CS101 Algorithms and Data Structures

Fall 2019

Homework 12

Due date: 23:59, December 15th, 2019

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 above may result in zero score.
8. In this homework, all the algorithm design part need the four part proof. The demand is in the next page. If you do not use the four 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 four-part solution, this will help us to score your algorithm. You should include **main idea, pseudocode, proof of correctness and run time analysis.** The detail is as below:

1. The **main idea** of your algorithm. This should be short and concise, at most one paragraph— just a few sentences. It does not need to give all the details of your solution or why it is correct. This is the single most important part of your solution. If you do a good job here, the readers are more likely to be forgiving of small errors elsewhere.

2. The **pseudocode** for your algorithm. 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.

3. 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.

4. 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. Four 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.

Pseudocode:

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') Greedy Cards

TA Wang and Yuan are playing a game, where there are n cards in a line. The cards are all face-up and numbered 2-9. Wang and Yuan take turns. Whoever's turn it is can take one card from either the right end and or the left end of the line. The goal for each player is to maximize the sum of the cards they've collected.

(a) Wang 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 Wang will lose if he plays this greedy strategy, assuming Wang goes first and Yuan plays optimally, but he could have won if he had played optimally.

(b) Yuan decides to use dynamic programming to find an algorithm to find an algorithm to maximize his score, assuming he is playing against Wang and Wang is using the greedy strategy from part (a). Help Yuan develop the dynamic programming solution by providing an algorithm with its runtime and space complexity.

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

(b) Let $A[1..n]$ denote the n cards in the line. Yuan defines $v(i, j)$ to be the highest score he can achieve if it's his turn and the line contains cards $A[i..j]$. Then the recursive formula should be

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

where

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

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

Here we assume that if there is a tie, Yuan takes the card on the left end.

2. (★★★ 10') Three Partition

Given a list of positive numbers, $a_1, \ldots, a_n$, can we partition $\{1, \ldots, n\}$ into 3 disjoint subsets, I, J, K such that:

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{\sum_{i=1}^{n} a_i}{3}$$

Devise and analyse a dynamic programming solution to the above problem that runs in time polynomial in $\sum_{i=1}^{n} a_i$ and n.

Our algorithm will consist of sub-problems indexed by i, j, k, $A(i, j, k)$, which denotes whether it is possible to find two disjoint subsets I_k and J_k of $\{1, \ldots, k\}$ such that $\sum_{m \in I_k} a_m = i$ and $\sum_{l \in J_k} a_l = j$. The solution to each sub-problem is either True or False. Using w to denote $\sum_{i=1}^{n} a_i$, we may assume that i, j are in the range from $0 - w$ and that k ranges from 0 to n. Then the recursion formula should be

$$A(i, j, k) = A(i - a_k, j, k - 1) \vee A(i, j - a_k, k - 1) \vee A(i, j, k - 1)$$

where the base case is $A(i, j, 0) = (i == 0) \wedge (j == 0)$. Our algorithm finally returns $A(w/3, w/3, n)$. The runtime of our algorithm is bounded by $w^2 n$.

3. (★★★ 10')Steel Beams

Given a list of integers $C = (c_1, ..., c_k)$ with $0 < c_1 < c_2 < ... < c_k$ and a target $T > 0$, the algorithm should output nonnegative integers $(a_1, ..., a_k)$ such that $\sum_{i=1}^{k} a_i c_i = T$ where $\sum_{i=1}^{k} a_i$ is as small as possible, or return 'not possible' if no such integers exist.

(a) State your recurrence relation.

(b) Prove correctness of your algorithm by induction.

(c) Find the running time of your algorithm.

We create a dynamic programming algorithm where, for each $n < T$, we will find the minimum integer combination that sum to n. The recurrence is

$$f(n) = \min_{1 \le i \le k} f(n - c_i) + 1$$

where the base case is $f(0) = 0$.

We compute T sub-problems, each one being a minimum of k values, so the running time is $O(Tk)$.

4. (★★★★ 10') Propositional Parentheses

You are given a propositional logic formula using only $\wedge$, $\vee$, T and F that does not have parentheses. You want to find out how many different ways there are to correctly parenthesize the formula so that the resulting formula evaluates to true. For example, the formula $\mathsf{T} \vee \mathsf{F} \vee \mathsf{T} \vee \mathsf{F}$ can be correctly parenthesized in 5 ways:

$$(\mathsf{T} \vee (\mathsf{F} \vee (\mathsf{T} \vee \mathsf{F})))$$
$$(\mathsf{T} \vee ((\mathsf{F} \vee \mathsf{T}) \vee \mathsf{F}))$$
$$((\mathsf{T} \vee \mathsf{F}) \vee (\mathsf{T} \vee \mathsf{F}))$$
$$((\mathsf{T} \vee \mathsf{F}) \vee \mathsf{T}) \vee \mathsf{F})$$
$$((\mathsf{T} \vee (\mathsf{F} \vee \mathsf{T})) \vee \mathsf{F})$$

of which 3 evaluate to true: $((\mathsf{T} \vee \mathsf{F}) \vee (\mathsf{T} \vee \mathsf{F}))$, $(\mathsf{T} \vee ((\mathsf{F} \vee \mathsf{T}) \vee \mathsf{F}))$ and $(\mathsf{T} \vee (\mathsf{F} \vee (\mathsf{T} \vee \mathsf{F})))$.

Give a dynamic programming algorithm to solve this problem. Describe your algorithm, including a clear statement of your recurrence, show that it is correct, and prove its running time.

Algorithm Discription:

Firstly, we use $T(i,j)$ to represent the way of getting true between the ith, jth element. And $F(i,j)$ to represent the way of getting false between the ith, jth element.

And then we can seperate the $T(i,j)$ into two subproblem. i to k and $k+1$ to j. If the operator between k and $k+1$ is $\vee$. Then $T(i,j)+=T(i,k)*T(k+1,j)+F(i,k)*T(K+1,j)+T(i,j)F(k+1,j)$. otherwise the operator is $\wedge$. Then $T(i,j)+=T(i,k)*T(k+1,j)$. And for the $F(i,j)$ is the same.

And for the input the x_i is the T or F, and y_i is the $\vee$ or $\wedge$.

Proof of correctness:

Since we can get the correct value for all the $T(i,i)$ and $F(i,i)$, and then for a certain j assume that we can get all the correct value of $k \leq j$ for the $T(i,i+k)$ and $F(i,i+k)$.

Then we can seperate the $T(i,j)$ into two subproblem. i to k and $k+1$ to j. If the operator between k and $k+1$ is $\vee$, we can get that the final formula get true when one of the formula between i,k or the formula between $k+1,j$ is true then the final formula can be true.

So $T(i,j)+=T(i,k)*T(k+1,j)+F(i,k)*T(K+1,j)+T(i,j)F(k+1,j)$

Otherwise, $T(i,j)+=T(i,k)*T(k+1,j)$

And we calculate the sub problem increasingly, so we can solve every subproblem successfully. So it's correct.

Runtime Analysis:

We do three loop. So the runtime is $O(n^3)$.

Algorithm 2 $ParenthesesTrue(x_1, y_1, x_2, y_2, \cdots, y_{n-1}, x_n)$

for $i = 1$ to n **do**
  if $x_i == T$ **then**
    $T(i,i) \leftarrow 1$, $F(i,i) \leftarrow 0$
  else
    $T(i,i) \leftarrow 0$, $F(i,i) \leftarrow 1$
  end if
end for
for $step = 1$ to n **do**
  for $i = 1$ to $(n - step)$ **do**
    $T(i, i + step) \leftarrow 0$, $F(i, i + step) \leftarrow 0$
    for $j = 0$ to $step$ **do**
      if $y_{i+j} == \vee$ **then**
        $T(i, i + step) = T(i, i + j) * T(i + j + 1, i + step) + F(i, i + j) * T(i + j + 1, i + step) + T(i, i + j)F(i + j + 1, i + step)$
        $F(i, i + step) = F(i, i + j) * F(i + j + 1, i + step)$
      else
        $F(i, i + step) = F(i, i + j) * F(i + j + 1, i + step) + F(i, i + j) * T(i + j + 1, i + step) + T(i, i + j)F(i + j + 1, i + step)$
        $F(i, i + step) = T(i, i + j) * T(i + j + 1, i + step)$
      end if
    end for
  end for
end for
return $T(1, n)$