NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL EXAMINATION II FOR
Semester 1 AY2022/2023

CS1010 Programming Methodology

November 2022 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 4 questions and comprises 8 printed pages, including this page.
2. The total mark for this assessment is 44. Answer **ALL** questions.
3. This is an OPEN BOOK assessment. You are only allowed to refer to any printed or handwritten materials.
4. You can assume that all the given inputs are valid and that the types `long` and `double` suffice for storing integer values and real values respectively, for the purpose of this examination.
5. Login to the special account given to you. You should see the following in your home directory:
   - The skeleton code `search.c`, `missing.c`, `group.c`, and `cluster.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 sample inputs and outputs
   - Two directories, `include` and `lib`, containing the files for the CS1010 I/O library.
   - The files `.clang-tidy` and `compile_flags.txt` for configuring `clang-tidy` and `clang` respectively.
   - The file `.vimrc` and the directory `.vim` that contains the default configuration and plugins for `vim`.
6. 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 to GitHub.
7. You can run the command `make` to automatically compile, run the tests (if compiled successfully), and run the command `clang-tidy` on your code.
8. Only the code written in `search.c`, `missing.c`, `group.c`, and `cluster.c` directly under your home directory will be graded. Make sure that you write your solution in the correct file. Failure to do so would result in 0 marks for the corresponding question.
9. There is a 1 mark deduction for every warning (including repeated warnings) generated by `clang` and `clang-tidy`. Programs that cannot be compiled would receive 0 marks.

1 Search (10 marks)

Consider a word puzzle game in which the goal is to find if a given word occurs in a given 6×6 grid of letters. The given word may appear horizontally (from left to right) vertically (from top to bottom) or diagonally (from top-left to bottom-right). For example, if the input is:

```
STERMS
CBVSXT
OALCPA
ISTOLT
PORDGU
ABGNMS
```

The word `TERMS` appears horizontally (in Row 0); the word `STATUS` appears vertically (in Column 5), and the word `BLOGS` appears diagonally (starting from Row 1 Column 1 to Row 5 Column 5).

```
STERMS
CBVSXT
OALCPA
ISTOLT
PORDGU
ABGNMS
```

Write a program `search` that reads the following in order from the standard input: (i) a 6×6 grid of characters, containing only capital letters A to Z; (ii) a positive integer n; and (iii) n words in capital letters to search for.

`search` prints to the standard output, for each word, the string "`yes`" if the word appears in the input grid, and the string "`no`" otherwise.

Your program must contain three functions: (i) `search_diagonal`, (ii) `search_vertical`, and (iii) `search_horizontal`, with five parameters each: `char **grid`, `size_t i`, `size_t j`, `char *word`, and `size_t l`. The functions search for `word` of length `l` in the `grid` starting at `i` and column `j`, in the diagonal, vertical, and horizontal direction respectively, and return `true` if the word is found and returns `false` otherwise. You may add additional functions as needed.

Grading Criteria

Correctness	6
Efficiency	0
Memory Management	1
Style	0
Documentation	3

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, where applicable. If memory management does not apply to your code, you will get this 1 mark for free.

Efficiency: No mark is allocated for efficiency. Your code is expected to run in $O(lmn)$ time when searching for a word of length l in a grid of size $m \times n$. We may deduct marks if your implementation performs redundant or duplicated work.

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

Sample Runs

```
ooiwt@pe101:~$ cat inputs/search.1.in
STERMS
CBVSXT
OALCPA
ISTOLT
PORDGU
ABGNMS
6
TERMS
STATUS
API
DOCS
BLOGS
PRIVACY
ooiwt@pe101:~$ ./search < inputs/search.1.in
yes
yes
no
no
yes
no
```

Comments: This question is a slightly more difficult version of `tictactoe` from Exercise 12. Instead of search of the symbol `X` or `O` diagonally, horizontally, or vertically, we search for a given string. This problem is also similar to `search` from Exercise 9. Here instead of search for a string in a 1D haystack, we are searching for it in a 2D haystack.

The solution is quite straight forward. Starting with every cell, we search horizontally, vertically, and diagonally until we find a matching string.

```
/**
 * Look for a target word in a given grid.
 *
 * @param[in] word The target the search.
 * @param[in] grid A 6x6 grid of letters to search in.
 *
 * @return true if the word is found; false otherwise.
 */
bool search_grid(char *word, char **grid) {
  size_t len = strlen(word);
  for (size_t m = 0; m < 6; m += 1) {
```

```
    for (size_t n = 0; n < 6; n += 1) {
      if (search_horizontal(grid, m, n, word, len) ||
          search_vertical(grid, m, n, word, len) ||
          search_diagonal(grid, m, n, word, len)) {
        return true;
      }
    }
  }
  return false;
}
```

The function above calls the three functions required by the question, to search for the word in three directions. The only tricky part here is that we need to be careful not to go out of bound of the 6×6 array. Since we need to do this multiple times, to avoid repetitive code we can handle this by writing a function.

```
/**
 * Check if a cell has a given letter.
 *
 * @param[in] grid   A 6x6 grid of letters.
 * @param[in] i      The row index of the cell to check.
 * @param[in] j      The col index of the cell to check.
 * @param[in] letter The character to check.
 * @return true if i and j are within bound and the grid cell (i,j) has
 *     the given letter.
 */
bool cell_has_letter(char **grid, size_t i, size_t j, char letter) {
  return i < 6 && j < 6 && grid[i][j] == letter;
}

/**
 * Search for a given word in the grid horizontally start from cell (m, n).
 *
 * @param[in] grid   A 6x6 grid of letters.
 * @param[in] m      The row index of the cell to start checking.
 * @param[in] n      The col index of the cell to start checking.
 * @param[in] word   The word to search for.
 * @param[in] len    The length of the word to search for.
 * @return true if word can be found in grid starting at (m, n); false otherwise.
 */
bool search_horizontal(char **grid, size_t m, size_t n, char *word, size_t len) {
  for (size_t idx = 0; idx < len; idx += 1) {
    if (!cell_has_letter(grid, m, n + idx, word[idx])) {
      return false;
    }
  }
  return true;
}

/**
 * Search for a given word in the grid vertically start from cell (m, n).
 *
 * @param[in] grid   A 6x6 grid of letters.
 * @param[in] m      The row index of the cell to start checking.
 * @param[in] n      The col index of the cell to start checking.
```

```c
 * @param[in] word   The word to search for.
 * @param[in] len    The length of the word to search for.
 * @return true if word can be found in grid starting at (m, n); false otherwise.
 */
bool search_vertical(char **grid, size_t m, size_t n, char *word, size_t len) {
  for (size_t idx = 0; idx < len; idx += 1) {
    if (!cell_has_letter(grid, m + idx, n, word[idx])) {
      return false;
    }
  }
  return true;
}

/**
 * Search for a given word in the grid diagonally start from cell (m, n).
 *
 * @param[in] grid   A 6x6 grid of letters.
 * @param[in] m      The row index of the cell to start checking.
 * @param[in] n      The col index of the cell to start checking.
 * @param[in] word   The word to search for.
 * @param[in] len    The length of the word to search for.
 * @return true if word can be found in grid starting at (m, n); false otherwise.
 */
bool search_diagonal(char **grid, size_t m, size_t n, char *word, size_t len) {
  for (size_t idx = 0; idx < len; idx += 1) {
    if (!cell_has_letter(grid, m + idx, n + idx, word[idx])) {
      return false;
    }
  }
  return true;
}
```

Common mistakes

- Many students avoid going out-of-bound of the 6×6 array by carefully computing the range in which `i` and `j` fall into when looping (e.g., `i + len <= 6`). This is fine as well, but doing so is bug-prone. Some students, for instance, incorrectly use the bound `i + len < 6`).
- Many students also looked for the starting letter before searching for the rest of the letter. This is fine, but it adds complexity to the code and is bug-prone – since the subsequent search starts at the second letter of the word to search for and some students made mistakes with the indices and the loop condition.
- Many students always passed in 0 for `i` and `j` in the search functions, and then increment `i` and `j` within them to loop through different letters in the grid. This design leads to complex code – since you now have three levels of nested loops. Bugs ensue.
- Several students search from bottom-right to top-left when searching diagonally. This is incorrect. Another common mistake is to search only along the main diagonal (i.e. `grid[i][i]`, `i` from 0 to 5.)
- Some students did not handle the case where the string to search is larger than 6 – resulting in either a memory error or an incorrect result of "yes".

- Note that once the code finds the string, it can stop searching and print "yes". Some students continue searching, leading to multiple "yes" being printed. Furthermore, the code should only print "no" after it completes searching all different directions. Some students print "no" once for every direction, leading to multiple "no"s being printed.

Marking scheme

For correctness, we deduct 1 mark per bug as usual. However, if a bug is repeated, we only deduct 0.5 marks for the second occurrence onwards. As listed in the common mistakes above, there are many bugs in many submissions. In cases where a submission has the correct approach but has 6 or more marks deducted, we may give 1 mark to correctness for effort.

We also deduct 1-3 marks if a student does not follow the given function specification, depending on how far the code is from the specification (different names, different parameters, etc.)

Most students who write the documentation receive 3 marks. 1-2 marks are deducted if a student did not follow the Doxygen format, did not document one or more of the parameters, did not documennt `@return`, or specify the wrong parameter type. A common error is to list `i` and `j` as `@param[in,out]`. This is not correct since `i` and `j` are passed by value, so any changes to them stays within the function only and does not affect the caller.

55 out of 244 students received at least 8 out of 10 marks.

2 Missing (10 marks)

You are given a list L of n integers. Each integer in L falls between the range of 1 and n (inclusive). Ideally, each integer in this range appears exactly once in the list. But this is not always the case. Our goal is to find out which number in the range of 1 to n is missing from the list. For example,

- Given $n = 5$ and $L = \langle 5, 3, 2, 1, 4 \rangle$, we have all the numbers in the range from 1 to 5. No integer is missing.
- Given $n = 10$ and $L = \langle 8, 7, 10, 9, 4, 6, 1, 8, 9, 2 \rangle$, both 3 and 5 are missing.

Write a program `missing` that reads, from the standard input, a positive integer n, followed by a list L of n integers, each integer falls between 1 and n. The program must print, to the standard output, the list of numbers between 1 to n that are missing from L, in increasing order. If none of the numbers is missing, print nothing.

Grading Criteria

Correctness	5
Efficiency	4
Memory Management	1
Style	0
Documentation	0

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, where applicable. If memory management does not apply to your code, you will get this 1 mark for free.

Efficiency: 4 mark is allocated for efficiency. To receive this mark, your solution must be correct or almost correct. Your code is expected to run in $O(n)$ time. We may deduct marks if your implementation performs 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:~$ ./missing
3
2 1 2
3
ooiwt@pe101:~$ ./missing
5
5 3 4 1 2
ooiwt@pe101:~$ ./missing
10
8 7 10 9 4 6 1 8 9 2
3
5
```

Comments: It is easy to score the 6 Correctness marks for this question. Here is an $O(n^2)$ solution: Repeatedly search for each integer 1 to n in the given array, and print out the integer if it is missing. An $O(n^2)$ receives 0 marks for efficiency.

Doing it in $O(n)$ is only slightly harder: we could build a lookup array that remembers which numbers have appeared in the inputs. We then scan through the lookup array to print the missing ones.

```
bool *find_existence(long *a, size_t n) {
  bool *exist = malloc(n + 1 * sizeof(bool));
  for (size_t i = 0; i < n; i += 1) {
    exist[i] = false;
  }
  for (size_t i = 0; i < n; i += 1) {
    exist[a[i]] = true;
  }
  return exist;
}

void print_missing(long *a, size_t n) {
  bool *exist = find_existence(a, n);
  for (size_t i = 1; i < n + 1; i += 1) {
    if (!exist[i]) {
      cs1010_println_size_t(i);
    }
  }
  free(exist);
}
```

We could also sort the array using counting sort and then scan through the sorted array to find the missing numbers. This would still achieve the same running time. The extra work incurred by sorting, however, is redundant for this question. We deduct 2 marks from efficiency for this.

Some students use the wrong type for the lookup array. It should be an array of `bool` since we only need to know if a number is missing or not. If a `long` array is used for counting how many times a number appears, it is OK. But if a `long` array is used to store if a number is there (1) or not (0), then we deduct 1 mark – just like in all our assignments/PE1, for using an integer to represent a boolean value.

151 out of 244 students received at least 8 out of 10 marks.

3 Group (12 marks)

In a class with n students $\{s_1, s_2, ..s_n\}$, the professor wishes to assign the students into groups. Every student must belong to exactly one group, and every group must contain at least one student. There is no restriction on group size. So, in the extreme case, all n students can be assigned to the same group. It is also possible for each student to be assigned to his/her own group, ending up with n groups of one student each. Many other group assignments are possible. Our goal in this question is to list out all possible ways n students can be assigned to groups.

For $n = 2$, there are two ways to assign the students:

- Both s_1 and s_2 are assigned into one group.
- s_1 and s_2 are assigned to a different group.

We can represent each possible way to assign the students as a sequence of numbers, by labeling each group with an integer ID, i.e., Group 1, Group 2, and so on. Suppose both students are assigned to the same group. We label the group as Group 1. Then the sequence that represents this assignment is `1 1`. If instead the students are assigned to two different groups, then we label the groups as Group 1 and Group 2. The sequence that represents this group assignment is `1 2`. All possible ways we can group two students are thus

```
1 1
1 2
```

Note that assignments `2 1` and `1 2` are considered equivalent, as only the labeling of group IDs differs. In this case, we only need to list the grouping with smaller lexicographical order among the equivalent grouping. This implies that the first student s_1 is always assigned to Group 1.

For $n = 3$, there are five ways to group the students:

```
1 1 1
1 1 2
1 2 1
1 2 2
1 2 3
```

Write a program `group` that reads, from the standard input, a positive integer n. The program must print, to the standard output, the list of all possible ways to assign n students to groups in lexicographical order.

Grading Criteria

Correctness	7
Efficiency	4
Memory Management	1
Style	0
Documentation	0

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,

where applicable. If memory management does not apply to your code, you will get this 1 mark for free.

Efficiency: 4 mark is allocated for efficiency. To obtain this mark, your approach must be correct. Your code is expected to run in $O(n \cdot f(n))$ time, where $f(n)$ is the number of ways to assign n students to groups. We may deduct marks if your implementation performs 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:~$ ./group
1
1
ooiwt@pe101:~$ ./group
2
1 1
1 2
ooiwt@pe101:~$ ./group
3
1 1 1
1 1 2
1 2 1
1 2 2
1 2 3
```

Comments: The question belongs to the class "generate all possible X", which you have seen in `permute`, `stone`, and `substring`. The general structure of the solution to this question is similar to the problems above: keep an array of items and recurse over all possibilities, once the last item is reached, print out the possibility generated.

The key to solving this question is to figure out how to recurse. For this problem, the recursion is easier to understand than the other similar problems above, since the situation is common in our daily life: Suppose you and a bunch of friends already divided yourselves up into groups and another friend wants to join. This friend can either join one of the existing groups, or (if nobody likes this friend) starts his/her own group.

This leads to the following solution:

```
void group(size_t current, size_t num_groups, size_t students[], size_t n) {
  if (current == n) {
    print(students, n);
    return;
  }
  // put current student in one of the exisitng groups
  for (size_t i = 1; i <= num_groups; i += 1) {
    students[current] = i;
    group(current + 1, num_groups, students, n);
  }
  // or start new group
```

```
  students[current] = num_groups + 1;
  group(current + 1, num_groups + 1, students, n);
}
```

To analyze the running time of the implementation above, note that each call to `group` leads to a possible assignment being printed. Printing each assignment takes $O(n)$, and there are $O(f(n))$ possible assignments. So the total time is $O(nf(n))$.

Mathematically-inclined students can read up more about Bell number, which is denoted as $f(n)$ in this question.

The marking for this question is rather straightforward – most students either got it correct (with 1-2 bugs) or completely wrong. A few students wrote complicated code to generate the outputs for the given test cases ($n < 4$) but their code fails for other ns. They receive 4 marks for correctness but no marks for efficiency.

Only 16 students received at least 10 out of 12 marks.

4 Cluster (12 marks)

One of the basic operations in contact tracing during a pandemic is to determine the infection clusters. If there is a path of transmission from person A to person B, then they belong to the same cluster.

Suppose we are given the information on who is in close contact with whom (and thus has transmitted the virus between them). We wish to count how many infection clusters are there.

We assume that "close contact" is a symmetric relation. If A is a close contact of B, then B is a close contact of A too. Because of this, we can represent the contact traces among n people as a lower triangular matrix (using a jagged 2D array). A proper type to store in each element of the matrix is `bool`. To simplify our life, however, we store each element of the matrix as a `char`, with '1' representing a close contact, '0' otherwise. The contact traces for n people are thus an array of n strings, each string containing characters of '0' and '1' only. The first row of the matrix is a string of length one; the second row is of length two; the third row is of length three, etc. The last character of each string (i.e., the diagonal of the matrix) is 1 since everyone has contact with him/herself.

We assume the virus transmits between close contacts only. We represent each person with an ID 0, 1, 2, ..., etc.

Suppose we have the following contact traces. The person with ID i has their information stored in Row i and Column i. Recall that if Row i and Column j is 1, it means that Person i is a close contact of Person j.

```
1
01
111
```

The contact traces above indicates that Person 2 is a close contact of Person 0 and Person 1. So there is only one cluster and all three of them belong to the same cluster.

As another example, the contact traces below shows 7 people.

```
1
01
001
0101
00101
010101
1000001
```

There are three clusters: (i) Person 1, Person 3, and Person 5; (ii) Person 2 and Person 4; and (iii) Person 0 and Person 6.

Write a program `cluster`, that reads from the standard input: (i) a positive integer n, (ii) followed by n lines of strings consisting of '1' or '0' representing the contact traces of these n people, The program prints, to the standard output, the number of clusters detected in the contact traces.

Grading Criteria

Correctness	7
Efficiency	4
Memory Management	1
Style	0
Documentation	0

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, where applicable.

Efficiency: Your code should take $O(n^2)$ time. Your solution must be correct or almost correct to receive the efficiency mark. We may deduct marks if your implementation performs 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:~$ ./cluster
7
1
01
001
0101
00101
010101
1000001
3
ooiwt@pe101:~$ ./cluster
5
1
01
001
0001
00001
5
```

Comments: This problem combines `contact` / `social` and `fill`. To count the clusters, we first need to identify the cluster. In `fill`, every pixel reachable from the initial pixel is colored with the same color and thus belongs to the same blob. Here, we want every person reachable from a given person i to be in the same cluster. Unlike in `fill` where we recursively explore four directions, here, we explore all close contacts of i, recursively. To avoid repeatedly counting the same cluster multiple times, we recursively remove the contact relationship of people in the same cluster (basically changing them from `1` to `0`), similar to how we change the colors of the adjacent pixels recursively.

Here is the code to count the clusters:

```
long count_cluster(char **network, size_t n) {
  long count = 0;
  for (size_t i = 0; i < n; i += 1) {
    for (size_t j = 0; j <= i; j += 1) {
      if (is_contact(network, i, j)) {
        count += 1;
        remove_contact(network, i, j);
        remove_cluster(network, n, i);
```

```
        remove_cluster(network, n, j);
      }
    }
  }
  return count;
}
```

and here is how we remove the contact relationship of people in the same cluster, recursively:

```
void remove_cluster(char **network, size_t n, size_t i) {
  for (size_t k = 0; k < n; k += 1) {
    if (is_contact(network, i, k)) {
      remove_contact(network, i, k);
      remove_cluster(network, n, k);
    }
  }
}
```

where `remove_cluster` is just:

```
void remove_contact(char **network, size_t i, size_t j) {
  if (j < i) {
    network[i][j] = NO_CONTACT;
  } else {
    network[j][i] = NO_CONTACT;
  }
}
```

To analyze the running time, noted that we iterate through the 2D array in `count_cluster`, which takes $O(n^2)$ time. We may possibly call `remove_cluster` for each (i, j) pair. It may appear that each `remove_cluster` takes $O(n)$, and thus the overall running time is $O(n^3)$. However, note that we only call `remove_cluster` if i and j are in contact (i.e., a value of `1`), and calling `remove_cluster` removes the contact (setting it to `0`). After the contact information becomes `0`, the pair is "passed over" in the scan. Since there are $O(n^2)$ elements in the array and we change the elements from 1 to 0 at most once, the overall running time is still $O(n^2)$.

A common mistake is to solve this purely iteratively without recursion. A common iterative approach is to consider only one-hop neighbors (or first-degree contacts) when forming clusters. A more sophisticated solution considers also two-hop neighbors. Regardless, a cluster could be formed, in the worst case, with an $(n-1)-hop$ neighbors, so these solutions are incorrect. Nevertheless, such iterative solutions that manage to solve the given test cases are given 4 marks for correctness and 0 for efficiency.

On the other hand, several students got the right idea about using recursion to tag all contacts in a cluster but did not solve them correctly. Such solutions received 4 marks for correctness and 4 for efficiency.

Only 19 students received at least 10 out of 12 marks.

END OF PAPER