Unity id : amanur

Name - Anusha Gururaja Manur

HOMEWORK - 3

1. *Purpose: Apply various algorithm design strategies to solve a problem, practice formulating and analyzing algorithms, implement an algorithm.* In the US, coins are minted with denominations of 50, 25, 10, 5, and 1 cent. An algorithm for making change using the *smallest* possible number of coins repeatedly returns the biggest coin smaller than the amount to be changed until it is zero. For example, 17 cents will result in the series 10 cents, 5 cents, 1 cent, and 1 cent.

a) (4 points) Give a recursive algorithm that generates a similar series of coins for changing *n* cents. Don't use dynamic programming for this problem.

Algorithm

amt = Total amount for which we we need to generate change
denomination = Array containing [50,25,10,5,1]

```
CoinChange(amt,denomination):
        If amt == 0
            Return 0

        current = denomination[0]

        If amt > current
            num_coins = amt // current
            amt=amt - num_coins*current
            print ( str(current) * num_coins)

        CoinChange(amt ,denomination[1:])
```

Description

We write a recursive algorithm which recursively divides the amount till becomes 0. Starting with the maximum denomination, we divide amt by that denomination to obtain the number of times that denomination can be used and subtract that value from amt, before the next recursive call which processes the updated amt with the next denomination. Since we have a 1 cent denomination, the algorithm will always terminate.

b) (4 points) Write an O(1) (non-recursive!) algorithm to compute the number of returned coins.

Algorithm
amt = Total amount for which we we need to generate change
denomination = Array containing [50,25,10,5,1]

```
1. count=0
2. for current in denomination:
3.     if amt>=current:
4.         num_coins=amt // current
5.         amt=amt - num_coins*current
6.         count=count+num_coins
7. Print (count)
```

Description

The algorithm is the same as above, but a non-recursive solution. We also track the number of coins used as opposed to the series of coins as done in 1a.

The above algorithm takes constant time O(1) as it runs for exactly 5 times (Number of denominations) and does not depend on the amount (n)

c) (1 point) Show that the above greedy algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents.

For a value of 13 with the given denominations, our algorithm would give a split of
13 = 10+1+1+1 as it considers the denominations in the decreasing order. This gives the minimum number of coins to be 4.

Whereas, 13 can be split as
13=6+6+1, which gives a minimum number of coins to be 3 which is the optimal solution.
Hence, the greedy algorithm does not always give the minimum number of coins.

d) (6 points) Given a set of arbitrary denominations $C = (c_1, ..., c_d)$, describe an algorithm that uses **dynamic programming** to compute the minimum number of coins required for making change. You may assume that C contains 1 cent, that all denominations are different, and that the denominations occur in increasing order.

Algorithm

```
1. def CoinChange(amount, C):
2. initialise DP[0....amount] = Infinity
3. n = len (C)
4. DP[0] = 0
5. for i = 1 to amount:
6.         for j = 1 to n:
```

```
7.                  if C[j] <=i :
8.                         temp = DP[i - C[j]] +1
9.                         if ( temp < DP[i]): # Finding minimum
10.                               DP[i] = temp
11.
12. return DP[amount]
```

Example
Example:
Let the amount to change be 13 cents and C = [1, 6, 10]
DP[0] = 0 from base case
DP=

0	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**

DP[1] = 1+min(DP[1-C[1]], DP[1-C[2]], DP[1-C[3]])
DP[1] = 1+ DP[0] = 1
(As DP[1-C[2]], DP[1-C[3]] are not calculated because of the if condition in line 7)
DP=

0	**1**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**

DP[2] = 1+min(DP[2-C[1]], DP[2-C[2]], DP[2-C[3]])
DP[2]=1+DP[1] =2
(As DP[2-C[2]], DP[2-C[3]] are not calculated because of the if condition in line 7)
DP=

0	**1**	**2**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**

.

.

.
DP[5] = 1+min(DP[5-C[1]], DP[5-C[2]], DP[5-C[3]])
DP[5]=1+DP[1] =2
(As DP[5-C[2]], DP[5-C[3]] are not calculated because of the if condition in line 7)
DP=

0	**1**	**2**	**3**	**4**	**5**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**

DP[6] = 1+min(DP[6-C[1]], DP[6-C[2]], DP[6-C[3]])
DP[6] = 1+min(DP[5], DP[0]) = 1+min(5, 0) = 1
(As DP[6-C[3]] is not calculated because of the if condition in line 7)

DP=

0	**1**	**2**	**3**	**4**	**5**	**1**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**	**inf**

.
.
.

DP[9] = 1+min(DP[9-C[1]], DP[9-C[2]], DP[9-C[3]])
DP[9] =1+min(DP[8], DP[3]) = 1+min(3, 3) = 4
(As DP[9-C[3]] is not calculated because of the if condition in line 7)
DP=

0	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**inf**	**inf**	**inf**	**inf**

DP[10] = 1+min(DP[10-C[1]], DP[10-C[2]], DP[10-C[3]])
DP[10] =1+min(DP[9], DP[4], DP[0]) = 1+min(4, 4,0) = 1

DP=

0	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**1**	**inf**	**inf**	**inf**

.
.
.

DP[13] = 1 + min(DP[13-C[1]], DP[13-C[2]], DP[13-C[3]])
DP[13] = 1+min(DP[12], DP[7], DP[3]) = 1+min(2, 2, 3) = 2

DP=

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
DP[i]	**0**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**1**	**2**	**2**	**3**

DP[13]=3

In the above algorithm, we build a table “DP” by storing values for amounts ranging from 1 to “amount” and accessing them when calculating change for higher values

DP[i] : minimum number of coins required to change i cents

DP[0] = 0, base case

DP[i] = $min_{1<=j<=n}$ DP[i−C[j] + 1 where i − C[j] ≥ 0 . C[j] is the j-th coin denomination

Proof of Correctness - by counter example

We can find coin denominations for any value as C contains 1 cent. Therefore DP[i] is defined for any value i>0.

To prove by contradiction, let us consider the smallest counter example X. The optimal solution requires y coins to change X cents. Let $c_1, c_2, ..., c_y$ be the coin denominations to represent X in the optimal solution, where $c_1 + c_2 + ... + c_y = X$ and $c_i \in C$ Let our example produce a result greater than y for the same value X.

=> DP[X]>y for our algorithm.

Since X is the smallest counter example, our algorithm $X - c_1$ cents will be correctly calculated by our algorithm, the difference being 1 coin

=> $DP[X - c_1] <= y - 1$

Since the difference between $X\ and\ X - c_1$ is a single coin c_1 ,

=> $DP[X] <= y$

This is a contradiction, therefore we can say that our algorithm produces the minimum coins needed to change X cents.

Time complexity
The i loop runs from 1 to amount and the j loop runs from 1 to number of denominations. Therefore, the worst case time complexity is O(amount*n), where n=number of denominations.

Space Complexity
Our approach using dynamic programming uses extra space. We store the minimum number of coins needed for every value <= amount. Therefore, the space complexity is O(amount).

e) (6 points) Implement the algorithm described in d). The code framework are given in the zip file: framework.zip. To avoid loss of marks please make sure that all provided test cases pass on remote-linux server by using the test file. Instructions for setting up remote-linux server and testing are given in the document HW3_Programming_Assignment_Setup.pdf.

Problem 2. **(10 points) In class we showed that multiplying two matrices**

C	**A * B**
m×p	**m×n n×p**

requires mnp scalar multiplications. You are given the following matrix chain:

A_1* * *A_2* * *A_3* * *A_4

20×25 25×5 5×10 10×30

$d_0 \times d_1$ $d_1 \times d_2$ $d_2 \times d_3$ $d_3 \times d_4$

a) (6 points) Fill the Table below with the missing values for *m[i,j]*. Also, for each *m[i,j]* put the corresponding value *k*, where the recurrence obtains its minimum value, next to it.

i\j	**1**	**2**	**3**	**4**
1	0	2500 (1)	3500 (2)	7000 (2)
2	Undefined	0	1250 (2)	5250 (2)
3	Undefined	Undefined	0	1500 (3)
4	Undefined	Undefined	Undefined	0

Use the table to answer the following questions:

b) (1 point) What is the minimum number of scalar multiplications required to compute *A_1 *A_2* **A_3* **A_4* ?**

Solution : 7000

c) (2 points) Give the optimal order of computing the matrix chain by fully parenthesizing the matrix chain below.

(A_1 * A_2) * (A_3 * A_4)

d) (1 point) How many scalar multiplications are used to compute (((A_1*A_2) * A_3) *A_4)? Keep the order of matrix multiplications indicated by the brackets. Justify your solution.

Solution:

$A_1 * A_2 = d_0 * d_1 * d_2 = 20 * 25 * 5 = 2500$

$(A_1 * A_2) * A_3 = d_{A_1 * A_2} * d_3 = 20 * 5 * 10 = 1000$

$((A_1 * A_2) * A_3) * A_4 = d_{(A_1 * A_2) * A_3} * d_4 = 20 * 10 * 30 = 6000$

Total scalar multiplication for $((A_1 * A_2) * A_3) * A_4 = 2500 + 1000 + 6000 = 9500$

Problem 3. Purpose: practice designing greedy algorithms. **(10 points) Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible.**

Describe a greedy algorithm for this problem. If the points are assumed to be sorted in increasing order your algorithm should run in time O(n). Be sure to describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

Solution:

Input : x1,x2,...,xn - set of houses in sorted order

d - maximum distance allowing reasonable reception.

Output : Points y1,y2,...,yk, such that for each i, there is at least one j with | yj - xi | ≤ d.

Algorithm -

We place a tower in such a way that it is as far away from the house as possible, but within d in order to cover maximum houses per tower. We place the tower at position y in such a way that all houses in [y-d,y+d] are covered by the tower at position y. If houses $x_1, ..., x_p$ are served by this tower, our problem reduces to a subproblem where we pick the next tower using the same logic, but for the reduced number of houses from $x_{p+1}, ..., x_n$. This becomes an iterative process.

1. Place the first tower at $x_1 + d$. This tower covers all houses in range $x_1 \; to \; x_1 + 2d$.
2. If p houses are covered under this tower, our problem gets reduced to finding minimum number of towers to cover the remaining houses $x_{p+1} \; to \; x_n$.
3. We continue this iteratively. Next, we place the next tower at $x_{p+1} + d$. This covers all the houses from $x_{p+1} + d \; to \; x_{p+1} + 2d$.
4. We continue this till all n houses are covered.

Psuedocode -

Let,
n - Number of houses
x - Array of house positions
y - Array of tower positions

t - Number of towers

```
1. y[1]=x[1]+d              # We initialize the first tower at position $x_1 + d$
2. t=1                      # Now we have 1 tower
3. for i=2 to n do          # For the rest of the houses,
       If | y[t] - x[i] | > d   # If house is beyond the range of current tower
           t=t+1            # Increase number of towers
           y[t]=x[i]+d      # By adding one at $x_i + d$
```

Example:

Input :
x = [3,4,9,14,18,19,22,24,26]
d = 3

- y1 = 3+3 =6 (Line 1)
- Houses covered by y1 = [3,4,9]
- Houses not covered by y1 = [14,23,19,31,32,15]

- y2 = 14+3 = 17
- Houses covered by y2 = [14,18,19]
- Houses not covered by y2 = [22,24,26]

- y3 = 22+3 = 25
- Houses covered by y2 = [22,24,26]
- Houses not covered by y2 = []

So we place the towers at position {6, 17,25}

Proof of Correctness - by contradiction

We are trying to find the most optimal positions to place towers such that a minimum number of towers cover all houses. Our algorithm places the tower at $x_i + d$, if a house is at x_i. Let us try to find other positions to place the tower to find a more optimal placement.

This implies that the tower can either be placed at position $x_i \leq t \leq x_i + d$.

- $t > x_i + d$
  We cannot place $t > x_i + d$, as this will result in t not covering x_i and will hence not lead to optimal placement.
- $t < x_i + d$.
  The greedy algorithm tries to place a tower in such a way that it covers as many houses as possible. Placing $t < x_i + d$, but as far away from x_i as possible so that house x_j is

also covered by the tower at *t*. Our greedy solution has a better chance of covering both x_i *and* x_j since it farther from x_i and hence much closer to x_j.

The same argument can be used for any house x_k and therefore, our algorithm works as good as the optimal algorithm.

Time complexity -

We loop from i=2 to n and we have one comparison of house with the tower per iteration. Therefore, we can say that the worst case time complexity is O(n).

Problem 4. Give a sequence of m MAKE-SET , UNION , and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(mlogn)$ time when we use union by rank only.

Solution:

Let us consider binomial trees to represent disjoint sets. We know that the height of a tree is $log_2 n$.

Looking at the properties of a binomial tree,

- A binomial tree has height k.
- The number of nodes is 2^k

Let us build a binomial tree of height $log_2 n$ using disjoint set operations. We have 3 such operations.

1. **MAKE-SET**
   We need n MAKE-SET operations to build the tree. Each MAKE-SET operation takes $\Theta(1)$ time.
2. **UNION**
   A binomial tree of height k has 2^k nodes. To build a binomial tree of height $log_2 n$ with $2^{log_2 n} = n$ nodes, we need $n - 1$ Union operations. Each operation takes $\Theta(log_2 n)$ time.
3. **FIND-SET**
   Since we have an overall of m operations, where MAKE-SET takes n operations and UNION takes (n-1) operations,
   the number of **FIND-SET** operations = $m - n - (n-1) = m - 2n + 1$ operations. Each operation takes $\Theta(log_2 n)$ time.

The overall time needed to perform the above operations is bounded below by the time taken by the **FIND-SET** operations, which is $\Omega((m - 2n + 1)(logn))$. If we provide sufficiently large values of m, such that $m-2n+1 \in \Omega(n)$, then we can say that $\Omega((m-2n+1)(logn)) = \Omega(mlogn)$