CPSC 413 May 13, 2019
University of Calgary

Homework Assignment # 1

Due: 23. May 2019, **11:59pm**

LATE ASSIGNMENTS WILL NOT BE ACCEPTED

Please fill out the cover page at the end of this assignment and attach it to your solutions.

Collaboration Rules:

In order to solve the questions, you are allowed to collaborate with students from the same course, but not with anybody else. If you do collaborate with other students, you have to list them in the appropriate fields on the cover page. You have to write up the solutions *on your own* in *your own words*. You can meet with your collaborators in order to generate ideas, but you are not allowed to write up the solutions during these meetings or to take any notes, pictures, etc. away from those meetings.

You are allowed to use literature, as long as you cite that literature in a scientific way (including page numbers, URLs, etc.). In any case, your solutions must be self-contained. For example, if you use a theorem or lemma that was not covered in the lecture and that is not "well-known", then you have to provide a proof of that lemma or theorem.

If you are in doubt whether a certain form of aid is allowed, ask your instructor!

Academic misconduct (cheating, plagiarism, or any other form) is a very serious offense that will be dealt with rigorously in all cases. A single offense may lead to disciplinary probation or suspension or expulsion. The Faculty of Science follows a zero tolerance policy regarding dishonesty. Please read the sections of the University Calendar under the heading "Student Misconduct".

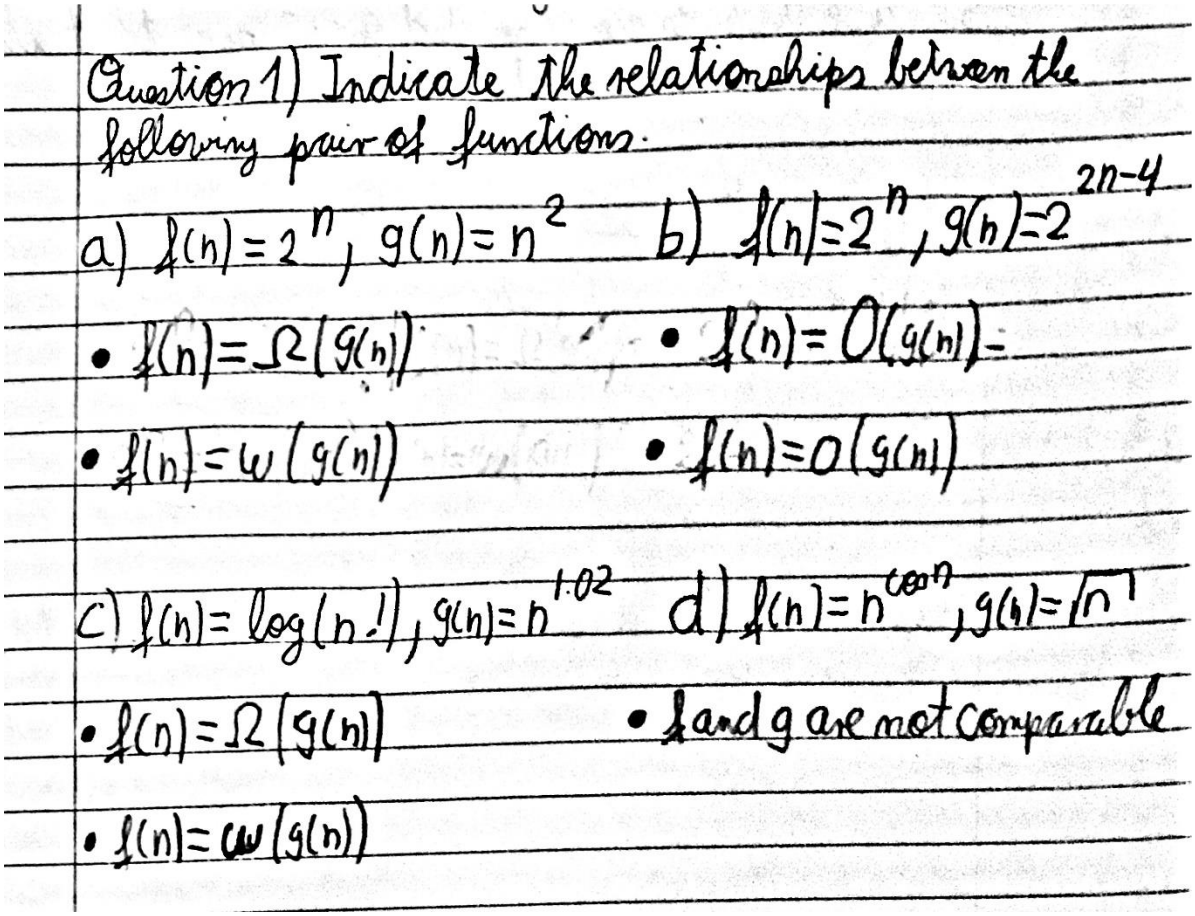
Question 1 For each of the following pairs of functions, $f(n)$ and $g(n)$, indicate whether $f = O(g)$, $f = o(g)$, $f = \Omega(g)$, $f = \omega(g)$, or $f = \Theta(g)$. Give an exhaustive list of all true relationships. No justification required.

(a) $f(n) = 2^n$, $g(n) = n^2$;

(b) $f(n) = 2^n$, $g(n) = 2^{2n-4}$;

(c) $f(n) = \log(n!)$, $g(n) = n^{1.02}$;

(d) $f(n) = n^{\cos n}$, $g(n) = \sqrt{n}$.



Question 2

(a) Look up the definition of a *biconnected* undirected graph on Wikipedia. Give a one sentence definition based on induced sub-graphs. Start your definition with "An undirected graph *tt* = (*V*, *E*) is biconnected, if . . . "

Question 2:

a) Definition of a biconnected undirected graph

An undirected graph G = (V, E) is biconnected if it is a connected graph that is not broken into disconnected pieces in any induced subgraphs of G.

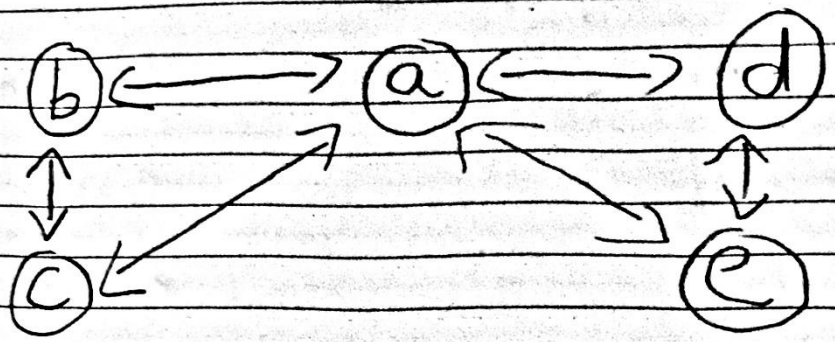
(If we create a graph G' with a subset of the vertices of G, G' would still be connected).

(b) For a *directed* graph $tt = (V, E)$, its *underlying undirected graph* is obtained by replacing every directed edge (u, v) with an undirected one $\{u, v\}$. (If (u, v) and (v, u) are both in E, then the underlying undirected graph still contains only one edge $\{u, v\}$.)
Give a *strongly connected* directed graph such that its underlying undirected graph is not biconnected.

Q2)

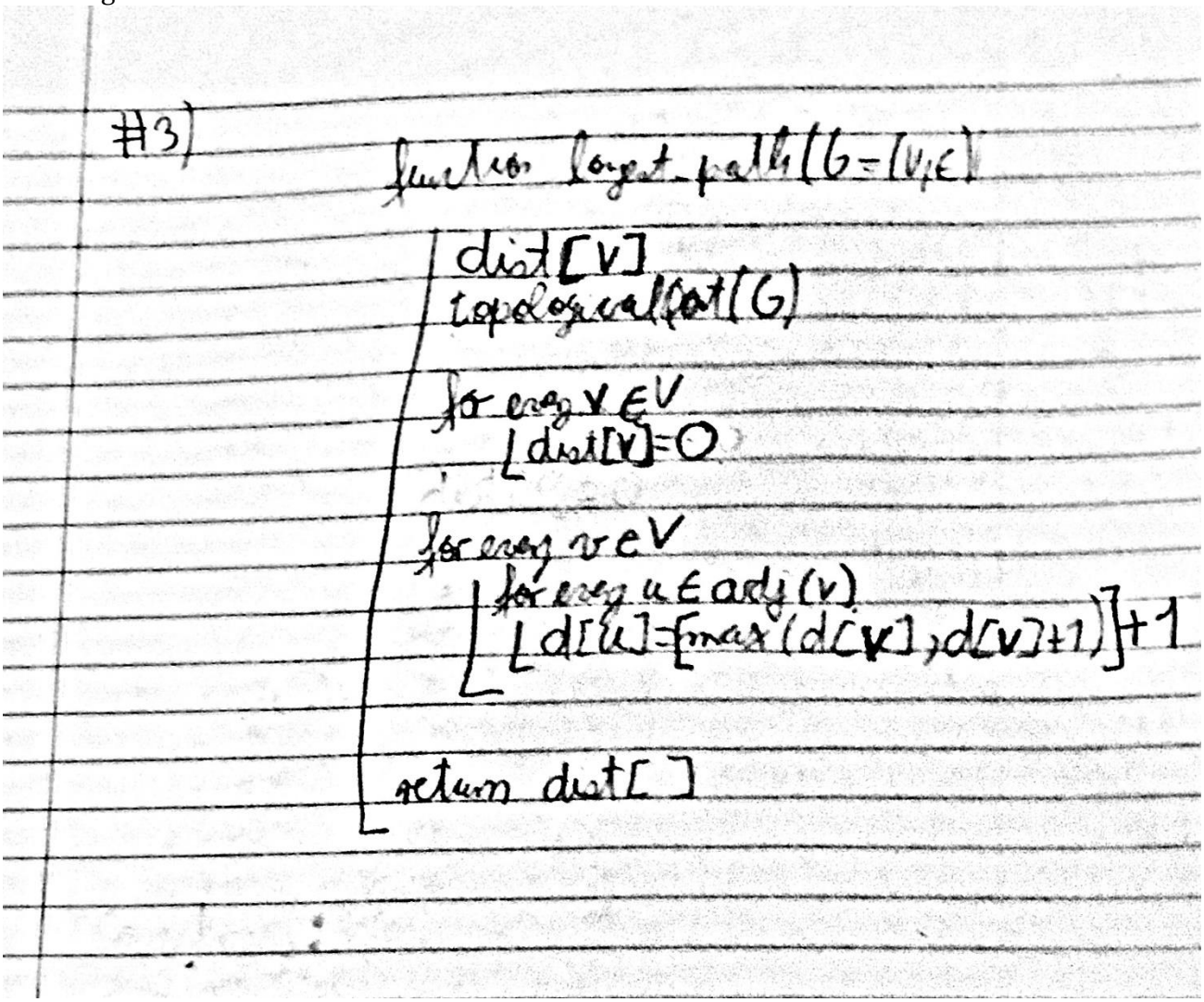
b) Give a strongly connected graph such that its underlying graph is not biconnected.

D.G.



* Eliminate a and we get that the underly undirected is disconnected into two different components.

Question 3 Design an algorithm that takes as input a DAG $G = (V, E)$ in adjacency list representation, and outputs for each vertex $v \in V$ the length $d(v)$ of the longest directed path that ends at v. For full marks, your algorithm should have worst-case running time $O(|V| + |E|)$. State your algorithm's worst-case running time and provide a brief justification for correctness and for your running time statement.



Running Time: O(V+E)

We know from lectures that topological sorting has a worst-case running time of O(V+E). The loop for every v then runs at worst case of O(V) and finally the final loop first iterates through all connected edges of which equals O(E) . It means that the overall running time is at most 2 (V+E) + c which equals O(V+E).

c**= number of constant operations

Proof of Correctness:

We will prove this algorithm works by Strong induction on the number of vertices =|v|. We will denote the number of vertices with the letter k

Base case:

k=1

Assume we run this algorithm with a graph of size 1 (k=1). It means that the graph has only one node. Since there is only one node in the graph, the starting node s doesn't need to be sorted and has no adjacent list. That's why after setting it source node (dist[s]) to 0, the algorithm then returns the dist[] array which in turns shows that the distance of the source node is equal to 0 as it should be.

k=2

In this case there can be two possibilities in a DAG graph with nodes "a" and "b"

1.-a points to b
2.-b points to a

In which the node that points to the other becomes the parent.
(In 1, a is b's parent)
(In 2,b is a's parent)

And for each respective case, the algorithm will assign the value of dist[]=0 to the parent (source node) and dist[]=1 to the child. Given that for every child node d[u] , the highest value of a parent +1 is assigned to it. As required.

Inductive Hypothesis:

Suppose the algorithm returns the correct dist[V] (longest path to each v in V) array for any graph of size 1<=k-1<k.

We want to show that the algorithm will then work for any graph of size k.

When running the algorithm with an input of k, we know that the algorithm first calculates the value of the parent nodes and then derived from that we can calculate subsequent values of dist[] for the child nodes. So in a graph of k values we can calculate the other k-1 dist[] (longest path values) starting from the source node successfully (by I.H.). Lastly, knowing that all we need to calculate is the distance of the last remaining node and that by the k-1 itineration and that we possess the dist[] value of all previous nodes including the adjacent parent nodes of the last node k, all we have to do is calculate dist[k] by using the methods previously mentioned.

Thus for a graph of size k where k is any integer >=1 it is proven by Strong Induction that the algorithm returns the correct values of dist[] for each of its vertices.

Question 4 Design an algorithm that outputs for a given list of n integers, one of the integers that occurs most often in the list. E.g., if the list is (1, 5, 3, 5, 1, 2, 7, 5, 3, 4, 9, 3), then the algorithm might output either 3 or 5, because both integers occur three times in the list, while every other integer occurs only once or twice. For full marks your algorithm should have worst-case running time $O(n \log n)$. State your algorithm's worst-case running time, and provide a brief justification for correctness and for your running time statement.

#4) Design an algorithm that returns the most frequent element of array

```
most_repeated(list[], size)
    heapsort(list[], size)
    max_count := 1
    current_count := 1
    most_frequent := list[0]

    for(i=1; i<size; i++)
        if(list[i] == list[i-1])
            current_count++
        else
            if(current_cont > max_count)
                max_count = current_cont
                most_frequent = list[i-1]
            current_cont := 1

    If(current_count > max_count)
        most_frequent := list[size-1]

    return most_frequent
```

Running Time: O(n log n)

We know from the course of CPSC 331 and from additional provided sources online, that the sorting algorithm of heapsort has a worst-case running time of O(n log n) and afterwards this algorithm goes through a loop for n (size of the array) times. Which means the total running time can be simplified to n log n+ n + c. Which is equal to O(n log n)
c**= number of constant operations

Proof of Correctness:

We will prove that the algorithm returns the correct most frequent element by strong induction on the size of array "n". We will assume, that without further explanation by the point the array reaches the loop, it has been successfully sorted by the function heapsort. (We will assure that whenever we enter the loop, the following loop invariants follow:
i)-Whenever we find a repeated element, current_count ++
ii)-Either during loop execution or after it, if current count > max_count, we will select a new element (the element we were counting for) to be our most frequent element in the array.
iii)-After loop executes, the algorithm returns the most frequent element and stops.

Base case:

n=1
Since there is only one element on array, the loop is skipped, and the value of the one element is returned as the most frequent element as required.

n=2

Now we have an array with 2 elements in total.

Here are two cases:

a)-Either array contains two equal elements (x,x)

b)-Or array contains two different elements (x,y) where x!=y

For a)
First, right before the loop we will set our max_count and current_count to be equal to 1. And our most frequent element is list [0] =x. (For this case size=2, and maxheap won't change the order of the array)

Then, we will enter the loop since i=1 < size =2. Then we would enter the first if statement. By looking at the element in position list [1] =x and comparing it to list [0]=x we can see they are equal to each other, Thus, we update and add 1 to current_count (current_count=2)
{ i) ✔}

And we will exit the loop since now i=2 =size.

Right after we read the if statement (if(current_count>max_count)) and we would enter it because current_count=2 > 1 = max_count.
And we would set x as our most frequent element. { ii) ✔}
And in the end x is returned. { iii) ✔ }

For b)
First, right before the loop we will set our max_count and current_count to be equal to 1. And our most frequent element is list[0]= x. (For this case size=2)

Then, we will enter the loop since i=1 < size = 2. Then we would skip the first if statement because list[0]=x is different to list[1]= y (because x!=y).

Thus we would enter the else statement that follows and given that at this point current_count=1 and max_count=1. The if statement inside else [if (current_count>max_count)] would be skipped and then current_count would be set as 1.

Then exiting the loop since now i=2 =size. The following if statement would be skipped since current_count = max_count and then the list[0]= x would be returned as the most frequent element. Since x and y are both the most frequent elements, x is a acceptable answer for this problem.
{ii ✔}
{iii ✔}
{Since there are no repeated elements AND current_count was not increased, we can say i) ✔ as well}

Inductive Hypothesis: Suppose that the algorithm correctly returns the most frequent element for an array of size k-1, 1<= k-1<k.

We want to prove that for an array of size k, our algorithm will return the most frequent element.

For an execution of the algorithm with an array of size k we have two possible cases:

- Element found at the kth position (after array is sorted) is different to the element in the (kth -1) position.

In this situation we know that the initial max_count was set to 1 originally and throughout the execution of the algorithm it has either stayed the same or increased. The last element is different to any other element before it because we sorted it at the beginning of the algorithm, thus when calculating that last element number of occurrences of the element (current_count), it would be equal to 1.Since there will be no change in max value, by the I.H., we output our most occurring integer.

{After k-1 steps we know that current_count was increased every time a repeated value was found and at step k, there were no repeated values found, thus current_count only increased when a repeated value was found i)✔}
{ii) ✔}
{iii) ✔}

- Element found at the kth position (after array is sorted) is equal to the element in the (kth-1) position

For this we have to consider two further subcases:

1. Current_count < max_count

Then in the last iteration of the loop we would enter the first if statement and increase current_count by 1. Making it so that by the end of the loop, current_count is either equal or still less than max_count, which in turn doesn't change the max_value of the array, and the most frequent element stays the same as the one for a k-1 array (which by I.H we can calculate successfully).

{ ii) ✔}

{ iii)✔}

{After k-1 steps we know that current_count was increased every time a repeated value was found and at step k, there was a repeated value found AND current_count was increased by 1 once. THUS

{ i) ✔}

2. Current_count = max_count OR Current_count > max_count

Since current_count>= max_count in the last iteration and given that we would increase current_count by 1 in the first if statement of the loop. We know have that now current_count > max_count, thus after the loop execution we would enter the following if statement and then set our "current" most frequent to the last couple of elements counted. Thus, we output the kth element (which is equal to the kth-1 element) as our most frequent element of the array. (By the I.H, we can say that in this case for an array of size k we can output the correct most frequent element since it is the same as the k-1 element found)

{ ii) ✔}

{ iii)✔}

{After k-1 steps we know that current_count was increased every time a repeated value was found and at step k, there was a repeated value found AND current_count was increased by 1 -once. THUS

{ i) ✔}

Since we have proven that for all cases we got the most frequent value as a result, it is proven by strong form of Induction that we can calculate the most frequent element of an array of size k, 1<=k with this algorithm.

Cover Page for CPSC 413
Homework Assignment #1

Name: Juan Luis de Reiset Jimenez-Carbo

Course Section: CPSC-413 Tut 02

Collaborators:

Question 1: Steve Khanna, Daniel Sohn, Coskun Sahin

Question 2: Steve Khanna, Daniel Sohn, Coskun Sahin

Question 3: Steve Khanna, Daniel Sohn, Coskun Sahin

Question 4: Steve Khanna, Daniel Sohn, Coskun Sahin

Other Sources:

Question 1: https://www.wolframalpha.com/

Question 2: https://en.wikipedia.org/wiki/Biconnected_graph

Question 3: https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

Question 4: https://www.geeksforgeeks.org/frequent-element-array/, https://brilliant.org/wiki/sorting-algorithms/

Declaration:

I have written this assignment myself. I have not copied or used the notes of any other student.

Date/Signature: Juan Luis de Reiset 23/05/2019