1. (a) *Algorithm Description*

We proceed with dynamic programming. Specifically, for any integer $0 \leq i \leq n$ and $0 \leq j \leq M$, we split the original problem into individual subproblem instances of the form $f[i][j]$, each representing a boolean value that is True *iff* there exists a multiset of items $1, \dots, i$ of size j. With a abuse of set notation we define:

$$f[i][j] = 1 \iff \exists\, S \subseteq \{1, \dots, i\} \ni \sum_{i \in S} a_i = j$$

At iteration i, we discuss the progress made by our dynamic program in one of the following ways:

(1) Item i occurs at least once in S. In this case, we select a multiset of items $1, \dots, i$ that has size exactly $j - a_i$. Here $f[i][j - a_i]$ indicates whether such multiset exists.

(2) Item i does not occur in S. In this case, the solution maps to a multiset of the items $1, \dots, i-1$ of size j. The answer of whether such multiset exists is found in $f[i-1][j]$.

Either one of the above avenues yields a solution, so $f[i][j]$ is True if at least one of the decision possibilities above results in a solution. Furthermore, we can formalize these decision states as a dependency relation among the subproblems of f. Namely:

$$f[i][j] = \begin{cases} f[i-1][j] & \text{if } j < a_i \\ \max\{f[i-1][j], f[i][j-a_i]\} & \text{otherwise} \end{cases}$$

The base case follows from the definition of $f[i][j]$, i.e., $f[i][0] = 1$ for all $0 \leq i \leq n$ and $f[0][j] = 0$ for all $1 \leq j \leq M$. The answer we are after is $f[n][M]$. Since we do not know where it resides, we try all possible pairs (i, j) and check if any of them evaluate to True—and this is achieved by taking the max across the intermediate answers to $f[n][M]$.

Note this is a two-dimensional dynamic programming problem with table size $(n+1, M+1)$. Each table entry is a boolean value (True/False) capturing the inclusion (or exclusion) of item i in the final multiset S. Each such decision incurs an $\mathcal{O}(1)$ work (deciding among the two previous precomputed subproblems which one is True) so the total running time is in the order of $\mathcal{O}(nM)$.

(b) *Algorithm Pseudocode*

Algorithm 1 Knapsack(A, M)

Input: A list of n item sizes $A[1 \cdots n]$ together with an integer M—the size of the knapsack.
Output: "yes" *iff* there exists a multiset S of items of A whose total size is equal to M; "no" otherwise.

```
1:  f[i][0] ← 1 for each i ∈ {0 ··· n}
2:  f[0][j] ← 0 for each j ∈ {1 ··· M}
3:  for i ← 1 to n do
4:      for j ← 1 to M do
5:          if j < A[i] then.
6:              f[i][j] ← f[i − 1][j]
7:          else
8:              f[i][j] ← max{f[i − 1][j], f[i][j − A[i]]}
9:          end if
10:     end for
11: end for
12: if f[n][M] = 1 then
13:     return "yes"
14: else
15:     return "no"
16: end if
```

2. (a) *Algorithm Description*

We proceed with dynamic programming. Specifically, for any integer $0 \leq i \leq n$ and $0 \leq j \leq M$, we split the original problem into individual subproblem instances of the form $f[i][j]$ denoting the maximum attainable value obtained using a knapsack of capacity j and the first i items $1, \ldots, i$. At iteration i, we face a decision problem; that is, either item size a_i is needed to achieve the optimal solution, or it isn't needed. Since we want to maximize $\sum_{a_i \in S} v_i$, we select the most profitable result among the two following decision states:

(1) Fill the knapsack with item i. In this case, we must select a subset of items $1, \ldots, i-1$ that have a combined size at most $j - a_i$. Assuming we do all the preceding $i-1$ choices correctly, then we will get $f[i-1][j-a_i]$ value out of items $1, \ldots, i-1$, so the cumulative total will be $f[i-1][j-a_i] + v_i$.

(2) Don't fill the knapsack with item i, so we'll re-use the optimal solution for items $1, \ldots, i-1$ that have an aggregate size at most j. That answer rests in $f[i-1][j]$.

We can formalize (1) and (2) as the following dependency relation among the subproblems of f:

$$f[i][j] = \begin{cases} f[i-1][j] & \text{if } j < a_i \\ \max\{f[i-1][j], f[i-1][j-a_i] + v_i\} & \text{otherwise} \end{cases}$$

Note the initial conditions for this problem are $f[i][0] = 0$ for all $0 \leq i \leq n$ and $f[0][j] = 0$ for all $0 \leq j \leq M$. The answer we seek is $f[n][M]$. Since we do not know which pair (i, j) yields $f[n][M]$, we try all possible combinations of i's and j's and pick the maximal among them. The pseudocode for this is provided below and boils down to filling out a two dimensional table of size $(n+1, M+1)$. Each entry of the table incurs an $\mathcal{O}(1)$ work (deciding among the two previous precomputed subproblems which one is larger) so the total running time is $\mathcal{O}(nM)$.

(b) *Algorithm Pseudocode*

Algorithm 2 KNAPSACK(A, V, M)

Input: A list of n item sizes $A[1 \cdots n]$, its corresponding values $V[1 \cdots n]$, and an integer M—the size of the knapsack.

Output: The maximum attainable value with a knapsack of size M.

```
1:  f[i][0] ← 0 for each i ∈ {0 ··· n}
2:  f[0][j] ← 0 for each j ∈ {0 ··· M}
3:  for i ← 1 to n do
4:      for j ← 1 to M do
5:          if j < A[i] then
6:              f[i][j] ← f[i − 1][j]
7:          else
8:              f[i][j] ← max{f[i − 1][j], f[i − 1][j − A[i]] + V[i]}
9:          end if
10:     end for
11: end for
12: return f[n][M]
```

3. (a) *Algorithm Description*

The maximum sum common subsequence (MSCS) problem has identical decision states to the longest common subsequence (LCS) problem discussed in class. So, without prizes for guessing, we proceed with dynamic programming. Specifically, for integers $0 \leq i \leq n$ and $0 \leq j \leq m$, we introduce individual subproblem instances of the form $f[i][j]$ representing the MSCS attainable from inputs $A[1 \cdots i]$ and $B[1 \cdots j]$. When computing $f[i]$[j], we take into account one of the following decision states:

(1) $A[i] = B[j]$. Then clearly the MSCS is obtained by maximizing the sum of the subsequences $A[1 \cdots i-1]$ and $B[1 \cdots j-1]$, then adding $A[i]$ (or $B[j]$) to both subsequences. Notice, however, that there is a small caveat here, namely: $A[i]$ could be negative! Since we want to maximize the sum we add $A[i]$ only if $A[i]$ is positive. This should satisfiably handle negative numbers in $A[1 \cdots i]$ and $B[1 \cdots j]$ and make our dynamic program robust against edge cases of this nature.

(2) $A[i] \neq B[j]$. Then clearly the optimal subsequence yielding MSCS is located in the shorter version of $A[1 \cdots i]$ and $B[1 \cdots j]$. Since we do not know which one it is, we solve both $f[i-1][j]$ and $f[i][j-1]$ and pick the corresponding result that yields the maximal sum among them.

Formalizing the aforementioned decision states as a dependency relation across the subproblems of f, we see that:

$$f[i][j] = \begin{cases} f[i-1][j-1] + \max\{A[i], 0\} & \text{if } A[i] = B[j] \\ \max\{f[i][j-1], f[i-1][j]\} & \text{if } A[i] \neq B[j] \end{cases}$$

The base case follows from the definition of $f[i][j]$, i.e., $f[i][0] = 0$ for all $0 \leq i \leq n$ and $f[0][j] = 0$ for all $0 \leq j \leq m$. We are after $f[n][m]$, but since we do not know which pairs of indices (i, j) yield $f[n][m]$, we try all permissible choices of i's and j's and pick the maximal one among them.

Due to the optimal subproblem structure similarity to LCS, we see that MSCS is also a two dimensional dynamic program of table size $(n+1, m+1)$. Retrieving the value of $f[i][j]$ at any given iteration takes $\mathcal{O}(1)$. There are $(n+1)(m+1)$ entries in the table, so we get a final, total running time of $\mathcal{O}(nm)$ as required! The pseudocode for this is provided below.

(b) *Algorithm Pseudocode*

Algorithm 3 MAX-SUM-COMMON-SUBSEQUENCE(A, B)

Input: An array $A[1 \cdots n]$ of n numbers and an array $B[1 \cdots m]$ of m numbers.
Output: The maximum sum common subsequence of A and B.

```
f[i][0] ← 0 for each i ∈ {0 ··· n}
f[0][j] ← 0 for each j ∈ {0 ··· m}
for i ← 1 to n do
    for j ← 1 to m do
        if A[i] = B[j] then
            f[i][j] ← f[i − 1][j − 1] + max{A[i], 0}
        else
            f[i][j] ← max{f[i][j − 1], f[i − 1][j]}
        end if
    end for
end for
return f[n][m]
```

4. (a) *Algorithm Description*

We proceed with dynamic programming. Specifically, for any $1 \leq i \leq n$, we split the original problem into individual subproblem instances of the form $f[i]$ representing the length of the longest monotonically increasing subsequence in $A[1 \cdots i]$ that ends with $A[i]$.

We compute the value of $f[i]$ by considering all of the $i-1$ elements in A both previous to $A[i]$ and smaller than it, i.e., $\{j \in [1 \cdots i-1] \ni A[i] > A[j]\}$. These are the potential candidates that A could be appended to in an increasing subsequence. Since we are after the longest one, we choose the $f[j]$ with maximal value, and set $f[i] = 1 + f[j]$ (thereby effectively adding $A[i]$ to the end of the longest monotonically increasing subsequence possible). So we have:

$$f[i] = 1 + \max\{f[j] \ni 1 \leq j < i \wedge A[i] > A[j]\}$$

The base case rests on $f[1] = 1$. Note $f[i]$ depends on all of the elements before it, so when we create our dynamic programming table, we start with the base case $f[1]$ and then progressively fill it from left to right. Once we determine the values for all $f[i]$, we just select the one with maximum value, and the algorithm is complete. Recovering the actual subsequence is a near trivial task of keeping track of which $A[i]$'s are added to the sequence and then backtrack them. The pseudocode for this is given below and has a time complexity of $\mathcal{O}(n^2)$ incurred by the two `for` loops in Steps 6 and 8.

(b) *Algorithm Pseudocode*

Algorithm 4 LONGEST-MONOTONICALLY-INCREASING-SUBSEQUENCE(A)

Input: An array $A[1 \cdots n]$ of n distinct numbers.
Output: The longest monotonically increasing subsequence S of A and its length l.

```
 1: l ← 0
 2: k ← 0
 3: S ← ∅
 4: p[1 ··· n] ← 0                          // predecessors for each i's -- useful for reconstructing S
 5:
 6: for i ← 1 to n do
 7:     f[i] ← 1
 8:     for j ← 1 to i − 1 do
 9:         if A[i] > A[j] and f[i] < f[j] + 1 then
10:             f[i] ← f[j] + 1
11:             p[i] ← j
12:         end if
13:     end for
14:     if l < f[i] then
15:         l ← f[i]
16:         k ← i
17:     end if
18: end for
19:
20: repeat
21:     S ← S ∪ {A[k]}
22:     k ← p[k]
23: until k = 0
24:
25: S ← REVERSE(S)
26: return (S, l)
```