Name: _____

Entry number: _____

There are 5 questions for a total of 75 points.

---

1. (5 points) You are given n items and a sack that can hold at most W units of weight. The weight of the i^{th} item is denoted by $w(i)$ and the value of this item is denoted by $v(i)$. The items are indivisible. This means that you cannot take a fraction of any item. Design an algorithm that determines the items that should be filled in the sack such that the total value of items in the sack is maximized with the constraint that the combined weight of the items in the sack is at most W. You may assume that all the quantities in this problem are integers. Give pseudocode, discuss running time, and argue correctness.

Solution: Let $V(i, w)$ denote the maximum profit that can be made using items $1, ..., i$ when the sack capacity of w. Given this, we have $V(0, w) = 0$ for any $w \geq 0$ and $V(i, 0) = 0$ for any $i \geq 0$. For $i > 1$ and sack capacity $w > 0$, we have:

$$V(i,w) = \begin{cases} V(i-1,w) & \text{If } w(i) > w \\ \max\{V(i-1,w), V(i-1,w-w(i)) + v(i)\} & \text{otherwise} \end{cases}$$

Given a sack of size w, if the i^{th} item cannot fit into the sack, then the maximum profit is the same as the maximum profit obtained using items $1, ..., i-1$. If the i^{th} item can fit into the sack, then the maximum profit is the maximum of the cases where you pick item i and when you do not pick item i. This is basically the logic behind the above recursive formulation.s

The following algorithm finds the maximum profit as well as the items that should be picked for maximum profit.

```
0-1-knapsack(v, w, W)
  - For i = 0 to n
      - V[i, 0] ← 0
      - P[i, 0] ← "↑"
  - For t = 0 to W
      - V[0, t] ← 0
  - For i = 1 to n
      - For t = 1 to W
          - If (w[i] > t)
              - V[i, t] ← V[i − 1, t]
              - P[i, t] = "↑"
          - elseif (V[i − 1, t] > V[i − 1, t − w[i]] + v[i])
              - V[i, t] ← V[i − 1, t]
              - P[i, t] = "↑"
          - else
              - V[i, t] ← V[i − 1, t − w[i]] + v[i]
              - P[i, t] = "↖"
  - FindItems(w, P)
```

```
FindItems(w, P)
  - i ← n; t ← W; S ← {}
  - While (i ≠ 0)
      - If (P[i, t] = "↑")
          - i ← i − 1
      - else
          - S ← S ∪ {i}
          - i ← i − 1
          - t ← t − w[i]
  - return(S)
```

Running time: The running time of the above algorithm is $O(n \cdot W)$ since the algorithm fills tables of size $n \cdot W$ and filling each table entry takes $O(1)$ time.

2. (15 points) You are given n types of coin denominations of values $v_1 < v_2 < ... < v_n$ (all integers). Assume $v_1 = 1$, so you can always make change for any integer amount of money C. Design an algorithm that makes change for an integer amount of money C with as few coins as possible. Give pseudocode, discuss running time, and argue correctness.

Solution: Let $M(i, c)$ denote the minimum number of coins needed to make a change for c when we are allowed only coins of values $v_1, v_2, ..., v_i$. First we note that for all $c > 0$, $M(1, c) = c$ and also for all i, $M(i, 0) = 0$. Since for the first case, we are allowed coins of value 1 and for the second case, there is no change needed to be made. Here is a recursive relation for $M(.,.)$

$$M(i,c) = \begin{cases} M(i-1,c) & \text{If } v_i > c \\ \min\{M(i-1,c), M(i,c-v_i)+1\} & \text{Otherwise.} \end{cases}$$

If $v_i > c$, then the no coins of value v_i can be used. Otherwise, we consider these cases:

1. Number of coins used to construct the change for c using coins with value $v_1, ..., v_{i-1}$, and
2. Number of coins used to construct the change for $(c - v_i)$ using coins of value $v_1, ..., v_i$ and plus one (coin of value v_i).

We pick the case which minimizes the number of coins.

Here is the "table-filling" algorithm for finding the minimum number of coins:

```
CoinChange(C)
  - For i = 1 to n
      - M[i, 0] ← 0
  - For c = 1 to C
      - M[1, c] ← c
  - For i = 2 to n
      - For c = 1 to C
          - If (v_i > c)then M[i, c] ← M[i − 1, c]
          - else M[i, c] ← min{M[i − 1, c], M[i, c − v_i] + 1}
  - return(M[n, C])
```

Here is the algorithm that outputs the number of coins of each value:

```
CoinChangeNum(C)
  - For i = 1 to n
      - M[i, 0] ← 0
      - P[i, 0] ← −1
  - For c = 1 to C
      - M[1, c] ← c
      - P[1, c] ← c − 1
  - For i = 2 to n
      - For c = 1 to C
          - If (v_i > c)
              - M[i, c] ← M[i − 1, c]
              - P[i, c] ← −1
          - else
              - If (M[i − 1, c] < M[i, c − v_i] + 1)
                  - M[i, c] ← M[i − 1, c]
                  - P[i, c] ← −1
              - else
                  - M[i, c] ← M[i, c − v_i] + 1
                  - P[i, c] ← c − v_i
  - Coins ← FindCoins(P, C)
  - return(Coins)
```

```
FindCoins(P, C)
  - For i = 1 to n
      A[i] ← 0
  - c ← C; i ← n
  - While(c ≠ 0)
      - While (P[i, c] ≠ −1)
          - A[i] ← A[i] + 1
          - c ← P[i, c]
      - i ← i − 1
  - return(A)
```

Running time: The running time of the above algorithm is $O(n \cdot C)$ since the algorithm fills a table of size $n \cdot C$ and filling each table entry takes constant amount of time.

3. (15 points) Recall, the problem of finding a minimum vertex cover of a tree. Suppose, you are given a rooted tree T with root r. For every node v, let $C(v)$ denotes the set of children of the node v in T. So, for a leaf node v, $C(v) = \{\}$. We will try to find the minimum vertex cover using Dynamic Programming. For any node v, let $M(v, 0)$ denote the size of the vertex cover S of the subtree rooted at v such that $v \notin S$ and S is the smallest such cover. Let $M(v, 1)$ denote the size of the vertex cover S of the subtree rooted at v such that $v \in S$ and S is the smallest such cover. So, the size of the minimum vertex cover of the given tree is $\min\{M(r, 0), M(r, 1)\}$. Give the recursive formulation for $M(.,.)$ and use this to give an algorithm for finding the size of minimum vertex cover. Your algorithm should also output a minimum vertex cover of T. Give pseudocode, discuss running time, and argue correctness.

Solution: For any leaf v in the tree we have $M(v, 1) = 1$ and $M(v, 0) = 0$ since the subtree rooted at v contains just v. We get the following recursive relation for any internal vertex u in the tree:

$$\begin{aligned} M(u,1) &= 1 + \sum_{v \in C(u)} \min\{M(v,0), M(v,1)\}, \text{ and} \\ M(u,0) &= \sum_{v \in C(v)} M(v,1) \end{aligned}$$

The reason for the above recursion is that the size of the vertex cover of minimum size of the tree rooted at u that includes vertex u will be equal to 1 plus the size of the vertex covers of trees rooted at its children. Furthermore, the size of the vertex cover of minimum size that does not include u will be equal to the sum of sizes of vertex covers of the trees rooted at its children that includes the child (since edges from u to its children need to be accounted for). We can compute these values while doing a post-order traversal of the given tree.

```
MinVCSize(T, r)
  - PostOrder(r)
  - return(min {M[r,0], M[r,1]})

PostOrder(r)
  - If (r is a leaf)
      - M[r,0] ← 0
      - M[r,1] ← 1
  - else
      - For all v ∈ C(r)
          - PostOrder(v)
      - M[r,1] ← 1 + Σ_{v∈C(r)} min {M(v,0), M(v,1)}
      - M[r,0] ← Σ_{v∈C(r)} M[v,1]
```

For finding a minimum vertex cover, we will do a pre-order traversal of the tree. The pseudocode is given below. Assume that M and S are global variables.

`MinVC`(T, r)
- `PostOrder`(r)
- `PreOrder`$(r, 0)$
- return(S)

`PreOrder`(r, b)
- If $(b = 1)$
  - $S \leftarrow S \cup \{r\}$
  - For all $v \in C(r)$
    - `PreOrder`$(v, 0)$
- elseif $(M[r, 0] < M[r, 1])$
  - For all $v \in C(r)$
    - `PreOrder`$(v, 1)$
- else
  - $S \leftarrow S \cup \{r\}$
  - For all $v \in C(r)$
    - `PreOrder`$(v, 0)$

Running time: The algorithm does a tree traversal which will take $O(n)$ time.

4. (20 points) You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in \{1, ..., n\}$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired. Give pseudocode, discuss running time, and argue correctness.

Solution: The first observation that one should make for this question is that it makes sense to cut the cloth only at integer coordinates since all objects are of integer size. For any size $x \times y$, consider all products with size matching $x \times y$ and let $p(x, y)$ denote the profit of the product of size $x \times y$ that gives maximum profit. If there is no product of size $x \times y$, then $p(x, y) = 0$. Let $P(x, y)$ denotes the maximum profit that we can obtain from a cloth of size $x \times y$. There are two cases: (i) we do not cut this cloth and use it for a product of size exactly $x \times y$, or (ii) cut it (there being $(x - 1) + (y - 1)$ sub-options of this) and then make maximum profit out of the two pieces created. This gives us the following recursive relationship:

$$P(x, y) = \max \left\{ p(x, y), \max_{1 \le h < x} \{P(h, y) + P(x - h, y)\}, \max_{1 \le v < y} \{P(x, v) + P(x, y - v)\} \right\}$$

Here is a "table-filling" algorithm that computes the above values:

`ClothCutting`(X, Y)

- Compute the values of $p[x, y]$ for all $1 \le x \le X$ and $1 \le y \le Y$
- For $x = 1$ to X
  - For $y = 1$ to Y
    - $P[x, y] \leftarrow \max \{p[x, y], \max_{1 \le h < x} \{P[h, y] + P[x - h, y]\}, \max_{1 \le v < y} \{P[x, v] + P[x, y - v]\}\}$
- return$(P[X, Y])$

Running time: Computing the values of $p(x, y)$ for all x, y takes time $O(X \cdot Y + n)$. The algorithm fills a table of size $X \cdot Y$ and filling each table entry takes time $O(X + Y)$. So, the total running time of the algorithm is $O(X \cdot Y \cdot (X + Y) + X \cdot Y + n)$.

5. (20 points) Consider the 0-1 Knapsack problem that we discussed in lecture. Given n positive integers $w_1, ..., w_n$ and another positive integer W, the goal is to find a subset S of indices such that $\sum_{i \in S} w_i$ is maximised subject to $\sum_{i \in S} w_i \leq W$. We studied a dynamic program for this problem that has a running time $O(n \cdot W)$. We know that this is bad when the integers are very large. The goal in this task is to improve the running time at the cost of optimality of the solution. Let us make this more precise. Let OPT denote the value of the optimal solution. For any given $\varepsilon \in (0, 1]$, design an algorithm that outputs a solution that has value at least $\frac{OPT}{(1+\varepsilon)}$. The running time of your algorithm should be of the form $O\left(\frac{f(n)}{\varepsilon}\right)$, where $f(n)$ is some polynomial in n. You need not give the best algorithm that exists for this question. As long as $f(n)$ in the running time is polynomial in n, you will get the full credit.
(Hint: Try modifying the dynamic programming solution for the knapsack problem.)

Solution: Let us assume that for all i, $w_i \leq W$. Since otherwise we can remove the item i and then solve the problem with the remaining items. We will first check if $\sum_i w_i \leq W$. If so, the optimal solution will pick all items. So, now we only need to focus on the case when $\sum_i w_i > W$. Let w be the weight of the item with maximum weight. Since $\sum_i w_i > W$, we have:

$$n \cdot w > W \tag{1}$$

A DP that does not work: Let us try to use the same Dynamic Programming formulation as in the class and try to "scale". We will see that this formulation has an issue. However, this will nicely set us up for the DP that works.

Given the problem instance $I = (w_1, ..., w_n, W)$ we create the following new instance of the 0-1 knapsack problem $I' = (w'_1, ..., w'_n, W')$, where for all $i, w'_i = \lfloor w_i/m \rfloor$, $W' = \lfloor W/m \rfloor$, and $m = \lfloor w/(2n/\varepsilon) \rfloor$. We solve the second instance using the dynamic programming that we discussed in the class. Let O' denote the optimal solution for $(w'_1, ..., w'_n, W')$. That is, O' is the set of items picked by the dynamic programming algorithm when executed on instance I'. Note that the running time of the algorithm in instance I' will be $O(n \cdot W') = O(n \cdot W/m) = O\left(\frac{n^2}{\varepsilon} \cdot \frac{W}{w}\right) = O(n^3/\varepsilon)$ (using inequality (1)). Let us analyze if we can return O' as an approximate solution to instance I. The fact that $\sum_{i \in O'} w'_i \leq W'$ does not ensure that $\sum_{i \in O'} w_i$ will be at most W. This is where the problem lies with this formulation.

We will use an alternative DP formulation. Consider $w'_1, ..., w'_n$ as in the above DP formulation. Let

$$L(i, w) = \min_{S \subseteq \{1,...,i\} \text{ s.t. } \sum_{i \in S} w'_i = w} \left\{ \sum_{i \in S} w_i \right\}.$$

For base cases, we have $L(i, 0) = 0$ for all $i \geq 0$ and $L(0, w) = \infty$ for all $w > 0$. Furthermore, we can write:

$$L(i, w) = \begin{cases} \min\{L(i-1, w - w'_i) + w_i, L(i-1, w)\} & \text{If } w'_i \leq w \\ L(i-1, w) & \text{Otherwise} \end{cases}$$

We can compute the values $L(i, w)$ similar to the way we compute the values $M(i, w)$ as in the previous formulation. We can also output the set $S \subseteq \{1, ..., i\}$ that minimizes $\sum_{i \in S} w_i$ such that $\sum_{i \in S} w'_i = w$ for a given (i, w). The following program computes these values:

```
0-1-Knap(w', w, W)
  - For i = 0 to n
      - L[i, 0] ← 0
      - P[i, 0] ← "↑"
  - For t = 0 to W
      - L[0, t] ← ∞
  - For i = 1 to n
      - For t = 1 to W
          - If (w'[i] > t)
              - L[i, t] ← L[i − 1, t]
              - P[i, t] = "↑"
          - elseif (L[i − 1, t] ≤ L[i − 1, t − w'[i]] + w[i])
              - L[i, t] ← L[i − 1, t]
              - P[i, t] = "↑"
          - else
              - L[i, t] ← L[i − 1, t − w'[i]] + w[i]
              - P[i, t] = "↖"
```

```
FindItems(w', P, t)
  - i ← n; S ← {}
  - While (i ≠ 0)
      - If (P[i, t] = "↑")
          - i ← i − 1
      - else
          - S ← S ∪ {i}
          - i ← i − 1
          - t ← t − w'[i]
  - return(S)
```

Let t be the largest index such that $L(n,t) \leq W$. Then we run the program `FindItems(`w', P, t`)` to find a set S. We return S as the approximate solution to the original problem.

First note that $\sum_{i \in S} w_i \leq W$. This follows from the definition of L. Let O denote the optimal solution to the given problem. We prove the following claim.

Claim 1: $\sum_{i \in O} w'_i \leq t$.

Proof. Since otherwise t will not be the largest index such that $L(n,t) \leq W$. □

Finally, we have

$$\sum_{i \in S} w'_i = t \geq \sum_{i \in O} w'_i$$

Multiplying both sides by m, we get:

$$
\begin{aligned}
& m \cdot (\sum_{i \in S} w'_i) \geq m \cdot (\sum_{i \in O} w'_i) \\
\Rightarrow \quad & \sum_{i \in S} w_i \geq m \cdot \left(\sum_{i \in O} \lfloor w_i / m \rfloor \right) \\
\Rightarrow \quad & \sum_{i \in S} w_i \geq \sum_{i \in O} w_i - mn \\
\Rightarrow \quad & \sum_{i \in S} w_i \geq OPT - (\varepsilon/2)w \\
\Rightarrow \quad & \sum_{i \in S} w_i \geq OPT - (\varepsilon/2) \cdot OPT \quad (\text{since } w \leq OPT) \\
\Rightarrow \quad & \sum_{i \in S} w_i \geq (1 - \varepsilon/2) \cdot OPT \geq \frac{OPT}{1+\varepsilon}
\end{aligned}
$$