William McLaughlin

Evan LeValley

Project 4 Report

Task 1b was our space-efficient Dynamic Programming algorithm. It promised similar results to 1a, but using less space and more time. The way we implemented this was by using an open hashing function as well as a linked list to store our data.

In 1a, the data was stored in a 2D array, which was always K = N*W. Our hash is currently running at K = W/2, which is much smaller than 1a. Our algorithm took input i,j, which gave it similar functionality to a standard list. The values were inserted at the specific index generated by the hashing function as an element of a linked list.

We chose our K initially by trial and error. While developing, we left it at K = (N*W) to make sure everything working without worrying about collisions. Later on, I saw Dr. Gill give W/2 as a hint, but when I implemented it at the time, it took too long to run for my patience. The lower the K, the smaller the space, but the more collisions.

Towards the final stages of our project, we remembered from in class that the weight and capacity values are relative to each other and can be manipulated as long as they remain the same relative to each other. My implementation checks the value of the smallest weight in the array. If that value is at least 1,000, my algorithm will divide every weight and the capacity by a constant based on

val = $10^{(\text{len(smallest)} - 2)}$ Then the weight and capacity area all divided by val.

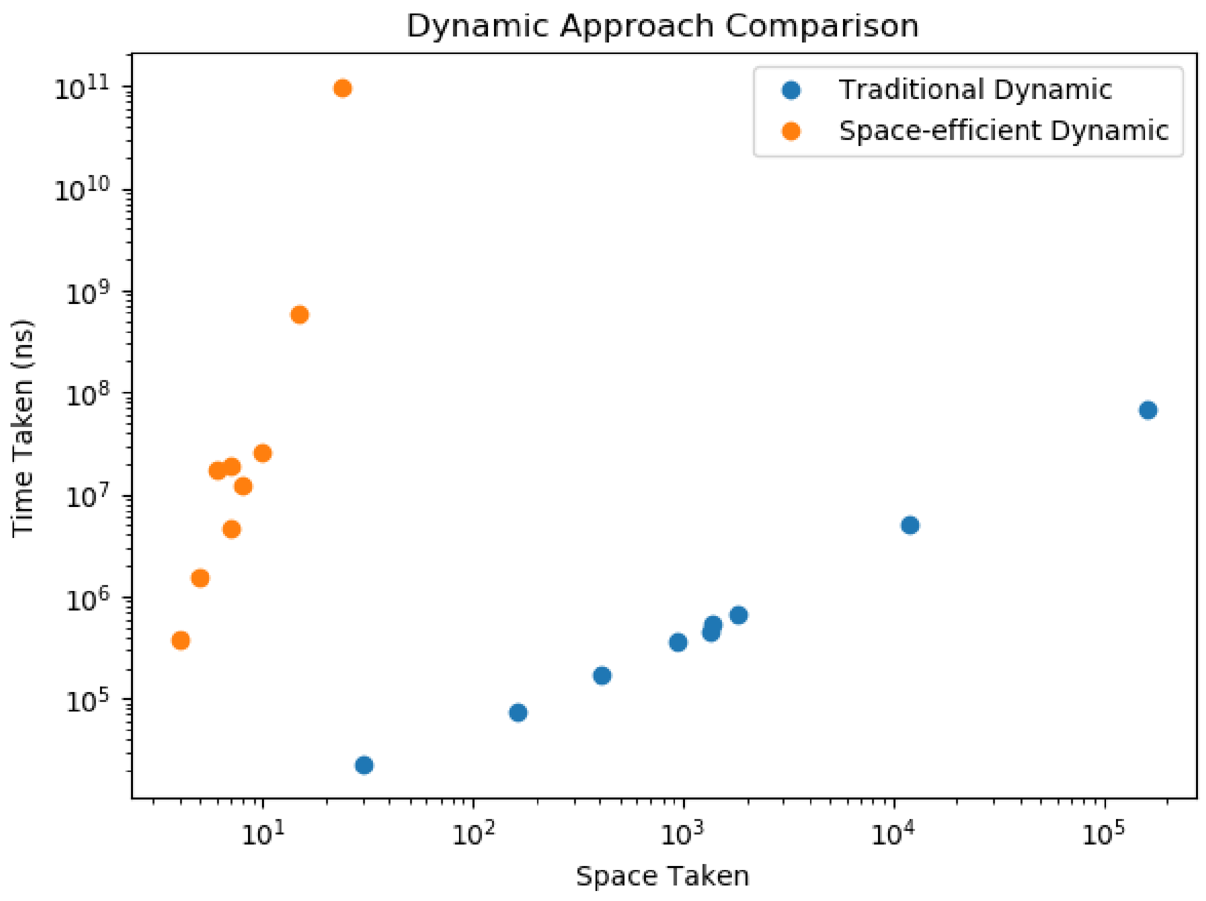
If the input was:

Cap = 3500

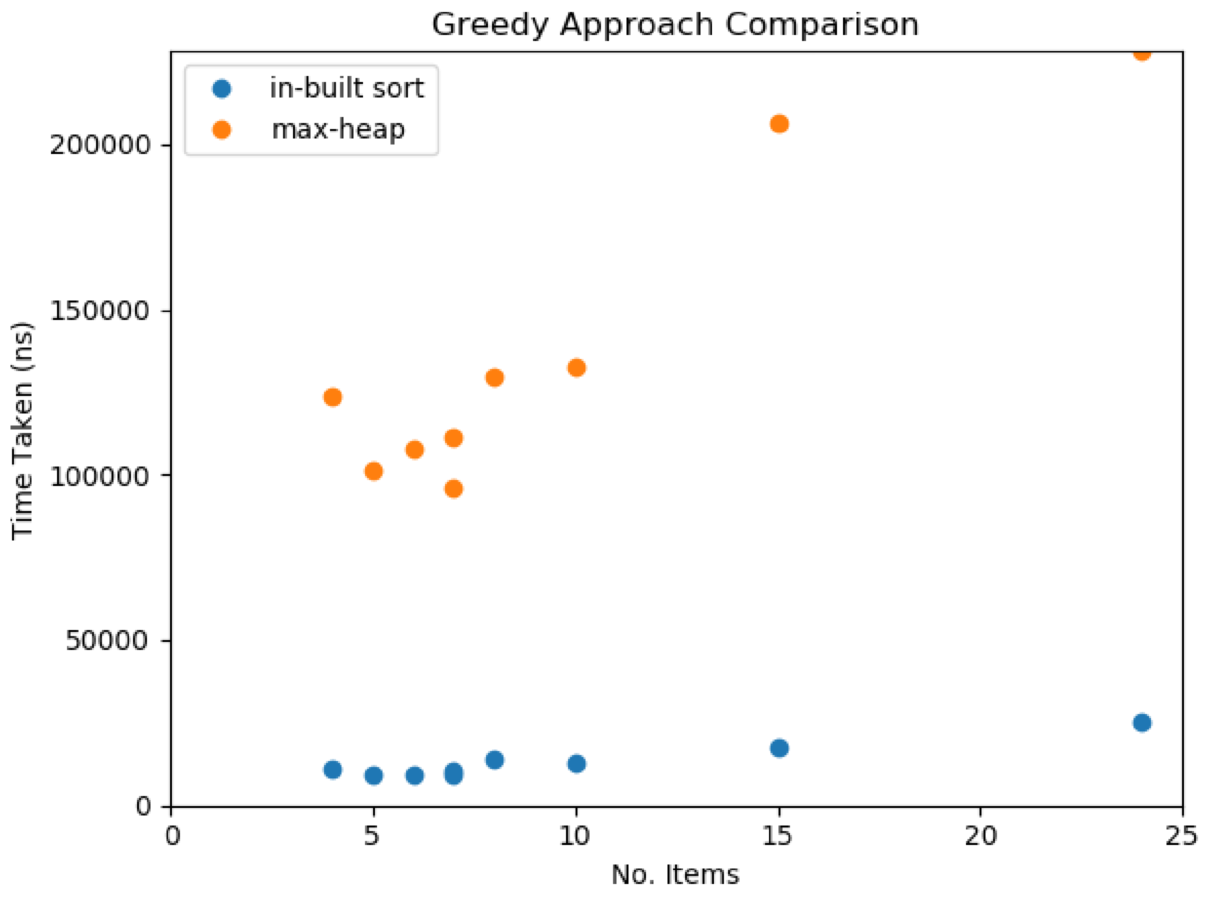
Weights = [12000, 450000, 1200, 55000]

Val = 100, Cap = 35, and the Weights become [120, 4500, 12, 550]

This change allowed us to change K = W/2 without taking much longer, and using much less space. It also ensures input of similar relative size will always be processed quickly, but at the cost of precision.



This is our graph for our Dynamic approach. We see the time taken for each input on the left and the amount of space taken on the right. Both of these data sets grow exponentially in each direction, which makes it trickier to show the data's trend. By putting the x and y-axis on a logarithmic scale, we can fit all the results on the same graph as well as show the trend for each. The space-efficient method uses W/2 space, which is one slot for every two input into it, while the Traditional Dynamic method is using N*W space and growing linearly.



This graph shows the trend of time taken vs. number of items fed to our greedy approach. Using in-built sort, the data was processed much faster than using our MaxHeap. In many cases, it took 1000-10,000% longer to use the heap than the built-in sort. This contradicts my initial hypothesis. The MaxHeap, using the bottom-up approach, should be able to build and sort itself in O(log n) time, while the list would need O(n) time. The reason this isn't reflected in our graph is the small data set and the effect of the overhead from building the MaxHeap. With longer lists of input, the MaxHeap should outperform the list.