# “Comparison of algorithms to optimize exam preparation”

BY

KANAGHA SANTHOSHINI S (19BAI1044)

MANASA YK (19BAI1108)

A project review submitted to

**DR. PATTABIRAMAN V**

**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING**

in partial fulfilment of the requirements for the course of

**CSE2003 –DATA STRUCTURES AND ALGORITHMS**

**J-Component**

in

**B.Tech. (Computer Science Engineering)**

**Date: 1.11.2020**

**ABSTRACT:**

Most people prepare for exams throughout a major portion of their lifetimes and while doing so, time management is a key factor. While revising for any examination, one must prioritize their time and choose to spend it wisely. Also, most people begin their preparation at the eleventh hour and it is essential for them to know how to efficiently user their time. In our project we attempt to tackle this problem and provide a simplistic solution.

We have created a simple, yet highly efficient and functional system which would provide users a way to choose and prioritize their time spent while preparing for exams based on the weightage of topics and the correspondent time required to work on them. It is a system which incorporates several algorithms to solve the above problem.

The reason we chose to use an array of algorithms was to be able to compare them based on their speed and accuracy and tr to determine the best algorithm to be utilized for our system. Further, it is worth mentioning that our system is based on the principle of the knapsack problem.

**INTRODUCTION:**

As mentioned above our system is in several ways a practical application of the knapsack problem where one would focus on two aspects: weights and values, and analyze what the maximum profit that one could carry based on the total capacity of the bag/sack; the best example would be that of a thief having to choose items of high value and low weight to fit his sack while they carry out their robbery.

We manipulated this system to work for a different scenario where one would have to prioritize and schedule their time while they prepare for exams. In our case, weights would be time spent on each topic, values would be the weightage or score that could be gained from each topic and total capacity would be the total preparation time that the user still has left.

We have approached this problem using four different algorithms: brute force algorithm, dynamic programming and greedy algorithm where we have tried two different approaches, one where we maximize (value) weightage first and another where we maximize score per unit hour first (value per unit). This was done in an attempt to be able to analyze the ways in which various algorithms work on the same scenario with a variety of inputs and compare them based on the time they take to execute the program and the accuracy/helpfulness of the responses they each provide us.

**LITERATURE REVIEW:**

**Knapsack problem: A case study of garden city radio (GCR)**

By: **Kumasi, Ghana Peasah O. K. 1 , Amponsah S. K. 2 and Asamoah D.**

Accepted on 5 October, 2010 and published in African Journal of Mathematics and Computer Science Research Vol. 4(4), pp. 170 -176, April 2011.

This paper deals with single knapsack problems, where one container must be filled with an optimal subset of items, and a capacity is mentioned. It also points out that the mentioned profits, weights and capacities are positive integers.

This paper solely focuses on a given situation where an FM station wants to broadcast as many valuable adverts as possible without exceeding the total time limit given. which adverts should be broadcast to obtain optimal income under the constraints of c seconds? The solution to this problem is given using 0-1 knapsack problem thus making it an application of it.This paper mentions the branch and bound methods and also specifies that the perforamance of both these classes largely depends on the size of the problems that need to be solved

Final conclusion given by this paper includes, a description of pile of adverts at GCR as a clear case of knapsack problem. It states that the adverts are chosen from piles based on their values.

For future works, they mention that there are many more efficient and effective ways of selecting these adverts that can be used in place of piling them, as this is time consuming and reduces the number of adverts that can be played in a period of time.

2017 International Conference on Computer Communication and Informatics (ICCCI -2017), Jan. 05 – 07, 2017, Coimbatore, INDIA

A study Report on Solving 0-1 Knapsack Problem with

Imprecise Data

**A study Report on Solving 0-1 Knapsack Problem with Imprecise Data**

By: **Jayashree Padmanabhan, Swagath S Madras Institute of Technology, Anna University Chennai, Tamil Nadu**

Published in the 2017 International Conference on Computer Communication and Informatics (ICCCI -2017), Jan. 05 – 07, 2017, Coimbatore, INDIA;

The main motive of this research paper is to extend the classical Knapsack to real time situations like capital budgeting, network planning, etc. where data is not so accurate. One of the classic examples mentioned here is the situation of budgeting in a developing firm.

Some of the problems mentioned in this paper include possibility of only pseudo polynomial algorithms like dynamic Programming algorithms and its application to only to only small capacity or weight limits. There is a solution provided to the smaller capacity issue which is the usage of various

They have worked on solving 0-1knapsack with imprecise weights and profits by including usage of trapezoidal functions, fuzzy models, linear problem formulation and application og genetic algorithm techniques and added correction factors.

The major limitations of the project are that it cannot be extended to higher weight limits and solving unbounded item is infeasible.

The final conclusion of this paper, which includes the working on imprecise data by adopting fuzzy theory and genetic representation of data, is that genetic shows better performance when larger inputs are used.

The future works include addition of ‘t’ values within the genetic information and modifying mutation functions with weighted probability based on their relative weight and profit values.

**Load Balancing Solution for Heterogeneous Wireless Networks based on the Knapsack Problem**

By: Zsuzsanna Ilona Kiss, Andrei Ciprian Hosu, Mihaly Varga and Zsolt Alfred Polgar

This paper considers load balancing scenario over a heterogeneous wireless networking environment is presented. The goal is to find an optimal distribution of the data flows on the available tunnels, in the above-mentioned conditions, such that the transmission capacity offered by the heterogeneous network is used efficiently and the user satisfaction is increased. The load balancing operation is performed by the access point which interacts closely with the gateway. In a real scenario the AP could represent a mobile device installed in a public transportation vehicle which offers Internet connectivity to the passengers.

They have done a comparative study on Brute force approach, dynamic approach and greedy approach.

Brute force approach: If there are n items to choose from, there will be 2n possible combinations of the items in the knapsack. The disadvantage of this method is the computational complexity, which is O(n2n). As it can be seen the complexity of this algorithm increases exponentially, therefor it can be applied only to small knapsack problems, including only several items.

Dynamic approach: The basic idea of dynamic programming is to compute the solutions of sub-problems once, store these solutions in a table and use them to compute the global solution of the optimization problem. The third step is the bottom-up computation of V [i, j]. The algorithm returns which flows are assigned to the tunnel and the total rate and utility of the assigned flows.

Greedy approach: Complexity of the greedy algorithms is lower than that of other algorithms and it is O(nlogn), where n is the number of items in the knapsack. Several greedy strategies can be defined including: GR,GRLW and GRBV.

**Solve Zero-One Knapsack Problem by Greedy Genetic Algorithm**

By: **Yuxiang Shao, Hongwen Xu and Weiming Yin**

The main aim of this paper is to find a combination of items whose weight does not exceed the knapsack’s capacity and to maximize the overall profit.

They have used a basic idea of making it an iterative process by improving selection then crossover mutation and then ensuring all restrictions are maintained.

Paper conclusions refers to the basis of improving the selection operator of genetic algorithm, integrating genetic algorithm with greedy algorithm and presenting a greedy genetic algorithm enhancing the searching ability of genetic algorithm and the speed.

**Analysis of 0/1 Knapsack Problem using Deterministic And Probabilistic Techniques**

By: **Ritika Mahajan and Sarvesh Chopra**

Published in the 2012 Second International Conference on Advanced Computing & Communication Technologies

This paper deals mainly with the analysis of different algorithmic approaches to deal the basic knapsack problem. The comparison is made between Dynamic programming, Greedy methods, Genetic algorithm where they have mentioned operators such as Crossover, Mutation, Fitness Measure and Selection Method.

Conclusion of the papers states that the Genetic algorithm is one of the most powerful approaches for the knapsack problem as it is accurate and quick compared to the other existing algorithms mentioned in the analysis.

International Journal of Advanced Engineering and Management Research

International Journal of Advanced Engineering and Management Research

**0/1 KNAPSACK PROBLEM: GREEDY VS. DYNAMIC-PROGRAMMING**

By: **Ali Al Etawi , Fatima Thaher Aburomman**

Published in International Journal of Advanced Engineering and Management Research

Here, the paper deals with an analysis of Greedy and Dynamic programming approaches to solve the 0/1 Knapsack problem. The results state that, when items chosen are compared: the difference is insignificant, time taken is compared: Dynamic programming is slower than Greedy algorithm.

Conclusions include that both these algorithms have certain advantages and disadvantages in various sectors.

**Computational Burden Analysis for Integer Knapsack Problems Solved with Dynamic** By: Dariusz Horla

Published in ICINCO 2017 - 14th International Conference on Informatics in Control, Automation and Robotics

This paper especially focuses on 1D knapsack problems solved using Dynamic and Greedy algorithms. Different comparisons and analysis were made based on: computational time and the number of items that can be handled.

This paper aimed at giving the answers to problems with applicable dimensions, related to solutions obtained in matter of seconds (if not – fractions of a second) on a standard PC and Matlab.

**DYNAMIC PROGRAMMING REVISITED: IMPROVING KNAPSACK ALGORITHMS**

By: **U. Pferschy, Graz**

Received on October 15, 1998 and revised on March 10, 1999

The contribution of this paper is twofold: At ﬁrst an improved dynamic programming algorithm for the bounded knapsack problem is given. It decreases the running time for an instance with n items and capacity c from O(n c log c) to O(nc),which is the same pseudopolynomial complexityas usually given for the 0–1 knapsack problem. In the second part a general approach based on dynamic programming is presented to reduce the storage requirements for combinatorial optimization problems where it is computationally more expensive to compute the explicit solution structure than the optimal solution value. Among other applications of this scheme it is shown that the 0–1 knapsack problem as well as

the bounded knapsack problem can be solved in O(nc) time and O(n +c) space.

This entirely focuses on various domains including Correctness, Running time and Space Requirements. It also mentions reduction schemes.

**METHODOLOGY:**

The algorithms used in our work are basic algorithms that are generally used for small datasets. We also need to acknowledge that are no known works to us that were created on the topic similar to, clearly as seen above. However, we do notice that these algorithms are fairly still in use and are applied in a variety of scenarios. Furthermore, the knapsack problem is one of the simplest practical application of a variety of data structures (in our case being multiple one-dimensional arrays). It is important to note that the knapsack can be solved using a variety of approaches, a few of which have been explored by us in our pursuit of creating a system that would make exam-preparation easier by optimizing one’s time spent on the basis of the weightage of topics, estimated time taken to prepare for each topic and approximate time available for the reason of preparation. Now to look into the methodologies/algorithms used:

**BRUTE FORCE ALGORITHM:** The brute force algorithm is one of the most straightforward approaches to solve any problem which relies solely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency. This algorithm is known for its wide applicability and simplicity in solving complex problems. A method of problem solving in which every possibility is examined and the best one (or a best one) is chosen. It is often implemented by computers, but it cannot be used to solve complex problems such as the travelling salesman problem or the game of chess, because the number of alternatives is too large for any computer to handle. However, since the knapsack problem does not have a huge set of combinations on a reasonable sized dataset it can be used to solve the knapsack problem and our application of it for optimizing exam preparation.

**DYNAMIC PROGRAMMING:** Dynamic Programming is a technique generally used for solving optimization problems essentially by breaking them down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems and accessing previously stored solutions as one progresses. Dynamic Programming is mainly an optimization over plain recursion, which have repeated calls for same inputs. This simple optimization reduces time complexities from exponential to polynomial. Dynamic Programming is generally used if the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, we can observe some over-lapping subproblems (unlike divide-and-conquer problem where one divides the problem into non-overlapping subproblems).

**GREEDY ALGORITHM:** Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So, the problems where choosing locally optimal also leads to global solution are best fit for Greedy. Generally, to solve a problem based on the greedy approach, there are two stages: scanning the list of items and optimization. The conditions define the greedy paradigm are that each stepwise solution must structure a problem towards its best-accepted solution and that it is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

And finally, all these approaches are utilized to solve our application of the knapsack problem.

**KNAPSACK PROBLEM:** The problem can be defined as a scenario where given two integer arrays to represent weights and profits of ’N’ items, we need to find a subset of these items that will give us maximum profit such that their cumulative weight is not more than a given number ‘C’. Each item can only be selected once, so either you put an item in the knapsack or not. Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials, selection of [investments](https://en.wikipedia.org/wiki/Investment) and [portfolios](https://en.wikipedia.org/wiki/Portfolio_(finance)), selection of assets for [asset-backed securitization](https://en.wikipedia.org/wiki/Securitization), and generating keys for the [Merkle–Hellman](https://en.wikipedia.org/wiki/Merkle%E2%80%93Hellman_knapsack_cryptosystem) and other [knapsack cryptosystems](https://en.wikipedia.org/wiki/Knapsack_cryptosystems).

**ALGORITHMS AND TECHNIQUES (PROPOSED SOLUTION):**

**ALGORITHMS:**

**BRUTE FORCE ALGORITHM:** If there are n items to choose from, then there will be 2n possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0’s and 1’s is generated which is of length n. If the ith symbol of a bit string is 0, then the ith item is not chosen and if it is 1, the ith item is chosen.

Algorithm for solving knapsack problem using brute force algorithm:

for i = 1 to 2n do

j <- n

tempWeight <- 0

tempValue <- 0

while (A[j]!= 0 and j > 0)

A[j] <- 0

j < - j – 1

A[j] < - 1

for k < - 1 to n do

if (A[k] = 1) then

tempWeight < - tempWeight + Weights[k]

tempValue < - tempValue + Values[k]

if ((tempValue > bestValue) AND (tempWeight < - Capacity)) then

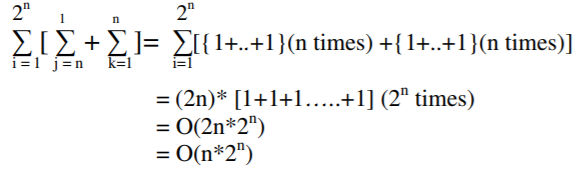
bestValue < - tempValue

bestWeight < - tempWeight

bestChoice < - A

return bestChoice

Time complexity analysis:



**DYNAMIC PROGRAMMING:** In dynamic programming, in a table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.   
The state DP[i][j] denotes the maximum value of ‘j-weight’ considering all values from ‘1 to ith ’. So, if we consider ‘wi’ we can fill it in all columns which have ‘weight values > wi’. Considering two possibilities which can take place: fill ‘w \i’ in the given column or do not fill ‘wi’ in the given column. Taking the maximum of these two possibilities, if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. Hence, we take the maximum of these two possibilities to fill the current state.

Algorithm for solving knapsack problem using dynamic programming:

int i, w;

int K[n + 1][W + 1];

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

{

for (w = 0; w <= W; w++)

{

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max((val[i - 1] + K[i - 1][w - wt[i -1]],(K[i [w]));

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

Time complexity analysis:

Let ‘N’ be the number of weight element and ‘W’ be the capacity. For every weight element we traverse through all weight capacities 1<=w<=W. Thus, the time complexity is **O(N\*W)**. Further the use of 2-D array of size ‘N\*W’ implies the space complexity is also O(N\*W).

**GREEDY ALGORITHM:** In our case, items can be broken into smaller pieces, hence the we can select fractions of items. According to the problem statement, there are n items in the store, weight of ith item is wi > 0, profit for ith item is pi > 0 and capacity of the Knapsack is W.

In this version of the knapsack problem, items can be broken into smaller pieces. So, the we may take only a fraction xi of ith item, implying 0 <= xi <= 1. The ith item contributes the weight xi \* wi to the total weight in the knapsack and profit xi \* pi to the total profit. Hence, the objective of this algorithm is to maximize sum of xi \* pi from 1 to n, subject to constraint that the sum of the product remains lesser than or equal to W.

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit. Thus, an optimal solution can be obtained by equating sum of xi \* pi from 1 to n to W. In this context, first we need to sort those items according to the value of pi/wi, so that ((pi+1)/(wi+1)) <= pi/wi. Here, x is used an array to store the fraction of items.

Algorithm for solving knapsack problem using greedy algorithm:

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

weight = W

break

return x

Time complexity analysis:

The provided items are already sorted into a decreasing order of pi/wi. This implies that the time taken by the while loop is O(n). Therefore, the total time including the sort is in **O(n\*logn)**.

**OUR APPROACH:**

Our problem is a modified version of [0/1 Knapsack](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/) where we have to either consider taking an item or ignoring it while placing a set of items in a bag. What changes in this question is the constraint conditions that we are given the time a particular topic takes and maximum time left for the exams.

Proceeding in the same way as solving a regular 0/1 Knapsack problem we consider reading a topic if it can be read in the given leftover time for the exam otherwise ignore that topic and move to next topic. This way we will calculate the maximum weightage marks sum that a person can score in the given time frame.

Flow of implementation:

* Read one topic and time required to cover it.
* If the leftover time is less than the time needed to cover theith topic then ignore that topic and move forward.
* Now two cases arise (We have to find the maximum of the two)
  1. Consider reading that topic.
  2. Ignore that topic.
* Now to find the maximum marks that can be achieved we have to backtrack our path of topics considered for reading. We will loop from bottom left of matrix to start and keep adding topic weight if we have considered it in the matrix.
* Now T[no\_of\_topics-1][total\_time-1] will contain the final marks.
* If the final marks are less than the passing marks then print **-1** else print the calculated marks.

Clearly the above is a general flow of implementation, however obviously it can change from one algorithm to another as we proceed through the system.

Base condition**:** When time is 0 or number of topics is 0 then the calculated marks will also be 0.

**ARCHITECTURE:**

Our system has 6 functions:

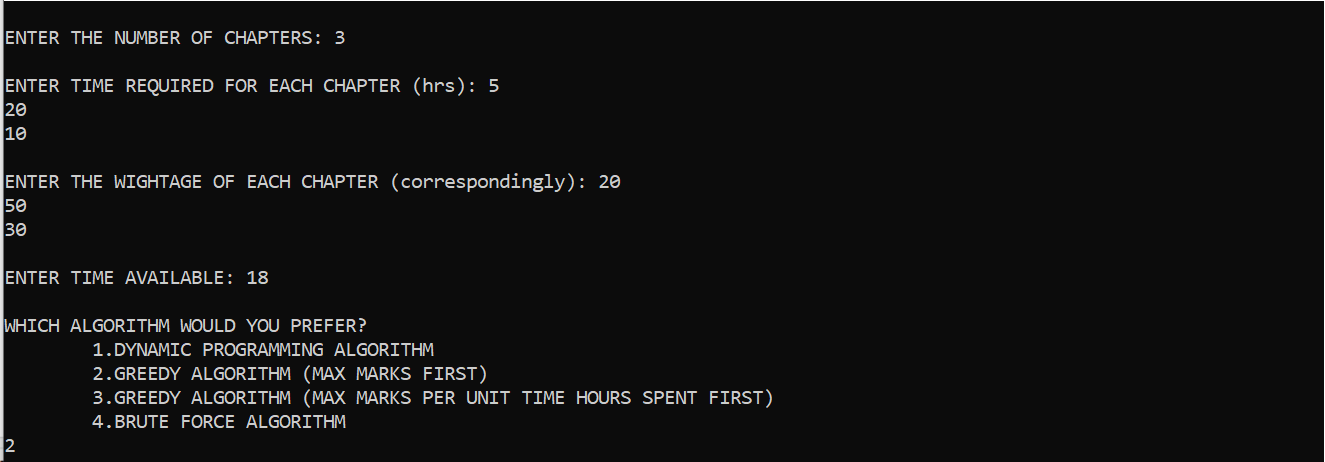
1. **bruteforce()**: This function has 4 arguments and they are the array of time taken for each topic as an integral value of weight[], array of weightage of each topic correspondingly as integral value of profit[], total time available as integral value of capacity and total number of topics as integral value of size and returns an integral value result which is the total score that can be expected found using the brute force algorithm.
2. **dynamicKnapsack()**: This function has 4 arguments and they are the array of time taken for each topic as an integral value of weight[], array of weightage of each topic correspondingly as integral value of profit[], total time available as integral value of capacity and total number of topics as integral value of size and returns an integral value result which is the total score that can be expected found using dynamic programming.
3. **greedy1()**: This function has 4 arguments and they are the array of time taken for each topic as an integral value of weight[], array of weightage of each topic correspondingly as integral value of profit[], total time available as integral value of capacity and total number of topics as integral value of size and returns an integral value result which is the total score that can be expected found using greedy algorithm which maximizes time taken for each topic(weight) first.
4. **greedy2()**: This function has 4 arguments and they are the array of time taken for each topic as an integral value of weight[], array of weightage of each topic correspondingly as integral value of profit[], total time available as integral value of capacity and total number of topics as integral value of size and returns an integral value result which is the total score that can be expected found using greedy algorithm which maximizes the weightage of each chapter(profit) per unit time taken for each topic (weight) first.
5. **printResult()**: This function is used to print the outputs including selected chapters, maximum marks that can be scored and minimum time spent. It has one argument which is the integral value of total profit which is the total mark that can be expected calculated using one of the above functions/algorithms. Since it is used only for the purpose of printing the outputs it returns 0.
6. **main()**: This is the main function of our system which essentially is used to get the inputs from the users (the inputs are customizable by each user every single time) and the inputs include total number of topics, the time taken to cover each topic in hours and weightage of each topic corresspondingly in that order. This would then lead the user to a menu driven program where the user will have to choose the algorithm they wish to use to solve their given problem, based on which the function corresponding to the said algorithm would be called and the printResult function would be called to diplay the results calculated using the chosen algorithm. Further, a clock function is also used to calculate the time taken to execute the particular function which would be displayed in the end of the main function. It returns a value of 0.

**INPUT:**

As mentioned earlier, this is a system with no fixed inputs and can be manipulated/should be provided by the user. These are entirely customizable inputs to fit the needs of each different user in each new scenario. However, the inputs to be provided are (in order):

1. Number of chapters/topics
2. Time required to cover each topic (in hours)
3. Weightage of each chapter (correspondingly)
4. Time available (in hours)
5. The algorithm to be used (from the given options)

An example of the above would be:

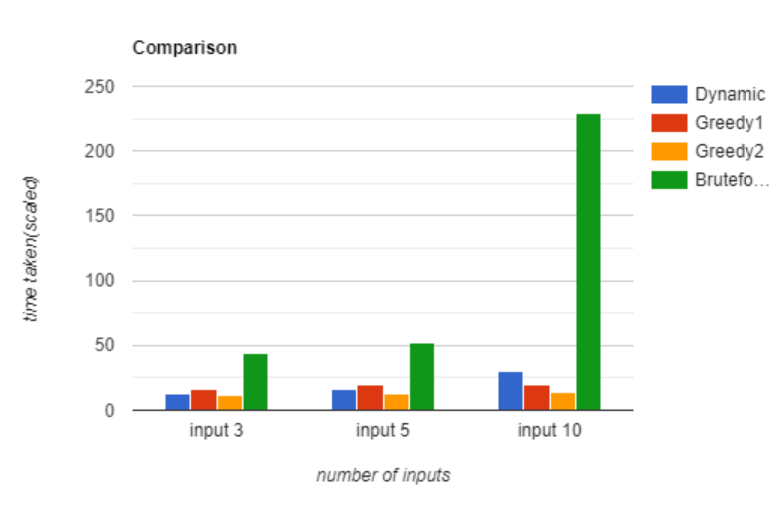


The above is an example when the user has to prepare for 3 topics, which will fetch them 20, 50 and 30 marks, take approximately 5, 20 and 10 hours to prepare respectively, has 18 hours left to prepare and chose to prepare using Greedy algorithm.

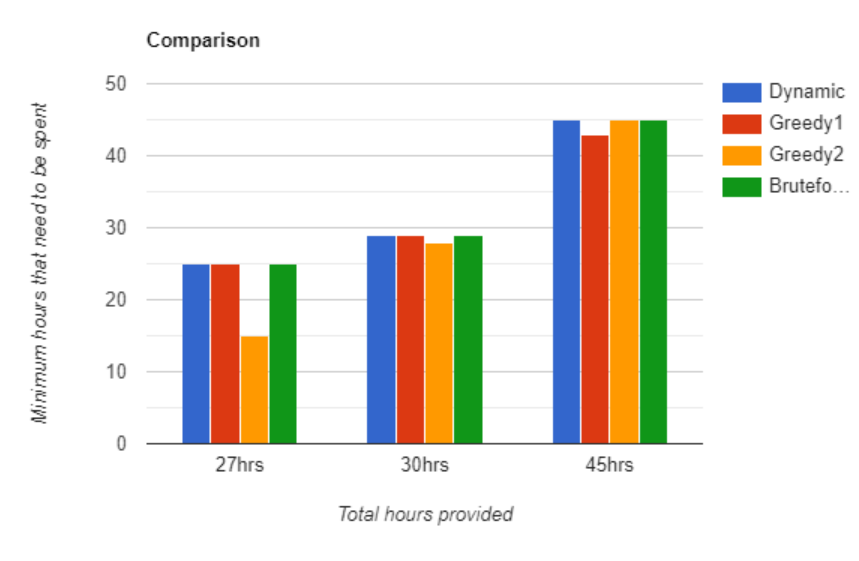
We also must note that since this is a system that is made for customizable inputs and doesn’t work on the basis of any data other than the data currently produced by the user, this system has no database.

**RESULTS AND DISCUSSION:**

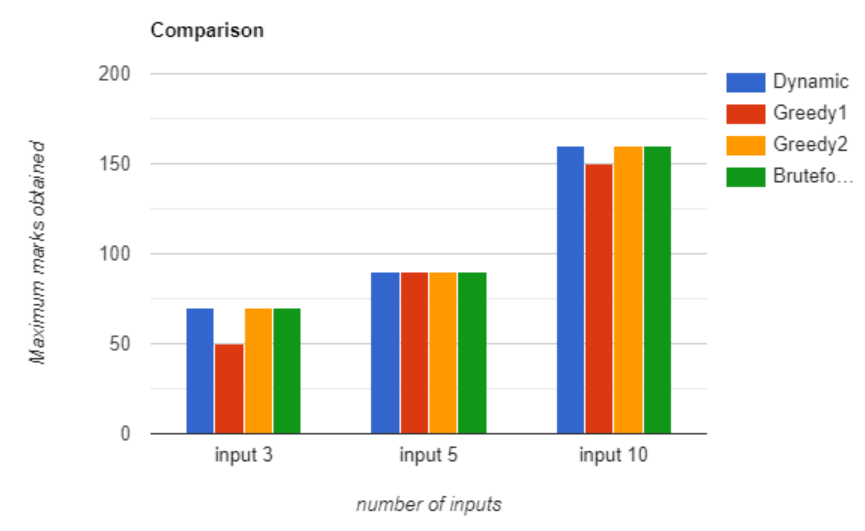
Other than the obvious implementation of our system to optimize exam preparation for all users, we also tried to compare a variety of algorithms on the basis of their accuracy and the time taken by each of them to produce a result.



Based on the above graph, in each scenario, it is evident that brute force takes the longest amount of time to execute the function and Greedy2 (where we maximize score per unit time) takes the least amount of time. Dynamic Programming is a close contender in terms of speed for smaller inputs, but as the size of inputs increases, clearly Greedy2 is the fastest algorithm.

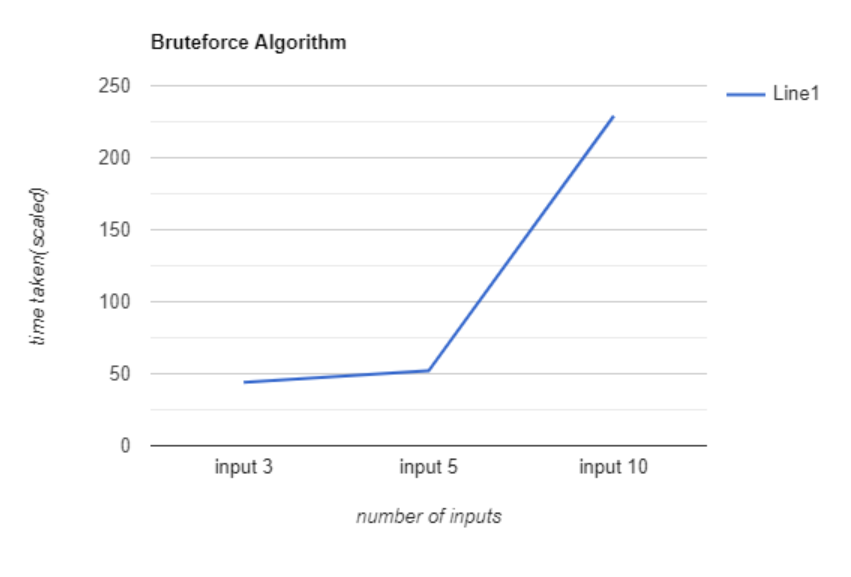


Comparing the various algorithms based on their output that they give for “minimum time to be spent”, we notice that the results are inconclusive and pretty close to each other for all the algorithms. Clearly, Dynamic Programming (DP) and brute force algorithm almost always gave the same output for this section regardless of the size of inputs. Both the greedy algorithms were largely variable in each scenario.

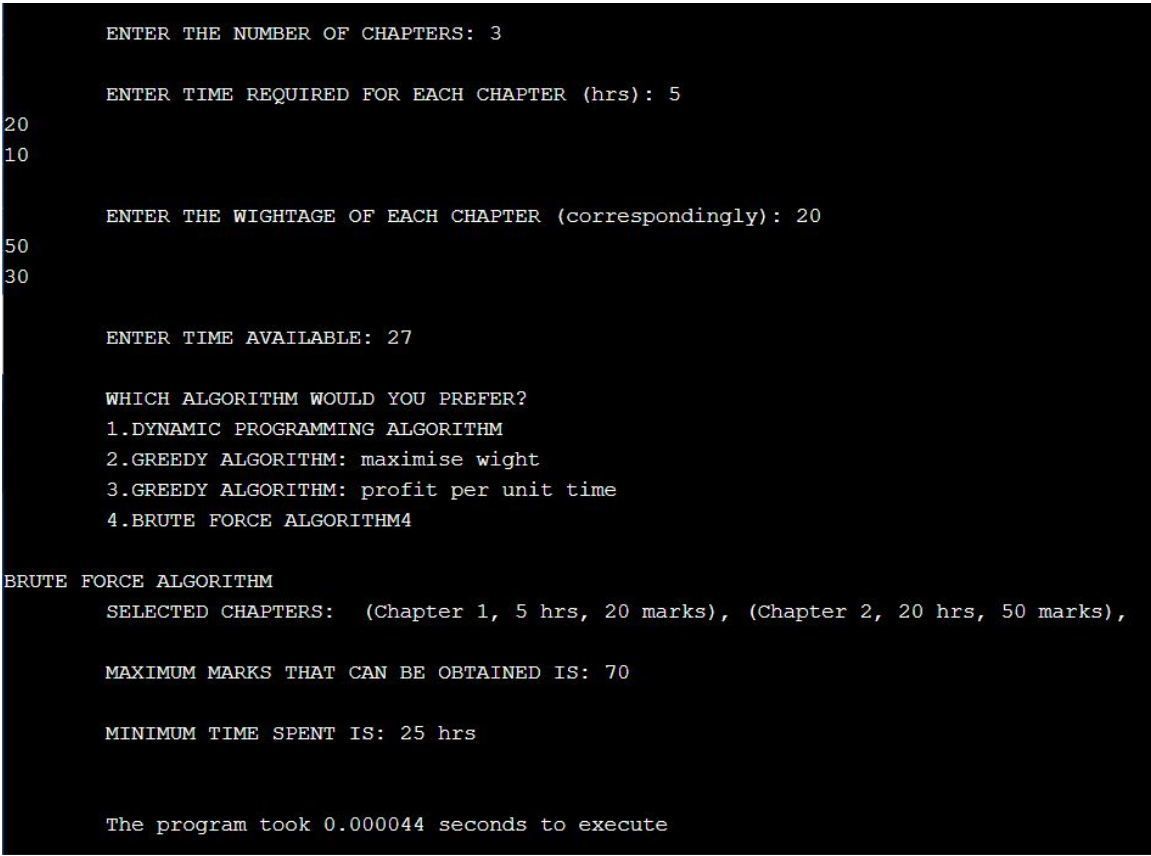


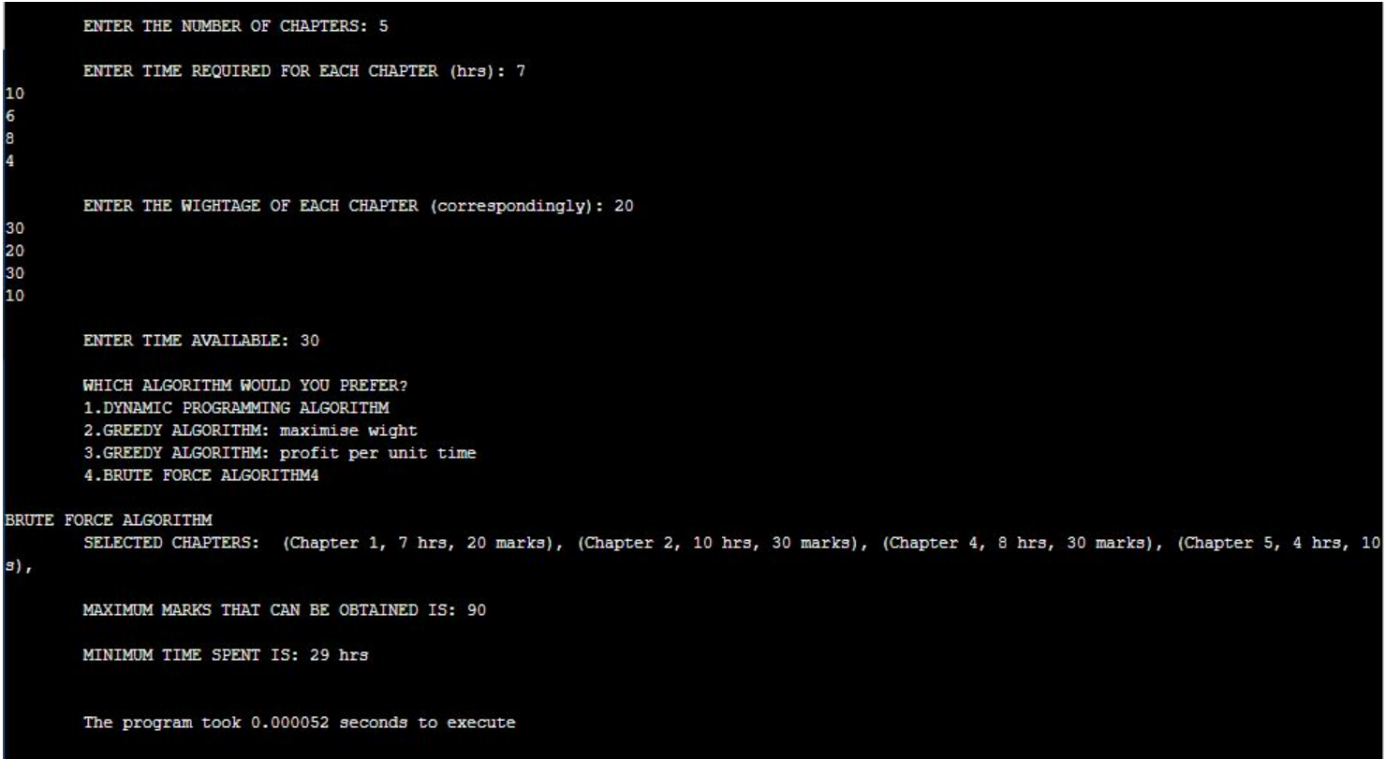
Again, we note that all the algorithms are pretty consistent with their outputs on the maximum marks that can be obtained by the user based on their inputs, with the only exception being Greedy1 approach which gives lower values of scores with certain inputs and might not be the best approach thus giving an edge to DP.

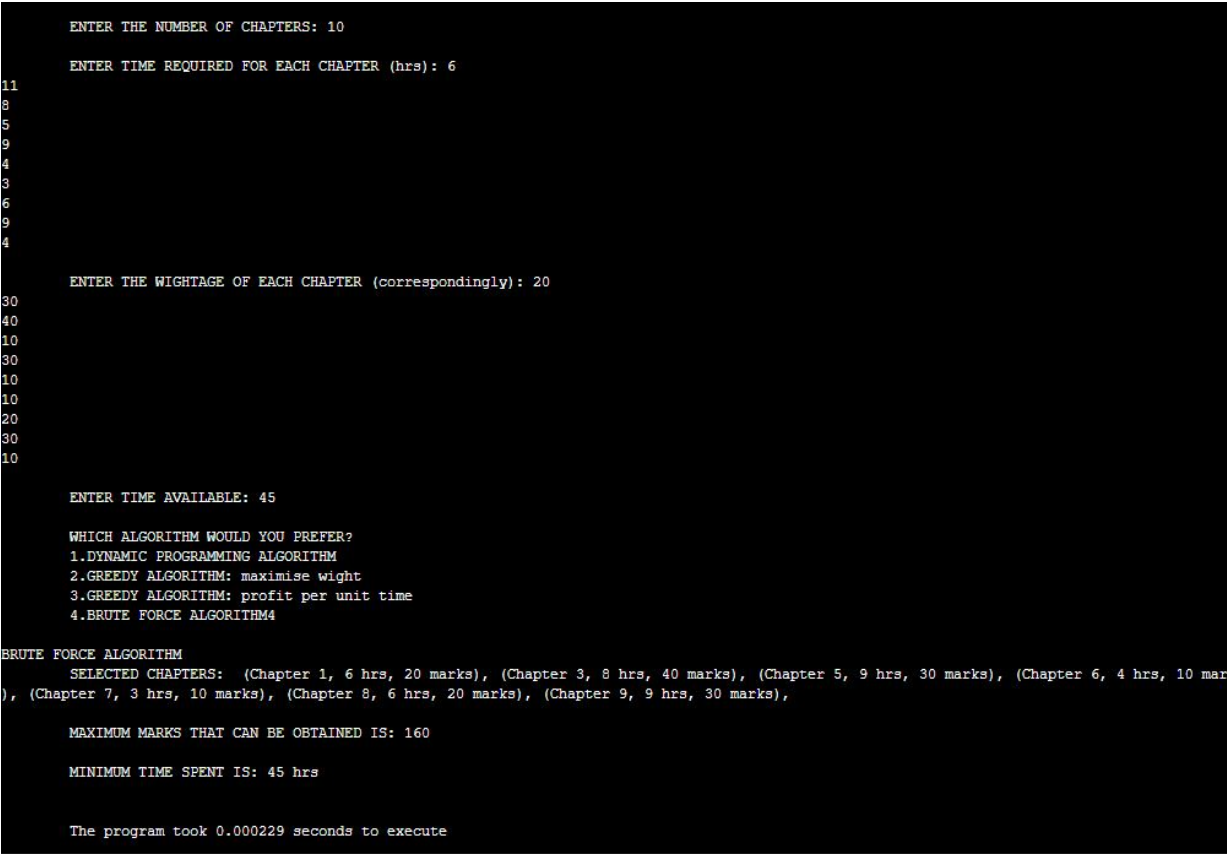
**BRUTE FORCE ALGORITHM:**



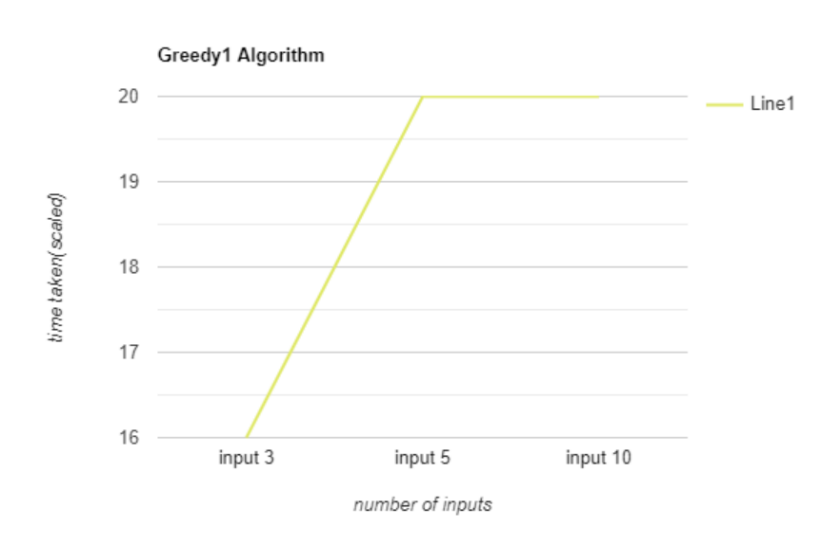
An obvious conclusion that anyone could make even before getting into the details is the least efficient algorithm was the brute force technique. This is because brute force is the simplest of techniques and approaches all methods to solve the problem, meaning it would take the longest possible time to arrive at a solution.





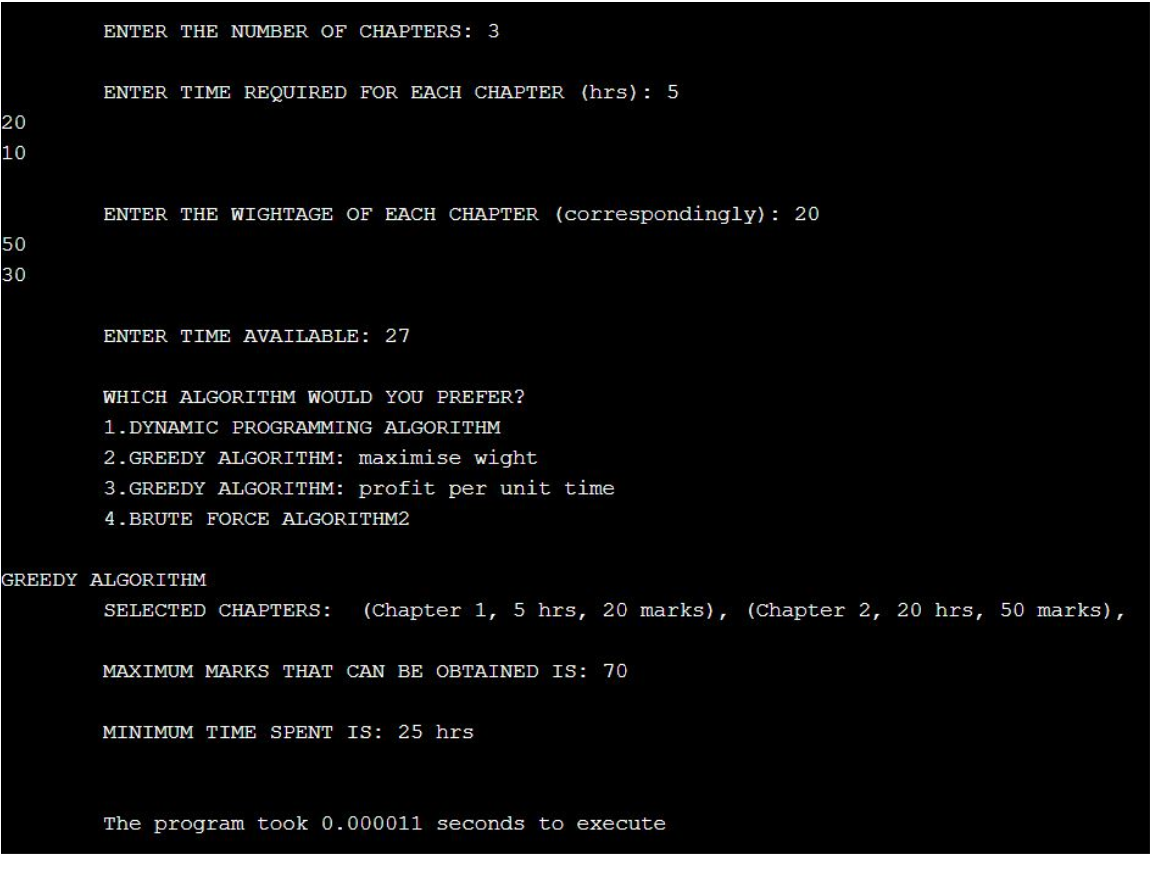


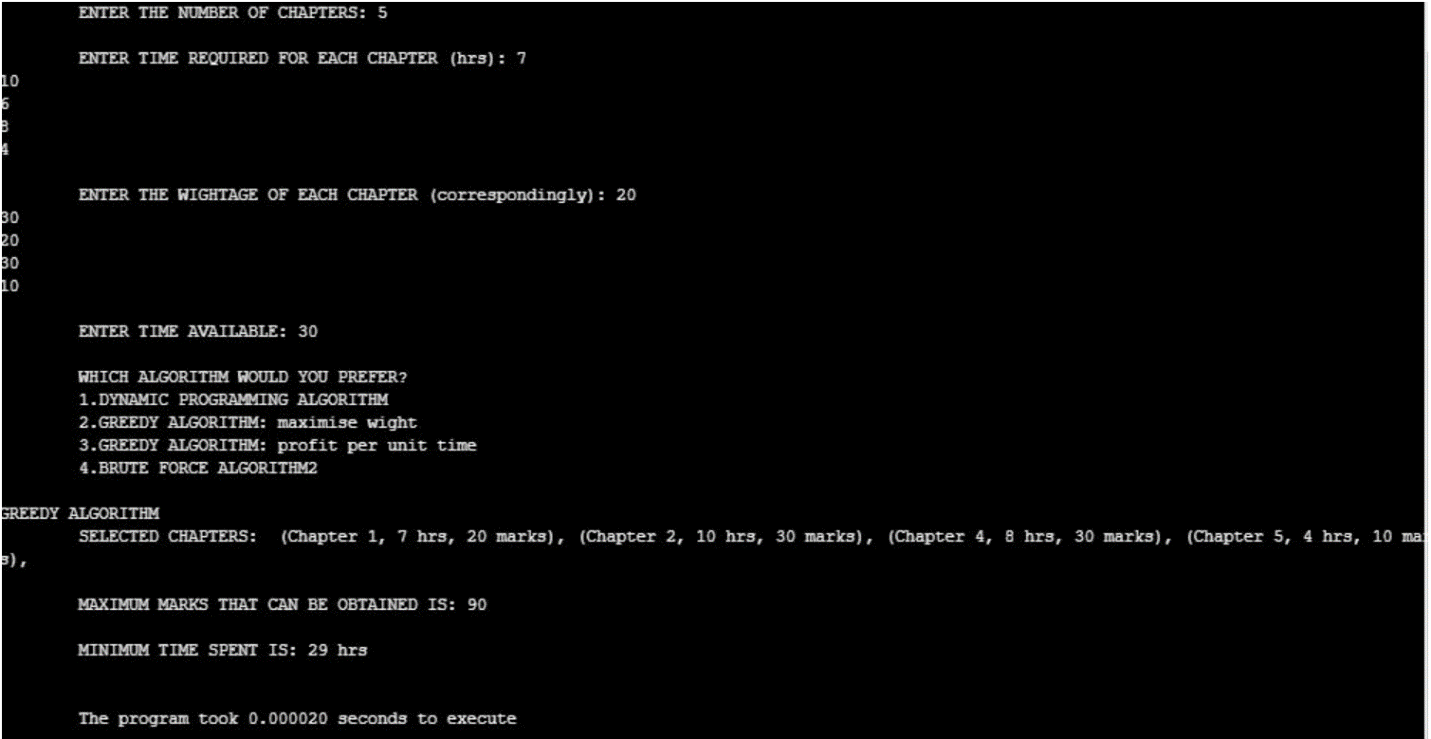
**GREEDY1 (MAXIMIZE SCORE FIRST):**

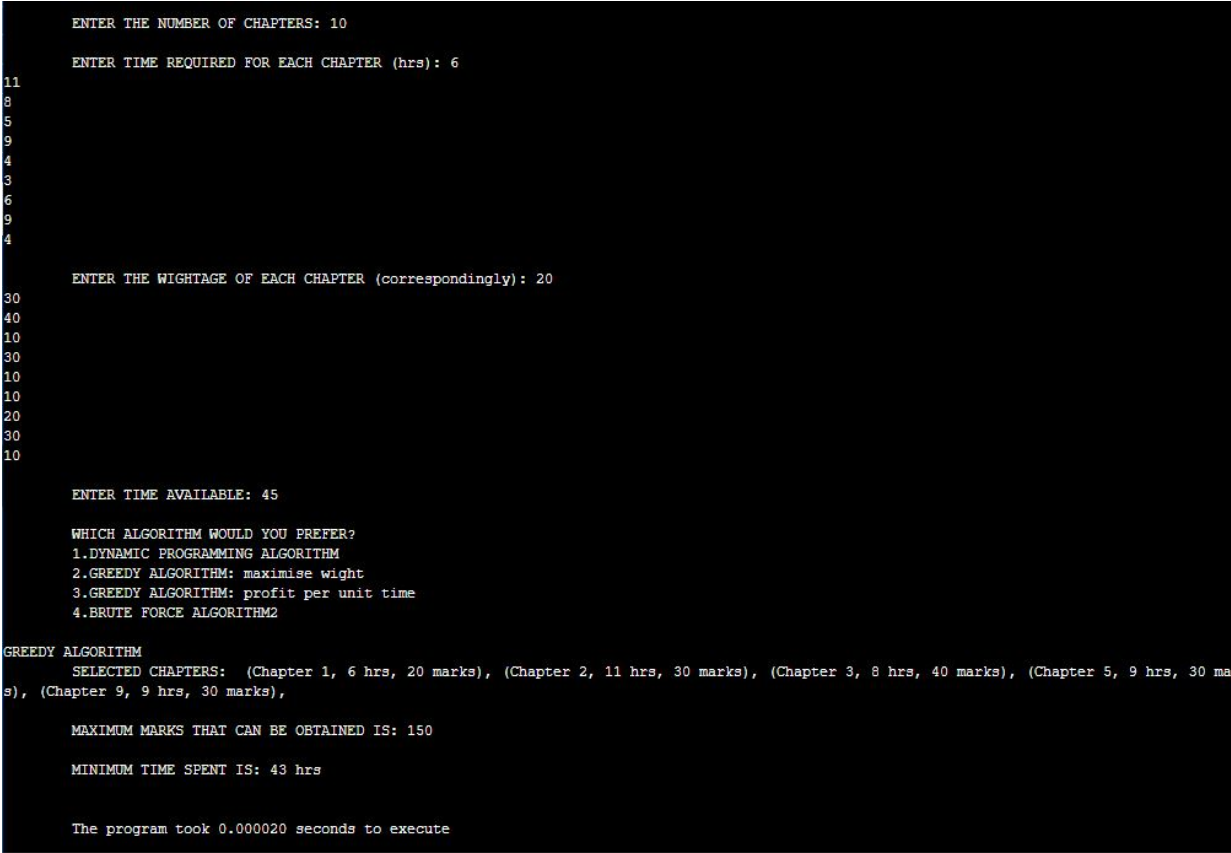


Another inference we made is that clearly our second approach of Greedy where we maximized profit per unit time first was evidently more efficient than the first approach where we maximized only the values first. Due to this very reason the second approach remains the most widely used method to solve the knapsack problem using the Greedy approach.

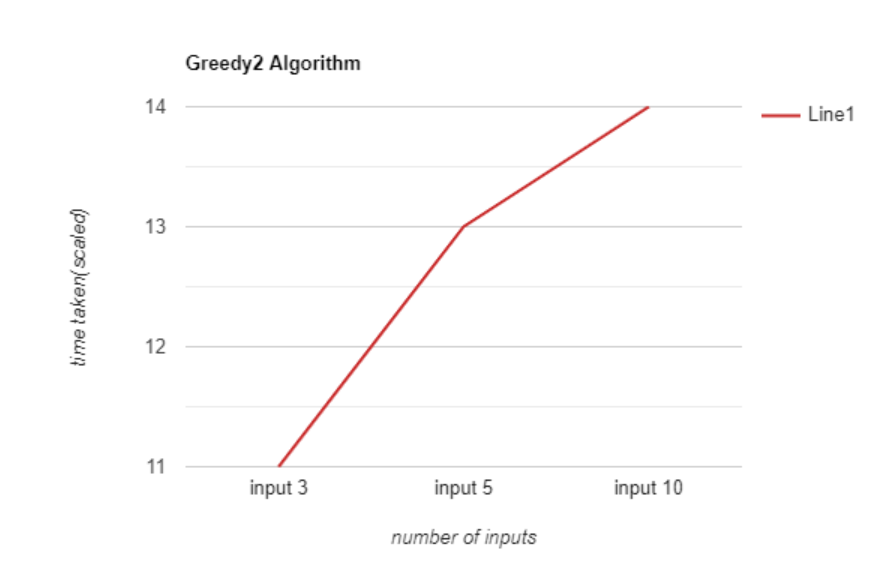
It is also worthy to mention that brute force algorithm and Greedy1 approach provided similar results for smaller inputs, but as we increase the number of inputs clearly, we notice that brute force algorithm gives a better analysis of chapters to study for higher scores. Below are the outputs given by Greedy approach where we maximized the values first for three different sets of inputs:



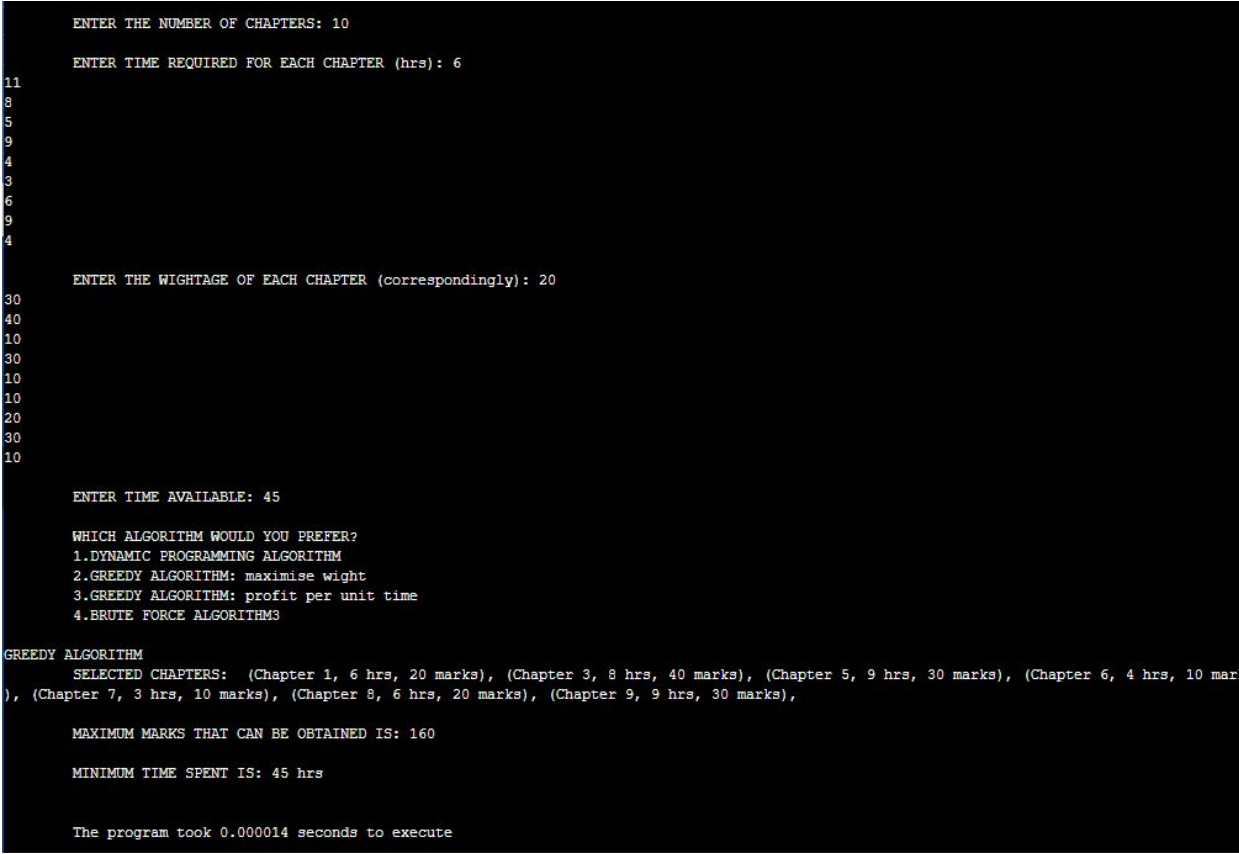
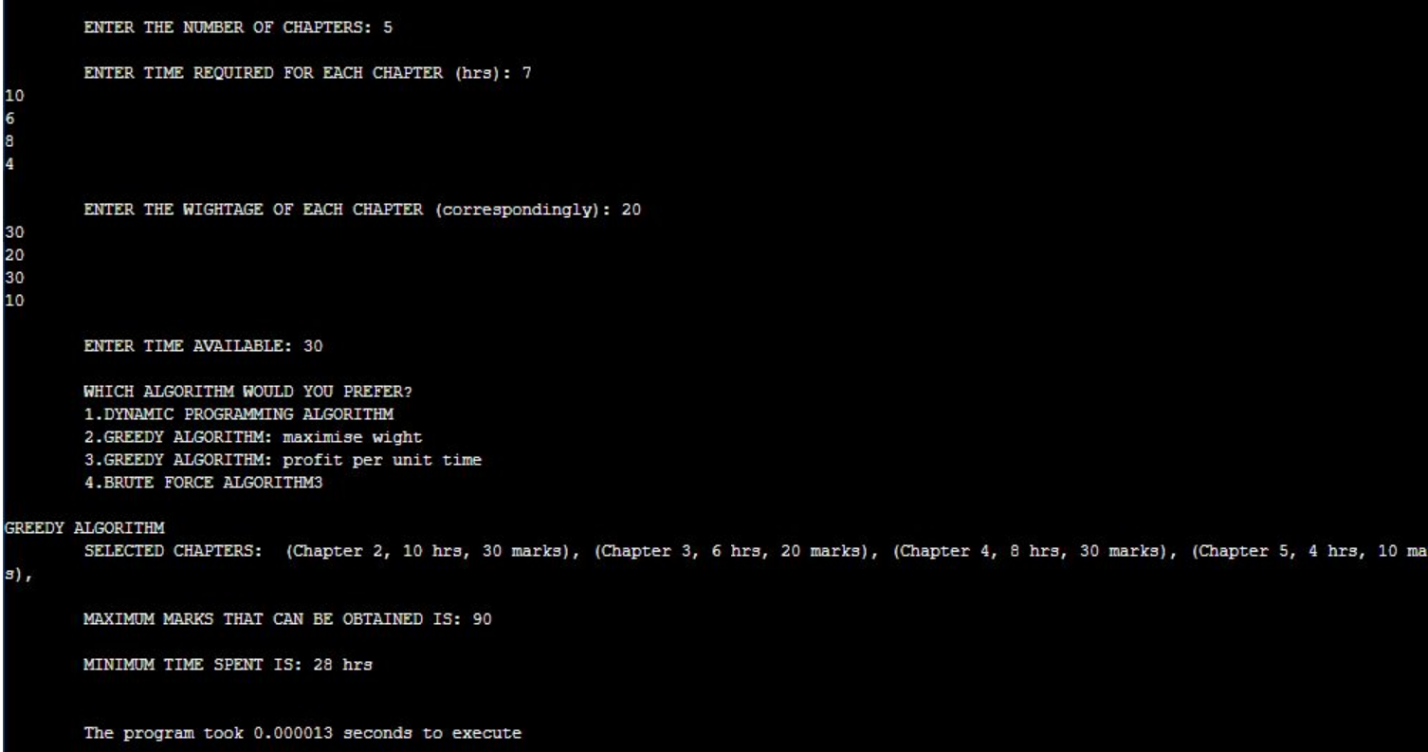
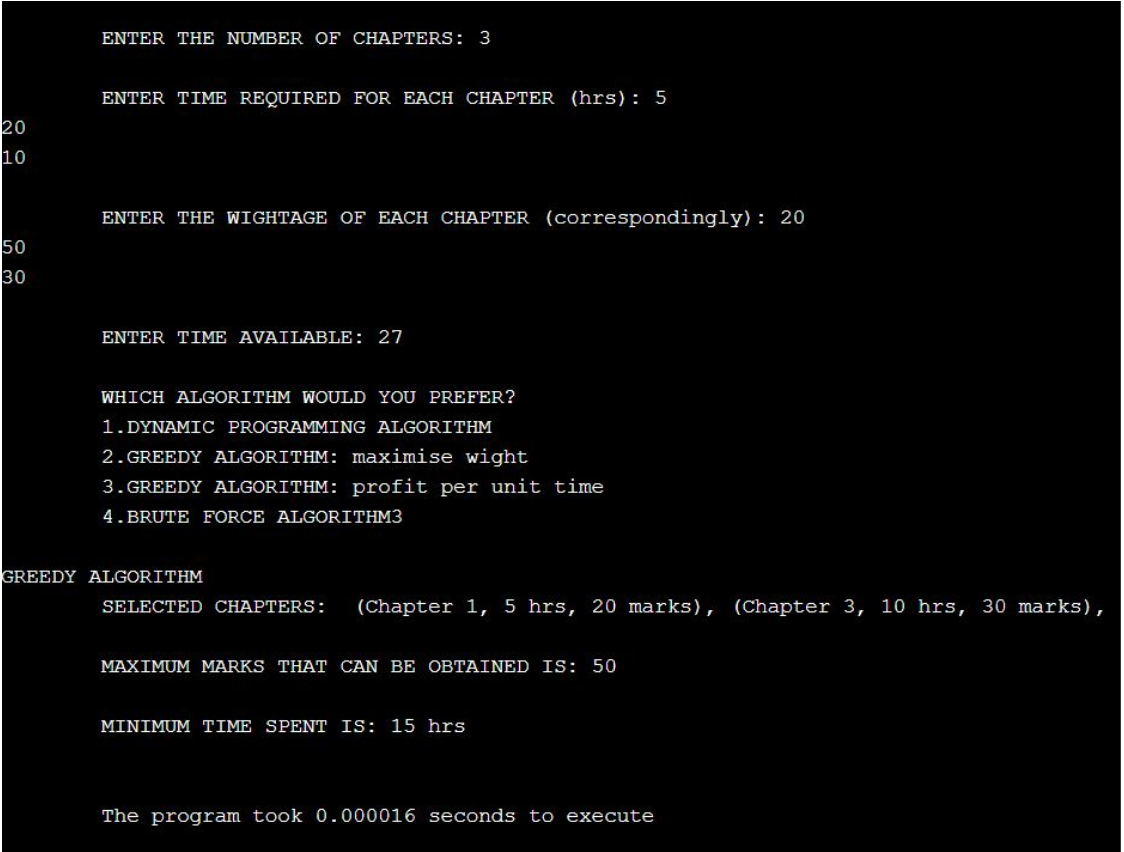




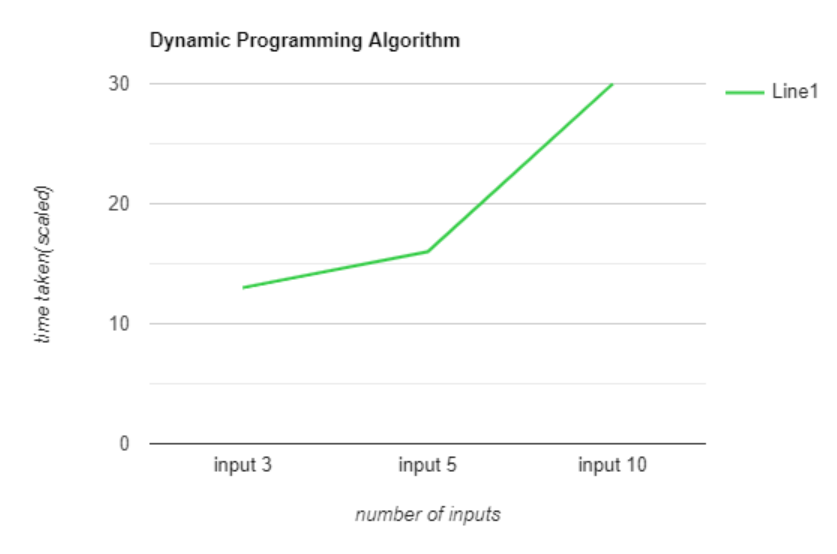
**GREEDY2 (MAXIMIZE SCORE PER UNIT TIME FIRST):**



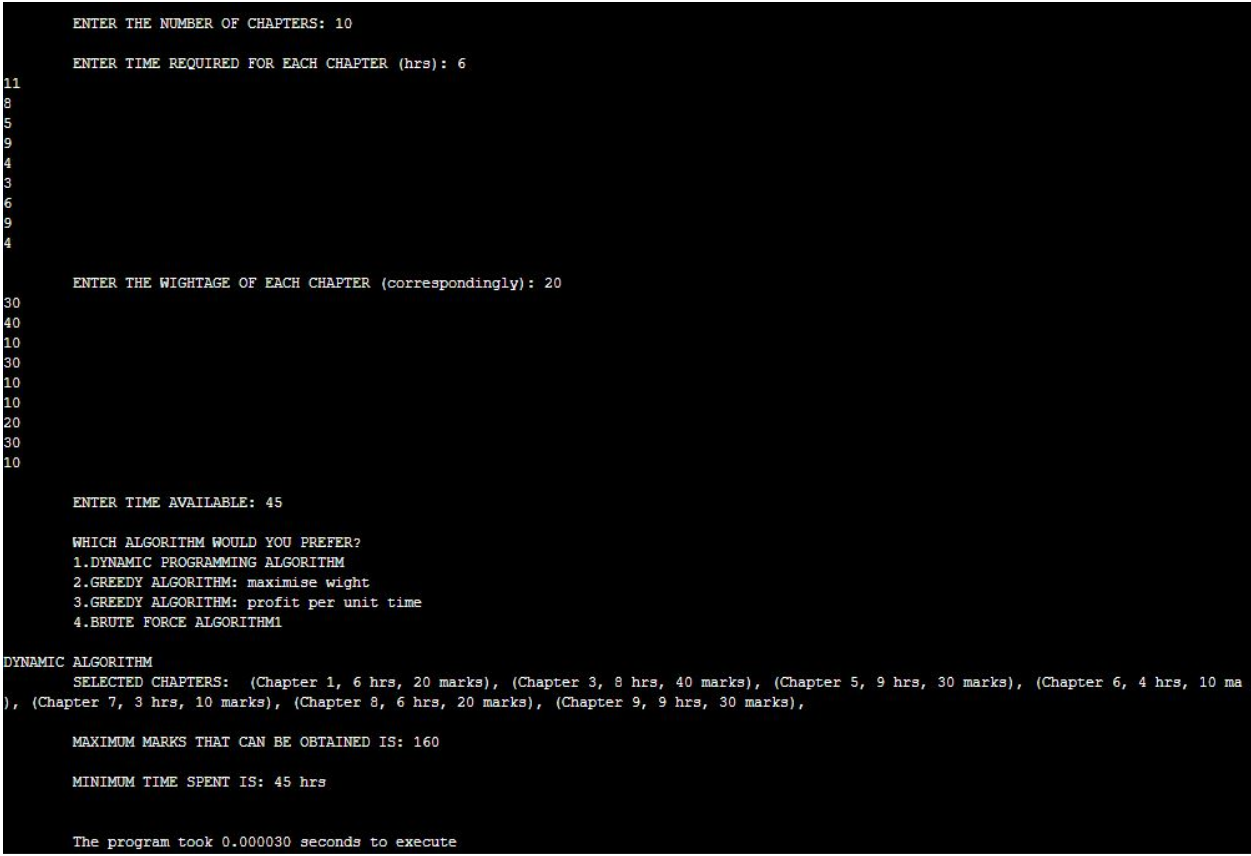
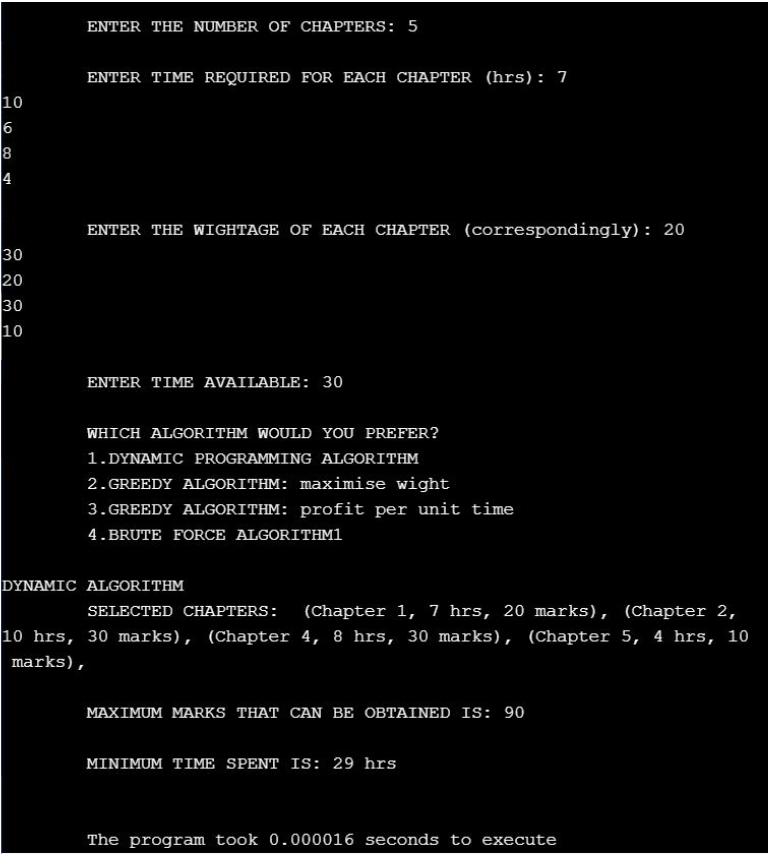
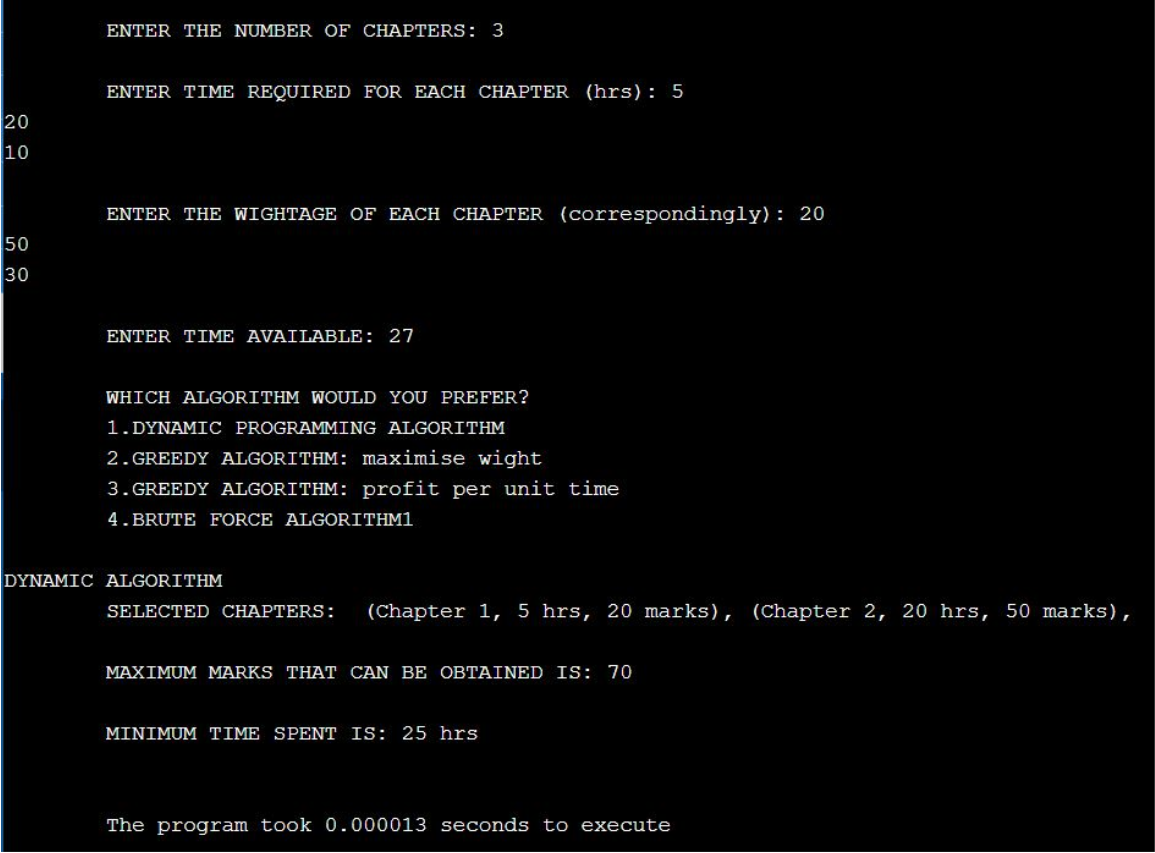
To compare between Greedy approach where we maximized profit per unit time first (to be mentioned as Greedy) and Dynamic Programming (DP), each method had its own advantages and disadvantages. Greedy has a lower runtime and memory requirement, but gives a lower value of score that can be obtained. Clearly DP takes longer to run and takes up more memory and hence is more expensive owing to the above-mentioned factors. However, DP is still more near to optimal for our needs than Greedy is. Greedy is based on locally optimal decisions made at every stage, what makes the method easy to implement, but the obtained solution might not be globally optimal. Below are the outputs given by Greedy (greedy2) approach:



**DYNAMIC PROGRAMMING:**



Below are the outputs given by Dynamic Programming:



**CONCLUSION:**

Our comparative analysis was based on parameters such as time taken for each algorithm to give the requested output, the variation in minimum hours required to complete the task and the chapters chosen that give us the maximum marks that can be obtained. All the mentioned algorithms give pretty accurate as well as similar results in terms of maximum marks that can be obtained. The major differences in the performance of these algorithms depends on the run time of these algorithms.

From various input sets, we have inferred that in terms of time taken, the most efficient is the Greedy2 algorithm which focuses on maximizing the profit per unit time, followed by Dynamic Algorithm, then the Greedy1 Algorithm which deals with maximizing the weights and the least efficient is the Brute force Algorithm. Furthermore, while comparing the maximum marks that can be obtained, statistically, Dynamic, Brute force and Greedy1 give similar results but Greedy2 algorithm gives a smaller value. This is because it focuses on profit per unit time. Thus, if compared fractionally, it still gives better results than the rest of the algorithms. Finally, in terms of maximum time spent, all the algorithms give similar results.

Thus, from all the above-mentioned observations, we conclude that Brute force algorithm is the least efficient algorithm. Greedy1 algorithm works well for smaller inputs but as the number of inputs increases, its efficiency decreases. Dynamic programming approach gives consistent results for different input formats and is thus more efficient than the previous algorithms. Greedy2 algorithm is the most efficient algorithm out of all the above-mentioned ones and thus is one of the most widely used method for solving the Knapsack problem and its various applications.

**REFERENCES:**

1. <https://academicjournals.org/journal/AJMCSR/article-full-text-pdf/84A238D39964>
2. <https://www.researchgate.net/publication/321259550_A_study_report_on_solving_0-1_knapsack_problem_with_imprecise_data/link/5c7cc353299bf1268d365b24/download>
3. <https://ieeexplore.ieee.org/document/5073116>
4. <https://ieeexplore.ieee.org/document/6168350>
5. <https://ieeexplore.ieee.org/document/7296381>
6. <https://ieeexplore.ieee.org/document/6168350>
7. <https://www.researchgate.net/publication/340647785_01_KNAPSACK_PROBLEM_GREEDY_VS_DYNAMIC-PROGRAMMING>
8. <https://www.scitepress.org/papers/2017/64153/64153.pdf>
9. <https://www.researchgate.net/publication/225180414_Dynamic_Programming_Revisited_Improving_Knapsack_Algorithms>
10. <https://iopscience.iop.org/article/10.1088/1742-6596/1069/1/012024>
11. <https://www.researchgate.net/publication/329390711_Dynamic_programming>
12. Thomas H Cormen,Charles E. Leiserson,Ronald L. Rivest and Cliﬀord Stein. Introduction to algo-rithms,third edition. pp 360-395
13. Longest common subsequence—Introduction LCS length,Techie delight,Coding made easy
14. MIT course on algorithms-Includes a video lecture on DP along with lecture notes
15. Dynamic programming and Longest common subsequence, Geeks for geeks,computer science portal for geeks
16. Richard bellman on the birth of dynamic programming,Stuart Dreyfus,university of California, Berkeley.
17. <https://www.codesdope.com/>
18. <https://www.geeksforgeeks.org/>
19. <https://www.researchgate.net/>
20. [www.javatpoint.com](http://www.javaypoint.com/)