# Programming Project I, Second Report

Illya Starikov, Claire Trebing, & Timothy Ott

Due Date: March 17, 2016

## 1 Abstract

As proposed in the first report, we would like to the compare the effeciency of different memory management methods via a 0-1 Knapsack. We discussed the current memory management methods, the different approaches we would like to use (Brute Force, Dynamic Programming, Greedy Solution) and our plan of implementations.

After implementing the pseudocode provided in Report #1, rigerous testing and extensive running of sample of data, we have completed conclusive evidence to our proposed problem.

In this report we would like to discuss our findings.

## 2 Implementation

As we discussed in our earlier report we decided to go with a more object-oriented approach to this problem. We created a class called `App` to represent each application running on our platform. Each instance of the `App` class contains (as properties[1]) an identifying number, its memory usage, and its cost to recover from storage. These properties are largely randomized in the construction of each instance. For example, every app has a random memory usage in the range of 32 to 1028; the cost is a derived value, taking anywhere from 20 to 50 percent of the memory usage of the `App` in question; we also have a property called `ratio` which consists of the cost property divided by the memory property and is calculated at creation of the instance by the constructor, this will be used in our Greedy Algorithm; and lastly each `App` is given an identifying number based on the index of the cell in the array in which they are initially placed. This ensures that each `App` will have a unique identifier to help differentiate them whilst checking our solutions against one another.

Initially we considered creating a separate class for our "Smartphone" or the platform on which our `Apps` would be running but later decided against this as this functionality could easily be represented simply by a list (or array) of all the `Apps`, cutting down on the amount of code we would need.

---

[1]Or member variables, depending on your preferred notation.

We implemented our three solutions as one overarching test program that produces arrays of apps in ever increasing input sizes and feeding those arrays to our algorithms one at a time, testing the average runtime over 10 iterations. The general flow of our program is as follows:

First we specify a list of input sizes that our program will act on, we will discuss this is in more detail further along but for now it should be noted that our first input size is 5. Our program then creates an array of newly created Apps of the specified input size[2]. These newly created apps have each of their properties randomly assigned by its constructor. Next, because our Greedy Solution requires the inputs be sorted based on its ratio variable and the other solutions do not, we sort the list using Pythons built in sort function.

Now we simply feed the newly sorted list to each of the solution functions in turn, timing the run time of each by taking a timestamp before running the algorithm, again after, then subtracting the two times. We run each program ten times per input and take the average of these times for greater accuracy. Once each solution algorithm has run ten times the program outputs the data (which includes the solution set, the amount of memory freed by the solution, the combined cost of the solution and the average time it took) to two text files in the directory of the program.

It is at this point that our program returns to the top of the loop for the next input. To ensure a adequate spread of input sizes we defined a list of base inputs which we initialized as $5, 10, 15,$ and $25$. Once the program finishes iterating over this list, testing each solution as mentioned above, the program returns to the beginning of the input list and multiplies each by 10 to the power of $n$, where $n$ is incremented with each new round of the loop back to the beginning of the base inputs. This entire process is placed in a while loop that is set to infinitely repeat so that we are able to get as many data sets as the program can output in the time we have allotted for testing.

## 3 Experiment

As described above, the tests were run for a twelve hour period getting our input size to $1,000$. Noting that the for input size of $1,000$ it took the Dynamic Programming approach 1334.81 seconds, the total time to get accurate results was *3 hours, 42 minutes, and 28 seconds*[3]. We thought this was a good cut off point.

To ensure accuracy, we decided to run the tests on two independent machines. These tests can be seen on the graphs, but not the data sets. The tests were fairly comparable, so we decided that displaying data from one data set was sufficient.

We noticed that the Brute Force solution was taking a drastically longer time to run (the complexity being $n \times 2^n$). After a sample run, the Brute Force algorithm returned a Memory Error for an input size of 50. Seeing as we were

---

[2]Again, in our first case this is 5.

[3]1334.81 seconds $\times 10$ iterations = 13348.1 seconds = 222.47 minutes.

not likely to be able to have sufficient enough time, we decided to limit the Brute Force to a maximum of 25 inputs. Below are the results.

## 3.1 Results Data Set

Below you can find the results from *one of our test*. As stated above, the Brute Force only goes to 25 inputs. Note that the input column is the Input size of applications, the Algorithm is the type of algorithm we used, the Memory was the goal memory we wanted to produce, the Cost was what we wanted to minimize, the Freed column is the memory that was actually freed (because we could go over the goal if it was an optimal solution) and finally the Time is the time taken (strictly of the algorithm in question) measured in seconds.

| Input | Algorithm | Memory | Cost | Freed | Time |
|-------|-----------|--------|------|-------|------|
| 5 | BRUTE | 609 | 206.960013686 | 769 | 0.0000652 |
| | DYNAMIC | 609 | 413.920027372 | 1538 | 0.008588 |
| | GREEDY | 609 | 206.960013686 | 769 | 0.00572586666667 |
| 10 | BRUTE | 1254.0 | 306.007083914 | 1307 | 0.0023169 |
| | DYNAMIC | 1254.0 | 913.397426561 | 3126 | 0.033041 |
| | GREEDY | 1254.0 | 418.629747 | 1961 | 0.0220281 |
| 15 | BRUTE | 1896.2 | 462.268333337 | 1980 | 0.0933534 |
| | DYNAMIC | 1896.2 | 706.822856172 | 3129 | 0.10722175 |
| | GREEDY | 1896.2 | 506.866912451 | 2383 | 0.0714820333333 |
| 25 | BRUTE | 2469.0 | 619.523226819 | 2479 | 113.6533167 |
| | DYNAMIC | 2469.0 | 1204.68191427 | 4035 | 56.92951935 |
| | GREEDY | 2469.0 | 620.454984919 | 2562 | 37.9530137333 |
| 50 | DYNAMIC | 5421.2 | 1664.89861527 | 7120 | 0.870269 |
| | GREEDY | 5421.2 | 1177.03320938 | 5519 | 0.43513695 |
| 100 | DYNAMIC | 9783.2 | 2809.24740865 | 11268 | 3.8071061 |
| | GREEDY | 9783.2 | 2392.11409139 | 9849 | 1.90355725 |
| 150 | DYNAMIC | 14995.8 | 4027.50565053 | 16594 | 9.4551353 |
| | GREEDY | 14995.8 | 3696.37456362 | 15462 | 4.7275798 |
| 250 | DYNAMIC | 26793.4 | 6591.81933106 | 28200 | 32.8375575 |
| | GREEDY | 26793.4 | 6272.26215744 | 27018 | 16.41878765 |
| 500 | DYNAMIC | 52354.6 | 12547.9234913 | 53429 | 192.5988691 |
| | GREEDY | 52354.6 | 12365.0250774 | 52842 | 96.2994775 |
| 1000 | DYNAMIC | 107148.2 | 25671.8344823 | 109000 | 1334.8144697 |
| | GREEDY | 107148.2 | 25256.6939401 | 107895 | 667.4074229 |

## 3.2 Results

While looking at the data we got back, we notice a very noticeable trend immediately: the exponential nature of the quasi-exponential nature of the algorithms. When plotting on an input size vs. time graph, we could see even the algorithm with $\mathcal{O}(n)$ (the Greedy Solution) complexity showed some exponential growth. Taking into consideration the RAM Model of Computing is not completely applicable in this situation, the results are more than staggering.
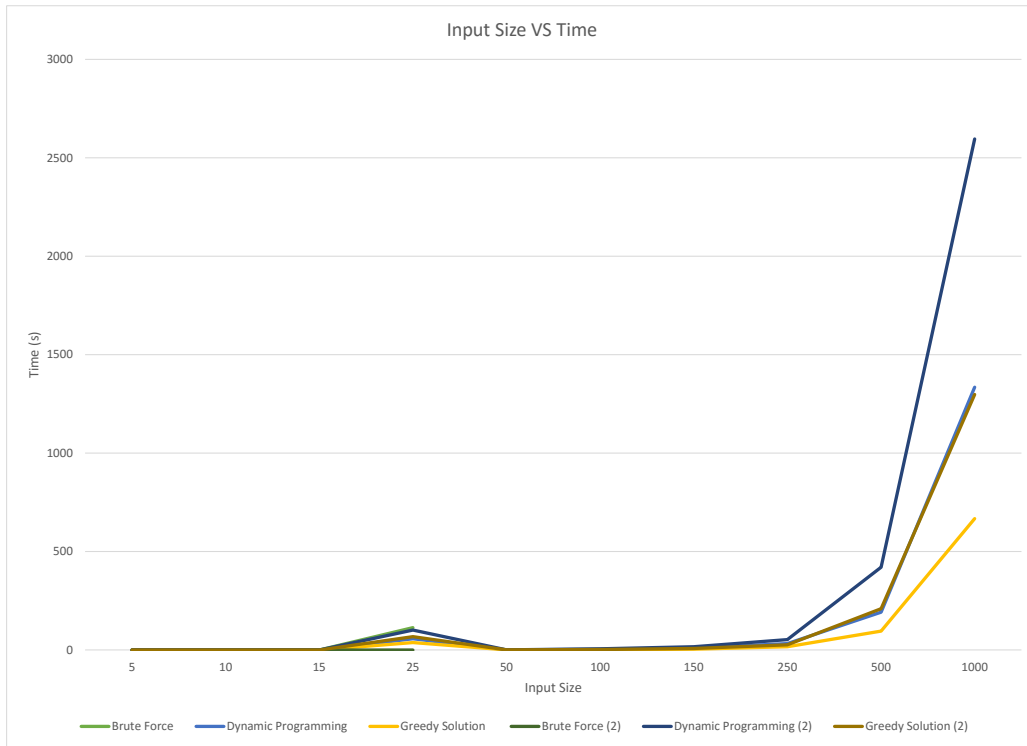


As can be seen in the above figure, although the input follows the pattern of $5n, 10n, 15n, 25n | n \in \mathbb{Z}^+$, the results typically follow a pattern of $2\times$ their predecessor.

When comparing the Time vs. Costs, the trend is also quite exponential. For, as our algorithms get a larger cost, the time is unquestionably larger. When actually looking at the Time vs. Accuracy, the clear cut winner is Greedy. The time saving are immense compared to Dynamic and Brute Force. On certain cases, Greedy can finish before Dynamic and Brute force combined.

**TIme VS Cost**
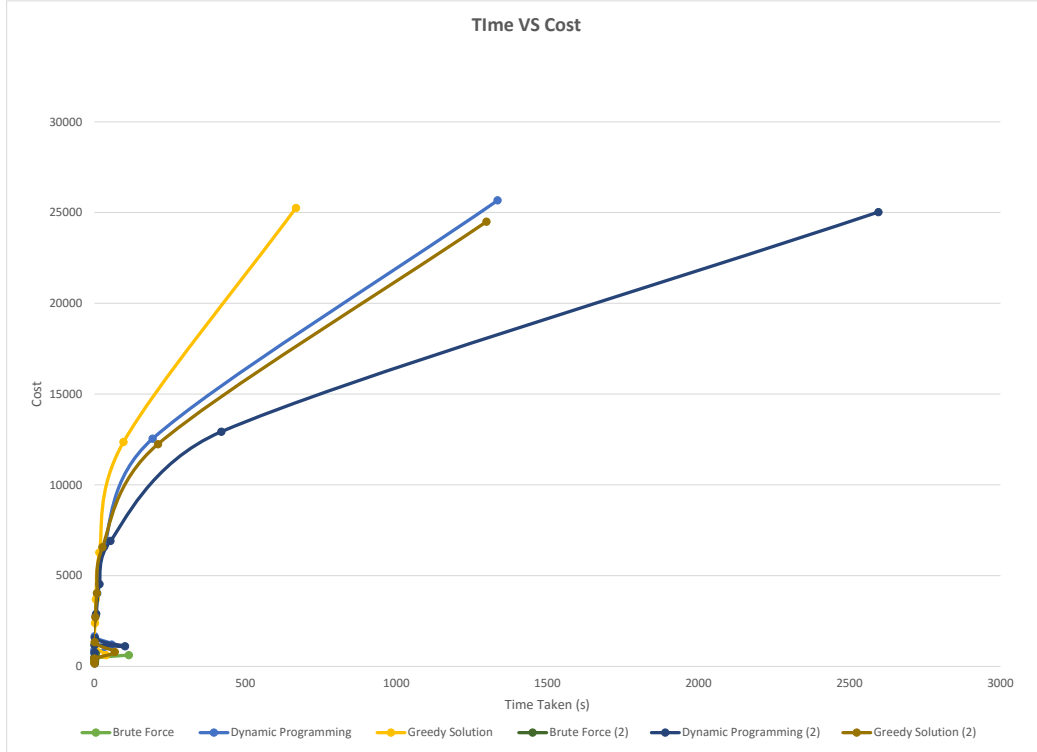
We also see that Greedy Solution outperformed the Dynamic in not only time but accuracy. We are unsure of the reasoning behind this, for our algorithm certainly seemed correct on paper.

Tying this back into the real world example, none of these solution would be able to work. Assuming a smartphone of $2\,\mathrm{GB}$ RAM[4], figuring an optimal solution would take a bare minimum of 38 seconds (not to mention 2 minutes if we are using the Brute Force.)

While this may not be applicable to the real world, it was an interesting thought experiment.

# 4   Roles

- Illya Starikov

  – Project Manager
  – Brute Force Development

---

[4]We are assuming our memory is equivalent to $1\,\mathrm{MB}$. Therefore, $2\,\mathrm{GB} = 2000\,\mathrm{MB}$

- Abstract, Experiment Writement

- Timothy Ott

  - Greedy Algorithm Development
  - Development, Conclusion, Interesting Results Writeup
  - Testing Documents

- Claire Trebing

  - Dynamic Programming Development
  - Experiment Writing Up
  - Quality Assurance

# 5  Interesting Results

In the process of going over the data generated in our experiments we noted several interesting results that we would like to highlight here.

- We noticed that for input sizes under roughly 15 the Brute Force method was both more accurate, which we expected, but also significantly faster than the other two methods. When we take into consideration that the Brute Force's complexity is $\mathcal{O}(n2^n)$ and the Greedy Solution is $\mathcal{O}(2n)$, this is an excellent highlight to the fact that the growth of the function is far more important than its initial run speeds.

- There is an interesting anomaly in the data set for the input size of 25. At this point the run times for each algorithm spike noticeably only to then immediately dip once again. What's more interesting is that as we mentioned earlier we ran these experiments twice, independently of one another on two different machines and this anomaly occurred in both rounds of experimentation. So, what is so special about 25 inputs? The short answer is we are not sure. Our best guess is that the increase and subsequent drop are somehow related to the Brute Force algorithm since that is where we stopped testing that particular algorithm. Particularly we think this may have been the fault of the language's garbage collection functionality but without further testing we cannot be sure.

- Also of note is that while initially our dynamic approach was a good deal less accurate than either the Brute Force or even the Greedy approach, as the input sizes increased the accuracy of the dynamic algorithm also increased.

# 6    Conclusion

Memory management is a complex issue with many interesting solutions of varying effectiveness. With this project we explored three approaches to solve this issue using a smartphone's memory management system as our testing ground. Our solutions took the forms of:

- A brute force approach, comparing every possible subset of solutions to find an optimal choice at the expense of a high complexity and eventual run-time;

- A dynamic programming solution which built a table of solutions to effectively try every solution but with a lower complexity due to being able to look up values that were already computed; and

- A greedy algorithm approach which took a ratio of the cost and memory and produced a solution that was simply taking the lowest ratio apps until the memory requirements were met, in effect trading the accuracy of the other two approaches for a significant increase in speed.

Through our data gathered during experimentation we see that these early predictions hold fairly consistently, despite some unaccounted for inaccuracy on the part of our dynamic solution. While it is tempting to label one solution as the best or optimal solution, this would be misleading as each approach shines under different circumstances. For instance, for small input sizes the brute force algorithm seemed to exceed both other solutions in terms of both accuracy and speed. For higher input sets the greedy algorithm was the clear winner in terms of speed but theoretically would lose out on accuracy to the dynamic solution every time. Thus, we see that is important to tailor your solutions to the problem and not simply favor one approach over another arbitrarily.