[**Algorithm Analysis**](https://classroom.google.com/u/1/c/NjE3MDMwNDQzNTRa)

**EMIRHAN BILGE BULUT**

**IZMIR**

**28.05.2020**

**1 – Overview**

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. (Unknown, 2018) The aim of this study is to design a system that will bring maximum profit for a car sales company using dynamic programming and calculate the minimum cost. In this section, differences in results in greedy and dynamic programming will also be examined.

**2- Scenario**

**Part-1**

Assume that you are the owner of a car company. Your company has enough employees to produce ‘p’ cars for each month. However, the number of the demand for the cars differs from month to month. You should design a sales plan for the next ‘x’ months. Consider ‘i’ is the index of each month (i=1,…,x) and m i is the demand for ith month. If your company needs to produce more than ‘p’ cars for a month, you can hire some interns, paying ‘d’ TL per car for that month. Moreover, if your company keeps any unsold car at the end the month, you should pay a ‘garage cost’. The garage cost will be calculated by the function G(j), where,

G(j) < G(j+1) , j > 0

G(j) > 0

**Part-2**

Besides, you must invest the payment that earned from your sales. Cost of each car is **‘B’** TL and you get half of the price at the beginning of the month and the rest will be taken at the end of the month. You have offers from **‘c’** different investment companies. Each investment company offers different rate for each month. At the end of each month, you can change your investment company by paying a taxes at a rate **‘t’** of your invested money or continue with the same investment company without paying any taxes. If dynamic programming was not used here, I would have to exponentially do more calculations exponentially every month. But since we keep the optimum values from the top for each row in our table, we gain speed.

**3- Solution and Strategies**

**Part 1**

In the first part, the first month to be considered was the situation that the first month came from the garage, but it was easily overcome by putting garage costs on top of this table. First, the work of the intern and then the waiting conditions in the garage were compared and the value of the lowest amount was written in the table. I wanted to do it without holding a two-dimensional array here (as in part 2), but I decided not to save space by thinking that there might be problems even though I kept the garage cost in the arraylist. The basic logic is to keep an intern and find the cost in the last month by adding the one from the previous month by choosing the one with the lowest price from the garage. Then, by sorting the section on the last line, we find the lowest cost when I get the first one. For the Greedy part, greedy tends to choose the best result instantly. In this case, he will never consider storage in the garage. Only for the first case, perhaps this could be calculated and optimized for a better result. But since greedy will want to choose the minimum cost, I don't think it will ever hold a result like storage in the garage. Even after the general algorithm does not have very high spikes, it will not hold much in the garage. Therefore, unlike dynamic programming, there will be no loop in the interior since it is not included in the optimum intern-garage cost calculation. Therefore, the algorithm complexity may appear lower. Greedy will deviate significantly in momentary bounce rates or high data, but its solution to the dynamic program will reach the exact result in a short time even if its algorithmic complexity is high.

**Part 2**

The logic in this section is much more practical. I did not need to hold a large table here. The reason was that he wanted us the maximum profit rather than how we followed that path. Both these programs reminded me of LCS algorithms. If he asked how he got it here, I would have designed it in this way by holding a table. The second important part is that I have examined examples of dynamic programming with many sales and cost calculations that find similar optimal results. Although creating a table is similar to graph-based solutions, I did not want to use array implementations for the space complexity in the memor. The Greedy part was very enjoyable and simple. Since Greedy wanted to choose the best result directly, it was an easy method to rank by taking the highest one by sorting the investment rates. Here, the most dominating factor of the algorithm's complexity will be the ranking algorithm. M sorting the Arra with C in each month will give a result like M x C. This will dominate the complexity of the Greedy part. In the dynamic programming section, the compare method includes n2 complexity. This is what will cause the algorithm to slow down here. Let's say the outermost cycle is the number of months, that is, the representation of M months. In this case, the algorithm complexity will be M x n2. Actually, doing things without separating functions could be clearer to understand the complexity of dynamic programming, but I did not see any negativity in order to use them in different places and also follow the divide and conquer path in the dynamic program.

**3- Algorithms and Complexities**

Dynamic programming allows us to do optimum calculations using less processing and memory without performing all calculations. Algorithmic complexity reduces from exponential to polynomial levels. But this algorithm measures complexity seriously.

Although it appears to be optimized, it is still higher than normal conditions. The complexity of our first Dynamic programming algorithm can be calculated as : (month)x(total number of cars to sell)2 , total space complexity will be (month ) x (number of cars to sell) because we keep our garage cost table. In fact, in dynamic programming algorithms, such optimum results can also make calculations more optimal. Although it is more difficult to optimize algorithm complexity, arrayLists with Vector structure can be used for space complexity, so that we can be flexible instead of array implementations and use parallelization in future calculations by saving space. Although there is no small data in this example, space optimization can also be improved from algorithms such as Linear Space Sequence Alignment (Unknown, Linear Space Sequence Alignment, 2010)to achieve larger and much more optimized results. From this study, instead of holding a matrix in the second part, with a dynamic arrayList structure, a space complexity such as (m x n) such as part 1 is reduced to n level by keeping a pre-result of the best result line.

Greedy algorithms give better results in terms of algorithmic complexity, but they cannot produce exact results like dynamic programming. They are generally preferred for their speed. The algorithm complexity for the first part is n2, but for the second part it is n2.

To summarize complexities

m = Month number

r = total number of cars to be produced

c = interest-bearing companies

|  |  |
| --- | --- |
| **DP 1** | **m x r x r** |
| **DP 2** | **m x c x c** |
| **Greedy 1** | **m x r** |
| **Greedy 2** | **m x r** |

**4 – Conclusion**

With dynamic programming, it is possible to analyze the things that can be solved with brute-force by reducing both time, space and algorithmic complexity according to much better and changing data. It has been revealed that when necessary, greedy can be efficient in situations where we do not want an exact solution-oriented and precise result, it can work faster than dynamic programming, but it can be used for situations where their accuracy is not necessarily confirmed. In greedy algorithms, if the difference between the data does not open much, it may be a good solution. But dynamic programming may be the solution that gives the clearest result. The study, which is a good example of logical solutions that enables us to save both time and memory, has emerged.

# References

Dumitru. (n.d.). *Topcoder*. Retrieved from Topcoder: https://www.topcoder.com/community/competitive-programming/tutorials/dynamic-programming-from-novice-to-advanced/

*GeeksforGeeks*. (n.d.). Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/count-number-of-paths-with-k-turns/

*GeeksforGeeks*. (2013). Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

Khalili, A. R. (2019, 10 20). Retrieved from https://dzone.com/articles/dynamic-programming

Roy, T. (2015). *Youtube*. Retrieved from Youtube: https://www.youtube.com/watch?v=NnD96abizww

Unknown. (2010, 07 13). *Linear Space Sequence Alignment*. Retrieved from ResearchGate: https://www.researchgate.net/publication/224159749\_Space\_Efficient\_Diagonal\_Linear\_Space\_Sequence\_Alignment

Unknown. (2018). *GeeksforGeeks*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/dynamic-programming/

Unknown. (n.d.). *GeeksforGeeks*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-k-times/

Unknown. (n.d.). *GeeksforGeeks*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

Unknown. (n.d.). *MIT*. Retrieved from MIT: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-algorithms-for-computational-biology-spring-2005/lecture-notes/lecture5\_newest.pdf