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)

```
$low \leftarrow 0$
$high \leftarrow n - 1$
while $low < high$ do
  $mid \leftarrow (low + high)/2$
  if $(k == A[mid])$ then
    return mid
  else if $k > A[mid]$ then
    $low \leftarrow mid + 1$
  else
    $high \leftarrow mid - 1$
  end if
end while
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:
Assume S is the input string. Let dp[i][j] denote the length of the longest TENET sequence in the subtring of S, where the substring starts at S[i] and ends at S[j]. then we have the following equation:

$$dp[i][j] = \begin{cases} dp[i+1][j-1]+2 & \text{if } S[i] == S[j] \\ max(dp[i+1][j], dp[i][j-1]) & \text{if } S[i]! = S[j] \end{cases}$$

As for initialization, we have:

$$dp[i][j] = \begin{cases} 1 & \text{if } i == j \\ 0 & \text{if } i > j \end{cases}$$

The final answer is dp[0][n-1].
Proof of Correctness:
The initialization is obviously: if $i == j$ then the substring is a single character which is the TENET sequence with length 1. If $i > j$ then the substring doesn't exist.
As for dp[i][j] $(i < j)$, if the head and tail of the substring are the same, then both the head and tail must be in the TENET sequence, the max length is dp[i+1][j-1] added by 2. If the head and tail are different, then the max length of substring is the larger value between dp[i+1][j] and dp[i][j-1].
Running time analysis:
The running time is $\Theta(n^2)$ since we have two level nested loop to build the dp array

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:

Denote the sum of integers in L as sum, the sum of integers in L that is attached with '-' as neg. Then we have $neg = \frac{sum-w}{2}$.

Define subproblems dp[i][j] as the number of subset which evaluates to j considering the first i elements of L. Then we have bellman equation :

$$dp[i][j] = \begin{cases} dp[i-1][j] & \text{if } L[i] > j \\ dp[i-1][j]+dp[i-1][j-L[i]] & \text{otherwise} \end{cases}$$

The base case:

$$dp[0][j] = \begin{cases} 1, & j = 0 \\ 0, & j \geq 1 \end{cases}$$

The output: dp[n][neg].

Proof of correctness:

We can see that the problem is transformed into "Subset Sum" problem, which is finding the number of subset in L that sums to the target value neg. Obviously if $i == 0, j \geq 1$ then dp[i][j] = 0. If $j == 0$, then dp[i][j] = 1 since we evaluate empty set to 0 and empty set is a subset of any set. If the i-th element is larger than j, then it cannot exist in the final subset, so we have dp[i][j] = dp[i-1][j]. Otherwise, there are two cases, if the final subset doesn't include the i-th element, then dp[i-1][j], if the final subset includes the i-th element, then the subset sum is changed, we have dp[i-1][j-L[i]]. We add these two cases together to get the total number of subsets from L[1...i] that sums to j: dp[i][j] = dp[i-1][j] + dp[i-1][j-L[i]].

Run time analysis:

The running time is $\Theta(n * neg)$ since we have two level nested loop for the length of list and the target value neg.

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 ($n \le 5$) 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) One possible arrangement is [2,2,9,3]. Tom first greedily takes the 3 from the right end, and then Jerry snatches the 9, so Jerry gets 11 and Tom a miserly 5. If Tom had started by craftily taking the 2 from the left end, he'd guarantee that he would get 11 and poor Jerry would be stuck with 5.

(b) **Main Idea:** Let $A[1..n]$ denote the n cards in the line. Jerry defines $v(i,j)$ to be the highest score he can achieve if it's his turn and the line contains cards $A[i..j]$. And Jerry can simplify your expression by expressing $v(i,j)$ as a function of $l(i,j)$ and $r(i,j)$, where $l(i,j)$ is defined as the highest score Jerry can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[i]$; also $r(i,j)$ is defined to be the highest score Jerry can achieve if it's his turn and the line contains cards $A[i..j]$,if he takes $A[j]$. Then the recursive formula should be

$$v(i,j) = \max(l(i,j), r(i,j))$$

where

$$l(i,j) = \begin{cases} A[i] + v(i+1, j-1) & \text{if } A[j] > A[i+1] \\ A[i] + v(i+2, j) & \text{otherwise} \end{cases}$$

$$r(i,j) = \begin{cases} A[j] + v(i+1, j-1) & \text{if } A[i] \ge A[j-1] \\ A[j] + v(i, j-2) & \text{otherwise} \end{cases}$$

Proof of Correctness: Let's consider the condition of $l(i,j)$, if Jerry takes the left one and then Tom will use greedy strategy to consider the cards sequence $A[i+1..j]$. If $A[j] > A[i+1]$, Tom will take $A[j]$ and then Jerry will consider the sequence $A[i+1..j-1]$. If If $A[j] \le A[i+1]$, Tom will take $A[i+1]$ and then Jerry will consider the sequence $A[i+2..j]$. And $r(i,j)$ is the same.
Time complexity: There are $n(n+1)/2$ sub problems and each one can be solved in $\Theta(1)$ time. 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 $g(s, i) := \sum_{t=1}^{s} f(t, i)$.

Solution:

(a) We use $f(s, i)$ to denote the number of sequences of length i with sum s. $f(s, i)$ is 0 when $i > 0$ and $s \leq 0$, and $f(s, 1)$ is 1 if $1 \leq s \leq m$. Otherwise it satisfies the recurrence:

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

There are a total of mn^2 subproblems and it takes $O(m)$ time to compute $f(s, i)$ from its subproblems, which leads to an $O(m^2n^2)$ DP algorithm. Our algorithm outputs $f(T, n)$.

(b) We define $g(s, i)$ as follows:

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

This is equal to

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

Using this recurrence, there are still mn^2 subproblems, but it takes $O(1)$ time to compute $g(s, i)$ from its subproblems, and thus there is a $O(mn^2)$ time DP algorithm. We can then obtain $f(T, n)$ via $g(T, n) - g(T - 1, n)$.