Task B: Maximising Knapsack value

Algorithm 1 DynamicKnapsack($items$, $capacity$, num_items, $filename$)

Input:
$items$ = list of (name, weight, value) tuples
$capacity$ = positive integer representing the knapsack's total capacity
num_items = number of items being considered
$filename$ = name of the output CSV file (e.g. testing.csv)

Output:
$selected\_items$: list of item names included in the optimal solution
$selected\_weight$: total weight of selected items
max_value: total value of selected items

```
Initialize dp[0...num_items][0...capacity] to None
Set dp[0][0...capacity] ← 0
function SOLVEKNAPSACK($i, w$)
    if $i = 0$ or $w = 0$ then
        dp[i][w] ← 0
        return 0
    if dp[i][w] is not None then
        return dp[i][w]
    (name, weight, value) ← items[$i - 1$]
    if weight > w then
        dp[i][w] ← SOLVEKNAPSACK($i - 1, w$)
    else
        without_item ← SOLVEKNAPSACK($i - 1, w$)
        with_item ← value + SOLVEKNAPSACK($i - 1, w - weight$)
        dp[i][w] ← max(without_item, with_item)
    return dp[i][w]
end function
max_value ← SOLVEKNAPSACK(num_items, capacity)
selected_items ← $\emptyset$, selected_weight ← 0, w ← capacity
for $i$ ← num_items down to 1 do
    if dp[i][w] $\neq$ dp[i - 1][w] then
        Append items[i - 1].name to selected_items
        selected_weight ← selected_weight + items[i - 1].weight
        w ← w - items[i - 1].weight
SAVECSV(dp, items, capacity, filename)
return selected_items, selected_weight, max_value
```

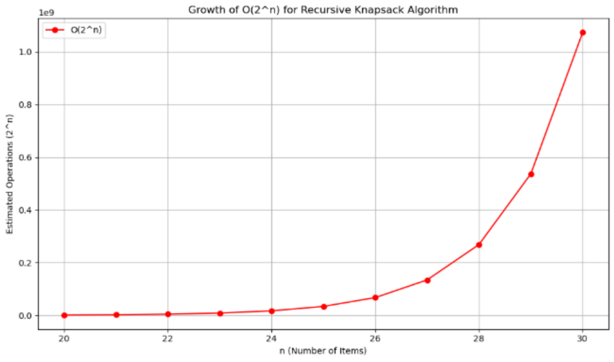
The benefit of dynamic programming is that there are no redundant calculations – as we use previously stored results in memory. The downside is that we sacrifice space for a slightly faster time complexity.

Task C: Analysis of the complete problem

Task A:

The pseudocode below:

```
    Include location, include weight, include value = RecursiveKnapsack(T, c – weight, k – 1)
    Exclude location, exclude weight, exclude value = RecursiveKnapsack(T, c, k -1)
```



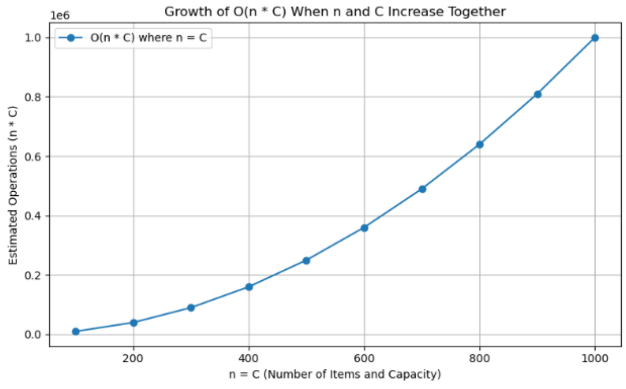
Calls the function *RecursiveKnapsack* from 2 different possibilities: including the item or excluding the item. The given time complexity for recursion is O(2^n), since we have n items and 2 choices. As seen with the graph attached, the rapid growth of this function makes it extremely inefficient for larger input sizes – specifically capacities of 25-30 and larger. The inefficiency comes from the redundant computations needed due to overlapping subproblems, which are addressed by the dynamic programming approach.

Task B:

In this approach, we use lazy/sparse programming to memorise previous results of subproblems, avoiding redundant calculations for the sacrifice of space.

We are solving each subproblem defined by:

- Item index i (given 0 to n items)
- Current weight w (0 to capacity C)



Hence, the number of subproblems is n * C, with each subproblem taking a constant time of O(1) to compute once memorization is used.

The Time complexity is O(n * C). The graph shown demonstrates the quadratic growth of the dynamic programming Knapsack algorithm, hinting that it performs poorly for larger outputs regardless of being extremely better than the recursive approach with a time complexity of O(2^n)

findItemsAndCalculatePath:

This function is responsible for finding the best items to collect and put in the knapsack using *knapsack.solveKnapsack(maze, csvFilename)* and finding the shortest path, starting from the entrance to the exit of the maze using *solver.solveMaze(maze, entrance, exit)*. Since these tasks are sequential, meaning that they run in order, we use the given formula $\text{Time}_{\text{Total}} = \text{Time}_A + \text{Time}_B$ since one runs after the other.

For Dynamic programming, where n = number of items in the maze, W is the knapsacks weight capacity, R x C is the size of the maze grid (rows x cols): O(n * W + R * C)

For Recursion, where n is the number of items given 2 options (include / exclude) and R x C is the size of the maze grid (rows x cols): O(2^n + R * C)

Empirical Design:

The values which are important in calculating the algorithmic complexity are:

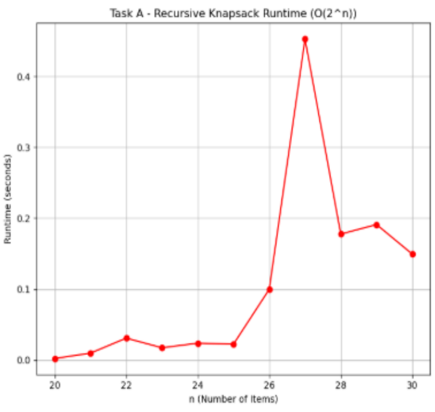
n: number of items (treasures) in the maze (denoted as n items)

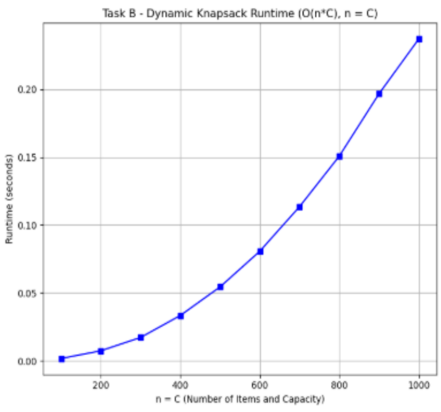
c: knapsack capacity (denoted by a positive integer k, where c is the capacity of the knapsack)

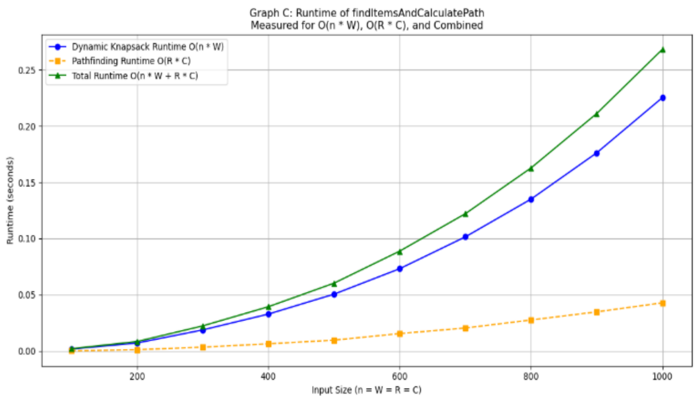
r x c (size of maze – Rows * Columns)

These values are important as they control the number of operations needed to run the program. n directly affects the number of subproblems in the knapsack, especially if dynamic programming is used. This also increases the runtime of the algorithm. Capacity c determines the combinations of item weights the algorithm needs to consider; thus a higher capacity increases the size of the dynamic programming table. The size of the maze determines the traversal for the chosen algorithm (in this case, BFS). Larger grids may mean more operations needed to explore or find a path.

Variables which we might ignore or aren't important (eg. They don't have an affect on time complexity) are:
CSV: an I/O (input/output) operation, normally constant time, depending on the size of the data saved.
Coordinates of entrance/exit: normally constant/doesn't change. Only affecting path length but not asymptotic complexity







Empirical Analysis:

The graphs A, B and C shown are the runtime for Tasks A, B and findItemsAndCalculatePath. Looking at the graphs above, we can conclude that Graph A, the exponential growth of the recursive algorithm from Task A becomes extremely unstable as n increases – this is shown after n >= 24. – indicating the inefficiency of this algorithm for larger inputs as they exhaust computational resources. Task B is slightly different – and shows a more manageable and predictable increase in runtime as both n items and knapsack capacity C increase. Though the growth is still polynomial, the growth is more suitable for larger inputs due to it being more scalable. Graph C includes 3 lines in the graph – separating the added time complexities.
When n = W (items = weight), the green curve with a time complexity of O(n * W) shows a steeper climb – this is noticeable towards the larger inputs which reflect a worse-case time complexity. The graph to the right compares the dynamic knapsack with findItems function. Analysing the yellow line, we see that the runtime depends on the size of the maze (R * C). the curve is flatter than the blue line which indicates that it scales more effectively as input increases. It also demonstrates better scalability as the it has a slower growth rate compared to the other 2 lines, confirming that its less demanding than the knapsack component

the results align with my theoretical complexity predictions, there are still certain discrepancies that may affect results; System specifications (such as CPU Speed, OS, and available RAM), input characteristics (such as number of items, knapsack capacity and the size of the maze) and python interpreter and IDE (background tasks, Python virtual environments, extensions, and packages installed). These might be responsible for the spike in runtime when n = 27, as shown in the graph above. The theoretical complexity still gives a strong estimate of performance trends. It provides a reliable foundation for understanding algorithmic behaviour, even if practical results can still be shaped by implementation and run-time conditions

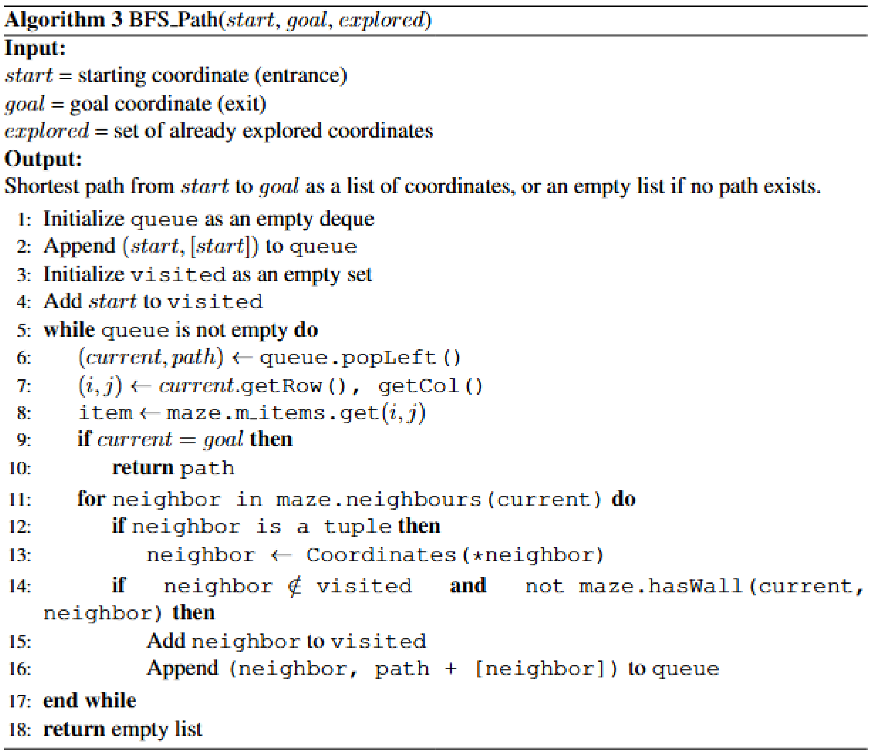
Task D: Maximising knapsack value

After doing some research, for this task I have decided to approach this problem using the greedy programming approach, with the help of BFS (Breadth First Search) to help search the maze (entrance -> exit) and search for treasures (search the neighbouring empty cells), using the knapsack logic to decide if it can be collected or not. An interesting twist to this task is that we have to maximize the value of the knapsack, while reducing the cost of the reward, calculated with:

Reward = knapsackValue – CellsExplored

After some research and consultation with my tutor, I decided to approach this solution in a specific way:

- Use BFS (Breadth First Search) to find the shortest path from the entrance to the exit
- Use BFS to search the nearest unexplored cell
- Use a greedy programming approach to pickup the treasure (regardless of its value)
- Use knapsack logic to determine if we can include the given treasure in the knapsack or if we have to exclude it.



Algorithm 3 BFS_Path($start$, $goal$, $explored$)

Input:
$start$ = starting coordinate (entrance)
$goal$ = goal coordinate (exit)
$explored$ = set of already explored coordinates
Output:
Shortest path from $start$ to $goal$ as a list of coordinates, or an empty list if no path exists.

```
Initialize queue as an empty deque
Append $(start, [start])$ to queue
Initialize visited as an empty set
Add $start$ to visited
while queue is not empty do
    $(current, path)$ ← queue.popLeft()
    $(i, j)$ ← $current$.getRow(), getCol()
    item ← maze.m_items.get$(i, j)$
    if $current = goal$ then
        return path
    for neighbor in maze.neighbours(current) do
        if neighbor is a tuple then
            neighbor ← Coordinates(*neighbor)
        if neighbor ∉ visited and not maze.hasWall(current, neighbor) then
            Add neighbor to visited
            Append (neighbor, path + [neighbor]) to queue
end while
return empty list
```

The pseudocode to the left explains one of the BFS loops – it uses a queue for the FIFO (First in, First Out) priority order, since we are going to be moving through cells. If then declares the entrance as visited, and puts it into the queue. From there, the while loop moves in the direction specified by the BFS algorithm, finding the shortest path from the entrance to the exit. Each cell it walks through to the exit, it then adds to the queue and adds it to the set 'visited', which keeps track of visited cells. It converts this from a tuple to the Coordinate object. If all of this is false (eg. Cell is visited or path crosses through a wall) it returns false and prints an empty list.

The other BFS algorithm works in a similar way – but instead of searching for the shortest path from entrance to exit, it searches for the nearest unexplored neighbouring cell.

This approach ensures balance between speed and efficiency – having a quick runtime while ensuring that the knapsack reward is maximised and the path is minimised. (from entrance to exit). The time complexity of the BFS algorithm is O(V + E), where V = number of vertices (cells in the maze) and E = number of edges (connections between cells which are not walls)

Algorithm 4 CollectTreasures($maze$, $current$, $exit$, $knapsack\_weight$, $knapsack\_value$, $collected$)

Input:
$maze$ = the maze structure with items
$current$ = current cell coordinate
$exit$ = goal cell coordinate
$knapsack\_weight$ = current total weight of the knapsack
$knapsack\_value$ = current total value of the knapsack
$collected$ = set of cells where items have already been collected
Output:
Updates internal state: optimal knapsack cells, weight, value, and reward.

```
 1: while $current \neq exit$ do
 2:     Print current position and knapsack stats (for debugging)
 3:     $(i, j) \leftarrow current$.getRow(), getCol()
 4:     if $maze$ has attribute m_items and $(i, j) \in maze$.m_items and $current \notin collected$ then
 5:         $(item\_weight, item\_value) \leftarrow maze$.m_items[$(i, j)$]
 6:         if $knapsack\_weight + item\_weight \leq$ knapsack capacity then
 7:             $knapsack\_weight \leftarrow knapsack\_weight + item\_weight$
 8:             $knapsack\_value \leftarrow knapsack\_value + item\_value$
 9:             Add $current$ to collected
10:             Update self.m_knapsack.optimalValue with $knapsack\_value$
11:             Update self.m_knapsack.optimalWeight with $knapsack\_weight$
12:             Append $(i, j)$ to self.m_knapsack.optimalCells
13:             Update self.m_reward using reward() method
14: end while
```

The pseudocode shown here goes through the logic of collecting the treasures, calculating the reward, and keeping track of the knapsack value and weight, as well as the location of the treasures.

The while loop executes if the current cell is not the designated exit – it then gets the row and column coordinates (I, j) then checks if the items in the knapsack are not in the collected set (a set where collected items are put and updated) if not, it checks if the capacity > weight. If less, it adds it to the knapsack and updates the weight, value and calculates the reward. If capacity > weight, it leaves it behind and doesn't pick it up.

While the algorithm discussed is great for uniform distribution – meaning that all cells have an equal change of spawning a treasure, we also have to consider different probabilities, such as linear-based distance skew or clustered zones. To adapt to this, I would modify the algorithm so it uses a probability map, where each cell has a probability score for generating a treasure. We can do this by using a heuristic search algorithm, such as A* which scores neighbouring cells by treasure probability. By doing this, we would prioritize the cells with a greater chance of containing treasures, filling the knapsack capacity to its limit. For models such as the linear-based distance skew or clustered zones, we would use a similar heuristics search as it favours cells further away from the entrance – meaning it will prioritize exploring cells where treasures are more likely. For clustered zones, I would incorporate a search which boosts priority of neighbours near known treasure clusters using the A* heuristic search algorithm.