# **Motivations**

The basic idea of our project is to implement the 0-1 knapsack problem using the brute force method and the dynamic programming method. We want to find the maximum value for a given knapsack problem for a total number of n items, with the weight of the knapsack defined as W, the values of the items defined as v1, v2,… vn and the weights of the items defined as w1,w2…wn. This is done using three different algorithms, the brute force algorithm, the bottom-up dynamic programming algorithm and the top-down dynamic programming algorithm. After implementing the three algorithms with respect to a set of test cases which vary based on the dataset size, we attempt to find which type of algorithm executes in the fastest amount of time, and overall, which algorithm works better, the brute force one or the dynamic programming one. We are using Java to find out which algorithm will provide us the most optimal solution for the knapsack problem. We then represent our results with the help of graphs represented in the form of time represented in nanoseconds vs dataset size to show which particular algorithm works best, and how its behavior varies when we change the dataset size. Based on this data, we then conclude on which algorithm provides the optimal and best solution for the knapsack problem.

**Experimental**

## **Data Set**

## To generate the data sets for Values and Weights, we relied on Java pseudo-Random API as part of the Standard Oracle JDK and generated variable size arrays to pass in each algorithm, and for the capacity size, we also used a random integer generator.

## The structure of the test scenario was designed by running all three algorithms in sequence and with one individual data set, to make sure all of them are working on the same data set. Also, to decrease the effect of cold run on our time calculation, we executed all algorithms in several iterations with the same data set size and captured their times separately, also trying to exclude all the IO work in middle of the algorithms, so in actual execution after development, testing and debugging, there would be no IO and logging mechanism.

## All the times were recorded in nanoseconds and processed the same way to increase accuracy however since each algorithm would be executed in several iterations with the same data set size, to make sure in each iteration we are re-generating a new random array as dataset to make sure caching won’t also have a direct reflection in captured times.

## **Execution Environment**

For execution environment we were using a 64-bit Windows Machine 8, with 16GB of RAM and a Intel Core i7 2.5 GHz, executed in a separate times in separate attempts when the windows restarted and fresh, with no internet connectivity and no other application’s running directly to mitigate the reflect of any noises in our captured time.

The algorithm implementation language is Java using standard Oracle JDK 1.8 update 45.

## **Test Scenarios**

In this project we have implemented three separate test cases, as below:

* Executing all algorithm implementations using variable different dataset but all would be in the same length, so as input keep n unchanged for all regression iterations and just randomly re-generate data set values which change the W in each iteration; by that, we tried to mimic regression and get the Average to mitigate noises affects (warm vs. cold data in memory can be reduced.)
* Executing all algorithm implementations using several different dataset but all would be in the same length and same W across all the iterations.
* Executing both Dynamic programing algorithms by keeping the n same in all iterations and randomly generating W and datasets.
* Executing both Dynamic programing algorithms in all iterations keeping n and W for datasets the same to decrease the effect of warm and cold.

Note that: we are executing each data set for all the included algorithms in each test case to make sure all are using the same input, and we are comparing the them against the same parameters.

# **Result**

Comparing Execution time Per Dataset Size

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set Size | Table 0-1 Knapsack Algorithm Implementations Name | | |
| Brute Force Run Time (Nano Sec) | Dynamic Programming Bottom-Up Run Time (Nano Sec) | Dynamic Programming Top-Down Run Time (Nano Sec) |
| 1 | 5.176448199 | 6.282058622 | 5.308843894 |
| 2 | 5.161136317 | 6.365615447 | 5.294300854 |
| 3 | 5.175654815 | 6.383379058 | 5.324324991 |
| 5 | 5.190854405 | 6.756501895 | 5.458873437 |
| 10 | 5.348161373 | 6.784796013 | 5.75907655 |
| 15 | 5.700906975 | 6.880408614 | 5.973071176 |
| 20 | 6.723929361 | 7.020152014 | 6.158504318 |
| 25 | 8.060454999 | 7.121487653 | 6.271405312 |
| 30 | 9.639010859 | 7.225251751 | 6.332748684 |
| 33 | 10.47196787 | 7.14150291 | 6.399261205 |
| 35 | 11.07260338 | 7.272859982 | 6.430354278 |
| 40 | 12.65685139 | 6.888266251 | 6.488346212 |

Table. 1. *Numerical Run time Data (All Algorithm, using same n but different W)*

In Table 1, it’s clearly showing how difference between numbers are around 2 times flowers in brute force for even a small number of n ~40.

Note that there are some spikes in the times which we have noticed it’s around the time that JVM Garbage collection is getting triggered and start cleaning up the memory, we have tried to mitigate that triggering time but using primitive as much as possible to decrease the amount of actual objects inside the heap, however after a time it was inevitable since the objects are piled in the heap memory.

### **Comparing the Results of Two Dynamic Programming Algorithms**

When we are comparing two dynamic programming, we clearly see that top down solution was working faster than the bottom up, however the top down is using a collection internally and the lookup mechanism was based on java standard hash function and for bottom up was relying on array index manipulation, we expect the hash code function took longer time, although the average running time of bottom up was longer than the top-down and it was showing top down can be a god solution for variable data sets as well.

Note that between comparing dynamic programming algorithms, if we keep the n and W the same or change the W only, there is not much a difference between the times, however if we keep the n the same and change the W in each iteration, there are few places that the times are getting closer which is showing the performance of them can be very similar for different W and the same data set *size*, although if there is an application that we need to keep both the same it seems top down is getting the result very faster.

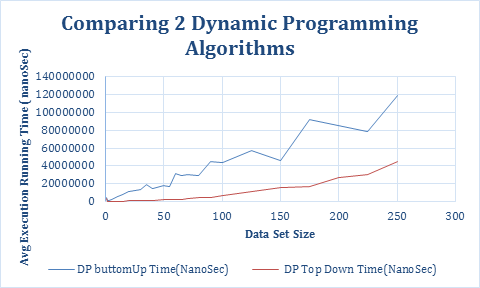


Fig. 1. *Comparison between running time of 2 dynamic algorithm implementations against different data set sizes. (keep the same n and W)*

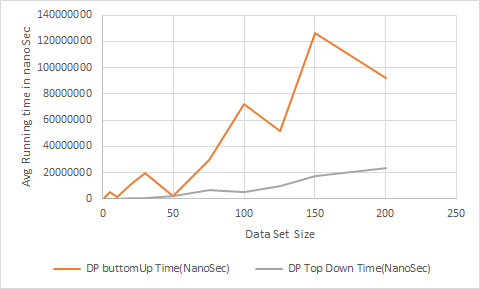
**

Fig. 2. *Comparison between running time of 2 dynamic algorithm implementations against different data set sizes. (keep the same n different W)*

### **Comparing Execution time Per Data set Size of All three Algorithms**

Since brute force was taking a very long time for dataset more than 40, we had a time limitation to execute that on larger data set, however even with data sets until 40, it’s charily showing that Brute force solution is taking longer time to execute and the most reasons is because of its running time complexity of calculating all the possibilities and finding the best one, across all the recursive calls.

Also note that regardless of keeping W the same or randomly updating it with the same size of n, the rate of brute force is really won’t change and it’s basically reflected by size of data set (n).

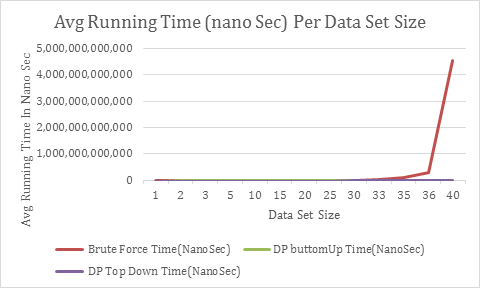


Fig. 3. *Comparison between running time of all 3 algorithm implementations against different data set sizes.(keep same n and W)*

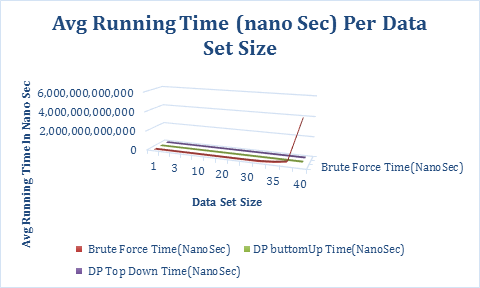


Fig. 4. *Comparison between running time of all three algorithm implementations against different data set sizes from another view to have all three visible.(same n and W different Viewpoint to see all algorithms and the growth rate)*

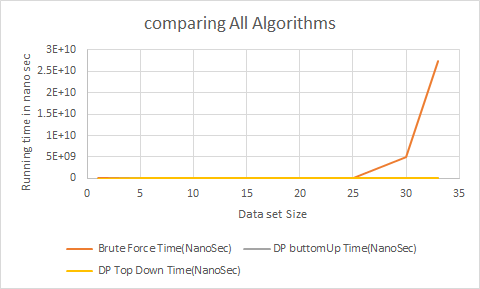


Fig. 4. *Comparison between running time of all three algorithm implementations against different data set sizes from another view to have all three visible.(same n and different W)*

## **Conclusion**

Based on the charts, it’s obvious that brute force algorithm is not practical for the larger datasets and it’s taking a long time to come up with an answer for example for data set size 40 integers it took around 15 hours to finish the processing. From the other side dynamic programming solutions were saving time by using the already calculated results and their calculating possibilities were very intelligent and narrowed which caused faster calculation, so the same data set sizes can be calculated less than a min. this is expressing that dynamic programming solutions are more practical for the larger data set sizes, and specifically between the two Dynamic programming implementations, the top-down was the fastest.

Aside of Technical Details of How these algorithms can be compared, we have learned the actual implementation of the algorithms can be challenging and required more debugging however in pseudocode, there are several low-level details that was being wrapped up in the abstraction layer and in the actual implementation are being exposed, e.g. there was a challenging debugging efforts were put on comparing the algorithm brute force into the actual Implementations and handling the boundaries and finding the correct results.

We also learned in practice if we are using dynamic programming solution for knapsack 0-1, we can use either of top down or bottom up, but we can also narrow it down and choose a special one depending on the data set nature and the Capacity along with the size of n.

//index in each level

Algorithm calculateBruteForce(int[] values, int[] weights, int capacity, int index) {

if (capacity <= 0 || index < 0 || index == values.length)

return 0;

end If

return max(values[index] + calculateBruteForce(values,weights,capacity-weights[index],index-1)

, calculateBruteForce(values,weights,capacity,index-1));

}

Algorithm KnapsackBottomUpDp(int[] values, int[] weights, int capacity) {

//n \* W = n X capacity array showing the final result;

int[][] selection = new int[values.length+1][capacity+1];

//init the final table.

for(int w = 0 ; w <= capacity;w++){

selection[0][w] = 0; //row 0 = 0

}

int n = weights.length;

for(int i = 0 ; i <= n ;i++){

selection[i][0] = 0;// all items in col 0 = 0

}

//start picking - iterating through

for(int i = 1 ; i <= n ; i++){

for(int w = 1 ; w <= capacity;w++){//iterating based on weights inside the selections

//can we go forward based on the remaining weights

if(weights[i-1] <= w){

//compare and see if it needs to pick it or not.

if( values[i-1] + selection[i-1][w-weights[i-1]] >= selection[i-1][w]){

//picked that since we still have place inside the knapsack

selection[i][w] = values[i-1] + selection[i-1][w-weights[i-1]];

end if else

selection[i][w] = selection[i-1][w]; //didn't pick that

end else

end if

}

}

print selection

}