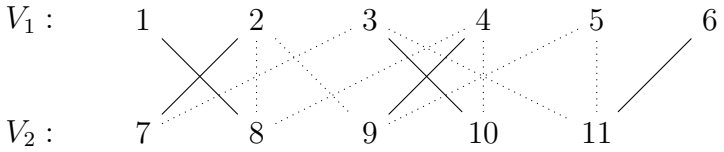
CSCI 3104 Spring 2018 Problem Set 10
Mohamed Al-Rasbi (07/25)

Problem 1

(15 pts total) A matching in a graph G is a subset $E_M \subseteq E(G)$ of edges such that each vertex touches at most one of the edges in E_M. Recall that a bipartite graph is a graph G on two sets of vertices, V_1 and V_2, such that every edge has one endpoint in V_1 and one endpoint in V_2. We sometimes write $G = (V_1, V_2; E)$ for this situation. For example:



The edges in the above example consist of all the lines, whether solid or dotted; the solid lines form a matching.
The bipartite maximum matching problem is to find a matching in a given bipartite graph G, which has the maximum number of edges among all matchings in G.

(a) *Prove that a maximum matching in a bipartite graph $G = (V_1, V_2; E)$ has size at most* $\min\{|V_1|, |V_2|\}$.

A matching in a bipartite graph means that we want to find a path such that each vertex has at most on edge. And in the case of bipartite graph, the edges are between vertices in V_1 and V_2. In other words, there cannot be an edge between vertices in the same set.
`Proof by contradiction:` Let's say that a maximum matching in a bipartite graph has size bigger that one of the two sets, V_1 or V_2, is true.
This means that the number of edges is bigger than the number of vertices in one of the sets, or both. And that cannot be true since a matching means that each vertex has 0 or 1 edge only.
If the number of edges is bigger than the number of vertices, then there exists at least one vertex that has 2 or more edges, and that cannot be true. Hence, the maximum matching in bipartite graph has size at most $\min\{|V_1|, |V_2|\}$.

(b) *Show how you can use an algorithm for max-flow to solve bipartite maximum matching on undirected simple bipartite graphs. That is, give an algorithm which, given an undirected simple bipartite graph $G = (V_1, V_2; E)$:*

(1) *constructs a directed, weighted graph G' (which need not be bipartite) with weights $w : E(G') \to \mathbb{R}$ as well as two vertices $s; t \in V(G')$,*

(2) *solves max-flow for $(G', w), s, t$, and*

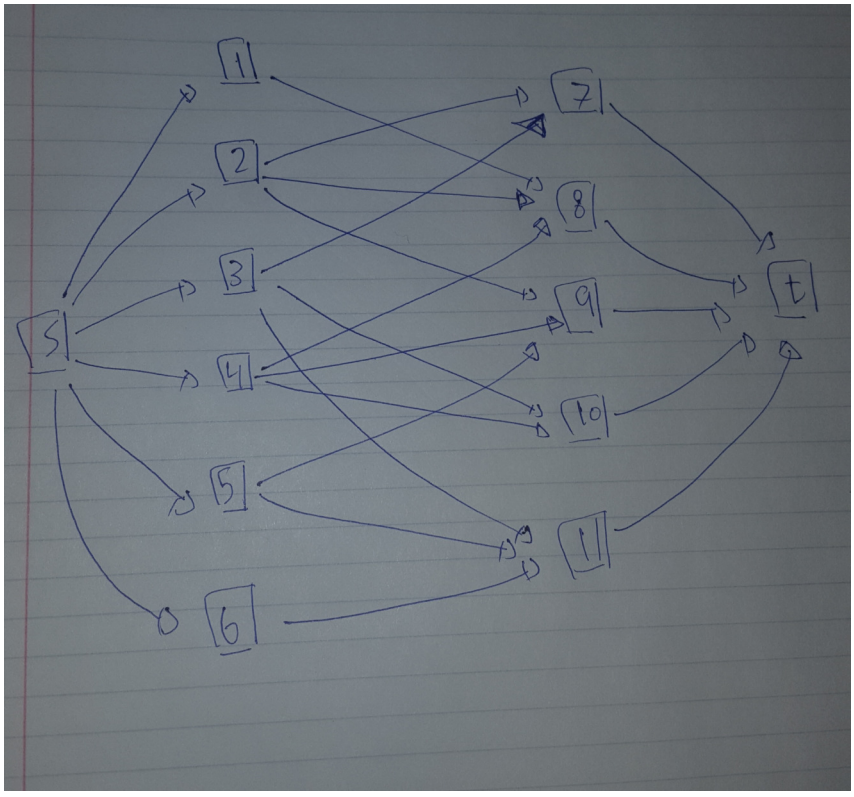
(3) *uses the solution for max-flow to find the maximum matching in G. Your algorithm may use any* `max-flow` *algorithm as a subroutine.*

`Algorithm to solve bipartite maximum matching:`
`1.` Build a flow network and add s, t vertices:
Connect s vertex to V_1 so there is a directed edge from s to each vertex in V_1. And connect t vertex to V_2 so there is a directed edge from each vertex in V_2 to t. Each edge will have weight 1 and will be directed from V_1 to V_2.

`2.` Find max-flow on the new graph using Ford-Fulkerson algorithm:
Find an augmenting path, computer the bottleneck capacity, then augment each edge and the total flow.

`3.` The max-flow is the maximum matching on the bipartite algorithm.

(c) *Show the weighted graph constructed by your algorithm on the example bipartite graph above.*



1. $s \to 1 \to 8 \to t$, flow = 1, can't use 1, 8
2. $s \to 2 \to 7 \to t$, flow = 2, can't use 2, 7
3. $s \to 3 \to 10 \to t$, flow = 3, can't use 3, 10
4. $s \to 4 \to 9 \to t$, flow = 4, can't use 4, 9
5. $s \to 5 \to 11 \to t$, flow = 5, can't use 5, 11

Vertex 6 won't have any edges since flow $= 5 = \min\{|V_1|, |V_2|\} = \min\{6, 5\}$

Problem 2

(20 pts total) In the review session for his Deep Wizarding class, Dumbledore reminds everyone that the logical definition of NP requires that the number of bits in the witness w is polynomial in the number of bits of the input n. That is, $|w| = poly(n)$. With a smile, he says that in beginner wizarding, witnesses are usually only logarithmic in size, i.e., $|w| = O(\log n)$.

(a) *Because you are a model student, Dumbledore asks you to prove, in front of the whole class, that any such property is in the complexity class P.*

The complexity class P is poly(n). If we modify the complexity class NP to be $O(\log n)$, then we know that complexity class is contained in poly(n).
The reason is because P always has complexity time less than or equal to NP, so this property, $O(\log n)$, is a polynomial reduction of P.

(b) *Well done, Dumbledore says. Now, explain why the logical definition of NP implies that any problem in NP can be solved by an exponential time algorithm.*

The logical definition of NP implies that the witness is poly(n). And since exponential time algorithms take more time the polynomial algorithms, NP can be solved by an exponential time algorithm.

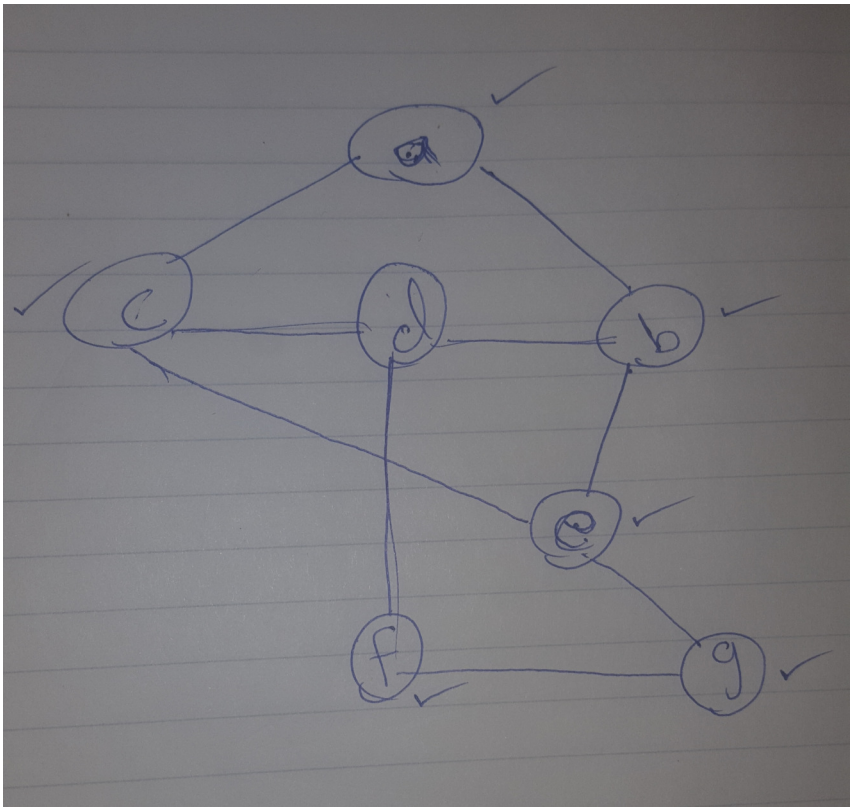
(c) *Dumbledore then asks the class: "So, is NP a good formalization of the notion of problems that can be solved by brute force? Discuss." Give arguments for both possible answers.*

Solving algorithms using brute force means that we have to try all combinations.
It is not a good idea to use brute force algorithms to solve an NP problems because those problem usually are hard to solve, but easy to prove a certain solution is true. Hence, it is going to take a lot of time to find all solutions for NP problem.
However, if NP problem is easy to solve, in other words NP = P, then it is not going to take a long time to find all the solutions. In that case it might be a good idea to use brute force.

Problem 3

(30 pts total) The Order of the Phoenix is trying to arrange to watch all the corridors in Hogwarts, to look out for any Death Eaters. Professor McGonagall has developed a new spell, Multi-Directional Sight, which allows a person to get a 360-degree view of where they are currently standing. Thus, if they are able to place a member of the Order at every intersection of hallways, they'll be able to monitor all hallways. In order not to spare any personnel, they want to place as few people as possible at intersections, while still being able to monitor every hallway. (And they really need to monitor every hallway, since Death Eaters could use Apparition to teleport into an arbitrary hallway in the middle of the school.) Call a subset S of intersections "safe," if, by placing a member of the Order at each intersection in S, every hallway is watched.

(a) *Formulate the above as an optimization problem on a graph. Argue that your formulation is an accurate reflection of the problem. In your formulation, show that the following problem is in NP: Given a graph G and an integer k, decide whether there a safe subset of size $\leq k$.*



1. In the graph above: all intersections have watchers except d.
2. The graph is an accurate reflection of the problem because we know that each watcher has a 360-degree view, which means that each vertex cover its adjacent

edges. If all adjacent edges of a certain vertex are covered by other vertices, then there is no need to place a watcher on that intersection (vertex).
3. This problem is in NP because we want to check if the number of watchers is at most equals k. In order to do that we need to apply an algorithm to find a safe subset of size $\leq$ k. The running time of such algorithm is polynomial, which is at worst case (put watchers on all intersections, from part b) is $\Theta(V^2)$. Since the running time is polynomial, then the problem is in P, which is a subset of NP.
Hence, the problem is in NP.

(b) *Consider the following greedy algorithm to find a safe subset:*

```
S = empty
mark all hallways unwatched
while there is an unwatched intersection
    pick any unwatched hallway; let u,v be its endpoints
    add u to S
    for all hallways h with u as one of its endpoints
        mark h watched
    end
end
```

Although this algorithm need not find the minimum number of people needed to cover all hallways, prove that it always outputs a safe set, and prove that it always runs in polynomial time.

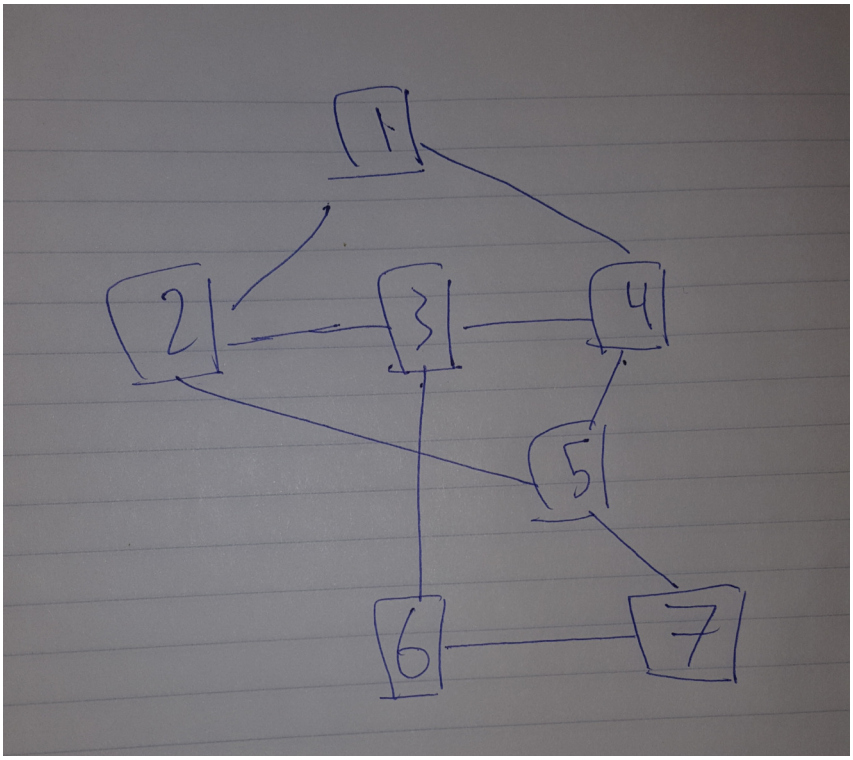
Proof by invariant:
Initialization: Before the outer loop, the loop invariant holds true since all hallways are unwatched, which results in an empty safe set.
Maintenance: Using proof by contradiction - let's assume that after the first iteration, the safe set will still be empty. That cannot be true since that the we will not enter the loop unless there is an unwatched intersection. If that is true, we will add a vertex to the safe set. Hence, after the first iteration the safe set will not be empty and we will be keep adding vertices until all of them are watched. Therefore, the loop invariant holds true.
Termination: Using direct proof - after the loop, all intersections (vertices) will be watched since we will break from the loop when all vertices are watched. And if all vertices are watched, then all of their adjacent edges are watched. Hence, we have a safe set.

Running time:
In the outer loop we will iterate through all vertices, which is V. In the inner loop we will iterate through all adjacent edges of each vertex, which is at most V-1 for each vertex.
Therefore, the running time is $O(V^2)$.

(c) *Note that, in order to be polynomial-time, an algorithm for this problem cannot simply try all possible subsets of intersections. Prove why not.*

It depends on the number of vertices to try all possible subsets of intersections. Therefore, to find all possible intersections in a graph with V vertices, it takes a running time of $O(2^V)$, which is exponential-time $\geq$ polynomial-time.

(d) *Give an example where the algorithm from (b) outputs a safe set that is strictly larger than the smallest one. In other words, give a graph G, give a list of vertices in the order in which they are picked by the algorithm, and a safe set in G which is strictly smaller than the safe set output by the algorithm.*



`List of vertices picked by the algorithm:` [1, 2, 4, 5, 6, 7]
`List of vertices that is minimum:` [1, 3, 5, 6, 7]

(e) *Consider the following algorithm:*

```
S = empty
mark all hallways unwatched
while there is an unwatched hallway
    pick any unwatched hallway; let u,v be its endpoints
    add u,v to S
    for all hallways h with u or v one of their endpoints
```

```
        mark h watched
    end
end
```

Prove that this algorithm always returns a safe set, and runs in polynomial time.

`Proof by loop invariant:`
`Initialization:` Before the loop, all hallways are unwatched. The loop invariant holds true since S is empty, which means no watchers are added to the safe set.
`Maintenance:` Let's assume, by contradiction that after the first iteration of the outer loop, the safe set S will remain empty. That cannot be true because we will not iterate through the loop unless we find one or more unwatched edges. And if we do, that means there are two vertices will be added to the set. Hence, after the first iteration, two vertices (watchers) will be added to the safe set. Therefore, the loop invariant holds true.
`Termination:` Let's assume that after the loop we will not have a safe set. That is not true since the loop's condition is if there is unwatched hallway, we will make it watched. Hence, after the loop all hallways will be marked watched, and the endpoints of every hallway will be added to the safe set. Therefore, at the end of the loop, all hallways will be watched.
Since all hallways are watched, then the algorithm return a safe set.

`Running time:`
The outer loop iterates through all hallways and checks if there is an unwatched hallway, and that takes E iterations. The inner loop iterates through the adjacent vertices of u and v, and that takes at most 2(V-1) iterations.
Therefore, the running time is $O(V \cdot E)$.

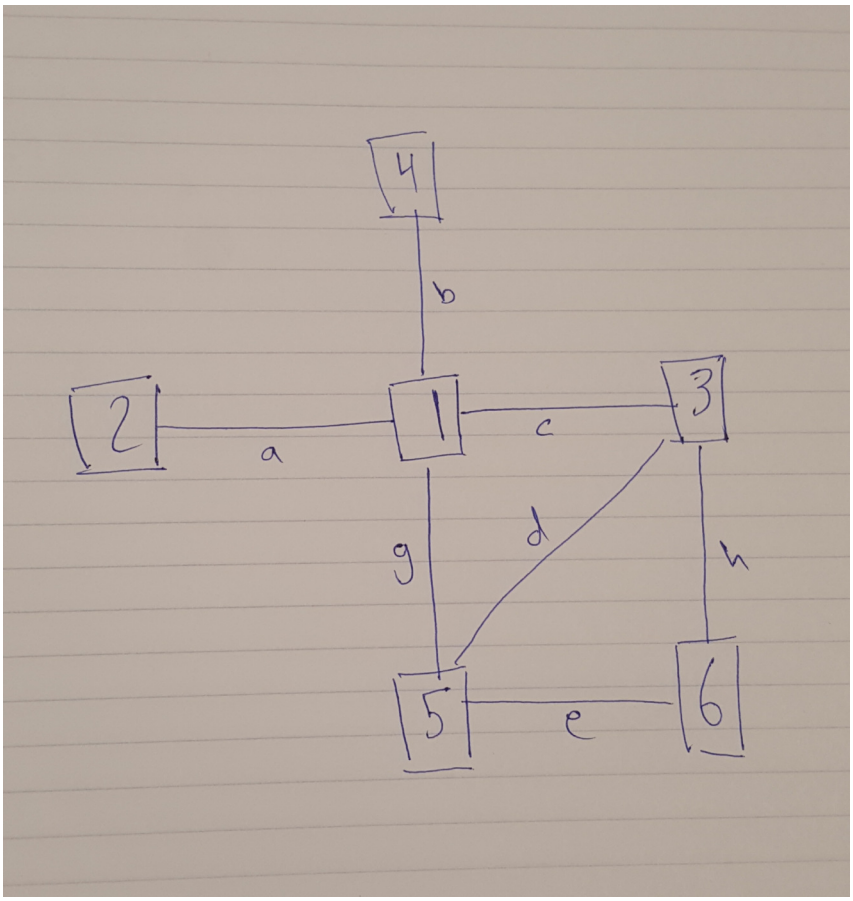
(f) *In any safe set of intersections, each hallway is watched by at least one member of the Order. Use this to show that the algorithm from (e) always outputs a safe set whose size is no more than twice the size of the smallest safe set. Note: you don't need to know what the smallest safe set is to prove this! All you need is the fact stated here.*
This is called a "2-approximation algorithm," because it is guaranteed to output a solution that is no worse than a factor of 2 times an optimal solution.

The biggest range between the algorithm in part 3e and the smallest safe set (at least on watcher on each edge), is (Worst case of 3e algorithm / Best case of smallest safe set).
- Best case of smallest safe set in any graph is that each edge has exactly one watcher (k watchers).
- Worst case of 3e algorithm is that each edge has exactly two watchers (2k watchers).

Therefore, (Worst case of 3e algorithm / Best case of smallest safe set) $= \frac{2k}{k} = 2$.
Which means that the 3e algorithm is guaranteed to output a solution that is no

worse than a factor 2 time of an optimal solution.

(g) *Does the algorithm from (b) always produce a safe set no bigger than that produced by the algorithm in (e)? If so, give a proof; if not, give a counterexample. A counterexample here consists of a graph, and for each algorithm, the list of vertices it chooses in the order it chooses them, such that the safe set output by algorithm (b) is at least as large as the safe set output by algorithm (e). If you are unable to give either a proof or a counterexample, then for partial credit give a plausible intuitive argument for your answer.*



`3b algorithm`: This algorithm always mark all vertices as watched. Hence, the number of watchers is 6.
`3e algorithm`: This algorithm will choose a random unwatched hallway, append end points to the safe set, then mark the end points' adjacent edges as watched, and repeat.
In the example above:
Choose g: mark g as watched, append 1 and 5, mark a, b, c, d, e as watched.
Choose h: mark h as watched, append 3 and 6, no more unwatched edges
Number of watched is 4.

Therefore, this is a counterexample that number of watchers in 3b > 3e.

(h) *Compare the greedy algorithm from (e) with the greedy algorithm from (b). Show which runs faster asymptotically? Which of these two algorithms would you rather use to solve the Order of the Phoenix's problem and why?*

`3b algorithm`: The asymptotic running time is $O(V^2)$.
`3e algorithm`: The asymptotic running time is $O(V \cdot E)$. Assuming we will iterate through all edges to find unwatched edges.
Therefore, 3b runs faster asymptotically than 3e because usually the number of edges (E) is bigger than the number of vertices (V). Also, the 3e algorithm finds less watchers than 3b algorithm, hence, it makes since that it takes a long time.

Even though 3b algorithm runs faster than 3e algorithm, I would choose 3e algorithm to solve the Order of Phoenix's problem because of the following reasons:
1. Order of Phoenix's problem tries to minimize the number of watchers. 3b algorithm always places watchers at all intersections, however, 3e algorithm find less watchers than 3b algorithm, which is what we want.
2. The asymptotic running time difference between the two algorithms is not that significant since both of them run in a polynomial time.

(i) *This problem is, in fact, NP-complete. Why does the 2-approximation polynomial-time algorithm from (e) not show that P=NP?*[1]

We now know that this algorithm is NP-complete, but we did not know that when we proved the polynomial running time of 3e. Hence, 3e showed that the set of solutions are in P, even though that we know it is NP-complete, which is a subset of NP. Therefore, 3e did not show that P=NP.

[1]Interestingly, it is known that if there were a 1.3606...-approximation algorithm for this problem in polynomial time, then it would follow that P=NP, but that is a very nontrivial theorem. Under a standard complexity-theoretic assumption, even if there were a 1.99999-approximation algorithm in polynomial time, it would follow that P=NP, but this assumption remains a conjecture, and opinion in the research community is divided on whether this conjecture is true or false. We will provide references to these results after the problem set has been handed in.

Problem 4

(20 pts extra credit) Every young wizard learns the classic NP-complete problem of determining whether some unweighted, undirected graph $G = (V, E)$ contains a simple path of length at least k (where both G and k are part of the input to the problem), known as the Longest Path Problem. Recall that a simple path is a path $(v_1, v_2, ..., v_\ell)$ where each (v_i, v_{i+1}) in the path is an edge, and all the v_i are distinct; its length is $\ell - 1$ (=the number of edges in the path).

(a) *Ginny Weasley is working on a particularly tricky instance of this problem for her Witchcraft and Algorithms class, and she believes she has written down a "witness" for a particular input (G, k) in the form of a path P on its vertices. Explain how she should verify in polynomial time whether P is or is not simple path of length k. (And hence, demonstrate that the problem of Longest Path is in the complexity class NP.)*

We know that a simple path is a path that start from vertex a and end at vertex b, such that each vertex in the path is distinct (which means that each vertex is only traversed once).
Knowing that information, we can say that a simple path has the property $E_{tot} = V_{tot} - 1$, where V and E are the vertices and edges in the path, respectively.
Hence, these are the steps to check if the path P is a simple path:
1. Initialize all vertices in the graph as "not visited".
2. Go through the path, if a vertex is not visited, then mark it as visited.
3. Count the number of visited vertices as we go through the path.
4. Count the number of edges as we go through the path (If we go from v_1 to v_2 and then from v_2 to v_1, that is +2 edges).
5. Finally, we should compare the total number of vertices in the graph to the total number of edges, if $E_{tot} = V_{tot} - 1$, then we have a simple path. Otherwise, we do not.

The running time complexity of this problem is $O(V + E)$ because at worst we will iterate through all vertices and all edges of the graph.
This is a linear running time, which is considered polynomial. Hence, we verified if a solution (graph P) is a correct solution to our problem (finding a simple path in a graph) in a polynomial time. Therefore, this problem is in the complexity class NP.

(b) *For the final exam in Ginny's class, each student must visit the Oracle's Well in the Forbidden Forest. For every bronze Knut a young wizard tosses into the Well, the Oracle will give a yes or no response as to whether, given an arbitrary graph G and an integer k, G contains a simple path of length k. Ginny is given an arbitrary graph G and must find the longest simple path in G. First, she realizes it would be useful to determine the length of the longest simple path. Describe an algorithm that will allow Ginny to use the Oracle to find the length of the longest simple path in G*

by asking it a series of questions, each involving a modified version of the original graph G and a number k. Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Longest Path decision problem in polynomial time, we can solve its optimization problem as well.)

The algorithm that we can use to find the longest simple path is as follows:

```
findLongestSimplePath(G, k):
    maxLength = 0
    maxPath = []
    vertices = [all vertices in G]
    for vertex in vertices:
        #find the longest simple path using DFS
        #Read how it works below
        p = DFS(G, vertex)

        if p.length > maxLength:
            maxLength = p.length
            maxPath = p.path

    if maxLength < k:
        return []

    return maxPath
```

This algorithm runs DFS on all vertices, starting from a certain vertex to find all possible simple paths.
DFS returns the longest simple path (read about it below). Then, we compare if the length of the longest simple path returned is >= k. If so, return the path.

`NOTE: How DFS works`
The DFS function will start from the passed vertex to find all simple paths. However, I want to note that it is going to update the distances of all vertices from the starting vertex. The important thing is: while we iterate trough a certain vertex's adjacent vertices and find a visited vertex, it will check if it can update the distance of that visited vertex, so that the new distance is bigger than the new one. If the distance is updated, then the parent will also be updated.

`Therefore, this is what DFS function returns:`
1. Maximum length (the vertex with the maximum distance)
2. Path from starting vertex to end vertex depending on the parent of the end vertex.

`Running time:`
The running time of DFS if $O(V + E)$. Since we are running DFS for V times,

the running time of this algorithm is $O(V(V+E)) = O(V^2 + VE)$, which is in polynomial time.
Therefore, this problem is in complexity time class P.

(c) *Next, once she knows the length ℓ of the longest simple path in G, Ginny must use the Oracle to actually find a path of length ℓ. Describe an algorithm that will allow Ginny to use the Oracle to find the longest simple path in G by asking it a series of questions, each involving a modified version of the original graph G and a number k of her choosing (for each question she can ask about a different graph G and a different number k). Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Longest Path decision problem in polynomial time, we can solve its search problem as well.)*

I will use the same algorithm in (3b) since the distances are updated in DFC, however, I will make the following changes to find a simple path of length l:
1. DFS will return a path of length l, otherwise, return -1.
2. Store the path and length in global variables, then return the path.

```
findLongestSimplePath(G, l):
    length = 0
    path = 0
    vertices = [all vertices in G]
    for vertex in vertices:
        #find the simple path of length l using DFS
        p = DFS(G, vertex, l)

        if p.length == l:
            length = p.length
            path = p.path
            break from loop

    if length != l:
        return []

    return path
```

This algorithm also run in polynomial time, which means that the complexity class of a search function is in class P.

Problem 5

(20 pts extra credit) Recall that the `MergeSort` *algorithm (Chapter 2.3 of CLRS) is a sorting algorithm that takes* $\Theta(n \log n)$ *time and* $\Theta(n)$ *space. In this problem, you will implement and instrument* `MergeSort`*, then perform a numerical experiment that verifies this asymptotic analysis. There are two functions and one experiment to do this.*

(i) *`MergeSort(A, n)` takes as input an unordered array A, of length n, and returns both an in-place sorted version of A and a count t of the number of atomic operations performed by* `MergeSort`*.*

(ii) *`randomArray(n)` takes as input an integer n and returns an array A such that for each $0 \leq i < n$, $A[i]$ is a uniformly random integer between 1 and n. (It is okay if A is a random permutation of the first n positive integers; see the end of Chapter 5.3.).*

(a) *From scratch, implement the functions* `MergeSort` *and* `randomArray`*. You may not use any library functions that make their implementation trivial. You may use a library function that implements a pseudorandom number generator in order to implement* `randomArray`*.*
Submit a paragraph that explains how you instrumented `MergeSort`*, i.e., explain which operations you counted and why these are the correct ones to count.*

```
 1 def sortAndMerge(leftArr, rightArr, t):
 2         newArr = []
 3         i = 0
 4         j = 0
 5         finished = False
 6         leftLen = len(leftArr)
 7         rightLen = len(rightArr)
 8         t += 8
 9
10         while(not finished):
11                 if(i == leftLen and j == rightLen):
12                         finished = True
13                         t += 3
14                 elif(i == leftLen):
15                         newArr.append(rightArr[j])
16                         j += 1
17                         t += 7
18                 elif(j == rightLen):
19                         newArr.append(leftArr[i])
```

```
20                          i += 1
21                          t += 7
22                  elif(leftArr[i] <= rightArr[j]):
23                          newArr.append(leftArr[i])
24                          i += 1
25                          t += 8
26                  else:
27                          newArr.append(rightArr[j])
28                          j += 1
29                          t += 6
30                  t += 1
31
32          return (newArr, t)
33
34  def mergeSort(A, n, t):
35          if(n == 2):
36                  if(A[0] > A[1]):
37                          A[0], A[1] = A[1], A[0]
38                          t += 6
39                  t += 3
40                  t += 1
41                  return (A, t)
42          if(n == 1):
43                  t += 1
44                  return (A, t)
45
46
47          mid = n / 2
48          t += 2
49
50          leftArr, t = mergeSort(A[0:mid], mid, t)
51          t += 3
52          # print leftArr, t
53          rightArr, t = mergeSort(A[mid:n], n - mid, t)
54          t += 4
55          # print rightArr, t
56          A, t = sortAndMerge(leftArr, rightArr, t)
57          t += 2
58          print A, t
59          return A, t
60
61
62  import random
63  def randomArray(n):
64          arr = []
```

```
65          for i in range(n):
66                  arr.append(random.randint(1, n))
67
68          return arr
```

This is a list of line numbers and the count for merge sort:
`lines(2 - 7)`: 6 variable assignments and 2 length functions, t += 8
`lines(11-13)`: 2 comparisons and 1 assignments, t += 3
`lines(14-17)`: 1 comparison, 2 additions, 2 assignments, 1 append, 1 array access, t += 7
`lines(18-21)`: 1 comparison, 2 additions, 2 assignments, 1 append, 1 array access, t += 7
`lines(22-25)`: 2 comparisons, 2 additions, 2 assignments, 1 append, 1 array access, t += 8
`lines(27-29)`: 2 additions, 2 assignments, 1 append, 1 array access, t += 6
`lines(30)`: one comparison (from the while loop condition)
`lines(35-40)`: 2 comparisons, 6 array access, 2 assignments, t += 10
`lines(42-44)`: 1 comparison, t += 1
`lines(47)`: 1 assignment, 1 division, t += 2
`lines(50)`: 2 assignments, 1 array access, t += 3
`lines(53)`: 2 assignments, 1 array access, 1 subtraction, t += 4
`lines(56)`: 2 assignments, t += 2

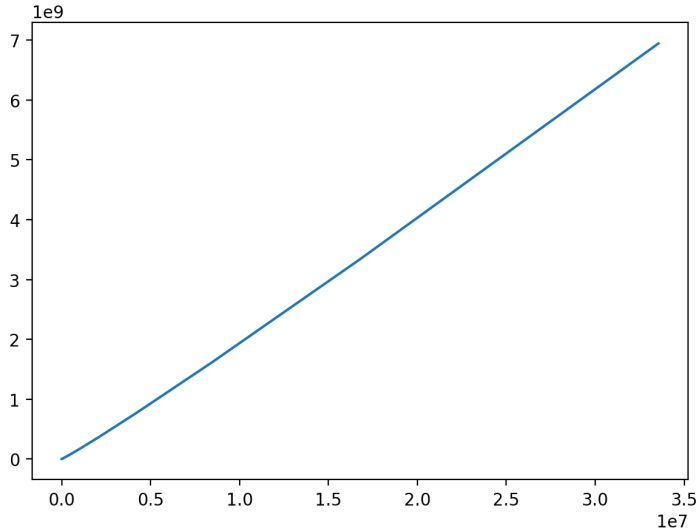
(b) *For each of* $n = \{2^4, 2^5, ..., 2^{26}, 2^{27}\}$*, run* ***`MergeSort(randomArray(n),n)`*** *five times and record the tuple* $(n, \langle t \rangle)$*, where* $\langle t \rangle$ *is the average number of operations your function counted over the five repetitions. Use whatever software you like to make a line plot of these 24 data points; overlay on your data a function of the form* $T(n) = An \lg n$*, where you choose the constant* A *so that the function is close to your data.*
Hint: To increase the aesthetics, use a log-log plot.

I wrote this code in python to find the x-axis (sizes from 4 to 25) and the y-axis (counts for each size):

```
for i in range(4, 26):
        averageCount = 0
        for j in range(5):
                t = 0
                A = []
                A, t = mergeSort(randomArray(2**i), 2**i, t)
                averageCount += t

        averageCount /= 5
        countsArr.append(averageCount)
```

Then I used matplotlib in python to plot:



The plot doesn't show the desired (or expected) line since the numbers increase exponentially.

These are the values:
`counts (y) = [598, 1495, 3531, 8063, 18150, 40485, 89049, 194593, 421855, 909235, 1949198, 4162070, 8848996, 18743463, 39584209, 83360124, 175114852, 366996911, 767556446, 1602222395, 3338654877, 6945791692]`
`sizes (x) = [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 20971]`

I used this quick script to find the value of A:

```
counts = [598, 1495, 3531, 8063, 18150, 40485, 89049, 194593, 421855,
          909235, 1949198, 4162070, 8848996, 18743463, 39584209, 83360124,
          175114852, 366996911, 767556446, 1602222395, 3338654877, 6945791692]
sizes = [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768,
         65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432]

import math
tmp = []
for i in range(22):
        x = counts[i] / (sizes[i] * math.log10(sizes[i]))
        tmp.append(x)
average = sum(tmp)
average /= 22
print average
```

`A = 28.6`

Worked with Abdullah Bajkhaif, Mahmoud Almansori, Omar Mohammed