Introduction to Algorithms and Data Structures 1st edition Cengage Solutions Manual

SKU: 9780357673560-SOLUTIONS

Solution and Answer Guide

CENGAGE, INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES, 1e, 2024, 9780357673560; CHAPTER 1: RECURSION

TABLE OF CONTENTS

REVIEW QUESTIONS

1. It is possible to define a recursive function without a base case.

   a. true

   b. false

   Answer: b

   Feedback: A recursive function must finish in a finite time, or number of calls. Thus it must reach a base case where it stops making recursive calls to itself, directly or indirectly.

2. Which of the following are parts of a recursive function definition?

   a. null cases

   b. base cases

   c. automatic calling

   d. composition

   Answer: b

   Feedback: A recursive function must call on itself; that part is called the recursive part. And it also must have a part where it stops calling on itself and gives a direct answer, which are the base cases.

3. When your program has a stack overflow error, it is because the program entered an infinite recursion.

   a. true

   b. false

   Answer: b

Feedback: The stack overflow error occurs when the space of memory called a stack is full. This might happen for reasons other than just infinite recursion. Although, when entering an infinite recursion, this will cause the stack to run out of space.

4. Lucas numbers L_n are defined as follows for a non-negative integer n:

   $L_0 = 2, L_1 = 1.$

   For $n > 1$, $L_n = L_{n-1} + L_{n-2}$

   What are the values of $L_2, L_3,$ and L_4?

   Answer: $L_2 = 3$, $L_3 = 4$, $L_4 = 7$

   Feedback:

   $L_2 = L_1 + L_0 = 1 + 2 = 3.$

   $L_3 = L_2 + L_1 = 3 + 1 = 4.$

   $L_4 = L_3 + L_2 = 4 + 3 = 7.$

5. Which of the following is true about the Lucas numbers definition given in Question 4?

   a. The function enters into an infinite recursion.

   b. It has two base cases.

   c. It has too many base cases.

   d. The recursion is ill defined since it depends on two values of Lucas numbers and not one.

   Answer: b

   Feedback:

   a. Incorrect. The function doesn't enter into an infinite recursion since for every value of *n* it reaches one of the two base cases.

   b. Correct. The Lucas number L_n depends on L_{n-1} and L_{n-2}, which are smaller cases than n by one and two, respectively. This means that at each recursive call the parameter n is reduced. Since n is a non-negative integer, the recursive calls will eventually reach $n = 1$ or $n = 0$. *n = 1* and *n = 0* are the two base cases in the recursive definition.

Commented [LP1]: AU: Include feedback for choices c and d.

6. Programmer A is assigned to write a procedure that displays numbers by subtracting 3 starting from *n* until it reaches a negative number. For $n = 14$, the output should appear as follows:

   $$14\ 11\ 8\ 5\ 2\ -1$$

   The proposed pseudocode solution by programmer A is the following:

```
count(n)
if n < 0
   display n
   finish the program
```

```
else
   display n
   count(n-3)
```

Which is true about the solution of programmer A?

a. The recursion is ill-defined because it doesn't have a base case.

b. It doesn't use recursion.

c. The recursion is ill-defined since it never reaches a base case.

d. It uses tail recursion.

Answer: d

Feedback: Notice that the function has the base case when n is negative. Since when the function calls itself it does it with parameter $n - 3$, you know it will reach a negative number, the base case. Thus, the recursion is well defined. Since the last part of the procedure or the returning value is a call to the same function, this function uses tail recursion.

7. The number of combinations of n elements into k elements, C_k^n, is the number of subsets with k elements that can be formed from a set with n elements, where n and k are non-negative numbers such that $n \geq k$. C_k^n is also known as the number of ways of choosing k elements from n indistinguishable elements. A recursive definition of it can be made as follows:

   (Base case 1) $C_n^n = 1$ for any $n \geq 0$.

   (Base case 2) $C_0^n = 1$ for any $n \geq 0$.

   (Recursion) $C_k^n = C_{k-1}^{n-1} + C_k^{n-1}$ for $0 < k < n$.

   Compute $C_0^0, C_0^1, C_1^1, C_0^2, C_1^2, C_2^2$ using the recursive definition.

   Answer: $C_0^0 = 1, C_0^1 = 1, C_1^1 = 1, C_0^2 = 1, C_1^2 = 2, C_2^2 = 1$

   Feedback: $C_0^0 = 1$ since it is one of the base cases A or B. $C_0^1 = 1$ since it is a base case B. $C_1^1 = 1$ since it is a base case A. $C_0^2 = 1$ since it is a base case B. $C_1^2 = C_{1-1}^{2-1} + C_1^{2-1} = C_0^1 + C_1^1 = 1 + 1 = 2$ by the recursion in C. $C_2^2 = 1$ since it is a base case A.

8. What prevents backtracking from testing all possible cases when building a solution?

   a. bounding condition

   b. infinite recursion

   c. base case

   d. backtracking stop condition

   Answer: a

Feedback:

a. Correct. Backtracking can, if necessary, test all possible solutions, like all the subsets in the subset sum problem. But using the bounding condition can skip several options that are no longer feasible solutions to the problem.

b. This is a state that a recursive function can enter if the base case is not defined correctly.

c. This is where a recursive function ends calling itself recursively.

d. There is no such term when talking of backtracking solutions.

9. When searching for the solution to a problem that might require recursion, what might be a good strategy to start?

Answer: Work out the smallest possible case.

Feedback: Working with small cases will give you insight on how the general solution can be formulated. The smallest possible case can be a part of the base case. This case is usually easy and, in an interview, this can show that you have the right mindset for problem solving.

10. When using backtracking to find a subset with sum 10 from $X = \{2,3,4,5,8\}$, what would be the next step in the algorithm after reaching $S = \{2,3,4,5\}$?

Answer: It stops searching this path since the sum of the elements of S is greater than 10.

Feedback: At each step the algorithm studies two possible options, add a new element to the set or not. The bounding condition is that the sum of the elements so far is not greater than the desired number. In the case of $S = \{2,3,4,5\}$, the sum of the elements is 14, so it is impossible to derive a solution from this set by adding new elements.

PROGRAMMING PROBLEMS

1. Write a program that uses recursion to display all the odd positive integers, in descending order, from *n* to 1. (LO 1.2.)

For $n = 12$, the program must display: 11, 9, 7, 6, 5, 3, 1

For $n = 7$, the program must display: 7, 5, 3, 1

Solution:

```
odds(n)
    if n = 1
        display n
    else
        if n mod 2 = 1
            display(n)
            odds(n-2)
        else
            odds(n-1)
```

2. The Lucas numbers are defined by:

   $L_0 = 2, L_1 = 1.$

   For $n > 1$, $L_n = L_{n-1} + L_{n-2}$

   Write a recursive function that computes the nth Lucas number. (LO 1.2.)

   Solution:

```
lucas(n)
    if n = 0
        return 2
    else-if n = 1
        return 1
    else
        return lucas(n-1) + lucas(n-2)
```

3. The number of combinations of n elements into k elements, C_k^n, is the number of subsets with k elements that can be formed from a set with n elements, where n and k are non-negative numbers such that $n \geq k$. C_k^n is also known as the number of ways of choosing k elements from n indistinguishable elements. A recursive definition of it can be made as follows:

   (Base case 1) $C_n^n = 1$ for any $n \geq 0$.

   (Base case 2) $C_0^n = 1$ for any $n \geq 0$.

   (Recursion) $C_k^n = C_{k-1}^{n-1} + C_k^{n-1}$ for $0 < k < n$.

   Use the previous definition to write a function that recursively computes all C_k^n for any $0 \leq k \leq n$ and n non-negative integers. (LO 1.2)

   Solution:

```
combinations(n, k):
    if k < 0 or n < k  or n < 0
        This is not defined.
    else
        if k = 0 or k = n
            return 1
        else
            return combinations(n-1, k-1) + combinations(n-1, k)
```

4. A palindrome is a string that can be read the same way from left to right or from right to left. Given a nonempty string *w* of size N, check if it is a palindrome or not.

   For example, the string *w* = "hannah" is a palindrome, but *w* = "haha" is not a palindrome. Strings with only one character are considered palindromes, thus *w* = "o" is a palindrome.

   Write a recursive function that checks if a given string is a palindrome. (LO 1.2) Assume that the chars of the string are indexed starting at 0.

Solution:

```
palindrome(w, start, end):
    if start = end
        return True
    else
        if w[start] not equal to  w[end]
            return False
        if end - start > 1
            return palindrome(w, start+1, end-1)
        else
            return True
```

The function is initially called as:

```
palindrome("abc", 0, 2)
```

5. Write a recursive function that receives an array X of size N and displays the elements of the array in reverse order. In this program it is assumed that the array starts at 1. (LO 1.2)

   For example, if $X = [1,3,5,7]$, the output should be "7 5 3 1".

   Assume that the indexing starts at zero.

   Solution:

```
reverse(w, N)
    display w[N-1]
    if N = 1
        return
    else
        reverse(w, N-2)
```

PROJECTS

Project 1-1: The Double Factorial

(LO 1.3)

The double factorial of a non-negative integer n denoted $n!!$ is defined by two cases:

1. If n is even, then $n!!$ is the product of all even numbers from 2 to n. For example, $8!! = 2 \times 4 \times 6 \times 8$. The case $2!! = 2$ since it is the only number from 2 to 2. As usual, $0!! = 1$.
2. If n is odd, then $n!!$ is the product of all odd numbers from 1 to n. For example, $9!! = 9 \times 7 \times 5 \times 3 \times 1$.

Use recursive functions to calculate the double factorial of any non-negative integer n.

Solution:

```
double_factorial(n)
    if n mod 2 == 0
        return even_double_factorial(n)
    else
        return odd_double_factorial(n)

even_double_factorial(n):
    if n == 0
        return 1
    else
        return n*even_double_factorial(n-2)

odd_double_factorial(n)
    if n == 1
        return 1
    else
        return n*odd_double_factorial(n-2)
```

Project 1-2: Binary Search

(LO 1.3)

Suppose you have a sorted array of integers, for example, X = [2, 6, 8, 12, 14, 15]. When asked to find what index corresponds to n = 14, you can implement a binary search method using recursion as follows. Given the array X, an index *start* is the least index allowed for the search, an index *last* is the highest index allowed for the search, and the key value is n:

1. Use the mid element between *start* and *last* as the index i.
2. If $X[i]$ is the desired value, then you have found the index.
3. If $X[i]$ is greater than the desired value (n), then keep searching only on the second half of the array.
4. If $X[i]$ is less than the desired value (n), then keep searching only on the first half of the array.
5. Repeat until *start* and *last* can't be recalculated.

For example, with X = [2, 5, 8, 12, 14, 15] the calls would be:

- n = 14, *start* = 0, *last* = 5. Since $X[2] = 8 < 14$, you search only on the second half of the array.
- n = 14, *start* = 3, *last* = 5. Since $X[4] = 14$, the desired index is 4.

Write a program that realizes a binary search for the key-value n in an array X and returns the index corresponding to the key value. Assume that the key value is present in the array and that the indexes start at 0.

Solution:

```
binary_search(X, n, start, last):
   idx = integer part of (start + last)/2
   if X[idx] = n
      return idx
   else
      if X[idx] < n:
         return binary_search(X, n, idx+1, last)
      else
         return binary_search(X, n, start, idx)
```

Project 1-3: Movements in the Hanoi Tower Solution

(LO 1.4)

You explored one potential solution to the Hanoi Tower problem in the text. But the solution only describes the steps to take in a very abstract way, since it is not so simple to follow all the recursive calls.

For the case with only one disk, the implemented solution requires only one move. For the case with two disks, three movements are required.

Write a program that computes recursively the number of movements necessary to solve the Hanoi Tower problem for *n* disks.

Solution:

```
tower(n, source, target, auxiliary):
    if n == 1
        return 1
    else
        steps = tower(n-1, source, auxiliary, target)
        steps = steps + 1
        steps = steps + tower(n-1, auxiliary, target, source)
        return steps
```

Project 1-4: All Substrings of a String

(LO 1.5)

The substrings of a string are sequences of characters from the original string that preserve the order of appearance. For example, for the string msg = “abcd”, the possible substrings are:

“a”, “b”, “ab”, “ac”, “bc”, “abc”, “ad”, “bd”, “abd”, “acd”, “bcd”, and “abcd”

Notice that not all strings formed with the characters of msg are considered substrings. For example, “bad” is not a substring since “a” is before “b” in msg.

Write a program that lists all possible substrings of a string msg. Assume that all characters in msg are different.

Solution:

```
substrings(txt, S, m)
    if m = 0
        return {txt[0]}
    else
        S1 = substrings(txt, S, m-1)
        S2 = copy of S1
        for sub_str in S1
            S2 = S2 ∪ {sub_str + txt[m]}
        S2 = S2 ∪ {txt[m]}
        return S2
```

Solution and Answer Guide

CENGAGE, INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES, 1e, 2024, 9780357673560; CHAPTER 1: RECURSION

TABLE OF CONTENTS

REVIEW QUESTIONS

1. It is possible to define a recursive function without a base case.

   a. true

   b. false

   Answer: b

   Feedback: A recursive function must finish in a finite time, or number of calls. Thus it must reach a base case where it stops making recursive calls to itself, directly or indirectly.

2. Which of the following are parts of a recursive function definition?

   a. null cases

   b. base cases

   c. automatic calling

   d. composition

   Answer: b

   Feedback: A recursive function must call on itself; that part is called the recursive part. And it also must have a part where it stops calling on itself and gives a direct answer, which are the base cases.

3. When your program has a stack overflow error, it is because the program entered an infinite recursion.

   a. true

   b. false

   Answer: b

Feedback: The stack overflow error occurs when the space of memory called a stack is full. This might happen for reasons other than just infinite recursion. Although, when entering an infinite recursion, this will cause the stack to run out of space.

4. Lucas numbers L_n are defined as follows for a non-negative integer n:

   $L_0 = 2, L_1 = 1.$

   For $n > 1$, $L_n = L_{n-1} + L_{n-2}$

   What are the values of $L_2, L_3,$ and L_4?

   Answer: $L_2 = 3$, $L_3 = 4$, $L_4 = 7$

   Feedback:

   $L_2 = L_1 + L_0 = 1 + 2 = 3.$

   $L_3 = L_2 + L_1 = 3 + 1 = 4.$

   $L_4 = L_3 + L_2 = 4 + 3 = 7.$

5. Which of the following is true about the Lucas numbers definition given in Question 4?

   a. The function enters into an infinite recursion.

   b. It has two base cases.

   c. It has too many base cases.

   d. The recursion is ill defined since it depends on two values of Lucas numbers and not one.

   Answer: b

   Feedback:

   a. Incorrect. The function doesn't enter into an infinite recursion since for every value of *n* it reaches one of the two base cases.

   b. Correct. The Lucas number L_n depends on L_{n-1} and L_{n-2}, which are smaller cases than n by one and two, respectively. This means that at each recursive call the parameter n is reduced. Since n is a non-negative integer, the recursive calls will eventually reach $n = 1$ or $n = 0$. *n = 1* and *n = 0* are the two base cases in the recursive definition.

6. Programmer A is assigned to write a procedure that displays numbers by subtracting 3 starting from *n* until it reaches a negative number. For $n = 14$, the output should appear as follows:

   $$14\ 11\ 8\ 5\ 2\ -1$$

   The proposed pseudocode solution by programmer A is the following:

```
count(n)
if n < 0
   display n
   finish the program
```

```
else
   display n
   count(n-3)
```

Which is true about the solution of programmer A?

a. The recursion is ill-defined because it doesn't have a base case.

b. It doesn't use recursion.

c. The recursion is ill-defined since it never reaches a base case.

d. It uses tail recursion.

Answer: d

Feedback: Notice that the function has the base case when n is negative. Since when the function calls itself it does it with parameter $n - 3$, you know it will reach a negative number, the base case. Thus, the recursion is well defined. Since the last part of the procedure or the returning value is a call to the same function, this function uses tail recursion.

7. The number of combinations of n elements into k elements, C_k^n, is the number of subsets with k elements that can be formed from a set with n elements, where n and k are non-negative numbers such that $n \geq k$. C_k^n is also known as the number of ways of choosing k elements from n indistinguishable elements. A recursive definition of it can be made as follows:

$$\text{(Base case 1) } C_n^n = 1 \text{ for any } n \geq 0.$$

$$\text{(Base case 2) } C_0^n = 1 \text{ for any } n \geq 0.$$

$$\text{(Recursion) } C_k^n = C_{k-1}^{n-1} + C_k^{n-1} \text{ for } 0 < k < n.$$

Compute $C_0^0, C_0^1, C_1^1, C_0^2, C_1^2, C_2^2$ using the recursive definition.

Answer: $C_0^0 = 1, C_0^1 = 1, C_1^1 = 1, C_0^2 = 1, C_1^2 = 2, C_2^2 = 1$

Feedback: $C_0^0 = 1$ since it is one of the base cases A or B. $C_0^1 = 1$ since it is a base case B. $C_1^1 = 1$ since it is a base case A. $C_0^2 = 1$ since it is a base case B. $C_1^2 = C_{1-1}^{2-1} + C_1^{2-1} = C_0^1 + C_1^1 = 1 + 1 = 2$ by the recursion in C. $C_2^2 = 1$ since it is a base case A.

8. What prevents backtracking from testing all possible cases when building a solution?

a. bounding condition

b. infinite recursion

c. base case

d. backtracking stop condition

Answer: a

Feedback:

a. Correct. Backtracking can, if necessary, test all possible solutions, like all the subsets in the subset sum problem. But using the bounding condition can skip several options that are no longer feasible solutions to the problem.

b. This is a state that a recursive function can enter if the base case is not defined correctly.

c. This is where a recursive function ends calling itself recursively.

d. There is no such term when talking of backtracking solutions.

9. When searching for the solution to a problem that might require recursion, what might be a good strategy to start?

Answer: Work out the smallest possible case.

Feedback: Working with small cases will give you insight on how the general solution can be formulated. The smallest possible case can be a part of the base case. This case is usually easy and, in an interview, this can show that you have the right mindset for problem solving.

10. When using backtracking to find a subset with sum 10 from $X = \{2,3,4,5,8\}$, what would be the next step in the algorithm after reaching $S = \{2,3,4,5\}$?

Answer: It stops searching this path since the sum of the elements of S is greater than 10.

Feedback: At each step the algorithm studies two possible options, add a new element to the set or not. The bounding condition is that the sum of the elements so far is not greater than the desired number. In the case of $S = \{2,3,4,5\}$, the sum of the elements is 14, so it is impossible to derive a solution from this set by adding new elements.

PROGRAMMING PROBLEMS

1. Write a program that uses recursion to display all the odd positive integers, in descending order, from *n* to 1. (LO 1.2.)

For $n = 12$, the program must display: 11, 9, 7, 6, 5, 3, 1

For $n = 7$, the program must display: 7, 5, 3, 1

Solution:

```
odds(n)
    if n = 1
        display n
    else
        if n mod 2 = 1
            display(n)
            odds(n-2)
        else
            odds(n-1)
```

2. The Lucas numbers are defined by:

   $L_0 = 2, L_1 = 1.$

   For $n > 1$, $L_n = L_{n-1} + L_{n-2}$

   Write a recursive function that computes the nth Lucas number. (LO 1.2.)

   Solution:

```
lucas(n)
    if n = 0
        return 2
    else-if n = 1
        return 1
    else
        return lucas(n-1) + lucas(n-2)
```

3. The number of combinations of n elements into k elements, C_k^n, is the number of subsets with k elements that can be formed from a set with n elements, where n and k are non-negative numbers such that $n \geq k$. C_k^n is also known as the number of ways of choosing k elements from n indistinguishable elements. A recursive definition of it can be made as follows:

   (Base case 1) $C_n^n = 1$ for any $n \geq 0$.

   (Base case 2) $C_0^n = 1$ for any $n \geq 0$.

   (Recursion) $C_k^n = C_{k-1}^{n-1} + C_k^{n-1}$ for $0 < k < n$.

   Use the previous definition to write a function that recursively computes all C_k^n for any $0 \leq k \leq n$ and n non-negative integers. (LO 1.2)

   Solution:

```
combinations(n, k):
    if k < 0 or n < k  or n < 0
        This is not defined.
    else
        if k = 0 or k = n
            return 1
        else
            return combinations(n-1, k-1) + combinations(n-1, k)
```

4. A palindrome is a string that can be read the same way from left to right or from right to left. Given a nonempty string *w* of size N, check if it is a palindrome or not.

   For example, the string *w* = "hannah" is a palindrome, but *w* = "haha" is not a palindrome. Strings with only one character are considered palindromes, thus *w* = "o" is a palindrome.

   Write a recursive function that checks if a given string is a palindrome. (LO 1.2) Assume that the chars of the string are indexed starting at 0.

Solution:

```
palindrome(w, start, end):
    if start = end
        return True
    else
        if w[start] not equal to  w[end]
            return False
        if end - start > 1
            return palindrome(w, start+1, end-1)
        else
            return True
```

The function is initially called as:

```
palindrome("abc", 0, 2)
```

5. Write a recursive function that receives an array X of size N and displays the elements of the array in reverse order. In this program it is assumed that the array starts at 1. (LO 1.2)

   For example, if $X = [1,3,5,7]$, the output should be "7 5 3 1".

   Assume that the indexing starts at zero.

   Solution:

```
reverse(w, N)
    display w[N-1]
    if N = 1
        return
    else
        reverse(w, N-2)
```

PROJECTS

Project 1-1: The Double Factorial

(LO 1.3)

The double factorial of a non-negative integer n denoted $n!!$ is defined by two cases:

1. If n is even, then $n!!$ is the product of all even numbers from 2 to n. For example, $8!! = 2 \times 4 \times 6 \times 8$. The case $2!! = 2$ since it is the only number from 2 to 2. As usual, $0!! = 1$.
2. If n is odd, then $n!!$ is the product of all odd numbers from 1 to n. For example, $9!! = 9 \times 7 \times 5 \times 3 \times 1$.

Use recursive functions to calculate the double factorial of any non-negative integer n.

Solution:

```
double_factorial(n)
    if n mod 2 == 0
        return even_double_factorial(n)
    else
        return odd_double_factorial(n)

even_double_factorial(n):
    if n == 0
        return 1
    else
        return n*even_double_factorial(n-2)

odd_double_factorial(n)
    if n == 1
        return 1
    else
        return n*odd_double_factorial(n-2)
```

Project 1-2: Binary Search

(LO 1.3)

Suppose you have a sorted array of integers, for example, *X* = [2, 6, 8, 12, 14, 15]. When asked to find what index corresponds to *n* = 14, you can implement a binary search method using recursion as follows. Given the array *X*, an index *start* is the least index allowed for the search, an index *last* is the highest index allowed for the search, and the key value is *n*:

1. Use the mid element between *start* and *last* as the index *i*.
2. If *X*[*i*] is the desired value, then you have found the index.
3. If *X*[*i*] is greater than the desired value (*n*), then keep searching only on the second half of the array.
4. If *X*[*i*] is less than the desired value (*n*), then keep searching only on the first half of the array.
5. Repeat until *start* and *last* can't be recalculated.

For example, with *X* = [2, 5, 8, 12, 14, 15] the calls would be:

- *n* = 14, *start* = 0, *last* = 5. Since *X*[2] = 8 < 14, you search only on the second half of the array.
- *n* = 14, *start* = 3, *last* = 5. Since *X*[4] = 14, the desired index is 4.

Write a program that realizes a binary search for the key-value n in an array X and returns the index corresponding to the key value. Assume that the key value is present in the array and that the indexes start at 0.

Solution:

```
binary_search(X, n, start, last):
   idx = integer part of (start + last)/2
   if X[idx] = n
      return idx
   else
      if X[idx] < n:
         return binary_search(X, n, idx+1, last)
      else
         return binary_search(X, n, start, idx)
```

Project 1-3: Movements in the Hanoi Tower Solution

(LO 1.4)

You explored one potential solution to the Hanoi Tower problem in the text. But the solution only describes the steps to take in a very abstract way, since it is not so simple to follow all the recursive calls.

For the case with only one disk, the implemented solution requires only one move. For the case with two disks, three movements are required.

Write a program that computes recursively the number of movements necessary to solve the Hanoi Tower problem for *n* disks.

Solution:

```
tower(n, source, target, auxiliary):
    if n == 1
        return 1
    else
        steps = tower(n-1, source, auxiliary, target)
        steps = steps + 1
        steps = steps + tower(n-1, auxiliary, target, source)
        return steps
```

Project 1-4: All Substrings of a String

(LO 1.5)

The substrings of a string are sequences of characters from the original string that preserve the order of appearance. For example, for the string msg = “abcd”, the possible substrings are:

“a”, “b”, “ab”, “ac”, “bc”, “abc”, “ad”, “bd”, “abd”, “acd”, “bcd”, and “abcd”

Notice that not all strings formed with the characters of msg are considered substrings. For example, “bad” is not a substring since “a” is before “b” in msg.

Write a program that lists all possible substrings of a string msg. Assume that all characters in msg are different.

Solution:

```
substrings(txt, S, m)
    if m = 0
        return {txt[0]}
    else
        S1 = substrings(txt, S, m-1)
        S2 = copy of S1
        for sub_str in S1
            S2 = S2 U {sub_str + txt[m]}
        S2 = S2 U {txt[m]}
        return S2
```

Instructor Manual

Cengage, Introduction to Algorithms and Data Structures, 1e ©24, 9780357673560;
Chapter 1: Recursion

Table of Contents

Purpose and Perspective of the Chapter

The purpose of this chapter is to introduce recursion, a technique used to solve problems where the case at hand is divided into smaller subtasks similar to the original one. Several basic concepts and methods of recursion are presented and examples are provided. The differences between direct and indirect recursion are then examined, as well as the use of recursion to solve the Tower of Hanoi problem. Finally, the chapter concludes with a section on backtracking and how it can be used to find solutions.

List of Student Downloads

Students should download the following items from the Student Companion Center to complete the activities and assignments related to this chapter:

- There are no student downloads for Chapter 1.

Chapter Objectives

The following objectives are addressed in this chapter:

01.01 Describe the benefits of using recursion.

01.02 Explain how to use recursive methods.

01.03 Explain the difference between direct and indirect recursion.

01.04 Use recursion to solve the Tower of Hanoi problem.

01.05 Use backtracking recursion to enumerate all the subsets of a given set.

Complete List of Chapter Activities and Assessments

The following table organizes activities and assessments by objective, so that you can see how all this content relates to objectives and make decisions about which content you would like to emphasize in your class based on your objectives. For additional guidance, refer to the Teaching Online Guide.

Chapter Objective(s)	Activity/Assessment	Source (i.e., PPT slide, Workbook)	Duration
01.01, 01.02	Activity 1.1: Knowledge Check	PPT Slide 18-19	5-10 minutes
01.03, 01.04, 01.05	Activity 1.2: Knowledge Check	PPT Slides 32-22	5-10 minutes

01.01, 01,02, 01.03, 01.04, 01.05	Activity 1.3: Discussion Questions	PPT Slide 34	25-30 minutes
01.01, 01,02, 01.03, 01.04, 01.05	Self-Assessment	PPT Slide 35	15-20 minutes
1.1, 1.2, 1.5	Review Questions	End of chapter	10-15 minutes
1.1, 1.2, 1.3	Programming Exercises	End of chapter	20-30 minutes
1.2, 1.3, 1.4, 1.5	Projects	End of chapter	30-60 minutes

[return to top]

KEY TERMS

backtracking: A strategy to find solutions to problems that incrementally build solutions. The search for a solution can be recursive or not.

base case: In a recursive function, the case in which the function will no longer call itself (recurse). It is expected that a recursive function reaches a base case.

bounding function or condition: A function or condition that is used during backtracking to determine if the search must continue or not.

direct recursion: When a function calls itself directly.

indirect recursion: When a function calls to itself but in an indirect manner.

recursion: A technique to solve problems where the case at hand is divided into smaller subtasks similar to the original one.

recursive function: A function that calls itself and ultimately reaches a base case.

stack overflow: An error that occurs when the special space of memory in a computer called a stack lacks space to store necessary data.

state of infinite recursion: The state of a program or procedure where it keeps making recursive calls forever.

tail recursion: When a function finishes with the recursive call.

well-defined recursion: A term used to describe a recursive algorithm that reaches an end state in a finite time and that every time it is used returns the same value.

[return to top]

CHAPTER OUTLINE

The following outline organizes activities (including any existing discussion questions in PowerPoints or other supplements) and assessments by chapter (and therefore by topic), so that you can see how all the content relates to the topics covered in the text.

I. **Section 1.1: Introduction to Recursion** (01.01, PPT Slides 4-12)
   a. **Recursion** is a technique to solve problems where the case at hand is divided into smaller subtasks similar to the original one
      I. An important topic to help understand several algorithms, specifically when solving a problem or executing a task that you can split into smaller subtasks
   b. The first structure required in recursion is the **base case**, the case in which the function will no longer call itself (recurse)
      I. The other structure is the recursion, the mechanism or description of how you split the current case into smaller cases
      II. A **recursive function** is a function that calls itself until it reaches a base case
   c. You can use recursion to define a function to compute x^n for any number x and non-negative integer n
   d. **Well-defined recursion** is a recursive algorithm that reaches an end state in a finite time and returns the same value every time it is used
      I. The recursive function calls itself using a smaller case of the initial problem
      II. The point where the function stops calling itself is the base case
   e. A **stack overflow** is an error that occurs when the stack doesn't have space to store necessary data
   f. When a recursive function or a loop doesn't reach a point when it stops, the program has entered a state of **infinite recursion**
   g. The use of recursion is not just a way to simplify algorithms
      I. It also enables you to create solutions using the divide-and-conquer strategy
         (1) Using the divide-and-conquer strategy, you try to divide a complex task into smaller subtasks

(2) The repeated application of this strategy eventually reaches a point where the subtask is so simple that it is possible to solve it directly

h. In tail recursion, the last command in the function is the recursive call
   I. Modern compilers create a more efficient code when it is used
   II. The following code makes use of tail recursion:

```
sum(n, s)
  if n = 1
       return n + s
  else
       return sum(n-1, s + n)
```

II. **Section 1.2: Examples of Recursive Methods** (01.02, PPT Slides 13-19)

a. The factorial function is used in many statistics, combinatorics, and computer science formulas
   I. Represents the number of permutations or different ways of ordering a list of elements
   II. Can be defined through a recursive formula

b. The Fibonacci sequence is defined recursively
   I. Each term in the sequence is obtained from the sum of the two preceding ones
   II. Related to the golden ratio (approximately 1.618)

c. The factorial of a non-negative integer n, usually written as " n!", is the product of all positive integers less than n

d. You can define the factorial of a function as:

```
factorial(n):
 if n = 0
      return 1
else
      return n*factorial(n-1)
```

or:

(Base) 0! = 1
(Recursion) For n>1, (n!) = (n)x(n–1)!

e. The Fibonacci sequence is a set of numbers created in a recursive form
   I. It is common to describe the Fibonacci sequence by F_n

f. **Activity 1.1: Knowledge Check** (01.01, 01.01, PPT Slides 18-19)
   I. The first structure required in recursion is the _____ case, which is the case in which the function will no longer call itself.
   Answer: base
   II. What type of error occurs when the stack doesn't have space to store necessary data?
   Answer: stack overflow

III. In _____ recursion, the last command in the function is the recursive call.
Answer: tail

III. **Direct and Indirect Recursion** (01.03, PPT Slides 20-23)
   a. **Indirect recursion** is when a function calls to itself but in an indirect manner
   b. **Direct recursion** is when the function calls itself directly
   c. You can calculate the square of a positive integer *n* using recursion with the help of the formula:

$$n^2 = n\ (n - 1) + n$$

IV. **The Tower of Hanoi** (01.04, PPT Slides 24-27)
   a. The Tower of Hanoi is a puzzle where you are given a tower formed by some disks and three pegs
      I. When you start, the disks are stacked in increasing order on one peg
      II. The challenge is to move all the disks from peg A to peg C, moving only one disk at a time, and never put a larger disk onto a smaller one

V. **Backtracking: Finding all Subsets** (01.05, PPT Slides 28-36)
   a. **Backtracking** is a technique where solutions are constructed from using smaller solutions
      I. It can be done recursively or not
   b. Backtracking is a strategy to find solutions to problems that incrementally build solutions
   c. The search for a solution can be recursive or not
   d. Backtracking usually provides a more systematic way to enumerate all the cases or potential solutions to a problem, which gives some guarantee that there are no missing ones
   e. In the brute-force approach, you test one by one every subset of *X* until you find one that satisfies the desired condition
   f. Consider the case of finding a subset *S* from $X = \{2,3,4,8\}$ such that the elements of *S* add up to 6
      I. You start with an empty set, that of course is not a solution since the sum of the elements is 0
      II. Now you have two options: include the 2 in the set or not
   g. **Activity 1.2: Knowledge Check** (01.03, 01.04, 01.05, PPT Slides 32-22)
      I. The _____ is a puzzle where you are given a tower formed by some disks and three pegs.
      Answer: Tower of Hanoi

II. Which technique is used when solutions to a problem are constructed by using smaller solutions?
Answer: Backtracking
III. Backtracking can be done recursively or not.
Answer: True

h. **Activity 1.3: Discussion Questions** (01.01, 01.03, 01.04, 01.05, PPT Slide 34)
I. Explain the difference between indirect and direct recursion.
II. Discuss how recursion is used to solve the Tower of Hanoi problem using Figure 1-11.
III. Review the benefits of recursion.
IV. Describe how backtracking might be used to solve problems in everyday life.

i. **Self-Assessment** (01.01-01.05, PPT Slide 35)
I. Describe your experiences developing programs and using programming software. Do you enjoy these kinds of tasks? Do you anticipate using this type of knowledge in your future career?
II. Give an example of how you could use what you learned in this chapter about recursion to improve the quality of data files.

[return to top]

Additional Resources

Internet Resources

- Learn Data Structures and Algorithms | DSA Tutorial: www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/
- Introduction to Recursion – Data Structure and Algorithm Tutorials: https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/
- Data Structure - Recursion Basics: https://www.tutorialspoint.com/data_structures_algorithms/recursion_basics.htm
- Real-world examples of recursion: https://stackoverflow.com/questions/105838/real-world-examples-of-recursion

[return to top]

APPENDIX

GENERIC RUBRICS

Providing students with rubrics helps them understand expectations and components of assignments. Rubrics help students become more aware of their learning process and progress, and they improve students' work through timely and detailed feedback. Customize these rubrics as you wish.

STANDARD DISCUSSION RUBRIC

Criteria	Meets Requirements	Needs Improvement	Incomplete
Participation	Submits or participates in discussion by the posted deadlines. Follows all assignment. instructions for initial post and responses. 5 points	Does not participate or submit discussion by the posted deadlines. Does not follow instructions for initial post and responses. 3 points	Does not participate in discussion. 0 points
Contribution Quality	Comments stay on task. Comments add value to discussion topic. Comments motivate other students to respond. 20 points	Comments may not stay on task. Comments may not add value to discussion topic. Comments may not motivate other students to respond. 10 points	Does not participate in discussion. 0 points
Etiquette	Maintains appropriate language. Offers criticism in a constructive manner. Provides both positive and negative feedback. 5 points	Does not always maintain appropriate language. Offers criticism in an offensive manner. Provides only negative feedback. 3 points	Does not participate in discussion. 0 points

STANDARD COLLABORATION RUBRIC

Criteria	Meets Requirements	Needs Improvement	Incomplete
Share knowledge	Consistently and actively contributes knowledge, opinions, and skills. 5 points	Contributes to the group only when prompted. 3 points	Does not contribute to the group. 0 points
Adjust to unforeseen circumstances	Seeks different solutions, approaches, and strategies in an effective, original, and/or creative way. 5 points	Seeks a single solution, approach, or strategy, but does not pursue better or original alternatives. 3 points	Relies on the first solution generated and does not offer other solutions. 0 points

Make decisions	Helps the group identify necessary changes and encourages group action for change. 10 points	Participates in needed changes only when prompted. 5 points	Does not participate in decision making. 0 points
Build consensus	Values, encourages, and acknowledges the work of other group members. Takes responsibility for the assignment that reflects minority and majority conclusions of the group. 10 points	Listens attentively to members of the group but contributes little to the group and assignment. 5 points	Does not participate in building consensus within the group. 0 points
Complete deliverables (only if applicable)	Completes all assigned components or deliverables with quality, creativity, and accuracy. Turns in all assigned components by agreed upon deadlines without reminders. 10 points	Completes all assigned components with bare minimum of quality and creativity. Requires reminders and follow-up from team members in order to complete deliverables. 5 points	Does not complete assigned components or deliverables or turns them in late. Assigned components are poorly executed, inaccurate, or not functioning. Does not respond to communication from team. 0 points