CS627 Assignment: Unit 3 - Discussion Board

Yiming Pan

September 2, 2020

Abstract

In this assignment, I would like to discuss two problems that can be solved by greedy algorithm.

1 Fractional Knapsack Problem

First I would like to spend some time to talk about KnapSack problem, as it is a perfect problem to practice greedy algorithm and dynamic programming.

As we know, there are two kinds of knapSack problem:

1. Fractional Knapsack Problem
2. non-Fractional Knapsack Problem

The Fractional Knapsack Problem is a very easy greedy algorithm example, while **the non-fractional one is a very hard problem that cannot be solved by Greedy algorithm.** Even using dynamic programming, I would say it is a hard dynamic programming problem. Since next week, we are going to discuss dynamic programming, I will discuss non-fractional knapsack next week.

1.1 Problem Description

Fractional knapsack problem allows us to take fractional weight from the item.

For example:

We have 3 items, A, B, C. A{60, 10}, B{100, 20}, C{120,30} with the first number as weight, second as value. For a knapsack of capacity 50, we need to maximize the value we can carry.

1.2 Algorithm

Since in fractional we don't have to take the item as a whole, we can easily solve this by using greedy algorithm. We use value/weight to calculate the cost for each item. Put the highest cost item in the knapsack first, the second and so on. It is a very straightforward approach.

1.3 Program Implementation in Java

```
public class KnapSack {

    public static void main(String[] args) {
        int[] weights = {10, 40, 20, 30};
        int[] values = {60, 40, 100, 120};
        int capacity = 50;

        double maxValue = getMaxValue(weights, values, capacity);
        System.out.println("Maximum value we can obtain = " +
                maxValue);

    }

    // function to get maximum value
    private static double getMaxValue(int[] weights,
                                      int[] values, int capacity) {

        double totalValue = 0d;

        // init an itemList to contain all the elements.
        Item[] itemList = new Item[weights.length];

        for(int i = 0; i < weights.length; i++)
        {
            itemList[i] = new Item(weights[i], values[i], i);
        }

        selectionSort(itemList, itemList.length);

        for (int i=0; i < itemList.length; i++)
        {
            if (capacity - itemList[i].weight >=0 )
            {
                totalValue = totalValue + itemList[i].value;
                capacity = capacity - (int)itemList[i].weight;
            }
            else
            {
                totalValue = totalValue + itemList[i].cost * capacity;
                break;
            }
        }
        // We need to sort this itemList by each item's cost.
        // For here, I will just use selection sort, because it's easy to use.
```

```
        // If we want to improve time complexity, we can always use other faster
           sortings.

        return totalValue;
    }

    public static class Item
    {
        double cost, weight, value, index;

        public Item(int weight, int value, int index)
        {
            this.weight = weight;
            this.value = value;
            this.index = index;
            this.cost = this.value/this.weight;
        }
    }

    private static void selectionSort(Item arr[], int n)
    {
        int maxIndex;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            maxIndex = i;
            for (int j = i+1; j < n; j++)
                if (arr[j].cost > arr[maxIndex].cost)
                    maxIndex = j;

            // Swap the found minimum element with the first
            // element
            Item temp = arr[maxIndex];
            arr[maxIndex] = arr[i];
            arr[i] = temp;
        }
    }


}
```

1.4 Discuss

For our approach, we sort the cost from high to low. And then, we put the highest cost item into the knapsack first and then second. As we know that greedy algorithm is defined as making optimal choice at each step to solve the problem. Our algorithm indeed always choses the highest cost item at each step, and as a result it is a greedy algorithm.

For the time complexity, this greedy algorithm's complexity pretty much depends on how we sort our cost list. As we can see in our code, we are using a selection sort(I put the selection sort part down below) to sort our cost list. Of course, we can improve our time complexity by using efficient sort like merge sort, and the time complexity would be $O(nlogn)$.

```
private static void selectionSort(Item arr[], int n)
{
    int maxIndex;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        maxIndex = i;
        for (int j = i+1; j < n; j++)
            if (arr[j].cost > arr[maxIndex].cost)
                maxIndex = j;

        // Swap the found minimum element with the first
        // element
        Item temp = arr[maxIndex];
        arr[maxIndex] = arr[i];
        arr[i] = temp;
    }
}
```

2 Real Coin Machine Problem

Coin machine is a very popular problem that occurs a lot in code interviews. Personally, I was asked to solve this problem in NYC by Bloomberg in February 2020. We can also find it in Leetcode. It is currently the Leetcode problem number 322. The algorithm for this problem is not hard at all and can be solved very intuitively. However, when doing coding, it is very easy to miss the edge cases.

Similar to the fractional knapsack problem, the coin machine problem can only be solved by greedy algorithm with certain constraints. That's why I called it the real-coin machine problem. It only works with real coin input, for example: { 1, 2, 5, 10, 20, 50, 100, 500, 1000}. If we have strange coins like this: {186,419,83,408}, then greedy algorithm won't work and it becomes a hard dynamic programming problem.

2.1 Problem Description

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Example 2:
Input: coins = [2], amount = 3
Output: -1

2.2 Algorithm

The algorithm is very simple, simply keep using the largest coin until the amount is smaller than the largest coin, then the second largest and so on.

2.3 Program Implementation in Java

```
class Solution {
    public int coinChange(int[] coins, int amount) {

        selectionSort(coins,coins.length);
        int totalTimes = 0 ;

        for (int i=0; i< coins.length; i++)
        {
            if (coins[i]<=amount)
            {
                amount = amount - coins[i];
                 i--;
                totalTimes ++;
            }
        }

        if (amount > 0 )
            return -1;

        return totalTimes;
    }

    private static void selectionSort(int arr[], int n)
    {
```

```
        int maxIndex;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            maxIndex = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] > arr[maxIndex])
                    maxIndex = j;

            // Swap the found minimum element with the first
            // element
            int temp = arr[maxIndex];
            arr[maxIndex] = arr[i];
            arr[i] = temp;
        }
    }
}
```

2.4 Discuss

Same as the fractional knapsack problem, first we sort the coins from high to low. Sometimes, you will be given a sorted coin list, so you don't have to sort it by yourself. As we know that greedy algorithm is defined as making optimal choice at each step to solve the problem. Our algorithm indeed always choses the highest value coin at each step, and as a result it is a greedy algorithm.

For the time complexity, this greedy algorithm's complexity also pretty much depends on how we sort our cost list. Again, I'm using a simple selection sort because I'm a lazy person. We can use a merge or a build in sort by the language. For example, in python I believe it's list.sort() which will be really easy to use. And I believe the the build-in function python uses is quick sort which is not bad. And it takes O(n) times to compare and finds the solution, so the total time complexity is $O(nlogn)$.

3 More thoughts on greedy algorithm

3.1 Advantage

1. Simplicity: Greedy algorithm is very simple to understand and use.
2. Optimal: In most cases, if a greedy algorithm can be used, it will be the optimal solution.
3. Efficiency: In most cases, if a greedy algorithm can be used, it will be the fastest solution.

3.2 Drawbacks

1. Hard to design: these two cases I mentioned are very simple ones. However, many greedy algorithms are very hard to design.

2. Hard to verify: Even you design a greedy algorithm, it will be very hard to prove that it's a correct solution.

3. Not safe during interview: Unless you have encountered the problem before and are sure of greedy is the correct solution, I would not recommend using greedy algorithm during interview. All the problems that can be solved by greedy algorithm can be solved safely either using dynamic programming or recursion. Since it is super hard to verify your greedy algorithm is correct, it is not safe to use it during interview.

4 Read Me

For your inconvenience, **A MS word file and pdf file of this article are both attached**. You might find exact same code that I used in this article at https://easonback26.github.io/ShadowArchive/. That is because it's my personal blog website.

References

[1] *Coin Change*, available at `https://leetcode.com/problems/coin-change/`.