CISC-365*
Test #1 Sample Questions
Fall 2019

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, **but will not be re-marked under any circumstances.**

The test will be marked out of 50.

QUESTION (15 marks)

Let A and B be two sets, each containing n integers in random order. Each of the sets is stored in an n-element array.

Create an algorithm to compute $A \cap B$ (that's "A intersect B"). Your algorithm should run in $O(n * \log n)$ time.

(A note on data structures: many people are tempted to solve problems like this using hash-tables which give O(1) ***expected case*** search time. Unfortunately the ***worst case*** search time for a hash-table is O(n).)

Express your algorithm in clear pseudo-code or a standard procedural language. You may assume that ***sort()*** is a built-in function that runs in $O(n * \log n)$ time.

Solution:

```
# assuming arrays are indexed from 1 to n

A_B_intersect = empty list

A.sort()            # using the built-in sort function
                    # Note:  syntax is not important here:
                    #        "sort(A)"  or "sort A" is fine

for i = 1 to n:
      use binary search to search A for B[i]        # takes O(log n)
                                                    # time
      if found, add B[i] to A_B_intersect
```

Analysis: the sort takes O(n*log n) time. The loop iterates n times and each iteration takes O(log n) time. Thus the loop takes O(n*log n). Thus the entire algorithm takes O(n*log n) time

Alternative solution:

```
A_B_intersection = empty list

A.sort()
B.sort()

A_pos = 1
B_pos = 1

while (A_pos <= n)  and  (B_pos <= n):
    if A[A_pos] == B[B_pos]:
        add A[A_pos] to A_B_intersection
        A_pos ++
        B_pos ++
    else if A[A_pos] < B[B_pos]:
        A_pos ++
    else:
        B_pos ++
```

Analysis: The sorting operations both take O(n*log n) time.
The loop iterates at most 2*n times, and does constant time work on each iteration so the loop takes O(n) time.
The sorting is the most time consuming part of the algorithm so the whole algorithm takes O(n*log n) time

Other solutions are certainly possible.

QUESTION (15 marks)

What is the computational complexity (ie the "big O" class) of this algorithm?

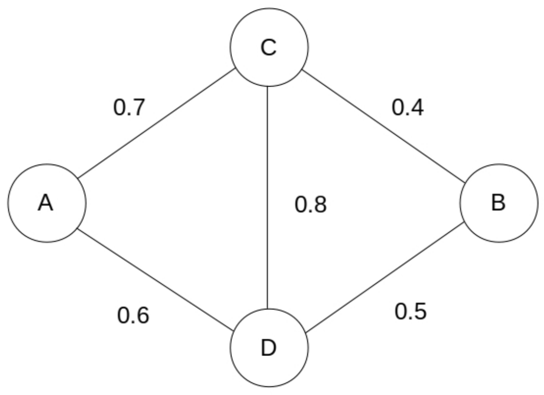
```
Mystery(n):
    if n <= 1:
        print 1
    else if n <= 100:
        print n
        Mystery(n-1)
    else:
        print n
        Mystery(n/2)
```

Solution:

When determining the Big O classification we only worry about what happens when n is arbitrarily large. So it really doesn't matter what happens when n <= 100 ... the "else" clause is the only one that matters. For this part, $T(n) = c + T(n/2)$ ... and we know that this means the algorithm is in $O(\log n)$

QUESTION (15 marks)

Consider the Path Product Problem: Given a graph G in which every edge is weighted with a number in the range [0 .. 1] , and given two identified vertices A and B, find a path from A to B that **maximizes** the **product** of the weights of the edges in the path.



For example in this graph the optimal path from A to B is A-D-B because 0.6 * 0.5 is greater than the product of the weights in any other path from A to B

Dijkstra's Algorithm be adapted to solve the Path Product Problem.

Dijkstra's Algorithm is stated on the next page, exactly as given in the course notes. This version finds the least-weight paths from A to all other vertices. You are **not** required to change it to terminate as soon as B is reached.

```
Dijkstra(W, A):
    Cost[A] = 0
    Reached[A] = True
    for each other vertex x:
        Reached[x] = False
    for each neighbour x of A:
        Estimate[x] = Weight(A,x)
        Candidate[x] = True
    for all other vertices z:
        Estimate[z] = infinity
        Candidate[z] = False
    while not finished:
        # find the best candidate
        best_candidate_estimate = infinity
        for each vertex x:
            if Candidate[x] == True and Estimate[x] < best_candidate_estimate:
                v = x
                best_candidate_estimate = Estimate[x]
        Cost[v] = Estimate[v]
        Reached[v] = True
        Candidate[v] = False
        for each vertex y:                    # update the neighbours of v
            if W[v][y] > 0  and  Reached[y] == False:
                if Cost[v] + W[v][y] < Estimate[y]:
                    Estimate[y] = Cost[v] + W[v][y]
                    Candidate[y] = True
                    Predecessor[y] = v
```

Explain how to modify this algorithm to solve the Path Product Problem. You don't need to copy the whole algorithm - just show the lines that need to change.

Solution:

The key to this problem is realizing that since all the edge weights are <= 1, every time we extend a path with another edge the resulting path product value must be <= the value before we added the edge. So if we choose the candidate x that has the **maximum** path product estimate, we know that path must be the best path to x – because all other paths to candidates have product values <= the one we have chosen, and extending those paths can only reduce their path product values. Thus any other path to x must have product value <= the path we have already found.

The changes required are simple – most relate to the fact that we are maximizing, not minimizing:

Old line	New line
Estimate[z] = infinity	Estimate[z] = 0
best_candidate_estimate = infinity	best_candidate_estimate = 0
if Candidate[x] == True and Estimate[x] < best_candidate_estimate:	if Candidate[x] == True and Estimate[x] > best_candidate_estimate:
if Cost[v] + W[v][y] < Estimate[y]:	if Cost[v] * W[v][y] > Estimate[y]:
Estimate[y] = Cost[v] + W[v][y]	Estimate[y] = Cost[v] * W[v][y]

The lines to be changed are highlighted in the code shown above.

QUESTION (15 marks)

Let A be an array of n distinct integers ($n \geq 3$), arranged so that the integers start out increasing, and then decrease. For example A might look like this:

A = [2, 5, 7, 93, 86, 81, 77, 34, 22, 11, 9, 8, 6]

Create an algorithm that finds the largest value in A in $O(\log n)$ time. Your algorithm must solve all instances of the problem, not just the one given in the example.

Solution:

We can adapt Binary Search – instead of seeking a particular value we are seeking a value that is larger than both its neighbours. If the value we look at is larger than one neighbour and smaller than the other, we know which side the largest value is on so we shrink our search area accordingly.

```
def Max_find(A):        # A is indexed 1 to n
    first = 1
    last = n            # we know the initial value of last is >= 3
    while last − first >= 2:
        mid = (first+last)/2          # integer division
        if (A[mid] > A[mid-1]) and (A[mid] > A[mid+1]):
            return mid                # we found it!
        else if A[mid] < A[mid-1]:
            # the max value lies to the left
            last = mid-1
        else:
            # the max value lies to the right
            first = mid+1
    # after the loop ... either first == last or first = last − 1
    if first == last:         # there's only one spot left
        return first
    else:                     # it's the larger of the two remaining
                              # values
        if A[first] > A[last]:
            return first
        else:
            return last
```

I've annotated my solution for explanatory purposes. Your solution would not need so many comments.

This solution is not unique.