ECS529U Algorithms and Data Structures **Week 6 Lab sheet**

This lab gets you to work with dynamic programming and greedy algorithms and also practically compare their efficiency by testing them on randomly generated inputs.

Marks (max 5): Questions 1-4: 1 each | Question 6: 0.5 | Questions 5,7: 0.25 each

Question 1

We define the Thribonacci sequence of numbers by the following function $thrib$:

- $thrib(0) = 0$
- $thrib(1) = 1$
- $thrib(2) = 2$
- $thrib(n) = thrib(n\text{-}1) + thrib(n\text{-}2) + thrib(n\text{-}3)$, if $n > 2$

Write a recursive Python function

```
def thrib(n)
```

that, on input `n`, returns $thrib(\mathtt{n})$. Then, change your function into a dynamic programming one:

```
def thribDP(n)
```

using memoisation.

Question 2

Change your DP function from Question 1 into a dynamic programming bottom-up one:

```
def thribDPBU(n)
```

using iteration.

Question 3

Write Python functions:

```
def coinSplitTime(n)
def coinSplitGDTime(n)
def coinSplitDPTime(n)
def coinSplitDPBUTime(n)
```

that run `coinSplit(n)`, `coinSplitGD(n)`, `coinSplitDP(n)` and `coinSplitDPBU(n)` respectively on input `n` and return the time taken for each of them to return.

Test your timing functions on values 10, 100, 1000, 10000 for `n` and fill in the next table. Use these two choices for `coin`:

coin1 = [200, 100, 50, 20, 5, 2, 1]   coin2 = [200, 199, 198, ..., 3, 2, 1]

To avoid waiting forever, if a run takes more than e.g. 15 seconds then kill it and fill in “timeout” in the table.

value n / coin array	10 / coin1	100 / coin1	1000 / coin1	10000 / coin1	10/ coin2	100 / coin2	1000 / coin2	10000 / coin2
coinSplit time (sec)								
coinSplitGD time (sec)								
coinSplitDP time (sec)								
coinSplitDPBU time (sec)								

Question 4

Using dynamic programming, write a Python function:

```
def closestSubsetDP(s,A)
```

that takes an integer `s` and an array of positive integers `A` and returns an array consisting of elements of `A` which add up to `s`. If there is no subset that adds up to `s`, the function should instead return the subset which adds up to the value closest to `s`. For example:

- if `A` is [12, 79, 99, 91, 81, 47] and `s` is 150, it will return [12, 91, 47] as 12+91+47 is 150
- if `A` is [15, 79, 99, 6, 69, 82, 32] and `s` is 150 it will return [69, 82] as 69+82 is 151, and there is no subset of `A` whose sum is 150.

Test the method with arrays generated by `randomIntArray(s,n)` from Lab 3. Try with:

```
A = randomIntArray(20,1000)
subset = closestSubsetDP(5000,A)
```

You can start from your recursive solution of Question 4 from Lab 5 (if you have not solved that question yet, you can wait until our solutions become available). Does this version run faster than the simple recursive one? What about when compared to the greedy one?

Question 5

Download the notebook `maze.ipynb` from QM+. This notebook defines a class `Maze` for representing 2D mazes. Each object of class `Maze` contains:

- A two-dimensional array `position` of boolean values that represents the maze. Each `position[h][w]` represents a position in the maze: if `position[h][w]` is `True` then the position is empty; if `position[h][w]` is `False` then the position is filled (i.e. it is a wall).
- Integers `width` and `height` storing the width and height of the maze respectively.
- Integers `enter` and `exit` for the entrance and exit of the maze. The entrance to the maze is at `position[0][enter]` and the exit at `position[height-1][exit]`, and they must both be true (i.e. represent empty positions). A maze will always have one entrance and one exit.
- A function `__str__(self)` that prints the maze into a string. Each free position is represented by a white space, and each wall position is represented by an `X`.

- A constructor `__init__(self,width,string)` that creates a new maze from a string of length `height*width`. The conversion is simple: the string must consist of white spaces and `X`'s, representing the positions of the maze, in a single sequence. The required width of the maze is also given as an argument (in order to know how to break down the `string` into lines of the maze).

  For example, we can build mazes from strings `mazestring1` and `mazestring2`. Note that in Python we use \ to mean that a line of code continues to the next line.

After downloading the code, construct at least one string representing a maze of your choice, use the `Maze` constructor to convert it into a `Maze` object, and then print the object.

Question 6

Let `m` be a maze. At any position `m.position[h][w]`, there are up to four moves that can be made: **Up** moves us to `m.position[h+1][w]`, **Left** moves us to `m.position[h][w-1]`, **Right** to the `m.position[h][w+1]`, and **Down** to `m.position[h-1][w]`.

A route through the maze can represented by a string containing the characters `U`, `L`, `R` and `D`, each standing for one of the four moves. The route needs to take us from the entrance to the exit of the maze without hitting any walls. For example, `ULLLLUUURRRU` represents a route through the maze represented by `mazestring1`. Also, `URRUULLULU` represents another route through the same maze which is shorter.

Work out an algorithm that finds a route through a maze and implement a function:

```
def findRoute(m)
```

that takes a maze `m` and returns a string representing a route in it, or `None` if it cannot find any routes in `m`.

The algorithm should be greedy. One idea could be the following: at each step, the algorithm picks a direction where there is a free position and moves to it. The algorithm could have a preference for the `U` direction instead of the other ones, since the general direction of a route is always going upwards.

Question 7

Being greedy, if the algorithm of Question 6 finds a route it does so quickly, but:

a) it may get stuck in a loop inside the maze, visiting the same positions over and over again
b) it may return `None` when there is a route.

Explain in your own words how the Dynamic Programming technique could be used to resolve problem (a) of your algorithm. How about problem (b)?