Worth: 5%

1. **[16 marks]**

   (a) **[5 marks]** Give a *detailed* argument that for all decision problems D and E, if $D \leq_p E$ and $E \in NP$, then $D \in NP$.

   (b) **[5 marks]** 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."

   (c) **[6 marks]** Show that if decision problem D is $coNP$-hard, then $D \in NP$ implies $NP = coNP$.

Solution. (a) Suppose $D \leq_p E$ 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, $V_E(x,c) =$ True for *some* c; for all no-instances x, $V_E(x,c) =$ 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 $D \leq_p E$).

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

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

$$\forall D' \in coNP, \quad D' \leq_p D.$$

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

***coNP* $\subseteq$ *NP*:** For all $D' \in coNP$, $D' \leq_p D$ (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. **[24 marks]**

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.*

For each of the following decision problems, recall that a cycle an an undirected graph is *simple* if it contains no repeated vertex or edge.

(a) **[8 marks]** ExactCycle:
Input: An undirected graph G and a non-negative integer k.
Question: Does G contain some simple cycle on **exactly** k vertices?

(b) **[8 marks]** SmallCycle:
Input: An undirected graph G and a non-negative integer k.
Question: Does G contain some simple cycle on **at most** k vertices?

(c) **[8 marks]** LargeCycle:
Input: An undirected graph G and a non-negative integer k.
Question: Does G contain some simple cycle on **at least** k vertices?

Solution. (a) ExactCycle $\in NP$: On input (G, k, c), where c is a list of vertices, verify that c contains exactly k vertices and that G contains every edge from one vertex in c to the next, and also from the last vertex back to the first one.
This takes only polynomial time and it outputs True for some value of c iff G contains some cycle on exactly k vertices.
ExactCycle $\notin P$: HamCycle $\leq_p$ ExactCycle (where HamCycle is the Hamiltonian Cycle, or "Rudrata Cycle", problem).

On input $G = (V, E)$, output (G, n), where $n = |V|$.

Clearly, (G, n) can be computed from G in polytime. Also, G contains a simple cycle on exactly n vertices iff G contains a Hamiltonian/Rudrata cycle, by definition.

(b) SmallCycle $\in P$:

```
for each v ∈ V:
    run BFS starting from v
    if BFS detects a cycle, traverse it to compute its length
    if the cycle contains k or fewer vertices:
        return True
return False    # after trying every v
```

This algorithm runs in polynomial time because it makes a linear number of calls to BFS (each one of which takes linear time), in addition to a linear number of cycle traversals (each one of which also takes linear time), in the worst-case.
Also, if the algorithm returns True, the graph contains some cycle on at most k vertices (because the algorithm has found such a cycle explicitly).
Finally, if the graph contains some cycle C on at most k vertices, then the algorithm returns True: when running BFS from one of the vertices on C, the algorithm will detect the existence of C, and the fact that C contains no more than k vertices (because BFS always finds paths, and cycles, that use as few edges as possible).
Note that using DFS instead of BFS would not be guaranteed to work because DFS may not find every short cycle in the graph.

(c) LargeCycle $\in NP$: On input (G, k, c), where c is a list of vertices, verify that c contains at least k vertices and that G contains every edge from one vertex in c to the next, and also from the last vertex back to the first one.
This takes only polynomial time and it outputs True for some value of c iff G contains some cycle on at least k vertices.
LargeCycle $\notin P$: HamCycle $\leq_p$ ExactCycle (where HamCycle is the Hamiltonian Cycle, or "Rudrata Cycle", problem).

On input $G = (V, E)$, output (G, n), where $n = |V|$.

Clearly, (G, n) can be computed from G in polytime. Also, G contains a simple cycle on at least n vertices iff G contains a Hamiltonian/Rudrata cycle, by definition.

□

3. **[20 marks]**

Consider the following "Partition" search problem.

Input: A set of integers $S = \{x_1, x_2, \ldots, x_n\}$ — each integer can be positive, negative, or zero.

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.)

(a) **[5 marks]** Give a *precise* definition for a decision problem "Part" related to the Partition search problem.

(b) **[15 marks]** 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. (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 ∑_{x∈S} |x|
    S_1 := ∅        # accumulator for the first subset
    t_1 := 0        # sum of elements of S_1
    S_2 := ∅        # accumulator for the second subset
    t_2 := 0        # sum of elements of S_2
    for each x ∈ S:
        S := S − {x}
        # see if it works to put x in S_1
        if (x + t_1 = t_2 and Part(S)) or (x + t_1 ≠ t_2 and Part(S ∪ {x + t_1 + T, t_2 + T})):
            S_1 := S_1 ∪ {x}
            t_1 := t_1 + x
        else:
            S_2 := S_2 ∪ {x}
            t_2 := t_2 + x
    return (S_1, S_2)
```

Correctness: Note that by construction, $T + \sum_{x \in S'} x \geq T/2 > x_i$ for all subsets $S' \subseteq S$ and all i. In other words, no matter which elements of S are added to T, the total is always positive and always strictly greater than every individual element of S.

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, $S = \emptyset$. 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 $t_1 = t_2 = 0$ and S can be partitioned into subsets (S_1', S_2') with equal sum.
- Suppose that $t_1 = t_2$ and S can be partitioned into (S_1', S_2'), at the start of one iteration. Then either $(S_1 \cup S_1', S_2 \cup S_2')$ or $(S_1 \cup S_2', S_2 \cup S_1')$ is a valid overall partition.
  Consider the element x removed from S and let S_1' be the subset that contains x.
  - 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.
  - If $x \neq 0$, then $x + t_1 \neq t_2$ and $S - \{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned into $\left(S_1' - \{x\} \cup \{x + t_1 + T\}, S_2' \cup \{t_2 + T\}\right)$. So the algorithm places x in S_1.
- Suppose that $t_1 \neq t_2$ and $S \cup \{t_1 + T, t_2 + T\}$ can be partitioned into (S_1', S_2'), at the start of one 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 $t_1 + t_2 + 2T \geq T$. 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.
  - If x can be placed in S_1, then either $x + t_1 = t_2$, in which case the rest of $S - \{x\}$ can be partitioned, or $x + t_1 \neq t_2$, in which case $S - \{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned. In either case, the algorithm correctly places x in S_1.
  - If x cannot be placed in S_1, then either $x + t_1 = t_2$, in which case the rest of $S - \{x\}$ cannot be partitioned, or $x + t_1 \neq t_2$, in which case $S - \{x\} \cup \{x + t_1 + T, t_2 + T\}$ cannot be partitioned. In either case, the algorithm correctly places x in S_2.

  In every situation, the loop invariant is maintained.

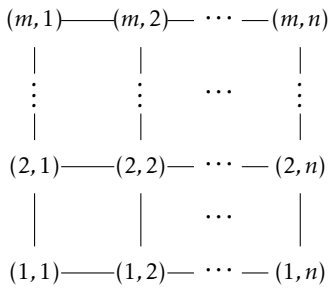
Runtime: Let $t(n)$ be the runtime of Part(S), where $n = |S|$. Then, PartSearch(S) runs in worst-case time $\mathcal{O}(n^2 + n\,t(n))$, because the main loop iterates n times, calling Part at most twice each time and taking no more than time $\mathcal{O}(n)$ to perform basic set operations.

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

□

4. **[20 marks]**

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!



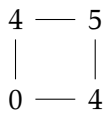
(a) **[5 marks]** Consider the following greedy algorithm to try and select street corners.

```
C := {(i, j) : 1 ≤ i ≤ m, 1 ≤ j ≤ n}    # C is the set of every available corner
S := ∅    # S is the current selection of corners
while C ≠ ∅:
    pick (i, j) ∈ C with the maximum value of p_{i,j}
    # Add (i, j) to the selection and remove it (as well as all corners adjacent to it) from C.
    S := S ∪ {(i, j)}
    C := C − {(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 larger profit.

(b) **[15 marks]** 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?)

Solution. (a) For the input below (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.

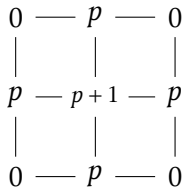
Let $A(G)$ be the smallest total profit of any independent set returned by the greedy algorithm, and $M(G)$ be the *maximum* profit of *all* independent sets of G. We prove that for all inputs G, $A(G) \geq M(G)/4$ — and that for at least one input G_0, $A(G_0) = M(G_0)/4$.

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 $p(v') \geq p(v)$ — 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 $p(v) \geq p(v_1)$, $p(v) \geq p(v_2)$, $p(v) \geq p(v_3)$, $p(v) \geq p(v_4)$. In other words, for

all $v \in S$, $4p(v) \geq p(v_1) + p(v_2) + p(v_3) + p(v_4)$, in the worst case, to "cover" all corners in T. Hence, $4p(S) \geq p(T)$, *i.e.*, $A(G) \geq M(G)/4$, as desired.

To show that $A(G)$ can be equal to $M(G)/4$, consider the input below. 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.



□

5. **[20 marks]**

You are using a multi-processor system to process a set of jobs coming in to the system each day. For each job $i = 1, 2, \ldots, n$, 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 $\{A_1, A_2, \ldots, A_m\}$. 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 $j = 1, 2, \ldots, m$ 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.

(a) **[15 marks]**

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 $s/2$. 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 $\frac{V}{2m}$.

Show that your algorithm achieves the required approximation ratio.

(b) **[5 marks]**

Give an example of an instance on which the algorithm you designed in part (5a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your instance, and what your algorithm finds.

Solution. (a)

ALLOCATE$(m, v[1,\ldots,n])$:
  $A[1] = \cdots = A[m] = []$ # $A[j]$ is the list of jobs assigned to processor j
  $a[1] = \cdots = a[m] = 0$ # $a[j]$ is the total time of jobs in $A[j]$
  for $i := 1, 2, \ldots, n$:
    # add job i to the processor with the current smallest total time
    find j such that $a[j]$ is minimum
    $A[j] := A[j] \cup \{v[j]\}$
    $a[j] := a[j] + v[j]$
  return $A[1], \ldots, A[m]$

For an arbitrary input $(m, v[1,\ldots,n])$, let OPT be the maximum spread possible and let GS be the spread of the solution returned by the algorithm (GS is short for "Greedy Spread"). We show that $GS \geq OPT/2$.

First, note that $OPT \leq V/m$. Otherwise, every processor would have a total strictly greater than V/m which would make the sum of the processors' runtime strictly greater than V, a contradiction.

Next, let $A[i], \ldots, A[m]$ be the solution returned by the algorithm, with total processor times $a[1], \ldots, a[m]$. Let $a[k]$ be the minimum processor time (so $a[k] \leq a[j]$ for all j and $GS = a[k]$).

For each processor j, let b_j be the running time of the *last* job added to $A[j]$. The fact that b_j is added to $A[j]$ instead of another processor means that $a[j] - b_j$ is no larger than any other processor total when b_j is added. In particular, $a[j] - b_j \leq a[k]$. Since $b_j \leq V/2m$, this means $a[j] \leq a[k] + V/2m$ for every j.

But this means $V = \sum_{j=1}^{m} a[j] \leq \sum_{j=1}^{m} (a[k] + V/2m) = m(a[k] + V/2m)$. Solving for $a[k]$ yields $a[k] \geq V/2m$.

Hence, $GS = a[k] \geq V/2m \geq OPT/2$, as desired.

(b) Consider the input $[6, 3, 3, 2, 2, 2]$.

The algorithm assigns job $[v_1] = [6]$ to A_1, jobs $[v_2, v_4, v_6] = [3, 2, 2]$ to A_2 (with total time 7) and jobs $[v_3, v_5] = [3, 2]$ to A_3 (with total time 5). So the spread of the algorithm's output is 5.

However, it is possible to achieve a spread of 6 by assigning jobs as follows: $A_1 = [v_1] = [6]$, $A_2 = [v_2, v_3] = [3, 3]$, $A_3 = [v_4, v_5, v_6] = [2, 2, 2]$.

□