Memet Rasidovski
13095150

CP312 Assignment 4

Q1: Post office denominations do not provide optimal solutions because coin change algorithm requires that the set of coins

$$S = \{m_1, m_2, \ldots, m_K\} \quad m_1 = 1 \quad m_i < m_{i+1} \text{ for all } m$$

all values of m_{i+1} must be divisable by m_i for all i, $1 < i < K$

if a coin doesn't exists in set S then the max number of coins needed to equal this coin is $|m_j - 1|$.

So to equal m_j will require $\left(\frac{m_{i+1}}{m_i} - 1\right) m_i = m_j - 1$

if we do this for all values 1 to $j-1$ (minus one since we have no coin equal to total)

$$\left(\frac{m_2}{m_1} - 1\right) m_1 + \left(\frac{m_3}{m_2} - 1\right) m_2 + \ldots + \left(\frac{m_j}{m_{j-1}} - 1\right) m_{j-1}$$

This is the total number of coins required By this Lemma we can see all values of $\frac{m_{i+1}}{m_i}$ will equal $m_j - 1$

This means all coins that have a higher multiple of it denomination will go to zero and the lowest number of coins needed is $m_j - 1$

Q1: Thus since post office denominations contains 34 and 70 this will not ~~produce~~ always produce the optimal result.

ex: $\left(\frac{m_{i+1}}{m_i} - 1\right) m_i = m_j - 1$ $\quad m_j = 5^k$, $m_i = 5^\alpha$ $\Rightarrow \left(\frac{5}{1} - 1\right) 5 = 5 - 1$

So the best way to make m_j is with $4\ m_i$

We can see if we start from the right hand side of the formula that by always picking the largest coin value $m_j - 1$ we end up with the lowest total as long as $\frac{m_{i+1}}{m_i}$ holds.

CP312 Assignment 4

Q2: The problem has optimal substructure Property by the subproblem of picking two processes and inverting them to reduce the lateness.

Let S be a set of processes

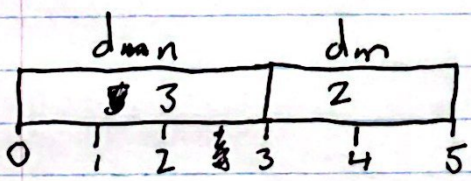
$$S = \{d_1, d_2, d_3, d \dots d_i\}$$

Picking two processes: d_m & d_n

Where d_m is due before d_n

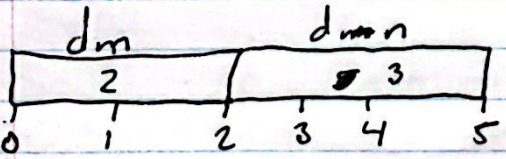
$$d_m \leq d_n$$

if by inverting the processes we can decrease the Lateness we solve the subproblem and do the same for all pairs of d. Then by sorting the entire set S by minimum process time the entire set will have minimum lateness.



Lateness equals 3

↓ inversion



Lateness equals 2

Q3: no. the problem cant be solved using dynamic programming. This is because the problem cannot be memoized. each comparison between processes will be different and require a calculation rather than a simple lookup.

Pseudocode

Q4:

```
nums = [ List of numbers ]
Plus = int of Plus signs
minus = int of minus signs

Sort (nums, descending_order)

Sum = nums.remove(0)

for i in nums:
    if Plus > 0:
        Sum += i
        Plus -= 1
    else:
        sum -= i
```

This approach produces the maximum by placing the largest number first. Then it will use the availble + signs to add the largest numbers. Then using the remaining - signs to subtract the smallest numbers or flip the largest negative numbers to positive.

~~Optimal Sub Structure~~

Optimal Solution

Supose S is not the optimal solution, then there must exist S' that is a more optimal solution.

for S' to be more optimal then there has to be a pair of numbers we can switch that will increase the total. these numbers are switch so n_1 is sub tracted and n_2 is now added if $n_1 > n_2$ there is no way to increase by switching thus contridicting the original statement. if $n_1 = n_2$ it contridicts because S will not change.

n_2 cannot be greater than n_1 because by our algorithm we always choose larger numbers to be added.

QS: Pseudocode — Memoized

```
coin = [list of coins]
total = desird total

func recur(total):
    if total == 0
        return 0

    ans = 1000 000 000

    for c in nums:
        if amount >= c:
            ans = min(ans, recur(total-c) + 1)
    return ans
```

recur(total)

The time complexity is $O(kN)$ where k is the total we are searching for

Q5: Pseudocode — Bottom-up

```
lst = {empty Hash table}
nums = [List of coins]
total = Desired total (int)

if (isSolvable() == True ):
    for i in range(Total):
        for j in ~~range~~ nums:
            lst[i] = 1000 000 000
            T = total

            ~~if~~ if i-j >= 0:
                lst[i] = min(lst[i], 1 + lst[i-j])

    return lst[total]

else:
    Print ("not solvable")
    return -1
```

The Bottom up approach produces a result of $O(kN)$ where k is the total. *.

Loop i is run k times
Loop j runs Len(nums) times