Problem 1

1. Give a detailed argument that for all decision problems D and E, if $D \leq_p E$ and $E \in NP$, then $D \in NP$.

   Proof. Assume D, E represented as formal languages. We wish to construct a polynomial-time verification algorithm for D. Since $D \leq_p E$, there exists a polynomial-time reduction function f such that for any $x \in \{0,1\}^*$, an instance $x \in D$ if and only if the transformed instance $f(x) \in E$. Also given $E \in NP$, there exists a polynomial time verification algorithm A such that

   $$E = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ where } y \in O(|x|^c) \text{ such that } A(x,y) = 1\}$$

   So we have

   $$D = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ where } y \in O(|f^{-1}(x)|^c) \text{ such that } A(f^{-1}(x), y) = 1\}$$

   Let $B(x,y) = A(f^{-1}(x), y)$, which is a composite of polynomial reduction function f and polynomial-time verification algorithm A, hence B is a polynomial-verification algorithm for D, so $D \in NP$ □

2. By analogy with the definition of NP-hardness, give a precise definition of what it means for a decision problem D to be coNP-hard.

   Solution. □

   D is coNP-hard if for all $L \in coNP$, $L \leq_p D$. In other words, D is coNP-hard if every problem in $coNP$ is polynomial-time reducible to D.

3. Show that if decision problem D is coNP-hard, then $D \in NP$ implies $NP = coNP$.

   Proof. 2 directions

   (a) Prove $coNP \subseteq NP$. Let arbitrary $L \in coNP$. Since D is coNP-hard, then $L \leq_p D$. Given D is NP, by result from part a of this question, we have $L \in NP$. So $L \in coNP \to L \in NP$ implies $coNP \subseteq NP$

   (b) Prove $NP \subseteq coNP$. Similarly, let arbitrary $L \in NP$, then $\overline{L} \in coNP$. Since D is coNP-hard, $\overline{L} \leq_p D$. Since D is NP, then by result from part a of this question, we have $\overline{L} \in NP$, this implies, $L \in coNP$. So $L \in NP \to L \in coNP$ implies $NP \subseteq coNP$

   Since $coNP \subseteq NP$ and $NP \subseteq coNP$, so $NP = coNP$ □

Problem 2

For each decision problem D below, state whether $D \in P$ or $D \in NP$, then justify your claim.

- For decision problems in P, describe an algorithm that **decides** the problem in polytime (including a brief argument that your decider is correct and runs in polytime).
- For decision problems in NP, describe an algorithm that **verifies** the problem in polytime (including a brief argument that your verifier is correct and runs in polytime), and give a detailed reduction to show that the decision problem is NP-hard for your reduction(s), you must use one of the problems shown to be NP-hard during lectures or tutorials.

1. ExactCycle
   - **Input** Undirected graph G and $k \in \mathbb{Z}$
   - **Question** Does G contain some simple cycle on **exactly** k vertices?

   Solution. □

   ExactCycle can be represented as

   $$\text{ExactCycle} = \{\langle G, k\rangle : G \text{ contains a simple cycle of size } k\}$$

   (a) Prove ExactCycle $\in NP$

   Proof. We prove this by finding an polynomial-time verification algorithm. Given an instance of ExactCycle $\langle G, k\rangle$. The certificate is a cycle of vertices $\langle v_0, \cdots, v_k\rangle$. The algorithm checks

   i. There are k unique vertex in the cycle, with $v_0 = v_k$
   ii. Each of (v_i, v_{i+1}) is a valid edge in E

   and outputs 1 (yes) if both checks are true and 0 (no) otherwise. Both steps can be done in polynomial time easily.

   i. We can check for uniqueness of vertices in the sequence by making $\binom{k}{2}$ pairwise comparison, which takes $O(k^2)$ time.
   ii. The second step is simply a look up in the graph and there are a total of k edges to verify, which takes a total of $O(k)$, assuming a constant time lookup in adjacency matrix representation of the graph.

   So the verification algorithm is a polynomial time algorithm. If the certificate is a simple cycle of size k, then the verification algorithm will output 1 accordingly as the checks the algorithm performs is equivalent in definition to a simple cycle of size k. If the certificate, either is not simple, does not contain a cycle, or contain invalid edges, the algorithm will output 0 accordingly. □

(b) Prove for all $L \in NP$, $L \leq_p$ ExactCycle (i.e. NP-hard)

Proof. By lemma in clrs, we can find a NP-complete problem Ham-Cycle and a polynoial time reduction algorithm mapping $x \in$ Ham-Cycle to $f(x) \in$ ExactCycle to prove that ExactCycle is NP-complete. Given an instance of Ham-Cycle $\langle G \rangle$, the reduction algorithm computes $k = |G.V|$ and outputs an instance of $\langle G' = G, k \rangle$ to ExactCycle. The transformation function f is polynomial, in fact constant as we are only computing the size of vertices in G. Now we prove that the transformation is a valid reduction

i. Suppose C is a hamiltonian cycle in G. Then we have $k = |G.V| = |C|$. We claim that C is a simple cycle of length k in G'. Indeed, we have $|C| = k$ by construction. Therefore there is a simple cycle of size k in G'

ii. Suppose there is a simple cycle of size k in G'. Let C be such simple cycle. We claim that C is hamiltonian cycle in G. There are k vertices in G by construction, the fact that C is a simple cycle of size k implies that C is a hamiltonian cycle, which is simply a simple cycle over every vertex (k of them).

□

2. LargeCycle

- **Input** Undirected graph G and $k \in \mathbb{Z}$
- **Question** Does G contain some simple cycle on **at least** k vertices?

Solution. □

LargeCycle can be represented as

$$\text{LargeCycle} = \{\langle G, k \rangle : G \text{ contains a simple cycle of size } \geq k\}$$

(a) Prove LargeCycle $\in NP$

Proof. We prove this by finding an polynomial-time verification algorithm. The algorithm is exactly that of the ExactCycle verification algorithm with one difference, we are checking if the certificate, a sequence of vertices, have length greater than or equal to k instead of testing if the length is equal to k. The complexity and correctness analysis follows similarly. □

(b) Prove for all $L \in NP$, $L \leq_p$ LargeCycle (i.e. NP-hard)

Proof. By lemma in clrs, we can find a NP-complete problem Ham-Cycle and a polynoial time reduction algorithm mapping $x \in$ Ham-Cycle to $f(x) \in$ LargeCycle to prove that LargeCycle is NP-complete. Given an instance of Ham-Cycle $\langle G \rangle$, the reduction algorithm computes $k = |G.V|$ and outputs an instance of $\langle G' = G, k\rangle$ to LargeCycle. The transformation function f is polynomial, in fact constant as we are only computing the size of vertices in G. Now we prove that the transformation is a valid reduction

i. Suppose C is a hamiltonian cycle in G. Then we have $k = |G.V| = |C|$. We claim that C is a simple cycle of length k in G'. Indeed, we have $|C| = k$ by construction. Therefore there is a simple cycle of size k in G', implying there is a simple cycle of size at least k in G'

ii. Suppose there is a simple cycle of size at least k in G'. Let C be such simple cycle. Since $|G'.V| = k$, the simple cycle has exactly size k. We claim that C is hamiltonian cycle in G. There are k vertices in G by construction, the fact that C is a simple cycle of size k implies that C is a hamiltonian cycle, which is simply a simple cycle over every vertex (k of them).

□

3. SmallCycle
   - **Input** Undirected graph G and $k \in \mathbb{Z}$
   - **Question** Does G contain some simple cycle on **at most** k vertices?

Solution. □

(a) Prove SmallCycle $\in NP$

Proof. We prove this by finding an polynomial-time verification algorithm. The algorithm is exactly that of the ExactCycle verification algorithm with one difference, we are checking if the certificate, a sequence of vertices, have length less than or equal to k instead of testing if the length is equal to k. The complexity and correctness analysis follows similarly. □

(b) Prove for all $L \in NP$, $L \leq_p$ SmallCycle (i.e. NP-hard)

Proof. .... □

Problem 3

Consider the following Partition search problem.

1. **Input** A set of integers $S = \{x_1, \cdots, x_n\}$ each integer can be positive, negative, or zero.
2. **Output** A partition of S into subsets S_1, S_2 with equal sum, if such a partition is possible; otherwise, return the special value *nil*. (S_1, S_2 is a partition of S if every element of S belongs to one of S_1 or S_2, but not to both.)

1. Give a precise definition for a decision problem PART related to the Partition search problem.

   Solution. □

   Given a set of integers $S = \{x_1, \cdots, x_n\}$ where each integer x_i can be positive, negative, or zero. We want to find if there exists a partition S_1, S_2 with equal sums.

$$\text{PART} = \left\{ \langle S \rangle : \exists S_1, S_2 \subseteq S \text{ s.t. } S_1 \cup S_2 = S \text{ and } S_1 \cap C_2 = \emptyset \text{ where } \sum_{x \in S_1} x = \sum_{x \in S_2} x \right\}$$

   Let PARTDECIDE be the algorithm that decides the decision problem PART, specifically

$$\text{PARTDECIDE}(S) = \begin{cases} 1 & \text{if } \langle S \rangle \in \text{PART} \\ 0 & \text{otherwise} \end{cases}$$

2. Give a detailed argument to show that the PARTITION search problem is polynomial-time self-reducible. (Warning: Remember that the input to the decision problem does not contain any information about the partition if it even exists.)

   Solution. □

   Now we provide an efficient algorithm utilizing PARTDECIDE, assumed to be efficient, to solve for PARTITION

```
Function Partition($S$)
    if Not PartDecide($S$) then
        return nil
    $S_1 \leftarrow \emptyset$
    while $|S| > 2$ do
        $x_i, x_j \in S$ be arbitrary
        $S' \leftarrow S \setminus \{x_i, x_j\}$
        if PartDecide($S' \cup \{x_i + x_j\}$) then
            $S \leftarrow S' \cup \{x_i + x_j\}$
            $S_1 \leftarrow S_1 \cup \{x_i + x_j \}$
        else
            $S \leftarrow S' \cup \{|x_i - x_j|\}$
    return $(S_1, S \setminus S_1)$
```

   (a) **Proof of correctness** Note that the for loop will terminate, as in each iteration the size of S decrements by one, specifically, x_i, x_j is removed from S and we selectively adds either $x_i + x_j$ or $|x_i - x_j|$ in S. Since S is sized, the for loop will terminate in $O(n)$ iterations. Now we claim that after the for loop has a loop variant

During i-th iteration, if $\textsc{PartDecide}(S_i) = 1$, then $\textsc{PartDecide}(S_{i+1}) = 1$

Proof by induction, the execution reaches line 4 if and only if $\textsc{PartDecide}(S)$ evalutes to 1. So when $i = 0$, the claim holds. For induction step, assume $\textsc{PartDecide}(S_i) = 1$. 2 cases for S_{i+1}

i. If $\textsc{PartDecide}(S \setminus \{x_i, x_j\} \cup \{x_i + x_j\}) = 1$ then assigning $S_{i+1} \leftarrow S \setminus \{x_i, x_j\} \cup \{x_i + x_j\}$ would imply $\textsc{PartDecide}(S_{i+1})$ evaluates to 1. In other words, we can decide to put x_i and x_j into the same partition and the resulting set with this restriction (i.e. x_i, x_j have to be in the same set) would still be able to yield an equal partition

ii. Otherwise, we cannot put x_i, x_j into the same partition. So we put x_i and x_j into different partitions. By induction hypothesis $\sum\limits_{x \in S_1} x = \sum\limits_{x \in S_2} x$, assume without loss of generality that $x_i \in S_{i1}$ and $x_j \in S_{i2}$ and $x_i > x_j$, then

$$\sum_{x \in S_1 \setminus \{x_i\}} x + x_i = \sum_{x \in S_2 \setminus \{x_j\}} x + x_j$$

$$\sum_{x \in S_1 \setminus \{x_i\}} x = \sum_{x \in S_2 \setminus \{x_j\}} x + |x_j - x_i|$$

So we have found an equal partition for $S_{i+1} = S \setminus \{x_i, x_j\} \cup \{|x_i - x_j|\}$. Therefore the claim holds in this case as well.

So when the for loop terminates, which it will as proved earlier, we have $|S_{two}| = 2$ such that $\textsc{PartDecide}(S_{two}) = 1$. Since there are only 2 elements in S_{two}, they must be equal. So algorithm's return value is correct.

(b) **Runtime** The for loop iterates $O(n)$ times, each performing some constant set operations together with $\textsc{PartDecide}$. Assume $\textsc{PartDecide}$ has a worst case runtime of $O(T)$, The worst case complexity of the algorithm is therefore $O(nT)$

Given that the algorithm is correct and the runtime, $O(nT)$, is polynomial to the worst case runtime of $\textsc{PartDecide}$. If T is polynomial, in other words, if there exists an efficient algorithm for solving $\textsc{PartDecide}$, then we can solve $\textsc{Partition}$ in (nT), which is still in polynomial time. Therefore the search problem $\textsc{Partition}$ is **self-reducible**

Problem 4

Your friends want to break into the lucrative coffee shop market by opening a new chain called The Coffee Pot. They have a map of the street corners in a neighbourhood of Toronto (shown on the right), and estimates $p_{i,j}$ of the profits they can make if they open a shop on corner (i, j), for each corner. However, municipal regulations forbid them from opening shops on corners $(i-1, j), (i+1, j), (i, j-1), and(i, j+1)$ (for those corners that exist) if they open a shop on corner (i, j). As you can guess, they would like to select street corners where to open shops in order to maximize their profits!

1. Consider the greedy algorithm to try and select street corners

```
$C \leftarrow \{(i,j) : 1 \leq i \leq m$ and $1 \leq j \leq n\}$
$S \leftarrow \emptyset$
while $C \neq \emptyset$ do
    Pick $(i,j) \in C$ with maximum value of $p_{i,j}$
    $S \leftarrow S \cup \{(i,j)\}$
    $C \leftarrow C \setminus \{(i,j),(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$
return $S$
```

   Give a precise counter-example to show that this greedy algorithm does not always find an optimal solution. State clearly the solution found by the greedy algorithm, and show that it is not optimal by giving another selection with a larger profit.

   Solution. $\square$

   Define the profit of a set of corners as follows

$$P(C) = \sum_{(i,j)\in C} p_{i,j}$$

   Let $m = n = 2$, so we have 4 corners to choose from

$$\begin{array}{ccc} (2,1) & \cdots & (2,2) \\ \vdots & & \vdots \\ (1,1) & \cdots & (1,2) \end{array}$$

   Now assign $p_{i,j}$ as follows

$$\begin{array}{ccc} 4 & \cdots & 3 \\ \vdots & & \vdots \\ 3 & \cdots & 1 \end{array}$$

   The greedy algorithm picks $(2,1)$ which has the largest initial profit 4, after removing corners $(1,1)$ and $(2,2)$, the greedy algorithm picks the remaining $(1,2)$ and adds it to corners C. Therefore

$$P(C) = 4 + 1 = 5$$

   Apparently the selection $\{(1,1),(2,2)\}$ is optimal and yields a total profit of $3+3=6$

2. Prove that the greedy algorithm from part (a) has an approximation ratio of 4. (Hint: Let S be the selection returned by the greedy algorithm and let T be any other valid selection of street corners. Show that for all $(i,j) \in T$, either $(i,j) \in S$ or there is an adjacent $(i',j') \in S$ with $p_{i',j'} \geq p_{i,j}$. What does this means for all $(i,j) \in S$ and their adjacent corners?)

Proof. Let S be selection returned by the greedy algorithm and let T be any other valid selection of street corners (including the optimal selection O). Now we claim that for all $(i,j) \in T$, either $(i,j) \in S$ or there exists an adjacent corner $(i',j') \in S$ such that $p_{i',j'} \geq p_{i,j}$. Let $(i,j) \in T$ be arbitrary. If $(i,j) \in S$, then done, otherwise $(i,j) \notin S$, this implies at some point the greedy algorithm removed (i,j) from the set of available corners C. The algorithm only removes adjacent corners to the one picked (i.e. one with largest $p_{i,j} \in C$), which we denote as (i_{picked}, j_{picked}). Note $p_{i_{picked},j_{picked}} \geq p_{i,j}$ since otherwise the algorithm would have picked $p_{i,j}$ instead. Since (i_{picked}, j_{picked}) and (i,j) are adjacent corners, the claim thus holds. Now we claim that for each $(i,j) \in S$, $4p_{i,j}$ is an upper bound on the total profit of any possible valid selection R of corners T restricted to $\{(i,j),(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$, we have either

(a) $(i,j) \in T$, and since T is a valid selection of corners, no adjacent corners to (i,j) is in T, and so $R = \{(i,j)\}$ with

$$4p_{i,j} \geq P(\{(i,j)\}) = P(R)$$

(b) Otherwise, $(i,j) \notin T$, then any adjacent corners may be selected by T, therefore $R = \{(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$. Since by previous conclusion proved, we have $p_{i,j}$ be greater than any of its adjacent vertices and so

$$4p_{i,j} \geq P(\{(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}) = P(R)$$

Therefore the claim holds. Note the set $\{(i,j),(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$ is exactly the set of corners being removed from C in each iteration of the loop. By loop exit condition, $C = \emptyset$. Therefore the cumulation of any valid selection R_i in each iteration equates to the T, i.e. $\bigcup_i R_i = T$. We claim that once the loop terminates the total profit $4P(S)$ is an upper bound on profit of any other possible selection $P(T)$

$$4P(S) = 4 \sum_{(i,j)\in S} p_{i,j} \geq \sum_i R_i = P(T)$$

Since T is arbitrary, let O be the optimal selection and we have

$$4P(S) \geq P(O) \rightarrow \frac{P(O)}{P(S)} \leq 4$$

Hence the greedy algorithm has an approximation ratio of 4. □

Problem 5

You are using a multi-processor system to process a set of jobs coming in to the system each day. For each job $i = 1, 2, \cdots, n$, you know v_i, the time it takes one processor to complete the job.

Each job can be assigned to one and only one of m processors no parallel processing here! The processors are labelled $\{A_1, A_2, ..., A_m\}$. Your job as a computer scientist is to assign jobs to processors so that each processor processes a set of jobs with a reasonably large total running time. Thus given an assignment of each job to one processor, we can define the **spread** of this assignment to be the minimum over $j = 1, 2, \cdots, m$ of the total running time of the jobs on processor A_j.
Example: Suppose there are 6 jobs with running times 3, 4, 12, 2, 4, 6, and there are m = 3 processors. Then, in this instance, one could achieve a spread of 9 by assigning jobs $\{1, 2, 4\}$ to the first processor (A_1), job 3 to the second processor (A_2), and jobs 5 and 6 to the third processor (A_3). The ultimate goal is find an assignment of jobs to processors that **maximizes** the spread. Unfortunately, this optimization problem is NP-Hard (you do not need to prove this). So instead, we will try to approximate it.

1. Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s, then your algorithm should produce a selection of processors for each job that has spread at least $\frac{s}{2}$. In your algorithm you may assume that no single job has a running time that is significantly above the average; specifically, if $V = \sum_{i=1}^{n} v_i$ denotes the total running time of all jobs, then you may assume that no single job has a running time exceeding $\frac{V}{2m}$

   Solution. □

   Briefly, the algorithm works by greedily assign the next longest job to the processor with the least current spread. Let A be a processor representing arbitrary set of jobs $i = 1, \cdots, n$. Define the total time T of a set of jobs as

$$T(A) = \sum_{i \in A} v_i$$

   We define spread s as follows

$$s = \underset{j \in 1,2\cdots,m}{\textsc{Min}} \sum_{i \in A_j} v_i = \underset{j \in 1,2\cdots,m}{\textsc{Min}} T(A_j)$$

   et S be the set of jobs $\{1, \cdots, n\}$. Let v be the corresponding run time. Let m be the number of processors. Let s_j be the spread of j-th processor A_j. Let Q be a minimum priority queue of processors A_j with keys corresponding to summation of runtime of all jobs in A_j

```
Function Approx-Max-Spread($S, v, m$)
    $Q \leftarrow$ MinPriorityQueue
    for $j = 1$ to $m$ do
        $A_j \leftarrow \emptyset$
        $A_j.key \leftarrow 0$
        Insert($Q, A_j$)
    while $S \neq \emptyset$ do
        $i \leftarrow$ arbitrary job in $S$
        $S \leftarrow S \setminus \{i\}$
        $A \leftarrow$ ExtractMin($Q$)
        $A \leftarrow A \cup \{i\}$
        $A.key \leftarrow \sum_{i \in A} v_i$
        Insert($Q, A$)
    return $A_j$ for $j = 1, \cdots, m$
```

We claim that

> assignments As returned by the algorithm Approx-Max-Spread is a polynomial-time 2-approximation algorithm for finding maximum spread.

Proof. 3 parts

(a) First we prove that A_j for $j = 1, \cdots, m$ are valid assignments. Note we initialize the empty set A_j and then for each iteration choose to pick an arbitrary job i in S and assign the job i to some A in the queue. By loop exit condition, S is an empty set, therefore we have assigned all jobs $i \in S$ to some processor A. Therefore, the returned sets of A_j are valid assignments including every $i = 1, \cdots, n$

(b) Now we prove the algorithm runs in polynomial-time. Note all priority queue impmented with heap takes $O(\lg n)$ worst case running time. The 2 for loop iterates m and n times respectively, each doing some priority queue operation and constant time assignment operations. Therefore the total runtime $O((m+n)\lg n)$ is in polynomial-time

(c) Now prove that the algorithm is a 2-approximation algorithm. Let the run time of all jobs be $V = \sum_{i=1}^{n} v_i$. The best assignment possible happens when every processor gets an equal share of runtime, which is a fixed $\frac{V}{m}$, in other words,

$$\sum_{i \in A_j} v_i = \frac{V}{m} \quad \text{for all } j = 1, \cdots, m$$

Let s^* be the spread of an optimal assignment. s^* is therefore bounded above by the spread of the best possible assignment.

$$s^* \leq \frac{V}{m}$$

Now we claim that $s \geq \frac{V}{2m}$. Prove by contradiction, assume the final spread from the algorithm be $s_f < \frac{V}{2m}$. Let A_f be the process with least total runtime when the algorithm terminates, i.e.

$$T(A_f) = s_f \leq T(A_j) \text{ for all } j = 1, \cdots, m$$

Since all jobs are assigned when program terminates, there must be at least one A for which $T(A) > \frac{V}{m}$. Otherwise, the total runtime of assignment returned $\sum_{j=1}^{m} T(A_j) < V$. Let A_j be the first processor with $T(A_j) > \frac{V}{m}$. At some point, the algorithm picks A_j as the processor because it has determined that $A_j.key = T(A_j) = \sum_{i \in A_j} v_i$ is the minimum of all processor and adds the next job v_i to A_j (where $A'_j = A_j \cup \{i\}$) such that $T(A'_j) > \frac{V}{m}$. However note that no single job has a running time exceeding $\frac{V}{2m}$, specifically, $v_i \leq \frac{V}{2m}$. Then we have

$$T(A'_j) = T(A_j) + v_i \quad \rightarrow \quad T(A_j) = T(A'_j) - v_i \geq T(A'_j) - \frac{V}{2m}$$

$T(A'_j) > \frac{V}{m}$ implies $T(A_j) > \frac{V}{2m}$. We have A_f as a processor with $T(A_f) \leq \frac{V}{2m}$ when the program terminates. Since we have only increased keys to processors in the queue, A_f will have a total runtime $T(A_f) \leq \frac{V}{2m}$ during the step where the program decides to pick v_i and add to A_j. This is a contradiction because the algorithm guarantees to pick the processor with minimum total runtime but $T(A_f) \leq \frac{V}{2m} < T(A_j)$. Therefore, the claim that $s \geq \frac{V}{2m}$ holds. Hence,

$$s^* \leq \frac{V}{m} \leq 2s$$

So then we proved that the algorithm described is a 2-approximation algorithm for finding maximum spread.

$\square$