NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 1 AY2020/2021

CS1010 Programming Methodology

November 2020 Time Allowed 120 Minutes

---

# INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 14 questions and comprises 18 printed pages, including this page.
2. The total marks for this assessment is 60. Answer **ALL** questions.
3. This is an **OPEN BOOK** assessment.
4. You can assume that in all the code given, no overflow nor underflow will occur during execution. In addition, you can assume that all given inputs are valid and fits within the specified variable type.
5. State any additional assumption that you make.

Note: This final is administered as an online e-exam due to COVID-19 pandemic. The questions have been reformatted and rephrased to fit into this document. The questions and comments are jointly written by Daren Ler and Ooi Wei Tsang.

1. (3 points) Consider the code fragment below:

```
long foo(long n) {
  long res = 1;
  for (long i = 1; i <= n; i += 1) {
    if (n % i == 0) {
      res *= n;
    }
  }
  return res;
}
```

Given that `n` is always a positive integer, which of the following functions (`goo`, `hoo`, `joo`, `koo`) are equivalent to `foo` above? Note that for this question, two functions are equivalent if they always return the same value for the same input `n`.

```
long goo(long n) {
  long res = n;
  long i = n;
  do {
    if (n % i == 0) {
      res *= n;
    }
    i -= 1;
  } while (i >= 1);
  return res;
}

long hoo(long n) {
  long res = 1;
  long i = n;
  do {
    if (n % i == 0) {
      res *= n;
    }
    i -= 1;
  } while (i > 1);
  return res;
}

long jj(long n, long i) {
  if (i == 1) {
    return n;
  }
  if (n % i == 0) {
    return jj(n * n, i - 1);
  }
  return jj(n, i - 1);
}

long joo(long n)
  return jj(n , n);
}

long kk(long n, long i) {
  if (i == 1) {
    return n;
```

```
  }
  if (n % i == 0) {
    return n * kk(n, i - 1);
  }
  return kk(n, i - 1);
}

long koo(long n) {
  return kk(n , n);
}
```

**Solution:** In `foo`, `res *= n` is executed every time `n` is divisible by `i`. So `res =` $n^k$, where $k$ is the number of factors of $n$.

In `goo`, `res *= n` is again executed every time `n` is divisible by `i`. But `res` is initialized to `n`, so it returns $n^{k+1}$.

`hoo` looks like `goo`, but `res` is initialized to 1 and the loop stops when `i` is 1. So `hoo` only computes $n^{k-1}$.

The function `jj` is called `n` times. The first parameter is squared every time it is a multiple of `i` (except for when `i` is 1).

The function `kk` is called `k` times. The return value of each call is multiplied by `n` each time `n` is divisible by `i` (except when `i` is 1). But the return value is initialized to `n`. So `koo` returns $n \times n^{k-1} = n^k$

`koo` is the only option that is equivalent to `foo`.

2. (3 points) Consider the running time of the three functions below using big O notation:

```
long bar(long n) {
  long k = 0;
  for (long i = 1; i < n*n; i += n) {
     k += i;
  }
  return k;
}

long cat(long n) {
   long k = 0;
   for (long i = n; i > 0; i /= 2) {
     k += i;
   }
   return k;
}

long dam(long n) {
   long k = 0;
   for (long i = 1; i < 1000; i += 1) {
     k += n;
   }
   return k;
}
```

Arrange the three functions, in increasing order of growth, in terms of their big O time complexity. A function with a higher order of growth should be listed later.

**Solution:** `bar` has the running time of $O(n)$; `cat` has the running time of $O(\log n)$; `dam` has the running time of $O(1)$.

So the correct ordering is `dam`, `cat`, `bar`.

3. (3 points) Consider the following program.

```
char *foo(char *x) {
  x = "cs1010";
  // Line A
  return x;
}

int main() {
  char *c = malloc(4);
  c = foo(c);
  // Line B
}
```

Which of the following statement(s) is/are correct?

(Select all correct options)

A. At Line A, the pointer `x` points to a memory location on the stack.

B. At Line A, the pointer `x` points to a memory location on the heap.

C. At Line A, the pointer `c` points to a memory location on the heap.

D. At Line B, the pointer `c` points to a memory location on the stack.

E. At Line B, the pointer `c` is pointing to a memory location on the heap.

**Solution:** Very few students got this correct.

At Line A, `c` is pointing to some location on the heap (it is "malloc"-ed). `x` is pointing to some location in the read-only memory region (data region). At Line B, `x` does not exist anymore, `c` is now pointing to where `x` pointed to (which is the data region).

So the only correct option is C.

4. (3 points) Consider the function below:

```
void qux() {
  char c;
  while (c > 0) {
    c = 1;
  }
}
```

Which of the following statements about the function above is true?

(Select all correct options)

A. The code causes a compilation error because `==` can only be used with integer types.

B. The code causes a compilation error because `c` has not been initialised.

C. The code causes a compilation error because we did not cast 0 and 1 to `char`.

D. The code may lead to an infinite loop.

E. The code will always lead to an infinite loop.

**Solution:** Since `c` is uninitialized, it might be positive or negative. If `c` is positive, then the function will enter an infinite loop. The correct option is D.

42% of students got this correct.

5. (3 points) Consider the following:

```
#define distance_square(a, b) (a*a + b*b)
```

What is the value of `x` after executing

```
long a = 1;
long b = 2;
long x = distance_square(a + 2, b + 3);
```

**Solution:**

```
long x = a + 2*a + 2 + b + 3*b + 3
       = 3a + 4b + 5
       = 3 + 8 + 5
       = 16
```

Most students got this correct.

6. (3 points) What is printed by the following program?

```
#include "cs1010.h"

long foo(long a, long *b, long **c) {
  **c += 1;
  *b += 2;
  a += 4;
  return a + *b + **c;
}

int main() {
   long p;
   long *q;
   long **r;
   p = 0;
   q = &p;
   r = &q;
   cs1010_println_long(foo(p, q, r));
}
```

**Solution:** Draw the stack diagram to understand what happens here: both `*b` and `**c` refer to the same memory location, while `a` is a copy. So after

```
**c += 1;
*b += 2;
```

The variable `p` has the value 3. `a` is passed by value so its value is 4.

```
a += 4;
```

Summing them up gives the answer 10.

Less than half of the students got this correct.

7. (3 points) What is the time complexity of the code below?

```
#include "cs1010.h"

long f(long n) {
  if (n < 2) {
    return 1;
  }
  for (long i = 0; i < n; i += 1) {
    cs1010_println_string("hello");
  }
  return f(n/3) + f(n/3) + f(n/3);
}
```

**Solution:** The recurrence relation for the running time of `f` is

$$\begin{aligned} T(n) &= 3T(n/3) + n \\ &= 3(3T(n/9) + n/3) + n \\ &= 9T(n/9) + n + n \\ &= 27T(n/27) + n + n + n \\ &\vdots \\ &= 3^k T(n/3^k) + kn \end{aligned}$$

When $k$ is $\log n$, we get $T(n) = O(n) + O(n \log n) = O(n \log n)$

Only slightly more than 1/3 of students got this correct.

8. (3 points) What is the output of the program below?

```
#include "cs1010.h"

void f(long n, long a) {
  if (n <= 0) {
    return;
  }
  f(n - 1, a + 1);
  cs1010_println_long(a);
  f(n - 1, a - 1);
}

int main() {
  f(3, 4);
}
```

**Solution:** The output should be

```
6
5
4
4
4
3
2
```

Nothing fancy here – just testing if students can keep track of the stack when tracing recursive calls.

Most students could do it.

9. (3 points) Consider the program below:

```
int main() {
  long *p = bar();
  cs1010_println_long(*p);
}
```

Which implementation of function `bar` below would lead to illegal memory access when the program is executed? Assume that `malloc` does not return `NULL`.

```
// A.
long *bar() {
   long x = 10;
   return &x;
}

// B.
long *bar() {
   long *px;
   *px = 10;
   return px;
}

// C.
long *bar() {
   long *px;
   px = (long *)malloc(1);
   *px = 10;
   return px;
}

// D.
long *bar() {
   return (long *)10;
}
```

**Solution:** A returns an address on the stack. This would lead to an illegal memory access.

For B, `px` is not initialized when we deference it. This would lead to an illegal memory access.

C allocates something on the heap and returns the memory address, but it does not allocate enough for a `long` variable. We only allocated one byte.

D returns the address 10. We don't have access to the memory address 10.

So all options would lead to illegal memory access.

10. (3 points) Consider the following (incomplete) function, which merges two sorted arrays `a1` and `a2` of size `len1` and `len2` respectively into the output array out of size `len1 + len2`. The arrays `a1` and `a2` may contain duplicate values and are sorted in non-decreasing order.

```
void merge(const long a1[], const long a2[], long out[], long len1, long len2) {
  long i = 0;
  long j = 0;
  long k = 0;
  while (i < len1 && j < len2) {
    if (a1[i] < a2[j]) {
      out[k] = a1[i];
      i += 1;
    } else {
      out[k] = a2[j];
      j += 1;
    }
    k += 1;
  }
  // Line C
}
```

Which of the following assertions must be true at Line C? (Select all correct answers)

A. `i != len1 || j < len2`

B. `i != len1 || k == len1 + j - 1`

C. `i != len1 || a1[len1 - 1] < a2[j]`

D. `i != len1 || out[k] == a1[len1 - 1]`

**Solution:** When we exit the loop, the negation of the loop condition must be true: `i >= len1 || j >= len2`.

This is not an available option, however, and so we have to look at the choices more carefully.

All the options have `i != len1` as part of the expression. So if `i != len1`, then all the given assertions are true. So it comes down to what must be true if `i == len1` when we exit the loop.

If `i == len1`, then the most recent path of execution must be `out[k] = a1[i]; i += 1`; when `i == len-1`. This means that `j < len2`. So `i != len1 || j < len2` is true. This branch happens when `(a1[i] < a2[j])`, which means `a1[len-1] < a2[j]` must be true.

When this happens, we have copied all `len1` items from `a1` over, and `j` items from `a2` over. So `k` must be `len1 + j`. The second option is wrong. Furthermore, we set `out[k]` to `a1[len-1]`, and increment `k`. So `out[k-1] == a1[len1-1]` must be true.

11. (3 points) Consider the following code snippet, which defines a structure called `box` and four functions, `titi`, `tutu`, `tata`, and `toto`.

```
struct box {
  long x;
};

void titi(struct box b) {
  b.x = 1;
}

void tutu(struct box *b) {
  b->x = 1;
}

struct box tata() {
  struct box b;
  b.x = 1;
  return b;
}

void toto(long *x) {
  *x = 1;
}
```

Which of the following correctly initialise the field `x` in `b` to 1? (Select all correct options)

```
// A.
struct box b;
titi(b);

// B.
struct box b;
tutu(&b);

// C.
struct box b = tata();

// D.
struct box b;
toto(&(b.x));
```

**Solution:** In Option A, a `struct` variable is passed by value. So `titi` does not change `b.x`. To do so, we need to pass `b` by reference, as shown in Option B.

Alternatively, we can return a new `struct` and update `b`, as shown in Option C.

We could also pass `b.x` by reference as in Option D.

12. (3 points) What is the output of this program?

```
#include "cs1010.h"

char* quack(char *input, int i, int j) {
  char* output = input;
  output[j] = output[i];
  output[i] = input[j];
  return output;
}

int main() {
  char input[] = "01234567";
  char* output = quack (
    quack (
      quack (
        quack (input, 3, 4),
      2, 5),
    1, 6),
  0, 7);
  cs1010_println_string(output);
}
```

**Solution:** First thing to realize is that `input` and `output` are pointing to the same arrays. So the lines

```
  output[j] = output[i];
  output[i] = input[j];
```

are the same as

```
  output[j] = output[i];
```

Since the input is:

```
  char input[] = "01234567";
```

The call

```
  quack (input, 3, 4),
```

copied `input[3]` to `input[4]`. So we get `01233567`. If we continue on, copying `input[2]` to `input[5]`, etc. We get `01233210` at the end.

13. (18 points) Study the function `moo` below.

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

void moo(long a[], long len) {
  long i = 0;
  while (i < len) {
    if (i == 0 || a[i] >= a[i - 1]) {
      i += 1;
    } else {
      swap(a, i, i-1);
      i -= 1;
    }
    // Line Z
  }
}
```

(a) (4 points) Suppose we call `moo` with the array `a` below as input.

```
long a[3] = {9, 8, 7};
moo(a, 3);
```

Show the value of `i` and the content of the array in the first six iterations of the while loop in `moo` at Line Z. The answers for the 1st and 6th iterations are already given.

**Solution:**

| `i` | `a` |
|---|---|
| 0 | `{8, 9, 7}` |
| 1 | `{8, 9, 7}` |
| 2 | `{8, 9, 7}` |
| 1 | `{8, 7, 9}` |

(b) (1 point) What is the content of array `a` when `moo` returns?

**Solution:** `{7, 8, 9}`

(c) (4 points) The function `moo` above is reproduced below with additional commented lines.

```
void moo(long a[], long len) {
  long i = 0;
  while (i < len) {
    if (i == 0 || a[i] >= a[i - 1]) {
      // Example: { a[i] >= a[i - 1] }
      i += 1;
      // Line V
    } else {
      // Line W
      swap(a, i, i-1);
      // Line X
      i -= 1;
      // Line Y
    }
```

```
  }
}
```

Write down the assertions that must be true at Line V, W, X, Y. Your assertion should relate either `a[i]` or `a[i-1]` to one of its adjacent elements in the array (i.e., relate `a[i]` to either `a[i-1]` or `a[i+1]`, or relate `a[i-1]` to either `a[i]` or `a[i-2]`).

For example, the assertion on Line 5 above is `a[i] >= a[i-1]`.

**Solution:** Line V: `a[i - 1] >= a[i - 2]`

Line W: `a[i] < a[i - 1]`

Line X: `a[i] > a[i - 1]`

Line Y: `a[i + 1] > a[i]`

(d) (3 points) The function `moo` above is reproduced below with comments Line P and Line Q.

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

The loop invariant for the while loop in `moo` is as follows:

The elements in `a[0] .. a[i-1]` (if any) are sorted in non-decreasing order.

Explain why, if this invariant is true at Line P, it will also be true at Line Q.

**Solution:** If either `i` is 0 or `a[i-1]` and `a[i]` is not an inversion, then we increase `i`, so `a[i-1] > a[i-2]`, and we assume `a[i-2]` to `a[0]` is already sorted.

Otherwise, `a[i-1]` and `a[i]` is an inversion. We decrease `i`. Since we assume `a[i-1]` to `a[0]` is sorted, `a[i-2]` to `a[0]` is also sorted. After decreasing `i`, `a[i-1]` to `a[0]` is still sorted.

(e) (2 points) Suppose the input array is already sorted in non-decreasing order, what is the running time of the function `moo`? Express your answer using Big O notation in its simplest form in terms of $n$.

**Solution:** We always increase `i` in this case. So running time is just $O(n)$.

(f) (4 points) Suppose the input array is inversely sorted in decreasing order, what is the running time of the function `moo`? Express your answer using Big O notation in its simplest form in terms of `n`, the number of elements in the input array, and explain how you derived your answer.

**Solution:** $O(n^2)$. For every pair of adjacent elements at position $(k, k+1)$ in the wrong order, we decrease `i`, `k` times. Since it is inversely sorted, we sum `k` from 1 to $O(n)$ which gives $O(n^2)$.

14. (6 points) Uncle Tan is selling masks in his shop. He packages the masks using three different types of packaging: A single pack contains one mask only. A double pack contains two masks. A triple pack contains three masks.

    At the end of the day, Uncle Tan looks back at how many masks he has sold and ponders about how many possible ways he could have sold that number of masks.

    For instance, if he sold three masks, there are four different ways:

    - he could have sold three single packs (1 + 1 + 1)
    - or a single pack, followed by a double pack (1 + 2)
    - or a double pack, followed by a single pack (2 + 1)
    - or a triple pack (3)

    If he sold four masks, there are 7 ways:

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

    Uncle Tan wants to write a recursive function to help him count how many ways he can sell $n$ masks. It takes $n$ as input and returns the number of ways to sell $n$ masks. Help him complete the function by filling in the body of the function (you do not have to write the function header again in your answer). A non-recursive function will receive 0 marks. Since Uncle Tan does not sell many masks each day, you do not have to worry about (i) stack overflow, (ii) integer overflow, and (iii) the efficiency of your program.

    ```
    long masks(long n) {

    }
    ```

    **Solution:**

    ```
    long masks(long n) {
      if (n == 1) {
        return 1;
      }
      if (n == 2) {
        return 2;
      }
      if (n == 3) {
        return 4;
      }
      return masks(n - 1) + masks(n - 2) + masks(n - 3);
    }
    ```

END OF PAPER