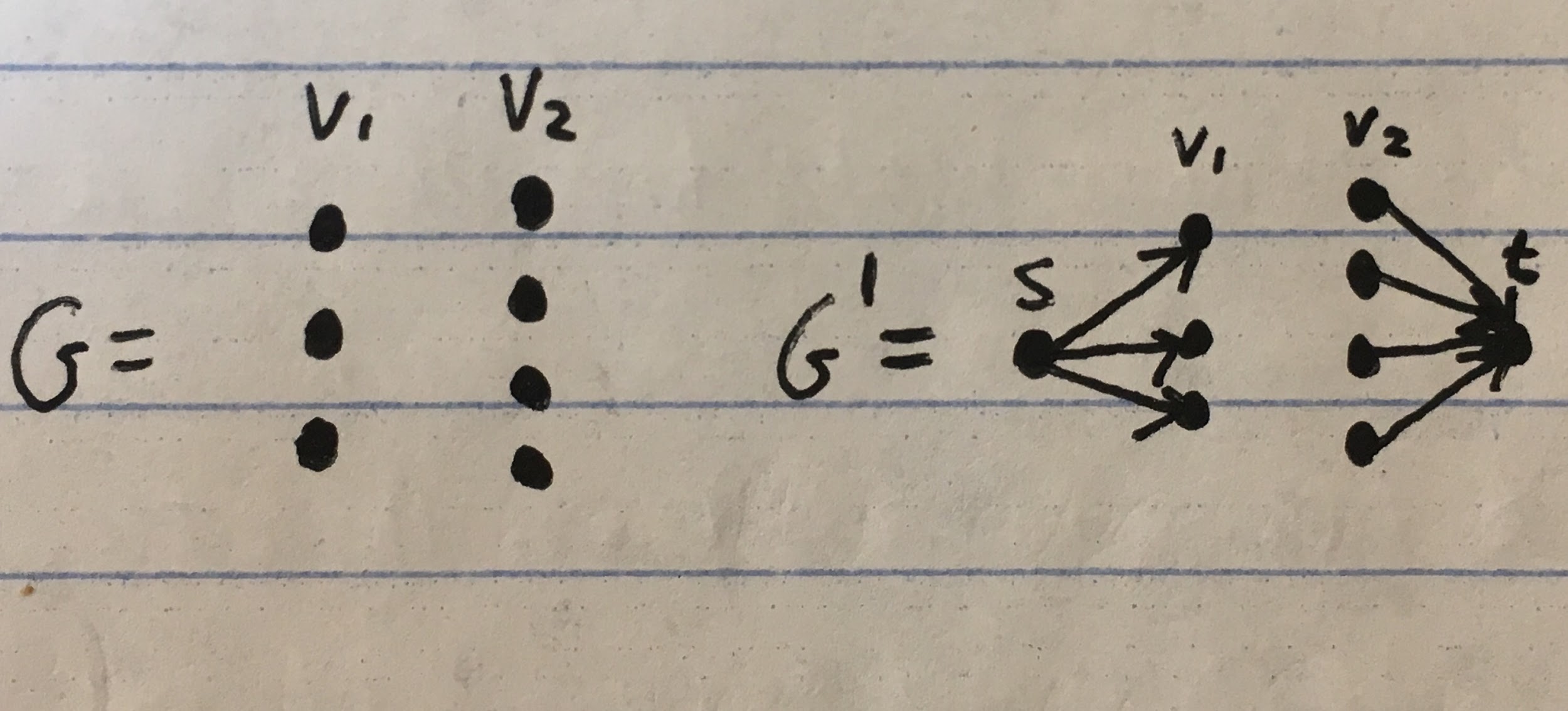
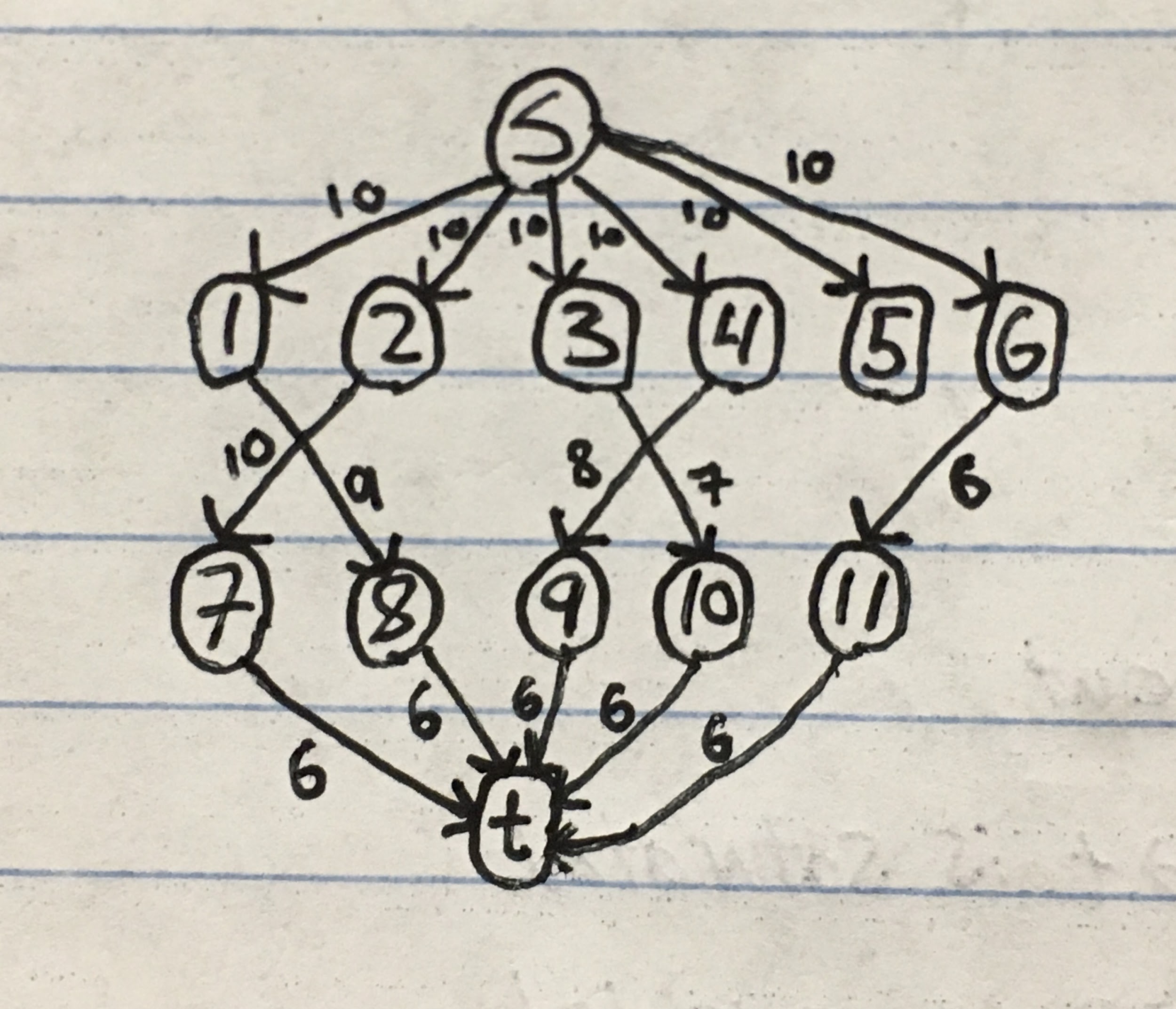
1a) The maximum matching in a bipartite graph G = (V1, V2; E) has size at most min{|V1|, |V2|} because, by the nature of the bipartite maximum matching problem, a vertice from V1 is joined with a vertice in V2 each time an edge is added. Once the amount of edges added equals the size of min{|V1|, |V2|} every vertice in min{|V1|, |V2|} will be connected to exactly one edge. Adding additional edges to the graph would require connecting an edge to a vertice in min{|V1|, |V2|} that is already connected to one, and since we are working with a bipartite graph that would break the rules defining our graph. Therefore we are limited to adding an amount of edges equal to the size of the smaller set of vertices when constructing a bipartite maximum matching.

1b) An algorithm for max flow to solve the the bipartite maximum matching on graph G would first construct a graph G’ as a copy of G and append a source node s and sink node t. Node s would be connected to all the vertices in V1 and node t would be connected to all vertices in V2. The direction of all edges would be pointing from s to t allowing flow in the direction of the sink node. The weights will be arbitrary, however they are needed in order to run the max flow algorithm (see fig.1).



Once G’ is constructed the Ford Fulkerson maximum flow algorithm can be used as a subroutine, however it will not be used on all edges in the graph, the subroutine will only run until the amount of edges equal to the amount of edges in min{|V1|, |V2|} have been handled. This is because we do not want to exceed the limit of the amount of edges in the maximum matching in G.

1c) 

2a) It can be proven that for a witness w where |w| = *O*(log*n*) its corresponding problem would be in the complexity class P because the amount of steps to produce an answer for any problem’s input string is at most Cnk (C and k as constants, and n as the length of the input string). In the worst case it is clear the problem is solved in polynomial time; by definition a problem of this form with a logarithmic sized witness would be in the complexity class P.

2b) The logical definition of NP implies that any NP problem can be solved by an exponential time algorithm because such an algorithm would simply be able to check all possible truth assignments for the NP problem. The definition of NP states that a problem is in NP if it's decided by a nondeterministic Turing machine to run in polynomial time. The largest possible amount of different branches of a nondeterministic problem is 2*O(n^k)*, so the absolute worst case execution time would be 2*O(n^k)* multiplied by a polynomial running time, still within the bounds of being solved in exponential time by an algorithm.

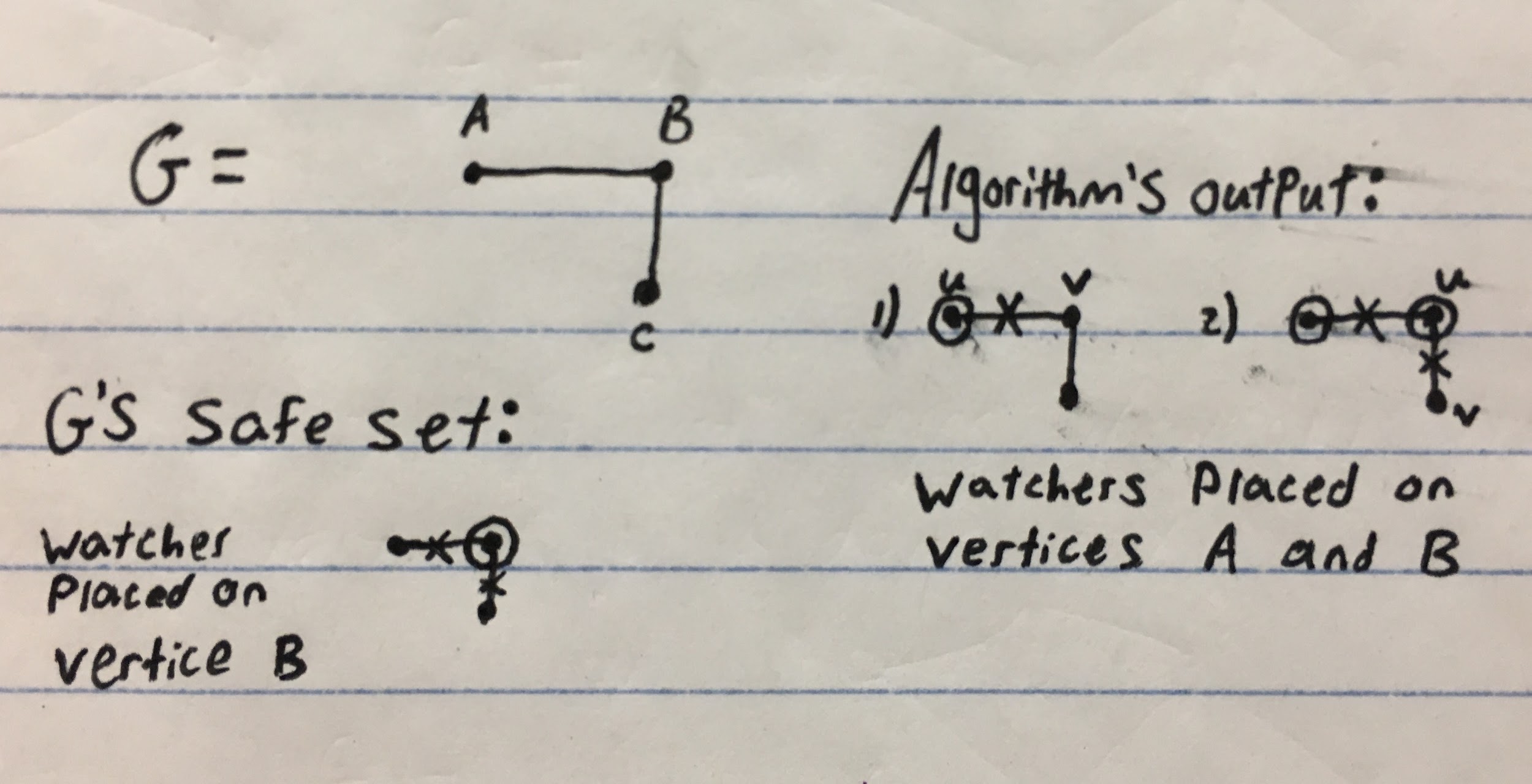
2c) It can be argued that NP is not a proper formalization of that notion because Exponential Time encompasses the problems solvable by brute force, therefore using NP as that formalization is not accurate.

It can also be argued that NP is a proper formalization for that notion because, by the definition of NP, its solvable problems are a subset of problems solved in exponential time, therefore the problems in NP can be solved by brute force.

3a) The process of finding a safe subset can be formulated as an optimization problem; the optimal solution is the least amount of watchers possible. In order to find the least amount of watchers a greedy algorithm can be implemented which adds a watcher to the vertice with the highest degree, or in other words, the intersection of the most unwatched hallways. Once a watcher has been placed the algorithm should set that intersection and its adjacent hallways as “watched” before continuing. This optimization problem is in NP because its solution can be calculated in polynomial time by searching for hallways that have an inefficient amount of watchers.

3b) The given algorithm will always output a safe set because its while loop is set to run until there are no unwatched hallways, and the algorithm will not mark a hallway as “watched” unless it has a watcher on an adjacent vertice, e.g. there will be no false positives. The algorithm certainly runs in polynomial time because it consists of nested loops which result in an asymptotic running time of O(n2).

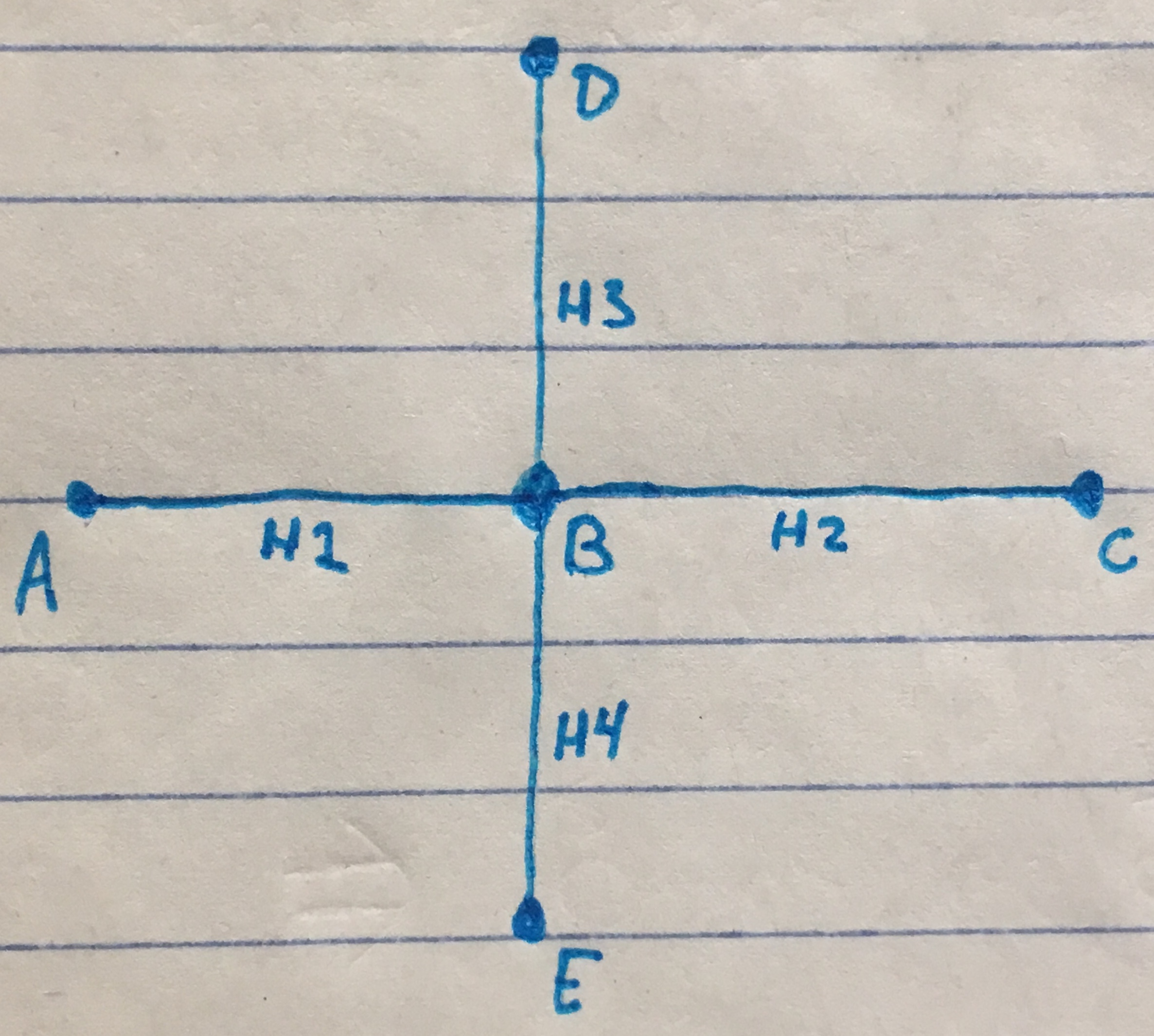
3c) By trying all the possible subsets of intersections the algorithm would run in exponential time (O(2n) n being the number of vertices), which is greater than polynomial time.

3d) 

3e) For the same reason as the algorithm in (3b) the given algorithm will always return a safe set; the while loop ensure that no hallways will be left unwatched and the process that marks hallways as “watched” will never incorrectly mark a hallway; every watched hallway has an adjacent watcher. This algorithm has slightly more atomic operations however it still has time complexity of O(n2) due to its nested loops.

3f) The safe set of the given algorithm will never be more than twice the size of the minimum safe set because it marks the hallways attached to both sides of a given edge as “watched”. Since each vertice in the graph will have at least one watched hallway attached the possible inefficiencies will never amount to a number than twice the size of the minimum safe set; the edges are marked before inefficient duplicate watchers are added.

3g) Counterexample showing that the algorithm from (3b) can produce a safe set bigger than that of (3e)



|  |  |
| --- | --- |
| Algorithm 3b | Algorithm 3e |
| u = A, v = B  Add A to S  H1 is marked as watched  u = D, v = B  Add D to S  H3 marked as watched  u = E, v = B  Add E to S  H4 marked as watched  u = C, v = B  Add C to S  H2 marked as watched | u = A, v = B  Add A and B to S  H1, H2, H3, and H4 are marked as watched |

In the above counterexample the algorithm in 3B resulted in a safe set with four watchers, whereas the algorithm in 3e resulted in a safe set with only two watchers.

3h) The algorithms in (3b) and (3e) have the same asymptotic running time of O(n2). The additional atomic operation in the inner loop of algorithm (3e) does not change its asymptotic running time. I would rather use the algorithm in (3e) to solve the Order’s problem because its ability to check the adjacent edges of each start and end node results in more efficient safe sets.

3i) The polynomial time algorithm in (3e) does not show that P=NP because it does not find the most efficient solution. Although a witness can be verified in polynomial time, we do not have an algorithm that can find a true solution to the optimization problem, thus the problem remains in NP-Complete.

4a) To verify in polynomial time whether P is or is not a simple path with length ≥ k Ginny’s algorithm should loop through each vertice of P while checking the current vertice against the rest of the vertices in P. This process can be done with two nested loops, resulting in a running time of O(n2), and effectively ensure that P is composed of distinct vertices and is of the necessary length.

4b) Ginny can find the length of the longest simple path in G by mimicking a binary search with her questions to the well. She can start by asking the well if there is a path of length x, with the number x being half of the total amount of vertices. If the response is “No” then she will divide x by two and ask again. If the response is “Yes” the variable x can be incremented by half of itself and Ginny can ask again. This search has a worst case time of log(n), so the cost of Knuts will always be less than a number that grows polynomially as a function of the number of vertices in G.

4c)I don’t know

5)

def randomArray(n):

A = []

for i in range(n):

A.append(np.random.uniform(1, n))

return A

def MergeSort(A, n):

t = 0;

if n > 1:

n = n//2 #floor division // rounds down after division

left = A[:n]

right = A[n:]

t+=4

mergeSort(left)

mergeSort(right)

i=0

j=0

k=0

while i < len(left) and j < len(right):

t+=2

if left[i] < left[j]:

A[k] = left[i]

i = i + 1

t+=2

else:

A[k]=right[j]

j = j + 1

t+=2

k = k + 1

t++

while i < len(left):

A[k]=left[i]

i = i + 1

k = k + 1

t+=4

while j < len(right):

A[k]=right[j]

j = j + 1

k = k + 1

t+=4

return t

(a) I implemented merge sort by recursively dividing the input array into two halves and sorting the subarrays. Afterwards the subarrays are merged together. While counting atomic operations I used a counter variable *t* and incremented it whenever the program would alter resources in memory, such as comparisons, swaps, and reading or writing data.

(b)