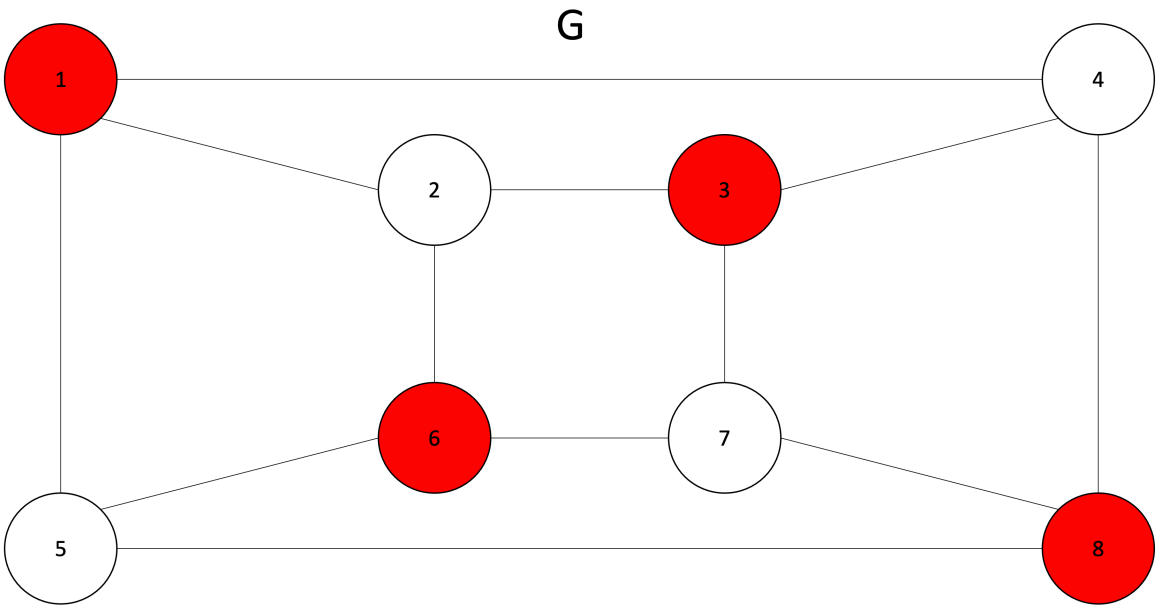
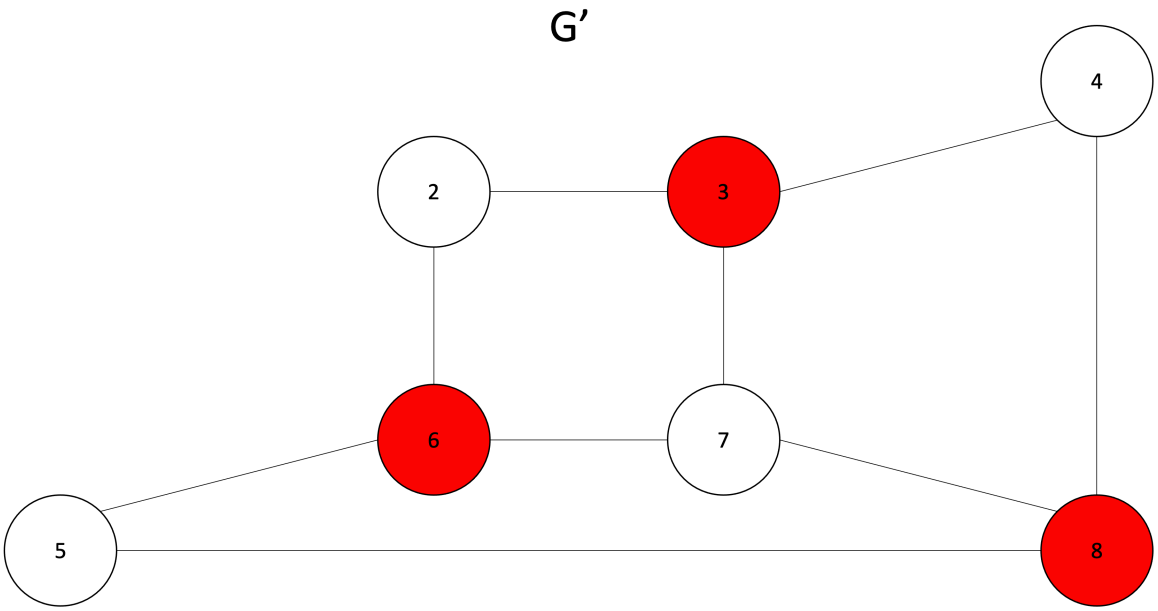
Exam

QUESTION 1

The following is a counterexample to show that Alice's claim is not true. Assume given the following undirected graph $G = \langle V, E \rangle$:

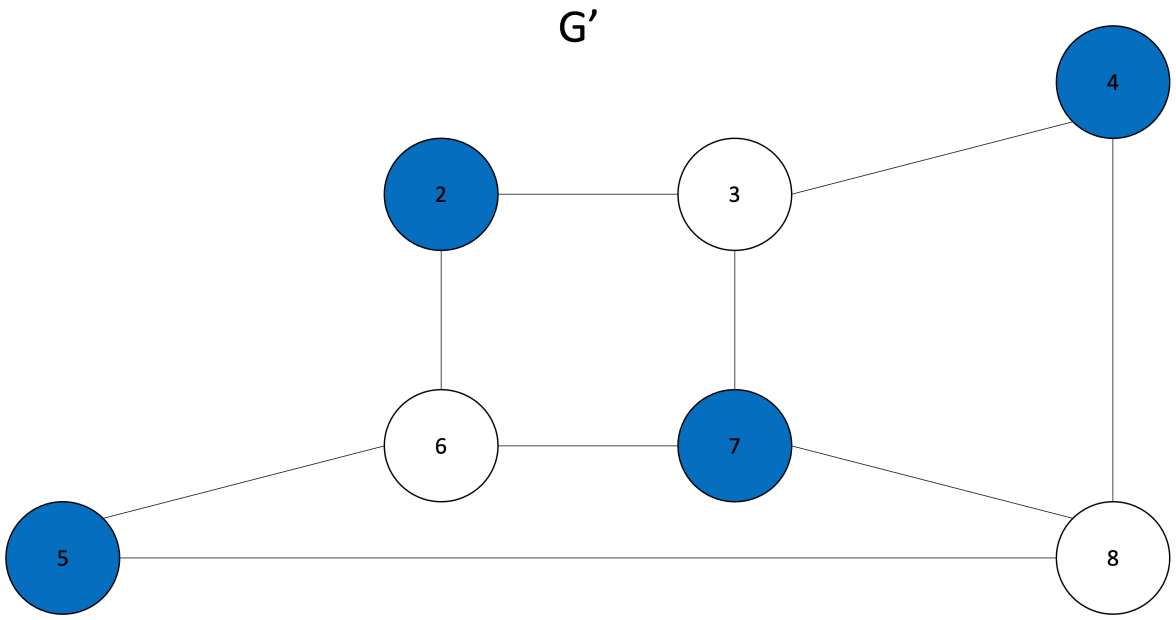


Now, attempt to carry out her claim. The independent set of maximum size in G for this graph, call it I, is the set of vertices coloured in red, namely $I = \{1, 3, 6, 8\}$. Now, let u represent the vertex labelled 1, making it $u \in I$. Then, construct the graph G', where G' is G with u and the edges incident on u removed:



Exam

The claim states that for G', $I \setminus \{u\}$ is an independent set of maximum size. In this case, the independent set that is constructed is $\{3, 6, 8\}$. However, this is **not an independent set of maximum size**. A larger independent set of G' is $\{2, 4, 5, 7\}$:



In this example, using Alice's claim, the independent set generated for G' is of size 3, but there exists a larger independent set of size 4 shown in blue. Therefore, since $I \setminus \{u\}$ is not an independent set of maximum size in G', then this proves that Alice's claim is not true.

Exam

QUESTION 2

(a) Yes, $SqSum$ is correct. Proof by exhaustion (case analysis):

For every invocation of $SqSum(p, q)$ there exist only two cases: either $p = q$ or $p \neq q$.

Case 1: $p = q$.

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 $p^2 = p \times p$, as shown on Line (1). Since every first invocation is $SqSum(1, n)$, the base case of this outermost call is $SqSum(1, 1)$, namely 1^2, which is indeed 1×1. Consequently, it is also true in the general case, for any $SqSum(p, q)$ where $p = q$.

Case 2: $p \neq q$.

This is the recursive case of the algorithm: it utilizes a divide-and-conquer approach and summation method to compute $SqSum$. For each invocation where $p \neq q$, l computes the left half of $SqSum$ on Line (4), r computes the right half on Line (5), and then they are added together to get $SqSum$ on Line (6). Now, just need to prove that $SqSum(p, m)$ and $SqSum(m + 1, q)$ are correct invocations for the left and right halves, respectively. First, observe that m calculates the midpoint between p and q, namely $m = \left\lfloor \frac{p+q}{2} \right\rfloor$. Thus, the following relation holds:

$$p \leq m < m + 1 \leq q$$

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

$$SqSum(p, q) = SqSum(p, m) + SqSum(m + 1, q)$$

Lastly, notice that for the l and r invocations of $SqSum(p, q)$, the difference between p and q decreases each time, so it **does not overlap** and correctly approaches its termination criteria (base case). More specifically, l computes $p^2 + (p + 1)^2 + \cdots m^2$, and r computes $(m + 1)^2 + (m + 2)^2 + \cdots + n^2$, covering the range of values without overlap.

Therefore, since it covers the entire range between p and q and avoids overlap, then $SqSum$ is indeed correct to compute $1^2 + 2^2 + \cdots + n^2$.

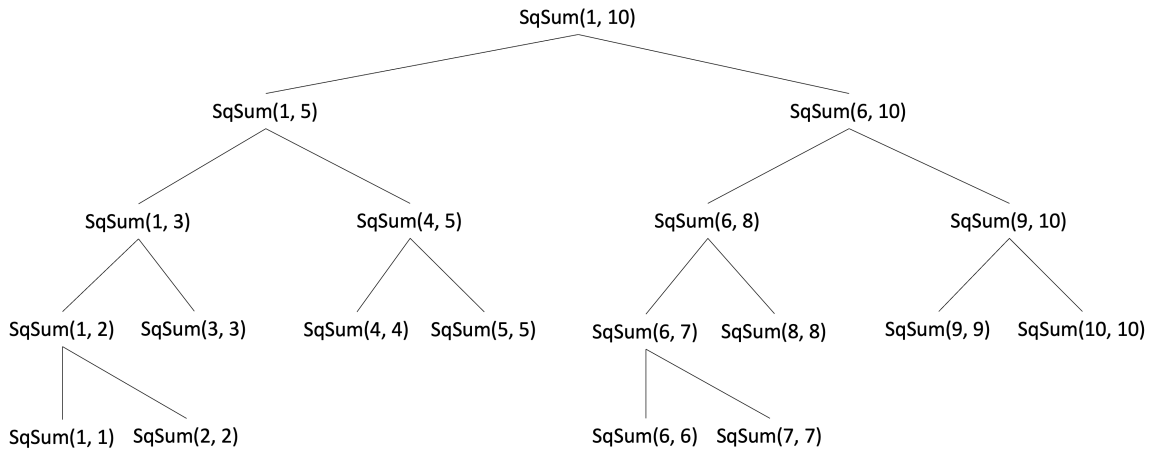
(b) $SqSum$ is a polynomial-time algorithm.

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 1 and n to calculate $1^2 + 2^2 +$

$\cdots + n^2$. To start, Line (1), Line (3), and Line (6) are constant-time operations. Line (4) performs invocations to $SqSum(p, m)$ until $m = p$, and then it will return due to the base case, thus covering the range from p to m. Similarly, Line (5) invokes $SqSum(m + 1, q)$ until $m + 1 = q$, covering the range from $m + 1$ to q. Since ranges (p, m) and $(m + 1, q)$ are covered, then together (p, q) is covered. Lastly, since it is invoked as $SqSum(1, n)$, where n is a positive integer input, and it is shown in (a) that there is no overlap, then the **time-efficiency is $\mathbf{O}(\boldsymbol{n})$**, deeming it indeed as a polynomial-time algorithm.

(c) The space-efficiency of $SqSum$ is $O(n)$, where n 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 $n = 10$, invoking $SqSum(1, 10)$:



In this example, the depth of this recursion tree is 4, which is indeed upper bounded by $O(n)$. Consequently, this also holds in the general case, where the depth of any recursion tree is bounded by $O(n)$. In addition, the input of interest for $SqSum$ is m and $m + 1$, where $m = \left\lfloor \frac{p+q}{2} \right\rfloor$. Thus, the number of splits in the tree is defined by $O\left(\frac{n+1}{2}\right)$, which can be simplified down to $O(n)$. Therefore, the space-efficiency of $SqSum$ is indeed $\boldsymbol{O(n)}$.

Exam

QUESTION 3

The following is a devised algorithm written in C++ for the given problem:

```cpp
vector<int> performKruskal(vector<vector<int>>& G) {
    vector<int> T(10, 0);
    // Finds a minimum spanning tree for graph G
    return T;
}

void augmentMST(vector<vector<int>>& H, vector<int>& U, int&
weightT) {
    for (int i = 0; i < H.size(); i++) {
        for (int j = 0; j < H[i].size(); j++) {
            for (int k = 0; k < U.size(); k++) {
                if (H[i][j] == U[k]) {
                    H[i][j]++;
                    weightT += U[k];
                    U[k] = 0;
                }
            }
        }
    }
}

bool hasUniqueMST(vector<vector<int>>& G, vector<int>& T) {
    vector<vector<int>> H = G;
    vector<int> U = T;
    int weightT = 0;

    augmentMST(H, U, weightT);

    vector<int> primeT = performKruskal(H);
    if (T == primeT) {
        return true;
    }
    int weightPrimeT = 0;
    for (int i = 0; i < primeT.size(); i++) {
        weightPrimeT += primeT[i] - 1;
    }
    if (weightT == weightPrimeT) {
        return false;
    }

    return true;
}
```

Exam

The algorithm consists of three functions: $performKruskal$, $augmentMST$, and $hasUniqueMST$. It starts off at $hasUniqueMST$, where it is given an undirected graph $G = \langle V, E, l \rangle$, as well as a minimum spanning tree of G, namely T, where $T \subseteq E$. In this method, it copies graph G into a graph H, copies T into a variable U, and calls the $augmentMST$ function. In this method, it iterates through all the vertices and edges of H, and for each edge, compares it to every edge in the minimum spanning tree U. If there is equivalence, then increment the weight of that edge by 1, count up the weight of that edge, and set that edge in the MST to visited, i.e., 0. Once that is done, it calls the $performKruskal$ function with this graph H and stores it into $primeT$. Note that the $performKruskal$ method presumably uses the Kruskal's minimum spanning tree algorithm from Lecture 6(a) to determine an MST. If the MSTs are equal, then return true. If not, then get the weight of $primeT$ and compare it to the weight of T: if they are equal, then return false, else return true.

The idea behind this algorithm is to introduce a heuristic that **prevents it from finding the same MST**. In this case, the proposed strategy is to increment the weight of all edges in the graph H that are included in the original MST, namely T, by 1. So, by the end of this procedure, graph H will be a copy of G, except that its edges included in the MST will have been incremented. Then, $performKruskal$ is invoked with this graph to determine an MST. If this function returns the same MST, then it is **guaranteed that this MST is unique**. If it returns a different MST, then the weights of these MSTs must be compared. If they are equal, then there exists an MST that is **different than the original**, and hence, T is not unique. If they are not equal, then the MST of $primeT$ is **higher weighted** than the MST of T, and hence, the graph G indeed has a unique MST, namely T. As a result, this algorithm is correct in finding MST uniqueness.

In terms of its time-efficiency, let n be the size of the input graph $G = \langle V, E, l \rangle$, and consider the $augmentMST$ function. This method iterates through all the vertices and edges of a graph of the same size as G, namely H, hence this is a $O(n)$ operation. And for each of these, it iterates through each edge in a set that is the same size as T, namely U, so this is also a $O(n)$ operation. Thus, together, this results in a total of $O(n^2)$ for $augmentMST$. For the subsequent operations, $performKruskal$ runs in time $O(n \log n)$, and then iteration through $primeT$, which stores the MST of H, would be a $O(n)$ operation. Hence, the overall time-efficiency of this algorithm is $O(n^2 + n \log n + n)$, **which simplifies to the desired runtime of $\boldsymbol{O(n^2)}$**.

Exam

QUESTION 4

(a) To prove this problem is in **NP**, utilize Definition 1 from Lecture 9(c). In particular, it will be shown that the given primality checking problem can be reduced to this problem. First, let A be the primality checking problem and let B be the stated problem in (a). Note that since the primality checking is assumed to be computable in polynomial-time, then this problem is in **NP**. The "if and only if" property: an input integer ≥ 2 is prime if and only if there is no integer, call it a, that is divisible by some $c \in \{2, 3, \dots, b\}$, where $2 < b < a$. As for the function, let a solution/witness/certificate for a **true** instance be such a $c \in \{2, 3, \dots, b\}$ that does divide a. A verification algorithm: given as input the instance and a solution for it, would check:

$$a \bmod c == 0 \,?\, immediately\ return\ true, else\ continue\ false$$

In other words, it would iterate through the elements in c, namely $\{2, 3, \dots, b\}$, and for each one, check if it divides a. This problem is different from (b) in that it **terminates once it finds the first one**. The size of the solution above is at worst n, where n is the size of the range of values in c **that it iterates up to**. This check can be carried out in time $O(n)$, hence it is polynomial-time computable. Thus, it can be said that $A \leq B$, and since A is in **NP**, then B, or the given problem in (a), is also in **NP**.

(b) This problem is similar to the one given in (a), except that it checks if no $c \in \{2, 3, \dots, b\}$ divides a. The main difference here is that it **finds the first false occurrence** in the set of values in c, namely $\{2, 3, \dots, b\}$, rather than finding the first true occurrence. This leads to a similar computationally complex decision problem. The "if and only if" property: an input integer ≥ 2 is prime if and only if there is no integer in a set, call it $c \in \{2, 3, \dots, b\}$, that can divide a. A solution/witness/certificate for a **true** instance is such a set $c \in \{2, 3, \dots, b\}$, where none of the values can divide a. A verification algorithm: given as input the instance and a solution for it, would check:

$$a \bmod c == 0 \,?\, immediately\ return\ false, else\ continue\ true$$

In other words, it would iterate through all the elements in c, and for each one, check if it divides a. Let n be the size of the entire range of values in c, then this check can be carried out in time $O(n)$, hence it is polynomial-time computable. Thus, it can be said that the primality check problem can be reduced to this problem, hence this problem is also in **NP**.

Exam

QUESTION 5

The following is a devised algorithm written in C++ for the given problem:

```
const int fairCoinToss() {
    int coinToss = rand() % 2;
    return coinToss;
}

const int getRandInt(const int& a, const int& b) {
    vector<int> v;

    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }

    auto it = v.begin();
    while (v.size() > 1) {
        if (fairCoinToss()) {
            it++;
        } else {
            v.erase(it);
        }
        if (it == v.end()) {
            it = v.begin();
        }
    }

    const int i = v[0];
    return i;
}
```

The main method for the function is $getRandInt$, and the following is an explanation of how it works. First, it initializes a vector, v, with values from the range a to b, inclusive. Next, it initializes an iterator starting at the beginning. Then, it performs the following procedure while the size of the vector is greater than 1: if $fairCoinToss$ returns *heads*, i.e., 1 in this case, simply increment the iterator, else, remove that element from the vector. At the end of this procedure, there will be one element remaining, this element is assigned to an integer i and is returned.

This method is correct because it utilizes the equal probability of $fairCoinToss$ to **either keep or erase an element** of the vector. This allows any of the values in the vector to be equally chosen, since **each element has a 50% chance** of being kept or erased every iteration. As a result, the final remaining element assigned to i is one that was equally likely to be chosen from the rest.

Exam

QUESTION 6

The goal is to prove that for every $S_{i,j} = \{s_i, s_{i+1}, \dots, s_j\}$, some member of $S_{i,j}$ is guaranteed to be chosen as pivot at some point in a run of $Quicksort(A, 1, n)$. Proof by construction:

Let the input array be $A_{i,j} = \{a_i, a_{i+1}, \dots, a_j\}$, let the low index pointer be $left = p$, let the high index pointer be $right = r$, and let the pivot be $pivot = q$. To start, omit the trivial case where the array only consists of one element, as that is already sorted. Now, initially, it is guaranteed that $left < right$, since low is initialized as 1, and n is assumed to be > 1. Therefore, the $left < right$ condition will always be true on the first run of $Quicksort(A, 1, n)$. Then, $Split(A, left, right)$ is run and the result is stored in $pivot$. The claim to prove is that this result stored in $pivot$ is guaranteed to be some member of $A_{i,j}$.

The $Split(A, left, right)$ function takes the input array A, and **picks some member of A as the pivot**, and stores the result into the $pivot$ variable. Moreover, the member it picks is **guaranteed to be between low and $high$**, inclusive, given that low and $high$ are valid indices of the array. The way in which it selects the pivot, e.g., $A[left]$, $A[right]$, $A[(left + right)/2]$, or randomly between the range, is immaterial of this proof. Therefore, what is left is to prove that low and $high$ are indeed valid indices for the recursive $Quicksort$ invocations.

The first call is $Quicksort(A, left, pivot - 1)$ to sort the left subarray of the pivot, and this is correct because all the items in the range $(left, pivot - 1)$ are $< pivot$ in value, since the $Split$ function moved every item $< pivot$ to the left of $pivot$, and since the item at $pivot$ is already in the appropriate position. This functionality is similar to sort the right subarray of the pivot, with $Quicksort(A, pivot + 1, right)$, as all items $> pivot$ have been moved to the right of $pivot$.

This algorithm continues to recurse until **there is only one item in the subarray**, then it reaches its base case, namely $left \geq right$. And at this point, **this item itself is already sorted**, so the algorithm is finished with sorting this section. Thus, the indices passed into the recursive $Quicksort$ parameters, namely $left$ and $right$, are valid and correct, **there is no overlap**.

Therefore, since $left$ and $right$ are correct, then the $Split$ function correctly selects a pivot from A and stores it into $pivot$, meaning for every $A_{i,j}$, some member of $A_{i,j}$ is guaranteed to be chosen as pivot until the algorithm finishes. Finally, since $S_{i,j}$ is simply the sorted version of $A_{i,j}$, **that contains the same items**, then by extension, some member of $S_{i,j}$ is guaranteed to be chosen as pivot at some point in a run of $Quicksort(A, 1, n)$, completing the proof.