National University of Singapore

TIC1002—Introduction to Computing and Programming II

Semester 2, 2018/2019

Time allowed: 2 hours

---

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **FOUR (4) questions** and comprises **TWELVE (12) pages** including this cover page.
3. Weightage of questions is given in square brackets. The maximum attainable score is 60.
4. This is an **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	/ 15	
Q2	/ 15	
Q3	/ 24	
Q4	/ 6	
Total	/ 60	

Question 1: Gravity is not optional! [15 marks]

Many 2D puzzles contain some kind of "blocks" falling towards the "ground". Let us consider the logic for such games in this question. Below is an illustration of two game states:

	0	*1*	*2*	*3*	*4*
0	@	%	S	N	#
1	@	.	S	N	#
2	.	*	.	.	#
3	.	.	.	X	$
4	.	.	T	.	8
5	.	$	T	.	8

$\Longrightarrow$

	0	*1*	*2*	*3*	*4*
0	.	.	.	.	#
1	.	.	.	.	#
2	.	.	S	.	#
3	.	%	S	N	$
4	@	*	T	N	8
5	@	$	T	X	8

A few basic elements of the game:

1. Each cell can be empty (represented as a period '**.**') or some blocks (represented as alphabets, digits and symbols).
2. The last row (row 5 in the examples) represents the ground.
3. Blocks floating in the air (see grid on the left) will fall to the ground once gravity is applied (see grid on the right).

Instead of solving the problem in one go, let us break down the problem into **simpler steps**. Below are some basic declarations:

```
#define NCOL  5    //number of columns
#define NROW  6    //number of rows
#define EMPTY '.'   //empty cell

char grid[NROW][NCOL];    //the entire grid, declared in the main()
```

A. Let us handle **ONE** column. The function `fall_one_step` attempts to move the block(s) down **ONE** space if possible. For example, if we use this function on column 3 in the example above, you get

	3
0	N
1	N
2	.
3	X
4	.
5	.

$\Longrightarrow$

	3
0	.
1	N
2	N
3	.
4	X
5	.

Implement the function `fall_one_step`. [8 marks]

```
bool fall_one_step( char grid[NROW][NCOL], int col )
//Purpose: Move block(s) down one step in column "col"
//Return: true if any block is moved, false if cannot move
{
    bool fall = false;

    for (int i = NROW-1; i > 0; i--) {
        if (grid[i][col] == EMPTY and grid[i-1][col] != EMPTY) {
            grid[i][col] = grid[i-1][col];
            grid[i-1][col] = EMPTY;
            fall = true;
        }
    }
    return fall;
}
```

B. Using the `fallOneStep()` function, apply gravity to **ALL** columns. Block(s) in each column will fall **as far as possible**, i.e. this function will change the grid as shown at the beginning of the question. [4 marks]

```
void apply_gravity( char grid[NROW][NCOL] )
//Purpose: Move block(s) as far as possible for ALL columns
{
   for (int i = 0; i < NCOL; i++)
       while (fallOneStep(grid, i));   //empty loop
}
```

C. Give time complexity for the function `apply_gravity()`. $O(NCOL \times NROW^2)$

[3 marks]

Question 2: A car costs how much?! [15 marks]

It is well known that Singapore has the most expensive car price in the world. Suppose the car price is composed of three components, two of which are described in the C++ structure below:

```
struct CarPrice {
    int OMV;   //original market value, the "raw" price of a car
    int COE;    //Certificate of Entitlement
};
```

The last component, Additional Registration Fee (ARF), is calculated base on the OMV as follows:

OMV	ARF applied
$20,000 and below	100% of OMV
$20,001 to $50,000	140% of OMV
$50,001 and more	180% of OMV

So, a car with OMV of $40,000 and COE of $35,000 will have a final price of

$$\underset{OMV}{\$40,000} + \underset{COE}{\$35,000} + \underset{ARF}{140\%} \times \$40,000 = \$131,000$$

A. Suppose we consider the desirability between *carA* and *carB* base on the following criteria:

1. Final price (OMV + COE + ARF): lower the more desirable.
2. If final price is the same, then **higher** COE is more desirable (for resale value).

Implement the function `compare_desirability` which takes carA and carB as inputs, and returns <0 if carA is more desirable than carB, >0 if carB is more desirable than carA, and 0 if both cars are equally desirable. You can add helper function(s) if needed. [8 marks]

```
int compareDesirability( CarPrice carA, CarPrice carB )
//Purpose: Compare the desirability of carA vs carB
//Return: 0    if carA is equally desirable compared to carB
//   <0  if carA is more desirable than carB (return any negative value)
//   >0  if carA is less desirable than carB (return any positive value)
{
   //Criteria 1. Final price
   int diff = final_price( carA ) - final_price( carB );
   if (diff != 0)
       return diff;

   //Critera 2. COE
   return carB.COE - carA.COE;  //note the reverse direction
}

int final_price( CarPrice car )
{
   double factor = 1;

   if (car.OMV >= 50001) {
       factor = 1.8;
   } else if (car.OMV >= 20001) {
      factor = 1.4;
   }
   return car.OMV + car.COE + factor * car.OMV;
//the last term is the ARF
}
```

B. Suppose an array of `CarPrice` structures is already **sorted** in ascending order of desirability. Adapt the standard binary search algorithm to search for a target CarPrice structure in the array. You can cross out/overwrite the code directly instead of rewriting the whole function. [4 marks]

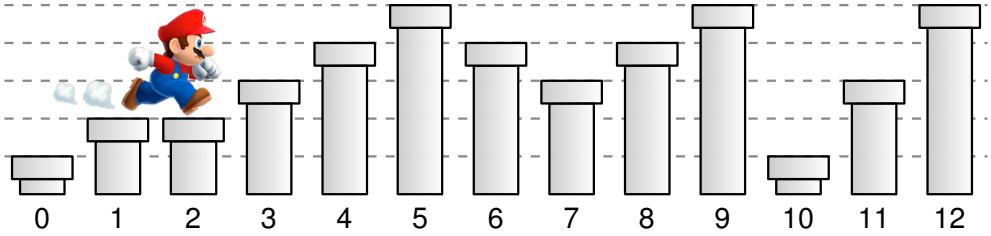
```
int search( CarPrice array[], CarPrice target, int low, int high )
//Purpose: Binary search for target carPrice in array[low...high]
//Return: -1   if target cannot be found
//        ≥0  the index of the target structure in array
{
    int idxOfX = -1, mid;

    while ( (low <= high) && (idxOfX == -1) ) {

        mid = (low + high) / 2;
        result = compareDesirability( a[mid], target );
        if ( a[mid] == target ) {
        if ( result == 0) {
            idxOfX = mid;

        if (result > 0) {
        if ( a[mid] < X ) {
            low = mid + 1;

        }  else {

            high = mid - 1;

        }
    }
    return idxOfX;
}
```

C. If `search()` from above found a match for `target`, do you think the matched CarPrice is the **exact same** as the `target`? (i.e. having the same OMV and COE). Why? [3 marks]

Yes. Since cars with the same desirability must have the same COE (2nd condition), that implies the two cars must have the same OMV too (1st condition) too.

Question 3: Mario Pipes [24 marks]

In a certain game, Mario has to jump from pipe to pipe reach end of the course.



The only issue is the Mario can only jump up or down at most one unit height from each pipe. He will lose the game if he encounters a jump that he cannot preform because it is either too high or too low.

In the example illustrated above, Mario will make it all the way to pipe 9 and not be able to continue onto pipe 10 because the drop is more than one unit height.

Suppose each pipe is modeled as the following struct:

```
struct Pipe {
    int height;
    Pipe *next;
};
```

where `height` is the unit height of the pipe, and `next` is a pointer to the next pipe in the course. The last pipe in the course will have the value `NULL` in next.

A. Implement a function `can_win` that takes as input the first pipe of the course, and returns `true` if Mario is able to complete the course, and `false` otherwise. You may assume that the course ultimately ends. Feel free to modify the input to pass by pointer or reference as needed.

Hint: You might want to use the inbuilt `abs(int x)` function which returns $|x|$. [8 marks]

```
bool can_win(Pipe *p) {
    while (p->next) {
        if (abs(p->height - p->next->height) > 1)
            return false;
        p = p->next;
    }
    return true;
}
```

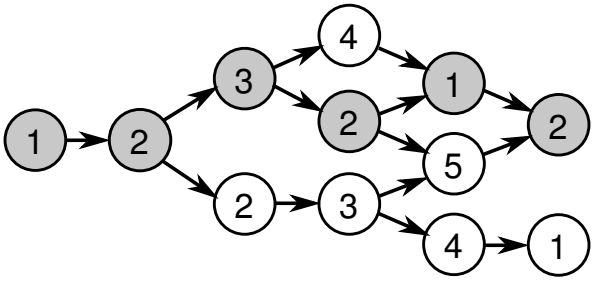
B. Suppose Mario has springs on his shoes, which allow him to safely jump down any height. At the same time, the springs will be able to propel him upwards to the same height as the fall if needed to reach the next pipe.

For example, in the illustration on the previous page, he can fall 4 units from pipe 9 to 10. The springs will be able to propel him up 4 units to clear the next pipe, which he does as pipe 11 is only 2 units higher. However, once on step 11, he will not be able to clear step 12 as it is 2 steps higher than 11 and he does not have a fall to use his spring shoes.

Implement a function `can_win_with_springs` that takes as input the first pipe of the course, and returns `true` if Mario is able to complete the course, and `false` otherwise. Feel free to modify the input to pass by pointer or reference as needed. [8 marks]

```
bool can_win_with_springs(Pipe *p) {
    int spring = 1;
    while (p->next) {
        int diff = p->height - p->next->height;
        if (diff + spring < 0)
            return false;
        spring = max(1, diff);
        p = p->next;
    }
    return true;
}
```

C. In an advanced level, the course is no longer a single path, but there could be forks along the way. The illustration below shows a top down view of the pipes that Mario can jump across. The numbers indicate the height of the pipe.



There are multiple path leading to the end of the course, but in this example, there is only one possible path that Mario can take without using spring shoes. All the other paths have jumps that are too high or low.

The struct modeling each Pipe is now modified as follows:

```
struct Pipe {
    int height;
    Pipe *next, *alt;
};
```

where the additional pointer `alt` refers to an alternate pipe to jump to. If no alternate exists it will be `NULL`. `next` will always point to a next pipe, unless it is one of the finishing pipes in the course then it will have the value `NULL`.

Implement a function `can_win_multi` that takes as input the first pipe of the course, and returns `true` if Mario is able to complete the course **without spring shoes**, and `false` otherwise. You may assume that there are no cycles and the course ultimately ends. Feel free to modify the input to pass by pointer or reference as needed. [8 marks]

```
bool can_win_multi(Pipe *p) {
    if (p->next == NULL)
        return true;
    if (abs(p->height - p->next->height) <= 1 and
        can_win_multiple(p->next))
        return true;
    if (p->alt and abs(p->height - p->alt->height) <= 1 and
        can_win_multiple(p->alt))
        return true;
    return false;

}
```

Question 4: Grab Makan [6 marks]

Grab Makan runs a delivery service with a fleet of different vehicles. Consider the following implementation where we model the delivery vehicles of Grab Makan.

```
class Vehicle {
private:
    int reach;

public:
    vector<Order> completed_orders;

    Vehicle(int reach) {
        this->reach = reach;
    }

    virtual bool deliver(Order order) {
        if (reach < order.distance) {
            cout << "Order is too far" << endl;
            return false;
        } else {
            cout << "Order delivered" << endl;
            completed_orders.push_back(order);
            return true;
        }
    }
};

class Motorcycle : public Vehicle {
private:
    int fuel;

public:
    Motorcycle(int fuel) : Vehicle(fuel) {
        this->fuel = fuel;
    }

    virtual bool deliver(Order order) {
        if (fuel < order.distance) {
            cout << "Not enough fuel" << endl;
            return false;
        } else if (Vehicle::deliver(order)) {
            fuel -= order.distance;
        }
    }
};
```

Assume that `Order` is a struct containing a public property distance, i.e.

```
struct Order {
    int distance;
    ... // other fields
};
```

A. What is written to standard output when the following lines of code are executed:

```
Vehicle *v = new Motorcycle(10);
Order o;

o.distance = 5;
v->deliver(o);

o.distance = 12;
v->deliver(o);

o.distance = 8;
v->deliver(o);

o.distance = 5;
v->deliver(o);

cout << v->completed_orders.size() << endl;
```

[6 marks]

```
Order delivered
Not enough fuel
Not enough fuel
Order delivered
2
```

— END OF QUESTIONS —

Scratch Paper

— E N D O F P A P E R —