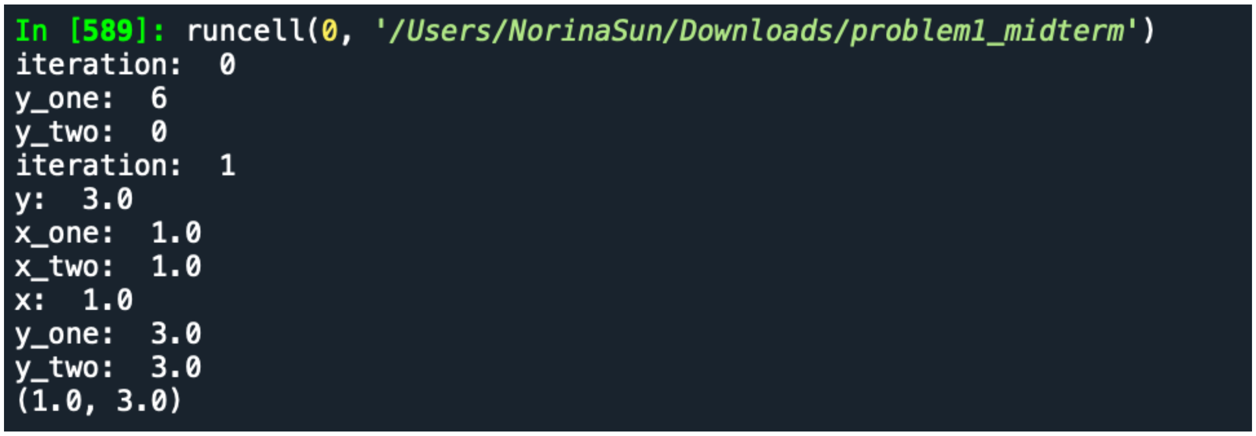
HEC MONTRĒAL

MATH 60607 MIDTERM FALL 2020

HUAI JUN SUN
11287405

PROBLEM ONE

Question 1



```
In [589]: runcell(0, '/Users/NorinaSun/Downloads/problem1_midterm')
iteration:  0
y_one:  6
y_two:  0
iteration:  1
y:  3.0
x_one:  1.0
x_two:  1.0
x:  1.0
y_one:  3.0
y_two:  3.0
(1.0, 3.0)
```

In one iteration of the algorithm, it terminates and we find that the results are x = 1 and y = 3.

Question 2

This algorithm will stop when the absolute difference between y_1 and y_2 is less than the value e, which in this case is sufficiently small. In other words, when the algorithm terminates, $y_1 \approx y_2 \approx y$ and in addition $x \approx x_1 \approx x_2$ (save for almost negligible differences of sizes < e).

We know this to be true, as each iteration of the algorithm forces the values of x and y towards their final values through the averaging process between y_1 and y_2 and x_1 and x_2. As the values of x and y (and subsequently x_1, x_2, y_1, y_2) are dependent through the relationships defined by $x_i \leftarrow \frac{y - b_i}{a_i}$ and $y_i \leftarrow a_i x + b_i \textit{ where } i \textit{ in } \{1,2\}$, we know that despite the condition for the loop depending only on the value y, by the iteration where $y \approx y_1 \approx y_2$, $x \approx x_1 \approx x_2$.

Knowing that when the algorithm stop, $y \approx y_1 \approx y_2$ and $x \approx x_1 \approx x_2$, we can algebraically prove what the final values of x and y will be.

For the value of y:

$$x = x_1 = x_2 = \frac{y - b_1}{a_1} = \frac{y - b_2}{a_2}$$

$$a_2 y - a_2 b_1 = a_1 y - a_1 b_2$$

$$y(a_2 - a_1) = a_2 b_1 - a_1 b_2$$

$$y = \frac{a_2 b_1 - a_1 b_2}{a_2 - a_1}$$

For the value of x:

$$y = y_1 = y_2 = a_1x + b_1 = a_2x + b_2$$

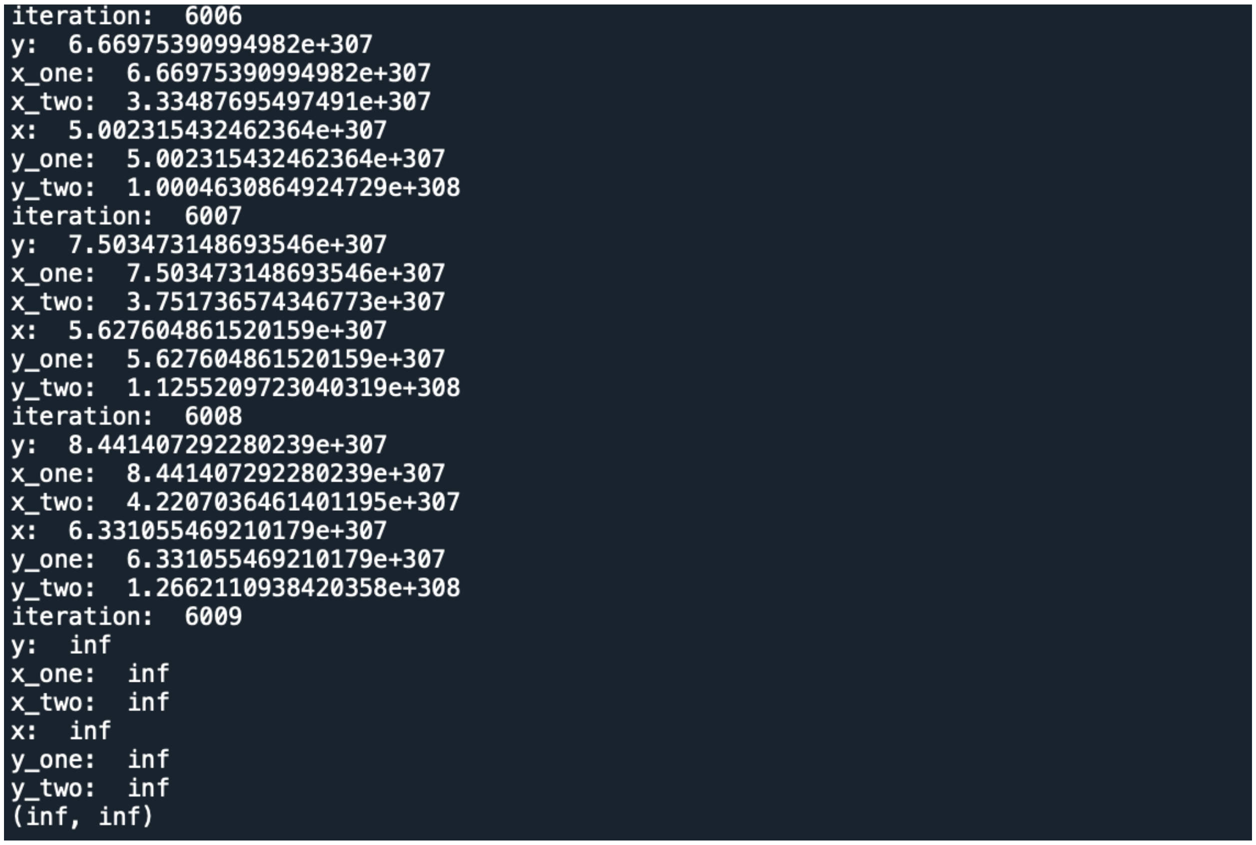
$$a_1x - a_2x = b_2 - b_1$$

$$x(a_1 - a_2) = b_2 - b_1$$

$$x = \frac{b_2 - b_1}{a_1 - a_2}$$

Question 3

Should the constraint ($a_1 * a_2 < 0$) not be respected, the algorithm will not stop and the values of x and y will head towards infinity.



```
iteration:  6006
y:  6.66975390994982e+307
x_one:  6.66975390994982e+307
x_two:  3.33487695497491e+307
x:  5.002315432462364e+307
y_one:  5.002315432462364e+307
y_two:  1.0004630864924729e+308
iteration:  6007
y:  7.503473148693546e+307
x_one:  7.503473148693546e+307
x_two:  3.751736574346773e+307
x:  5.627604861520159e+307
y_one:  5.627604861520159e+307
y_two:  1.1255209723040319e+308
iteration:  6008
y:  8.441407292280239e+307
x_one:  8.441407292280239e+307
x_two:  4.2207036461401195e+307
x:  6.331055469210179e+307
y_one:  6.331055469210179e+307
y_two:  1.2662110938420358e+308
iteration:  6009
y:  inf
x_one:  inf
x_two:  inf
x:  inf
y_one:  inf
y_two:  inf
(inf, inf)
```

PROBLEM TWO

Question 1

The path with the maximum capacity from A to C is A → B → D → C with a capacity of 5. Though an edge exists between A → C directly, it's capacity is only 4. If we take the route A → B → D → C, the minimum capacity of all of the individual edges is 5. We can confirm that this route provides the best solution or is one of the best solutions as there are only three edges connecting to node C, with capacities of 4, 5, and 2. Thus, there exists no solution which could provide a capacity greater than 5.

Question 2

We see that it is impossible to expect a path from A → B with a capacity greater than 6. As node B only has three connecting edges with capacities of 6, 4, and 6 we know that at maximum any route from any node to B can at maximum have the capacity of 6.

Question 3

The algorithm I proposed is a modified version of Dijkstra's Algorithm. The two main differences are that:

1. Rather than performing a summation at each step to calculate the cost of travelling to the node, I take the minimum capacity values of the edges as the assigned value.
2. When it comes to updating the assigned value, I take the largest value available rather than the smallest one.

Some smaller differences to make the algorithm work:

3. All nodes are initiated with a capacity of negative infinity rather than positive infinity.
4. As a quirk of the algorithm, I begin with the capacity of the starting node at positive infinity rather than negative infinity. This is later changed to a value of 0 at the end of the algorithm as it should be. Beginning the algorithm with the capacity at negative infinity for the start node would have resulted in undesirable behaviour.

A brief overview of the algorithm:

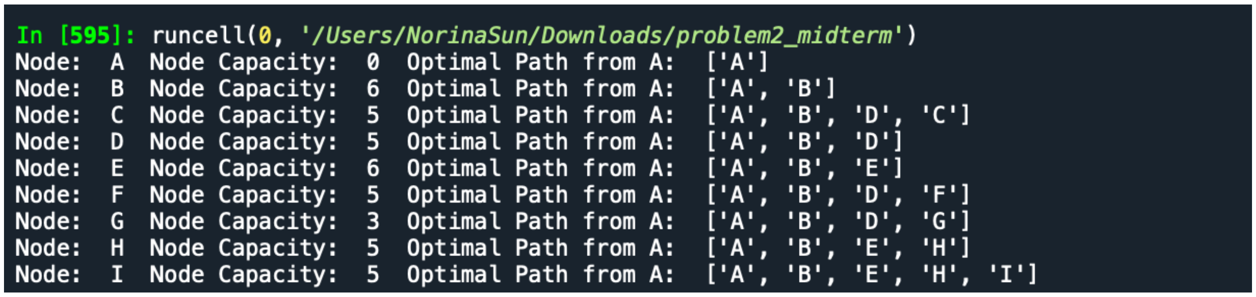
1. Initiate a matrix of size n*n (n = the number of nodes) to represent the edges between nodes and their respective values (capacity).
2. Create an object node, which has attributes: id, capacity, path, and visited.
3. We begin with the start node
    a. Loop through each node (observation node) other than the current node. If there exists a vertex:
        i. Assign the capacity of this node (observation node) to ***max*(*min*(current node capacity, edge value), observation node capacity)***

        ii. If the capacity is updated to either the current node capacity, or the edge value, then we update the path attribute of the observation node to the path of the current node + observation node.
    b. We mark the current node as visited
    c. We remove the node from the list of nodes to visit
4. We determine the next node to visit as the node with the largest current capacity and go through the same process outlined above.
5. We continue until the list of nodes to visit is empty.

Please see the attached python code for the full details.

Question 4

Having applied the algorithm described above to figure 1 starting at node A, we see the following results:



```
In [595]: runcell(0, '/Users/NorinaSun/Downloads/problem2_midterm')
Node:  A  Node Capacity:  0  Optimal Path from A:  ['A']
Node:  B  Node Capacity:  6  Optimal Path from A:  ['A', 'B']
Node:  C  Node Capacity:  5  Optimal Path from A:  ['A', 'B', 'D', 'C']
Node:  D  Node Capacity:  5  Optimal Path from A:  ['A', 'B', 'D']
Node:  E  Node Capacity:  6  Optimal Path from A:  ['A', 'B', 'E']
Node:  F  Node Capacity:  5  Optimal Path from A:  ['A', 'B', 'D', 'F']
Node:  G  Node Capacity:  3  Optimal Path from A:  ['A', 'B', 'D', 'G']
Node:  H  Node Capacity:  5  Optimal Path from A:  ['A', 'B', 'E', 'H']
Node:  I  Node Capacity:  5  Optimal Path from A:  ['A', 'B', 'E', 'H', 'I']
```

Question 5

I estimate that the complexity of the proposed algorithm is O(n^2). This is because I have a for loop nested within a while loop. Thus, the algorithm is required to loop through the values of n, n^2 times for a given number of nodes n.

PROBLEM THREE

Question 1

If channels 3, 5, and 8 are selected then only channel 9 can be used as a fourth channel. Channel 10 cannot be selected, as it 10 – 8 = 5 – 3 = 2. Channel 11 is also unable to be selected as 11 – 8 = 8 – 5 = 3.

Question 2

My proposed algorithm is available in detail in the python code attached. A brief overview of it is here:

1. Combined the existing set of selected channels and the new proposed channel into a list. Calculate the absolute difference between any value in this list and itself, and add this difference to a difference list. It is important to distinguish that the difference between any two values will not be calculated twice in this algorithm.
2. Generate a secondary list which only contains the unique values of the difference list.
3. Compare the lengths of the difference list and the secondary list. If they differ, it implies that there exists a duplicate in the difference list, and the new channel proposed is not viable.

It is extremely important to note that this algorithm operates on the assumption that the existing set of selected channels already meets the conditions described in the question. This algorithm will not work otherwise.

Question 3

In order to maximize the number of channels selected in a range of values, I propose that we seek to contain one instance of each possible difference value. For example, should the range be from 1 to 20, I would begin with the lowest value 1. The minimum possible difference would at this stage would be 1, thus we simply add this value to our current maximum selected set value of 1, and we are returned the value 2 which we add to the selected channel set. Next, we seek to have a difference of 2, thus we add the value 2 to the current maximum value in the selected set which is now 2, and are returned a value of 4.

We continue this pattern of generating the next number by adding the next largest difference not used to the current largest value in the selected set. Of course, we run into two main issues:

1. For example where 4 – 2 = 2, but also 4 – 1 = 3. Thus, by adding the value 4 to our selected set, the difference of both 2 and 3 are no longer available. To remedy this, at each new addition we calculate all of the differences between the selected set and remove these values from the list of difference to cover.

2. Another issue we run into is that in some cases adding the next smallest available difference to the largest available channel in the selected channel set will return a value which generates a duplicate difference value with another channel in the selected channel set. To remedy this, we use the algorithm designed in question 2 to perform a check if the proposed channel is valid. Should it not be valid, we simply increment its value by 1 until it is.

Ultimately, I propose this algorithm as a faster way of finding the maximum possible set values within a range compared to simply checking each value in the range incrementally. While I am unable to prove that this algorithm will yield the optimal solution within the scope of this exam, I am fairly confident that it provides an **acceptable** solution.

Details of how this algorithm is implemented can be found in the corresponding attached python code.

PROBLEM FOUR

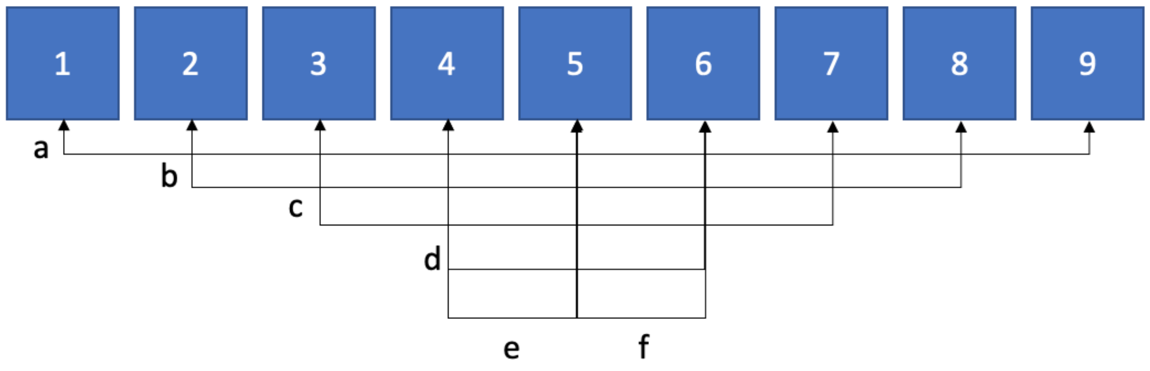
Question 1

The sequences that respect the decreasing difference condition are:

{2, 11, 4, 7}, {2, 11, 7, 4}, {11, 2, 7, 4}, and {11, 7, 4, 2}.

Question 2

Consider a sorted list of numbers. In order to respect the decreasing difference condition, all one would need to do is sequence the numbers alternating from each end of the list at each turn. Please see the visual below:



a > b > c > d > e = f

A list respecting the condition would thus be: {1,9,2,8,3,7,4,5,6}.

Now, I did not want to make the assumption that the list provided would be sorted, thus one possible solution would simply be an algorithm that performed a sort on the provided list, and then compiled a solution using the technique I outlined above.

Another solution is the algorithm I propose here:

1. Give a sequence, I collect the minimum value of the sequence as the current value.
2. Then, while the length of the provided sequence is greater than 1:
    a. Add the current value to the solution.
    b. Calculate the absolute difference between the current value and all other values in the sequence and store these values in a list.

   c. Then find the maximum value in this list, and store its index in the sequence in a variable named “index”.
   d. Store the current value in another variable named “old_value”.
   e. Set the current value as the value at “index”, effectively setting it to the value in the sequence the furthest from the old value.
   f. Remove the old value from the sequence.
3. Finally, add the last remaining value in the sequence to the solution.

You can see details of this algorithm in the corresponding python code.

Both of these algorithms would produce a viable solution. This one I estimate has a complexity of O(n^2) since it is primarily composed of a for loop inside of a while loop. Most sorting algorithms also have a complexity of O(n^2) thus the first solution would likely also be of complexity O(n^2). However, there do exist sorting algorithms with time complexity of better than O(n^2), so perhaps if you had a data structure that supported one of those sorting algorithms then the first solution would be better.