1. (a) Suppose $D \leqslant_p E$ and $E \in NP$. Then, there is a polynomial-time computable function f such that for all inputs x (for D), $f(x)$ is an input for E and x is a yes-instance for D iff $f(x)$ is a yes-instance for E. Also, there is a verifier $V_E(x,c)$ for E that runs in polynomial time and such that for all yes-instances x, $V_E(x,c) = \mathsf{True}$ for *some* c; for all no-instances x, $V_E(x,c) = \mathsf{False}$ for *all* c.

   We construct a verifier V_D for D as follows.

   $V_D(x,c)$: **return** $V_E(f(x),c)$

   Note that V_D runs in polynomial time (as a function of size(x)) because f is computable in polytime and V_E runs in polytime.

   Also, $V_D(x,c)$ outputs True for some value of c iff $V_E(f(x),c)$ outputs True for some value of c (by the construction of V_D) iff $f(x)$ is a yes-instance for E (by definition, since V_E is a verifier for E) iff x is a yes-instance for D (by definition, since $D \leqslant_p E$).

   Hence, V_D is a polytime verifier for D so $D \in NP$, by definition.

   (b) Decision problem D is *coNP*-hard iff

$$\forall D' \in coNP, D' \leqslant_p D.$$

   (c) Suppose D is *coNP*-hard and $D \in NP$.

   ***coNP* $\subseteq$ *NP*:** For all $D' \in coNP$, $D' \leqslant_p D$ (because D is *coNP*-hard) so $D' \in NP$ (because $D \in NP$).

   ***NP* $\subseteq$ *coNP*:** For all $D' \in NP$, $\overline{D'} \in coNP$, by definition of *coNP* (where $\overline{D'}$ is the complement of D').
   Then $\overline{D'} \in NP$ (since $coNP \subseteq NP$ was shown above) so $\overline{\overline{D'}} = D' \in coNP$.

   Hence, $NP = coNP$.

2. (a) The PARTITION *decision* problem is defined as follows.

   Input: A set of integers $S = \{x_1, x_2, \ldots, x_n\}$.

   Question: Is there some partition of S into subsets S_1, S_2 with equal sum?

   (b) To show that PARTITION is polytime self-reducible, first suppose that PART is an algorithm that solves the PARTITION decision problem, *i.e.*, for all input sets S, PART(S) = True if S can be partitioned, PART(S) = False otherwise.

   We write an algorithm to solve the PARTITION search problem, as follows.

```
PartSearch(S):
    if not Part(S): return NIL
    T ← 2 Σ_{x∈S} |x|
    S1 ← ∅        # accumulator for the first subset
    t1 ← 0        # sum of elements of S1
    S2 ← ∅        # accumulator for the second subset
    t2 ← 0        # sum of elements of S2
    for each x ∈ S:
        S ← S − {x}
        # see if it works to put x in S1
        if (x + t1 = t2 and Part(S)) or (x + t1 ≠ t2 and Part(S ∪ {x + t1 + T, t2 + T})):
            S1 ← S1 ∪ {x}
            t1 ← t1 + x
        else:
            S2 ← S2 ∪ {x}
            t2 ← t2 + x
    return (S1, S2)
```

   Correctness: Clearly, PARTSEARCH returns the correct value if S cannot be partitioned.

If S can be partitioned, then the following fact is a loop invariant:

Either ($t_1 = t_2$ and S can be partitioned) or ($t_1 \neq t_2$ and $S \cup \{t_1 + T, t_2 + T\}$ can be partitioned).

When the loop terminates, $S = \varnothing$. Then it is not possible to have $t_1 \neq t_2$ because this would imply that $t_1 + T \neq t_2 + T$ and since both $t_1 + T$ and $t_2 + T$ are positive, the set $\{t_1 + T, t_2 + T\} = S \cup \{t_1 + T, t_2 + T\}$ cannot be partitioned. So $t_1 = t_2$, and this means (S_1, S_2) is a correct partition of the original S (since t_1 is the sum of elements of S_1 and t_2 is the sum of elements of S_2).

Correctness of the loop invariant:

- This is clearly true at the start of the loop, because S can be partitioned into subsets S'_1, S'_2 with equal sum, and $t_1 = t_2 = 0$.
- Suppose that the loop invariant is true at the start of one iteration of the loop

  Case 1: Suppose that $t_1 = t_2$ and that S can be partitioned into (S'_1, S'_2), at the start of the iteration.

  Consider the element x removed from S and suppose, without loss of generality, that $x \in S'_1$.

  Sub-case A: If $x = 0$, then $x + t_1 = t_2$. Also, $S - \{x\}$ can be partitioned by placing x in either subset S_1 or S_2 (this will not change any of the sums). So the algorithm places x in S_1.

  Sub-case B: If $x \neq 0$, then $x + t_1 \neq t_2$. In this case, $S - \{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned into $\big(S'_1 - \{x\} \cup \{x + t_1 + T\}, S'_2 \cup \{t_2 + T\}\big)$, because (S'_1, S'_2) partitions S and $t_1 + T = t_2 + T$. So the algorithm places x in S_1.

  In both sub-cases, the algorithm places x in S_1.

  Case 2: Suppose $t_1 \neq t_2$ and $S \cup \{t_1 + T, t_2 + T\}$ can be partitioned into (S'_1, S'_2), at the start of the iteration. Note that $t_1 + T$ and $t_2 + T$ cannot both belong to the same subset of the partition, as there would be no way for other numbers from S to add up to $t_1 + t_2 + 2T \geqslant T$. Without loss of generality, assume $t_1 + T \in S'_1$ and $t_2 + T \in S'_2$.

  Consider the element x removed from S.

  Sub-case A: If $x \in S'_1$, then either $x + t_1 = t_2$, in which case or $x + t_1 \neq t_2$, in which case

  Sub-case A: If $x + t_1 = t_2$, then x can be placed in S_1 iff $S - \{x\}$ can still be partitioned; otherwise, x must be placed in S_2. We know this because

  Sub-case B: If $x + t_1 \neq t_2$, then x can be placed in S_1 iff $S - \{x\} \cup \{x + t_1 + T, t_2 + T\}$ can still be partitioned; otherwise, x must be placed in S_2. We know this because

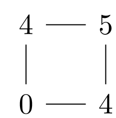
  If $x \in S'_1$ in the partition, then $(S - \{x\}) \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned (because x is added to $t_1 + T$ as part of S'_1 anyway). If $(S - \{x\}) \cup \{x + t_1 + T, t_2 + T\}$ cannot be partitioned, then it must be because $x \in S'_2$ in the partition—since x must belong to either S'_1 or S'_2.

  Either way, the algorithm places x in a subset that works.

Runtime: Let $t(n)$ be the runtime of PART(S), where $n = |S|$. Then, PARTSEARCH(S) runs in worst-case time $\mathcal{O}(n^2 + n\,t(n))$, because the main loop iterates n times, calling PART at most twice each time and taking no more than time $\mathcal{O}(n)$ to perform basic set operations.

Hence, by definition, PARTITION is polytime self-reducible.

3. (a) For the input on the right (where we've indicated the profit of each corner), the algorithm returns the corners with profits 5 and 0, for a total of 5. However, picking both corners with profits 4 would give a total profit of 8 instead.



(b) Note that the input to the problem can be represented as an undirected graph G, and valid selections of corners are the same as independent sets in G.

Let $A(G)$ be the smallest total profit of any independent set returned by the greedy algorithm, and $M(G)$ be the *maximum* profit of *all* independent sets of G. We prove that for all inputs G, $A(G) \geqslant M(G)/4$—and that for at least one input G_0, $A(G_0) = M(G_0)/4$.

Let S be any independent set returned by the algorithm, and T be any independent set with maximum profit in G. For all $v \in T$, if $v \notin S$, then there is some corner $v' \in S$ such that $(v', v) \in E$ and $p(v') \geqslant p(v)$ — otherwise, v could be added to S. When v was removed from G by the algorithm, it was because of some adjacent corner v' being added to S, which means that at that point, v' had the largest profit among all remaining corners, including v.

Since no corner in S has more than 4 neighbours, for all $v \in S$, there are at most 4 corners $v_1, v_2, v_3, v_4 \in T$ such that $p(v) \geqslant p(v_1)$, $p(v) \geqslant p(v_2)$, $p(v) \geqslant p(v_3)$, $p(v) \geqslant p(v_4)$. In other words, for all $v \in S$, $4p(v) \geqslant p(v_1) + p(v_2) + p(v_3) + p(v_4)$, in the worst case, to "cover" all corners in T. Hence, $4p(S) \geqslant p(T)$, *i.e.*, $A(G) \geqslant M(G)/4$, as desired.

To show that $A(G)$ can be equal to $M(G)/4$, consider the input on the right. The algorithm will select the corners with profits $(p + 1) + 0 + 0 + 0 + 0 = p + 1$ while the maximum profit is obtained by picking the corners $p + p + p + p = 4p$. This is not quite the desired factor of 4, though it can be made arbitrarily close to it by picking p large enough. To get a factor of 4 exactly, simply set the profit of the middle "corner" to p. Then, the selection returned by the algorithm is no longer *guaranteed* to be sub-optimal, but it is *possible* that the algorithm will select the middle "corner" first, and end up with a solution whose value is exactly $1/4$ of the optimum.

$$\begin{tikzcd}
0 \arrow[r, dash] \arrow[d, dash] & p \arrow[r, dash] \arrow[d, dash] & 0 \arrow[d, dash] \\
p \arrow[r, dash] \arrow[d, dash] & p+1 \arrow[r, dash] \arrow[d, dash] & p \arrow[d, dash] \\
0 \arrow[r, dash] & p \arrow[r, dash] & 0
\end{tikzcd}$$