Unit II Algorithmic Problem Solving

CHAPTER 8

Iteration and recursion

Learning Objectives

After learning the concepts in this chapter, the students will be able

- To know the concepts of variants and invariants used in algorithmic techniques.
- Apply algorithmic techniques in iteration and recursion process.

There are several problems which can be solved by doing the same action repeatedly. Both iteration and recursion are algorithm design techniques to execute the same action repeatedly. What is the use of repeating the same action again and again? Even though the action is the same, the state in which the action is executed is not the same. Each time we execute the action, the state changes. Therefore, the same action is repeatedly executed, but in different states. The state changes in such a way that the process progresses to achieve the desired input-output relation.

Iteration: In iteration, the loop body is repeatedly executed as long as the loop condition is true. Each time the loop body is executed, the variables are updated. However, there is also a property of the variables which remains unchanged by the execution of the loop body. This unchanging property is called the loop invariant. Loop invariant is the key to construct and to reason about iterative algorithms.

Recursion: Recursion is another algorithm design technique, closely related to iteration, but more powerful.Using recursion, we solve a problem with a given input, by solving the same problem with a part of the input, and constructing a solution to the original problem from the solution to the partial input.

DO YOU KNOW?

E W Dijkstra was one of the most influential pioneers of Computing Science. He made fundamental contributions in diverse areas such as programming language design, operating systems, and program design. He coined the phrase "structured programming" which helped lay the foundations for the discipline of software engineering. In 1972, he was awarded ACM Turing Award, considered the highest distinction in computer science. Dijkstra is attributed to have said "Computer science is no more about computers than astronomy is about telescopes."

8.1 Invariants

Example 8.1. Suppose the following assignment is executed with (u, v) = (20,15). We can annotate before and after the assignment.

```
-- before: u, v = 20, 15
u, v :=u+5,v-5
-- after: u, v = 25, 10
```

After assignment (u, v) = (25, 10). But what do you observe about the value of the function u + v?

before: u + v = 20 + 15 = 35

after: u + v = 25 + 10 = 35

The assignment has not changed the value of u + v. We say that u + v is an invariant of the assignment. We can annotate before and after the assignment with the invariant expression.

```
-- before: u + v = 35
u, v : = u + 5, v - 5
-- after : u + v = 35
```

We can say, u + v is an invariant: it is 35 before and after. Or we can say u + v =35 is an invariant: it is true before and after.

Example 8.2. If we execute the following assignment with (p, c = 10, 9), after the assignment, (p, c) = (11, 10).

```
-- before : p, c = 10 , 9
p, c := p + 1, c+1
-- after: p, c = 11 , 10
```

Can you discover an invariant? What is the value of p - c before and after?

before: p — c = 10 — 9 = 1

after: p — c = 11 — 10 = 1

We find that p - c = 1 is an invariant.

In general, if an expression of the variables has the same value before and after an assignment, it is an invariant of the assignment. Let P(u, v) be an expression involving variables u and v. P(u, v)[u, v:= el, e2] is obtained from P(u, v) by replacing u by el and v by e2 simultaneously. P(u, v) is an invariant of assignment u, v := el, e2 if

P(u,v) [u,v := el, e2] = P(u,v)

Example 8.3. Show that p - c is an invariant of the assignment

p, c := p + 1, c + 1

Let P(p, c) = p - c. Then

P (p, c) [p, c := p + 1, c + l]

= p — c [p, c := p + 1, c + l]

= (p + 1) — (c + 1)

= p — c

= P(P , c)

Since (p - c)[p, c := p+l, c+l] = p - c, p - c is an invariant of the assignment p, c := p + 1, c + 1.

Example 8.4. Consider two variables m and n under the assignment

m, n := m + 3, n - 1

Is the expression m + 3n an invariant?

Let P(m, n) = m + 3n. Then

P(m, n) [m, n := m + 3, n — l]

= m + 3n [m, n := m + 3, n — l]

= (m + 3) + 3(n — l)

= m + 3 + 3n — 3

= m + 3n

= P(m, n)

Since (m + 3n) [m, n : = m + 3, n - 1] = m + 3n, m + 3n is an invariant of the assignment m, n := m + 3, n - l.

8.2 Loop invariant

In a loop, if L is an invariant of the loop body B, then L is known as a loop invariant.

```
while C
-- L
B
-- L
```

The loop invariant is true before the loop body and after the loop body, each time. Since L is true at the start of the first iteration, L is true at the start of the loop also (just before the loop). Since L is true at the end of the last iteration, L is true when the loop ends also (just after the loop). Thus, if L is a loop variant, then it is true at four important points in the algorithm, as annotated in the algorithm and shown in Figure 3.1.

1. at the start of the loop (just before the loop)
2. at the start of each iteration (before loop body)
3. at the end of each iteration (after loop body)
4. at the end of the loop (just after the loop)

```
1.  -- L, start of loop
        while
            C
2.      -- L, start of iteration
            B
3.          -- L, end of iteration
4.  -- L, end of loop
```

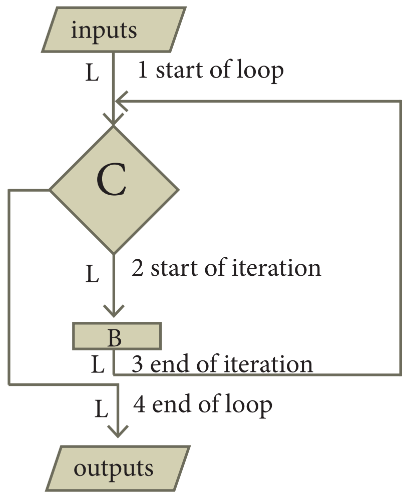


Figure 8.1: The points where the loop invariant is true

To construct a loop,

1. Establish the loop invariant at the start of the loop.
2. The loop body should update the variables, so as to progress toward the end, and maintain the loop invariant, at the same time.
3. When the loop ends, the termination condition and the loop invariant should establish the input-output relation.

8.3 Invariants — Examples

The loop invariant is true in four crucial points in a loop. Using the loop invariant, we can construct the loop and reason about the properties of the variables at these points.

Example 8.5. Design an iterative algorithm to compute a^n. Let us name the algorithm power(a, n). For example,

power(10, 4) = 10000

power (5, 3) = 125

power (2, 5) = 32

Algorithm power(a, n) computes a^n by multiplying a cumulatively n times.

$$a^n = \underbrace{a x a x \dots x a}_{\text{n times}}$$

The specification and the loop invariant are shown as comments.

```
power (a, n)
-- inputs: n is a positive integer
-- outputs: p = a^n
p, i := 1, 0
while i ≠ n
    -- loop invariant: p = a^i
    p, i :=p X a, i+1
```

The step by step execution of power (2, 5) is shown in Table 8.1. Each row shows the values of the two variables p and i at the end of an iteration, and how they are calculated. We see that $p = a^i$ is true at the start of the loop, and remains true in each row. Therefore, it is a loop invariant.

iteration	p	p x a	i	i+1	a^i
0	1		0		2^0
1	2	1x2	1	0 + 1	2^1
2	4	2x2	2	1 + 1	2^2
3	8	4x2	3	2 + 1	2^3
4	16	8x2	4	3 + 1	2^4
5	32	16x2	5	4+1	2^5

Table 8.1: Trace of power (2, 5)

When the loop ends, $p = a^1$ is still true, but i = 5. Therefore, $p = a^5$. In general, when the loop ends, $p = a^n$. Thus, we have verified that power(a, n) satisfies its specification.

Example 8.6. Recall the Chocolate bar problem of Example 6.11. How many cuts are needed to break the bar into its individual squares?

We decided to represent the number of pieces and the number of cuts by variables p and c respectively. Whenever a cut is made, the number of cuts increases by one and the number of pieces also increases by one. We decided to model it by an assignment.

```
p, c := p + 1, c+1
```

The process of cutting the bar can be modeled by a loop. We start with one piece and zero cuts, p = 1 and c = 0. Let n be the number of individual squares. When the number of pieces p equals the number of individual squares n, the process ends.

```
p, c : = 1 , 0
while p ≠ n
   p, c := p + 1, c+1
```

We have observed (in Example 8.2) that p - c is an invariant of the assignment p, c := p + 1, c + 1. Let p - c = k, where k is a constant. The points in the algorithm where p - c = k is true are shown in the algorithm below, and in the flowchart of Figure 8.2.

```
     p, c : = 1 , 0
1.  -- p - c = k
   while p ≠ n
2.     -- p - c = k
       p, c := p+1, c+1
3.     -- p - c = k
4.  --p-c=k,p=n
```

The loop invariant p- c = k is True at the start of the loop (line 1). Moreover, at the start of the loop, p- c = 1. Therefore, k = 1, and the loop invariant is p - c = 1

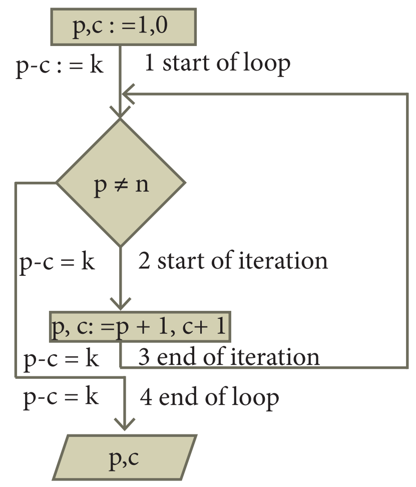


Figure 8.2: The points where the loop invariant is true

When the loop ends (line 4), the loop invariant is still true (p - c = 1). Moreover, the loop condition is false (p = n). From p - c = 1 and p = n,

1. **p — c = 1 loop invariant**
2. **p = n end of the loop**
3. **n — c = 1 from 1, 2**
4. **c = n — 1 from 3**

When the process ends, the number of cuts is one less than the number of squares.

Example 8.7. There are 6 equally spaced trees and 6 sparrows sitting on these trees,one sparrow on each tree. If a sparrow flies from one tree to another, then at the same time, another sparrow flies from its tree to some other tree the same distance away, but in the opposite direction. Is it possible for all the sparrows to gather on one tree?

Let us index the trees from 1 to 6. The index of a sparrow is the index of the tree it is currently sitting on. A pair of sparrows flying can be modeled as an iterative step of a loop. When a sparrow at tree i flies to tree i + d, another sparrow at tree j flies to tree j — d. Thus, after each iterative step, the sum S of the indices of the sparrows remains invariant. Moreover, a loop invariant is true at the start and at the end of the loop.

At the start of the loop, the value of the invariant is

$$S = 1 + 2 + 3 + 4 + 5 + 6 = 21$$

When the loop ends, the loop invariant has the same value. However, when the loop ends, if all the sparrows were on the same tree, say k, then S = 6k.

S = 21,	loop invariant at the start of the loop
S = 6k,	loop invariant at end of the loop
6k= 21,	loop invariant has the same value at the start and the end
21 is a multiple of 6	

It is not possible — 21 is not a multiple of 6. The desired final values of the sparrow indices is not possible with the loop invariant. Therefore, all the sparrows cannot gather on one tree.

Example 8.8. Consider the Chameleons of Chromeland of Example 6.3. There are 13 red, 15 green, and 17 blue chameleons on Chromeland. When two chameleons of different colors meet they both change their color to the third one (for example, if a red and a green meet, both become blue). Is it possible to arrange meetings that result in all chameleons displaying blue color?

Let r, g, and b be the numbers of red, green and blue chameleons. We can model the meetings of two types as an iterative process. A meeting changes (r, g, b) into (r-1, g-1, b+2) or (r-1, g+2, b-1) or (r+2, g-1, b-1). Consider, for example, the meeting of a red and a green chameleon.

r, g, b := r-1, g-1, b+2

The difference in the numbers of any two types either do not change or changes by 3. This is an invariant.

$$r - 1 - (g - 1) = r - g$$
$$r - 1 - (b + 2) = (r - b) - 3$$
$$g - 1 - (b + 2) = (g - b) - 3$$

This is true for all three cases. If any two types differ in number by a multiple of 3 at the start of the iterative process, the difference can be reduced in steps of 3, to 0, when the iterative process ends. However, at the start,

$$r - g \quad = 13 - 15 \quad = -2$$

g - b = 15 - 17 = -2

b - r = 17 - 13 = 4

No two colors differ in number by a multiple of 3. Therefore, all the chameleons cannot be changed to a single color.

Example 8.9. Jar of marbles: You are given a jar full of two kinds of marbles, white and black, and asked to play this game. Randomly select two marbles from the jar. If they are the same color, throw them out, but put another black marble in (you may assume that you have an endless supply of spare marbles). If they are different colors, place the white one back into the jar and throw the black one away. If you knew the original numbers of white and black marbles, what is the color of the last marble in the jar?

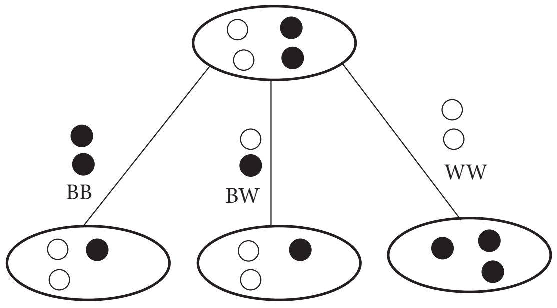


Figure 8.3: State changes in the jar marbles

The number of white and black marbles in the jar can be represented by two variables w and b. In each iterative step, b and w change depending on the colors of the two marbles taken out: Black Black, Black White or White White. It is illustrated in Figure 8.3 and annotated in the algorithm below.

```
1  while at least two marbles in jar
2     -- b , w
3     take out any two marbles
4     case both are black -- BB
5        throw away both the marbles
6        put a black marble back
7        -- b = b '-1, w = w',  b+w = b'+w' -1
8     case both are white  --WW
9        throw away both the marbles
10         put a black marble back
11         --b = b'+1, w = w'-2, --b+w = b'+w'-1
12  else                   --BW
13         throw away the black one
14         put the white one back
15         -- b = b'-1, w = w', b+w = b'+w'-1
```

For each case, how b, w and b+w change is shown in the algorithm, where b' and w' are values of the variables before taking out two marbles. Notice the way w changes. Either it does not change, or decreases by 2. This means that the parity of w, whether it is odd or even, does not change. The parity of w is invariant.

Suppose, at the start of the game, w is even. When the game ends, w is still even. Moreover, only one marble is left, w+b = 1.

1	w + b = 1	end of the loop
2	w = 0orw = 1	from 1
3	w is even	loop invariant
4	w = 0	from 2,3
5	b = 1	from 1,4

Last marble must be black. Similarly, if at the start of the game, there is an odd number of whites, the last marble must be white.

One last question: do we ever reach a state with only one marble? Yes, because the total number of marbles b+w always decreases by one at each step, it will eventually become 1.

8.4 Recursion

Recursion is an algorithm design technique, closely related to induction. It is similar to iteration, but more powerful. Using recursion, we can solve a problem with a given input, by solving the instances of the problem with a part of the input.

Example 8.10. Customers are waiting in a line at a counter. The man at the counter wants to know how many customers are waiting in the line.

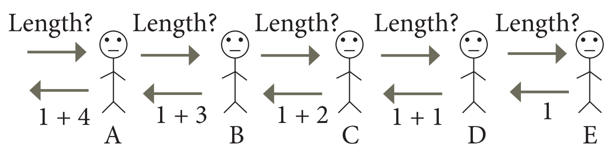


Figure 8.4: Length of a line

Instead of counting the length himself, he asks customer A for the length of the line with him at the head, customer A asks customer B for the length of the line with customer B at the head, and so on. When the query reaches the last customer in the line, E, since there is no one behind him, he replies 1 to D who asked him. D replies 1+1 = 2 to C, C replies 1+2 = 3 to B, B replies 1+3 = 4 to A, and A replies 1+4= 5 to the man in the counter

8.4.1 Recursive process

Example 8.10 illustrates a recursive process. Let us represent the sequence of 5 customers A, B, C, D and E as

[A,B,C,D,E]

The problem is to calculate the length of the sequence [A,B,C,D,E]. Let us name our solver length. If we pass a sequence as input, the solver length should output the length of the sequence.

length [A,B,C,D,E] = 5

Solver length breaks the sequence [A,B,C,D,E] into its first customer and the rest of the sequence.

first [A ,B,C,D,E] = A

rest [A ,B,C,D,E] = [B ,C,D,E]

To solve a problem recursively, solver length passes the reduced sequence [B,C,D,E] as input to a sub-solver, which is another instance of length. The solver assumes that the sub-solver outputs the length of [B,C,D,E], adds 1, and outputs it as the length of [A,B,C,D,E].

length [A,B,C,D,E] = 1 + length [B,C,D,E]

Each solver

1. receives an input,
2. passes an input of reduced size to a sub-solver,
3. receives the solution to the reduced input from the sub-solver, and produces the solution for the given input

as illustrated in Figure 8.5.

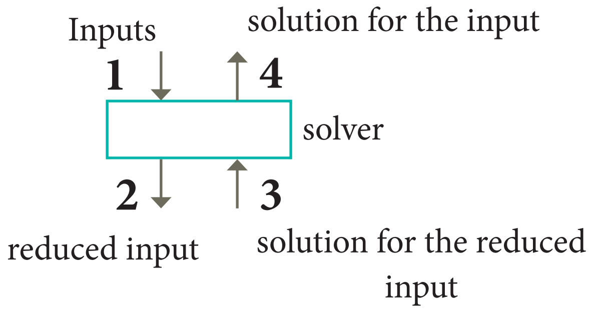


Figure 8.5: One instance of a solver in a recursive process

Figure 8.6 shows the input received and the solution produced by each

solver for Example 8.10. Each solver reduces the size of the input by one and passes it on to a sub-solver, resulting in 5 solvers. This continues until the input received by a solver is small enough to output the solution directly. The last solver received [E] as the input. Since [E] is small enough, the solver outputs the length of [E] as 1 immediately, and the recursion stops.

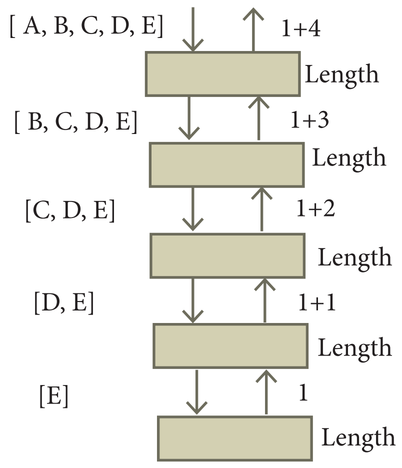


Figure 8.6: Recursive process with solvers and sub-solvers

The recursive process for length [A,B,C,D,E] is shown in Figure 8.7.

1	length [A,B,C,D,E]
2	= 1 + length [B,C,D,E]
3	= 1 +1 + length [C,D,E]
4	= 1 + 1+ 1 + length [D,E]
5	= 1 + 1+ 1 + 1 + length [E]
6	= 1 + 1 + 1 + 1 +1
7	= 1 +1 +1 + 2
8	= 1 + 1 + 3
9	= 1 + 4
10	= 5

Figure 8.7: Recursive process for computing the length of a sequence

8.4.2 Recursive problem solving

Each solver should test the size of the input. If the size is small enough, the solver should output the solution to the problem directly. If the size is not small enough, the solver should reduce the size of the input and call a sub-solver to solve the problem with the reduced input. For Example 8.10, solver's algorithm can be expressed as

$$\text{length of sequence} = \begin{cases} 1 & \text{if sequence has only one customer} \\ 1 + \text{length of tail}, & \text{otherwise} \end{cases}$$

To solve a problem recursively, the solver reduces the problem to sub-problems, and calls another instance of the solver, known as sub-solver, to solve the sub-problem. The input size to a sub-problem is smaller than the input size to the original problem. When the solver calls a sub-solver, it is known as recursive call. The magic of recursion allows the solver to assume that the sub-solver (recursive call) outputs the solution to the sub-problem. Then, from the solution to the sub-problem, the solver constructs the solution to the given problem.

As the sub-solvers go on reducing the problem into sub-problems of smaller sizes, eventually the sub-problem becomes small enough to be solved directly, without recursion. Therefore, a recursive solver has two cases:

1. **Base case:** The problem size is small enough to be solved directly. Output the solution. There must be at least one base case.
2. **Recursion step:** The problem size is not small enough. Deconstruct the problem into a sub-problem, strictly smaller in size than the given problem. Call a sub-solver to solve the sub-problem. Assume that the sub-solver outputs the solution to the sub-

problem. Construct the solution to the given problem.

This outline of recursive problem solving technique is shown below.

```
solver (input)
  if input is small enough
  construct solution
else
  find sub_problems of reduced input
  solutions to sub_problems = solver for each sub_problem
  construct solution to the problem from
solutions to the sub_problems
```

Whenever we solve a problem using recursion, we have to ensure these two cases: In the recursion step, the size of the input to the recursive call is strictly smaller than the size of the given input, and there is at least one base case.

8.4.3 Recursion — Examples

Example 8.11. The recursive algorithm for length of a sequence can be written as

```
length (s)
-- inputs : s
-- outputs : length of s
  if s has one customer -- base case
    1
  else
    1 + length(tail(s)) -- recursion step
```

Example 8.12. Design a recursive algorithm to compute a^n. We constructed an iterative algorithm to compute a^n in Example 8.5. a^n can be defined recursively as

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \times a^{n-1} & \text{otherwise} \end{cases}$$

The recursive definition can be expressed as a recursive solver for computing power(a, n).

```
power (a, n)
-- inputs: n is an integer , n ≥ 0
-- outputs : a^n
if n = 0 -- base case
  1
else    --recursion step
        a × power (a, n-1)
```

The recursive process with solvers for calculating power(2, 5) is shown in Figure 8.8.

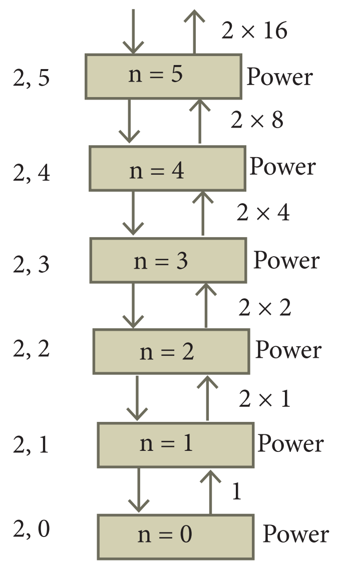


Figure 8.8: Recursive process with solvers for calculating power(2, 5)

The recursive process resulting from power(2, 5) is shown in Figure 8.9.

```
power (2,5)
= 2 × power (2,4)
= 2 × 2 × power(2,3)
= 2 × 2 × 2 × power(2, 2)
= 2 × 2 × 2 × 2 × power (2,1)
= 2 × 2 × 2 × 2 × 2 × power (2,0)
= 2 × 2 × 2 × 2 × 2 × 1
= 2 × 2 × 2 × 2 × 2
= 2 × 2 × 2 × 4
= 2 × 2 × 8
= 2 × 16
= 32
```

Figure 8.9: Recursive process for power(2, 5)

Example 8.13. A corner-covered board is a board of $2^n \times 2^n$ squares in which the square at one corner is covered with a single square tile. A triominoe is a L-shaped tile formed with three adjacent squares (see Figure 8.10). Cover the corner-covered board with the L-shaped triominoes without overlap. Triominoes can be rotated as needed.

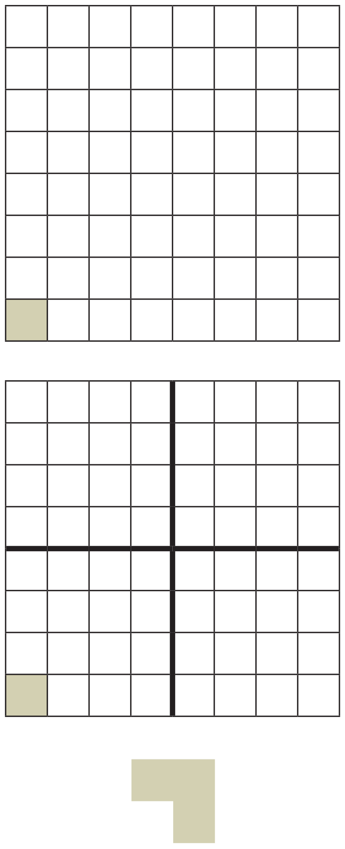


Figure 8.10: Corner-covered board and triominoe

The size of the problem is n (board of size $2^n \times 2^n$). We can solve the problem by recursion. The base case is n = 1. It is a 2×2 corner-covered board. We can cover it with one triominoe and solve the problem. In the recursion step, divide the corner-covered board of size $2^n \times 2^n$ into 4 sub-boards, each of size $2^{n-1} \times 2^{n-1}$, by drawing horizontal and vertical lines through the centre of the board. Place a triominoe at the center of the entire board so as to not cover the corner-covered sub-board, as shown in the left-most board of Figure 8.11. Now, we have four corner-covered boards, each of size $2^{n-1} \times 2^{n-1}$.

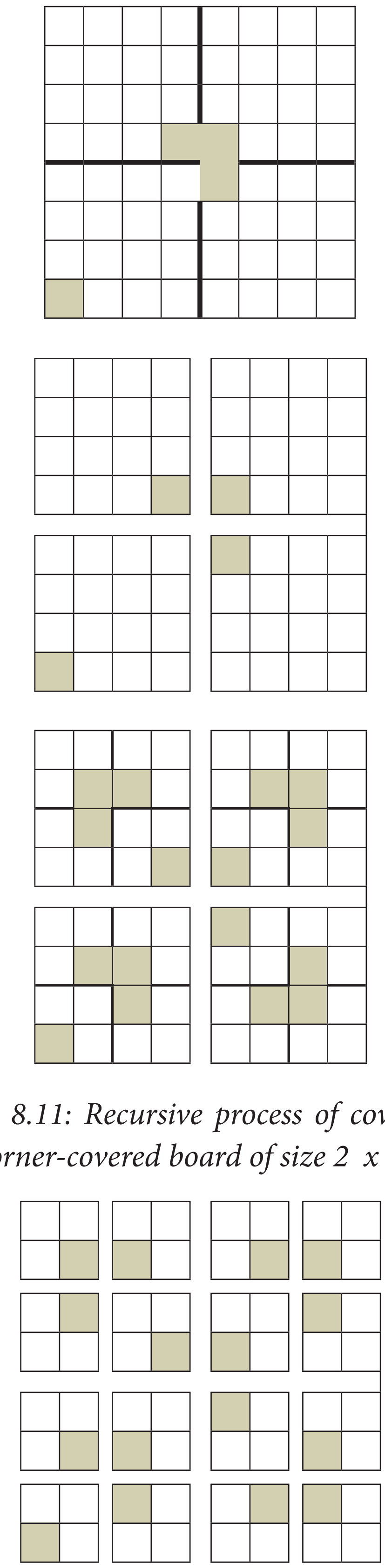
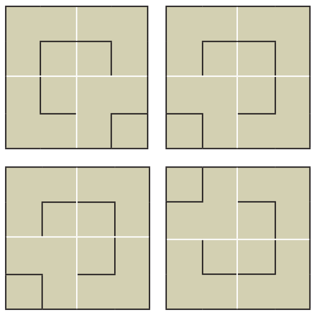


Figure 8.11: Recursive process of covering a corner-covered board of size $2 \ x \ 2^3$



We have 4 sub-problems whose size is strictly smaller than the size of the given problem. We can solve each of the sub-problems recursively.

```
tile corner_covered board of size n
  if n = 1 -- base case
      cover the 3 squares with one triominoe
  else   -- recursion step
      divide board into 4 sub_boards of size n-1
      place a triominoe at centre of board ,
      leaving out the corner_covered sub -board
      tile each sub_board of size n-1
```

The resulting recursive process for covering a 2^3 x 2^3 corner-covered board is illustr*ated in Figure 8.11.*

Points to Remember

- Iteration repeats the two steps of evaluating a condition and executing a statement, as long as the condition is true.
- An expression involving variables, which remains unchanged by an assignment to one of these variables, is called an invariant of the assignment.
- An invariant for the loop body is known as a loop invariant.
- A loop invariant is true.
- (a) at the start of the loop (just before the loop)
- (b) at the start of each iteration (before loop body)
- (c) at the end of each iteration (after loop body)
- (d) at the end of the loop (just after the loop)
- When a loop ends, the loop invariant is true. In addition, the termination condition is also true.
- Recursion must have at least one base case.
- Recursion step breaks the problem into sub-problems of smaller size, assumes solutions for sub-problems are given by recursive calls, and constructs solution to the given problem.
- In recursion, the size of input to a sub-problem must be strictly smaller than the size of the given input.

Evaluation

SECTION – A

Choose the correct answer

1. A loop invariant need not be true

   (a) at the start of the loop. (b) at the start of each iteration
   (c) at the end of each iteration (d) at the start of the algorithm

2. We wish to cover a chessboard with dominoes, $\boxed{}\boxed{}$ the number of black squares and the number of white squares covered by dominoes, respectively, placing a domino can be modeled by

   (a) b := b + 2 (b) w := w + 2 (c) b, w := b+1, w+1 (d) b := w

3. If m x a + n x b is an invariant for the assignment a, b : = a + 8, b + 7, the values of m and n are

   (a) m = 8, n = 7 (b) m = 7, n = -8 (c) m = 7, n = 8 (d) m = 8, n = -7

4. Which of the following is not an invariant of the assignment?

   m, n := m+2, n+3

   (a) m mod 2 (b) n mod 3 (c) 3 X m - 2 X n (d) 2 X m - 3 X n

5. If Fibonacci number is defined recursively as

$$\mathbf{F(n)} = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

   to evaluate F(4), how many times F() is applied?

   (a) 3 (b) 4 (c) 8 (d) 9

6. Using this recursive definition

$$\mathbf{a}^{n} = \begin{cases} 1 & \text{if } n = 0 \\ a \times a^{n-1} & \text{otherwise} \end{cases}$$

   how many multiplications are needed to calculate a^{10}?

   (a) 11 (b) 10 (c) 9 d) 8

SECTION-B

Very Short Answers

1. What is an invariant?
2. Define a loop invariant.
3. Does testing the loop condition affect the loop invariant? Why?
4. What is the relationship between loop invariant, loop condition and the input- output recursively

5. What is recursive problem solving?
6. Define factorial of a natural number recursively.

SECTION-C

Short Answers

1. There are 7 tumblers on a table, all standing upside down. You are allowed to turn any 2 tumblers simultaneously in one move. Is it possible to reach a situation when all the tumblers are right side up? (Hint: The parity of the number of upside down tumblers is invariant.)

2. A knockout tournament is a series of games. Two players compete in each game; the loser is knocked out (i.e. does not play any more), the winner carries on. The winner of the tournament is the player that is left after all other players have been knocked out. Suppose there are 1234 players in a tournament. How many games are played before the tournament winner is decided?

3. King Vikramaditya has two magic swords. With one, he can cut off 19 heads of a dragon, but after that the dragon grows 13 heads. With the other sword, he can cut off 7 heads, but 22 new heads grow. If all heads are cut off, the dragon dies. If the dragon has originally 1000 heads, can it ever die? (Hint:The number of heads mod 3 is invariant.)

SECTION - D

Explain in detail

1. Assume an 8 × 8 chessboard with the usual coloring. "Recoloring" operation changes the color of all squares of a row or a column. You can recolor re-peatedly. The goal is to attain just one black square. Show that you cannot achieve the goal. (Hint: If a row or column has b black squares, it changes by (|8 - b) - b|).

2. Power can also be defined recursively as

$$\mathbf{a}^{\mathbf{n}} = \begin{cases} 1 & \text{if } n = 0 \\ a \times a^{n-1} & \text{if } n \text{ is odd} \\ a^{n/2} \times a^{n/2} & \text{if } n \text{ is even} \end{cases}$$

Construct a recursive algorithm using this definition. How many multiplications are needed to calculate a^{10}?

3. A single-square-covered board is a board of 2^n x 2^n squares in which one square is covered with a single square tile. Show that it is possible to cover the this board with triominoes without overlap.