National University of Singapore

TIC1002—Introduction to Computing and Programming II

Semester 2, 2017/2018

Time allowed: 2 hours

---

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **SIX (6) questions** and comprises **TWENTY (20) pages** including this cover page.
3. Weightage of questions is given in square brackets. The maximum attainable score is 100.
4. This is a **OPEN** book assessment. However, no electronic devices are allowed.
5. Five additional minutes of reading time will be given before the start of the assessment. You may read the paper but are not allowed to write anything during this time.
6. Write all your answers in the space provided in this booklet.
7. You are allowed to write with pencils, as long as it is legible.
8. **Please write your student number below.**

STUDENT NO:	S	O	L	U	T	I	O	N	S

(This portion is for the examiner's use only)

Question	Marks	Remarks
Q1	/ 20	
Q2	/ 15	
Q3	/ 12	
Q4	/ 12	
Q5	/ 24	
Q6	/ 17	
Total	/ 100	

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: C Expressions [20 marks]

There are several parts to this question which are to be answered independently and separately. Each part consists of a fragment of C code. Write the **exact output** produced by the code in **the answer box**. If an error occurs, or it enters an infinite loop, state and explain why.

You may show workings **outside the answer box** in the space beside the code. Partial marks may be awarded for workings if the final answer is wrong.

Assume that all appropriate preprocessor directives e.g., `#include <stdio.h>`, etc. have already been defined.

A. [5 marks]

```
char s[11] = "unpuzzling";
for (int i = 0; i < 10; i++) {
    switch (s[i]) {
    case 'u':
        i++;
    case 'z':
        s[i-1] = s[i+1];
        break;
    case 'n':
        s[i] = s[i+1];
    }
}
printf("%s", s);
```

pnpzlzligg

This question tests understanding of switch statements and fall-through of case.

B. [5 marks]

```
int a[11] = {0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0};
for (int i = 0; i < 10; i++) {
    int x = i;
    for (int k = i; k < 10; k++) {
        if (a[k] > a[x]) {
            x = k;
        }
    }
    printf("%d", a[x]);
}
```

55555543210

This question is basically selection sort. It tests if student can recognise algorithms in code.

C. [5 marks]

```
typedef struct a {
    int x;
    int y;
    char s[10];
    struct a *a;
} a;

a x = {2, 5, "Carpe ", NULL};
a y = {1, 3, "Diem!", &x};
x.a = &y;
printf("%c,%c,%c",
       y.s[x.a->a->x],
       x.s[y.a->y],
       x.a->s[y.a->a->x]);
```

`e, ,i`

This question tests pointers in structs.

D. [5 marks]

```
int f(int x, int y) {
    if (x % y) {
        return x+y;
    } else if (x > y) {
        return f(x-y, y*2) + x;
    } else {
        return f(x+1, y-x) + y;
    }
}

int main() {
    int x = 7;
    int y = 2;
    printf("%d", f(x, y));
}
```

`9`

This question tests if students recognize the true/false value of integers.

Question 2: Arithmetic Progression Sum [15 marks]

In the course, we discussed a simple example of *Arithmetic Progression* (AP) where the successive item differs by 1, e.g. 1, 2, 3, ..., 49, 50. The progression can be generalized to start from any number a_1 with a difference ("skip"), d, of any positive number, e.g. 3, 5, 7, 9 is a AP with $a_1 = 3$ and a skip of $d = 2$ between each term.

The nth term of an AP can be obtained from the formula $a_n = a_1 + (n-1)d$. Given the number of terms in the progression, we can easily find out the last term. For example, $a_4 = 3 + (4-1) \times 2 = 3 + 6 = 9$.

Similarly, we can generalized the sum of Arithmetic Progression as: $S = \sum_{i=1}^{n} a_n = \frac{n(a_1+a_n)}{2}$. For example, the sum of $3,5,7,9 = \frac{4(3+9)}{2} = 24$.

Suppose we have written a function `void print_ap_table(int start, int row, int col);` to print out a **Table of AP sum**. The row number (1, ..., `row`) represents the *skip*, i.e. row 1 has skip of 1, row 2 has skip of 2 etc. The column number (1, ..., `col`) indicates the number of terms in the AP, i.e. column 1 is the sum of 1 term in the AP, column 2 is the sum of 2 terms in the AP, etc.

Below is a sample output for `print_ap_table(3, 4, 5)`:

	1	*2*	*3*	*4*	*5*
1	3	7	12	18	25
2	3	8	15	24	35
3	3	9	18	30	45
4	3	10	21	36	55

A few examples to explain the output:

- Row 1, column 4: AP is $3,4,5,6$ as the start term is 3, skip is 1, number of terms is 4. The sum is $3+4+5+6 = 18$.
- Row 3, column 5: AP is $3,6,9,12,15$, sum is 45.
- Row 4, column 2: AP is $3,7$, sum is 10.

The function `print_ap_table()` is mostly completed as follows:

```
1  void print_ap_table(int start, int num_row, int num_col)
2  {
3      for (int row = 1; row <= num_row; row++) {
4          for (int col = 1; col <= num_col; col++) {
5              printf("%d\t", ap_sum(....)); //Point A
6          }
7          printf("\n");
8      }
9
10 }
```

The only missing piece of information is the function `int ap_sum(int start, int terms, int skip);` which takes:

- `start`: The first term, i.e. a_1
- `terms`: The number of terms in the AP.
- `skip`: The difference between successive term in the AP.

This function then returns the sum of the specified AP.

A. Complete the function call to `ap_sum()` at *Point A* by filling in the appropriate arguments. [2 marks]

```
ap_sum( start, col, row );
```

B. Give an **iterative** implementation of the `ap_sum()` function: [4 marks]

```
int ap_sum( int start, int terms, int skip)
{
    int i, cur_term, sum = 0;

    for (cur_term = start, i = 0; i < terms; i++, cur_term += skip) {
        sum += cur_term;
    }
    return sum;
}
```

C. Give an **recursive** implementation of the `ap_sum()` function: [4 marks]

```
int ap_sum( int start, int terms, int skip)
{
    if (terms == 1) {
        return start;
    }

    return start + ap_sum(start+skip, terms-1, skip);
}
```

D. Give an implementation of the `ap_sum()` function with **better time complexity** than the iterative version: [4 marks]

```
int ap_sum( int start, int terms, int skip)
{
    int i, last, sum = 0;

    last = start + (terms -1)*skip;
    sum = terms / 2.0  * (start+last);

    return sum;
}
```

E. Time complexity for the above code is: [1 mark]

$O(1)$

Question 3: Sorting Algorithm Properties [12 marks]

For each of the following questions, you are given an original array of structures and a **working** sorting algorithm with certain property. Give the resultant array after the sorting algorithm is applied on the original array. If there are multiple answers, you should choose one that **demonstrates** the sorting property of the algorithm. For example, if the algorithm is **unstable**, you should give an answer that highlight the "instability".

A. A **stable** sorting algorithm that sorts according to the **number**. [3 marks]

Input:

{9, 'Z'}	{5, 'W'}	{1, 'Z'}	{3, 'L'}	{5, 'Z'}	{8, 'E'}	{5, 'A'}

Output:

{1, 'Z'}	{3, 'L'}	{5, 'W'}	{5, 'Z'}	{5, 'A'}	{8, 'E'}	{9, 'Z'}

B. A **stable** sorting algorithm that sorts according to the **character**. [3 marks]

Input:

{9, 'Z'}	{5, 'W'}	{1, 'Z'}	{3, 'L'}	{5, 'Z'}	{8, 'E'}	{5, 'A'}

Output:

{5 ,'A'}	{8, 'E'}	{3, 'L'}	{5, 'W'}	{9, 'Z'}	{1, 'Z'}	{5, 'Z'}

C. A **unstable** sorting algorithm that sorts according to the **number**. [3 marks]

Input:

{9, 'Z'}	{5, 'W'}	{1, 'Z'}	{3, 'L'}	{5, 'Z'}	{8, 'E'}	{5, 'A'}

Output:

{1, 'Z'}	{3, 'L'}	{5, 'Z'}	{5, 'W'}	{5, 'A'}	{8, 'E'}	{9, 'Z'}

D. A **unstable** sorting algorithm that sorts according to the **character**. [3 marks]

Input:

{9, 'Z'}	{5, 'W'}	{1, 'Z'}	{3, 'L'}	{5, 'Z'}	{8, 'E'}	{5, 'A'}

Output:

{5 ,'A'}	{8, 'E'}	{3, 'L'}	{5, 'W'}	{1, 'Z'}	{9, 'Z'}	{5, 'Z'}

Question 4: Binary Search with Duplicates [12 marks]

The **binary search** implementation discussed in this course does not handle **duplicate** values in the sorted array. We will look at a simple way to handle such cases.

Firstly, let us modify the the binary search function so that it returns two values (startIndex, endIndex) to indicate the starting and ending indices of the target value. If the target value cannot be found, then both indices are set to -1.

Since C/C++ does not allow more than one return value, we will use two **pass-by-pointer** parameters instead. Below is a partially completed implementation:

```
1  void binarySearch( int a[], int N, int X, int *startIdx, int *endIdx) {
2
3     int mid, low = 0, high = N-1;
4     *startIdx = -1;
5     *endIdx = -1;
6
7     while ( (low <= high) && (*startIdx == -1) ) {
8        mid = (low + high) / 2;
9        if ( a[mid] == X ) {
10           *startIdx = mid;  // Modified: partially completed
11           *endIdx = mid;    //
12       } else if ( a[mid] < X ) {
13           low = mid + 1;
14       } else {
15           high = mid - 1;
16       }
17    }
18 }
```

A. Give the indices `startIdx` and `endIdx` for the following test case based on the partially completed code above:

`a[]`	`N`	`X`	`startIdx` = ?	`endIdx` = ?
1, 3, 5, 7, 9, 9, 9, 11	8	9	5	5

[2 marks]

B. Give the indices `startIdx` and `endIdx` for the following test case based on the partially completed code above:

`a[]`	`N`	`X`	`startIdx` = ?	`endIdx` = ?
1, 3, 5, 7, 9, 9, 9, 11	8	6	-1	-1

[2 marks]

C. Provide an implementation to give the correct `startIdx` and `endIdx`. You are only allowed to add code at the indicated location below: [6 marks]

```
void binarySearch( int a[], int N, int X, int* startIdx, int* endIdx)
{
  int mid, low = 0, high = N-1;
  *startIdx = -1;
  *endIdx = -1;

  while ( (low <= high) && (*startIdx == -1) ) {
    mid = (low + high) / 2;
    if ( a[mid] == X ) {
```

```
// fill in your code here
 *startIdx = mid;
 while(*startIdx > 0 && a[*startIdx - 1] == X){
      (*startIdx)--;
 }
 *endIdx = mid;
 while(*endIdx < N-1 && a[*endIdx + 1] == X){
      (*endIdx)++;
 }
```

```
  } else if ( a[mid] < X ) {
    low = mid + 1;
  } else {
    high = mid - 1;
  }
}
```

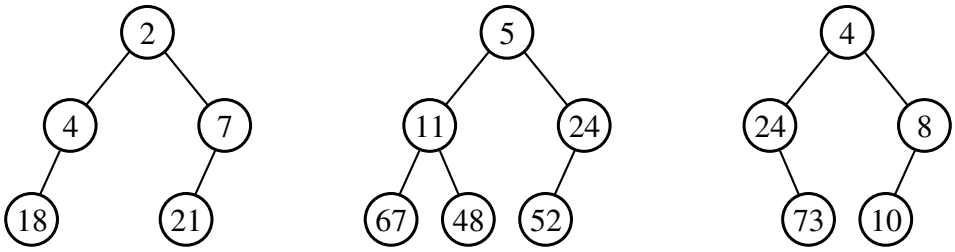
D. What is the worst case time complexity of this new implementation? [2 marks]

$O(N)$

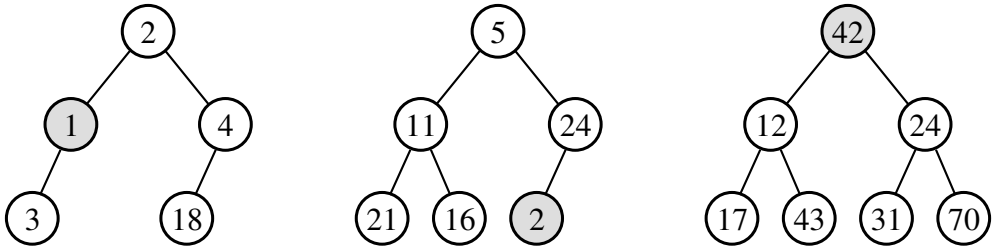
Question 5: Heaps [24 marks]

For the purpose of this question, a heap is a binary tree that satisfies this property: **the value of a node is smaller or equal to the values of all its children**. (More specifically this is a *minimum binary heap*. It is slightly different from the binary search tree we discussed in class.)

Some examples of valid heaps:



Here are some examples of invalid heaps, with the offending node highlighted:



A node in the heap is implemented as a `struct` as follow:

```
typedef struct node {
    int value;           // contains the value of the node
    struct node *left;   // the left child, NULL if none
    struct node *right;  // the right child, NULL if none
    struct node *parent; // the parent, NULL if none, i.e. the root
} Node;
```

A. Given the properties of this heap ADT, suggest a scenario where this ADT would be useful. [2 marks]

Since the minimum node will always be at the root, it can be used in situations where we always want direct access to the node of the minimum value.

One direct application would be a priority queue, where the minimum node will always be dequeued. For example, a queueing system where the customer with the lowest queue number will always be chosen next.

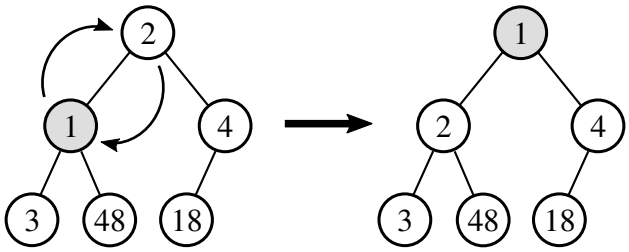
B. Write a function `valid_heap` which takes as input the root node of a heap, and returns `true` if the heap is valid (according to the property stated), and `false` otherwise. [6 marks]

```
bool valid_heap(Node *root) {
    if (!root)    // Reach the bottom
        return true;
    // check left child
    if (root->left && root->left->value > root->value)
        return false;
    // check right child
    if (root->right && root->right->value > root->value)
        return false;
    // Both child must also be valid
    return valid_heap(root->left) && valid_heap(root->right);




}
```

C. It is useful to be able to swap nodes in a heap. For example, we can swap the nodes in an invalid heap to make it valid like this:



Write a function `swap_nodes` that takes two nodes as input and swap them.
Hint: The structure of the heap is preserved after swapping. [4 marks]

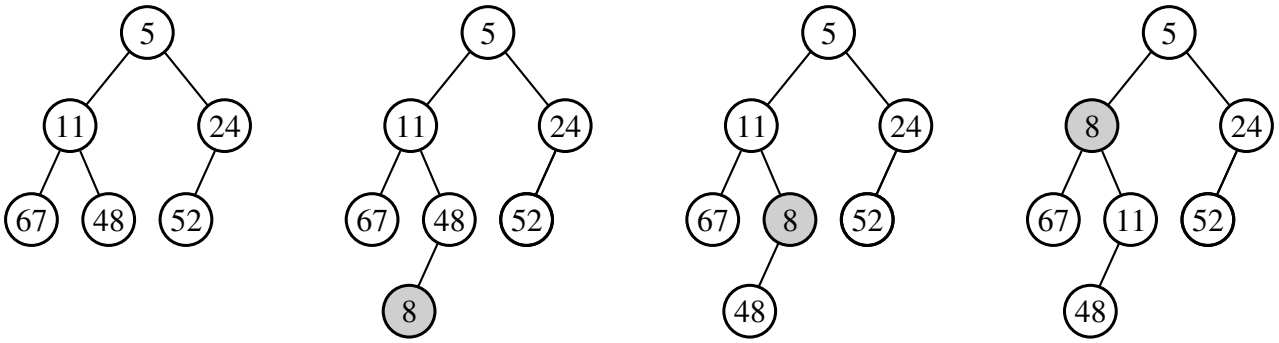
```
void swap_nodes(Node *n1, Node *n2) {
    int temp = n1->value;
    n1->value = n2->value;
    n2->value = temp;
}
```

The key idea is since the structure is preserved, there is no need to swap the nodes. We can just swap the values.

Otherwise, it is very troublesome to rework the structure.

D. To add a new value into a heap, we first create a new node and attach it as a new left to the bottom of the heap. Next, to make it a valid heap, we compare the value of the new node with its parent. If the new node is smaller than the parent, we swap them. This "bubbling up" continues until the new node finds a place which makes the heap valid.

Example:



Original heap | 8 added to the bottom | Swap with parent | Swap again. Done!

Implement the function `bubble_up` which takes as input, the **newly added node** to a valid heap, and performs the swapping needed to make the heap valid. Assume that the new node has been attached to the bottom of the heap.

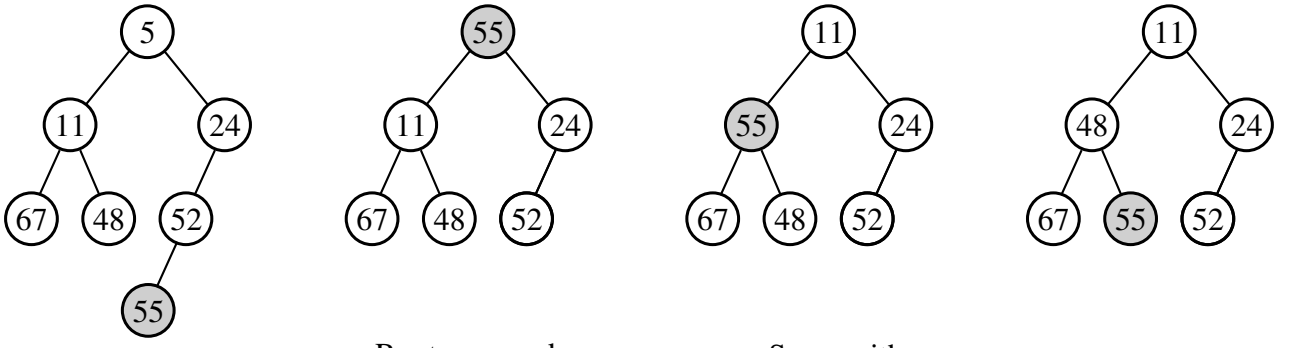
You may reuse the function `swap` that you defined earlier. [6 marks]

```
void bubble_up(Node *n) {
    // Note order of the condition is important
    // Must make sure there is a parent before we can compare with it
    while (n->parent && n->parent->value > n->value) {
        swap(n, n->parent);  // only swap the values
        n = n->parent;       // assign to parent
    }
}
```

E. Because of the property of the heap, the root will always be the smallest value found in the heap. When removing the root, we perform the following strategy:

First, a bottom node is moved to replace the root that was extracted. Next this node is compared with its children, and swapped with the smallest child (this ensures that after swapping the new parent will be smaller than both children.) This "bubble down" process continues until no more swaps take place, which means the heap is now valid.

Example:



Original heap | Root removed, replaced with 55 | Swap with smallest child | Swap again. Done!

Suppose there is a function `Node * min_node(Node *n1, Node *n2, Node *n3)` which takes in three nodes pointers and return a pointer to the node with the smallest value. Suppose `min_node` will also ignore pointers with the value `NULL`.

Implement the function `bubble_down`, which takes as input the newly swapped in root of a heap after the old root is extracted, and perform the "bubble down" process to make the heap valid. You may reuse the function `swap` that you defined earlier. [6 marks]

```
void bubble_down(Node *n) {
    Node *min;
    while ((min = min_node(n, n->left, n->right)) != n) {
        swap(n, min);
        n = min;
    }
}

// Using recursion
void bubble_down(Node *n) {
    Node *min = min_node(n, n->left, n->right);
    if (n != min) {
        swap(n, min);
        bubble_down(min);
    }
}
```

Question 6: The Last Jedi [17 marks]

For this question, we will be working with C++. Consider the following code:

```cpp
#include <iostream>
#include <string>
#include <set>
using namespace std;

class ForceUser {
private:
    string name;
protected:
    set<string> _powers;
public:
    ForceUser(string name) { this->name = name; };

    virtual string get_name() { return name; };

    void activate(string power) {
        if (_powers.count(power) == 0)
            cout << get_name() << " does not know force " << power << endl;
        else
            cout << get_name() << " performs force " << power << endl;
    }
};  // ForceUser


class Jedi : public ForceUser {
public:
    Jedi(string name) : ForceUser(name) {
        _powers = {"jump", "heal", "mind trick"};
    }
};  // Jedi


class Sith : public ForceUser {
public:
    Sith(string name) : ForceUser(name) {
        _powers = {"jump", "lightning", "choke"};
    }

    string get_name() {
        return "Darth " + ForceUser::get_name();
    }
};  // Sith
```

A. What will be the output when following statements are compiled and run? [2 marks]

```
Jedi rei("Rei");
rei.activate("jump");

Sith vader("Vader");
vader.activate("mind trick");
```

```
Rei performs force jump
Darth Vader does not know force mind trick
```

This question tests simple tracing of OOP inheritance and overriding.

B. Explain what the `virtual` keyword at line 15 does, and state how it will affect the output of part A if the `virtual` keyword was removed. [3 marks]

The virtual keyword states that the member function will be redefined in the subclass. It delays the binding of the function call to runtime instead of compile time.

Without `virtual` the line `vader.activate("mind trick");` will output "`Vader does not know force mind trick`" because `activate` will call `ForceUser::get_name()`.

C. Suppose Rei learns a new power. Since `rei._powers` is a set, we can use its method `insert` to append a new item into the vector by writing the following code:

```
rei._powers.insert("lift");
```

However, the code fails to compile with an error: Apparently, we can fix it by defining a new public method in the `Jedi` class between lines 30 and 31 as follow:

```
void add_power(string power) {
    _powers.insert(power);
}
```

Explain the reason for the problem and why it now works with this new addition. [2 marks]

It is because `protected` properties cannot be accessed from outside the class or subclasses, but they can be accessed from within the class.

Rei tries out this method by doing this:

```
ForceUser luke = Jedi("Luke");
luke.add_power("lift");.
```

Explain why does this fail even after the new method is added? [2 marks]

This new method is added only to Jedi, and not ForceUser. The variable `luke` was declared as a `ForceUser` and thus it does not have the method bound.

Explain the implications of adding this line to the `ForceUser` class (at line 16):

```
virtual void add_power(string power) = 0;
```

[2 marks]

This will make ForceUser and Sith abstract classes, and Sith can no longer be initialized, i.e. we will not be able to create Vader.

D. It turns out that Siths are known by their alias rather than their real names. We can modify the constructor to take in a name and alias, and have their real names shown only when passing a `true` into `get_name`, like so:

```
Sith emperor("Sidious", "Palpatine");
emperor.activate("choke");
cout << emperor.get_name() << endl;
cout << emperor.get_name(true) << endl;
```

produces the output:

```
Darth Sidious performs force choke
Darth Sidious
Palpatine
```

State the modifications that has to be done solely to class `Sith` to support this change. You do not need to rewrite all the code, just provide snippets of the modifications. [6 marks]

A new property has to be declared in `Sith` to store the real name:

```
private:
    string real_name;
```

The constructor will have to be modified as such:

```
Sith(string name, string alias) : ForceUser (alias) {
    powers = {"jump", "lightning", "choke"};
    real_name = name;
}
```

The member function `get_name` has to be overloaded:

```
string get_name(bool real) {
    if (real) {
        return real_name;
    } else {
        return ForceUser::get_name(); // Cannot simply return name
                                      // because it is private
    }
}
```

This question tests the ability to implement overloaded functions.

— END OF QUESTIONS —

Scratch Paper

Scratch Paper

— E N D O F P A P E R —