consumed.

Let's first write a couple of helper functions:

```
bool is_open(char c) {
  return c == '(' || c == '{' || c == '[' || c == '<';
}
```

The above checks if a given character is an open bracket.

```
bool is_matching_close(char c, char d) {
  return (c == '(' && d == ')') ||
    (c == '{' && d == '}') ||
    (c == '<' && d == '>') ||
    (c == '[' && d == ']');
}
```

The above checks if a pair of input characters are matching brackets.

And now, the recursive function to consume a valid string. It returns the index to the first character that is not consumed.

```
int consume_valid(char *string, int begin) {
  if (string[begin] == '\0') {
    return begin;
  }
  if (!is_open(string[begin])) {
    return begin;
  }

  int end = consume_valid(string, begin+1);
  if (is_matching_close(string[begin], string[end])) {
    return consume_valid(string, end + 1);
  }
  return begin;
}
```

Let's first look at Lines 2-4. If the input string is empty, then by Rule 1 it is always valid.

Lines 5-7 check if the input string starts with a closing bracket. If so, it cannot be a valid string and the function consumes nothing. By Rule 2, it must start with an open bracket. So we return `begin` since this is the index to the first unconsumed character.

When we reach Line 9, we can assert that the string begins with an open bracket. So we consume the string from index `begin + 1` onwards recursively. By wishful thinking, we can assume that this function will consume a valid string and returns the index of the first unconsumed character.

We now check on Line 10, if this unconsumed character (`string[end]`) is a matching close bracket to `string[begin]`, following Rule 2(ii). If it does not match, then we return `begin` (Line 13) since `string[begin]` is the first unmatched character (so the function spews out all the consumed characters and un-consumes them).

If Line 10 checks out, then the string from `begin` to `end` is valid, and we continue to recursively consume from character `end + 1` onwards.

In `main`, we call the function as follows:

```
int end = is_valid(str, 0);
if (str[end] == '\0') {
  cs1010_println_string("yes");
} else {
  cs1010_println_string("no");
}
```

END OF PAPER