Kyle Hippe 300518740

COMP 361 Design and Analysis of Algorithms

Andrew Lensen

19 August 2019

0-1 Dynamic Programming

Pseudocode:

This problem requires the use of a memoized table very similar to the Global Sequence Alignment problem presented last assignment.

This means that there will be two loops, the outer loop iterating through all the potential items, and the inner loop iterating through possible weights 1 through the maximum capacity of the knapsack. To determine the appropriate value at each index, we must calculate the maximum value of either the value of the cell directly above (meaning we would not include the item at the specific row) and the value of the cell at the row above and the current column minus the weight of the current item plus the value of the current item (meaning we would include the item). Once the table is established, we may assume the maximum value of the knapsack is the value at the cell in the last row and last column. To backtrack we initialize a list representing the items to include and initialize row counters and column counters. We assign the rows and column values to the start at the cell of highest value (last row, last column) and begin by evaluating the cell above. If the two numbers are equal, then we do not use the item at that index, insert a 0 at the head of the list, and we update the row to be row-1 and leave the columns unchanged. If the values are different, then we use that item, insert a 1 at the beginning of the list, and update our rows to be row-1 and our column to be column – weight of the item. Once 'rows' is equal to zero, the back trace is complete and we have a list of 1's and 0's that correlate to the possible items and indicate whether or not we place them in our knapsack. The pseudocode for these operations is listed below:

```
//items is a list of items(has both weight and value)
//capacity is an integer representation of weight capacity of the knapsack
knapsack(items, capacity)
    result = [number of items][capacity]
    initialize all values in results table to 0

    for rows 1 to number of items
        for columns 0 to capacity
            if the weight current item > current column
                propogate value from above row
                result[rows][columns] = result [rows -1][columns]
            else
                determine if its best to add item or not
                result[rows][columns] = max(result[row-1][columns],
                result[row-1][column - weight of item] + value of item)
```

```
//table is a two dimensional list and is the results of the knapsack method
backtrace(table)
    result = [length of items]

    currentRow = length of items - 1
    currentColumn = capacity - 1
    while currentRow is not 0
        if table[currentRow][currentColumn] == table[currentRow-1][currentColumn]
            insert 0 in first position of result
            currentRow -= 1
        else
            insert 1 in the first position of result
            currentRow -= 1
            currentColumn = currentColumn - weight of the item added
```

Correctness:

Each time execution reaches the if: else: statement, the table[row][column] represent the maximum score for a knapsack with 'column' weight and 'row' items (taken in order from the items list). From here we have three options:

> (1) The weight of the item is too great to put in the knapsack and the current value is taken from the row above
>
> (2)The weight of the item is less than that allowed in the knapsack but its value is not large enough to be put in the knapsack (it is better to take the value from the row above) so the current value is taken from the row above
>
> (3) the weight of the item is less than that allowed in the knapsack and its value is great enough to warrant adding it to the knapsack with the value coming from the previous row and the column minus the item weight, plus the item value (table[row-1][column-weight]+value)

Initialization: For the first iteration, we are at row 1 and column 0, this means that the knapsack has no capacity and we are looking at the first item. This falls into case 1, the item is too great to put in the knapsack and the value is propagated from the row above it (a value of 0 since that is what the table values are initialized to)

Maintenance: For any iteration where the current counters are row = n and column = m, assume that for all 1...n and 0...m the table is populated with correct values. At the current n and m values we have three choices.

> (1) If the weight of the item is greater than the current column (m) then the value is propagated from the row above, and since the table up to n and up to m is correctly populated, this is the current maximal value of a knapsack with m capacity and with n items available.

(2) In this case, the knapsack has space for the item, but adding the item will not result in the maximal value, so the value is propagated from the row directly above. This results in the maximal value at this current row and column since the previous indexes of the table were filled in correctly and it is not more valuable to insert the item into the knapsack (the other choice)

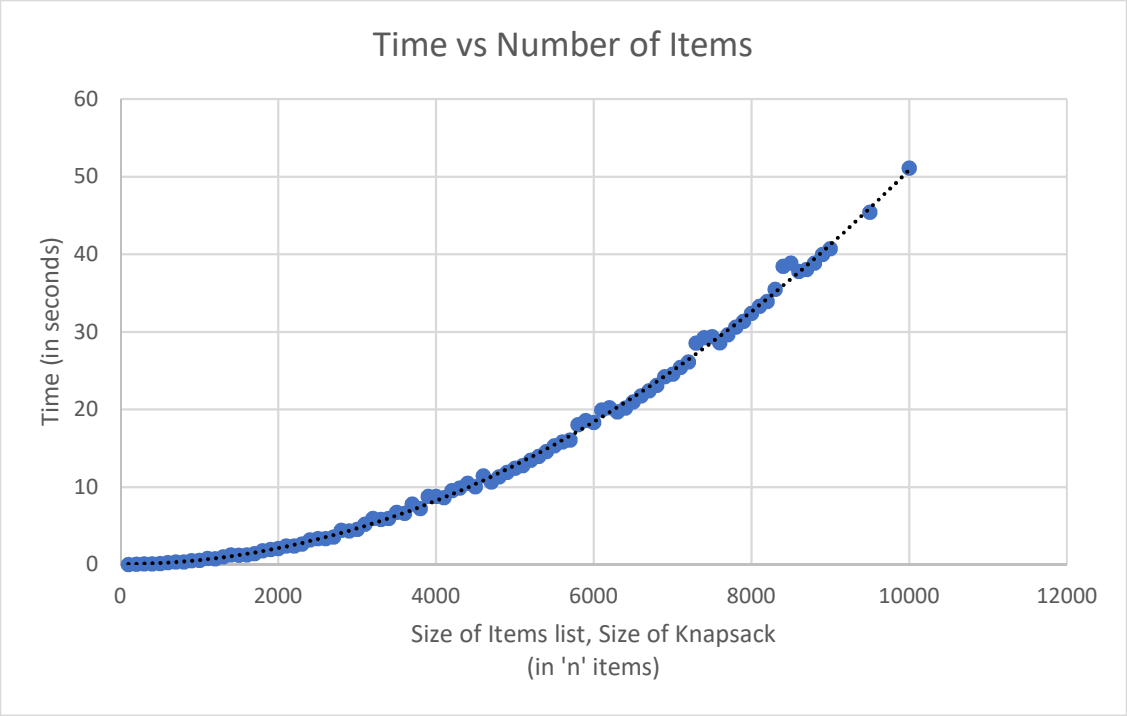
(3) In this case, the knapsack has space for the item and adding the item will result in the maximal value. The value is equal to table[row-1][column-weight of item] + the value of the item. And since all previous indexes of the table are assumed optimal for that weight and item combination, this will result in the maximal value for the current weight and item combination.

Termination: Both loops will terminate once rows is equal to the number of items and the columns is equal to the total weight capacity of the knapsack. Once this happens the value at table[length of items list][total weight capacity] will represent the maximal value of the current weight and item list combination since it holds for the first iteration, and all subsequent iterations after until the termination (rows = length of items list, columns = total weight capacity) state.

Testing:

The theoretical complexity of this algorithm (in the worst case) is $O(n^2)$ where n represents the total number of items, as the majority of the complexity arises from the generation of the table, and given a case were the length of the items list (the rows of the table) are equal to the weigh capacity of the knapsack (the columns of the table) this indicates a time complexity of $O(n^2)$. Since accessing the items list, calculating the values at the table and other actions that go along with generation of the table are constant time this does not affect the time complexity.
The backtracking of this algorithm is O(n) since there can be at most n items used in the knapsack and if every item were used in the knapsack the backtracking would have to visit every square diagonal (row-1, column-1) in the table. But since this is of lesser degree than $O(n^2)$, it does not play into the big –O calculation.

For testing, I generated random items lists where the number of items in the items list were equal to the capacity (the worst possible case) and ran tests ranging from a size of 100 (number of items and capacity of knapsack) to 10,000 items and capacity. Each test was run 5 times to generate an average trial time. This time is graphed vs the size of the items list/capacity below:



The trendline represents a n^2 line of best fit and fits the data very closely.

Since the trendline and the data are so close to each other, we can infer that the implementation of this algorithm is $O(n^2)$ for randomly generated data.

0-N Brute Force

Pseudocode:

Even though the problem has changed, and this is now a 0-N Knapsack problem instead of a 0-1, the approach of a brute force algorithm is much simpler than a dynamic approach. Instead of using a table and memorizing results, we simply enumerate all possible knapsacks and choose the knapsack with the highest value that still obeys the weight limit.

To do this, we recursively generate a list of every possible item combination and then iterate over the list to find the best combination.

To generate every possible combination, we must first check for our base case, which is when the items list has no more items in it. At this point we will return a list of an empty list. If we are not at the base case, there are still items left in our items list, we will create a new list of all the combinations and then for each item, recursively call the enumeration method with a headless list of items (a list of items with the head/first index removed). Inside this for loop, we will append the current list and the current list plus the first item in the items list to the all combinations list. Then once all items are used (the items list is empty), we return the all combinations list. The pseudocode is shown below:

```
//Brute force algorithm
brute(items)
    //base case
    if the length of items is 0
        //return an empty list of lists
        return [[]]

    allCombinations = []
    for combinations in brute(items[1:]
        append combinations and [combinations +[items[0]]] to allCombinations

    return allCombinations
```

Backtracking is trivial in this program as all that is needed is one traversal of the list and store a reference to the maximal set.

Correctness:

Correctness for this algorithm does not need to be as strict since we are not choosing any optimal solution/sub-solution until we have all possible solutions and there really are no sub-solution logic to worry about.

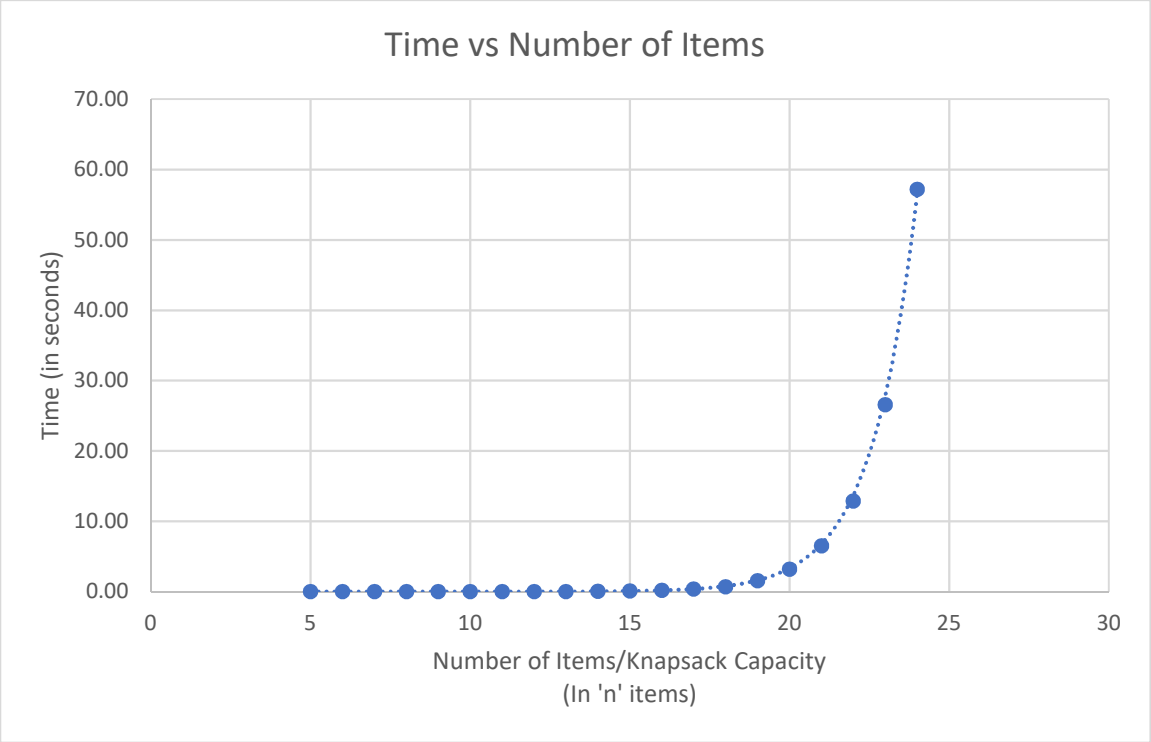
In order to prove correctness, we need to verify that the function provided in pseudocode above will produce a list containing every possible combination of items. If the items list has a length of 0 (our base case) the function will return a list of an empty list, which is every possible combination of 0 items. If our items list has a length of 1, then we create a blank list and for each item in the list, recursively generate a list of that item and that item + every other item/combination of items in the items list (in this case no other items) leaving us with a list containing an empty list and a list containing the only item in the items list. If our input items list has a length of 2, then things get a bit trickier. We enter the recursion once, and still have items left in the list, so we create a list of the first item, then we create a list of the first and second items, and then we create a list of just the second items. At this point the items list is now empty and every possible combination has been explored. This sequence of events happens for all sizes larger than 1, and since we are always recursing on an items list with 1 less element, we are guaranteed to hit termination since the function will not run on an empty list.

Testing:

The Brute Force algorithm is, as the name suggests, very inefficient. With the recursive generation of all possible combinations we get an efficiency of $O(2^n)$ where n is the number of items in the list. Given constant access time into the list as well as O(n) for the traversal of each combination to generate the weight and value of each combination, there is no significance of these in relationship to $O(2^n)$ so the efficiency of this algorithm is $O(2^n)$.

Testing this program was a bit tricky since the algorithm was implemented in python and since I did not use any specific memory management technique, the program ran out of available

memory at items list lengths around 25. Due to this, only items list ranging in size from 5 to 25 were included. Each trial was run 5 times to generate an average time which was used for graphing purposes. The graph is shown below:



The trendline on this graph is a 2^n trendline that is fit to the data, and due to how closely it fits, we can infer that this algorithm runs in line with a $O(2^n)$ time complexity.

0-N Dynamic Programming

Pseudocode:

The 0-N Dynamic programming problem is nearly identical to the 0-1 version of the problem so much of the pseudocode remains unchanged. However, I will go through the pseudocode again below.

The largest difference is the generation of inputs. In the 0-1 version of the problem, the algorithm accepted inputs as is, that is as a list of objects with weight and value attributes. This program however accepts a list of objects with attributes weight, value, and quantity. In order to make the 0-N version work a method had to be added to expand the list of items to accommodate for items that had a quantity larger than 1. To do this, a new items list was instantiated with no items in it. Then, we pass over each element in the old items list and as we go if there is an item with a quantity greater than 1, we append n copies of the item with quantity changed to 1 to the list. If a specific item has a quantity of 1, then we add this item to the list. This creates a new list of items with all quantity of 1, but duplicate entries of some items to represent multiple of the same item, rather than increasing the quantity. From there nothing is changed and memoization begins. This means that there will be two loops, the outer loop iterating through all the potential items, and the inner loop iterating through possible weights 1 through the maximum capacity of the

knapsack. To determine the appropriate value at each index, we must calculate the maximum value of either the value of the cell directly above (meaning we would not include the item at the specific row) and the value of the cell at the row above and the current column minus the weight of the current item plus the value of the current item (meaning we would include the item). Once the table is established, we may assume the maximum value of the knapsack is the value at the cell in the last row and last column. To backtrack we initialize a list representing the items to include and initialize row counters and column counters. We assign the rows and column values to the start at the cell of highest value (last row, last column) and begin by evaluating the cell above. If the two numbers are equal, then we do not use the item at that index, insert a 0 at the head of the list, and we update the row to be row-1 and leave the columns unchanged. If the values are different, then we use that item, insert a 1 at the beginning of the list, and update our rows to be row-1 and our column to be column – weight of the item. Once rows is equal to zero, the back trace is complete and we have a list of 1's and 0's that correlate to the possible items and indicate whether or not we place them in our knapsack. The pseudocode for these operations is listed below:

```
//Changes quantity of all items to 1 and expands the items list
//This is done so that our code can evaluate how many of each item to bring
//items is a list of items, has weight, value, and quantity attributes
expandItems(items)
    newItems = []
    for i 0 to number or items
        if items[i] has a quantity greater than 1
            for number of instances of each item
                append a new item with identical weight, value but quantity of 1
to newItems
        else
            append a new item with identical properties to newItems


//items is a list of items(has weight, value, quantity)
//capacity is an integer representation of weight capacity of the knapsack
knapsack(items, capacity)
    result = [number of items][capacity]
    initialize all values in results table to 0

    for rows 1 to number of items
        for columns 0 to capacity
            if the weight current item > current column
                propogate value from above row
                result[rows][columns] = result [rows -1][columns]
            else
                determine if its best to add item or not
                result[rows][columns] = max(result[row-1][columns],
                result[row-1][column - weight of item] + value of item)
```

```
//table is a two-dimensional list and is the results of the knapsack method
backtrace(table)
    result = [length of items]

    currentRow = length of items - 1
    currentColumn = capacity - 1
    while currentRow is not 0
        if table[currentRow][currentColumn] == table[currentRow-1][currentColumn]
            insert 0 in first position of result
            currentRow -= 1
        else
            insert 1 in the first position of result
            currentRow -= 1
            currentColumn = currentColumn - weight of the item added
```

Correctness:

The correctness of this algorithm does not differ from the correctness of the 0-1 dynamic solution as the only salient difference between the two is how the input is generated.

The input generated from the algorithm described in pseudocode above will always produce the correct amount of data as it is merely expanding the list of items based on the quantity of each item. The items list that is provided on that function is based off of the total number of items, not just the total number of unique items so there will always be ‘n’ items provided to the memoization function. From there memoization occurs and follows the correctness below:

Each time execution reaches the if: else: statement, the table[row][column] represent the maximum score for a knapsack with ‘column’ weight and ‘row’ items (taken in order from the items list). From here we have three options:

> (1) The weight of the item is too great to put in the knapsack and the current value is taken from the row above
>
> (2)The weight of the item is less than that allowed in the knapsack but its value is not large enough to be put in the knapsack (it is better to take the value from the row above) so the current value is taken from the row above
>
> (3) the weight of the item is less than that allowed in the knapsack and its value is great enough to warrant adding it to the knapsack with the value coming from the previous row and the column minus the item weight, plus the item value (table[row-1][column-weight]+value)

Initialization: For the first iteration, we are at row 1 and column 0, this means that the knapsack has no capacity and we are looking at the first item. This falls into case 1, the item is too great to put in the knapsack and the value is propagated from the row above it (a value of 0 since that is what the table values are initialized to)

Maintenance: For any iteration where the current counters are row = n and column = m, assume that for all 1...n and 0...m the table is populated with correct values. At the current n and m values we have three choices.

> (1) If the weight of the item is greater than the current column (m) then the value is propagated from the row above, and since the table up to n and up to m is correctly populated, this is the current maximal value of a knapsack with m capacity and with n items available.
>
> (2) In this case, the knapsack has space for the item, but adding the item will not result in the maximal value, so the value is propagated from the row directly above. This results in the maximal value at this current row and column since the previous indexes of the table were filled in correctly and it is not more valuable to insert the item into the knapsack (the other choice)
>
> (3) In this case, the knapsack has space for the item and adding the item will result in the maximal value. The value is equal to table[row-1] [column-weight of item] + the value of the item. And since all previous indexes of the table are assumed optimal for that weight and item combination, this will result in the maximal value for the current weight and item combination.

Termination: Both loops will terminate once rows is equal to the number of items and the columns is equal to the total weight capacity of the knapsack. Once this happens the value at table[length of items list][total weight capacity] will represent the maximal value of the current weight and item list combination since it holds for the first iteration, and all subsequent iterations after until the termination (rows = length of items list, columns = total weight capacity) state.

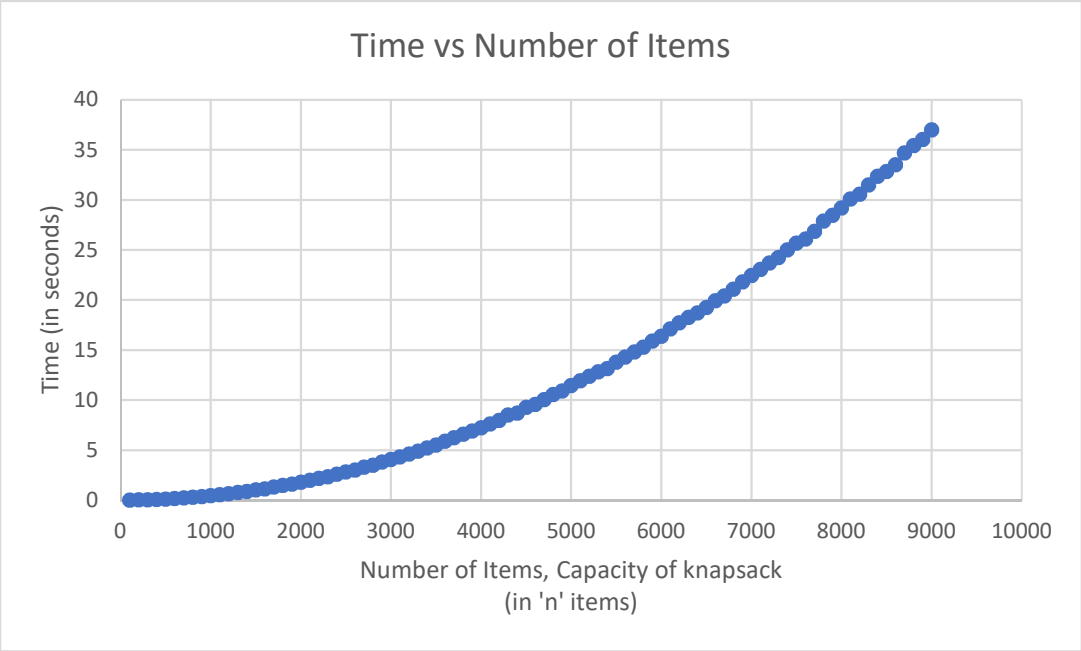
Testing:

Due to the similarity between the 0-1 Dynamic programming algorithm and the 0-N Dynamic programming algorithm, much of the information will remain the same between these two sections.

The only major difference between the 0-1 Dynamic algorithm and the 0-N Dynamic algorithm is the expansion of the items list in the latter. This results in an additional O(n) time complexity but given that the complexity is already $O(n^2)$ for generating the table, this is not significant. Once again, since accessing the items list, calculating the values at the table and other actions that go along with generation of the table are constant time this does not affect the time complexity.
The backtracking of this algorithm is O(n) since there can be at most n items used in the knapsack and if every item were used in the knapsack the backtracking would have to visit every square diagonal (row-1, column-1) in the table. But since this is of lesser degree than $O(n^2)$, it does not play into the big –O calculation.

Similarly to the 0-1 section, I generated random items lists where the number of items in the items list were equal to the capacity (the worst possible case) and ran tests ranging from a size of 100 (number of items and capacity of knapsack) to 9,000 items and capacity. Each test was run 5

times to generate an average trial time. This time is graphed vs the size of the items list/capacity below:



There is a n^2 fit line in the graph, but the data marks are covering the line. This indicates that the data very closely represents a $O(n^2)$ time complexity.

0-N Graph Search

Pseudocode:

The approach to solving the 0-N knapsack problem with a graph search algorithm is the most different part of this assignment. There is no table generation or memoization, and there is no generation of every possible combination of items. Instead, each point in the game, or 'state' is represented by a 'node' that contains a list indicating the amount of each item in the current state, the current value, and the current weight. Each node will have a list of children that represent an increment of every possible item (independently) without either going over the weight limit or having a value that is not optimized (more about this part later). This is a general overview, a more in-depth description of the pseudocode and the pseudocode itself follows:

Before starting there is a global variable that contains a hashed list of all the nodes that have been explored (initialized to empty) to allow for constant time checking if a node has already been explored. To begin, the search method has three parameters, a rootnode, the capacity of the knapsack, and the items list. First, we add the rootnode to the list of explored nodes and then call the generate children for the current node.

The generate children method takes three parameters, the current node, the items list, and the total capacity. This method will create a temporary node that is initialized with the values of the node before but with single item incremented by 1, and the corresponding value and weight increase. It will then check if the current weight is over the capacity, and if it is not over capacity

it will check if the potential node generated has already been visited, and if it has not been visited, then it will finally run a greedy knapsack problem with the current weight (since the greedy solution will almost always be sub-optimal), and if the current value is greater than that generated by the greedy algorithm then we will add the temporary node to the children's list of the root node. Once again, the node will only be added if the weight is less than or equal to capacity, the current solution is greater than the greedy solution, and the node has not already been visited.

The generate children method then returns to the caller (the search method) and the search method then iterates through the list of children of the current node, and if the value of that node is greater than the previously stored max value, then the pointers to the current max node are updated. Regardless of whether the current node is a maximal node, the search method is then recursively called on the current child and the process repeats until there are no more children to be explored.

This method is a pseudo queue and will explore the graph in a depth-first manner. The pseudo code is as follows below:

```
//Psuedo code for 0N Graph
*********************************************************************************
//rootnode is a node (list of included items, weight, value)
//capacity is an int
//items is a list of sorted items (based on value/weight)
search(rootNode, capacity, items)
    add current node to hash set of explored nodes

    //this method generates the children on the current node
    generateChildren(rootNode, items, capacity)

    for i <- 0 to length of children of current node
        if the current child value is greater than previous max value
            maxvalue = value of current child node
            maxnode = current child node

    search(rootNode.children[i], capacity, items)
*********************************************************************************
//this is where most of the complexity lies
def generateChildren(node, items, capacity)
    for i <- 0 to length of items
        if the current item has more of itself available
            if adding the item will not go over capacity
                if the potential node generated has not already been explored
                    if the value generated is larger than the greedy solution for
current subproblem
```

```
                        generate a child node and append it to rootnode's
children list

*************************************************************************
//items are already sorted based on value/weight ratio
greedyCheck(items, currentWeight)

    finalValue <- 0
    for i <- 0 to length of items
        for j <- 0 to quantity of current item
            if adding the item does not go over currentWeight
                increment finalValue by value of current item

    return finalValue
```

Correctness:

The correctness of this algorithm is straightforward as it follows a similar heuristic as the brute force solution, just with more checks along the way.

In order to prove correctness for this algorithm, we need to prove that it will initiate correctly, maintain correctness throughout execution, and then prove it provides correct results at termination.

For initiation, since the initial nodes items list is initialized to all 0's (indicating that at the current state, no items are included in the knapsack) and the weight and value of the node area also initialized to 0 this provides a correct initial state for the graph search algorithm.

Maintenance of the algorithm is straightforward as for each node we visit, we generate its acceptable children (described above in the pseudocode) and then search the children for a node (a state of the items) that is greater than our current maximal values. If a new max exists, the pointer for the max node is updated. Regardless of whether the max node is found or not the search algorithm is called on this child node and it is explored, and the cycle continues. Once the current branch terminates (the quantity of a specific item is exhausted) then the program bubbles back to the next item in the list and explores all acceptable states that follow. Each state must be a unique state since we check if the node has already been visited and do not add it to our children's list (the list that is explored) if it has already been explored. This ensures that there are no duplicate states and since we can only add a maximum of 1 item to each new state, this ensures that every possible state is explored.

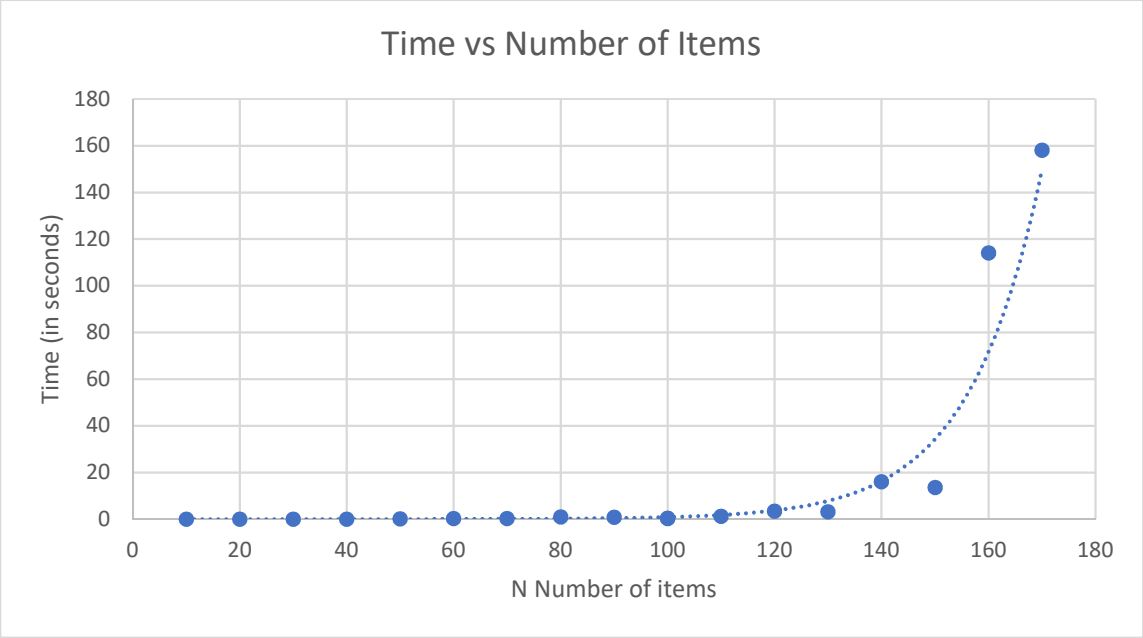
At termination, we know we have reached the maximal value of the specific instance of the problem because the program will terminate once all children nodes have no possible children and every node in the graph has been added to the explored set. And since we were keeping track of the maximal node while building/exploring the graph, we know we have the best state of the game, that is the state that maximizes the value of the knapsack while maintaining a weight less than or equal to the capacity of the knapsack.

Testing:

The time complexity for the 0-N graph search algorithm is rather tricky. Its asymptotic complexity is horrendous (discussed soon) but in practice given certain datasets, it is much more efficient than its asymptotic complexity and even in some cases more efficient than the dynamic programming solution (discussed in the analysis section).

The asymptotic time complexity of this algorithm is $O(2^n)$ because in the worst case, the algorithm will generate a new node (a state) for every possible combination of items in the same way that the brute force algorithm does. The additions to this implementation are sorting the items generated by their value/weight ratio, a O(n log n) efficiency due to python's implementation of sorting, O(n) traversal of each node's item list to calculate the weight and value of each node, O(n) greedy choice algorithm done during the generation of nodes, and constant access time into the items list. Since none of these are greater than $O(2^n)$ efficiency, they do not affect the big-O calculations leaving the algorithm with a final asymptotic complexity of $O(2^n)$.

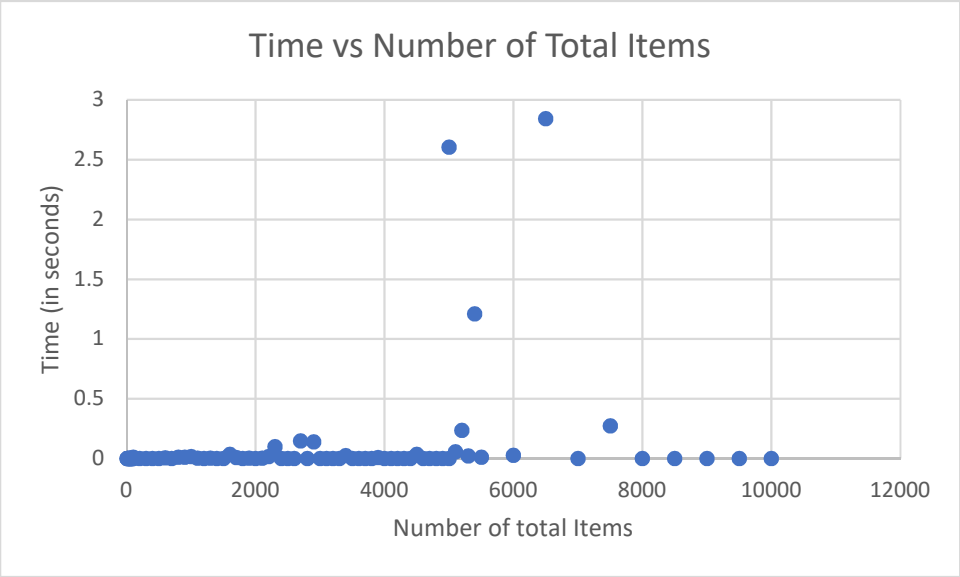
Testing this with completely random data as well as items list that had lengths equal to the capacity of the knapsack proved to not be this algorithm's strong suit as the ECS lab computers ran out of memory around 170 items. Each trial was run 5 times to calculate an average time, and this was necessary here more than any other area because there were extremely high amounts of variability in the execution time of each trail. The graph showing this is shown below.



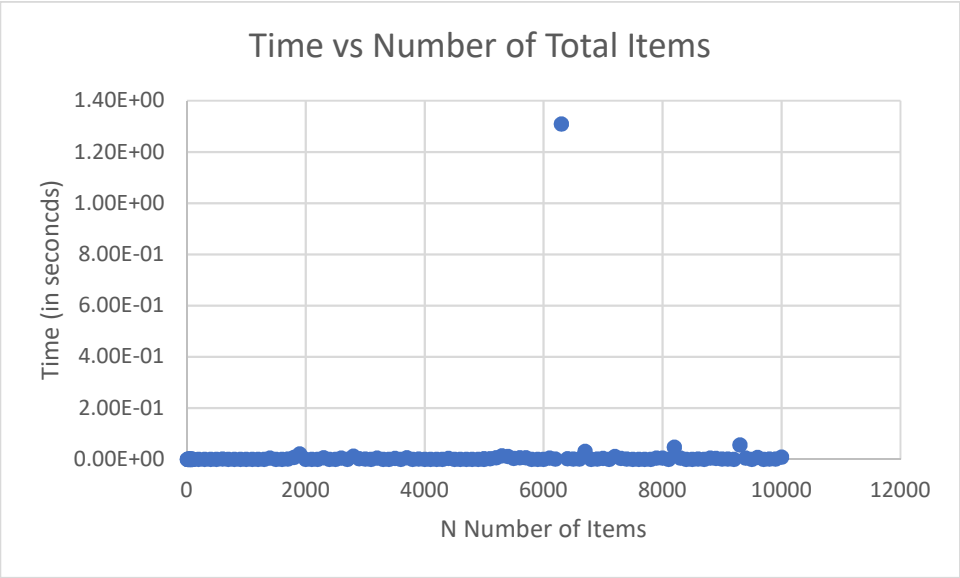
The trendline is roughly 2^n however the line does not completely fit, and there is not quite enough data to thoroughly show this trend.

However, this is not the end of the story, when creating items list and test cases, we can cater to the strengths of the graph search algorithm and create much faster executions by changing a couple values. The first value we can change is the number of unique items. If we limit the

number of unique items for each trial (meaning there are lots of the same object, but very few unique objects) we can reduce the execution time immensely as shown below.



This brought down the maximal execution time to just under three seconds and allowed there to be 10,000 total items. This is a astronomical increase in efficiency and it all came about due to the selection of inputs to something that optimizes the graph searches strength. The other sanitization of input that led to a drastic increase in efficiency was if we limited the capacity of the knapsack to ten percent of the total number of items. This allowed us to prune the number of children each node has very efficiently. The results are shown below and echo what is shown above.



The efficiency here was nearly linear, and this is all due to selecting certain inputs which cater to this algorithm's strengths (not having large amounts of children nodes, and therefore having less recursive calls in total).

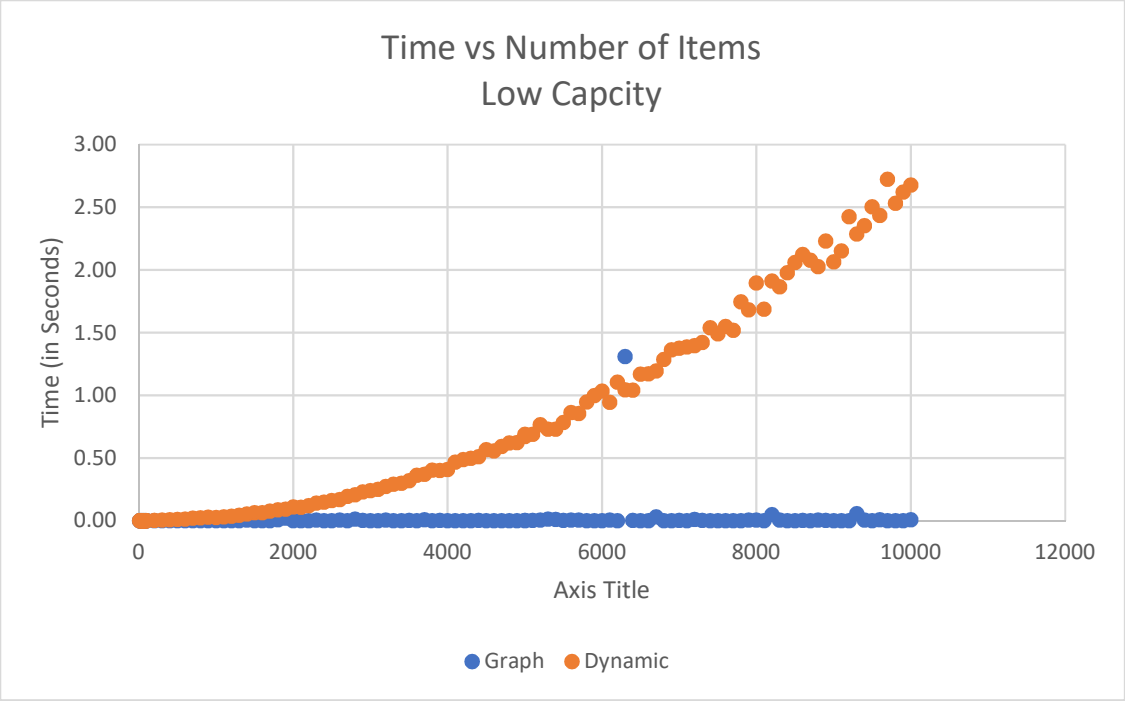
Overall, sanitization of input allowed for massive improvements in both efficiency and size of problem allowed as although in the worst case this algorithm performs identically to brute force, with pruning, and correctly selecting the problems to apply this to we can drastically speed up results.

Analysis

The Graph search approach and Dynamic programming approaches (as implemented by me) are quite different. My dynamic programming solution has an efficiency of $O(n^2)$ while my implementation of the graph search algorithm has $O(2^n)$. With such a vast difference in time complexity, they, in theory, should not be compared at all. However, as demonstrated briefly above, if we select the proper inputs and problems, we can drastically improve the graph searches actual execution times, and while it may not be as consistent, can beat the dynamic solution on certain problems. However, in the same way that some problems are best suited for the graph search algorithm, some instances of the problem put the graph search implementation towards its worst case consistently.

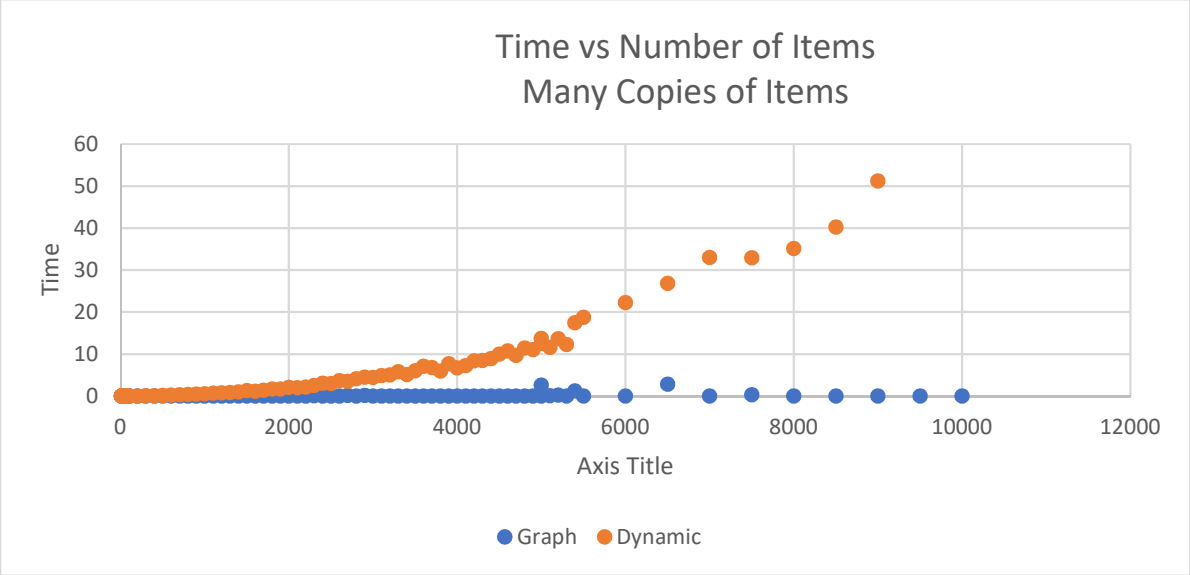
If we are to consider completely random problem generation, that is where the length of items is equal to the capacity of the knapsack (in the way I have set it up) and there are nearly completely random generation of an items weight, value, and quantity, we cannot even begin to compare the graph search and dynamic solution. The dynamic solution was not able to solve problems with a weight/capacity over 170 and did so in upwards of 3 minutes time with the later trials while the dynamic solution was able to solve problems with 10,000 items and a capacity of 10,000 in under a minute on average. However, once we start to more rigorously select the problems we give to the algorithms, we start to see some interesting results.

The first notable problem selection is when we limit the capacity of the knapsack to ten percent of the total number of items, in other words the knapsack is very small but there is a lot of items to choose from. In this case the graph search algorithm vastly outperforms the dynamic solution as shown by the graph below:



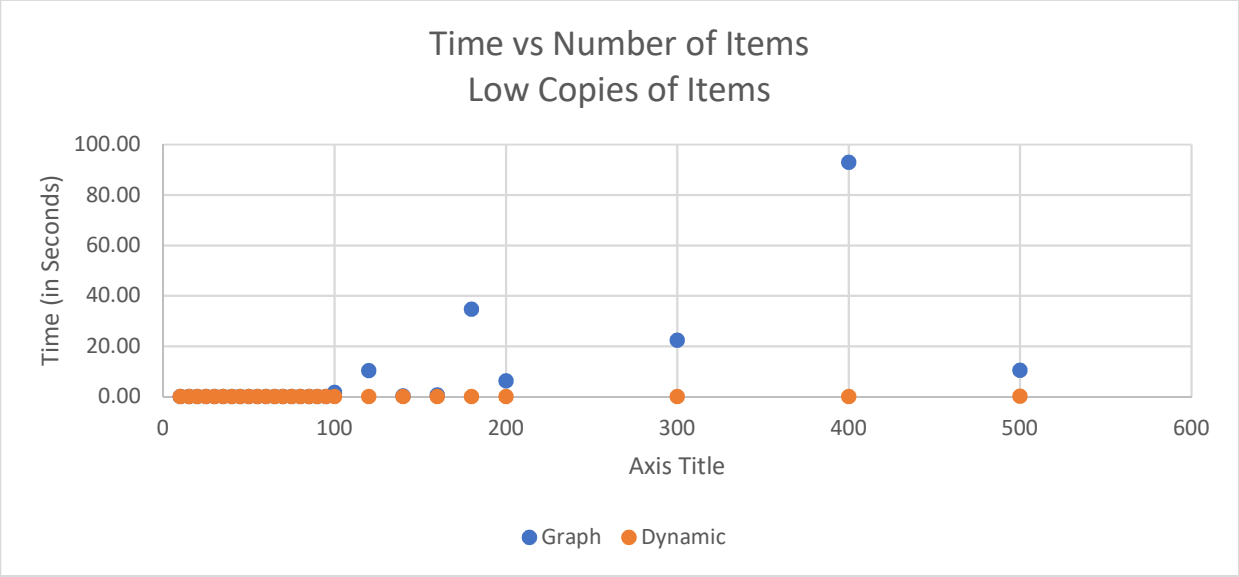
As we can see, the dynamic solution increases in a roughly n^2 fashion while the graph search implementation appears almost linear. This is a huge improvement for the graph search as we are able to reach and items length of 10,000 which is nearly a tenfold improvement over the completely rando data, and the fact that it beats the dynamic solution on average, by a great deal none the less.

The second notable problem selection is when we limit the number of unique items, that is there are many copies of few different types of items. When we do this, we get a similar speed up to the last generation problem as shown below:



The Dynamic solution, as the total number of items increases, increases roughly at an n^2 path as shown before, but the dynamic solution appears, once again, nearly linear. This is all due to the problem generation, and since there are a fewer number of unique items, each state in the graph search algorithm has many less children, as there are less unique items. This drastically cuts down on the recursion requirements for the execution, and the improvements are quite notable, as shown above.

As good as these results are, however, it is not all great for the graph search algorithm. As shown with the completely random generation, there are specific problems that do not cater to the graph search's strengths. A specific instance of this is if there are low amounts of copies of items, and high numbers of unique items. This means that each node will have a much higher chance of having a higher number of children, thus increasing the recursion costs for the graph search algorithm. The results, shown below, represent a problem generation where each item can have at maximum 5% of the total items, meaning there will be many more unique items than before.



As shown above, the graph search algorithm severely struggles with larger data sets and cannot execute any problem in the manner described above with a total items list longer than 500. And, at the same time it executes the problems with much larger variation, and on average much greater time.

These differences between graph search and dynamic programming highlight the need to choose the correct algorithm for the problem, as choosing a non-optimal solution can severely hinder the efficiency of solving the problem.