Task 1

1. Prove that for any choice of the sequence of n operations, the amortized cost per operation is O(1).

Let $\Phi = \sum_i C[i] \; mod \; 3$ where $C[i] \; mod \; 3$ can be 0, 1, 2

1. $C[i] = 0$ for all i at the start, hence $\phi(0) = 0$ is true.
2. Since counter $C[i]$ cannot be negative (it only increases as per description), $C[i] \; mod \; 3$ is always non-negative (either 0, 1, or 2), $C[i] \; mod \; 3$ is also always non-negative, and hence $\sum_i C[i] \; mod \; 3$ is always non-negative (summation of non-negative $C[i] \; mod \; 3$). Hence, $\phi(i) \geq 0, \; \forall i$ is true.

Operation: increment(i) if:	T(i)	recursion	$\Delta\phi(i) = \phi(i) - \phi(i-1)$	Amortised cost C(i)
C[i] mod 3 = 0	1	No	+1	1 + 1 = 2
C[i] mod 3 = 1	1	No	+1	1 + 1 = 2
C[i] mod 3 = 2	1 + T(i + 1) + T(i + 2)	Yes, to increment(i + 1), increment(i + 2)	-2 + Change in potential from recursive calls	-1 + Amortised cost from recursive calls

The first 2 cases are already a constant amortised cost of +2.

Consider the third case: let the operation that triggered recursion be O_t - triggering operation.

Let the number of non-recursive calls triggered by O_t be N, recursive calls be R.

Executing non-recursive calls reduces the number of pending calls by 1 (cos it just executes itself so 1 less call to do)..
Executing recursive calls reduces the number of pending calls by 1 (itself) but spawns 2 more, so net change is +1 pending calls.
Since we start with 1 pending call (the very first one that triggers the rest of the calls) and end with 0 pending operations (finished executing everything),

$1 + N(-1) + R(1) = 0$

Hence $1 - N + R = 0$

$N = R + 1$

$actual \; cost \; of \; O_t = T(O_t)$

$= number \; of \; nonrecursive \; calls + number \; of \; recursive \; calls$

$= N + R$

$= 2R + 1 \qquad (\because N = R + 1)$

Across all those calls change in potential function caused by O_t is

$\Delta\phi(O_t) = (+1) * N + (-2) * R$ (as per table above)

$= R + 1 - 2 * R$ ($\because N = R + 1$)

$= 1 - R$

$amortised\ cost\ of\ O_t = C(O_t)$

$= T(O_t) + \Delta\phi(O_t)$ (as per definition)

$= (2R + 1) + (1 - R)$ ($\because T(O_t) = 2R + 1,\ and\ \Delta\phi(O_t) = 1 - R$)

$= R + 2$

Across n operations:

Hence, $\sum_{i=1}^{n} C(i) = (R_1 + 2) + (R_2 + 2) + \ldots + (R_n + 2)]$ ($\because C(O_t) = R + 2$)

$= 2n + \sum_{i=1}^{n} R_i$

Let actual cost across n operations be A_n

However by definition from lecture:

$\sum_{i=1}^{n} C(i) = actual\ cost\ across\ n\ operations\ A_n + \phi(n) - \Phi(0)$

To prove that amortised cost of n operations $\in O(n)$:

1. For a fixed i, recursive call only happens every third increment, hence $\sum_{i=1}^{n} R_i \leq \frac{A_n}{3}$

   Because $actual\ cost\ A_n = total\ number\ of\ increments$
2. From $\Delta\phi(O_t) = 1 - R$, since R is number of recursive calls and therefore is non-negative, $\Delta\phi(O_t) \leq 1$, ie potential never grows by more than 1 for any operation. $\phi(n) - \Phi(0) \leq n$ since there are n operations and potential for each does not grow more than by 1 at each operation

Hence, $\sum_{i=1}^{n} C(i) = 2n + \sum_{i=1}^{n} R_i \leq 2n + \frac{A_n}{3}$

Hence $A_n + \phi(n) \leq 2n + \frac{A_n}{3}$ ($\because \sum_{i=1}^{n} C(i) = A_n + \phi(n)$)

$\frac{2 * A_n}{3} + \phi(n) \leq 2n$

$\frac{2 * A_n}{3} \leq 2n$ $(\because \phi(n) \geq 0)$

$A_n \leq 3n$

Since $\phi(n) \leq n$ and $A_n \leq 3 * n$:

$A_n + \phi(n) \leq 4 * n$

$\sum_{i=1}^{n} C(i) \leq 4 * n$ $(\because \sum_{i=1}^{n} C(i) = A_n + \phi(n))$

Hence amortised cost across n operations $\sum_{i=1}^{n} C(i) \in O(n)$,

and amortised cost of 1 operation $\in O(1)$ (proven).

---

2. Prove that there exists a choice of the sequence of n operations such that some operation in the sequence incurs cost Ω(n).

Say there are n operations:
1. Increment(1)
2. Increment(2)
3. Increment(3)
…
n - 3. increment(n - 3) -> currently counter array looks like [1, 1, 1, 1, 1, 1, 1, … 0, 0, 0…] with first n - 3 counters being 1s and (n - 2)th counter onwards being 0.
n - 2. increment(1): array looks like [2, 1, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]
n - 1. increment(2) : array looks like [2, 2, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]
n. increment(1). The cost incurred by this operation will be Ω(n).

Explanation:

Initiation

After the first n - 1 operations, the array is [2, 2, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …] where the first n - 3 counters are non-zero and n - 2 counter onwards are all 0s. For all of them, the cost is Ω(1).

Table 1

Let x = 1. Then C[x] = 2, C[x + 1] = 2, C[x + 2] to C[n - 3] = 1, C[n - 2] onwards = 0. Ie the pattern is such that the counters are 2, 2, and then 1s. The next operation is increment(x) (i.e. the last step in my sequence of operations).

Propagation

Firstly, perform the last operation, ie increment(x) = increment(1):
Iincrement(x) Counter array now is[3, 2, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …].
Since C[x] % 3 = 0, it incurs 2 recursive calls:

1. increment(x + 1): [3, 2, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]
   -> [3, 3, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]. C[x + 1] % 3 = 0 now so 2 more recursive calls:
   i. increment(x + 2): [3, 3, 1, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …] -> [3, 3, 2, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]
   ii. increment(x + 3): [3, 3, 2, 1, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …] -> [3, 3, 2, 2, 1, 1, 1, 1, 1, 1, …, 0, 0, 0 …]
2. increment(x + 2)

At this point, let x' = x + 2. The pattern is the same as in *Table 1*, ie C[x'] = 2, C[x' + 1] = 2, C[x' + 2] to C[n - 3] = 1, C[n - 2] onwards = 0, and next operation is increment(x') = increment(x + 2) . Hence this pattern will repeat for as long as there is sufficient 1s to continue that pattern. To progress the pattern from x to x + 2, cost of 4 is required (as shown by pink increment operations: increment(x), increment(x + 1) increment(x + 2), increment(x + 3)) .

X in such patterns is always an odd number (1, 3, 5, …)

Termination

If n is even, there is n - 3 i.e. an odd number of non-zero counters. Once the pattern reaches 0s, ie once $x_{max} = n - 5$ and the last 1 is $n - 3$, the following will happen: [… 2, 2, 1, 0, 0, 0, 0 …] ->[… 3, 2, 1, 0, 0, 0, 0 …] ->[… 3, 3, 1, 0, 0, 0, 0 …] ->[… 3, 3, 2, 0, 0, 0, 0 …] ->[… 3, 3, 2, 1, 0, 0, 0 …] ->[… 3, 3, 3, 1, 0, 0, 0 …] ->[… 3, 3, 3, 2, 0, 0, 0 …] ->[… 3, 3, 3, 2, 1, 0, 0 …] And then recursion stops. Hence, for x from $x_1 = 1$ to $x_{max} = n - 5$, the pattern repeats, requiring a cost of 4 for each increment, and to terminate recurrence, an additional cost of 7 is needed (corresponding to 7 transitions of the counter array). Total cost for last operation starting from increment(1) = $\frac{(n-5)-1}{2} * 4 + 7 =$ $2 * (n - 6) + 7 = 2 * n - 5$ $\in \Omega(n)$	If n is odd, there is n - 3 i.e. an even number of non-zero counters. Once the pattern reaches 0s, ie once $x_{max} = n - 6$ and the last 1 is $n - 3$, the following will happen: [… 2, 2, 1, 1, 0, 0, 0 …] ->[… 3, 2, 1, 1, 0, 0, 0 …] ->[… 3, 3, 1, 1, 0, 0, 0 …] ->[… 3, 3, 2, 1, 0, 0, 0 …] ->[… 3, 3, 2, 2, 0, 0, 0 …] ->[… 3, 3, 3, 2, 0, 0, 0 …] ->[… 3, 3, 3, 3, 0, 0, 0 …] ->[… 3, 3, 3, 3, 1, 0, 0 …] ->[… 3, 3, 3, 3, 1, 1, 0 …] ->[… 3, 3, 3, 3, 2, 1, 0 …] And then recursion stops. Hence, for x from $x_1 = 1$ to $x_{max} = n - 6$, the pattern repeats, requiring a cost of 4 for each increment, and to terminate recurrence, an additional cost of 9 is needed (corresponding to 9 transitions of counter array). Total cost = $\frac{(n-6)-1}{2} * 4 + 9$ $2 * (n - 7) + 9 = 2 * n - 5$ $\in \Omega(n)$

Hence, for provided n operations, the cost for last n^{th} operation $\in \Omega(n)$ (proven).

---

3. Prove that there exists a finite sequence of operations such that the process does not terminate.

Sequence of operations:

1. Increment(1)
2. Increment(2)
3. Increment(1)

After the first 2 operations, the following transformations happened:
[0, 0, 0, 0, 0 …] -> [1, 0, 0, 0, 0 …] -> [1, 1, 0, 0, 0 …]

Table 1:

Let x = 1. The pattern is such that C[x] = 1, C[x + 1] = 1, C[x + 2] onwards are all 0s. Next action is increment(x) (which is the 3rd operation in my sequence).

Next, the following happens: when we call increment(x) = increment(1), the counter array is now [1, 1, 0, 0, 0 …] -> [2, 1, 0, 0, 0 …]. This causes recursion to happen, since C[x] mod 2 = 0, so its gonna do 2 recursive calls:

1. increment(x + 1): The following transformation proceeds: [2, 1, 0, 0, 0 …] -> [2, 2, 0, 0, 0 …]. Now C[x + 1] mod 2 = 0 so it causes 2 recursive calls
    i. increment(x + 2) and [2, 2, 0, 0, 0 …] -> [2, 2, 1, 0, 0 …]
    ii. increment(x + 3): [2, 2, 1, 0, 0 …] -> [2, 2, 1, 1, 0 …]
2. increment(x + 2)

At this point let x' = x + 2. C(x') = 1, C(x' + 1) = 1, C(x' + 2) = 0, which is the same as that at the start but with x' instead of x. The next action is now increment(x') = increment(x + 2) . Hence the pattern is the same as at the start in *Table 1*. Hence there is a loop that goes on indefinitely provided there are infinite integers which it is.

Hence, a finite number of operations (3 specifically) result in an infinite number of increments.

Task 2

1. Prove that Extend-MIS-in can be solved in polynomial time.

Uh so basically what you need to do is just check if S itself is an independent set first.
How it's done: for every single edge e = (u, v) in G, check if both vertices u and v are inside S.

1. If there is at least one e = (u, v) where both u and v are inside S -> S is not independent -> return No. This is because if S is already not independent, then any I such that S ⊆ I will not be independent, as it will contain that same edge e = (u, v), and since both u and v are already inside S they will be inside I as well, making it not independent.
2. Else return Yes. There are 2 possibilities here:
    a. If S is already maximal, then set I = S, and S ⊆ I is true such that I is maximal.
    b. If S is not maximal, it is later possible to first set I = S, and then just keep adding vertices to I such that it has no edge with any of the existing vertices in I (greedily extend). once it's no longer possible to do -> I becomes maximal (by definition).

Either way, its possible to find I such that S ⊆ I and I is maximal independent set.
Since this algorithm only does the following 2 things:
1. Marking each vertex as either inside or not inside S: O(|V|) (eg create a boolean array)
2. Checking for each edge $e \in E$ in G, whether both ends are inside S using boolean array from step 1: complexity O(|E|)
Total complexity T(G = (V, E)) = O(|V| + |E|)
and there is a polynomial number of edge $e \in E$ and vertices $v \in V$, this algorithm runs in polynomial time. Hence, Extend-MIS-in can be solved in polynomial time.

---

2. Prove that Extend-MIS ≤P Extend-MIS-out.

Extend-MIS(G, Si, So) -> I such that I is maximal independent set of G, and $S_i \subseteq I$ and $I \cap S_0 = \varnothing$

What to do: form a set S' where each vertex (say a) inside is adjacent to one of the vertices from S_i (say b) and only that one vertex: $e = (a, b)$. Then assign $S'' = S' \cup S_0$. Then run Extend-MIS-out(G'', S'').

Proof:

1. If there is I such that $S_i \subseteq I$ and $I \cap S_0 = \varnothing$. In G'', I is still independent and avoids all points in S'' because they are either adjacent to I's points or are in S_0 such that $I \cap S_0 = \varnothing$. Hence $I \cap S'' = \varnothing$, and Extend-MIS-out(G'', S'') would return a Yes.
2. If there is a maximal independent set J in G'' such that $J \cap S'' = \varnothing$. For each vertex v in S_i there is an adjacent point in S'', and since that adjacent's point only neighbour is in J, v would be in J (otherwise the adjacent point would have to be n J which is not possible since its been excluded). Thus $S_i \subseteq J$. Also since $S_o \subseteq S''$ by its definition,

$J \cap S_0 = \varnothing$. Hence both conditions are satisfied, and Extend-MIS(G, Si, So) would return a Yes.

Complexity: to reduce from Extend-MIS to Extend-MIS-out, you need to copy all vertices (O|V|), edges (O(|E|)), create copy of each vertex in Si (O(|Si|) upperbounded by O(|V|) as there may not be more vertices in Si than in V since V are all vertices in G already), and create edge between new vertices and those in Si (exactly (O(|Si|) since there is exactly 1 edge per vertex in Si).
Hence total time complexity is O(|V| + |E| + 2O(|Si|)) = O(3|V| + |E|) = O(|V| + |E|).
Since number of edges and vertices are both polynomial, reduction runs in polynomial time, and Extend-MIS-out is also polynomial, entire algo is polynomial.

---

3. Extend-MIS-out ≤P Extend-MIS-in ?
No. Counterexample:
G = (1), S = (1). Hence, G' = (1, v*), S' = (v*).
Then Extend-MIS-out(G, S) will output is no, since there is only 1 vertex in G which is also in S, hence there is no vertices in G / S, hence no vertices can even make a maximal independent set such that S is not included (the only way to achieve maximal independent set is by including vertex 1 which is in S hence answer is NO).

However, Extend-MIS-in(G', S') will output yes, because G has 2 vertices, 1 and v*, while S' has one vertex v*. Hence maximal independent set I = {v*} and $S \subseteq I$ since $S = I$.

A No instance of Extend-MIS-out is mapped to a Yes instance of Extend-MIS-in, hence it is not a correct reduction.

---

4. Using the oracle, design an algorithm that solves Extend-MIS-unique in polynomial time.

1. Run Extend-MIS(G, Si, So). If output is No, then return No straightaway (because since there isn't any maximal independent set I to begin with, then there won't be a **unique** maximal independent set I either).
2. For every single vertex v in V / $S_0 \cup S_i$, check if there is a maximal set with that vertex and without that vertex. It is done by doing the following:
    a. Run Extend-MIS(G, Si $\cup \{v\}$, So) to forcibly include v
    b. Run Extend-MIS(G, Si, So $\cup \{v\}$) to forcibly exclude v

    If the answer for both is Yes (meaning there is maximal set both with and without v), return a No, else continue checking other vertices in V / $S_0 \cup S_i$ until such a vertex is found.
3. If all vertices in step 2 did not output 2 Yes for 2 checks, return a Yes (since there is no such vertex which with or without it both have a maximal independent set I.

To proof correctness, got 3 cases:

1. No maximal set to begin with

Such case algo will output No as per step 1.

2. If the maximal set is not unique, algo correctly outputs No:

This is because, if maximal set I is not unique, (say there are 2: I_1 and I_2), then there is a possibility that at least one of the vertices is in one I but not the other, such that either $v \in I_1 / I_2$ or $v \in I_2 / I_1$, otherwise sets would be completely identical.

1. $v \in I_1 / I_2$, then in which case excluding v would make Extend-MIS still detect I_2 and output Yes, including v would make Extend-MIS detect I_1 or I_2 and still output Yes. Hence our algo successfully determine that the maximal set is not unique.
2. $v \in I_2 / I_1$ argument is the same as 1.

Hence if u check every single vertex in V / $S_0 \cup S_i$, you will definitely check the described v as well, and that specific check will output 2 Yes for 2 checks, and our algo will correctly return a No.

2. If maximal set is unique, algo correctly outputs Yes:

If the maximal set I is unique, call it I. Then for every single vertex v in V / $S_0 \cup S_i$, its either in I or not in I.

1. If v in I:
    a. Including it will output a Yes since all vertices inside the maximal set are there so all good.
    b. Excluding it will output a No because a vertex is missing from maximal set, so maximal set is no longer maximal, and since there is no alternative maximal set in place, output a No.
2. If v is not in I:
    a. Excluding it won't affect maximal set I so its a Yes
    b. Including it forcibly will output a No because since there is unique maximal set, there wont be an alternative set which would have v as well, so output a No.

Hence not a single check would return 2 Yes for both checks, and our algo correctly outputs a Yes.

If G has a total of V vertices, there is less than O(|V|) checks that have to be made. If runtime of oracle Extend-MIS is T(n) (polynomial), then total runtime for my algo is O(|V|) * T(n), which is also polynomial.

Ie since polynomial functions are closed under compositions, and oracle is polynomial, my algo is also polynomial.