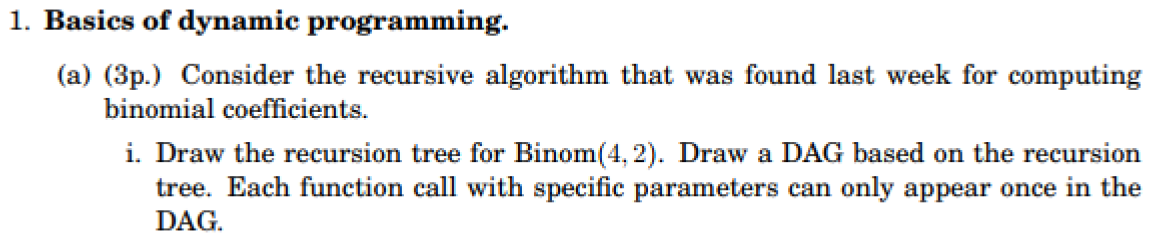
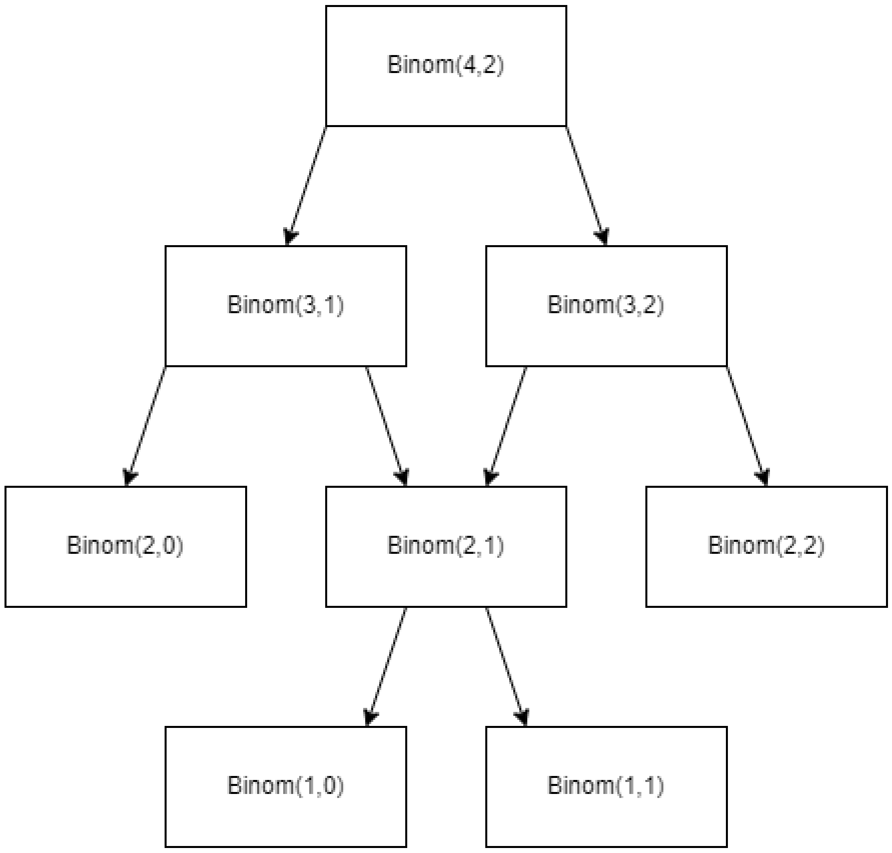


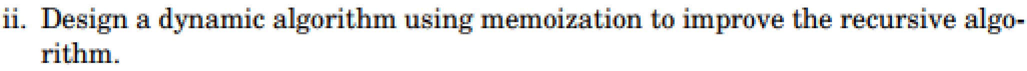
1. **Basics of dynamic programming.**

   (a) (3p.) Consider the recursive algorithm that was found last week for computing binomial coefficients.

   i. Draw the recursion tree for Binom(4, 2). Draw a DAG based on the recursion tree. Each function call with specific parameters can only appear once in the DAG.

The recursive formula for binomial coefficients is:
$Binom(n,k) = Binom(n-1,k-1) + Binom(n-1,k)$, where the base case is when $n = k$ or $k = 0$.
The recursion tree for Binom(4,2) is:





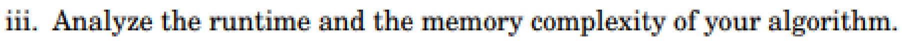
ii. Design a dynamic algorithm using memoization to improve the recursive algorithm.

```
memoizedTable = {} # This is a Python dictionary

def binomial_coefficient(n, k):
    assert n >= k
    if k == 0 or k == n:
        return 1
    key = (n, k)
    if key not in memoizedTable:
        memoizedTable[(n,k)] = binomial_coefficient(n-1, k) +
binomial_coefficient(n-1, k-1)
        return memoizedTable[(n,k)]
    else:
        return memoizedTable[(n,k)]

print(binomial_coefficient(4,2))
```

The memoized table saves the precomputed values. This table is in a form of Pascal’s triangle



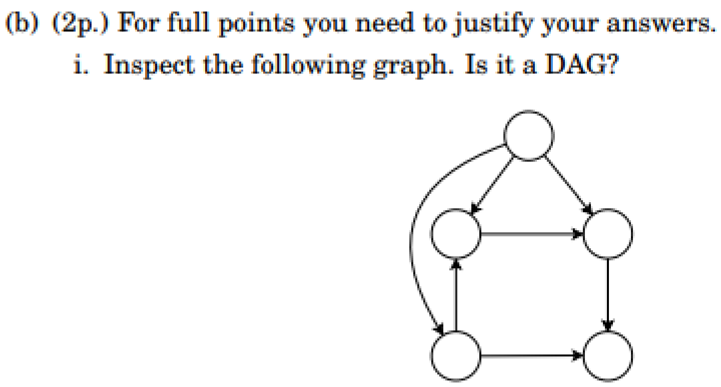
iii. Analyze the runtime and the memory complexity of your algorithm.

- Time Complexity of my algorithm:

Given that all precomputed (n,k) values are saved inside the dictionary, we can be sure that it will run rerun those values again if they are encountered again. In other words, at most configuration is calculated and we know that at the base case, it runs in $O(1)$. Therefore, this algorithm runs in $O(nk)$

- Memory complexity of my algorithm

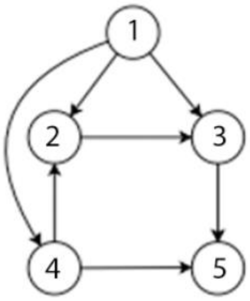
Because the dictionary only saves keys (n, k) such that $k \leq n$ and n runs from 1 to n and k runs from 1 to k, there are n rows and (n-k) columns. Therefore, the memory complexity is $O(n(n-k))$



(b) (2p.) For full points you need to justify your answers.

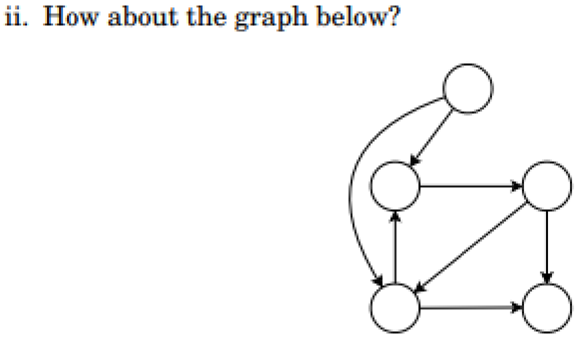
i. Inspect the following graph. Is it a DAG?

This is a DAG because there is no cycles in this graph.



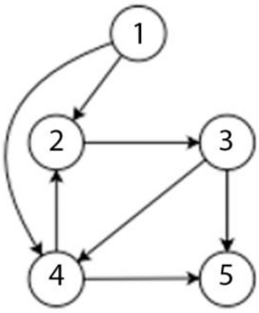
First, node 1 has only outcoming edges and node 5 has only incoming edges, so we can safely omit them when we search for cycles in this graph, because cycles cannot go through nodes that solely have outcoming or incoming edges.

=> There are only nodes 2,3,4 left. We can see that node 4 goes to node 2, node 2 goes to node 3. But node 3 doesn’t go back to node 4. Therefore, there isn’t a cycle in the remaining nodes

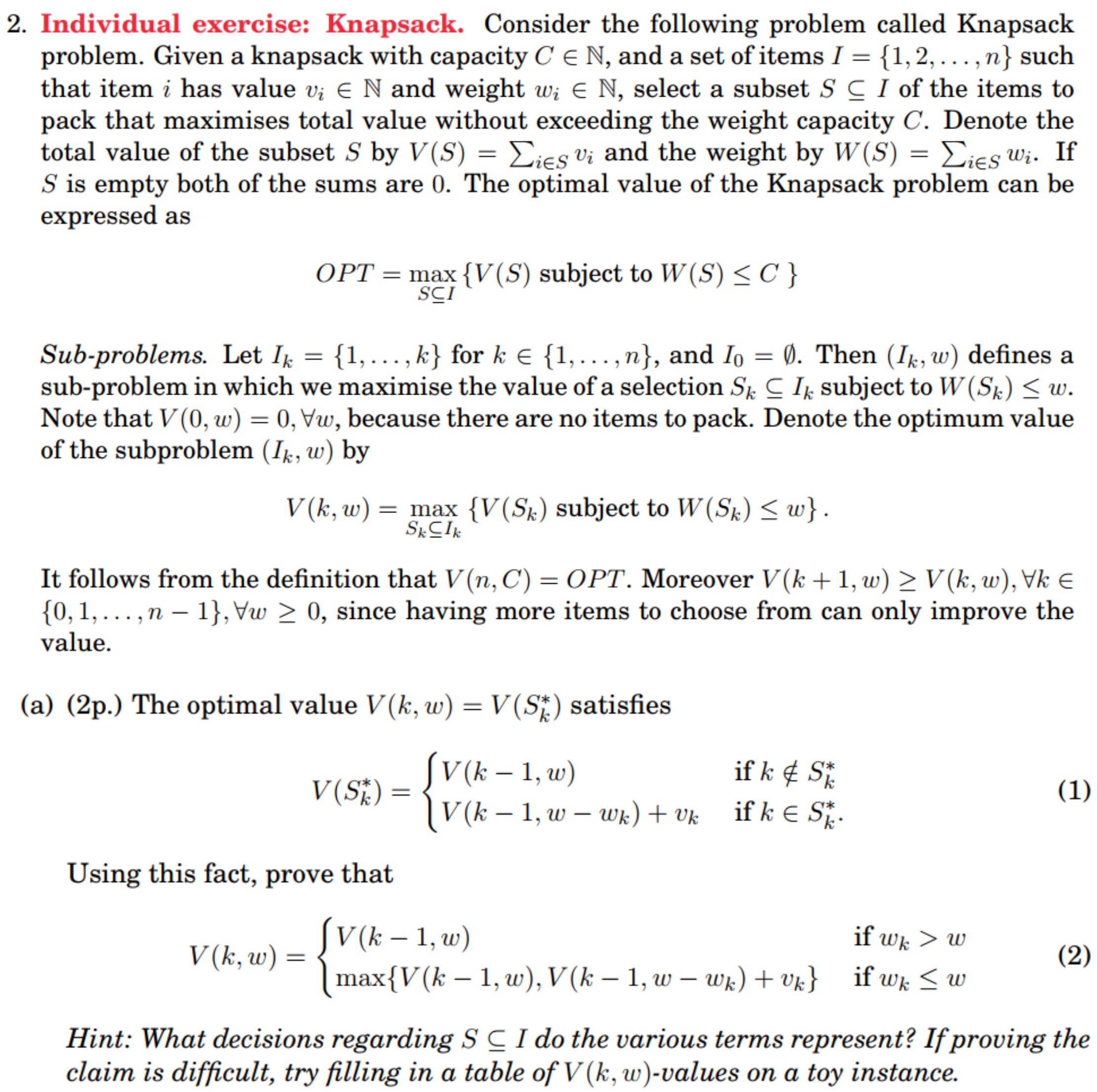


ii. How about the graph below?

This is not a DAG because there is a cycle in this graph



Like part i. above, we can omit nodes 1 and 5 because they only have either incoming or outcoming edges. There is an edge going from node 2 to node 3, node 3 to node 4, and node 4 to node 2. Therefore, it exists a cycle that goes around nodes 2-3-4 in this graph



2. **Individual exercise: Knapsack.** Consider the following problem called Knapsack problem. Given a knapsack with capacity $C \in \mathbb{N}$, and a set of items $I = \{1, 2, \ldots, n\}$ such that item i has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$, select a subset $S \subseteq I$ of the items to pack that maximises total value without exceeding the weight capacity C. Denote the total value of the subset S by $V(S) = \sum_{i \in S} v_i$ and the weight by $W(S) = \sum_{i \in S} w_i$. If S is empty both of the sums are 0. The optimal value of the Knapsack problem can be expressed as

$$OPT = \max_{S \subseteq I} \{V(S) \text{ subject to } W(S) \leq C\}$$

Sub-problems. Let $I_k = \{1, \ldots, k\}$ for $k \in \{1, \ldots, n\}$, and $I_0 = \emptyset$. Then (I_k, w) defines a sub-problem in which we maximise the value of a selection $S_k \subseteq I_k$ subject to $W(S_k) \leq w$. Note that $V(0, w) = 0, \forall w$, because there are no items to pack. Denote the optimum value of the subproblem (I_k, w) by

$$V(k, w) = \max_{S_k \subseteq I_k} \{V(S_k) \text{ subject to } W(S_k) \leq w\}.$$

It follows from the definition that $V(n, C) = OPT$. Moreover $V(k+1, w) \geq V(k, w), \forall k \in \{0, 1, \ldots, n-1\}, \forall w \geq 0$, since having more items to choose from can only improve the value.

(a) (2p.) The optimal value $V(k, w) = V(S_k^*)$ satisfies

$$V(S_k^*) = \begin{cases} V(k-1, w) & \text{if } k \notin S_k^* \\ V(k-1, w-w_k) + v_k & \text{if } k \in S_k^*. \end{cases} \tag{1}$$

Using this fact, prove that

$$V(k, w) = \begin{cases} V(k-1, w) & \text{if } w_k > w \\ \max\{V(k-1, w), V(k-1, w-w_k) + v_k\} & \text{if } w_k \leq w \end{cases} \tag{2}$$

Hint: What decisions regarding $S \subseteq I$ do the various terms represent? If proving the claim is difficult, try filling in a table of $V(k, w)$-values on a toy instance.

First case:

Interpretation of $V\left(S_k^*\right) = V\left(k-1, w\right)$ if $k \notin S_k^*$: if item k is not included in the knapsack, then the optimal subproblem value is the same as the one when we do not consider the item k at all with the same weight.

Proof by contradition for this case: Suppose that $V\left(S_k^*\right) \neq V\left(k-1, w\right)$ if $k \notin S_k^*$ is correct. Then there must be another selection such that $V'\left(k, w\right) > V\left(k-1, w\right)$. From the subproblem value $V\left(k-1, w\right)$, we know that the optimal selection does not have the item k in the next iteration, so $V\left(k-1, w\right)$ does not increase. This is a contradiction and there does not exist any $V'\left(k, w\right)$ such that $V'\left(k, w\right) > V\left(k-1, w\right)$ if $k \notin S_k^*$ => $V\left(S_k^*\right) = V\left(k-1, w\right)$ if $k \notin S_k^*$ is always correct.

Second case:

Interpretation of $V\left(S_k^*\right) = V\left(k-1, w-w_k\right) + v_k$ if $k \in S_k^*$: if item k is included in the knapsack, then the optimal subproblem value is the one when we do not consider the item k with the new weight subtracted by the weight of item k from the current weight, plus the value of the item k.

Proof by contradition for this case: Suppose that $V\left(S_k^*\right) \neq V\left(k-1, w-w_k\right) + v_k$ if $k \in S_k^*$ is correct, then there must be another selection S_{k-1}^{*} with a higher value and it means that $V'(k-1, w-w_k) > V(k-1, w)$. When we add the kth item to S_k^*, we have $V'(k-1, w-w_k) + \mathrm{v}_k > V(k, w)$. Now we have a subset of the first k items with higher value than $V(S_k^*)$, which is the optimal value by definition. This is a contradiction

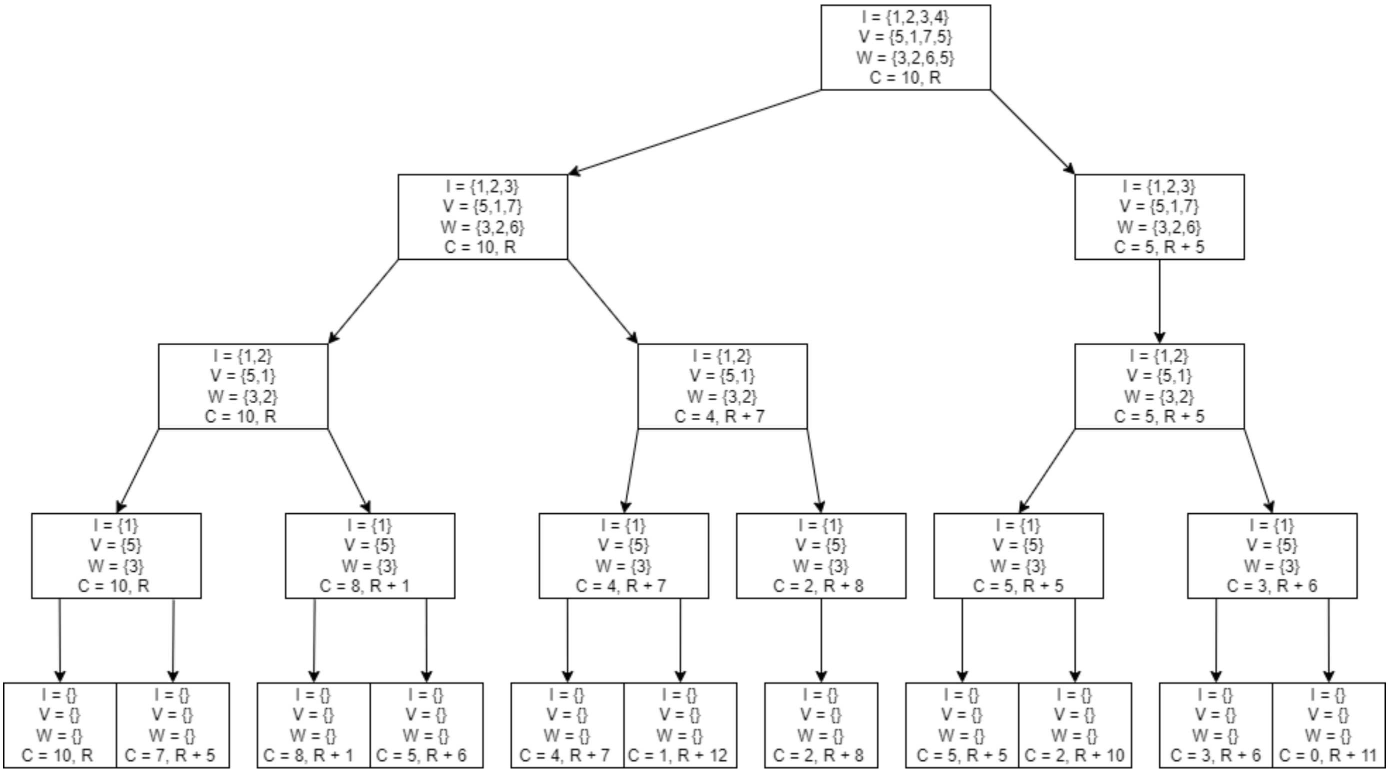
=> $V(S_k^*) \neq V(k-1, w-w_k) + v_k$ if $k \in S_k^*$ is always correct.

(b) (1p.) Consider the following pseudocode for the recursive algorithm solving the problem. Draw the recursion tree for the problem for the following items and capacity 10.

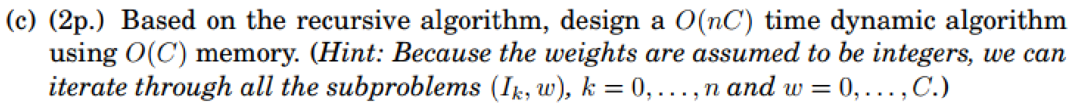
item	value	weight
1	5	3
2	1	2
3	7	6
4	5	5

```
Algorithm 1: Knapsack($I, (v_k, w_k)_{k \in I}, C$)
$n \leftarrow |I|$
if $I = \emptyset$ or $C = 0$ then
    return 0
end
else
    if $w_n > C$ then
        /* Remove the last item, it can't fit.                          */
        return Knapsack($I \setminus \{n\}, C$)
    else
        return $\max\{$Knapsack($I \setminus \{n\}, C$), Knapsack($I \setminus \{n\}, C - w_n$) $+ v_n\}$
    end
end
return $V[n, C]$
```

The recursion tree for the given items and capacity is as follows:


I = {1,2,3,4} V = {5,1,7,5} W = {3,2,6,5} C = 10, R
I = {1,2,3} V = {5,1,7} W = {3,2,6} C = 10, R
I = {1,2,3} V = {5,1,7} W = {3,2,6} C = 5, R + 5
I = {1,2} V = {5,1} W = {3,2} C = 10, R
I = {1,2} V = {5,1} W = {3,2} C = 4, R + 7
I = {1,2} V = {5,1} W = {3,2} C = 5, R + 5
I = {1} V = {5} W = {3} C = 10, R
I = {1} V = {5} W = {3} C = 8, R + 1
I = {1} V = {5} W = {3} C = 4, R + 7
I = {1} V = {5} W = {3} C = 2, R + 8
I = {1} V = {5} W = {3} C = 5, R + 5
I = {1} V = {5} W = {3} C = 3, R + 6
I = {} V = {} W = {} C = 10, R
I = {} V = {} W = {} C = 7, R + 5
I = {} V = {} W = {} C = 8, R + 1
I = {} V = {} W = {} C = 5, R + 6
I = {} V = {} W = {} C = 4, R + 7
I = {} V = {} W = {} C = 1, R + 12
I = {} V = {} W = {} C = 2, R + 8
I = {} V = {} W = {} C = 5, R + 5
I = {} V = {} W = {} C = 2, R + 10
I = {} V = {} W = {} C = 3, R + 6
I = {} V = {} W = {} C = 0, R + 11

⇨ Solution: the optimal packings is items = {1, 3}, with weights = 9 and maximum value = 12



(c) (2p.) Based on the recursive algorithm, design a $O(nC)$ time dynamic algorithm using $O(C)$ memory. (*Hint: Because the weights are assumed to be integers, we can iterate through all the subproblems* (I_k, w), $k = 0, \dots, n$ *and* $w = 0, \dots, C$.)

In the “tradition” dynamic algorithm for 0/1 knapsack problem, a table of size n x C is required, where n is the number of items and C is the capacity. Upon closer analysis, the computation for the value of table[i][j] only depends on the solutions from the previous row. In other words, we can use only a table of one row with the size of (C + 1), where table[C] stores the final result. For each iteration in an n-for loop, we can overwrite previous values in the table, which is no longer needed for the current computation.

The dynamic algorithm in Python language is

```
# values[] stores the values for each item
# weights[] stores the weights for each item
# n is the number of items
# C is the maximum capacity of bag
# table[C+1] to store final result
def knapSack(values, weights, n, C):
    table = [0]*(C+1);
    for i in range(n):
        for j in range(C, weights[i] - 1,-1):
            table[j] = max(table[j] , values[i] + table[j - weights[i]]);
    return table[C];
```

Time Complexity: there are two for-loops, one runs for n times and another for C times. In total, the dynamic algorithm runs in O(nC) time

Memory complexity of the dynamic algorithm: we only need a table of size (C + 1) to store the current latest results => The auxiliary table is required in O(C + 1) or O(C)