First Name: Zhicheng

Last Name: Yan

Student ID: 1002643142

First Name: Zhihong

Last Name: Wang

Student ID: 1002095207

First Name: Kecheng

Last Name: Li

Student ID: 1001504507

We declare that this assignment is solely our own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters.

This submission has been prepared using LaTeX.

Question 1

(a). solution:
If $D \leq_p E$ and $E \in NP$,
Then, there is a function f which is polynomial-time computable.
$\forall$ 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.
$\exists$ a verifier $V_E(x, c)$ for E that is in polynomial time such that:
$\forall$ yes-instances x, $\exists c$, $V_E(x, c)$ = True;
$\forall$ no-instances x and $\forall$ c, $V_E(x, c)$ = False.
For D, let $V_D(x, c)$: return $V_E(f(x), c)$
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, $\exists c$, $V_D(x, c) =$ True
$\iff \exists c$, $V_E(f(x), c) =$ True (by the construction of V_D)
$\iff f(x)$ is a yes-instance for E (by definition, V_E is a verifuer for E)
$\iff x$ is a yes-instance for D (by definition, $D \leq_p E$).
Hence, V_D is a polytime verifier for D, so $D \in NP$, by definition.

(b). solution:
Decision problem D is coNP-hard $\iff \forall D' \in coNP, D' \leq_p D$

(c). solution:
If D is coNP-hard and $D \in NP$, then:

Case 1: $coNP \subseteq NP$:
$\forall$ $D' \in coNP, D' \leq_p D$ # D is coNP-hard
then $D' \in NP$ # $D \in NP$

Case 2: $NP \subseteq coNP$:
$\forall$ $D' \in NP, \overline{D'} \in coNP$ # by definition of coNP, $\overline{D'}$ is the complement of D'
then $\overline{D'} \in NP$ # $coNP \subseteq NP$ by Case 1
then $\overline{\overline{D'}} = D' \in coNP$

Therefore, $NP = coNP$.

Question 2
Assume $n = |V|, m = |E|$;
C is a list of vertices, the length of C is x, the index of C begin at 1;
HCP = Hamiltonian Cycle Problem.

(a). solution:

Case 1: $ExactCycle \in NP$:
Algorithm:

```
1  Verifier(G=(V, E), k, C):
2      if x not equal to k:
3          return False
4      if (C[x], C[1]) not in E:
5          return False
6      for i = 2 to x:
7          if (C[i-1], C[i]) not in E:
8              return False
9          for j = i-1 to 1:
10             if C[j] = C[i]:
11                 return False
12     return True
```

Worst Case Time Complexity:
By line 4, it costs n.
By line 6 to 9, they cost n^2.
So, the worst case time complexity is $O(n^2)$, which is polynomial time.

Justification:
This algorithm outputs true iff C contains exactly k non-repeating vertices with each vertex connecting to the next as an edge (i.e. G contains some cycle on exactly k vertices).
Therefore, the answer for (G, k) is true iff $\exists C$ such that $Verifier$ outputs true.

Case 2: $ExactCycle \notin P$: $HCP \leq_p ExactCycle$
When input $= G = (V, E)$, Output = (G, n)
(G, n) can be computed from G, and the complexity is in polytime (by loop over vertex set).
Therefore, G contains a simple cycle on exactly n vertices iff G contains a Hamiltonian Cycle (by definition from CLRS).

(b). solution:

SmallCycle $\in P$:
Algorithm:

```
1  SmallCycle(G=(V, E), k, C):
2      for all v in V:
3          run BFS on v
4          if exist cycle:
5              compute its length by traversing it
6          if the number of vertices in cycle <= k:
7              return True
8      return False
```

Worst Case Time Complexity:
By line 2 and 3, they cost $n(n+m)$
So, the worst case time complexity is $O(n^2 + nm)$, which is polynomial time.

Justification:
By line 6, if the algorithm outputs True, the graph contains some cycle on at most k vertices.
Our algorithm will check if there exist some cycle in G by using *BFS* (By CLRS, *BFS* can finds fewest edges cycles) on one of the vertices on C, and C contains less or equal to k vertices.

(c). solution:

Case 1: *LargeCycle* $\in NP$:
Algorithm:

```
1  Verifier(G=(V, E), k, C):
2      if x < k:
3          return False
4      if (C[x], C[1]) not in E:
5          return False
6      for i = 2 to x:
7          if (C[i-1], C[i]) not in E:
8              return False
9          for j = i-1 to 1:
10              if C[j] = C[i]:
11                 return False
```

```
12      return True
```

Worst Case Time Complexity:
By line 4, it costs n.
By line 6 to 9, they cost n^2.
So, the worst case time complexity is $O(n^2)$, which is polynomial time.

Justification:
Our algorithm returns True for some value of C iff G contains some cycle on at least k vertices. On input (G, k, C), the algorithm will verify if C contains at least k vertices and G contains every edge from one vertex in C to the next.

Case 2: $LargeCycle \notin P$: $HCP \leq_p LargeCycle$
When input $= G = (V, E)$, Output = (G, n)
(G, n) can be computed from G, and the complexity is in polytime (by loop over vertex set).
Therefore, G contains a simple cycle on exactly n vertices iff G contains a Hamiltonian Cycle (by definition from CLRS).

Question 3

Assume $n = |S|$, the time complexity of $Part$ is $p(n)$.

(a). solution:
$Part$ definition:
Input: $S = \{x_1, x_2, ..., x_n\}$ (a set of integers)
Question: Can we partition S into subsets S_1, S_2 and make sure $sum(S_1)$ is equal to $sum(S_2)$?
Time Complexity: $p(n)$

(b). solution:
Want To show: that Partition is polytime self-reducible.
Assume $\forall$ input sets S, S can be partitioned $\rightarrow$ $Part(S)$ = True.
Algorithm:

```
1  PartSearch(S):
2      if Part(S) == False:
3          return NIL
4      T = 2*(the sum of absolute values of all integers in S)
5      initialize subsets S1 and S2 as empty sets
6      initialize the total sum of subsets t1 and t2 as 0
7      for all elements x in S:
8          S = S\{x}
9          S3 = S union {x+t1+T, t2+T}
10         if x + t1 == t2 and Part(S):
11             S1 = S1 union {x}
12             t1 = t1 + x
13         elif x + t1 != t2 and Part(S3):
14             S1 = S1 union {x}
15             t1 = t1 + x
16         else:
17             S2 = S2 union {x}
18             t2 = t2 + x
19     return (S1, S2)
```

Worst Case Time Complexity:
By line 4, it costs n for summing up.
By line 7, it costs n for looping.
By line 8, it costs n for removing.
By line 2, 10 and 13, they cost $p(n)$ by assumption.

So, the worst case time complexity is $O(n(n+p(n)))$, which is polynomial time.
Therefore, Partition is polytime self-reducible.

Justification:
Our algorithm will return NIL if S cannot be partitioned. If S can be partitioned, then here we have the loop invariant:

$t_1 = t_2$ 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, $S = \emptyset$.
If $(t_1 \neq t_2) \wedge (t_1 + T > 0 \wedge t_2 + T > 0)$, the set $\{t_1 + T, t_2 + T\} = S \cup \{t_1 + T, t_2 + T\}$ cannot be partitioned.
So, we cannot get $t_1 \neq t_2$.
So, $t_1 = t_2$, (S_1, S_2) we assumed in part (a) is the partition of S.

Correctness of the loop invariant:
Prove by induction:

Base Case:
At the beginning of loops, S can be partitioned into empty subsets S_1, S_2, and t1 = t2 = 0.

Inductive Steps:
Inductive Hypothesis: Assume the loop invariant is True at the start of one iteration of the loop.

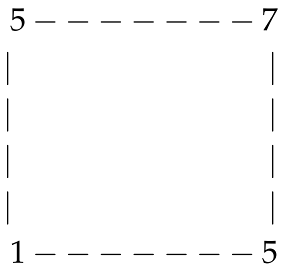
Case 1:
At the beginning of loop, assume $t_1 = t_2$ and S can be partitioned into (S_1, S_2), x removed from S, $x \in S_1$.
(i): If $x = 0$, then $x + t_1 = t_2$.
Our algorithm places x in S_1 because $S \backslash \{x\}$ can be partitioned by placing x in subset S_1 or S_2, which will not change the sums.
(ii): If $x \neq 0$, then $x + t_1 \neq t_2$.
Our algorithm places x in S_1 because (S_1, S_2) partitions S and $t_1 + T = t_2 + T$, so $S \backslash \{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned into $(S_1 \backslash \{x\} \cup$

$\{x + t_1 + T\}, S_2 \cup \{t_2 + T\})$.
In all cases, our algorithm places x in S_1.

Case 2:
At the beginning of loop, assume $t_1 \neq t_2$ and $S \cup \{t_1 + T, t_2 + T\}$ can be partitioned into (S_1, S_2), x removed from S, $t_1 + T \in S_1$ and $t_2 + T \in S_2$ because it is impossible for other numbers from S to add up to $t_1 + t_2 + 2T \geq T$, so $t_1 + T$ and $t_2 + T$ cannot both at the same subset of the partition.
(i): If $x \in S_1$, then no matter $x + t_1 = t_2$ or $x + t_1 \neq t2$. Because if $x \in S_1$, then $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned (because x is added to $t_1 + T$ as part of S_1).
(ii): If $x + t_1 = t_2$, then $x \cup S_1$ iff $S\backslash\{x\}$ can still be partitioned; otherwise, $x \cup S_2$. Because if $x \in S_1$, then $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned (because x is added to $t_1 + T$ as part of S_1); if $x \in S_2$, then $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ cannot be partitioned.
(iii): If $x + t_1 \neq t_2$, then $x \cup S_1$ iff $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ can still be partitioned; otherwise, $x \cup S_2$. Because if $x \in S_1$, then $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ can be partitioned (because x is added to $t_1 + T$ as part of S_1); if $x \in S_2$, then $S\backslash\{x\} \cup \{x + t_1 + T, t_2 + T\}$ cannot be partitioned.

Therefore, for all cases, the algorithm works.

Question 4
(a)



(1)

The graph (1) above show the counter example of the greedy algorithm indicated in the question. The algorithm will return the corners with profits 7 and 1, for a total profit 8. However picking the corners with profits 5 will give a total profit 10.

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

Let M(G) be the maximum profit of all independent sets of G returned by the greedy algorithm, and L(G) be the smallest total profit of any independent set of G.

Want to show: for all inputs G, $L(G) \geq M(G)/4$ and that for at least one input G_0, $L(G_0) = M(G_0)/4$.

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 exist some corner $u \in S$ such that $(u, v) \in E$ and $p(u) \geq p(v)$ according to the algorithm.

When v was removed from G by the algorithm, it was because of some adjacent corner u being added to S, which means that at that point, u had the largest profit among all remaining corners, including v.

Note no corner in S can have 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 $p(v) \geq p(v_1)$, $p(v) \geq p(v_2)$, $p(v) \geq p(v_3)$, $p(v) \geq p(v_4)$. By combine all those inequalities, we can see that $4p(v) \geq p(v_1) + p(v_2) + p(v_3) + p(v_4)$. In the worst case, it can cover all corners in T.

Therefore, $4p(S) \geq p(T)$, (i.e. $L(G) \geq M(G)/4$).

$$\begin{array}{c}
0----p---0 \\
| \qquad\quad | \qquad\quad | \\
| \qquad\quad | \qquad\quad | \\
p--p+1---p \\
| \qquad\quad | \qquad\quad | \\
| \qquad\quad | \qquad\quad | \\
0----p---0
\end{array} \tag{2}$$

To show that L(G) can be equal to M(G)/4, consider the input from graph (2). The greedy algorithm in the question will select the corners with profits $(p+1)+0+0+0+0 = p+1$. However the maximum profit is actually obtained by picking the corners $p+p+p+p = 4p$.

Although it is not exactly factor of 4, it can be made arbitrarily close by make p large enough.

We can simply set the profit of the middle corner p instead of p+1. However, our algorithm may no longer return the selection to be sub-optimal, but it's possible that the algorithm to choose the middle corner first, and finally come up with a solution whose profit value is exactly 1/4 of the optimum.

Question 5

(a)

(i) Algorithm Description

We would like to use greedy algorithm to approximately maximize the spreads. When scheduling each job, the greedy choice of the processor is that has the minimum total time of jobs assigned to it. Then, get the time spent of the processor that has minimum total running time.

(ii) Pseudo-code

```
job_schedule({j_1, ..., j_n}, {A_1, ..., A_m}):
    A_i.running_time = 0 for each i from 1 to m
    for each job j_i in {j_1, ..., j_n} do
        pick a processor A has minimum A.running_time
        assign j to A
        A.working_time += v_i
    endfor
    minimum_running_time := the running time of the processor
        that has the minimum running time
    return minimum_running_time
```

(iii) Show algorithm achieves the required approximation ratio

Let S be the spread, and $S*$ be the maximum spread By the assumption if the question, we know that $v_l \leq \frac{V}{2m}$ for each job from v_1 to v_n.

We want to show that $S \geq \frac{1}{2}S^*$:

By the definition of the spread, the spread is always less than or equal to the average running time for each processor(i.e. $S \leq \frac{V}{m}$ and $S^* \leq \frac{V}{m}$. Also, for all the processors A_1 to A_n. There must has a processor A_i which the total running time T_i is above or equal to the average running time (i.e. $T_i \geq \frac{V}{m}$), where i is an integer from 1 to m. Then before the job j_k which is the latest scheduled to the processor A_i is assigned to that processors. The total running time $T_i - v_k$ was the minimum of all processors according to the greedy choice(i.e.$T_i - v_k \leq S$). By the assumption, we know that $v_k \geq \frac{V}{2m}$, then $T_k - v_j \geq \frac{V}{m} - \frac{V}{2m} = \frac{V}{2m}$. We can get that $S \geq T_k = \frac{V}{2m} = \frac{1}{2} * \frac{V}{m} \geq \frac{1}{2}S^*$. Finally we get $S \geq \frac{1}{2}S^*$, which shows that the algorithm above achieved the required approximation ratio.

(b) Example: 9 jobs with a running times $v_1 = 5$,$v_2 = 4$,$v_3 = 3$,$v_4 = 2$,$v_5 = 4$,$v_6 = 4$,$v_7 = 5$,$v_8 = 5$,$v_9 = 4$ and there are $m = 3$ processors

The algorithm above gets the spread of 10 (assigned jobs $\{1, 6, 9\}$ to the

first processor, jobs $\{2, 5, 8\}$ to the second processor, jobs $\{3, 4, 7\}$ the third processor).
But the optimal solution gets the spread of 12 (assigned jobs $\{2, 5, 6\}$ to the first processor, jobs $\{1, 3, 9\}$ to the second processor, jobs $\{4, 7, 8\}$ the third processor).
So, the algorithm from part(a) does not get an optimal solution.