Programming Project I, First Report

Illya Starikov, Claire Trebing, & Timothy Ott

Due Date: March 07, 2016

1 Abstract

Smartphone users launch many apps everyday, however one of the most fundamental things a smartphone does is abstracted away: memory management.

Although smartphones have advanced significantly in many ways compared to their first predecessors(RAM, architecture, processors), deactivation, (the process of "the operating system needing to choose and remove some apps from the memory", a subproblem of memory management) is a solution that is often less-than-perfect. Although Java's Garbage Collection and Swift's Automatic Reference Counting (ARC) have sufficed, there are other methods.

In this project I propose to solve this problem using three techniques:

- Brute Force
- Dynamic Programming
- Greedy Solution

2 Introduction and Motivation

As stated previously, memory management is solved in a less-than-perfect manner. Although current technology suffices, we would like to compare algorithms to show the significant gains via three different approaches (Brute Force, Dynamics Programming, and Greedy).

3 Proposed Solution

For our project we decided to take a more skeuomorphic and object oriented approach, modeling objects after their real world counterparts, such as `Application` or `Smartphone`. As for the approaches, we have the following solutions:

3.1 Brute Force

For the brute force method, we knew that we had to check every possible subset (and for a set of size n, we know there to be 2^n subsets). We used this to our advantage, creating a lookup table modeled after a truth table. As an example, suppose we have a three items in our knapsack:

Items A	Item B	Item C	Item Number
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

We notice that the knapsack combination can be directly summarized by the n-bit binary representation. Using the previous example of three item knapsack:

$$\text{Subset } \#0 : 0_{10} = 000_2 = \sim (ABC) \quad = \quad \text{No Items} \tag{1}$$

$$\text{Subset } \#5 : 5_{10} = 101_2 = A \sim (B)C \quad = \quad \text{Items A, B} \tag{2}$$

$$\text{Subset } \#7 : 7_{10} = 111_2 = ABC \quad = \quad \text{Items A, B, C} \tag{3}$$

Now that we have have a represenation of every subset, we can multiply the columns value by the knapsack value to get the item's value into the lookup table. Extending the example, suppose we have the following table:

App	Memory	Cost
A	2	4
B	3	6
C	7	9

Multiplying the columns by the values produces the following results:

App A	App B	App C
0×4	0×6	0×9
0×4	0×6	1×9
0×4	1×6	0×9
0×4	1×6	1×9
1×4	0×6	0×9
1×4	0×6	1×9
1×4	1×6	0×9
1×4	1×6	1×9

$=$

App A	App B	App C
0	0	0
0	0	9
0	6	0
0	6	9
4	0	0
4	0	9
4	6	0
4	6	9

Adding the columns across we produce the following, and factoring the weight:

Items A	Item B	Item C	Total Memory	Total Cost
0	0	0	0	0
0	0	9	9	7
0	6	0	6	3
0	6	9	15	10
4	0	0	4	2
4	0	9	12	9
4	6	0	10	5
4	6	9	19	12

Now finding the minimal value is straightforward, find the minimal value is traversing down the additive value while looking at the size it frees up. Suppose we have to free up 10 blocks of memory, items A, C would be the most efficient choice.

The pseudocode is as follows, with a few notes:

- Comparison is done while "generating" a table.
- A table is not generate, but the binary representation is used instead.

3.1.1 Pseudocode

```
knapsackBrute(items, knapsackSize)
    minumum = 0
    for i = 0 to 2^n - 1
        subset = binaryToInteger(i)
        sum = 0

        for i to subset.length
            sum = sum + item[i].benefit * subset[i]
            size = size + item[i].weight * subset[i]

        if size <= knapsackSize && sum < minumum
            minumum = sum
            greatestSubset = i

    subset = binaryToInteger(greatestSubset)
    for i = 0 to subset.size
        if subset[i] == 1
            optimalSolution.append(item[i])

    return optimalSolution
```

3.1.2 Time Complexity

Analyzing this algorithm, we can see the complexity:

$$\sum_{k=1}^{n} 2^k \times \sum_{k=1}^{n} c = \mathcal{O}(n2^n) \tag{4}$$

We run in $\mathcal{O}(n2^n)$ time.

3.2 Dynamic Programming

3.3 Greedy Solution

After considering several greedy algorithm solutions we settled on selecting based on a ratio of cost and memory freed by deactivating a particular app. Once each of the apps ratio has been evaluated (and the ratio has been stored as a member variable) we sort the array of apps in an ascending order according to this new value. Because the apps are now in order in ascending cost per unit of memory we can now simply add apps to the solution set in the order they appear in the array until our memory requirements are met. For example, if we take our example set of three items:

App	Memory	Cost
A	2	4
B	3	6
C	7	9

We first take the ratio of each items cost over the capacity of memory it takes up and sort the elements accordingly. Doing this gives us the following updated table:

App	Memory	Cost	Ratio
C	7	9	1.29
A	2	4	2
B	3	6	2

And now we simply add apps until our memory requirement has been met or exceeded. Suppose we need to free up 9 units of memory, according to this approach the optimal solution would be items C and A.

It is worth noting, however that this solution doesnt always produce the absolute optimal solution. If we use the previous example of freeing up 10 units of memory this algorithm produces a solution of all the apps, A, B and C as opposed to the actual optimal solution of app C and B. This inaccuracy is a sacrifice in order to benefit from the much lower run time of the other solutions.

We could perhaps introduce a second greedy choice into the algorithm to improve the accuracy but we would do so at the cost of performance and the solutions chosen by your initial solution are accurate enough for our purposes.

3.3.1 Pseudocode

```
knapsackGreedy()
    A[] = Array of Apps
    S[] = Array of solution set
    M = Desired amount of memory to be freed
    sm = Amount of memory freed by solution set

    n = A.length

    for 0 to n
      A[i].ratio = A[i].cost/A[i].memory

    mergesort(A)

    GreedKnap(A, S)
      S[0] = A[0]
      sm = A[0].memory
      i = 1
      while (sm < M AND i < n)
        S[i] = A[i]
        sm += A[i].memory
```

3.3.2 Time Complexity

Analyzing this algorithm, including both going over the array to produce the ratio that will guide our greedy choices and choosing the actual solution set, we come up with the complexity of:

$$\sum_{k=1}^{n} k + \sum_{k=1}^{n} k = \mathcal{O}(2n) \tag{5}$$

This algorithm therefore runs in $\mathcal{O}(2n)$ time.

4 Plan of Experiments

The main objective of our experiments is to extrapolate time and efficiency from the algorithms for sufficiently large input, so our plan of experiments is thus:

1. Generate a random array of apps, and a smartphone with n memory.
2. Take a date stamp.
3. Run the Brute Force algorithm for the random array of apps and the smartphone.
4. Take another date stamp, record the difference.

5. Return to step 2 and run the Dynamic Programming algorithm.
6. Likewise, return to step 2 and run the Greedy algorithm.
7. Record the results from the experiment.
8. Return to step 1, repeat 9 additional times, take the average.
9. Run the algorithm again for a scale factor of n (the input size).

5 Team Roles

Illya Starikov Project Management, Development (Brute Force)

Timothy Ott Development (Greedy), Architecture

Claire Trebing Development (Dynamic), Quality Assurance, Documentation