Josh Dow

April 21, 2019

1 Recurrence For Counting Coins With Arbitrary Denominations

A recurrence for the minimum number of coins problem can be seen below. Given an amount n and the array of denominations $denom[]$:

$$LeastCoinCount(n) = min \begin{cases} 0 & \text{if n = 0} \\ 1 + LeastCoinCount(n - denom[j]) & \forall denom[j] \leq n \end{cases}$$

2 Pseudocode For Divide and Conquer Algorithm Of Counting Coins With Arbitrary Denominations

Based on the above recurrance $LeastCoinCount(n)$, a divide and conquor algorithm can be made to solve the Coin Counting problem. The trivial solution is when no coins are required to give the change ($n = 0$). Subproblems are generated from the change required subtracting each possible coin denomination until 0 additional change is required. The solutions are then combined by taking the minimum amount of all of the possible coins at each level. This leads to the following pseudo-code:

```
1. int LeastChange(int n, int[] denom)
2.     if(n ==0):
3.         return 0
4.     int minChange = n
5.     for(int i in denom):
6.         if (i ≤ n):
7.             minChange = min(minChange, LeastChange(n-i, denom))
8.     return minChange + 1
```

3 Correctness of the Divide and Conquer Algorithm

In order to prove correctness of the above algorithm several claims would need to be established:
Claim 3.1: The first is that if there is no change reqired $n = 0$, then no coins are required $count = 0$.
Claim 3.2: Since pennies are always available ($\exists i$ such that $denom[i] = 1$), the worst case change making is $n/1 = n$ since all other denominations are greater than 1.
Claim 3.3: The next step would be to prove through normal induction on n that the correct coin count for any change would be returned. Then the use of a loop invariant to prove the minimum count is being returned, along side with a bound function to prove termination of the loop, would complete the proof.

4 Dynamic Programming Algorithm For Counting Coins with Arbitrary Denominations

In the above divide and conquer solution, several subproblems are solved multiple times. This can be eliminated through the use of a Dynamic Programing algorithm. Here the solution is built in a bottom up fashion. So rather than dividing from the top down, more trivial solutions are combined for more complex ones. For the above algorithm the trivial instances are any time $n = 0$ and perfect change has already been made. From here subproblems of increasing difficulty can be combined based on the possible denominations of the coins. Since we want to build the subproblem solutions from the case $i = 0$ up to $i = n$, an incrementing loop is used, which checks for the solution of the subproblem for $n = i$ based on the solutions for the subproblems for all of the indexes at i minus a denomination of a coin. In this manner the subproblems are solved once each up to n.
Assuming a given array of coin denominations $denom[]$, and a change amount n, a dynamic example can be seen below:

```
1. leastChangeDynamic(int n, int[ ] denom):
   //Fill array of length n with n, (O(n) running time). Solutions held here
2. solutions = int[n].fill(n)
3. solutions[0] = 0
4. for i in range(0,n):
5.     for j in range(0, denom.length-1):
6.         if (denom[j] ≤ n):
7.             solutions[i] = min(solutions[i], solutions[i-denom[j]]+1)
8. return solutions[n]
```

5 Discussion Of Loop Invariants For Above Algorithm

The loop invariants for the above algorithm are shown below as follows. For the outer loop, the invariants are:
Claim 5.1: Loop invariants for the outer loop:

- n is an non-negative integer
- $0 \leq i \leq n$, with i being an integer
- For all indexes $x < i$, $solutions[x]$ holds the minimum amount of coins to make change for x

Claim 5.2: When the algorithm is executed with the preconditions satisfied, and the loop is reached, then the loop invariant will be satisfied at this time.
Claim 5.3: If the loop invariant is held at the beginning of an execution of a loop iteration, then it will also be satisfied at the end of the execution of the iteration.
Claim 5.4: Loop invariants for the inner loop:

- n is a non-negative integer
- $0 \leq j \leq denom.length$, with j being an integer
- $solutions[i] = min(n, solutions[i - denom[x]] + 1), \forall x < j$

Claim 5.5: When the algorithm is executed with the preconditions satisfied, and the loop is reached, then the loop invariant will be satisfied at this time.
Claim 5.6: If the loop invariant is held at the beginning of an execution of a loop iteration, then it will also be satisfied at the end of the execution of the iteration.

The above claims would be proven by inspection of the code at key points to satisfy the assertions above. The key points that need to be tested are: at the beginning of every execution of each loop, the end of every execution of the loop, the beginning of every loop body, and at the end of every loop body.

6 Proof Of Partial Correctness

Claim: The *change* algorithm is partially correct, when considered as an algorithm for the "Counting Coins with Arbitrary Denominations" problem.

Proof. Consider an execution of this algorithm with the precondition for the "Counting Coins with Arbitrary Denominations" problem satisfied. Then a non-negative integer n is given as input. The problems postcondition is "the number

$$c = \sum_{i=0}^{n-1} change[i]$$

of coins used, in a solution "change" for this instance of the "Making Change with Arbitrary Denominations", is returned as output." We will look at this when establishing partial correctness in these cases.

For all $n \geq 0$, steps 1, 2, 3, 4 are executed and the first loop is reached and executed. This could either terminate or never terminate.

- If the execution of this loop terminates, that means that the loop invariants are satisfied when the execution of the loop ends. This means that n is a non-negative integer, i is an integer where $0 \leq i \leq n$, and for all indexes $x < i, solutions[x]$ holds the minimum amount of coins to make change for x.
- What also must happen is that the loop test in line 4 must be checked and failed. This will mean that the entire range from 0 to n was accessed and then it jumps down to line 8 where it returns $solutions[n]$, which is an integer. This follows the problems postcondition as needed to establish this cases claim.
- If the execution of the loop does not end, then the algorithm will not either. Then this would establish partial correctness.

Also if $n \geq 1$, after the first loop is accessed, the second loop at line 5 is reached and executed.

- If the execution of this loop terminates, that means that the loop invariants are satisfied when the execution of the loop ends. This means that n is a non-negative integer, $0 \leq j \leq denom.length$, with j being an integer, and for all indexes $x < j, solutions[i] = min(n, solutions[i - denom[x]] + 1)$.
- What also must happen is that the loop test in line 5 must be checked and failed. This will mean that the entire range from 0 to $denom.length - 1$ was accessed and then it breaks to the outer loop where it either eventually returns $solutions[n]$, which is an integer, or never terminates.
- If the execution of the loop does not end, then the algorithm will not either. Then this would establish partial correctness.

□

7 Bound Function for Dynamic Programming Algorithm

There are two different bound functions for the above algorithm.

Claim 7.1: The bound function for the inner loop is $f_{b_{inner}}(denom.length, j) = denom.length - j$. This is correct because of the following three reasons.

- Since $denom.length$ is the length of an array as an integer and j is an integer value, $denom.length - j$ will always be integer valued.

- When the body of the loop is executed, the counter j is increased by 1 and $denom.length$ is not changed. So then the value of $f_{binner}(denom.length, j)$ is decreased by 1.

- If the value of $f_{binner}(denom.length, j)$ ever equals zero, that'll mean $denom.length - j = 0$. Since that is the bottom end of the range on the loop, it'll will continue on with the execution of the code past the loop.

Since all of the conditions required to be a bound function are met by $f_{b_{inner}}$, it can be used to show termination of the inner loop.

Claim 7.2: The bound function for the outer loop is $f_{b_{outer}}(n, i) = n - i$. This is very similar to the inner loop.

- Since n is an integer input and i is an integer value, $n - i$ will always be integer valued.

- When the body of the loop is executed, the counter i is increased by 1 and n is not changed. So then the value of $f_{bouter}(n, i)$ is decreased by 1.

- If the value of $f_{bouter}(n, i)$ ever equals zero, that'll mean $n - i = 0$. Since that is the bottom end of the range on the loop, it'll continue on with the return statement at line 8.

Since all of the conditions required to be a bound function are met by $f_{b_{outer}}$, it can be used to show termination of the outer loop.

8 Proof Of Termination Using Bound Functions

To prove that the termination for loops one must prove two things:

1. If the problem's precondition is satisfied when an execution of the algorithm including this loop begins, then every execution of the loop body ends

2. A bound function for the loop exists.

Claim: When the *change* algorithm is executed when the precondition for the "Counting Coins with Arbitrary Denominations" problem is satisfied, then this execution terminates.

Proof. Consider an execution of this algorithm with the precondition for the "Counting Coins with Arbitrary Denominations" problem satisfied.A non-negative integer n is given as input.

For all $n \geq 0$, steps 1, 2, 3, 4 are executed and the first loop is reached and executed. The body of this loop has only 3 steps and they never effect the conditions of the loop. Since the loop is for the range 0 to n, those are the

values that would need to be affected so that the loop doesn't end. We can see by inspection of the code that the only line that changes the value of something is line 7, which can change values in the integer array *solutions*. This certainly terminates whenever it is executed. It also follows by the second part of the definition, which is that the loop has a bound function. This bound function is $n - 1$ as shown in the previous question.

For all $n \geq 0$, after the first loop is reached and executed, it will reach the second loop on line 5. The body of this loop has only 2 steps and they both never also effect the conditions of the loop. Since the loop is for the range 0 to $denom.length - 1$, those are the values that would need to be affected so that the loop doesn't end. We can see by inspection of the code that the only line that changes the value of something is line 7, which can change values in the integer array *solutions*. This has no effect on *denom.length*. This then certainly terminates whenever it is executed. It also follows by the second part of the definition, which is that the loop has a bound function. This bound function is $denom.length - j$ as shown in the previous question.

After both of these loops have been terminated, the algorithm will surely terminate since the only line left to execute is the return statement at line 8. □

9 Asymptotic Notation

Claim: The algorithm *LeastChangeDynamic* has a worst case running time of $O(n \times m)$ with n being the change required and m being the length of the array $denom[]$.

Proof. The first stage in this proof is to find the run time cost before the outer loop execution begins. Assuming that an array can be made and initialized in $O(n)$ steps, then there are $O(n)+1$ steps before the outer loop execution begins. The second stage in this proof is to find the worst case running time for the inner loop. The inner loop iterates through each member of the $denom[]$ array, with either 1 or 2 steps from the uniform cost criterion. Therefore supposing that the length of the array $denom[]$ is m, then at most there are $2m$ steps in the inner loop. The third stage is to find the cost of the outer loop. The outer loop iterates from 0 to n, each time executing the inner loop with the worst case run time of $O(m)$. This means that the outer loop has the total run time of $O(n \times m)$ with n being the amount of change required, and m being the number of possible denominations. The total algorithm therefore has the worst case run time of $O(n) + 1 + O(n \times m) + 1$ which can be simplified to the fastest growing term of $O(n \times m)$. □

10 Pseudocode For Memoized Algorithm Of Counting Coins With Arbitrary Denominations

To solve this in a Memorized manner, several steps were followed. The first was to solve the problem recursively as done in part 2. The next step is to find a data structure to hold the solutions. For this problem, an array of integers was again chosen to hold the solved problems, with the index representing the change amount and the value representing the minimum number of coins to make that value. The function to initialize this array was taken from the Lecture 11 slides, here it is called the $InitializeSolution$ function. The function initializes an array of integers of length n to all $Null$ values, then returns it. It does this in $3n + 7$ steps. The next step was to make a function to generate and store the solutions based on the Divide and Conquer algorithm. $RecursiveMemo$ does this, checking if the problem has been solved, returning the saved solution if possible, solving the problem otherwise, then saving and returning the solution. A final method is required to check for trivial solutions, call the initialization method, as well as the recursive method. This is $MemoMain$.

```
1.  int RecursiveMemo(int n, int[] denom, int[] solutions)
2.     if(n ==0):
3.        return 0
4.     if(solutions[n] != Null):
5.        return solutions[n]
6.     int minChange = n
7.     for(int i in denom):
8.        if (i ≤ n):
9.           minChange = min(minChange, leastChange(n-i, denom))
10.    solution[n] = minChange + 1
11.    return minChange + 1
```

```
1. MemoMain(int n, int[] denom):
2. if(n==0):
3.    return 0
4. solutions = InitializeSolutions(n)
5. return RecursiveMemo(n, denom, solutions)
```

11 Sketch Of A Proof Of Correctness Of Main Method

The following claims would be made in order to prove the correctness of the algorithm $MemoMain$:

- **Claim 1:** If i is an integer such that $0 \leq i \leq n$ and that $RecursiveMemo$ is being has not been executed previously as part of an execution of the $MemoMain$ algorithm, then $solutions[i] = Null$ for every integer i. This would be proved by inspection of the code and showing that the only place where an index of $solution[]$ is updated is in the $RecusiveMemo$ method.

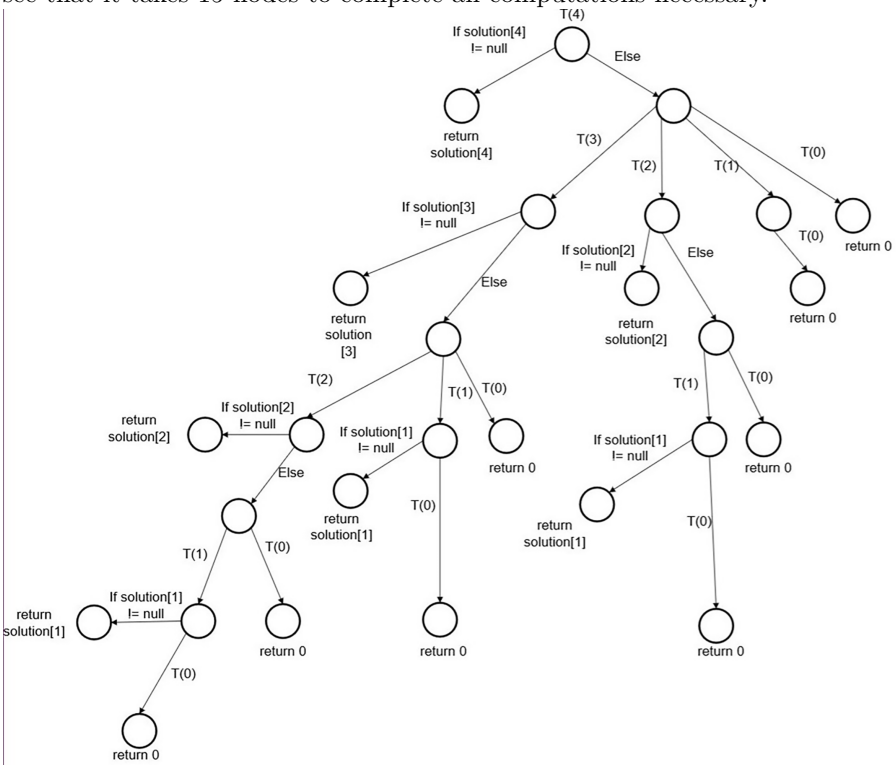
- **Claim 2:** Suppose i is an integer with $0 \leq i \leq n$, and that the $RecursiveMemo$ is executed with input i, with $solutions[i]$ being the minimum number of coins to make i. Then the minimum number of coins is returned to make i and $solutions[i]$ remains correct upon termination. This would be proved by inspection of the code and showing that the value is returned before any modifications are made.

- **Claim 3:** For every integer i with $0 \leq i \leq n$, every execution of the $RecursiveMemo$ on input i ends, with the minimum coin count being returned as output. Furthermore, $solutions[i]$ is equal to this value when the execution ends. This would be proven through induction on i, with $i = 0$ being the base case.

- **Claim 4:** The algorithm $MemoMain$ is correct. This would be proven through inspection of the code, as well as the use of Claim 3 when the input n is greater than or equal to 1.

12 Recursion Tree Method For Proof Of Asymptotic Worst Case

It can be shown that by following each "Else" branch to the next decremented T(n-i) it is seen that our algorithm does in fact operate at O(n * m). In our example below where n first equals 4, we then know that denom[] can at maximum contain [1,2,3,4]. Then n * m is 16, and by following out branches down we see that it takes 15 nodes to complete all computations necessary.



13 Discussion Of Making Change With Arbitrary Denominations

The above Dynamic algorithm can be modified to solve the Making Change with Arbitrary Denominations problem. The approach we took to solve this is to construct a second 2 dimensional array. It has n entries of $denom[].length$, with each index in the inner array holding the number of coins of that denomination to make change for n. The solutions are constructed bottom up still, with the simpler solution $(n - denom[j])$ being copied into the 2D array at position n, and then incrementing the value in that array at position j. In this way it is possible to keep track of how many of each coins are required for each

subproblem solution. Some sample pseudo-code is shown below:

```
1. leastChangeDynamic(int n, int[] denom):
   //Fill array of length n with n, (O(n) running time). Solutions held here
2. solutions = int[n].fill(n)
3. coinsRequired = int[n][denom.length]
4. solutions[0] = 0
5. int denomIndex = 0 //Penny
6. for i in range(0,n):
7.     for j in range(0, denom.length-1):
8.         if (denom[j] ≤ n):
9.             if(solutions[i] > solutions[i-denom[j]]+1):
10.                solutions[i] = solutions[i-denom[j]]
11.                denomIndex = j
12.    coinsRequired[i] = coinsRequired[i-denom[denomIndex]]
13.    coinsRequired[i][denomIndex] = coinsRequired[i][denomIndex] + 1
14. return solutions[n]
```

It can be seen here that the cost of the algorithm before the loops has increased. It is now $O(n) + O(n \times m) + 1$ with the second element being how long it takes to initialize the 2D array for the solutions. The inner loop has also changed, however it remains in the $O(m)$ category. The outer has increased as well, now requiring an array of length m to be copied. This does not change the overall notation for the loop however, as it remains $O(n \times m)$. This means that the fastest growing term remains $O(n \times m)$, the same as the original algorithm.