CS1Bh Lecture Note 5

Case Study: Coin Changing

In this note we consider a new case study in algorithm design. We will be able to put in practice what we have learnt in the last three lectures about the divide-and-conquer approach, greedy algorithms and dynamic programming.

The problem we consider is that of giving change in coins for a given amount of money. This is a very common activity, whenever a salesperson gives you change in a shop they are doing this process. Here we consider the generalised problem where the coin set is a parameter of the problem. Stating the problem succinctly:

> *Given a* coin set *consisting of k coins with values $c_0 > c_1 > \ldots c_{k-1}$, and an amount a to be changed, find non-negative integers $n_0, n_2 \ldots n_{k-1}$ so that $\Sigma_{i=0}^{k-1} n_i c_i = a$. In addition there should be no other non-negative integers $m_0, \ldots m_{k-1}$ such that $\Sigma_{i=0}^{k-1} m_i c_i = a$ and $\Sigma_{i=0}^{k-1} m_i < \Sigma_{i=0}^{k-1} n_i$.*

Notice that this is again an optimization problem, like most of the problems we have solved in the previous lectures.

In the remainder of this note we consider a variant of this problem in which we are only interested in finding the minimum number of coins needed. The solutions given here could easily be modified to return the numbers of each coin type. This choice reduces the amount of code with little loss of interest in the methods.

The divide-and-conquer approach

The key observation is that we can split the potential solutions into two disjoint classes, those solutions that use c_0 at least once and those that do not:

Use c_0 at least once. In this case the number of coins required is 1 (for a use of c_0) plus the number of coins needed to solve changing $a - c_0$ using $c_0, \ldots c_{k-1}$.

Do not use c_0. In this case we just need to know the number of coins needed to change a using $c_1, \ldots c_{k-1}$.

The solution is the minimum of these two quantities. Notice however that in the case $a < c_0$ only the 'do not use c_1' option is available.

In giving a solution we need to consider two more issues:

- Provide a basis for the recursion (cases in which no recursive call is needed). There are two such cases:

  - When a is zero then we need no coins.
  - When a is non-zero and we have no coins to offer change with – in this case the answer should be infinity. In our implementation we just use `Integer.MAX_VALUE`.

- Guarantee that the solution does not loop endlessly, that is, that the basis cases are evenually reached. For this, consider the parameter $(a + k)$. We can see that it always decreases for either of the steps (use c_0 or not), and that it cannot decrease for ever without reaching one of the basis cases.

The class `Change` below implements this solution in Java. The values of a coin set are stored in an array `coins`. For instance, the UK set corresponds to `coins[0]=100`, `coins[1]=50`,`coins[2]=20`, `coins[3]=10`, `coins[4]=5`,`coins[5]=2`, `coins[6]=1`. The method `change()` accepts an amount and a coin set as input and returns the minimal number of coins whose values add up to the amount.

```
class Change {
  private static int[] c;

  public static int change(int amount, int[] coins){
     c = coins;
     return change(amount,0);
  }
  private static int change(int amount, int j) {
    if (amount == 0) return(0);
    if (j == c.length) return(Integer.MAX_VALUE);
    if (amount < c[j]) return(change(amount,j+1));
    else {
      int c1 = change(amount,j+1);
      int c2 = 1 + change(amount-c[j],j);
      if (c1 < c2) return(c1);
      else return(c2);
    }
  }
}
```

This solution is extremely inefficient – can you estimate its running time in terms of the amount and number of coins?

The greedy strategy

We now consider the greedy strategy followed by salespeople in shops when giving change. In this approach we always try to give as many large coins as possible before considering smaller coins. The class `Greedy` implements this strategy. Notice that we assume that the values of the coin system are stored in decreasing order.

```
class Greedy {

  public static int change(int amount, int[] coins){
    int noCoins = 0;

    for (int i=0; i<coins.length; i++) {
      if (amount >= coins[i]) {
        noCoins += amount / coins[i];
        amount  %= coins[i];
      }
    }
    if (amount > 0) return(Integer.MAX_VALUE);
    else return(noCoins);
  }
}
```

The change giving method just loops through the coins from largest to smallest. It computes for each value the largest number of coins that can be substracted from the amount, and updates the amount.

The greedy strategy is optimal for the UK coin set, but not for others. Can you come up with such a set? One useful fact is that a sufficient condition for the greedy strategy to produce the optimal result is that the coins in the coin set satisfy: $c_i \geq 2c_{i+1}$ for $0 \leq i < k-1$. Most currencies in the world design their coin sets to ensure this property.

Can you estimate the running time in terms of the amount and number of coins?

Dynamic programming

The inefficiency of the recursive solution is due to repeated computation of the same calls of the `change()` method. We apply the techniques introduced in Lecture Note 4. The recursive calls of `change()` in the class `Change` are of the form `change(x,i)`, where `x` ranges between `0` and the initial value of `amount` and `i` is an index between `0` and `coins.length`. So rather than use a recursive call to compute the intermediate results, the first step is to store them in a table `tab[amount+1][coins.length+1]`. The next step is to get rid of the recursion by filling the table 'bottom up' with respect to the tree of recursive calls. This means computing the entries `tab[i][j+1]` and `tab[i-coins[j]][j]` before the entry `tab[i][j]`. For this, we observe that is is enough to fill the elements `tab[i][j]` of the table in *ascending* order for `i` and *descending* order for `j`. This is easily achieved by means of a nested loop.

The class `Dynamic` implements this solution. The time and space complexity is $O(ak)$ for an amount a and a set of k coins.

```
class Dynamic {

  public static int change(int amount, int[] coins){
    int[][] tab = new int[amount+1][coins.length+1];

    for(int i = 0; i <= amount; i++)
      tab[i][coins.length] = Integer.MAX_VALUE;
    for(int j = 0; j < coins.length; j++)
      tab[0][j] = 0;

    for(int i = 1; i <= amount; i++) {
      for(int j = coins.length-1; j >= 0; j--) {
        if (i < coins[j]) tab[i][j] = tab[i][j+1];
        else {
          int c1 = tab[i][j+1];
          int c2 = 1 + tab[i-coins[j]][j];
          if (c1 < c2) tab[i][j] = c1;
          else tab[i][j] = c2;
        }
      }
    }
    return(tab[amount][0]);
  }
}
```

```
class Dynamic2 {

  public static int change(int amount, int[] coins){
    int[] tab = new int[amount+1];

    tab[0]=0;
    for(int i = 1; i <= amount; i++) {
      tab[i] = Integer.MAX_VALUE;
      for(int j = coins.length-1; j >= 0 ; j--) {
        if (i >= coins[j]){
          int c = 1 + tab[i-coins[j]];
          if (c < tab[i]) tab[i] = c;
        }
      }
    }
    return(tab[amount]);
  }
}
```

Finally, the last step is to optimise the memory requirements of the program. In order to compute the entries of the `j`-th column we do not need those of the columns `j+2`, ..., `coins.length`, and so it is not necessary to store them. In fact, there is no need to store the `j+1`-st column either, we can reuse the space for the `j`-th column (can you see why?). This solution is imlemented in `Dynamic2`, and has a space complexity of only $\Theta(a)$, instead of $\Theta(ak)$. However, a can still be large. Can we do even better? For this we observe that in order to compute `tab[i]` we do not need all the values `tab[i-1]` ... `tab[0]`, but only those in the range

`[tab[i-1]` ... `tab[i-coins[0]]`

(recall that we assume that the coin set is stored in descending order, and so `coins[0]` is the largest coin value). Imagine that `coins[0]` is 100. For computing `tab[101]` we do not need `tab[0]` anymore. So we can reuse `tab[0]` for `tab[101]`, `tab[1]` for `tab[102]`, etc.! This is implemented in the class `Dynamic3`. This solution has a space complexity of $\Theta(c_0)$

```
class Dynamic3 {

  public static int change(int amount, int[] coins){
    int k = coins[0]+1;
    int[] tab = new int[k];

    tab[0]=0;
    for(int i = 1; i <= amount; i++) {
      tab[i%k] = Integer.MAX_VALUE;
      for(int j = coins.length-1; j >= 0 ; j--) {
        if (i >= coins[j]){
          int c = 1 + tab[(i-coins[j])%k];
          if (c < tab[i%k]) tab[i%k] = c;
        }
      }
    }
    return(tab[amount%k]);
  }
}
```

Stuart Anderson and Javier Esparza.
Javier Esparza, February 3rd, 2003.