NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL EXAMINATION II FOR
Semester 1 AY2020/2021

CS1010 Programming Methodology

November 2020 Time Allowed 2 Hours 30 Minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 5 questions and comprises 12 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 `tictactoe.c`, `sun.c`, `replace.c`, `soil.c` and `substring.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 `tictactoe.c`, `sun.c`, `replace.c`, `soil.c` and `substring.c` directly under your home directory will be graded. Make sure that you write your solution in the correct file.
10. Marking criteria are (i) correctness and (ii) style. One mark is allocated for style for each question. The rest is allocated for correctness. For instance, a 4-mark question has 1 mark allocated for style and 3 marks allocated for correctness.
11. Note that correctness is defined in the broad sense of using the various programming constructs in C (type, function, variable, loops, conditionals, arithmetic expressions, logical expressions) properly – *not just producing the correct output.*

12. You should write code that is clean, neat, and readable to get the mark allocated to style.

Note: The original Practical Exam II is conducted as an online exam. The questions have been reformatted to fit into paper format. The questions are set by Dr. Daren Ler and Ooi Wei Tsang.

1 Tic Tac Toe (6 marks)

The game of Tic-Tac-Toe is typically played on a game board that consists of a 3-by-3 grid. Two players called X and O, take turns to mark each grid cell with X and O respectively. Player X wins if she put X in either the same column, the same row, or diagonally on the game board. Similarly, Player O wins if she put O in either the same column, the same row, or diagonally on the game board.

Write a program `tictactoe` that reads in a Tic-Tac-Toe board from the standard input. The board is represented by three lines of text, each with three characters consisting of either `X`, `O` or `?`. `?` indicates that the cell is empty and has not been marked by either player yet. You can assume that the board represents a valid Tic-Tac-Toe game.

Your program must determine if the game has ended and there is already the winner, given the game board. If there is a winner, print out the winner (X or O). Otherwise, print out the string ”no winner”. You can assume that there is at most one winner.

Your program must contain a boolean function called `has_won` that takes in the game board and a player, and returns whether the given player has won.

Marking Scheme

Style	1 marks
Correctness	3 marks
Efficiency	0 marks
Documentation	2 marks

Efficiency: No marks is allocated for efficiency. Your code is expected to run in $O(1)$ time. We may deduct marks if your implementation performs obvious redundant or duplicated work.

Documentation: You should document every function in this question according to the CS1010 documentation requirement.

Sample Runs

```
ooiwt@pe101:~/$ ./tictactoe
XO?
X?O
X??
X
ooiwt@pe101:~/$ ./tictactoe
OX?
XO?
X?O
O
ooiwt@pe101:~/$ ./tictactoe
OXO
XX?
O?O
no winner
```

Comments:

This question is meant to be a giveaway.

Given a 3×3 2D array, you only need to scan through each row, each column, and each diagonal, to see if a given player has marked all three cells.

```c
/**
 * Check if a given player has won.
 *
 * @param[in] ttt The tic-tac-toe game board
 * @param[in] player The player to check
 *
 * @return true if the player has won.  false
 * otherwise.
 */
bool has_won(char *ttt[3], char player) {
    // check row
    for (long row = 0; row < 3; row += 1) {
       if (ttt[row][0] == player &&
           ttt[row][1] == player &&
           ttt[row][2] == player) {
           return true;
       }
    }
    // check col
    for (long col = 0; col < 3; col += 1) {
       if (ttt[0][col] == player &&
           ttt[1][col] == player &&
           ttt[2][col] == player) {
           return true;
       }
    }
    // check diagonal
    if (ttt[0][0] == player &&
        ttt[1][1] == player &&
        ttt[2][2] == player) {
       return true;
    }
    if (ttt[0][2] == player &&
        ttt[1][1] == player &&
        ttt[2][0] == player) {
       return true;
    }
    return false;
}
```

Many students appeared to have difficulties answering this question after PE2 (one commented on spending an hour on this) and the teaching team is puzzled about why. After grading, however, it is clear that most students solved this problem the hard way, taking an overly complicated approach and ignoring the requirement of the question given to help them.

Using variable size array

One way we simplify this 2D array question (compared to 18/19) is to use a square 2D array and a fixed-size 2D array. Having a fixed size 2D array means that you can declare your array with a size of 3, and therefore no need to check for NULLs and free the array after use. This significantly reduces the amount of code that you need to write.

Compare

```
char *board[3];
```

to

```
char **board;
board = malloc(sizeof(char *) * 3);
if (board == NULL) {
  // print something
  return 1;
}
  :
  :
free(board);
```

Writing a `has_won` method

The question asked for a `has_won` method that takes in the player as a parameter and determines if the given player has won. Many students do not follow this, but instead, try to solve the problem in their way.

One approach is to pass the player by reference, look for the winner in `has_won`, and set the winner instead. This could lead to several possible bugs. Some students incorrectly look for three same characters in a row and included `?` as a possible winner (this would not have happened if you follow the question, since `?` is not a player!).

The other approach is that, instead of taking in a player as an argument, some students replicated the logic to check for `X` and `O` within `has_won`. This resulted in many lines of cut-and-paste code, and when students do not cut and paste and replace `X` with `O` correctly, bugs ensue.

Unnecessary Variables

Many students introduced unnecessary variables, complicated the code and obfuscated the logic. For instance,

```
   // check row
    int i = 0;
    for (long row = 0; row < 3; row += 1) {
       if (ttt[row][i] == player &&
           ttt[row][i + 1] == player &&
           ttt[row][i + 2] == player) {
           return true;
       }
    }
```

Since `i` is never changed, this variable is redundant. The more variables you used, the higher the likelihood that you introduce bugs. Indeed several students got caught by this.

Logical Expressions

To check if more than two variables are the same, you cannot write something like

```
      if (ttt[row][i] == ttt[row][i + 1] == ttt[row][i + 2] == player)
```

You should use the `&&` operator.

Memory Leaks

Many students still do something like this:

```
board[i] = malloc(3);
board[i] = cs1010_read_word();
```

causing memory leaks.

2 Sun (6 marks)

A prime number is a positive integer larger than 1 that is divisible only by 1 and itself.

A triangular number is a number of the form $\frac{x(x+1)}{2}$, where x is a positive integer. The first few triangular numbers are 1, 3, 6, 10, 15. There are $O(\sqrt{n})$ triangular numbers smaller than n.

Wei-Zhi Sun, a mathematician, conjectured in 2008 that, any positive integer, except 216, can be expressed as $p + t$, where p is either 0 or a prime number, and t is either 0 or a triangular number.

Write a program `sun` that, reads in a positive integer n, and print all possible pairs of p and t, in increasing order of p, such that $p + t = n$. If no such pairs can be found, print nothing.

Marking Scheme

Style	1 marks
Correctness	2 marks
Efficiency	3 marks
Documentation	0 marks

Efficiency: Your code should run in $O(n)$ time. In addition, we will deduct marks if your code has obvious redundant or duplicated work. Your code must be correct or almost correct to receive the efficiency marks.

Documentation: You do not have to write Doxygen documentation for each function. You are still encouraged to comment on your code to help the grader understand your intention.

Sample Runs

```
ooiwt@pe101:~/$ ./sun
10
0 10
7 3
ooiwt@pe101:~/$ ./sun
18
3 15
17 1
ooiwt@pe101:~/$ ./sun
216
```

Comments: This is meant to be another easy question that we expect most students to be able to answer. There are several approaches we can solve this.

Given n, we wish to find two numbers that adds up to n, one is either 0 or a prime, the other is either 0 or a triangular number.

Approach 1

A naive approach is the following:

```
for (long i = 0; i <= n; i += 1) {
        if (is_triangular(n-i) && is_prime(i)) {
                // output i and n-i
        }
}
```

where `is_prime` is the usual prime checker (but treating 0 as prime) and `is_triangular` is a new function that checks if a number is a triangular number.

How can we check if a number is a triangular number? This is not trivial and requires a bit of high-school mathematics. A triangular number is in the form of $x(x+1)/2$ for integer x. Given a number k, to check if $x(x+1)/2$ equals k for some integer k, we need to solve the quadratic equation, to find if the root to the equation above is an integer. But, finding the root involves floating point operations, and `double` might not accurately represent an integer.

Approach 2

We actually do not need to check if a number is triangular or not. Unlike prime, where there is no formula to describe a prime, a triangular number can be generated by the formula $\frac{x(x+1)}{2}$. So, we can generate all triangular numbers, and check:

```
long tri = 0;
for (long x = 0; tri <= n; x += 1) {
        tri = x*(x+1)/2;
        if (is_prime(n - tri)) {
                // output tri and n - tri
        }
}
```

This algorithm is rather straightforward, but there is one issue, we need to print the primes in increasing order, and the code above does the opposite.

One way to fix it is to reverse the order where we generate the triangular number. We can start with the largest triangular number less than n.

```
long tri = 0;
long max;
// find the smallest triangular number > n
for (max = 0; tri <= n; max += 1) {
        tri = max*(max+1)/2;
}
// step through triangular numbers in decreasing order.
for (long x = max - 1; x >= 0; x -= 1) {
        tri = x*(x+1)/2;
        if (is_prime(n - tri)) {
                // output tri and n - tri
        }
}
```

The running time for this approach is $O(n)$ as required. The first for loop above is $O(\sqrt{n})$. In the second for loop, since there are $O(\sqrt{n})$ triangular numbers, we call `is_prime` $O(\sqrt{n})$ times, and each `is_prime` invocation is $O(\sqrt{n})$.

Approach 3

The third approach tries to fix the reverse order of printing in another way. We have seen how to reverse a number recursively in PE1. We can use the same trick here.

```
void sun(long n, long x) {
        long tri = x*(x+1)/2;
        if (tri > n) {
                return;
        }

        // print the rest
        sun(n, x+1);

        // print this pair if sums to n
        if (is_prime(n - tri)) {
                // output tri and n - tri
        }
}
```

We call `sun` at most $O(\sqrt{n})$ time, and each call involved `is_prime` which is $O(\sqrt{n})$ time. So it works out to be $O(n)$.

Common Errors

When solving a problem, it is important to consider the boundary cases. In the context of loops, this requires checking for the loop initialization and termination condition carefully. Many students produced bugs that involve wrong initialization (should it be 0, 1, or 2?) or termination condition (should it be `<` or `<=`?).

Checking if a Number is Prime

Many students treat 1 as a prime number. 1 is not prime!

Surprisingly, despite having gone through how to check for primes many times, including a question specifically on the intricacy of the loop conditions for prime checking in the midterm, several students still do not do this correctly.

Checking if a Number is Triangular

As mentioned above, triangular number is something that we can generate with a formula, unlike a prime. Some students went for a solution that checks if a given number is a triangular number, with a code that looks like this.

```
for (long i = min; i <= max; i += 1 {
        if (i*(i+1)/2 == n) {
                return true;
        }
}
return false;
```

We can set `min` to 0, to handle to the special case of treating 0 as a triangular number. But many students incorrectly set `max` to `sqrt(n)` – perhaps to emulate the checks for prime or perhaps due to the question stating that there are $O(\sqrt(n))$ triangular numbers. This is too small and it will not correctly check all possible triangular numbers. There is no mathematical basis for giving $\sqrt{n}$ as the bound for `i`.

If you do the math, you will find that max should be $\frac{\sqrt{8n+1}-1}{2}$. You can also bound it with $\sqrt{2n}$.

Efficiency

Many students gave a $O(n\sqrt{n})$ solution, which looks something like:

```
for (long i = 0; i <= n; i += 1) {
        if (is_prime(i) && is_triangular(n - i)) {
                // print pair
        }
}
```

You will lose 3 marks for efficiency.

Note that even if we accept $O(n\sqrt{n})$ solution, the order of checking for prime and triangular matters – we should check the faster one first. If you have a $O(1)$ algorithm for checking triangular numbers, you should check that first, before checking for prime, which is $O(\sqrt{n})$, to take advantage of short circuiting and avoid redundant work.

3 Replace (11 marks)

Given a string, we wish to repeatedly perform the search and replace operation on the string. In each search-and-replace operation, we wish to replace all occurrences of a target substring with another replacement string.

For instance, given the string Tengineering, if we replace the target Tin with the replacement Ter, we will end up with the string Tengereererg. Next, if we replace Tere with TO, we get TengOOrg.

The replacement happens by scanning the input string from left to right. Each occurrence is replaced in sequence. For instance, if we replace the occurrences Taa in the input Taaaa with Tb, the output is Tbb.

You can assume that the target string to replace is either longer or equal to the length of the replacement.

Write a program Treplace that performs a sequence of search-and-replace operation on an input string.

The input of the program is as follows:

- The first line contains the string s to be operated on. You can assume that the input string does not contain white spaces.
- The second line contains a positive integer k, which is the number of search-and-replace operations.
- In the next k lines, each line contains two strings, s_1 and s_2, with the length of $s_2 \leq$ length of s_1.

The output of the program is a single line containing the output string.

CONSTRAINT: You are not allowed to call methods from the C string library, except Tstrlen.

Marking Scheme

Style	1 marks
Correctness	5 marks
Memory Management	3 marks
Efficiency	2 marks
Documentation	0 marks

Memory Management: To obtain full marks for memory management, make sure that your code does not have memory leaks, even if memory allocation fails.

Efficiency: To obtain full marks for efficiency, your code only needs to run in $O(nk)$ times for each search-and-replace operation, where n is the length of the input string, and k is the length of the target string. Your code must be correct or almost correct to receive the efficiency marks.

Documentation: You do not have to write Doxygen documentation for each function. You are still encouraged to comment on your code to help the grader understand your intention.

Sample Runs

```
ooiwt@pe101:~/$ ./replace
aaaa
```

```
1
aa b
bb
ooiwt@pe101:~/$ ./replace
engineering
2
in er
ere 0
eng0Org
```

Comments: This question is not as straightforward, but it builds on top of the needle and the haystack exercise question.

The following is the code to search for the needle (`search_for`) in the haystack (`word`), starting at position `i` in `word` .

```c
bool is_match(char *word, size_t i, char *search_for)
{
    size_t word_len = strlen(word);
    size_t search_len = strlen(search_for);
    size_t j = 0;
    for (j = 0; j < search_len && i + j < word_len; j += 1) {
        if (word[i + j] != search_for[j] ) {
            return false;
        }
    }
    if (j == search_len) {
        return true;
    }
    return false;
}
```

Suppose we found the target substring to search for, we can override the characters in `word` , starting from position `i` , by copying the replacement substring into `word` .

```c
void substitute(char *word, size_t i, size_t target_len, char *replace_with) {
    size_t replace_len = strlen(replace_with);
    size_t j = 0;
    while (j < replace_len) {
        word[i + j] = replace_with[j];
        j += 1;
    }
          :
}
```

So far, things are pretty straightforward. The more involved part is this: Since the replacement can be shorter than the target, what do we do with the remnants of the target string? Suppose we have `aaabbbccc` and we want to replace `bbb` with `_` , `word` becomes `aaa_bbccc` . What do we do with `bb` ?

Approach 1

We can override the remaining `bb` with `ccc` to compact the string.

```
void substitute(char *word, size_t i, size_t target_len, char *replace_with) {
    size_t replace_len = strlen(replace_with);
    size_t j = 0;
    while (j < replace_len) {
        word[i + j] = replace_with[j];
        j += 1;
    }
        // j now points to the end of the replacement string
        size_t word_len = strlen(word);
        while (i + target_len < word_len) {
                word[i + j] = word[i + target_len];
                i += 1;
        }
    word[i + j] = 0;
}
```

This will give us the correct out, but it will result in lots of redundant work in copying the remaining of string, if there are multiple instances of the target substring to replace.

Imagine we have the string `aaaaa......a` of length n, and we are replacing `aa` with `b`. After the first replacement, we have to copy $n-2$ times. After the second replacement, we have to copy $n-4$ times, etc. This leads to a $O(n^2/k)$ run time.

In fact, many of the copy above is redundant, and we hope that students can realize this point.

To be efficient, we only need to compact the string once after we replace all the instances of the target with the replacement.

Approach 2

We mark the remaining `bb` as removed, without compacting the string.

This approach is similar to what some students have used on the `mastermind` problem. We can have a second boolean array to mark if a character is removed, or just replace the character with something that never appears in the original string. Since the string is guaranteed not to contain white spaces, in the code below, we simply replace `bb` with white spaces.

```
void substitute(char *word, size_t i, size_t target_len, char *replace_with) {
    size_t replace_len = strlen(replace_with);
    size_t j = 0;
    while (j < replace_len) {
        word[i + j] = replace_with[j];
        j += 1;
    }
    while (j < target_len) {
        word[i + j] = ' ';
        j += 1;
    }
}
```

When all the substitution is done, we go through the string one more time and remove all the spaces.

```
void compact(char *word)
{
    size_t src = 0;
    size_t dst = 0;
    size_t word_len = strlen(word);

    while (src < word_len) {
        if (word[src] != ' ') {
            word[dst] = word[src];
            dst += 1;
        }
        src += 1;
    }
    word[dst] = 0;
}
```

The last step takes $O(n)$ time, so is within the bound of $O(nk)$.

Putting everything together, we have:

```
void replace(char *word, char *search_for, char *replace_with)
{
    size_t word_len = strlen(word);
    size_t search_len = strlen(search_for);
    for (size_t i = 0; i < word_len; i += 1) {
        if (is_match(word, i, search_for)) {
            substitute(word, i, search_len, replace_with);
            i += search_len - 1;
        }
    }
        compact(word);
}
```

This question is meant to be manageable by most students. Three functions, about 10 lines each, suffice to solve the problem. We notice that, however, many students wrote long and complex functions, most of which are due to "self-inflicted" complexity. Some common unnecessary code written by students are:

- read in the string to be replaced and its replacement as a line of text, then parse the string looking for spaces, to determine which part of the line is the string to be replaced and which part of the line is the replacement. This step is not needed if the students use `cs1010_read_word()` to read two words directly.

- read in the list of strings to be replaced and its replacement into an array of strings. This array has to be dynamic, and so using it comes with all the work you need to manage the memory properly (checks for NULL, free after use, free the past allocated memory if NULL, etc.) Doing so immediately increases the complexity of the solution up a notch. Storing into an array is not needed, since after you are done with the replacement, you are not going to use it anymore. So, the two strings can just be stored into two variables. The two variables can be reused for the next pair.
- use a temporary string to store the result after replacement. This is also unnecessary since the new string is guaranteed to be of equal length or shorter. Students who use a temporary string have to dynamically allocate the memory for it, and so have to do all the work to manage its memory and remember to free it after use. Furthermore, since multiple replacements may be needed, this becomes a common source of bugs – the second replacement needs to be done on the temporary string, not the original string.

Marking Schemes

For memory management, you need to at least check for NULL and free the dynamically allocated memory to score 1 point. If you do some of that but missed out on a few, you are awarded 2 marks. A perfect memory management code (no leaks even when malloc fails) is needed for 3 marks.

For efficiency, you need to be close to the correct solution to get the 2 marks. Note that, the intricacy of the efficiency comes from dealing with what happened with the replacement is shorter (whether you copy after every replacement or after all the replacement is done). So, if your solution does not deal with this situation, it is considered not close enough to the solution and no efficiency mark will be given.

4 Soil (11 marks)

In the farm owned by Old McDonald, a plot of land is divided into a grid of m-by-n cells. Old McDonald has hired you to look for a cell with the perfect soil salinity level to plant his crops. Armed with a salinity meter, you planned to check each of the $m \times n$ cells one-by-one, to find the cell with the right salinity level.

Just as you were about to start, Old McDonald pointed to a map of the grid and said, "There is something peculiar about this plot of land. If you walk through the cells row-by-row, from left to right, the soil salinity increases. If you walk column-by-column, from top to bottom, the salinity level increases. Further, no two cells have the same salinity level."

"Perfect," you said, "Now I only need to check $O(m + n)$ of the cells instead of $O(mn)$ cells!"

Write a program `soil` to simulate the process to search for salinity levels in a plot of land, by searching for a given integer in a 2D array of integers. Your program must read the following from the standard input.

The first line contains three positive integers, m, n, and k. m and nn represent the number of rows and columns of the 2D array.

The next m rows in the input contain the values to be stored in the 2D array. Each row has n integers. The following properties are guaranteed in the 2D array:

- In each row, the integers are in strictly increasing order
- In each column, the integers are in strictly increasing order
- The integers are unique in the 2D array.

The next k values represent the queries, i.e., the integers to search for. For each query, your program should look for it in the given 2D array, and either

- print the row and column indices of the 2D array that contains the value, separated by space, on its own line,
- or print the string "not found" if the corresponding value cannot be found.

Example

The following is a valid input 2D array:

```
1 4 8
2 5 9
3 7 10
```

Each row is sorted in increasing order from left to right, and each column is sorted in increasing order from top to bottom.

If the query to search for is 5, the program should print `1 1`. If the query is 6, the program should print `not found`.

Marking Scheme

Style	1 marks
Correctness	3 marks
Memory Management	1 marks
Efficiency	6 marks
Documentation	0 marks

Memory Management: You do not have to handle the case where memory allocation fails, but you should still ensure that your program does not have any memory leaks if it completes successfully.

Efficiency: Your algorithm must take $O(m+n)$ times to search a given query to obtain full marks. An $O(mn)$ solution is trivial and will receive 0 marks for efficiency. Any solution slower than $O(m+n)$ but faster than $O(mn)$ will receive partial marks for efficiency. Your solution must be correct or almost correct to receive the efficiency mark.

Documentation: You do not have to write Doxygen documentation for each function. You are still encouraged to comment on your code to help the grader understand your intention.

Sample Runs

```
ooiwt@pe101:~/$ ./soil
3 3 4
1 4 8
2 5 9
3 7 10
1
5
6
10
0 0
1 1
not found
2 2
ooiwt@pe101:~/$ ./soil
5 5 1
1  2  11 16 26
3  4  12 17 27
5  6  13 18 28
7  8  14 19 29
10 15 20 25 30
9
not found
```

Comments: This is another easy question to score partial marks on correctness.

The naive $O(mn)$ approach is to scan the 2D array with two for loops to look for a query. This would guarantee you 5 marks. This is kind of a give away too (easier to get than 6 marks of `tictactoe` I would say).

If you work a little more, you can get 8 marks, with a $O(m \log n)$ algorithm: For each row, do a binary search for the query. This should also be easy since the algorithm for binary search can be applied directly without any change.

We expect most students to be able to get at least 5 - 8 marks given how easy it is.

To get the last 3 marks, you need a $O(m + n)$ solution. This is a bit harder but we expect the As students can solve it.

Note that the $O(mn)$ solution above does not exploit the fact the 2D array is "row-sorted" and "column-sorted". The $O(m \log n)$ solution only exploits the fact that the 2D array is sorted in every row. To get $O(m + n)$, we need to use the fact that it is sorted in every row and every column.

This can be done by starting from the top-right corner (or bottom right), and comparing `a[0][n-1]` with the query `q`. Initially, the search area is the whole 2D array. We incrementally reduce the search area. If `q` is larger than `a[0][n-1]`, what can we know about where `q` is? Since every row is sorted, we know that it cannot be in the first row. We update the top-left corner of the search area to `a[1][n-1]`. On the other hand, if `q` is smaller than `a[0][n-1]`, we know that `q` cannot be in the right-most column. We update the top-left corner of the search area to `a[0][n-2]`.

So, after every comparison, we shrink the search area by one row or by one column. It only takes $O(m + n)$ until we search the whole 2D array.

Here is what the code looks like:

```c
void search(long *a[], long nrows, long ncols, long q) {
    long i = 0;
    long j = ncols - 1;
    while (i < nrows && j >= 0) {
        if (a[i][j] > q) {
            j -= 1;
        } else if (a[i][j] < q)
            i += 1;
        } else {
           cs1010_print_long(i);
           cs1010_print_string(" ");
           cs1010_println_long(j);
           return;
        }
    }
    cs1010_print_string("not found");
}
```

Given how easy it is to get 8 marks using binary search and 5 marks using exhaustive search, it is surprising that less than 35 students received 5 marks or more. Many students go straight for the efficient (but harder to get correct) $O(m+n)$ solution, but ended up with the wrong solution.

A common wrong solution is the following: given a query q, search for the row where the first element has the largest value less than q. Then, search only in this row for q. This is not correct. q can be in any row where the first element is less than q. So you need to search every such row (leading to a $O(mn)$ solution).

Another common wrong solution is to imitate binary search and divide the land into four quadrants. Then compare the query with the plot in the middle. Depending on whether the query is bigger or smaller, search in one of the quadrants. This is also incorrect since we can only eliminate one of the four quadrants for each comparison. A correct solution using this approach would need to recursively search in the remaining three quadrants.

The point of this question is to see whether students can use the sorted properties given in the data to come up with a faster algorithm than the naive solution – something you have seen in the searching/sorting units and Assignment 6. This is related to pattern recognition skills in computational thinking. Despite everyone doing well in Assignment 6, students seem to be having difficulties doing so in PE2.

Similar to `replace`, there is some self-inflicted complexity in the solution, such as allocating a dynamic array to store the elements and read in the salinity one-by-one, instead of using `cs1010_read_long_array`. Once you use a dynamic array, the complexity of your code increases significantly.

Another surprise is the number of buggy binary search implementations I encountered. Since this is an open-book PE, I did not expect this – just copy the one from the notes, and you will be fine!

5 Substring (11 marks)

Given a string, print all possible substrings of length k of the string.

The order in which the substrings are printed is defined as follows. Let s be in the input string; x and y be two substrings of s. Suppose that x_i and y_i is the first character that the substrings differ. If x_i appears before y_i in s, then x should be printed before y.

For instance, if $k = 4$ and the input is `singa`, the output should be

```
sing
sina
siga
snga
inga
```

Write a program `substring` that reads in a string s followed by a positive integer k from the standard input. You can assume that s contains only lowercase letters `a` to `z` with no repetition and k is at most the length of s.

The program should print to the standard output, all possible substring of s of length k in the order defined above, one substring per line.

Marking Scheme

Style	1 marks
Correctness	9 marks
Memory Management	1 marks
Efficiency	0 marks
Documentation	0 marks

Memory Management: You do not have to handle the case where memory allocation fails, but you should still ensure that your program does not have any memory leaks if it completes successfully.

Efficiency: There is no requirement for the running time of your algorithm. We may, however, still deduct marks if your code has obvious redundant or duplicated work.

Documentation: You do not have to write Doxygen documentation for each function. You are still encouraged to comment on your code to help the grader understand your intention.

Sample Runs

```
ooiwt@pe101:~/$ ./substring
singa 4
sing
sina
siga
snga
inga
ooiwt@pe101:~/$ ./substring
numbers 7
numbers
ooiwt@pe101:~/$ ./substring
```

```
done 1
d
o
n
e
```

Comments: This question requires finding all possible substrings of length k and is a simple tweak to the `permutation` problem.

We expect students who understand the solution to `permutation` to be able to solve this.

Recall that in `permutation` solution, we move the characters around and then recursively permute. This is because permutation involves all possible orders of the characters in the string.

In `substring`, the problem is slightly different, we maintain the order, but for each character, we may or may not include it in the substring.

Consider `abcd` and we want all substrings of length 2. If we include `a` in the substring, then we need to find all substrings of length 1, for `bcd`. If we exclude `a` in the substring, then we need to find all substrings of length 2, for `bcd`.

So we need to include or exclude a character while we recursively generate substrings. We also need to keep track of how many characters have been included (or excluded). There are multiple ways to do this. In the following, I use a boolean array to keep track of whether a character is included or not, and a long variable k to keep track of how many characters still need to be excluded.

```
/**
 * @param word     The original word.
 * @param chosen   A boolean array indicate if a character is included in the
 *                 substring.  chosen[i] is true if word[i] is included.
 * @param remains How many characters remaining to be excluded from curr onwards.
 * @param curr     The index of the first character of the remaining string to
 *                 construct the substring.
 * @param end      The last index of `word`
 */
void combine(char* word, bool chosen[], long remains, long curr, long end)
{
    if (remains == 0) {
        print(word, chosen);
        return;
    }
    if (curr != end) {
        // include curr and exclude remains from the rest
        combine(word, chosen, remains, curr+1, end);

        // exclude curr and remains-1 from the rest
        chosen[curr] = false;
        combine(word, chosen, remains-1, curr+1, end);

        // restore
        chosen[curr] = true;
    }
}
```

This recursive function is invoked with

```
  combine(word, chosen, end-k, 0, end);
```

where `chosen` is a boolean array initialzed to all `true` and `end` is `strlen(word)`.

The first base case for the recursion is when there is no more characters to exclude from the substring. In this case, we just print out the string, using the `chosen` array as the filter.

The other base case is when `curr == end`. In this case, we have reach the end of the string, and if `remains` is not yet 0, we can never construct a substring of size k. We just do nothing and return 0.

For the rest of the recursion, we choose to include or exclude the current character, and then recursively generate the substrings.

The code for `print` is

```
void print(char* word, const bool *chosen)
{
  size_t len = strlen(word);
  for (size_t i = 0; i < len; i += 1) {
    if (chosen[i]) {
      putchar(word[i]);
    }
  }
  putchar('\n');
}
```

You can further improve the code by avoiding redundant work. Note that, if `end - curr == remains`, then all the remaining characters should be excluded to have a valid substring of length k. So we can just exclude everything from `curr` onwards and print out the substring without a need to recurse further.

Many students just copied the code for permutation for this question. Some did a bit more by truncating the output from permutation to the required number of characters. Finding substrings is a different problem and requires a different solution. Even as some elements of `permute` (such as the recursion, the base case) remain, some thoughts into the relations between the two problems and how they differ is required.

END OF PAPER

This page is intentionally left blank.