NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL EXAMINATION 2 FOR
Semester 1 AY2023/2024

CS1010 Programming Methodology

November 2023 Time Allowed: 2.5 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 5 questions and comprises 10 printed pages, including this page.
2. The total mark for this assessment is 37. Answer **ALL** questions.
3. This is an OPEN BOOK assessment. But you are only allowed to refer to any printed or handwritten materials. No digital reference materials are allowed.
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 `prime.c`, `common.c`, `reversi.c`, `mode.c` and `matchmake.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
   - 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`.
7. 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.
8. You can run the command `make` to automatically compile, run the tests (if compiled successfully), and run the command `clang-tidy` on your code.
9. There is a time limit of 1 second for each test case given.

10. Only the code written in `prime.c`, `common.c`, `reversi.c`, `mode.c` and `matchmake.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.

11. Grading criteria:

    - Passing all the given test cases does not guarantee that your code will receive full marks.
    - The main marking criteria is correctness and is defined in the broad sense of using the various programming constructs in C (type, function, variable, loops, conditionals, arithmetic expressions, logical expressions) and managing the memory (allocation, deallocation) properly – not just producing the correct output.
    - Marks may be deducted for code that performs redundant or repetitive work. Some questions come with efficiency requirement that the submitted solution has to be adhered to to receive full marks.
    - Up to 1 mark may be deducted for style. You should write code that is clean, neat, and readable to avoid style mark deduction.

12. 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 Prime (6 marks)

Given a list of k positive integers ($k > 0$), we are interested in answering the following questions:

- Is there exactly one integer in the list that is a prime?
- Is there more than one integer in the list that is a prime?
- Is every integer in the list prime?

Task

Write a program `prime` that reads in a positive integer k and a list of k positive integers, and answer the three questions above in order, by printing out `yes` or `no` to the standard output.

The program `prime-main.c` has been given to you. It contains the function `is_prime`, which returns true if a given number is a prime and returns false otherwise. Do no change `prime-main.c`. Complete the program by completing the three functions below in `prime.c`, which answers the three questions above in order.

```
bool has_exactly_one_prime(long list[], size_t k) { .. }

bool has_more_than_one_primes(long list[], size_t k) { .. }

bool has_only_primes(long list[], size_t k) { .. }
```

You may add additional functions if necessary.

Efficiency Requirement

To get full marks, each function must not call `is_prime` more times than necessary.

Sample Runs

```
ooiwt@pe100:~$ ./prime
5
3 5 7 11 13
no
yes
yes
ooiwt@pe100:~$ ./prime
4
8 6 4 2
yes
no
no
ooiwt@pe100:~$ ./prime
1
13
yes
no
yes
```

Comments: This is a relatively straightforward question that assesses if students know how to avoid duplicate work and redundant work. To avoid duplicate work, avoid calling `is_prime` on the same number within a function more than once. To avoid redundant work, each function must return as soon as it can determine the answer.

2 marks are awarded for each function. As long as the function returns the correct answer without unnecessary calls to `is_prime`, full marks are given. Here is a sample answer:

```
bool has_exactly_one_prime(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
      if (count > 1) {
        return false;
      }
    }
  }
  return count == 1;
}

bool has_more_than_one_primes(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
      if (count > 1) {
        return true;
      }
    }
  }
  return false;
}

bool has_only_primes(long list[], size_t k) {
  for (size_t i = 0; i < k; i += 1) {
    if (!is_prime(list[i])) {
      return false;
    }
  }
  return true;
}
```

The only tricky part is to ensure that `has_exactly_one_prime` does not return `true` when there is no prime number in the list.

A function that does not ensure that `is_prime` is called only when necessary receives 0 marks, even when the code returns the correct answer. Some students wrote the following:

```
bool has_exactly_one_prime(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
    }
```

```
  }
  return count == 1;
}

bool has_more_than_one_primes(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
    }
  }
  return (count > 1);
}

bool has_only_primes(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
    }
  }
  return (count == k);
}
```

Such a solution always calls `is_prime` k times, regardless of the input.

A small number of students try to optimize the number of calls to `is_prime` by remembering past calls to `is_prime` *across* different calls to the three functions.

```
#define PRIME 1
#define NOT_PRIME 0

bool has_exactly_one_prime(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (is_prime(list[i])) {
      count += 1;
      list[i] = PRIME;
    } else {
      list[i] = NOT_PRIME;
    }
  }
  return count == 1;
}

bool has_more_than_one_primes(long list[], size_t k) {
  long count = 0;
  for (size_t i = 0; i < k; i += 1) {
    if (list[i] == PRIME) {
      count += 1;
      if (count > 1) {
        return true;
      }
    }
  }
```

```
  return false;
}

bool has_only_primes(long list[], size_t k) {
  for (size_t i = 0; i < k; i += 1) {
    if (list[i] == NOT_PRIME) {
      return false;
    }
  }
  return true;
}
```

All calls to `is_prime` are done in `has_exactly_one_prime`. This is a good trick, but the approach is problematic for several reasons: First, k calls to `is_prime` are made no matter what. This could be significantly more than necessary. Second, the functions `has_more_than_one_primes` and `has_more_than_one_primes` can no longer be independently called. The three functions written are tightly tied to the given `main`. If a programmer calls these functions in a different order, a wrong value will be returned. Such an approach shows that students do not understand the basics of what a C function is. Finally, it changes `list` so the original values are lost. The caller would have to create a copy of the list before calling the functions. (In retrospect, the parameter `list` should have been defined as `const` to weed out such attempts).

We gave up to 3 marks for such an approach.

2 Common (7 marks)

Given a list of words, we wish to find out which alphabet is most commonly used, treating uppercase and lowercase letters as the same. If two alphabets appear an equal number of times, we break ties by preferring the earlier alphabet in the alphabetical order.

Task

Write a program `common` that reads a positive number n from the standard input, followed by n words comprising only lower and uppercase letters of the alphabet. The program should print to the standard output the most commonly used alphabet in uppercase, followed by how many times this letter has appeared in the input.

Efficiency Requirement

Suppose there are k characters in the input, your code should run in $O(k)$ times to receive full marks. A solution that takes $O(k \log k)$ time will receive at most 4 marks. Solution that takes $O(k^2)$ time will receive at most 2 marks.

Sample Runs

```
ooiwt@pe100:~/$ ./common
3
abc Aa BCD
A
3
ooiwt@pe100:~/$ ./common
1
sUpErCaLiFrAgIlIsTiCeXpIaLiDoCiOuS
I
7
ooiwt@pe100:~/$ ./common
9
THE quick Brown fOx juMps
oveR the lazy dog
O
4
```

Comments: This is another straightforward question that combines elements from exercise questions `up` and `counter` as well as `max` from the notes.

The key idea is to build a frequency table that counts how many times each character appears.

```c
void fill_freq(size_t freq[], char *str)
{
  size_t i = 0;
  while (str[i] != '\0') {
    if ((str[i] >= 'a') && (str[i] <= 'z')) {  // lowercase
```

```
      str[i] += ('A' - 'a');  // convert to upper case
    }
    freq[str[i] - 'A'] += 1;  // map from letter to index
    i += 1;
  }
}
```

This function is called for every word in the input:

```
size_t freq[26] = {0};
for (size_t i = 0; i < n; i += 1) {
  fill_freq(freq, word[i]);
}
```

Filling up the frequency table takes $O(k)$ time.

After this, we only need to scan through the frequency table and find the most frequently occured letter. This algorithm is from the very first lecture of CS1010. The only tricky part here is that we use `>` to break ties by choosing the smaller index.

```
size_t find_index_of_max(size_t freq[])
{
  size_t max_index = 0;
  for (size_t i = 1; i < 26; i += 1) {
    if (freq[i] > freq[max_index]) {
      max_index = i;
    }
  }
  return max_index;
}
```

We then print out the answers:

```
size_t index = find_index_of_max(freq);
putchar((char)index + 'A');
cs1010_println_string("");
cs1010_println_size_t(freq[index]);
```

Despite the simplicity of this question, however, only 90+ students received full marks. Common mistakes include:

- Not managing the memory correctly (e.g., did not check for memory allocation error)
- Calling `strlen` repeatedly in a loop. Since `strlen` is $O(k)$, technically the code runs in $O(k^2)$ time (when the input is one word) and a maximum of 2 marks was to be awarded. This mistake, however, is so widespread that we decided to deduct 1 mark for this.
- Using a dynamically allocated array for an array of which the size is known (e.g., a frequency table of size 26). A fixed-length array should be used instead.

A bad habit among the students is the use of ASCII values (97, 65, etc) to refer to the letters, instead of the more readable character literals (e.g., `'a'`, `'A'`). Several students introduced bugs due to the usage of incorrect ASCII values.

3 Reversi (8 marks)

The game Reversi is played on an 8×8 board, where two players with black and white pieces take turns to place their pieces on empty cells on the board. Let's call the players Player 0 and 1.

Player 1, when placing a piece on the board, may capture the pieces of Player 0. This capture process is done by considering eight different directions (up, down, left, right, and the four diagonals). All consecutive pieces of Player 0 that fall between the new piece being placed and an existing piece of Player 1, in a straight line in any of the eight directions, will be captured, flipped, and owned by Player 1.

Examples

Suppose we have the following board, with the two color pieces represented as `0` and `1`. Empty cells are represented with `.` on the board.

```
........
......1.
.10010..
.100..01
.11.100.
......1.
........
........
```

Suppose Player 1 places a piece of `1` at position X below:

```
........
......1.
.10010X.
.100..01
.11.100.
......1.
........
........
```

Player 1 will capture the 0 immediately to the left and the two 0s at the bottom. We end up with the following board:

```
........
......1.
.100111.
.100..11
.11.101.
......1.
........
........
```

However, suppose Player 1 places a `1` piece at the position Y below:

```
........
......1.
.10010..
.100Y.01
.11.100.
......1.
........
........
```

The player captures four pieces: the two pieces on the left, one piece on the lower right, and one piece on the top right. We get the following:

```
........
......1.
.10011..
.1111.01
.11.110.
......1.
........
........
```

Task

Write a program `reversi`, which reads in the state of the current board, and finds the maximum number of pieces that can be captured by Player 1 by placing `1` on an empty cell on the board. We break ties by preferring placement towards the top and the left. The program should print out the number of pieces flipped and the resulting state of the board.

It is possible to place a piece without capturing any piece from Player 0.

Sample Run

```
ooiwt@pe100:~$ ./reversi
........
......1.
.10010..
.100..01
.11.100.
......1.
........
........
4
........
......1.
.10011..
.1111.01
.11.110.
......1.
........
........
```

Comments: The logic required to solve this task is relatively simple. This question mainly assesses if students know how to break the problem down into smaller tasks. The length of code that a student would write and the amount of bugs due to copy-pasting can be significantly different for students who decompose this solution into functions.

To solve this, we need to iterate through the empty positions in the board, count how many pieces would be flipped, and then find the maximum among these empty positions.

```
long max = -1;
long max_r = 0;
long max_c = 0;
for (long r = 0; r < 8; r += 1) {
  for (long c = 0; c < 8; c += 1) {
    long count = count_flips(board, r, c);
    if (count > max) {
      max = count;
      max_r = r;
      max_c = c;
    }
  }
}
```

A very common mistake among students is that they initialize the "maximum so far" incorrectly, setting `max` to 0. This is disappointing since our first lesson on algorithm involves finding the maximum and, during the tutorials, we spent quite a bit of time on how to initialize the "maximum so far": We either (i) set it to one of the candidates for maximum, or (ii) set it to something smaller than all the candidates. We even spent time on analyzing the invariant of the loop. Setting `max` to 0 and position to (0, 0) is neither (i), because position (0, 0) might not be empty, nor (ii), since an empty position might not flip any opponent's pieces.

The next part of this solution is to count how many flips are there for each position. Many students copy-and-pasted 8 copies of the same code (one for each direction), and ended up with subtle mistakes (`+` becomes `-`, a row becomes a column, `>` becomes `<`, etc). The resulting code is a few pages long, making it hard to read, debug, and fix.

A simpler way is to abstract out the directions as parameters, as you have seen in the code we showed for `maze` and `fill`.

```
long count_flips_in_one_dir(char **board, long r, long c, long r_step, long c_step) {
  long count = 0;
  do {
    r += r_step;
    c += c_step;
    count += 1;
  } while (in_range(r, c) && board[r][c] == '0');

  if (in_range(r, c) && board[r][c] == '1') {
    return count - 1;
  }
  return 0;
}

long count_flips(char **board, long r, long c)
{
```

```
  if (board[r][c] != '.') {
    return -1;
  }

  long dir[8][2] = {{1, -1}, {1, 1}, {-1, 1}, {-1, -1},
                    {0, -1}, {0, 1}, {1, 0}, {-1, 0}};

  long total = 0;
  for (long i = 0; i < 8; i += 1) {
    total += count_flips_in_one_dir(board, r, c, dir[i][0], dir[i][1]);
  }
  return total;
}
```

After counting, we can update the board as follows:

```
void flip_in_one_dir(char **board, long r, long c, long r_step, long c_step) {
  do {
    board[r][c] = '1';
    r += r_step;
    c += c_step;
  } while (in_range(r, c) && board[r][c] == '0');
}

void flip(char **board, long r, long c)
{
  long dir[8][2] = {{1, -1}, {1, 1}, {-1, 1}, {-1, -1},
                    {0, -1}, {0, 1}, {1, 0}, {-1, 0}};

  board[r][c] = '1';
  for (long i = 0; i < 8; i += 1) {
    if (count_flips_in_one_dir(board, r, c, dir[i][0], dir[i][1]) != 0) {
      flip_in_one_dir(board, r, c, dir[i][0], dir[i][1]);
    }
  }
}
```

The usual marking scheme for CS1010 deducts one mark for each occurrence of a bug, even if the bug is repeated. This marking scheme rewards those students who abstract out common functionalities into functions, rather than copy-pasting their code, since within a function, a buggy code only manifests once (and hence penalized once). The number of students who copy-pasted a chunk of code 8 times (or 16 times, if we include both counting and flipping) is too many. As such, we ran out of marks to deduct very quickly. For this question, we made an exception in the marking scheme, and deducted marks for a bug only once, if the bug is copy-pasted to other parts of the code (particularly, for a different direction).

4 Mode (8 marks)

A tutor has processed the marks of an exam and produced a bar chart representing the histogram of the mark distribution. The marks are integers and fall between 0 to $n-1$. The number of students scoring mark i, or the frequency of i, is denoted as $f(i)$. The mode of the distribution is k (The mode of the mark distribution is the mark scored by the most number of students). We assume that there is a single mode.

The mark distribution follows a (possibly skewed) bell curve. This means that, for $i \leq k, j \leq k$, if $i < j$, then $f(i) \leq f(j)$. For $i \geq k, j \geq k$, if $i > j$, then $f(i) \leq f(j)$.

The bar chart is drawn on a canvas with m rows and n columns. The bars of the histogram are marked with `#`. The empty pixels on the canvas are marked with `.`. There are $f(i)$ number of `#`s on Row i. In each row, reading from left to right, all occurrences of `#`s (if any) appear before `.`.

Example

The example below shows a histogram with 8 rows, for marks 0 to 7. There is one student each scoring 0 and 1 marks. Two students each scored 2 marks and 5 marks. Three students scored 3 marks and four students scored 4 marks. No students scored 6 marks or above.

```
#....
#....
##...
###..
####.
##...
.....
.....
```

The mode of the mark distribution is 4. There are four students scoring 4 marks.

Task

Given the histogram of mark distribution, write a program called `mode` that finds the mode k and $f(k)$. The program should read the following from the standard input:

- Two integers m and n. m is the number of rows and n is the number of columns.
- m lines of text, containing only `.` and `#`, representing the histogram.

The program writes to the output, the mode k and the number of students scoring k.

The main program `mode-main.c` has been given to you. Do not change this file. Solve this problem by filling in the function `find_mode` in the file `mode.c`, with the following header:

```
void find_mode(char **histogram, size_t m, size_t n,
    size_t *mode_index, size_t *mode_frequency) { .. }
```

The parameters are as follows: `histogram` contains a 2D array of characters corresponding to the histogram. `m` and `n` are the number of rows and the number of columns respectively. `mode_index` and `mode_frequency` are pointers to the mode (i.e., k) and the frequency $f(k)$ (i.e., the number of students scoring k).

You may add additional functions if necessary.

Efficiency Requirement

To qualify for full marks, your function `find_mode` must run in $O(\log m \times \log n)$ when all $f(i)$ is unique and $O(m \times \log n)$ when some of the adjacent marks have the same frequency.

An $O(mn)$ algorithm is trivial. Solving it this way will not earn any marks. Any algorithm faster than $O(mn)$ and slower than $O(\log m \times \log n)$ (when all $f(i)$ is unique) qualifies you for 4 marks at most.

Sample Runs

```
ooiwt@pe100:~$ ./mode
8 5
#....
#....
##...
###..
####.
##...
.....
.....
4
4
ooiwt@pe100:~$ ./mode
10 6
......
......
#.....
##....
###...
###...
###...
###...
###...
#####.
9
5
```

Comments: We need two binary search runs. One across the rows to find the peak, the other across the columns to find the last `#`.

The following code finds the value of the histogram at a particular index in $O(\log n)$ time. This makes use of the propery that all the `#`s appear before the `.`s in each row, so we can avoid scanning the entire row.

```
size_t score(char **histogram, size_t r, size_t m) {
  size_t i = 0;
  size_t j = m - 1;
  while (i < j) {
    size_t mid = (i + j) / 2;
```

```
    if (histogram[r][mid] == '#' && histogram[r][mid+1] == '#') {
      i = mid + 1;
    } else {
      j = mid;
    }
  }
  return i+1;
}
```

With the above function, we can find the mode using a binary search, by making use of the property that the scores increase and then decrease. This is similar to the problem `valley` from the exercises.

```
size_t index_of_mode(char **histogram, size_t i, size_t j, size_t m) {
  while (i < j) {
    size_t mid = (i + j) / 2;
    if (score(histogram, mid, m) > score(histogram, mid + 1, m)) {
      j = mid;
    } else {
      i = mid + 1;
    }
  }
  return i;
}
```

Putting it together, we have

```
void find_mode(char **histogram, size_t m, size_t n, size_t *mode_index, size_t *mode_frequency)
  *mode_index = index_of_mode(histogram, 0, m - 1, n);
  *mode_frequency = score(histogram, *mode_index, n);
}
```

5 Matchmake (8 marks)

A set of n participants have registered themselves for a matchmaking event. Each participant has also selected zero or more participants they are interested in meeting with at the matchmaking event. The event organizer needs to come up with a pairing of the participants and schedule them to dine together. Each participant can only dine together with one other participant, and the participants have to mutually indicate their interests to each other in order to dine together. Some participants might be left out of the schedule and not paired with another. These participants dine alone.

The participants' interest is marked with a matrix of $n \times n$ containing only `1` and `0`s. If we index the participants with 0 to $n - 1$, then a `1` in row i and column j means that participant i is interested in meeting participant j. Conversely, a `0` in row i and column j means that participant i is not interested in meeting participant j. The diagonal of the matrix is always 0.

Given n and the matrix representing the indicated interests to meet each other, write a program that prints out all possible schedules.

Example

Suppose we have the following $n = 3$ participants. Participant 0 is interested in meeting 1 and 2. Both participants 1 and 2 are interested in meeting participant 0, but not each other.

```
011
100
100
```

There are three possible schedules. The first schedule is for 0 to dine with 1. Participant 2 is left out. The second possible schedule is for 0 to dine with 2. Then Participant 1 is left out. Finally, the organizers might decide not to pair anyone. All three participants dine alone.

Suppose we add Participant 3 who is interested in meeting everyone else and everyone else is interested in meeting Participant 3. The input is:

```
0111
1001
1001
1110
```

In this case, there are eight possibilities:

- 0 dines with 1; 2 dines with 3.
- 0 dines with 1; 2 and 3 dine alone.
- 0 dines with 2; 1 dines with 3.
- 0 dines with 2; 1 and 3 dine alone.
- 0 dines with 3; 1 and 2 dine alone.
- 1 dines with 3; 0 and 2 dine alone.
- 2 dines with 3; 0 and 1 dine alone.
- All four participants dine alone.

Task

Write a program called `matchmake` that reads in a positive integer n ($n \geq 2$), followed by n strings representing the interest matrix. Each string contains only `0` and `1` and is of length n. The program must print all possible schedules.

Each schedule is printed on one line and must be printed in the following order. Paired participants are printed first. For each pair i and j ($i < j$), print i followed by `-` followed by j. Print them in increasing order of the ids of the first participant in the pair. There should be one space between each pair. Finally, print all the participants who are not paired, in increasing order of their ids, with a space in between each id.

The different possible schedules, however, can be printed in any order. When `test.sh` compares the output from your program with the expected output, it will first sort the schedules printed in lexicographical order. As long as the sorted schedules are the same as the expected output of a test case, your program is considered to have passed the test case.

```
ooiwt@pe100:~$ ./matchmake
3
011
100
100
0-1 2
0-2 1
0 1 2
ooiwt@pe100:~$ ./matchmake
4
0111
1001
1001
1110
0-1 2-3
0-1 2 3
0-2 1-3
0-2 1 3
0-3 1 2
1-3 0 2
2-3 0 1
0 1 2 3
```

Comments: Solving this question involves enumerating all possible schedules. We first need an array to represent the schedule. Let's call the array `dine_with`, where `dine_with[i]` records who `i` dines with (could be itself). We set `dine_with` to an invalid id (`UNMATCH`) if it is not yet determined.

The idea, like all other enumeration questions, is to fix one possibility, and then recursively explore the rest of the possibilities. When we are done exploring all possibilities, we print out what we have, and then backtrack.

```
void generate(char **match, long n, long dine_with[], long curr)
{
```

```
  if (curr == n) {
    // Reach the last user.  Print out the schedule
    print(dine_with, n);
    return;
  }
  if (dine_with[curr] != UNMATCH) {
    // User curr is already scheduled to dine with someone.
    // Continue to the next user
    generate(match, n, dine_with, curr + 1);
  } else {
    // User curr is not yet paired with someone.
    // Find someone to pair with
    for (long i = curr + 1; i < n; i += 1) {
      if (dine_with[i] == UNMATCH && is_interested(match, curr, i)) {
        // Pair with i
        dine_with[curr] = i;
        dine_with[i] = curr;
        // Continue to the next user
        generate(match, n, dine_with, curr + 1);
        // Unpair with i and try other pairing
        dine_with[i] = UNMATCH;
        dine_with[curr] = UNMATCH;
      }
    }
    // Pair with self
    dine_with[curr] = curr;
    // Continue to the next user
    generate(match, n, dine_with, curr + 1);
    // Unpair with self
    dine_with[curr] = UNMATCH;
  }
}
```

Four students solved this fully and were awarded 8 marks. A few students wrote the right recursive structure and conditional checks but had various bugs. We awarded 4 partial marks (before other deductions such as styles and warnings). Other solutions that are too far away from the correct ones (e.g., does not recurse, recurse only once (and thus output only one schedule)) received 0.

END OF PAPER