ShanghaiTech University

CS101 Algorithms and Data Structures

Fall 2021

Homework 11

Due date: 23:59, December 19, 2021

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL NAME to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero grade.
8. In this homework, all the algorithm design part need the three part proof. The demand is in the next page. If you do not use the three part proof, you will not get any point.
9. In the algorithm design problem, you should design the correct algorithm whose running time is equal or smaller than the correct answer. If it's larger than the correct answer, you cannot get any point.

Demand of the Algorithm Design

All of your algorithm should need the three-part solution, this will help us to score your algorithm. You should include **main idea, proof of correctness and run time analysis.** The detail is as below:

1. The **main idea** of your algorithm. You should correctly convey the idea of the algorithm in this part. It does not need to give all the details of your solutions or why it is correct. For example, in the dynamic programming, you should define a function $f(\cdot)$ in words, including how many parameters are and what they mean, and tell us what inputs you feed into f to get the answer to your problem. Then, write the base cases along with a recurrence relation for f. If you do a good job here, the readers are more likely to be forgiving of small errors elsewhere.

   You can also include the **pseudocode** in the answer, but this is not necessary. The purpose of pseudocode is to communicate concisely and clearly, so think about how to write your pseudocode to convey the idea to the reader. Note that pseudocode is meant to be written at a high level of abstraction. Executable code is not acceptable, as it is usually too detailed. Providing us with working C code or Java code is not acceptable. The sole purpose of pseudocode is to make it easy for the reader to follow along. Therefore, pseudocode should be presented at a higher level than source code (source code must be fit for computer consumption; pseudocode need not). Pseudocode can use standard data structures. For instance, pseudocode might refer to a set S, and in pseudocode you can write things like "add element x to set S." That would be unacceptable in source code; in source code, you would need to specify things like the structure of the linked list or hashtable used to store S, whereas pseudocode abstracts away from those implementation details. As another example, pseudocode might include a step like "for each edge $(u, v) \in E$", without specifying the details of how to perform the iteration.

2. A **proof of correctness**. You must prove that your algorithm work correctly, no matter what input is chosen. For iterative or recursive algorithms, often a useful approach is to find an invariant. A loop invariant needs to satisfy three properties: (1) it must be true before the first iteration of the loop; (2) if it is true before the ith iteration of the loop, it must be true before the $i + 1$st iteration of the loop; (3) if it is true after the last iteration of the loop, it must follow that the output of your algorithm is correct. You need to prove each of these three properties holds. Most importantly, you must specify your invariant precisely and clearly. If you invoke an algorithm that was proven correct in class, you don't need to re-prove its correctness.

3. The asymptotic **running time** of your algorithm, stated using $O(\cdot)$ notation. And you should have your **running time analysis**, i.e., the justification for why your algorithm's running time is as you claimed. Often this can be stated in a few sentences (e.g.: "the loop performs $|E|$ iterations; in each iteration, we do $O(1)$ Find and Union operations; each Find and Union operation takes $O(\log |V|)$ time; so the total running time is $O(|E| \log |V|)$"). Alternatively, this might involve showing a recurrence that characterizes the algorithm's running time and then solving the recurrence.

0: Three Part Proof Example

Given a sorted array A of n (possibly negative) distinct integers, you want to find out whether there is an index i for which $A[i] = i$. Devise a divide-and-conquer algorithm that runs in $O(\log n)$ time.

Main idea:

To find the i, we use binary search, first we get the middle element of the list, if the middle of the element is k, then get the i. Or we seperate the list from middle and get the front list and the back list. If the middle element is smaller than k, we repeat the same method in the back list. And if the middle element is bigger than k, we repeat the same method in the front list. Until we cannot get the front or the back list we can say we cannot find it.

Algorithm 1: Binary Search(A)

```
1  low ← 0;
2  higt ← n − 1;
3  while low < hight do
4  |  mid ← (low + high)/2;
5  |  if k == A[mid] then
6  |  |  return mid;
7  |  else if k > A[mid] then
8  |  |  low ← mid + 1;
9  |  else
10 |  |  high ← mid − 1
11 |  end if
12 end while
13 return −1
```

Proof of Correctness:

Since the list is sorted, and if the middle is k, then we find it. If the middle is less than k, then all the element in the front list is less than k, so we just look for the k in the back list. Also, if the middle is greater than k, then all the element in the back list is greater than k, so we just look for the k in the front list. And when there is no back list and front list, we can said the k is not in the list, since every time we abandon the items that must not be k. And otherwise, we can find it.

Running time analysis:

The running time is $\Theta(\log n)$.

Since every iteration we give up half of the list. So the number of iteration is $\log_2 n = \Theta(\log n)$.

1: (10') TENET

A TENET sequence is a nonempty string over some alphabet that reads the same forward and backward. For example, "civic", "bbbb" and all strings of length 1 are all TENET sequence. In this question, we want to find the longest TENET sequence that is a subsequence of a given input string. For example, given the input "character", your algorithm should return 5 (or "carac"). Note that the subsequence is different from substring, where subsequence may not be consecutive.

Give an algorithm using dynamic programming, prove your algorithm and show the running time complexity of your algorithm.

Hint: the running time complexity of your algorithm shouldn't be worse than $O(n^2)$

Main Idea:

Clarification. Let S be a string of length l. Let $S[i]$ be the i^{th} character of the string. The index of the strings start at 1.

Def. $OPT(i,j)$: max length of TENET sequence between the i^{th} character and the j^{th} charcter.

Goal. $OPT(1,l)$: max length of TENET sequence of the string (the longest TENET sequence of the string).

Bellman Equation.

$$OPT(i,j) = \begin{cases} 0, & i > j \\ 1, & i = j \\ OPT(i+1, j-1) + 2, & i < j \text{ and } S[i] = S[j] \\ \max(OPT(i+1,j), OPT(i,j-1)), & i < j \text{ and } S[i] \neq S[j] \end{cases}$$

The base case is

$$OPT(i,j) = \begin{cases} 0, & i > j \\ 1, & i = j \end{cases}$$

Pseudo Code:

Algorithm 2: TENET

input : a string S of size l

output: the length of the longest TENET sequence

```
Intialize OPT by an n × n matrix with all entries equal to 0, index start at 1.
for i ← l to 1 do
    // Base case
    OPT[i][i] ← 1
    for j ← i + 1 to l do
        if S[i] = S[j] then
            OPT[i][j] ← 2 + OPT[i + 1][j − 1]
        else
            OPT[i][j] ← Max(OPT[i + 1][j], OPT[i][j − 1])
        end if
    end for
end for
return OPT[1][l]
```

Proof of Correctness:

Optimality:

case 1. $i > j$.

There is no chacracters between i and j when $i > j$, so $OPT(i, j) = 0$.

case 2. $i = j$.

The character between i and j is $S[i] = S[j]$, so $OPT(i, j) = 1$.

case 3. $i < j$ and $S[i] = S[j]$.

This means $S[i]$ can be the first chcaracter and $S[j]$ can be the last character of the TENET sequence simultaniously. Then we choose both of them. The new TENET is the concat of $S[i]$, longest TENET sequence from $i + 1$ to $j - 1$ and $S[j]$. The length is $2 + OPT(i + 1, j - 1)$.

Suppose $OPT(i, j)$ is not optimal, there must exists a TENET sequence T' in $[i, j]$ not include both $S[i]$ and $S[j]$. The length of T' $OPT'(i, j) > OPT(i, j)$

If $OPT'(i, j) = 1$, then $OPT'(i, j) = OPT(i, i) = 1$, $OPT(i, j) = 1$. This is contradict to that $OPT'(i, j) > OPT(i, j)$.

If $OPT'(i, j) > 1$, W.L.O.G., suppose T' choose $S[i]$ but not choose $S[j]$. There exist $k, i < k < j$ that $S[i] = S[k]$ and $S[k]$ is the last character of T'. Then we can change $S[k]$ to $S[j]$, and $OPT(i, j)$ is equal to $OPT'(i, j)$ and thus is optimal. This is contradict to that $OPT(i, j)$ is not optimal.

Therefore, when $S[i] = S[j]$, $OPT(i, j)$ is optimal.

case 4. $i > j$ and $S[i] \neq S[j]$.

This means $S[i]$ and $S[j]$ can not be the first and last character respectively on the same time. The longest TENET sequence should be either the longest TENET sequence excluding $S[i]$, the one excluding $S[j]$, or the one exclude both. That is either the $OPT(i, j-1)$, the $OPT(i+1, j)$, or the $OPT(i+1, j-1)$. The last case is included in $OPT(i+1, j)$ and $OPT(i, j-1)$. Therefore, $OPT(i, j) = \max(OPT(i+1, j), OPT(i, j-1))$ is optimal.

Correctness:

Base.

$OPT(i, j) = 0$ for $j - i < 0$ and $OPT(i, j) = 1$ for $j - i = 1$. Therefore the recurrence will break down when $j - i \leq 0$.

Topological order.

All the subproblem $OPT(i, j)$ depend on strictly smaller $j - i$, so the subproblmes form a DAG.

All the subproblem $OPT(i, j)$ depend on $OPT(a, b)$ where $a \geq i, b \leq j, (a, b) \neq (i, j)$.

Since i is iterate down from n to 1 and j is iterate up from $i + 1$ to n. When calculating $OPT[i][j]$, all $OPT[a][b]$ with $a \geq i, b \leq j, (a, b) \neq (i, j)$ has been calculated. Therefore, the iteration order is a valid topological order of the subproblem DAG.

Time Complexity:

The for loop cost $O(n^2)$ in total. Therefore the time complexity is $O(n^2)$

2: (10') How many expressions?

You are given a list of non-negative integers $L = [l_1, ..., l_n]$ with length n, and a value w.
From the list L, you can construct an expression in the following way:

- attach a symbol '+' or '-' before each number in L
- concatenate all the numbers to get the final expression

Your task is to determine the number of different expressions that you could construct from L, which evaluates to the target value w.

For example, let L = [5, 6], $w = -1$, the expression "+5-6" will evaluate to w.

Give a dynamic programming algorithm to solve this problem, prove your algorithm and show the running time complexity.

Hint: You can try to convert it into the problem of finding the number of different subsets in L that sums to a particular target value.

Main Idea:

The problem can be converted into finding the number of different subsets in L that sums to

$$c = \frac{w + \sum_{i=1}^{n} l_i}{2}$$

Clarification. Let $H = \sum_{i=1}^{n} l_i$

Def. $OPT(i, h)$: the number of different subsets in $[l_1, \cdots, l_i]$ sums to h, h must be integer.

Goal. $OPT(n, c)$: the number of different subsets in L sums to c.

Bellman Equation.

$$OPT(i, h) = \begin{cases} 1, & i = 0 \text{ and } h = 0 \\ 0, & i = 0 \text{ and } h \neq 0 \\ 0, & h > H \text{ or } h < 0 \\ OPT(i-1, h) + OPT(i-1, h-l_i), & i > 0 \text{ and } h \leq H \end{cases}$$

The base case is

$$OPT(i, h) = \begin{cases} 1, & i = 0 \text{ and } h = 0 \\ 0, & i = 0 \text{ and } h \neq 0 \\ 0, & h > H \text{ or } h < 0 \end{cases}$$

Pseudo Code:

Algorithm 3: CountExpressions

input : a list of integers L of size n, the aimed value w

output: the number of the expressions

```
c ← w + Sum(L)
if c ∤ 2 or c > 2 × Sum(L) then
    // Impossible to get the the value.
    return 0
end if
c ← c/2
Intialize OPT by an (n + 1) × (c + 1) matrix with all entries equal to 0, index start at 0.
// Base case
OPT[0][0] ← 1
for i ← 1 to n do
    for h ← L[i] to c do
        OPT[i][h] ← OPT[i − 1][h − L[i]]
    end for
end for
return OPT[n][c]
```

Proof of Correctness:

Correctness of Conversion:

Let $S = [l_{s_1}, \cdots, l_{s_k}]$ be the list of numbers have '+' attached before. $U = [l_{u_1}, \cdots, l_{u_m}]$ be the list of numbers have '−' attached before. We have

$$\sum_{t=1}^{n} l_t = \sum_{t=1}^{k} l_{s_t} + \sum_{t=1}^{m} l_{u_t}$$

The original question is to find the number of S and U such that

$$\sum_{t=1}^{k} l_{s_t} - \sum_{t=1}^{m} l_{u_t} = w$$

Adding $\sum_{t=1}^{n} l_t$ to both sides

$$\begin{aligned}
\sum_{t=1}^{n} l_t + \sum_{t=1}^{k} l_{s_t} - \sum_{t=1}^{m} l_{u_t} &= w + \sum_{t=1}^{n} l_t \\
\sum_{t=1}^{k} l_{s_t} + \sum_{t=1}^{m} l_{u_t} + \sum_{t=1}^{k} l_{s_t} - \sum_{t=1}^{m} l_{u_t} &= w + \sum_{t=1}^{n} l_t \\
2\sum_{t=1}^{k} l_{s_t} &= w + \sum_{t=1}^{n} l_t \\
\sum_{t=1}^{k} l_{s_t} &= \frac{w + \sum_{i=1}^{n} l_i}{2}
\end{aligned}$$

Therefore, the question is equal to find the number of subsets S of L that sum to

$$c = \frac{w + \sum_{i=1}^{n} l_i}{2}$$

Correctness of algorithm:

Since all the elements in L is integer, if c is not an integer, there's no solution to get the value. If c is bigger than the sum of all elements in$L$, it is also impossible to get the value. If c is less than 0, it's also impossible to get the solution since all the elments in L is positive.

Base case.

If $i = 0$, $S = \emptyset$ and the sum of the elements in S is 0. Only when $h = 0$, $OPT(i, h) = 1$. Otherwise, $OPT(i, h) = 0$. Therefore $OPT(0, h)$ is the number of different subsets in $[l_1, \cdots, l_i] = \emptyset$ sums to h.

Induction hypothesis.

Supose for $i = k$, $OPT(i, h)$ is the number of different subsets in $[l_1, \cdots, l_i]$ sums to h.

Induction steps.

When $i = k + 1$, there are two cases.

Case 1. Select l_i to add to h. Then the answer should be equal to the number of different subsets in $[l_1, \cdots, l_{i-1}]$ that sums to $h - l_{i-1}$.

Case 2. Not select l_i to add to h. Then the answer should be equal to the number of different subsets in $[l_1, \cdots, l_{i-1}]$ that sums to$h$.

The total number should be the sum of tow cases. Therefore, $OPT(i, h) = OPT(i-1, h-l_i) + OPT(i-1, h)$

Time Complexity:

The time complexity of Sum function is $O(n)$. The time complexity of the for loop is $O(nc)$. Therefore, the time complexity is $O(nc)$. Where

$$c = \frac{w + \sum_{i=1}^{n} l_i}{2}$$

3: (15') Greedy doesn't work

Tom and Jerry are playing an interesting game, where there are n cards in a line. All cards are faced-up and the number on every card is between 2-9. Tom and Jerry take turns. In anyone's turn, they can take one card from either the right end or the left end of the line. The goal for each player is to maximize the sum of the cards they have collected.

(a) Tom decides to use a greedy strategy: "on my turn, I will take the larger of the two cards available to me." Show a small counterexample ($n \leq 5$) where Tom will lose if he plays this greedy strategy, assuming Tom goes first and Jerry plays optimally, but he could have won if he had played optimally.

(b) Jerry decides to use dynamic programming to find an algorithm to maximize his score, assuming he is playing against Tom and Tom is using the greedy strategy from part (a). Help Jerry to develop the dynamic programming solution.

(a)
Consider the following card sequence:

3, 9, 4, 2

If Tom played greedy and Jerry played optimally, the game goes as follows.

(t1) Tom will take 3. Sequence: 9, 4, 2; Tom: 3; Jerry: 0;

(t2) Jerry will play optimally. Because $9 > 4 + 2$, Jerry must take 9. Sequence: 4, 2; Tom: 3; Jerry: 9;

(t3) After that, Tom will take 4, and left 2 for Jerry. Sequence: 2; Tom: 7; Jerry: 9;

(t4) Jerry take 2. Tom:7; Jerry: 11. Jerry won.

Then Tom will play use another strategy S' and there exist an optimal strategy S. Let $T'(L)$ be the sum gained under strategy S' and T be the sum gained under strategy S. Since S is optimal, $T \geq T'$. If Tom played under S' and Jerry played optimal, the game goes as follows:

(t1) Tom take 2. Sequence: 3, 9, 4; Tom: 2; Jerry: 0;

(t2) If Jerry take 4, then Tom will take 9 next trun. If Jerry take 3, then Tome will also take 9 next turn. Take either end will produce same result, so both the decision at this stage is optimal. Let Jerry take 3. Sequence: 9, 4; Tom: 2; Jerry: 3;

(t3) Tom take 9. Sequence: 4; Tom: 11; Jerry: 3.

(t4) Jerry take 4. Tom: 11; Jerry: 7. Tom won.

$T' = 11$. Since $T \geq T' = 11$. If Tom play optimally, Tom will also win. Therefore, in this example, if Tom use greedy, he will lose; but if he use optimal strategy, he will win.
(b)
Main Idea:

Clarification. Let S be the total sum of all the cards. If one player wins, he must cellect cards whose sum bigger than $\frac{S}{2}$. Let a_i be the number of i^{th} card.

Def. $\mathrm{OPT}(l, r)$: the max sum the player can collect if the uncollected cards is from l^{th} to r^{th} if player takes turn.

Goal. $\mathsf{OPT}(1, n)$: the max sum the player can collect if the uncellected cards is from 1^{th} ton^th (the initial cards sequence, because Jerry plays first) when Jerry takes turn.

Bellman Equation.

$$\mathsf{OPT}(l, r) = \begin{cases} 0, & l > r \\ a_l, & l = r \\ \max\{a_l + \mathsf{OPT}(l+1, r-1), a_r + \mathsf{OPT}(l, r-2)\}, & l < r \text{ and } a_{l+1} < a_r \text{ and } a_l < a_{r-1} \\ \max\{a_l + \mathsf{OPT}(l+1, r-1), a_r + \mathsf{OPT}(l+1, r-1)\}, & l < r \text{ and } a_{l+1} < a_r \text{ and } a_l \geq a_{r-1} \\ \max\{a_l + \mathsf{OPT}(l+2, r), a_r + \mathsf{OPT}(l, r-2)\}, & l < r \text{ and } a_{l+1} > a_r \text{ and } a_l < a_{r-1} \\ \max\{a_l + \mathsf{OPT}(l+2, r), a_r + \mathsf{OPT}(l+1, r-1)\}, & l < r \text{ and } a_{l+1} > a_r \text{ and } a_l \geq a_{r-1} \end{cases}$$

The base case is

$$\mathsf{OPT}(l, r) = \begin{cases} 0, & l > r \\ a_l, & l = r \end{cases}$$

Pseudo Code:

Algorithm 4: GreedyDontWork

input : a list of numbers of the card A of size n

output: the maximun number Jerry can get.

```
// To Avoid overflow, assign extra space for OPT
Intialize OPT by an (n+2) × (n+2) matrix with all entries equal to 0, index start at 0.
// Base case
for i ← 1 to n do
    OPT[i][i] ← A[i]
end for
for length ← 2 to n do
    for l ← 1 to n − length + 1 do
        r ← l + length − 1
        // If Jerry select left card,
        // Tom will select cards depending on A[l + 1] and A[r]
        if A[l + 1] < A[r] then
            // Tom takes right
            left ← l
        else
            // Tom takes left
            left ← l + 1
        end if
        // If Jerry select right card,
        // Tom will select cards depending on A[l] and A[r − 1]
        if A[l] < A[r − 1] then
            // Tom takes right
            right ← r − 1
        else
            // Tom takes left
            right ← r
        end if
        // If Jerry takes left, increase the left pointer by 1.
        // If Jerry takes right, decrease the right pointer by 1.
        OPT[l][r] ← Max(A[l] + OPT[left + 1][right], A[r] + OPT[left][right − 1])
    end for
end for
return OPT[1][n]
```

Proof of Correctness:

We will proove $OPT(l, r)$ is optimal by mathemathical induction. Let the length of the sequence be $s = r - l + 1$.

Base case. When $s = 0$, $OPT(l, r) = 0$. When $s = 1$, there's only one card to choose, $OPT(l, r) = a_l$.

Induction Hypothesis. Suppose when $s \leq k$, $OPT(l, r)$ is optimal.

Inducntion steps. When $s = k + 1$, $r - l + 1 = k + 1$. There are two choice:

(1) Jerry choose left, namely a_l. Then the rest cards are from $l+1$ to r. There are two cases:

   (a) $a_{l+1} < a_r$. Tom use greedy, and he will choose a_r. The rest cards will be from $l+1$ to $r-1$, and $s' = s-2 = k-1 < k$. Therefore, by *induction hypotehsis*, $\mathsf{OPT}(l+1, r-1)$ is optimal.

   (b) $a_{l+1} >= a_r$. Tom will choose a_{l+1}. The rest cards will be from $l+2$ to r, and $s' = s-2 = k-1 < k$. Therefore, by *induction hypotehsis*, $\mathsf{OPT}(l+2, r)$ is optimal.

   Therefore, if Jerry choose left. The maximun Jerry can get is

$$\mathsf{OPT}_{\mathtt{left}}(i,j) = \begin{cases} a_l + \mathsf{OPT}(l+1, r-1), & a_{l+1} < a_r \\ a_l + \mathsf{OPT}(l+2, r), & a_{l+1} \geq a_r \end{cases}$$

(2) Jerry choose right, namely a_r. Then the rest cards are from l to$r-1$. There are also two cases:

   (a) $a_l < a_{r-1}$. Tom use greedy, and he will choose a_{r-1}. The rest cards will be from l to $r-2$, and $s' = s-2 = k-1 < k$. Therefore, by *induction hypotehsis*, $\mathsf{OPT}(l, r-2)$ is optimal.

   (b) $a_l \geq a_{r-1}$. Tom use greedy, and he will choose a_l. The rest cards will be from $l+1$ to $r-1$, and $s' = s-2 = k-1 < k$. Therefore, by *induction hypotehsis*, $\mathsf{OPT}(l+1, r-1)$ is optimal.

   Therefore, if Jerry choose right. The maximun Jerry can get is

$$\mathsf{OPT}_{\mathtt{right}}(i,j) = \begin{cases} a_l + \mathsf{OPT}(l, r-2), & a_l < a_{r-1} \\ a_l + \mathsf{OPT}(l+1, r-1), & a_l \geq a_{r-1} \end{cases}$$

The algorithm choose the bigger one of the two cases. $\mathsf{OPT}(i,j) = \max(\mathsf{OPT}_{\mathtt{left}}(i,j), \mathsf{OPT}_{\mathtt{right}}(i,j))$. Therefore, $\mathsf{OPT}(i,j)$ is not less than any decision, therefore is optimal.

Time Complexity:

The base case initializaiton took $O(n)$. The for-loop took $O(n^2)$. So the time complexity is $O(n^2)$.

4: (10') Counting Targets

We call a sequence of n integers $x_1, ..., x_n$ valid if each x_i is in $\{1, ..., m\}$.

(a) Give a dynamic programming-based algorithm that takes in n, m and "target" T as input and outputs the number of distinct valid sequences such that $x_1 + ... + x_n = T$. Your algorithm should run in time $O(m^2n^2)$.

(b) Give an algorithm for the problem in part (a) that runs in time $O(mn^2)$.
Hint: let $f(s, i)$ denotes the number of length-i valid sequences with sum equal to s. Consider defining the function $g(s, i) := \sum_{t=1}^{s} f(t, i)$.

(a)

Main Idea:

Def. $f(s, i)$: the number of length-i valid sequences with sum to s.

Goal. $f(T, n)$: the number of length-n valid sequence with sum to T.

Bellman Equation.

$$f(s,i) = \begin{cases} 0, & s \le 0 \text{ and } i \neq 0 \\ 0, & s \neq 0 \text{ and } i = 0 \\ 1, & s = 0 \text{ and } i = 0 \\ \sum_{k=1}^{m} f(s-k, i-1), & s > 0 \text{ or } i > 0 \end{cases}$$

The base case is

$$f(s,i) = \begin{cases} 0, & s \le 0 \text{ and } i \neq 0 \\ 0, & s \neq 0 \text{ and } i = 0 \\ 1, & s = 0 \text{ and } i = 0 \end{cases}$$

Pseudo Code:

Algorithm 5: CountExpressions

input : an integer n, and an integer T

output: the number of the sequence

```
// T is bigger than the possible value.
if T > n * m then
    return 0
end if
Intialize f by an (T + 1) × (n + 1) matrix with all entries equal to 0, index start at 0.
// Base case
f[0][0] = 1
for i ← 1 to n do
    for s ← 1 to T do
        for k ← 1 to m do
            if s − k ≥ 0 then
                f[s][i] ← f[s][i] + f[s − k][i − 1]
            end if
        end for
    end for
end for
return f[T][n]
```

Proof of Correctness:

We will prove that our algorithm is correct. *Base case.* When $i = 0$, only when $s = 0$, sum of 0 integers equals to0. Therefore, $f(0, 0) = 1$ and $f(0, s) = 0$.

Induction Hypothesis. Suppose when $i = k$, $f(s, i)$ is correct.

Induction Steps. When $i = k + 1$. x_i can vary from 1 to$m$. Let $x_i = j, 1 \le j \le m$. For any target value s

$$x_1 + \cdots + x_i = s$$

Substract x_i from each sides.

$$x_1 + \cdots x_{i-1} = s - x_i = s - j$$

That is to find a sequence of integers $x_1, \cdots, x_{i-1}$ sums to $s - j$. By *induction hypothesis*, $f(s - j, i - 1)$ is the correct answer to this question. Therefore,

$$f(s, i) = \sum_{j=1}^{m} f(s - j, i - 1)$$

is correct.

Time Complexity:

The for-loop takes $O(nTm)$ where $T = O(mn)$. The time complexity is $O(m^2n^2)$.

(b)

Main Idea:

Def. $g(s, i) = \sum_{t=1}^{s} f(t, i)$.

Therefore,

$$\begin{aligned} f(s,i) &= \sum_{j=1}^{m} f(s-j, i-1) \\ &= \sum_{k=s-m}^{s-1} f(k, i-1) \\ &= \sum_{k=1}^{s-1} f(k, i-1) - \sum_{k=1}^{s-m-1} f(k, i-1) \\ &= g(s-1, i-1) - g(s-m-1, i-1) \end{aligned}$$

Goal. $f(T, n)$: the number of length-n valid sequence with sum to T.

Pseudo Code:

Algorithm 6: CountExpressions

input : an integer n, and an integer T
output: the number of the sequence

```
// T is bigger than the possible value.
if T > n * m then
    return 0
end if
Intialize f by an (T + 1) × (n + 1) matrix with all entries equal to 0, index start at 0.
Intialize g by an (T + 1) × (n + 1) matrix with all entries equal to 0, index start at 0.
// Base case
f[0][0] = 1
g[0][0] = 1
for i ← 1 to n do
    for s ← 1 to T do
        if s − m > 0 then
            f[s][i] ← g[s − 1][i − 1] − g[s − m − 1][i − 1]
        else
            f[s][i] ← g[s − 1][i − 1]
        end if
        g[s][i] ← g[s − 1][i] + f[s][i]
    end for
end for
return f[T][n]
```

Proof of Correctness:

We simply make $f(s,i) = g(s-1, i-1) - g(s-m-1, i-1)$. The correctness have been proven in (a) and the def. part in (b).

Time Complexity:

The for-loop take $O(nT)$ time. $T = O(mn)$. Therefore, the time complexity is $O(mn^2)$.

A, B, C, D, E, F, G, H, I, J, K, L, M, N, OPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz, r,

$$1234567890$$

$$\Gamma(s) = \int_0^{+\infty} x^{s-1} e^{-x} dx$$