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(·) 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(·) 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 ← 0
high ← n − 1
while low < high do
  mid ← (low + high)/2
  if (k == A[mid]) then
    return mid
  else if k > A[mid] then
    low ← mid + 1
  else
    high ← 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:

Name the original sequence s. Declare a two-dimensional array f for dynamic programming, whose size is $n \times n$. $f[i,j]$, where $0 \leq i,j \leq n$, stands for the length of the longest TENET sequence from position i to j. Initialize $f[i,j]$ to 0 for $i > j$, and $f[i,j]$ to 1 for $i = j$, namely

$$f[i,j] = \begin{cases} 0 & i > j \\ 1 & i = j \end{cases}$$

. Then we can fill in the dp table f with the state transition function, for $i < j$

$$f[i,j] = \begin{cases} f[i+1,j-1]+2 & s[i] = s[j] \\ max(f[i+1,j], f[i,j-1]) & s[i] \neq s[j] \end{cases}$$

, and notice that we have to fill in the dp table from bottom to top, from left to right. The result we want locates at $f[0,n-1]$.

Algorithm 2 Find Longest TENET(s)

```
while i < n do
  while j < n do
    f[i,j] ← 0 if i > j
    f[i,j] ← 1 if i = j
    i ← i + 1
    j ← j + 1
  end while
end while
i ← n − 1
j ← 0
while i ≥ 0 do
  while j < n do
    if i < j then
      if s[i] == s[j] then
        f[i,j] = f[i+1,j−1] + 2
      else
        f[i,j] = max(f[i+1,j], f[i,j−1])
      end if
    end if
    i ← i − 1
    j ← j + 1
  end while
end while
return f[0,n−1]
```

Proof of correctness:

1. The initialization let $f[i,j] = 0$ if $i > j$, and $f[i,j] = 1$ if $i = j$, which obviously makes sense since $f[i,i]$ only correspond to one letter in the original sequence, which is a TENET of length 1, and we define $f[i,j] = 0$ if $i < j$ since we don't consider a sequence from large index to small index as valid.

2. While processing $f[i,j]$, given that data processed before $f[i,j]$ are correct, especially $f[i,j-1], f[i+1,j], and\ f[i+1,j-1]$. If the character s[i] is the same with $s[j]$, then $s[i]$ and $s[j]$ can definitely be added to the TENET from i to j, so we can get the length by adding 2 to the length of TENET from $i+1$ to $j-1$, namely $f[i+1,j-1]+2$. If the character $s[i]$ is different from $s[j]$, we can only add either one of them to the TENET from i+1 to j-1, so we take a max from $f[i+1,j]$ and $f[i,j-1]$.

3. The output is $f[0,n-1]$, which stands for the length of the longest TENET from 0 to n-1, namely in the original sequence.

Running time analysis:
The running time is $O(n^2)$, since there are two doulbe nested loops of n^2 time complexity in our algorithm, each time has $\Theta(1)$ complexity.

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:

Let P denotes the sum of all non-negative integers that is attached with '+', and let N denotes the sum of all non-negative integers that is attached with '-'. Let S denotes the sum of all integers in L, $S = \sum_{i=1}^{n} = P+N$. So, $w = P-N$, then $P = \frac{w+S}{2}$. So, the question is equivalent to **finding the number of combinations in L whose sum equals to P=(w+$\sum_{i=1}^{n} l_i$)/2**. From this statement, we know intuitively that if $w + \sum_{i=1}^{n}$ is odd, then there is no chance to achieve the requirement.
Declare a two-dimensional array f for dynamic programming, whose size is $(n + 1) \times (P + 1)$. $f[i, j]$ stands for the number of combinations in the first i integers that can make a sum to j. Initialize $f[0, 0] = 1$, and $f[0, j] = 0$ for $j \geq 1$.
Then fill in the dp table with the state transition function, for i>0,

$$f[i,j] = \begin{cases} f[i-1,j] & l_i > j \\ f[i-1,j] + f[i-1,j-l_i] & l_i \leq j \end{cases}$$

If $w + \sum_{i=1}^{n} l_i$ is odd, return 0 before do the algorithm above. If it is even, the result we want locates at $f[n, P]$, where $P = \frac{w+\sum_{i=1}^{n} l_i}{2}$.

Proof of correctness:

1. We Initialize $f[0, 0] = 1$ and $f[0, j] = 0$ where j>0, which obviously makes sense since we have nothing to choose from, we can only get a zero.

2. While processing $f[i, j]$, given that data processed before $f[i, j]$ are correct. We can get a sum j in two ways, whether to add j-th integer in the combination or not. If we don't add it into the combination, the number of means should be $f[i - 1, j]$, omitting the i-th integer; otherwise, if we add the i-th integer into the combination, the number of means should be $f[i - 1, j - l_i]$. Therefore, $f[i, j] = f[i - 1, j] + f[i - 1.j - l_i]$. But for $j - l_i < 0$, by considering $f[i - 1, j - l_i] = 0$, so specially $f[i, j] = f[i - 1, j]$.

3. The output is $f[n, P]$, where $P = \frac{w+\sum_{i=1}^{n} l_i}{2}$, meaning the number of combinations in L that add up to p. Its correctness have been proved in the idea part by maths.

Running time analysis:

The running time is $O(nP)$, where $P = \frac{w+\sum_{i=1}^{n} l_i}{2}$. Also $O(n \sum_{i=1}^{n} l_i)$, since $w = O(\sum_{i=1}^{n} l_i)$.
Since the dp table is of size $(n + 1) \times (P + 1)$, the complexity of fill in each grid is $O(1)$, so the total time complexity should be $O(nP)$.

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.

Part a

counterexample: $[3, 9, 2, 1]$.

By greedy, Tom would take 3 first since $3 > 1$, then Jerry would take 9, then Tom 2 Jerry 1, making Tom had 5 in total while Jerry had 10.

There is an optimal approach for Tom. Tom might take 1 first, and Jerry took 3 for optimal, then Tom 9 Jerry 2, makeing Tom had 10 in total while Jerry had 5.

So, greedy doesn't work.

Part b

Main idea:

Denote the original card deck as $a[1 \dots n]$. Declare a teo-dimensional array f for dynamic programming, whose size is $n \times n$. $f[i,j]$, where $0 \le i,j \le n$, stands for the largest sum that Jerry can get from index i to j. Initialize $f[i,j] = 0$ for every $i < j$, and $f[i,j] = a[i]$ for every $i = j$, and $f[i,j] = max(a[i], a[j])$ for every $i + 1 = j$. Namely

$$f[i,j] = \begin{cases} 0 & i < j \\ a[i] & i = j \\ max(a[i], a[j]) & i + 1 = j \end{cases}$$

Jerry have to decide whether to choose from the left end of the deck or the right end. For simplicity, we would declare two auxiliary two-dimensional arrays l and r, both of size $n \times n$. $l[i,j]$, where $0 \le i,j \le n$, stands for the option that Jerry chooses the left end when facing the deck from index i to j. $r[i,j]$ is defined likewise.

So, we get the idea that

$$f[i,j] = max(l[i,j], r[i,j]).$$

and l, r should be respectively:

$$l[i,j] = \begin{cases} a[i] + f[i+2,j] & a[i+1] > a[j] \\ a[i] + f[i+1,j-1] & otherwise \end{cases}$$

$$r[i,j] = \begin{cases} a[j] + f[i+1,j-1] & a[i] > a[j-1] \\ a[j] + f[i,j-2] & otherwise \end{cases}$$

Notice that we should fill in the dp table from bottom to top, from left to right.

If this time Tom goes first, the optimal solution for Jerry should be

$$\begin{cases} f[2,n] & a[1] > a[n] \\ f[1,n-1] & otherwise \end{cases}$$

; otherwise, Jerry goes first, the optimal solution for Jerry should be

$$f[1,n].$$

*Assume Tom alway choose from the right end in this part if two valid options are equal.

Proof of correctness:

1. The initialization let $f[i,j] = 0$ if i>j, which makes sense that this is not a valid range. It let $f[i,j] = a[i]$ if i=j, which makes sense since there is only a[i] to choose. It let $f[i,j] = max(a[i], a[j])$ if $i + 1 = j$, there is only a[i] and a[j], Jerry can just choose the larger one of them.

2. While processing $f[i,j]$, given that data processed before $f[i,j]$ are correct, especially $f[i+2,j], f[i+1,j-1], f[i,j-2]$. Let consider Jerry taking from left end a[i], then Tom would choose the larger one from a[i+1] or a[j], after that, Jerry would have to consider the optimal from i+2 to j or i+1 to j-1 respectively. And Jerry's taking from right end processes likewise.

3. If Jerry goes first, the output is $f[1,n]$, which stands for the optimal solution facing the whole deck. If Tom goes first, the output is $f[2,n]$ if a[1]>a[n], or $f[1,n-1]$ otherwise. That is Tom would choose the larger one of a[1] and a[n], Jerry should choose from the rest.

Running time analysis:
The running time is $O(n^2)$.
The complexity of operation on each grid of the dp table is $\Theta(1)$, and we have n^2 to deal with (including initialization).

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

Part a

Main idea:

Declare a two-dimensional array f for dynamic programming, whose size is $(n + 1) \times (T + 1)$. $f[i, j]$, where $0 \leq i, j \leq n$, stands for the number of means that i numbers sum up to j. Initialize $f[0, 0] = 1$, $f[0, j] = 0$ for j>0, and $f[i, j] = 0$ for i>j. Then fill in the dp table with the state transition function

$$f[i, j] = \sum_{k=max(0,j-m)}^{j-1} f[i - 1, k].$$

The result we want locates at $f[n, T]$.

Proof of correctness:

1. Initializing $f[0, j] = 0$ for j>0 obviously makes sense since there is no number. Then define $f[0, 0] = 1$ specially, and it makes sense because actually we have nothing and can only get a sum of zero. Initializing $f[i, j] = 0$ for i>j makes sense since the sum of i integers must be greater or equal to i, never can we achieve j<i.

2. While processing $f[i, j]$, given that data processed before $f[i, j]$ are correct. There can be m means to get the sum j by adding one integer, the integer can be whatever value in $\{1, 2, \ldots, m\}$. So we can get $f[i, j]$ equals to the sum of number of means that i-1 integers add up to $\{j - m, \ldots, j - 1\}$, namely $f[i, j] = \sum_{k=j-m}^{j-1} f[i - 1, k]$, but if k<0, we simply define $f[i - 1, k] = 0$.

3. The output is $f[n, T]$, which means the number of ways that add n integers in $\{1, \ldots, m\}$ to T.

Running time analysis:

The running time is $O(m^2n^2)$.
The dp table is of size $(n + 1) \times (T + 1)$, the complexity of filling each grid is $O(m)$. And since $T = O(mn)$, so the time complexity of this algorithm is $O(mnT) = O(m^2n^2)$.

Part b

Main idea:

Now think of a new dp table g of the same size as f, being the latter's prefix sum. To be detailed, we regard $g[i,j]$ as the sum of $f[i,0],\ldots,f[i,j]$, namely $g[i,j]=\sum_{k=0}^{j} f[i,k]$, and the actual meaning of it is **the number of ways that make the sum of i integers in $\{1,\ldots,m\}$ be less or equal to j**.

Initialize $g[0,j]=1$ for every valid j, and $g[i,j]=0$ for $i>j$.

Then fill in the rest of g with the state transition function,

$$g[i,j]=\begin{cases} g[i-1,j-1]-g[i-1,j-m-1]+g[i,j-1] & j-m\geq 1 \\ g[i-1,j-1]+g[i,j-1] & j-m<1 \end{cases}$$

The result we want equals to $g[n,T]-g[n,T-1]$.

Proof of correctness:

1. We initialize $g[0,j]=1$, since we can only get zero with nothing, and j is greater than zero, which satisfy the definition of $g[i,j]$. And we also initialize $g[i,j]=0$ for i>j, since the sum of i integers in $\{1,\ldots,m\}$ must be greater or equal to i.

2. While processing $g[i,j]$, given that data processed before $g[i,j]$ are correct. Since $g[i,j]$ stands for the number of ways that make the sum of i integers in $\{1,\ldots,m\}$ be less or equal to j, I'd like to seperate it into two cases:

   (a) The first is the number of ways that make the sum of i integers in $\{1,\ldots,m\}$ be strictly less than j, or saying **less or equal to j-1**, which is $g[i,j-1]$.

   (b) The second is the number of ways that make the sum of i integers in $\{1,\ldots,m\}$ just equals j. According to last part, this should be the sum of $f[i-1,j-m],\ldots,f[i-1,j-1]$. Now we have $g[i-1,j-1]=\sum_{k=0}^{j-1} f[i-1,k]$, and $g[i-1,j-m-1]=\sum_{k=0}^{j-m-1} f[i-1,k]$. So, the number of this case in terms of g should be $g[i-1,j-1]-g[i-1,j-m-1]$. Specially if j-m-1<0, we simply consider $g[i-1,j-m-1]=0$.

3. The output of this algorithm is $g[n,T]-g[n,T-1]$. We can intuitively understand that as $g[n,T]$ is the number that the sum of n integers less or equal to T, similarly that is less or equal T-1 for $g[n,T-1]$. So the difference of them is just the number of ways that n integers in such range add up to T, which is just what we want.

Running time analysis:

The running time is $O(mn^2)$.

The size of the dp table is $(n+1)\times(T+1)$, and the time complexity to fill in each grid is $O(1)$. So the time complexity of the algorithm should be $O(nT)$, and since $T=O(nm)$, so the time complexity should be $O(mn^2)$.