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 $\mathrm{O}(\cdot)$ 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.

Counterexample: When n is 4, and the sequence of the 4 cards is 3, 6, 8, 4. If Wang plays this greedy strategy, he will take 4 from the right end first for $4 > 3$, and then Yuan takes 8 from the right end. Then Wang takes 6 from the right end for $6 > 3$ and Yuan takes 3. In this case, the sum of Wang's cards is $4+6 = 10$ and the sum of Yuan's cards is $8+3 = 11$. $11 > 10$. Hence, Yuan wins and Wang loses.

However, Wang could have won if he takes 3 from the left end first. Then, if Yuan takes 6 from the left end, Wang can take 8 from the left end and then Yuan takes the last 4. If Yuan takes 4 from the right end, Wang can take 8 from the right end and then Yuan takes the last 6. In both cases, the sum of Wang's cards is $3 + 8 = 11$ and the sum of Yuan's cards is $4 + 6 = 10$. $11 > 10$. Hence, Wang could have won with this strategy.

(b) Yuan decides to use dynamic programming 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.

Main idea:

1. Use $C[1...n]$ to denote the n cards in the line.
2. Use $l(i, j)$ to denote the largest score Yuan can achieve if it is his turn and he takes $C[i]$ from $C[i...j]$(from the left end).
3. Use $r(i, j)$ to denote the largest score Yuan can achieve if it is his turn and he takes $C[j]$ from $C[i...j]$(from the right end).
4. Use $s(i, j)$ to denote the largest score Yuan can achieve if it is his turn and he takes cards from $C[i...j]$.
5. We can get $s(i, j) = max\{l(i, j), r(i, j)\}$.

$$l(i,j) = \begin{cases} C[i] + s(i+1, j-1) & if \quad C[i+1] < C[i] \\ C[i] + s(i+2, j) & if \quad C[i+1] >= C[j] \end{cases}$$

$$r(i,j) = \begin{cases} C[j] + s(i+1, j-1) & if \quad C[i] >= C[j-1] \\ C[j] + s(i, j-2) & if \quad C[i] < C[j-1] \end{cases}$$

Pseudocode:
Initialize i with 0 and j with n. $C[0] = 0$. Then we can get $s(1, n)$. (The pseudocode is on the next page)

Algorithm 2 MaximizeScore($C[i...j]$)

```
if (j − i == 1) then
    s(i, j) ← min{C[i], C[j]}
else if (j == i) then
    s(i, j) ← 0
else
    if (C[j] > C[i + 1]) then
        i ← i + 1
        j ← j − 1
        l(i, j) ← C[i]+ MaximizeScore(C[i...j])
    else
        i ← i + 2
        l(i, j) ← C[i]+ MaximizeScore(C[i...j])
    end if
    if (C[i] >= C[j − 1]) then
        i ← i + 1
        j ← j − 1
        r(i, j) ← C[j]+ MaximizeScore(C[i...j])
    else
        j ← j − 2
        r(i, j) ← C[j]+ MaximizeScore(C[i...j])
    end if
    if (l(i, j) >= r(i, j)) then
        s(i, j) = l(i, j)
    else
        s(i, j) = r(i, j)
    end if
end if
return s(i, j)
```

Proof of Correctness:

1. To initialize, when n is 1, Yuan will take 0 score.
2. When n is 2, Yuan can only take the smaller one because Wang takes the card first.
3. Every turn, Yuan can take from either left or right end, so the optimal solution of each turn is one of them that takes larger score.
4. If it is optimal to take card from the left end, the optimal method must include the left end card(the ith card) and the optimal situation of the remaining cards. In the remaining cards, if the left end card(the $(i+1)$th card) is larger than the right end card(the jth card), Wang will take the left end card in greedy and the optimal situation arises from the $(i+2)$th card to the jth card. Otherwise, the optimal situaton arises from the $(i+1)$th card to $(j-1)$th card.
5. The case that it is optimal to take card from the right end is similar to the previous case.

Running time analysis:

There are $\frac{n(n+1)}{2}$ sub-problems and each costs $\Theta(1)$ time. Hence the time complexity is $O(n^2)$.

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.

Main idea:

1. Break the problem into subproblems $sub(i, j, k)$ and each subproblem is either True or False depending on whether we can find two subsets I_x and J_x such that

$$\sum_{m \in I_x} a_m = i$$

$$\sum_{n \in J_x} a_n = j$$

2. $\sum_{i=1 \to n} a_i = w$, $sub(i, j, k) = sub(i - a_k, j, k - 1) \vee sub(i, j - a_k, k - 1) \vee sub(i, j, k - 1)$. When $i = j = k = 0$, it is True; Otherwise, it is False.

3. Use $S[1...n]$ to denote the set of $\{a_1, ...a_n\}$. Calculate the sum of the set.

4. If the sum is not divisible by 3, it cannot be partitioned into 3 equal disjoint subsets.

5. If the sum is divisible by 3, check whether the partitioned 3 subsets with sub_sum equaling to $\frac{sum}{3}$ exist or not by considering each item in any 3 of the subsets one by one.

   (a) Include the current item in set I and recur for remaining items with remaining sum.

   (b) Include the current item in set J and recur for remaining items with remaining sum.

   (c) Include the current item in set K and recur for remaining items with remaining sum.

6. Save subproblems into the memory to avoid repeatedly computing.

Pseudocode:(It is on the next page)

Proof of Correctness:

It is easier to address 2_partition problem, and we reduce 2_partition problem to 3_partition problem.

Running time analysis:

The time complexity is $O(w^2 n)$, which is polynomial.

Algorithm 3 SubSet(S, n, i, j, k)

```
if (i == 0 and j == 0 and k == 0) then
    return True
end if
if n < 0 then
    return False
end if
I ← False
if (i − S[n] >= 0) then
    I ← SubSet(S, n − 1, i − S[n], j, k)
end if
J ← False
if (A ==True and j − S[n] >= 0) then
    J ← SubSet(S, n − 1, i, j − S[n], k)
end if
K ← False
if (A ==True and B ==True and k − S[n] >= 0) then
    K ← SubSet(S, n − 1, i, j, k − S[n])
end if
return I or J or K
```

Algorithm 4 Partition(S, n)

```
if (n < 3) then
    return False
end if
sum ← the sum of the n items
if (sum/3  !=  0) then
    return False
else
    return SubSet(S, n − 1, sum/3, sum/3, sum/3)
end if
```

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.

1. Use array $C[k]$ to store the integers $c_1, ..., c_k$.
2. Use array $N[k]$ to store the smallest coefficient of c_i.
3. It is similar to coin changing problem. When the change is i yuan, the minimum number of coins required is equal to the ith currency minus all possible occurrences of coins that are less than i yuan plus one.

$$N[i] = min(N[i], N[i - C[j]] + 1), \quad if \quad i > 0$$
$$N[i] = 0, \quad if \quad i = 0$$
$$N[i] = \infty, \quad if \quad i < 0$$

(b) Prove correctness of your algorithm by induction.

1. Suppose $i = 0$, $c_0 = 0$. When $i = 1$, $a_i = \lfloor \frac{T}{a_i} \rfloor$.
2. Suppose when $i = k(k > 0)$, it is correct.

$$N[k] = min(N[k], N[k - C[j]] + 1)$$

3. When $i = k + 1$,

$$N[k + 1] = min(N[k + 1], N[k + 1 - C[j]] + 1)$$

It is correct.

(c) Find the running time of your algorithm.

The time complexity is $O(kT)$.

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:

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

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

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.

Main idea:

1. We know that $\mathsf{T} \vee \mathsf{T} = \mathsf{T}$,$\mathsf{T} \wedge \mathsf{T} = \mathsf{T}$,$\mathsf{F} \vee \mathsf{F} = \mathsf{F}$,$\mathsf{F} \wedge \mathsf{F} = \mathsf{F}$,$\mathsf{F} \vee \mathsf{T} = \mathsf{T}$,$\mathsf{F} \wedge \mathsf{T} = \mathsf{F}$. And if we replace any of the T or F with compound propositions, it is also true.
2. Every $\vee$ or $\wedge$ can separate the proposition into two parts. For the first $\vee$ or $\wedge$, it divides the whole proposition into a T or F and a subproposition.
   (a) If it is $\wedge$ and one part is F, then the proposition must be F.
   (b) If it is $\vee$ and one part is T, then the proposition must be T, the number of the methods to parenthesize the proposition and make it T equals the number of the methods to parenthesize the subproposition.
   (c) Otherwise, separate the subproposition part with the method above recursively and we can get the outcome of the first situation.
3. For the left $\vee$ or $\wedge$, use the same method to decide the number.

Proof of Correctness:

It is based on the properties of propositions.

Running time analysis:

The number of $\vee$ or $\wedge$ is m. The number of T or F is n. The time complexity is $O(mn^2)$.