COL351
Assignment 2

Mallika Prabhakar (2019CS50440)
Sayam Sethi (2019CS10399)

September 2021

Contents

1 Question 1

Question 1

Question. *Alice, Bob, and Charlie have decided to solve all exercises of the Algorithms Design book by Jon Kleinberg, Éva Tardos. There are a total of n chapters, $[1, \ldots, n]$, and for $i \in [1, n]$, x_i denotes the number of exercises in chapter i. It is given that the maximum number of questions in each chapter is bounded by the number of chapters in the book. Your task is to distribute the chapters among Alice, Bob, and Charlie so that each of them gets to solve nearly an equal number of questions.*
Device a polynomial time algorithm to partition $[1, \ldots, n]$ into three sets S_1, S_2, S_3 so that $\max\{\sum_{i \in S_1} x_i, \sum_{i \in S_2} x_i, \sum_{i \in S_3} x_i\}$ is minimized.

Solution. We propose a *Dynamic Programming* solution for this problem. The idea is to generate all possible combinations of S_1, S_2, S_3 and then find the best combination of out them. The naïve solution will have an exponential complexity ($O(3^n)$) and hence it needs to be modified so that it can be executed in polynomial time complexity.
We make the following observations to optimise our solution:

1. To find the optimal parititon of S, only the sum of each of S_1, S_2, S_3 matters
2. Order of picking elements for each set doesn't affect the solution
3. Fixing the sum of S_1 and S_2 uniquely identifies the sum of S_3

Using these observations we come up with the following DP table:

$$\begin{aligned}
dp(i, s1, s2) &= dp(i-1, s1 - S[i], s2) \vee dp(i-1, s1, s2 - S[i]) \vee dp(i-1, s1, s2) \\
&\qquad \forall i \in \{1, \ldots, n\}; s1, s2 \in \{1, \ldots, sum(S)\} \\
dp(i, p, q) &= \perp, \ \forall i \in \{1, \ldots, n\}; p, q < 0 \\
dp(0, 0, 0) &= \top
\end{aligned} \tag{1}$$

where $dp(i, s1, s2)$ is $\top$ if we can generate atleast one partition using the first i elements such that any two partitions have sums $s1$ and $s2$

Claim 1.1. *The dp table generated using the Equation 1 is correct, i.e., $dp(i, s1, s2) = \top$ iff there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$*

Proof. We will prove the correctness of the claim by induction on i.
Base case: $i = 0$
$dp(0, s1, s2)$ is $\top$ only when $s1 = s2 = 0$ and $\perp$ otherwise. We know that we can generate only three empty sets using the first 0 elements and thus their sums will all be 0. Therefore the base case is true.
Induction step: Assume that the claim is true for $i - 1$. Consider $dp(i, s1, s2)$,
The i^{th} element will be present in exactly one of S_1, S_2, S_3. Therefore, we have three cases:

1. $S[i]$ is in S_1, then the sum of S_1 upto the first $i - 1$ elements will be $s1 - S[i]$ and the sum of the other two sets doesn't change

2. $S[i]$ is in S_2, then the sum of S_2 upto the first $i-1$ elements will be $s2 - S[i]$ and the sum of the other two sets doesn't change

3. $S[i]$ is in S_3, then the sum of S_3 upto the first $i-1$ elements will be $sum(S[1:(i-1)]) - (s1 + s2)$ and the sum of the other two sets doesn't change

Thus, the only possibilities for valid partition sums using the first i elements are exactly those when we can generate atleast one of the above three paritions using the first $i-1$ elements. The transition equation given in Equation 1 exactly captures this. We have shown that for $dp(i, s1, s2)$ to be $\top$, atleast one of $dp(i-1, s1-S[i], s2), dp(i-1, s1, s2-S[i]), dp(i-1, s1, s2)$ must be $\top$. From induction, we know that the $(i-1)^{\text{th}}$ row of the table is true iff there exists a valid parititon. Therefore, if $dp(i, s1, s2) = \top$ then there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$. ($\implies$)

We still have to prove the converse, i.e., if there exists a partition using the first i elements with sums $s1, s2, sum(S[1:i]) - (s1 + s2)$, then $dp(i, s1, s2) = \top$.
To prove this, we observe that the dp table considers all possible sums since $s1, s2$ iterate in the range $\{1, \ldots, sum(S)\}$. Therefore, if there exists a valid solution, the dp table considers it and is assigned $\top$. Else it is assigned $\perp$. ($\impliedby$) $\square$

Now that we have proved the correctness of Equation 1, we present an algorithm for computing the same:

Algorithm 1 DP solution for partitioning

procedure PARTITION(S)
    $n \leftarrow size(S)$
    $s \leftarrow sum(S)$
    $dp \leftarrow$ table of size $(n+1) \times (s+1) \times (s+1)$ initialised with $\perp$
    $dp(0, 0, 0) \leftarrow \top$
    for i in $[1, n]$ **do**
        for $s1$ in $[0, s]$ **do**
            for $s2$ in $[0, s]$ **do**
                $dp(i, s1, s2) \leftarrow dp(i-1, s1, s2)$
                if $s1 \geq S[i]$ **then**
                    $dp(i, s1, s2) \leftarrow dp(i, s1, s2) \vee dp(i-1, s1 - S[i], s2)$
                end if
                if $s2 \geq S[i]$ **then**
                    $dp(i, s1, s2) \leftarrow dp(i, s1, s2) \vee dp(i-1, s1, s2 - S[i])$
                end if
            end for
        end for
    end for

19: $bestPair \leftarrow (-1, -1)$
20: $leastMax \leftarrow \infty$
21: **for all** $(s1, s2) \in [0, s] \times [0, s]$ **do**
22: **if** $dp(n, s1, s2) = \top$ and $\max(s1, s2, s-(s1+s2)) < leastMax$ **then**
23: $leastMax \leftarrow \max(s1, s2, s-(s1+s2))$
24: $bestPair \leftarrow (s1, s2)$
25: **end if**
26: **end for**
27: **return** $Backtrack(dp, S, bestPair)$ ▷ return the partition after backtracking on the DP table
28: **end procedure**

The `Backtrack` routine used in the last step generates the partitions which give the optimal values of the sum. It is given as:

Algorithm 2 Backtracking to generate parition

procedure BACKTRACK$(dp, S, (s1, s2))$
$n \leftarrow size(S)$
$S_1 \leftarrow \{\}$
$S_2 \leftarrow \{\}$
$S_3 \leftarrow \{\}$
for i in $[n, 1]$ **do**
if $s1 \geq S[i]$ and $dp(i-1, s1-S[i], s2) = \top$ **then**
$S_1 \leftarrow add(S_1, i)$
$s1 \leftarrow s1 - S[i]$
else if $s2 \geq S[i]$ and $dp(i-1, s1, s2-S[i]) = \top$ **then**
$S_2 \leftarrow add(S_2, i)$
$s2 \leftarrow s2 - S[i]$
else
$S_3 \leftarrow add(S_3, i)$
end if
end for
assert $(s1 = 0$ and $s2 = 0)$
return (S_1, S_2, S_3)
end procedure

We have shown the correctness of the DP table in Claim 1.1, therefore, to prove correctness of Algorithm 1, we need to show the correctness of the `for` loop from lines $21-26$ and Algorithm 2. The `for` loop iterates over all possible states of $(s1, s2)$ and finds the best state out of the valid states (where $dp(i, s1, s2) = \top$). Since the states explored by the `for` loop are exhaustive, the optimal solution is selected.
To prove the correctness of Algorithm 2, we notice that $dp(i, s1, s2)$ is $\top$ only if there exists a valid partitioning. Therefore, when updating $i, s1, s2$ in the `for` loop, we move from one valid partition to another. Therefore, Algorithm 2 generates the correct partitioning. This

completes the proof of correctness for Algorithm 1.

Space complexity: We create a *dp* table of size $(n+1) \times (s+1) \times (s+1)$ in Algorithm 1 and we generate the sets S_1, S_2, S_3 which have a total size of n in Algorithm 2. We use constant space everywhere else. Therefore the total space complexity of the solution is

$$O(n \times s \times s) + O(n) = O(n \times s^2) = O(n \times (n \times \max(S))^2) = O(n \times (n \times n)^2) = O(n^5) \quad (2)$$

Time complexity: We run nested `for` loop in Algorithm 1 (lines $6 - 18$) having a total of $n \times (s+1) \times (s+1)$ iterations and each iteration taking $O(1)$ time. We run another `for` loop (lines $21 - 26$) which has $(s+1) \times (s+1)$ iterations and each iteration again takes $O(1)$ time. We run a `for` loop of n iterations in Algorithm 2 and each iteration takes $O(1)$ time (*add* can be implemented in $O(1)$ using a list or vector). All other operations take $O(1)$. Therefore the total time complexity is given by:

$$O(n \times s \times s) + O(s \times s) + O(n) = O(n \times s^2) = O(n^5) \quad (3)$$

We have thus obtained a polynomial time algorithm for parititoning the given set S into S_1, S_2, S_3 such that $\max\{\sum_{i \in S_1} x_i, \sum_{i \in S_2} x_i, \sum_{i \in S_3} x_i\}$ is minimized. $\square$

2 Question 2

2.1 2.1

2.1

Question. *insert question*

Solution. Proof is left as an exercise. □

2.2 2.2

2.2

Question. *insert question*

Solution. Proof is left as an exercise. □

2.3 2.3

2.3

Question. *insert question*

Solution. Proof is left as an exercise. □

3 Question 3

3.1 3.1

3.2 3.2

4 Question 4

4.1 4.1

4.1

Question. *You are given a set of k denominations. Devise a polynomial time algorithm to count the number of ways to make change for Rs.n, given an infinite amount of coins/notes of denominations, $d[1], ..., d[k]$.*

Solution. The assumptions made are that the number of coins of every denomination are infinite and they are integral values.
We solved this problem using dynamic programming. Given the cost n and array of possible denominations $denom$ with size k, we create $dpTable$ which is an $(n+1)$ array. $dpTable[i]$ counts the number of ways to generate value i using the given denominations. The answer is obtained by observing value of the last element $dpTable[n]$.

Algorithm 3 Find total possible combinations of denominations to achieve value of n

```
procedure COMBINATIONS(denom, n)
    k ← size(denom)                              ▷ number of types of denominations
    dpTable ←1D-zero array of size (n + 1)
    dpTable[0] ← 0                    ▷ there is trivially one way to generate sum 0
    for i in [1, n + 1) do
        for j in [1, k + 1) do
            if i ≥ denom[j] then          ▷ denomination should not be greater than i
                dpTable[i] ← dpTable[i] + dpTable[i − denom[j]]
            end if
        end for
    end for
    return dpTable[n][k]
end procedure
```

□

Proof of correctness. □

Proof of termination. Here, we have a finite table of size $(n + 1)$. We iterate through the entire table and exit successfully in any case. Hence the algorithm terminates. □

Time Complexity. Deciding factors for time-complexity in big-Oh notation are going through the entire $dpTable$ and running a for loop with k iterations at each index of the table. Time complexity $= O(n \times k)$
This is a polynomial time solution. □

Space Complexity. We create a $dpTable$ of size $n+1$ and use constant space everywhere else.
Space complexity $= O(n)$ □

4.2 4.2

4.2

Question. *You are given a set of k denominations. Device a polynomial time algorithm to find a change of Rs.n using the minimum number of coins.*

Solution. The assumptions made are that the number of coins of every denomination are infinite and they are integral values.

Algorithm 4 Find total possible combinations of denominations to achieve value of n

```
procedure LeastCurr(denom, n)
    k ← size(denom)                                ▷ number of types of denominations
    dpArr ← array of size n + 1 initialised with ∞
    dpArr[0] ← 0                                   ▷ Base case: no coin needed n = 0
    for index in [1, n + 1] do
        for i in [0, k) do
            if index − denom[i] ≥ 0 then
                dpArr[index] ← min(dpArr[index], dpArr[index − denom[i]] + 1)
            end if
        end for
    end for
    return dpArr[n]
end procedure
```

□

Proof of correctness. □

Proof of termination. We iterate through the entire array of size n and exit successfully in any case. Hence the algorithm terminates. □

Time Complexity. Deciding factors for time-complexity in big-Oh notation are going through the entire *dpArray* and running a for loop of k iterations for each index.
Time complexity $= O(n \times k)$
This is a polynomial time solution. □

Space Complexity. We create a *dpArray* of size $n+1$ and use constant space everywhere else.
Space complexity $= O(n)$ □