1.5em 0pt

CS218, Spring 2025, **Due: 11:59pm, Fri 5/9, 2025**
Assignment #3 [1] **Required Programming Due: 11:59pm, Fri 5/2, 2025**
Name: Lakhan Kumar Sunilkumar
Student ID: 862481700

---

1 (2.5 + 0.5 pts) Everything on sale!

1. (0.3 pt)

   So let's assume that an optimal set S contains two items i, j in the same group g, and while keeping the condition $(p_i - t_i) \geq (p_j - t_j)$. In S, i is bought for at discount (so, a benefit i.e $p_i - t_i$) while j is full-price (it is zero benefits).
   Suppose we take out j. In that case, the total cost decreases but the total benefit remains the same, so the new set is at least as good in benefit and uses fewer items which is contradicting the optimality. Therefore, no two items from one group can both be chosen.

   Removing such full-price items reduces the total cost without reducing the total benefit, and it also aligns with Yihan's preference to carry fewer items. Hence, the optimal strategy includes at most one item per group.

2. (0.4 pt)

   By using a group knapsack DP with space optimization to maximize total benefit under budget W.

   Let us consider a *DP array:*

   $$dp[w] = \max\{\text{total benefit achievable with total cost } \leq w\},$$

   for $0 \leq w \leq W$. Initialize

   $$dp[w] = 0 \quad \text{for all } 0 \leq w \leq W.$$

   Group items by their group ID g_i.
   For each group k:
   (a) Make a copy: $\text{prev}[w] \leftarrow dp[w]$ for all w.
   (b) For each item i in group k with sale cost t_i and benefit $b_i = p_i - t_i$, update in reverse:

   $$\text{for } w = W, W-1, \ldots, t_i: \quad dp[w] \;=\; \max\big(dp[w],\ \text{prev}[\,w - t_i\,] + b_i\big).$$

   From this we can understand that $dp[W]$ is the highest total benefit achievable within budget W.

   Time and Space Complexity:
   Each item is processed once for each feasible w, so the overall time is $\mathcal{O}(nW)$. We use $\mathcal{O}(W)$ space.

3. (0.8 pt)

   For solving Yihan's preference for carrying fewer items, the dynamic programming algorithm from Question 2 is modified to track both the maximum benefit and the minimum number of items required to achieve it.

   We define a DP table $dp[w] = (\text{benefit}, \text{items})$, where:

   - `benefit` is the maximum benefit achievable with total cost w,
   - `items` is the fewest number of items used to achieve that benefit.

   Let's initialize the array as:

   $$dp[0] = (0, 0), \quad \text{and all other } dp[w] = (-\infty, \infty)$$

   And for each group of items, we create a temporary copy of the DP array (to prevent using more than one item from the same group). Then, for each item i in the group (with sale price t_i and benefit $b_i = p_i - t_i$), we update the DP as follows: 1

   $$dp[w] = \max\left(dp[w],\ (dp'[w - t_i].\text{benefit} + b_i,\ dp'[w - t_i].\text{items} + 1)\right)$$

[1] Some of the problems are adapted from existing problems from online sources. Thanks to the original authors.

Here, the comparison is made such that:

- A higher benefit is preferred,
- If benefits are equal, the smaller item count is preferred.

This makes sure that for any given budget that Yihan is provided with, we are tracking both the highest benefit and the minimal number of items needed to achieve it.

At the end, we check all $dp[w]$ for $w \leq W$ and select the one with which has the highest benefit and among those, we report the smallest item count.

This algorithm maintains a time complexity of $\mathcal{O}(nW)$ and solves the problem efficiently whilst considering both constraints: maximizing benefit and minimizing the number of items.

Time and Space Complexity

$$\text{Time: } O(nW), \quad \text{Space: } O(W).$$

4. (0.5 pt)

To analyze and calculate the list of items that gives the highest benefit with the fewest number of items (as computed in Question 3), let us consider these changes in the DP table to include item tracking.

Let us consider this: for each total cost w, maintain:

$$dp[w] = (\text{benefit}, \text{items}, \text{trace})$$

where:

- `benefit` is the maximum benefit for cost w,
- `items` is the fewest number of items used,
- `trace` is the list of item indices used to achieve this result.

Let us consider these as the initialization:

$$dp[0] = (0, 0, \emptyset), \quad \text{all other } dp[w] = (-\infty, \infty, \emptyset)$$

So for each group, we should make a temporary copy of dp. For each item i in the group with sale price t_i and benefit $b_i = p_i - t_i$, update:

$$dp[w] = \max\left(dp[w],\ (dp'[w - t_i].\text{benefit} + b_i,\ dp'[w - t_i].\text{items} + 1,\ dp'[w - t_i].\text{trace} \cup \{i\})\right)$$

As before, the comparison is done by:

- Preferring a higher benefit,
- For equal benefits, preferring fewer items.

After processing all groups, scan $dp[w]$ for $w \leq W$ and select the entry with the maximum benefit and fewest items. The corresponding trace gives the list of items to buy.

Time and Space Complexity:
Time Complexity: $\mathcal{O}(nW)$, as we update each budget entry at most once per item.

Space Complexity: $O(W)$

5. (0.5 pt)

For this scenario, Yihan is given the chance to purchase an unlimited number of units of each item, and the discount price t_i applies to every unit. The goal is to maximize the total benefit without exceeding the budget W.

To solve this, we use a variation of the **unbounded knapsack problem**. Since all items still belong to the groups and we know that only one type per group can be chosen, we now need to consider that for each group we should use the best single item to use with unlimited copies.

Initialization

$$dp[0] \leftarrow 0, \qquad dp[w] \leftarrow -\infty \quad (\forall\, 1 \leq w \leq W).$$

Plan of Attack:

(a) For each group, select one item (type) to use.

(b) For each such selected item i with cost t_i and benefit $b_i = p_i - t_i$, calculate its total contribution using an unbounded knapsack DP.

(c) For each budget value w, try all multiples k such that $k \cdot t_i \leq w$, and update:

$$dp[w] = \max(dp[w],\ dp[w - k \cdot t_i] + k \cdot b_i)$$

This will be in a loop for each group, and the final $dp[W]$ gives the maximum achievable benefit.

Time Complexity:
For each item, we may loop through up to W budget values and should check all multiples k, resulting in $\mathcal{O}(nW^2)$.

This updated approach correctly handles the unbounded case while still respecting group constraints, producing the highest possible benefit within budget.

6. (0.5 pt bonus)

Yes, we can optimize the algorithm from Problem 5 to run in $\mathcal{O}(nW)$ time by applying an efficient unbounded knapsack dynamic programming strategy.

In an unbounded knapsack, not all multiples k of an item should be checked every time. Instead, we can iterate through budget values in increasing order and update the DP array in-place, allowing each item to be reused multiple times naturally.

Initialization

$$dp[0] \leftarrow 0, \quad dp[w] \leftarrow -\infty \quad (\forall\, 1 \leq w \leq W).$$

Plan of attack:

(a) Group items by their group ID.

(b) For each group, we select the single best item i to use as we know that only one item type per group is allowed.

(c) For this item with cost t_i and benefit $b_i = p_i - t_i$, make the changes in the DP array for all $w \geq t_i$:

$$dp[w] = \max(dp[w],\ dp[w - t_i] + b_i)$$

So this approach ensures that multiple copies of item i are considered while still making sure that the one-item-per-group is still restricted.

So by constantly iterating forward over the budget and by allowing repeated inclusion of each selected item type per group, we achieve the same result as in Problem 5 but with significantly better efficiency.

Time Complexity:
For each of the n items, we iterate over the budget once, resulting in $\mathcal{O}(nW)$ total runtime.

2 Morse Code (2 pts)

1. (0.2 pt)

   The Morse code for the word `UCR` is:

   $$\texttt{U} = \texttt{..-}, \quad \texttt{C} = \texttt{-.-.}, \quad \texttt{R} = \texttt{.-.}$$

   So, the concatenated Morse code without separators is:

   $$\texttt{..--.-..-.}$$

   Because Morse code is not a prefix code, this sequence can be interpreted in multiple valid ways. Here are three example interpretations:

   - `UCR`: `..- -.-. _-.` $\rightarrow$ 1 (U) + 1 (C) + 2 (R) = 4 strokes
   - `UKF`: `..- -.- _-.` $\rightarrow$ 1 (U) + 2 (K) + 3 (F) = 6 strokes
   - `EEQF`: `. _ --.- _-.` $\rightarrow$ 3 (E) + 3 (E) + 2 (Q) + 3 (F) = 11 strokes

   Among these, `UCR` has the fewest strokes: **4**.

2. (1.2 pt)

   A Morse code string X (consisting of dots `.` and dashes `-`) is given, and we are supposed to find the interpretation that uses the fewest total strokes in a handwriting. As we know that, Morse code is not prefix-free, multiple interpretations of X are possible. From that we are looking out for the one which has the minimal stoke count

   Plan of Attack:
   Use Dynamic Programming

   We use a one-dimensional DP array $f[i]$ for setting the minimum number of strokes needed to interpret the first i characters of the Morse string X. Let the length of X be n, and index X from 1 to n.

   We initialize:

   $$f[0] = 0 \quad \text{(no characters means 0 strokes)}$$
   $$f[i] = \infty \quad \text{for } i > 0$$

   We then build the solution using the following recurrence:

   $$f[i] = \min_{\substack{0 \le j < i \\ c \in A}} \{ f[j] + s[c] \qquad \text{if } \texttt{check}(X, j+1, i, c) = \texttt{true} \}$$

   - Here, $s[c]$ denotes the number of strokes needed to write letter c.
   - The function $\texttt{check}(X, j+1, i, c)$ returns `true` if the substring from position $j+1$ to i in X matches the Morse code for character c.
   - The algorithm tries all valid cuts j and all characters $c \in A$ (A–Z), checking whether the suffix $X[j+1..i]$ matches any Morse code.

The minimum number of strokes required to decode the full string is stored in $f[n]$, where n is the length of X.

Time Complexity
By Assuming that the max length of any Morse code known is constant, in this case it is at most 4 for this problem, and that the `check`(X, i, j, c) function runs in $\mathcal{O}(1)$ time, the overall time complexity is:

$$\mathcal{O}(n \cdot L), \quad \text{where } L = \max \text{ Morse code length} \leq 4$$

Hence, it is effectively $\mathcal{O}(n)$ for practical purposes, since we only try up to 26 letters and substrings of limited size at each step.

3. (0.3 pt)

The time complexity of the algorithm is $\mathcal{O}(n \cdot L \cdot 26)$, where:

- n is the length of the Morse code string X,
- L is the maximum possible length of a Morse code character (which is at most 4),
- 26 is the number of uppercase English letters (A–Z).

Since both L and 26 are constants, the time complexity simplifies to $\mathcal{O}(n)$

Therefore, the algorithm runs in linear time with respect to the input size.

4. (0.3 pt)

We are given the Morse code string:

$$X = \texttt{--..--.}$$

Using the dynamic programming algorithm from Question 2, we compute the fewest number of strokes by checking all valid segmentations of the string that match any letter's Morse code.

One optimal interpretation is:

$$\texttt{--..}\ (Z), \quad \texttt{--.}\ (G)$$

This corresponds to the word "ZG", and the stroke counts are:

$$s(\texttt{Z}) = 1, \quad s(\texttt{G}) = 2$$

So the total number of strokes is:

$$1 + 2 = 3$$

The smallest number of strokes is **3**.