# Question 1

The following is a counterexample to show that Alice’s claim is not true. Assume given the following undirected graph :

A picture containing timeline

Description automatically generated

Now, attempt to carry out her claim. The independent set of maximum size in for this graph, call it , is the set of vertices coloured in red, namely . Now, let represent the vertex labelled , making it . Then, construct the graph , where is with and the edges incident on removed:

A picture containing timeline

Description automatically generated

The claim states that for , is an independent set of maximum size. In this case, the independent set that is constructed is . However, although is a valid independent set, it is **not an independent set of maximum size**. A larger independent set of is :

Chart

Description automatically generated with medium confidence

In this example, using Alice’s claim, the independent set generated for is of size , even though there exists a larger independent set of size shown in blue. Therefore, since is not an independent set of maximum size in , then this proves that Alice’s claim is not true.

# Question 2

1. Yes, is correct. Proof by exhaustion (case analysis):

For every invocation of there exist only two cases: either or .

Case 1: .

This is the base case of the recursive algorithm, and this is correct because computing the square of two **equivalent** numbers is equal to multiplying them together, or , as shown on Line (1). Since every first invocation is , the base case of this outermost call is , namely , which is indeed . Consequently, it is also true in the general case, for any where .

Case 2: .

This is the recursive case of the algorithm: it utilizes a divide-and-conquer approach and summation method to compute . For each invocation where , computes the left half of on Line (4), computes the right half on Line (5), and then they are added together to get on Line (6). Now, just need to prove that and are correct invocations for the left and right halves, respectively. First, observe that calculates the midpoint between and , namely . Thus, the following relation holds:

This, together with the base case, does indeed cover the entire range between and inclusive, so all the values are accounted for. Thus, the following recurrence holds:

Lastly, notice that for the and invocations of , the difference between and decreases each time, so it does not overlap and correctly approaches its termination criteria (base case). More specifically, computes , and computes , covering the range of values without overlap.

Therefore, since it covers the entire range between and and avoids overlap, then is indeed correct to compute .

1. is a polynomial-time algorithm.

Although it seemingly performs a divide-and-conquer strategy to compute the recursive case, it still has to cover the entire range of values between and to calculate . To start, Line (1), Line (3), and Line (6) are constant-time operations. Line (4) performs invocations to until , and then it will return due to the base case, thus covering the range from to . Similarly, Line (5) invokes until , covering the range from to . Since ranges and are covered, then together is covered. Lastly, since it is invoked as , where is a positive integer input, and it is shown in (a) that there is no overlap, then the time-efficiency is , deeming it indeed as a polynomial-time algorithm.

1. The space-efficiency of is , where is the positive integer input.

Since this algorithm is recursive, its space is characterized by the use of the recursive call stack, so in this case, it is represented by the depth of its recursion tree. For example, the following is the recursion tree for the input , invoking :

Diagram

Description automatically generated

In this example, the depth of this recursion tree is , which is indeed upper bounded by . Consequently, this also holds in the general case, where the depth of any recursion tree is bounded by . In addition, the input of interest for is and , where . Thus, the number of splits in the tree is defined by , which can be simplified down to . Therefore, the space-efficiency of is indeed .

# Question 3

# Question 4

# Question 5

# Question 6