# Knapsack Problem

Daniel Braithwaite

September 27, 2015

# 1 Dynamic Knapsack 0-1

## 1.1 Algorithm

Given a list of items $item_1, .....item_n$, say we have a table T that in position T[i][j] stores the best solution's for packing items $item_1, ....item_i$ into a bag with capacity j. We can use the following recurrence

$$max \begin{cases} T[i-1][j] \\ T[i-1][j - items[i].weight] + items[i].value \end{cases} \tag{1}$$

```
1:  procedure DNAMICKNAPSACK01(capacity, items)
2:      T ← int[items.length][capacity + 1]
3:      for i = 0, i < items.length, i ← i + 1 do
4:          for j = 0, j < capacity + 1, j ← j + 1 do
5:              Fill out cell based on recurrence
6:      solution ← [0....0] (n zeros)
7:      i ← items.length − 1
8:      j ← capacity − 1
9:      while j > 0 do                    ▷ while we have available capacity
10:         Use recurrence to figure out what action we took to get this value
11:         if We took item i to get here then
12:             j ← items[i].weight
13:             solution[i] ← 1
14:         i ← i − 1
```

## 1.2 Proof

We want to work our way towards the recurrence we used in the dynamic programming algorithm. Say we have $n$ items we are trying to put into a bag of capacity $C$. Ether we put the $n^{th}$ item in the bag or we didn't. From this we get the following two sub problems.

1. **We took the $n^{th}$ item:** Here we are solving the sub problem where we have items $item_1, ....., item_{n-1}$ and bag with capacity $C - item_n.weight$.
2. **Didn't take $n^{th}$ item:** Here the sub problem is the items $item_1, ....., item_{n-1}$ and bag with capacity $C$.

This observation gives us our recurrence, now we need to show optimal sub structure for both of these cases

1. Say we took the $n^{th}$ item. Let $S$ be set of items taken giving the optimal solution for this problem. Assume for a contradiction that $S - \{item_n\}$

isn't the optimal solution for the problem $item_1, ....., item_{n-1}$ with a bag of capacity $C - item_n.weight$. Let $S'$ be an optimal solution for this sub problem. But then we could take $S'$ and add $n$ back in. This would give us a better solution than $S$ for the whole problem. A contradiction!

2. Now say we didn't take the $n^{th}$ item. Let $S$ be the optimal selection of items. Assume for a contradiction that $S$ is not the optimal solution for the sub problem $item_1, ....., item_{n-1}$ with capacity $C$. But then we could take a better solution say $S'$, not choose the $n^{th}$ item and have a solution better than $S$. A contradiction!

Now we see the Knapsack 0-1 problem as optimal substructure. Now we must prove the algorithm is correct. First we look at filling out the table.

Fill out the table must be correct as we just proved the problem had optimal substructure using that recurrence. The last thing to show is that recovering the solution from the table is always correct.

To start this we see that this also uses the recurrence we defined to determine what action was taken. If an item was taken then we mark it as taken in the solution. With ether action we move back in the table to where that solution came from until we have $j$ at 0 which means that we have no more available capacity.

## 1.3 Complexity

### 1.3.1 Filling Out Table

The main part of the algorithm cost is here. The table is of size $weight * numItems$ and no matter the input data the whole table has to be filed out. The cost of using the recurrence is $O(1)$. Therefore the cost for filling out the table is

$$\theta(nC) \tag{2}$$

### 1.3.2 Finding Solution

The cost of this involves moving through the table and recovering the decisions at each point. Finding what decision was made is $O(1)$. At worst case you would be moving one square in the table at a time. Therefore finding the solution has cost

$$O(n + C) \tag{3}$$

### 1.3.3 Overall Cost

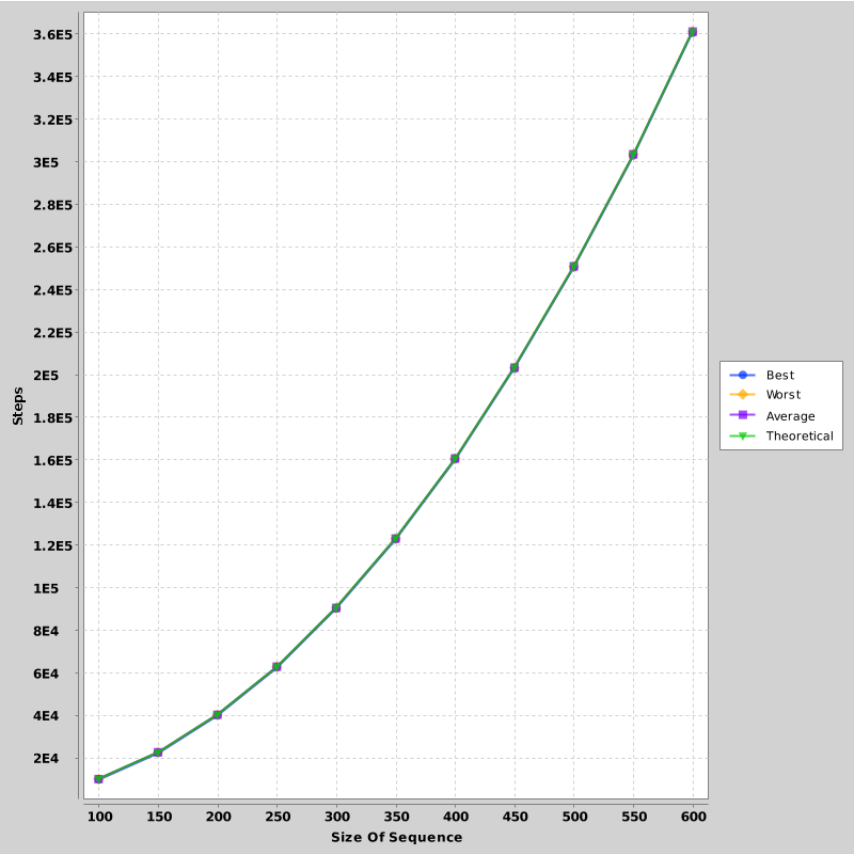
Combing the cost of the previous sections we see this algorithm has the following cost

$$\theta(nC) \tag{4}$$

## 1.4 Testing

I plotted the lines for the best, average and worst case data this algorithm could encounter. I defined the worst case data to be as follows.

1. **Best Case** is when reconstructing the solution we can just do it all in one step. So for this to happen we set all the item values and weights to be the same.
2. **Worst Case** is when we take as many steps through the table as possible. To get this we set all the weights to be 1.
3. **Average Case** is just randomly generated data




No matter the input for this algorithm the dynamic programming table will have to be filled. The only thing that will cause a difference will be how long it will take to recover the solution. However this cost is minimal compared to the cost of computing the table. So for some input size $n$ if we have best, worst or average case data the number of steps will be almost identical for all of them. This is why you can only see one line, as they are all overlapping. All the lines fit perfect with the theoretical.

# 2 Brute Force Knapsack 0-N

## 2.1 Algorithm

```
1: procedure BRUTEFORCEKNAPSACK01(capacity, items)
2:     w, b ← [0, ....., 0]                    ▷ number of entries equals items.length
3:     while true do
4:         w[0] ← w[0] + 1
5:         for index i in w do ▷ Handle when we are taking to much of an item
6:             if w[i] > item_i.mutiplicity then
7:                 w[i] ← 0
8:                 w[i + 1] ← w[i + 1] + 1
9:         if we have taken all of the last item then    ▷ this is true if we have
   enumerated all possible solutions
10:            break
11:        weight ← weight of possible solution w
12:        if weight ⩽ capacity then
13:            Store solution in b if the value is better
       return b
```

## 2.2 Proof

The brute force algorithm works by enumerating all the possible options and storing the one with the best value if its weight is less or equal to the capacity

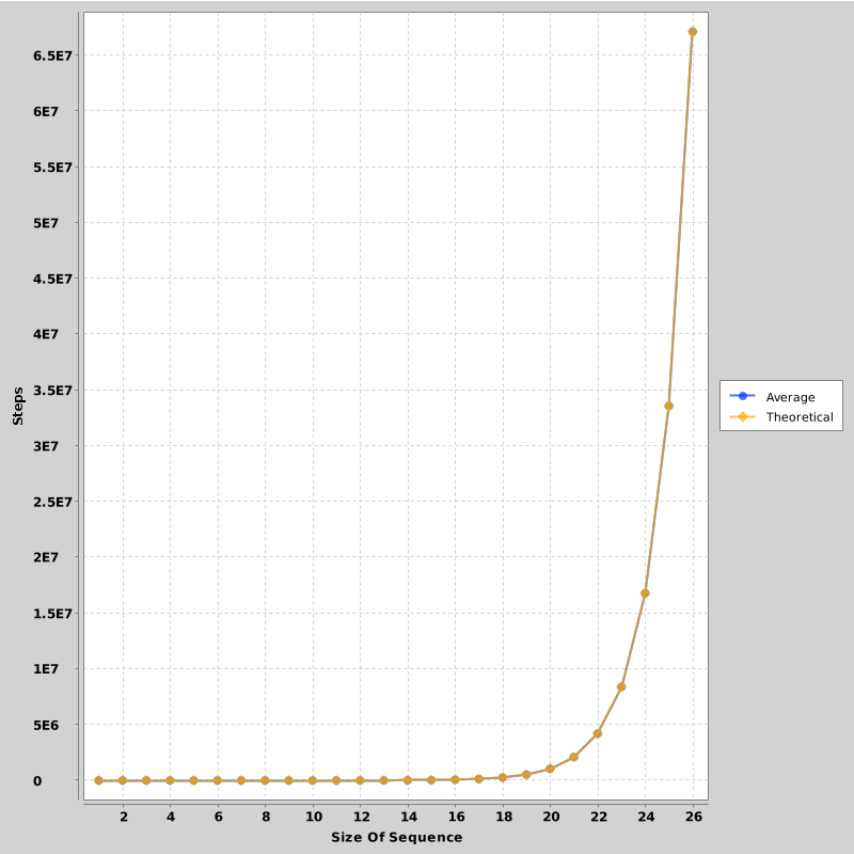
## 2.3 Complexity

This algorithm has to enumerate all the possible combinations of items. No matter what the input data is (best, worst, average case) it will still have to enumerate all the solutions as it has no way of being able to tell if it has found the best one except for checking the all the remaining ones. We get the complexity to be

$$\theta(2^n) \tag{5}$$

## 2.4 Testing

There is no best, worst or average case, as discussed in **2.3**. As such we only graphed the average case and the theoretical line. When graphing this we only ran it on very small data sets as it is a very inefficient algorithm.
As you can see in the graph the average case line matches up perfectly with the theoretical line.




# 3 Problem Conversion 0-N to 0-1

The following 0-N algorithms will convert there 0-N knapsack problems to 0-1 problems. Here we will detail the algorithm for doing this with the cost. Rather than actually producing a new list of items and working with that we will create a mapping. Say our item list has size $n'$ and this list represents $n$ items. Then our mapping take numbers in the range 0 to n and map them to numbers between 0 and $n'$.

## 3.1 Algorithm

```
1: procedure createMapping(items)
2:     mapping ← []
3:     i ← 0, j ← 1, k ← 0
4:     while k < items.length do
5:         itemMapping.map(i to k)
6:         if j = items[k].multiplicity then
7:             j ← 0
8:             k ← k + 1
9:         i ← i + 1
10:        j ← j + 1
       return mapping
```

## 3.2 Proof

For each item passed in has a multiplicity (i.e. how many of that item we have) if we examine the algorithm we can clearly that it will create a mapping to an item item.multiplicity times. From this we see that the algorithm is correct.

## 3.3 Cost

We can clearly see that the loop is going to iterate exactly once for each of the $n$ items making the cost

$$\theta(n) \tag{6}$$

# 4 Dynamic Knapsack 0-N

## 4.1 Algorithm

Here we can use our previous 0-1 knapsack algorithm to solve this one. We just need to create an algorithm to convert a 0-N knapsack problem to a 0-1 problem and then once solved convert the solution back. The key difference being that if we want to access item k then really what we are doing is accessing the item k is mapped to ($mapping[k]$).

```
1:  procedure DNAMICKNAPSACK0N(capacity, items)
2:      mapping ← createMapping(items)
3:      T ← int[mapping.length][capacity + 1]
4:      for i = 0, i < mapping.length, i ← i + 1 do
5:          for j = 0, j < capacity + 1, j ← j + 1 do
6:              Fill out cell based on recurrence
7:      solution ← [0....0]                          ▷ items.length number of 0's
8:      i ← items.length − 1
9:      j ← capacity − 1
10:     while j > 0 do                          ▷ while we have available capacity
11:         Use recurrence to figure out what action we took to get this value
12:         if We took item i to get here then
13:             j ← items[mapping[i]].weight
14:             solution[mapping[i]]++
15:         i ← i − 1
```

## 4.2 Proof

We see this algorithm works by converting the 0-N knapsack problem into a 0-1 knapsack problem and uses the 0-1 dynamic algorithm to solve it. The solution is then converted back to 0-N format.

We know that the conversion to 0-1 format is correct, we also know that the dynamic algorithm for solving the 0-1 knapsack is correct. So all we need prove is that the conversion of the solution back to 0-N is correct.

By inspecting the algorithm we see that our solution has an entry for each of the original items. We are using the recurrence to determine if an item was taken as we traverse the table. If an item was taken then we need to figure out what one of the original items it maps to and increment our count of that original item in the solution. We know we are able to do this correctly because our mapping is correct.

## 4.3 Complexity

There are two parts for this algorithm we will explore the complexity for each of them and then we can come up with an overall complexity

### 4.3.1 Solving The 0-1 Problem

We already know the complexity of this and it is

$$\theta(nC) \tag{7}$$

### 4.3.2 Recovering Solution

The complexity of recovering the solution is the same as it was in the 0-1 dynamic solution. We only changed how the items where accessed

$$\emptyset(n + C) \tag{8}$$

### 4.3.3 Overall Complexity

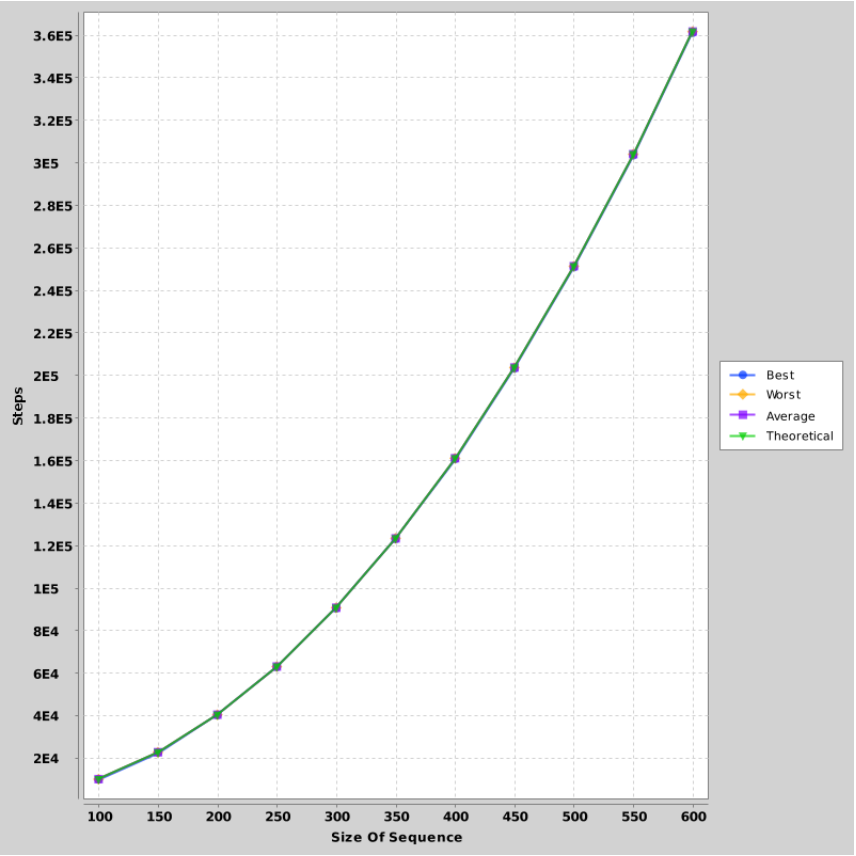
The dominating cost is still that of solving the 0-1 Knapsack problem giving this algorithm

$$\theta(nC) \tag{9}$$

## 4.4 Testing

**Please Note:** For the testing of all the knapsack 0-N algorithms we used data where none of the items had a multiplicity over 1 (for simplicity). This **wouldn't** effect the results as the conversion will still run and the rest of the algorithm will run for the same amount of time had there been less input items but the same total n items

We tested this data on the same type of best, average and worst case data as we did for the 0-1 dynamic knapsack and as expected got the same result.




We can see that all the lines match up perfectly with the theoretical data

# 5 Graph Search Knapsack 0-N

To start with we want to convert the 0-N problem to a 0-1 problem. This is so we can use a similar idea to the dynamic programming table[1]. Each node will be represented by the following triple $< i, j, value, previous >$ where $0 \leqslant i \leqslant numItems$ and $0 \leqslant j \leqslant capacity$ . The i and j mean the node contains the best solution for the items 0 to i-1 for a knapsack of capacity j.

A node is connected to at most two others. Let $n$ be a node. The possibility's are ether we take item n.i or we don't. In the first case n is linked to a node where i = n.i + 1, and j = n.j + item[i] (as long as n.j + item[i] is $\leqslant$ capacity. In the second case n is linked to the node where i = n.i + 1, and j = n.j. As we are building the graph we want to keep track of the current best value of each node, along with that we keep track of the previous node that was used to get this value.
For simplicity we will sometimes refer to a node as just ¡i,j¿. Then we can refer to the value and previous node as ¡i,j¿.value and ¡i,j¿.previous

## 5.1 Algorithm

---

1: **procedure** GRAPHKNAPSACK0N($capacity, items$)
2: $items' \leftarrow convertTo01Problem(items)$
3: $fringe \leftarrow []$
4: add root node $< 0, 0, 0, null >$
5: **while** fringe is not empty **do**
6: $current \leftarrow nodefromfringe$
7: remove current from fringe
8: **if** $current.i + 1 > items'$ **then**
9: return to top of loop
10: get node $< current.i + 1, current.j >$ if it doesn't exist add it to fringe.
11: update $< current.i + 1, current.j >$ if our current path to the node is better
12: **if** there is enough available weight to take item j **then**
13: get node $< current.i + 1, current.j + items'[i].weight >$ if it doesn't exist add it to fringe
14: update that node if there is a better path to it
15: $solution \leftarrow [0, ....., 0]$ ▷ same length as items

---

[1] http://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf

## 5.2 Proof

We will need to prove two things for this algorithm to be correct. First being that our method of generating the graph will explore all the possibility's. Second we need to show that we can recover the best path through the graph, thus giving us the optimal solution and finally show that the part of the algorithm that recovers the solution also correctly converts it back to a 0-N solution.

We think of nodes as representing a sub problem. For example the node $< 2, 5 >$ is saying we have considered the items 0 and 1, and that we have used up 5 of our total capacity. From here we have two choices ether we take item 2 (taking item 2 also depends on weather we have enough spare capacity) or we don't take item $i$. Meaning there is an edge between $< 2, 5 >$ and the nodes $< 3, 5 >$ and $< 3, 5 + 1 >$ (assuming the weight of item 2 is 1). If we make this more general and apply it to some node $< i, j >$ then clearly we are searching all the possible combinations. So if we store the node with the best value as we are building this binary tree then by the end we clearly will have a node that holds the best possible value

Now we have a node that can tell us the best value possible when trying to put all these items into a back with a known capacity. But what we are really interested in is the combination of items. As we are building the graph when we link two nodes with an edge, say $n$ to $n'$, we examine the current value of $n'$ and see if the value of $n$ (plus the value of item $n.i$ we are taking it) is greater than it. If it is then we store $n$ as the previous node in $n'$. We do this for all nodes so at the end we can just step back through the tree starting from the best node giving us the solution.

Finally we look at the part of the code that recovers the solution. This starts at the node in the graph that has the best value stepping back though the nodes using the previous value. At any iteration in the loop we have a node $n$ and the one before it $n_p$. We compare $n.j$ and $n_p.j$ if they are equal then we didn't take an item so just set $n$ to $n_p$. If they are not equal then we update the solution by using our mapping $solution[mapping[n_p.i]]{+}{+}$. We increment the item $n_p.i$ as moving from $n_p$ to $n$ meant we where acting on the item $n_p.i$. This clearly gives us a 0-N solution that is correct.

## 5.3 Complexity

This algorithm has two components that we will explore separately.

### 5.3.1 Constructing The Graph

We can think of constructing the graph as traversing all the nodes in the graph. Which at worst case there will be as many nodes as there are cells in the dynamic programming table. However the key thing to note is that this is at worst, there

often will be less nodes than this. Giving us the following complexity where n is the number of items (that is not the number of items in the input array but the sum of there multiplicity's) and C is the total capacity of the bag

$$O(nC) \tag{10}$$

#### 5.3.2 Finding The Solution

At any node we know that it is connected to at most 2 nodes, there for what we generated is a binary tree. We also know the two following things. That there is going to be at most n + 1 layers (one for every possible value of i, $0 \leqslant i \leqslant n$). The second thing we know that as we are recovering the solution we are only stepping back through the tree. Therefore at worst we will have to step though the height of the tree giving us the complexity

$$O(n+1) \tag{11}$$

#### 5.3.3 Overall Complexity

The dominating cost of the algorithm is that of constructing the graph so we get the algorithm complexity to be

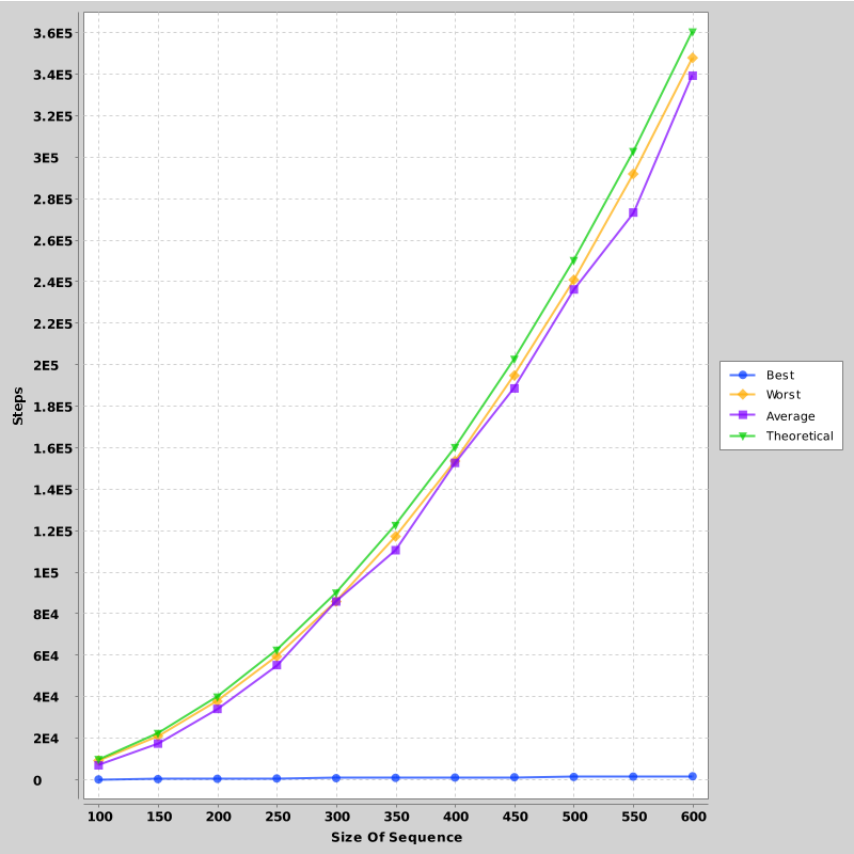
$$O(nC) \tag{12}$$

### 5.4 Testing

The graph below displays the best, average and worst case for the graph search algorithm, we defined these as follows.

1. **Best Case** is when we can construct the graph with only having one layer. If we set all the items to have a capacity that will consume all available space we can get this.
2. **Worst Case** is when we set each of the items to have a different weight, so for example for a set of 3 items we have $item_1.weight = 1, item_2.weight = 2, item_3.weight = 3$. This will give us the most possible nodes without much overlap. If we chose to do the same worst case data as we did for the dynamic programming we would do better than average as there would be a lot of overlap between nodes.
3. **Average Case** is just randomly generated data

As we see in the graph this is much more varied than the other algorithms. The best case is very different from the average and worst case which are quite close together. However they are all less than the theoretical line which I graphed as $n * C + n + 1$ as this is the maximum number of steps the algorithm can take.




# 6 Graph Search vs Dynamic

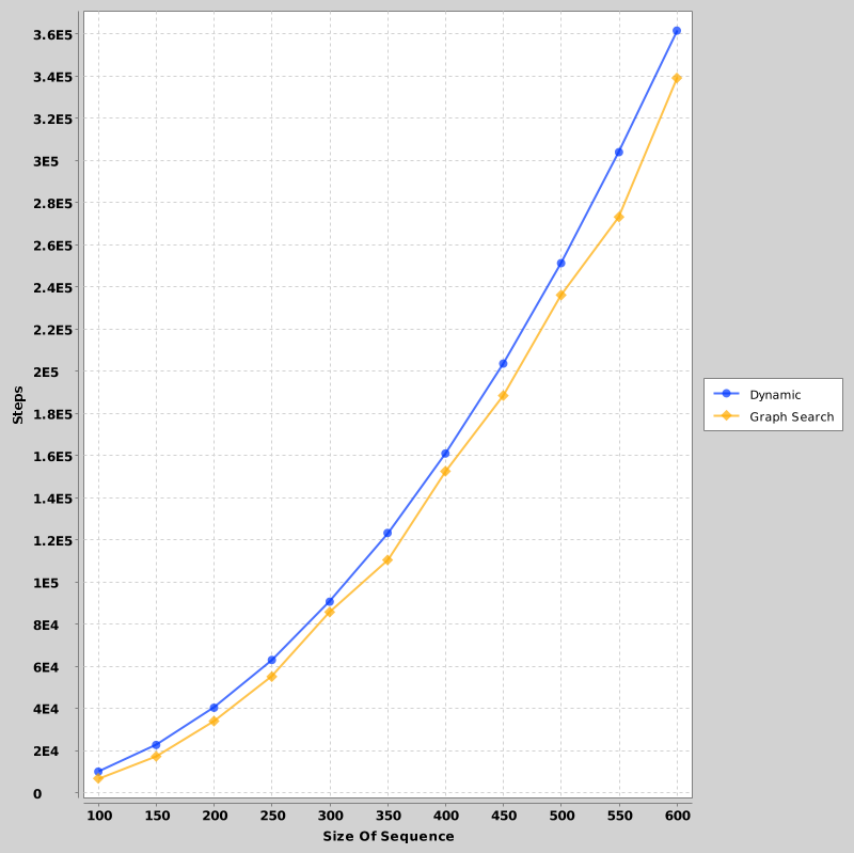
If we examine the order costs of the algorithm we would expect that the graph search would out perform the dynamic algorithm, or atleast perform the same as it. The graph search algorithm has a complexity of $O(nC)$ meaning it is bounded from above by $nC$. Compared to the complexity of the dynamic algorithm $\theta(nC)$ meaning it is bounded from above and below by $nC$.

If we examine the algorithms more closely we can also see this as noted in the dynamic algorithm complexity section the algorithm will always have to fill out the entire table no matter the input data. We also noted previously that in the graph search it was only worst case where we would have to traverse $nC$ number of nodes.

We can conclude from this that the graph search algorithm will perform at most $nC$ steps where as the dynamic algorithm will always perform $nC$ steps.

We can also examine the cost of reconstructing the solution for each of the algorithms. Even here the graph search algorithm is faster, with a complexity of $O(n+1)$ compared to the dynamic algorithm where recovering the solution is $O(n+C)$.

With this in mind we compare the best case of the dynamic algorithm and the worst case of the graph search.




As we can see in the graph the graph search algorithm clearly out performs the dynamic algorithm.