NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 1 AY2022/2023

CS1010 Programming Methodology

November 2022 Time Allowed 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 16 questions and comprises 14 printed pages, including this page.
2. Write all your answers in the answer sheet provided.
3. Please write and shade your student number in the corresponding box in the answer sheet.
4. The total marks for this assessment is 90. Answer **ALL** questions.
5. This is an **OPEN BOOK** assessment.
6. **Unless otherwise specified**, you can assume that in all the code given: (i) no overflow nor underflow will occur during execution; (ii) all given inputs are valid and fits within the specified variable type; (iii) compile without syntax errors; and (iv) all the required header files are already included.

Part I

Multiple Choice Questions (36 points)

- For each of the questions below, **write your answer in the corresponding answer box on the answer sheet.** Each question is worth 3 points.
- If multiple answers are equally appropriate, pick one and write the chosen answer in the answer box. Do NOT write more than one answer in the answer box.
- If none of the answers is appropriate, write X in the answer box.

1. (3 points) Assuming that an integer `n` has more than two digits. To check if the last two digits of an integer `n` are the same, we can use the following logical expression:

   A. `(n % 10) == 0`

   B. `(n / 100) == (n / 10)`

   C. `(n % 100) == (n % 10)`

   D. `((n % 100) % 11) == 0`

   E. `((n / 10) % 10) == (n / 10)`

   Write X in the answer box if none of the choices above is correct.

   Solution: D.

   This question tests if students understand the `/` and `%` operators. An alternative is to check `((n / 10) % 10) == (n % 10)` (so Option E is almost correct).

2. (3 points) Which of the following declaration of `count` would cause the program below to print `Wakanda` forever, in an infinite loop? Note: Overflow/underflow may happen for this question.

```
while (count >= 0) {
  cs1010_println_string("Wakanda");
  count -= 1;
}
```

   A. `double count = 100.0;`

   B. `size_t count = 100;`

   C. `char count = 100;`

   D. `long count = 100;`

   Write X in the answer box if none of the choices above is correct.

   Solution: B.

   This question assesses if students understand the behavior of commonly used types and the concepts of underflow/overflow. Recall that `size_t` stores unsigned value, so it will never be negative. Thus, the loop condition is always true and once we enter the loop, we will never exits.

3. (3 points) Consider the function below:

```
bool check(bool a, bool b, bool c, bool d) {
  if (a) {
    if (b) {
      return true;
    }
  }
  if (c) {
    return true;
  }
  if (d) {
    return true;
  }
  return false;
}
```

The body of the function can be rewritten as

A. `return a && b && c && d`;

B. `return a || b || c || d`;

C. `return (a || b) && (c || d)`;

D. `return (a || b) && c && d`;

E. `return (a && b) || c || d`;

Write X in the answer box if none of the choices above is correct.

Solution: E.

This question assesses if students understand the flow of `if-else` statements, logical expressions, and their equivalence.

4. (3 points) Analyze the function below. `cond` is a boolean function that checks if `i` satisfies certain unknown condition.

```
void corge(long m, long n) {
  long i = m;
  while (cond(i)) {
    if (i % n == 0) {
      return;
    }
    i -= 1;
  }
  // Line A
}
```

What can we assert at Line A?

(i) `(i + 1) % n != 0`

(ii) `i % n != 0`

(iii) `i < m`

A. (i) only

B. (ii) only

C. (iii) only

D. (i) and (iii) only

E. (ii) and (iii) only

F. (i) and (ii) only

G. (i), (ii), and (iii)

Write X in the answer box if none of the choices above is correct.

Solution: X.

This question assesses if students are able to reason about the behavior of code that involves loops.

If we enter the loop and we reach Line A, then we know that the code did not return in the middle of the loop due to `i % n == 0`. We can assert that (i) is true (the `+1` is due to `i` decrementing by 1 after the `if` statements. Furthermore, since we start with `i == m` before the loop, we know that `i < m` due `i -= 1` being run at least once.

However, since we do not know whether `cond(i)` is true we cannot tell whether we ever enter the loop or not. The only assertion we can make at Line A is `i <= m`. So, none of (i), (ii), and (iii) are correct.

5. (3 points) Analyze the buggy function below:

```
void mew(size_t left, size_t right) {
  if (left > right) {  // Line B
    return;
  }
  size_t mid = (left + right)/2;
  mew(left, mid);
  mew(mid, right);
}
```

A call to `mew` might lead to infinite recursion and a stack overflow. Which of the following changes to Line B suffices to fix the bug?

(i) Change the condition at Line B to `left >= right`.

(ii) Change the condition at Line B to `left == right`.

(iii) Change the condition at Line B to `left + 1 >= right`.

A. (i) only

B. (ii) only

C. (iii) only

D. (i) and (iii) only

E. (ii) and (iii) only

F. (i) and (ii) only

G. (i), (ii), and (iii)

Write X in the answer box if none of the choices above is correct.

Solution: C.

This question tests if students are able to trace through the behavior of divide-and-concur recursive calls. First, students should understand why `mew` is buggy. It is possible for `mid` to be the same as `left` or `right`. For example, a call to `mew(0,1)` would lead to `mid` being 0 and so `mew(0, 1)` is called over and over again.

For (i), if we change the condition to `left >= right`, then we continue with the recursive calls if `left < right`. It is possible for `left` to be `right - 1`. Then `mid` would be `(2*left + 1)/2`, which is just `left`. So we are calling `mew(left, right)` again, leading to infinite recursion.

For (ii), if we change the condition to `left == right`, then we continue with the recursive calls if `left != right`. This is a weaker condition than (i) and the same as (i) could happen.

For (iii), we change the condition to `left + 1 >= right`, i.e., we continue with the recursive calls if `left + 1 < right`, or `left + 2 <= right`. When `left + 2 == right`, `mid` is `(2*left + 2)/2`, which is `left + 1` or `right - 1`. So both recursive calls continue without calling `mew(left, right)` again.

6. (3 points) Consider the three functions below.

```
void bat() {
  char *str[3];
  str[0] = "bat";
}

void cow() {
  char *str = malloc(4);
  str = "cow";
}

void fox() {
  char str[4] = "fox";
}
```

Which function(s), when executed, would cause a memory-related error?

A. `bat` only.

B. `cow` only.

C. `fox` only.

D. `bat` and `cow` only.

E. `cow` and `fox` only.

F. `bat` and `fox` only.

G. `bat` , `cow` , and `fox` .

Write X in the answer box if none of the choices above is correct.

Solution: B.

Question 6 checks if students are familiar with memory management and common memory-related errors.

`bat` declares three strings but initialized the first one. The other two strings are not initialized but there are never used. So there is no error. `cow` allocates memory for a string of 4 characters on the heap and assigned it to `str` . It subsequently assigns `str` to another string. The allocated memory can never be accessed again and will be leaked. `fox` allocates memory for a string of 4 characters on the stack and initializes it. Memory allocated on the stack is automatically reclaimed once the function exits.

7. (3 points) Consider the following 2D array declaration inside `main`.

   ```
   long image[480][640];
   ```

   Which of the expression below has the same value (i.e., refers to the same memory address) AND the same type as `image + 1`?

   A. `&image[1][0]`

   B. `&image[0][1]`

   C. `&image[1][1]`

   D. `&image[1]`

   E. `image[0] + 1`

   Write X in the answer box if none of the choices above is correct.

   Solution: D.

   This question checks if students are familiar with array decays and 2D arrays. `image[0]` is a 1D array of `long`; `image` decays into `&image[0]`, which is the memory address of the first row, a 1D array. So the type of `image` is a pointer to an array of `long`. This excludes A-C as the answer. `image + 1` performs pointer arithmetic on this address and advances the pointer by one row. `image + 1` thus is a memory address that points to the second row, i.e., `&image[1]`.

8. (3 points) Consider the code snippet below. Assume that `nrows` and `ncols` are declared and initialized.

   ```
   long *list[nrows];

   list[0] = calloc(ncols * nrows, sizeof(long));
   for (size_t i = 1; i < nrows; i += 1) {
     list[i] = list[i - 1] + ncols; // Line D
   }
   list[0][0] = 0;                  // Line E
   free(list);                      // Line F
   ```

   Which of the following statements about the code above is correct?

   (i) The code would cause illegal memory access at Line D if `calloc` fails to allocate memory.

   (ii) The code would always cause illegal memory access at Line E because `list[0][0]` is not allocated on the heap.

   (iii) The code would cause a memory deallocation error when `list` is free at Line F.

   A. (i) only

   B. (ii) only

   C. (iii) only

   D. (i) and (iii) only

E. (ii) and (iii) only
F. (i) and (ii) only
G. (i), (ii), and (iii)

Write X in the answer box if none of the choices above is correct.

Solution: C.

This question checks if students understand memory accesses and know how to properly manages dynamic memory allocation for 2D arrays.

For (i), if `calloc` fails, then `list[0]` would be `NULL`. Line D would update `list[i]` to illegal memory addresses. But since the content of the memory addresses is never accessed on Line D, (i) is incorrect. Note that illegal memory access occurs on Line E if `calloc` fails.

(ii) is incorrect, because `list[0][0]` IS allocated on the heap.

(iii) is correct. Line F tries to free `list`, which is allocated on the stack. To correctly deallocate the memory, Line F should be `free(list[0])`.

9. (3 points) What is the running time of the function below in terms of n, the size of the input array?

```
void waldo(long a[], long n) {
  long k = n;
  while (k > 0) {
    for (long i = 0; i < n; i += 1) {
      a[i] += (k % 10);
    }
    k /= 10;
  }
}
```

   A. $O(10^n)$
   B. $O(n^{10})$
   C. $O(\log n)$
   D. $O(n)$
   E. $O(n \log n)$

   Write X in the answer box if none of the choices above is correct.

   Solution: E.

   This question assesses if students are able to analyze the big-O running time of a given code.

   There is a nested loop here. The inner `for` loop runs $O(n)$ times for each `k`. The outer `while` loop keeps dividing `k` by 10 until it reaches 0, so it runs $O(\log n)$ time. The total running time is thus $O(n \log n)$.

10. (3 points) The running time of a recursive algorithm can be characterized by the following recurrence relation:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + \log n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

   What is $T(n)$?

   A. $O(1)$
   B. $O(\log \log n)$
   C. $O(\log n)$
   D. $O(\log^2 n) = O(\log n \cdot \log n)$
   E. $O(n)$
   F. $O(n \log n)$

   Write X in the answer box if none of the choices above is correct.

Solution: D.

This question assesses if students are able to solve a recurrence relation. We have

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + \log n \\
&= T(\frac{n}{4}) + \log\frac{n}{2} + \log n \\
&= T(\frac{n}{8}) + \log\frac{n}{4} + \log\frac{n}{2} + \log n \\
&= T(\frac{n}{2^i}) + \log\frac{n}{2^{i-1}} + ... + \log\frac{n}{2} + \log n
\end{aligned}$$

When $\frac{n}{2^i}$ becomes 1, i is $\log n$. So we have

$$\begin{aligned}
T(n) &= T(1) + \sum_{k=0}^{\log n} \log\frac{n}{2^{k-1}} \\
T(n) &= 1 + \sum_{k=0}^{\log n} (\log n - \log 2^{k-1}) \\
&= \log n \log n - \sum_{k=0}^{\log n} (k-1)\log 2 \\
&= O(\log n \log n)
\end{aligned}$$

11. (3 points) Consider the following program:

```
struct store {
  bool x;
};

struct store muk(stuct store s, struct store *p) {
  struct store t = {
    .x = false;
  }
  *p = s;
  s = t;
  return s;
}

int main()
{
  struct store p;
  struct store r;
  struct store s;
  r = muk(s, &p);
  // Line G
}
```

What can we assert at Line G?

A. `s.x == false`

B. `p.x == false`

C. `r.x == s.x`

D. `p.x == r.x`

E. `p.x == s.x`

Write X in the answer box if none of the choices above is correct.

Solution: E.

This question tests if students are aware of how `struct` is initialized, assigned, passed into a function, and returned from a function. `s` is passed by value so it does not change. At Line G, `s` remains undefined. `p` is passed by value. So The line `*p = s` inside `muk` updates `p` in `main` to `s`.

Note that another assertion we could write at Line G is that `r.x == false`.

12. (3 points) Consider the following program:

```
void thud(long x, long y) {
  if (x < 0 || x > 2) {
    return;
  }
  if (y < 0 || y > 2) {
    return;
  }
  cs1010_print_long(x);
  cs1010_print_long(y);
  thud(x - 1, y);
  thud(x + 1, y);
  thud(x, y - 1);
  thud(x, y + 1);
}

int main()
{
  thud(1, 1);
}
```

Which of the following describes the behavior of the program when it is executed?

A. It prints `1101211011`.

B. It prints `110100102021221202`.

C. It prints `1111111111111111...` forever.

D. It prints `1101110111011101...` and then crashes with a stack overflow error.

Write X in the answer box if none of the choices above is correct.

Solution: D.

Question 12 assesses if students are able to trace through non-linear recursive code. We can trace through the calls as follows:

```
thud(1, 1) -> thud(0, 1) -> thud(-1, 1) -> return
                         -> thud(1, 1) -> thud(0, 1) -> ..
```

So we get into infinite recursion with `thud(1, 1)`. The code, therefore, prints `11011101..` until the stack overflows.

Part II

Short Questions (54 points)

Answer all questions in the space provided on the answer sheet. Be succinct and write neatly.

13. (12 points) **Loops**

For each of the functions below, state the mathematical expression that `foo` computes as a function of n and (if applicable) k. Make each mathematical expression as general as possible (i.e., covers as many values of n and k as possible). If there is a range of values for n and k in which the code does not work, state them in the answer.

Solution: Every expression is worth 2 marks. The range is 1 mark.

(a) (3 points) What does `foo` compute?

```
long foo(long n) {
  long i = 0;
  while (i <= n) {
    i += 1;
  }
  return i;
}
```

Solution: $n + 1$ with the range $n \geq -1$.
Alternatively, we accept $\max(0, n + 1)$, which works for all integer n.

(b) (3 points) What does `foo` compute?

```
long foo(long n, long k) {
  long i;
  for (i = 0; i < n; i += k) {
  }
  return i;
}
```

Solution: This computes the smallest positive multiple of $k \geq n$, or $\lceil \frac{n}{k} \rceil k$.
The code works for the range $k > 0, n \geq 0$. When $k \leq 0$, we get into an infinite loop, since i never increases. On the other hand, if $n < 0$, the code never enters the loop and returns 0.

(c) (3 points) What does `foo` compute?

```
long foo(long n, long k) {
  while (n > k) {
    n -= k;
  }
  return n;
}
```

Solution: $(n-1)\%k+1$

The code works for the range $k>0, n>0$.

When $k \leq 0$, we get into an infinite loop, since n never decreases. On the other hand, if $n<0$ (and $k>0$), the code never enters the loop and returns n. When n is 0 and k is 1, the function returns 0 but the expression evaluates to 1.

Alternatively, we also accept $n\%k$, which works for $k>0, n \geq 0$. Furthermore, n cannot be a multiple of k. When n is a multiple of k, k is returned.

(d) (3 points) What does `foo` compute?

```
long foo(long n, long k) {
  long i = 0;
  while (n > k) {
    n -= k;
    i += 1;
  }
  return i;
}
```

Solution: $(n-1)/k$.

The code works for the range $k>0, n>0$ for reasons similar to Part (c). The only difference is that it does not work for the case $n=0$. When n is 0, the function returns 0, but $(n-1)/k$ evaluates to -1.

14. (14 points) Consider the algorithm below:

```
void swap(long a[], size_t i, size_t j) {
  long temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

void bar(long a[], size_t n) {
  size_t i = 0;
  while (i < n - 1) {
    // Line P
    if (a[i] > a[i + 1]) {
      swap(a, i, i + 1);
      i = 0;
    } else {
      i += 1;
    }
    // Line Q
  }
}
```

(a) (6 points) Trace through what happens to the array `a[3] = {5, 1, 2};` if we call `bar(a, 3)`, by writing down the values of `i` and content of `a` at the end of each iteration (i.e., at Line Q).

Note that you may not need to fill in all the rows in the given table on the answer sheet.

Solution:

```
i=0, a=5 1 2
i=0, a=1 5 2
i=1, a=1 5 2
i=0, a=1 2 5
i=1, a=1 2 5
i=2, a=1 2 5
```

1 mark for each iteration. The result after the first iteration is already given so you get 1 mark for free!

(b) (4 points) The invariant of the loop is the following: " `a[0]..a[i]` is sorted in non-decreasing order." Assuming that this assertion holds at Line P, explain why it also holds at Line Q.

Solution: if `a[i] < a[i+1]`, then the sequence `a[0]..a[i+1]` is sorted. We increase `i` by 1. So the assertion `a[0]..a[i]` is sorted is true. Otherwise, `i` becomes 0. We have only one element that is trivially sorted.

For marking,

- 2 marks each for explaining why assertion holds for "if" and "else" branch.
- -1 mark if students did not explicitly show that the "if" branch, assertion holds because the new list contains just one element and is trivially sorted.
- -1 mark if students did not show that after i+=1 , the assertion reverts from a[0]...a[i+1] is sorted back to a[0]...a[i] .

Some common mistakes include (i) not explaining the general case, (ii) saying that because the assertion is true at line P, the "if" loop will never execute.

(c) (4 points) Express the running time of this algorithm using the big-O notation in terms of n. Briefly explain your answer.

Solution: $O(n^3)$. In the worst case, we have $O(n^2)$ pairs of elements not in order. Each scan $O(n)$ fixes one such pair.

To get full marks, students must give the right big O and provide the right explanation. If a student gives the wrong big O, at most 1 mark can be given if the student identifies the worst case correctly as being decreasing/non-increasing/inversely sorted. A common mistake is that most students say is $O(n^2)$, but don't forget this is not exactly like bubble sort because we reset i to 0 after a swap, so that incurs an extra $O(n)$ factor.

15. (18 points) Given (i) an integer `q` and (ii) an array of integers `a` with `n` elements, we wish to find if there is a combination of the elements in `a` that sums up to `q`. For instance, if the array is `{3, 11, -1, 9}`, we can find a combination of 9 and 11 that sums to 20; but we cannot find any combination that would sum to 6.

The following is a recursive boolean function that takes in an array `a`, the starting index `i`, its length `n`, and a query `q`. The function returns `true` if there is a combination of elements in `a[i]..a[n-1]` that sums to `q` and returns `false` otherwise.

```
bool can_sum_to(long a[], size_t i, size_t n, long q) {
  if (<condition1>) {
    return true;
  }
  if (<condition2>) {
    return false;
  }
  return <recursive call>;
}
```

You do not have to worry about (i) stack overflow or (ii) the efficiency of your program.

(a) (4 points) Complete the two `if` conditions, which are the base cases that check for inputs that are trivially `true` or `false`.

(b) (6 points) Complete the recursive call. Your answer should consist of a single return statement and appropriate use of short-circuiting to avoid unnecessary checks.

(c) (4 points) Let the running time of the function above be $T(n)$. Write down the recurrence relation for $T(n)$ and its base case $T(1)$.

(d) (4 points) Expression the worst case running time of the function `can_sum_to` using big O notation. Either show your workings or cite a similar analysis given in the lecture.

Solution: Here is one way this problem could be solved:

```
bool can_sum_to(long a[], size_t i, size_t n, long q) {
  if (a[i] == q) {
    return true;
  }
  if (i >= n - 1) {
    return false;
  }
  return can_sum_to(a, i + 1, n, q - a[i]) || can_sum_to(a, i + 1, n, q);
}
```

Common Mistakes

- Have the correct recursive calls but returning `true` if `q == 0`. This is incorrect since the function would return true even if there is no combination of elements in `a` that sums to 0.

- Only checks if two elements sum to `q`. This can be solved in $O(n^2)$ and can be solved with two loops. A common recursive solution given is incorrect and involves moving two pointers from left and right until they sum to `q`:

```
bool can_sum_to(long a[], size_t i, size_t n, long q) {
  if (a[i] + a[n - 1] == q) {
    return true;
  }
  if (i >= n - 1 || n <= 0) {
    return false;
  }
  return can_sum_to(a, i + 1, n, q) || can_sum_to(a, i, n - 1, q);
}
```

Another variation is to check if two consecutive elements sum to `q`:

```
bool can_sum_to(long a[], size_t i, size_t n, long q) {
  if (a[i] + a[i + 1] == q) {
    return true;
  }
  if (i == n - 2) {
    return false;
  }
  return can_sum_to(a, i + 1, n, q);
}
```

The running time of the solution can be written as the recurrence relation: $T(n) = 2T(n-1) + 1$ and $T(1) = 1$. This can be solved as $T(n) = O(2^n)$, similar to the Tower of Hanoi analysis.

Marking Scheme

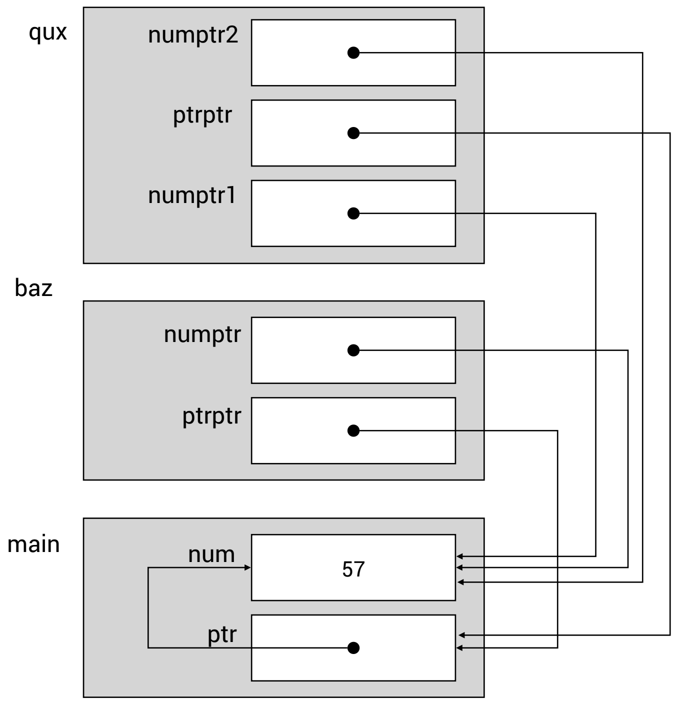
- (a) and (b) are marked together since (a) depends on (b). If the recursive call in (b) is wrong, no marks are given for (a).
- (c) and (d) are marked together. They depend on (a) and (b); so if (a) and (b) are completely wrong, no marks are given for (c) and (d). However, if (b) still shows some form of non-linear recursion with two calls to `can_sum_to`, we still consider the answer to (c) and (d).
- For (c), we deduct one mark if $T(1)$ is missing. We give two marks if $T(n) = 2^nT(1)$ is given instead.

16. (10 points) Consider the program below.

```
void qux(long **ptrptr, long *numptr2) {
  long *numptr1 = numptr2;
  *ptrptr = numptr2;
  *numptr1 = 57;
  // Line X
}

void baz(long **ptrptr, long *numptr) {
  qux(ptrptr, numptr);
}

int main()
{
  long *ptr;
  long num = 1;

  baz(&ptr, &num);
}
```

Draw the content of the call stack when the execution reaches Line X using the notations similar to what has been used in CS1010. Label all your stack frames, variables, and values on the call stack. You may use arrows to denote pointers, instead of using the actual memory addresses.

Solution:



Marking Scheme

We give 1 mark for each correct variable, except `ptr` in `main`, which is worth two marks.

The remaining 2 marks are allocated to drawing the correct stack – with `main` at the bottom and the rest stacked on top. If the order of the stack is reversed, we give one mark. If the stack frames are positioned arbitrarily (e.g., horizontally or artistically arranged), we give 0 marks.

END OF PAPER