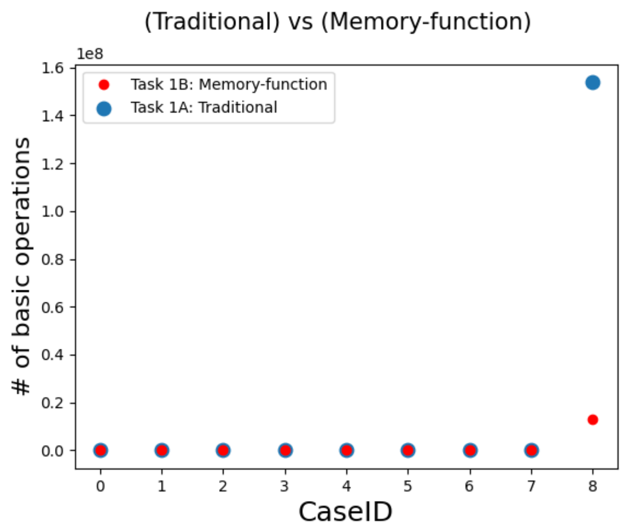
Project 04: Knapsack

Jacques Sarraffe

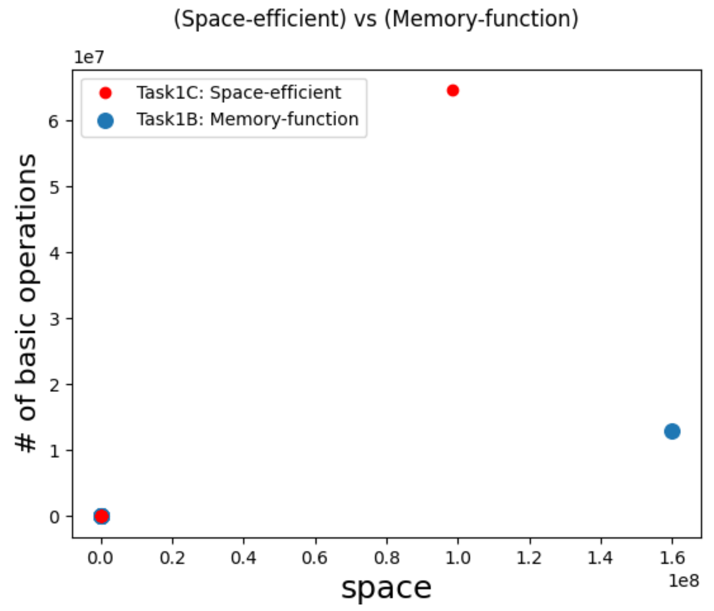
CS 415

Task 1



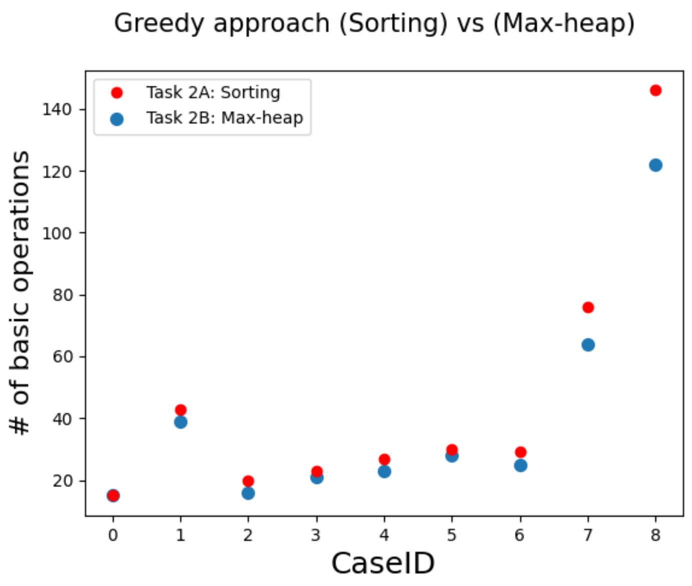
The classic dynamic programming approach works bottom-up and fills a table with solutions to all smaller subproblems, making its Space and Time Complexity clearly in Θ(nW). A disadvantage of this approach is that it solves subproblems that are unnecessary to the given problem, which is not an obstacle for the top-down brute force approach. Combining the strengths of the top-down and bottom-up approaches, we use the recursive formula, which always depends on the one right above it and the one above it and somewhere to the left, depending on the weight on the ith row. Once we reach the base case, the function unwinds. The function checks if the corresponding entries in the table are calculated yet, and If they are there, you can retrieve them; otherwise, it is calculated by the recursive call, then inserted into the table. So only computations we need for our solution are the ones that are getting computed. You can see a clear benefit from the graphs above as there are more significant gains as the size gets more extensive, like when we hit caseID 8; there is a large gap between the two points, with the memory function being significantly faster. Nevertheless, both algorithms are still in Θ(nW).

Task 1C



If the hash function were to distribute n keys among m cells of the table equally, each bucket would be approximately n/(size of hash table). The load factor plays a significant role in the number of "buckets" and "Nodes" looked at in successful and unsuccessful searches. With our hash function, the assumption is that we usually want to have the load factor close to one; this implies that having an α too small means there will be a ton of open spaces in the hash table, and having it too close to one would mean large "buckets" aka LinkedList, we will have to traverse. Multiplying the number of keys in the traditional table by 2/3 was the sweet spot that made this surprisingly more efficient. When experimenting with different sizes of k, the lowest number of basic operations occurred when the size of the hash table, k, was (n * W) *(2/3). So I'm assuming the number of keys needed to be inserted is probably a little more than (n*w) * (2/3). Looking at the graph, it seems I have found an amazingly efficient scheme. I save around half the space once we get to caseID 8. Solving this problem using a hash function is a perfect example of space and time trade-off.

Task 2



Evident by the plot, the algorithms have the same time complexity. At every caseID, the Max-Heap version of this algorithm is slightly faster in most cases. We know that Task2A depends on the sorting algorithm merge sort, which is Θ(n log n). Then the greedy algorithm does n computations, so the complexity for the greedy approach using sorting is Θ(n log n) in all cases. The difference in the max heap version of the problem is that once finished transforming the array in the heapification step that is Θ(n), you then proceed to do a variable amount of logn operations until the given threshold for the capacity has been hit. Thus, its best-case situation ends up being Θ(x log x), with x being the variable amount of delete max operations that needed to occur before the weights overflow the given capacity.