ShanghaiTech University

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 + 1st 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(·) 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)

```
$low \leftarrow 0$
$high \leftarrow n - 1$
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') 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.

Solution:

(a) Assume the cards are $[2, 3, 9, 4]$.

If Wang decides to use greedy strategy, then Wang goes first and take 4, then Yuan takes 9, Wang takes the 3 and Yuan takes 2. Wang gets 7 in total while Yuan gets 11 in total, so Wang loses.

If Wang plays in optimal, then Wang takes 2 in the first round and then Yuan takes either 3 or 4, which ensures that Wang would take 9 in his next turn. Wang would get 11 in total while Yuan would get 7 in total, so Wang wins in the optimal strategy.

(b) **Main idea:**

Assume the cards are represented as an array A. To maximize score, Yuan should always take the card that is optimal in the remaining cards. In Yuan's turn, he can either take the left card or the right card of the remaining array. Since Wang always takes the larger card, we can figure out the largest sum for Yuan's next turn by comparing the results of choosing the left card and the right card in this turn, and we can use an array to store all larger sums for further usage.

Pseudocode:

Algorithm 2 greedyCards(A,i,j)

```
while $i \le x < j$ and $i \le y < j$ do
  $M[x, y] \leftarrow 0$
end while
while $i \le a < b < j$ do
  if $A[b] \ge A[a+1]$ then
    $L \leftarrow A[a] + M[a+1, b-1]$
  else
    $L \leftarrow A[a] + M[a+2, b]$
  end if
  if $A[a] > A[b-1]$ then
    $R \leftarrow A[b] + M[a+1, b-1]$
  else
    $R \leftarrow A[b] + M[a, b-2]$
  end if
  $M[a, b] \leftarrow max(L, R)$
end while
return $M[i, j]$
```

Proof of correctness:

In Yuan's turn, he has two choices, either take the left card or the right card, and he should always make the choice that will maximum his total score. Suppose that in one turn, the remaining cards Yuan faces are represented as $A[i, j]$. If Yuan choose the left card, then the remaining cards become $A[i+1, j]$ and then is Wang's turn. Since Wang always take the larger card (and suppose he always take the right card if two cards are equal) , there will be two possible situations:

1. If $A[j]$ is larger than or equal to $A[i+1]$, then Wang would take $A[j]$, meaning Yuan can only choose cards from $A[i+1, j-1]$ in his turn and Yuan again finds the optimal in $A[i+1, j-1]$.
2. If $A[j]$ is smaller than $A[i+1]$, then Wang would take $A[i+1]$, meaning Yuan can only choose cards from $A[i+2, j]$ in his turn and Yuan again finds the optimal in $A[i+2, j]$.

This is actually a recursive relation but we can use an array to temporarily store intermediate results.

The relation is similar if Yuan take the right card and Yuan always choose the larger result between choosing left card or choosing right card.

Complexity analysis:

The initialization needs $O(n^2)$ time and there are $O(n^2)$ iterations where each iteration needs $O(1)$, therefore the running time is $O(n^2)$. Since the algorithm needs an $n \times n$ array, the space 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.

Solution:

Main idea:

Since the problem asks if there exists three subsets that have equal sums, we can simplify this problem into two subproblems: does there exist two subsets of $\{a_1, \ldots, a_n\}$ for a given subset of $\{1, \ldots, n\}$ that have sums equal to given values? Define the subproblem as $P(i, j, k)$ where i and j are the given value and k is the largest index of the subset. And we have the following relation:

$$P(i, j, k) = P(i - a_k, j, k - 1) \mid\mid P(i, j - a_k, k - 1) \mid\mid P(i, j, k - 1)$$

The final result can be obtained from $P(\frac{1}{3} \sum_{i=1}^{n} a_i, \frac{1}{3} \sum_{i=1}^{n} a_i, n)$.

Pseudocode:

Suppose all integers are stored in array A and the following algorithm solves the problem be transforming the original problem to $P(\frac{1}{3} \sum_{i=1}^{n} a_i, \frac{1}{3} \sum_{i=1}^{n} a_i, n)$.

Algorithm 3 threePartition(A,k)

```
$total \leftarrow \sum_{i=1}^{k} A[i]$
$i \leftarrow \frac{1}{3}total$, $j \leftarrow \frac{1}{3}total$
while $0 \le a \le i$ do
  while $0 \le b \le j$ do
    if $a == b == 0$ then
      $M[a, b, 0] \leftarrow TRUE$
    else
      $M[a, b, 0] \leftarrow FALSE$
    end if
  end while
end while
while $1 \le c \le k$ do
  while $0 \le a \le i$ do
    while $0 \le b \le j$ do
      $M[a, b, c] \leftarrow (M[a - A[k], b, c - 1) \mid\mid M[a, b - A[k], c - 1] \mid\mid M[a, b, c - 1])$
    end while
  end while
end while
return $M[i, j, k]$
```

Proof of correctness:

For the base case, where $a == b == 0$, the sum of the set is 0 and we can have the correct answer from above algorithm. For non-base cases, suppose we want to find $P(i, j, k)$ and suppose all answers before this subproblem is correctly resolved. Since a_k can only be place into one of the three subsets, then $P(i, j, k)$ is TRUE if at least one of the following conditions is held:

1. $P(i - a_k, j, k - 1)$ is TURE, meaning a_k should be placed into the first subset;
2. $P(i, j - a_k, k - 1)$ is TURE, meaning a_k should be placed into the second subset;
3. $P(i, j, k - 1)$ is TURE, meaning a_k should be placed into the third subset;

If none of the conditions is held, then $P(i, j, k)$ must be FALSE since there is no where no place a_k.

Running time analysis:

The calculation of the sum of $\{1, \ldots, n\}$ can be finished in $O(n)$. Since $a, b \in [0, \frac{1}{3}\sum_{i=1}^{n} a_i]$, $c \in [1, n]$ and each operation in iterations is $O(n)$, therefore the running time is $O((\sum_{i=1}^{n} a_i)^2 \cdot n)$.

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.

Solution:

Main idea:

Define $Opt(n)$ which gives the minimum $\sum_{i=1}^{k} a_i$ such that $\sum_{i=1}^{k} a_i c_i = n$. Then the subproblem is to find the minimum solution in which one integer $c_i \in C$ occurs one less time. The recurrence relation is:

$$Opt(n) = min(Opt(n - c_i)) + 1\, i \in [1, k]$$

Use an array $A[n, k]$ (initialize to zeros) to store the occurrence of $\{c_1, \dots, c_n\}$.

Pseudocode:

```
Algorithm 4 steelBeams(C,n,k)
$M[0] \leftarrow 0$
while $1 \le i \le n$ do
  $M[i] \leftarrow -1$
end while
while $0 \le a \le n$ do
  $last\_add \leftarrow -1$
  while $1 \le b \le k$ do
    if $C[b] \le a$ then
      if $M[a - C[b]] < M[a]$ then
        $M[a] \leftarrow M[a - C[b]]$
        $A[a, b] \mathrel{+}= 1$
        if $last\_add \mathrel{!}= -1$ then
          $A[a, last\_add] \mathrel{-}= 1$
        end if
        $last\_add \leftarrow b$
      end if
    end if
  end while
  $M[a] \mathrel{+}= 1$
end while
return $(M[n] \mathrel{!}= -1)\ ?\ P[n]\ :\ No\_Solution$
```

Proof of correctness:

For the base case, it is clear that no integer should occur, thus the result only consists zeros.

For other cases, suppose $Opt(n-c_i)$ is opotimal and we get $(a_1, \ldots, a_j, \ldots, a_k)$ for $Opt(n-c_i)$, then there exists $(a_1, \ldots, a_j + 1, \ldots, a_k)$ such that $\sum_{i=1}^{k} a_i c_i = n$ while $\sum_{i=1}^{k} a_i$ might not be minimum, so we always find the index j that minimize $\sum_{i=1}^{k} a_i$ for $Opt(n)$ and we are done. If there's no $Opt(n - c_i)$ for all is, then there would be no solution for $Opt(n)$ since we can't find a integer that is suitable.

Running time analysis:

Since for each $a \in [0, T]$ the algorithm needs to go through all $c_i \in C$ and every operation in iteration needs $O(1)$, therefore, the total running time 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 $\mathsf{T} \vee \mathsf{F} \vee \mathsf{T} \wedge \mathsf{F}$ 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: $((\mathsf{T} \vee \mathsf{F}) \vee (\mathsf{T} \wedge \mathsf{F}))$, $(\mathsf{T} \vee ((\mathsf{F} \vee \mathsf{T}) \wedge \mathsf{F}))$ and $(\mathsf{T} \vee (\mathsf{F} \vee (\mathsf{T} \wedge \mathsf{F})))$.

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.

Solution:

Main idea:

Since parenthesizing the formula means changing the priority of calculation, we can bipartite the formula according to priorities around specific origins that are logic operators in the formula. Each time we bipartite the formula and continue bipartite the subformulas until the last subformulas are consisted of only T an F. Then calculate each subformula's value and calculate the binary relation between them according to the particular origin and pass the calculated relation to the upper subformula until the original formula is calculated. Denote the binary values in the formula as $v_1, \ldots, v_n$ and denote the binary operators as $o_1, \ldots, o_{n-1}$, define $T(i,j)$ as the number of ways to parenthesize the subformula $(v_i o_i \ldots o_{j-1} v_j)$ that results in true, define $F(i,j)$ as the number of ways to parenthesize the subformula $(v_i o_i \ldots o_{j-1} v_j)$ that results in false. We can obtain the recurrence relation as:

$$T(i,j) = \begin{cases} 1, & if\ i == j\ and\ v_i == T \\ 0, & if\ i == j\ and\ v_i == F \\ \sum\limits_{i \le m \le j}^{o_m = \vee} (T(i,m) \cdot T(m+1,j) + T(i,m) \cdot F(m+1,j) + F(i,m) \cdot T(m+1,j)) + & \\ \qquad \sum\limits_{i \le m \le j}^{o_m = \wedge} T(i,m) \cdot T(m+1,j), & if\ i < j \end{cases}$$

$$F(i,j) = \begin{cases} 0, & if\ i == j\ and\ v_i == T \\ 1, & if\ i == j\ and\ v_i == F \\ \sum\limits_{i \le m \le j}^{o_m = \wedge} (F(i,m) \cdot F(m+1,j) + T(i,m) \cdot F(m+1,j) + F(i,m) \cdot T(m+1,j)) + & \\ \qquad \sum\limits_{i \le m \le j}^{o_m = \vee} F(i,m) \cdot F(m+1,j), & if\ i < j \end{cases}$$

The final result for this problem can be obtained from $T(1,n)$.

Pseudocode:

Suppose the binary values are stored in array $V[n]$ and the binary operators are stored in array $O[n-1]$. Suppose the indexes of arrays starts from 1.

Algorithm 5 parentheses(V,O,i,j)

```
while $i \le a \le j$ do
  while $i \le b \le j$ do
    if $a == b$ then
      if $V[a] == TURE$ then
        $T[a,b] \leftarrow 1,\ F[a,b] \leftarrow 0$
      else
        $T[a,b] \leftarrow 0,\ F[a,b] \leftarrow 1$
      end if
    else
      $T[a,b] \leftarrow 0,\ F[a,b] \leftarrow 0$
    end if
    $T\_count \leftarrow 0,\ F\_count \leftarrow 0$
    while $i \le m \le j$ do
      if $O[m] == \vee$ then
        $T\_count += (T[a,m] \cdot T[m+1,b] + T[a,m] \cdot F[m+1,b] + F[a,m] \cdot T[m+1,b])$
        $F\_count += F[a,m] \cdot F[m+1,b]$
      else
        $T\_count += T[a,m] \cdot T[m+1,b]$
        $F\_count += (T[a,m] \cdot T[m+1,b] + T[a,m] \cdot F[m+1,b] + F[a,m] \cdot T[m+1,b])$
      end if
    end while
    $T[a,b] \leftarrow T\_count,\ F[a,b] \leftarrow F\_count$
  end while
end while
return $T[1,n]$
```

Proof of correctness:

For the base case, where the formula consists of only one operator and two values, the result is correct. For other cases, suppose the subformulas of one fomula has been correctly calculated, if the operator in this formula is $\vee$, then the formula is TRUE iff at least one of its subformulas is TRUE, the formula is FALSE iff both of its subformulas are FALSE; if the operator in this formula is $\wedge$, then the formula is False iff at least one of its subformulas is FALSE, the formula is TRUE iff both of its subformulas are TRUE.

Running time analysis:

Since $a, b \in [1, n]$ thus there are $O(n^2)$ ways to parentheses the formula for a given origin, and there are n origins can be chosen. Therefore, the total time is $O(n^3)$.