Design and Analysis of Algorithms

Recursion

B.E. CSE A, Semester 4 (2019-2020)

R. S. Milton

Department of Computer Science and Engineering

SSN College of Engineering

16 December 2019

Contents

1. Recap

1. Specify the problem.
2. Design an algorithm to solve the problem.
3. Construct the algorithm which is correct by construction.
4. Analyse running time of the algorithm.

2. Preview

1. Solve problems using recursion.
2. Construct recursive algorithms.
3. Trace recursive process.
4. Tail recursion and iterative process.
5. Analyse the running time of recursive algorithms.

3. Induction

To prove $P(n)$, a property of a structure of size n

1. Base case: Prove $P(0)$ or $P(1)$.
   The property holds for the structure of size 0 or 1
2. Induction step: Prove $P(n-1) \rightarrow P(n)$.
   - Assume that the property holds for a sub-structure of size $n-1$ (induction hypothesis), and
   - Prove that the property holds fo the structure of size n.

4. Recursive process

- Recursion is an algorithm design technique, closely related to induction.
- Similar to iteration, but more expressive.
- Solve a problem with a given input, by solving instances of the problem with parts of the input.

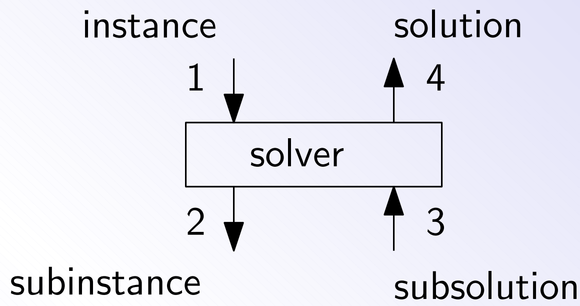


Figure 1: One instance of a solver in a recursive process

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 – the subsolution,
4. constructs the solution for the given input from the subsolutions.

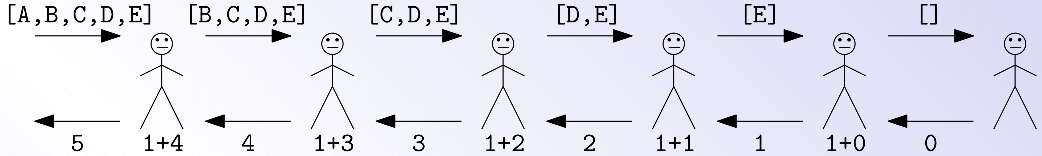


Figure 2: Length of a list

- Each solver reduces the size of the input by one and passes it on to a sub-solver, resulting in 5 solvers.
- Until the input to a solver is small enough to output the solution directly.

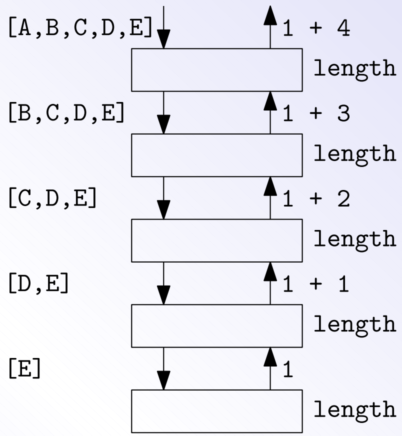


Figure 3: Recursive process with several instances of solver

```
    length [A,B,C,D,E]
  = 1 + length [B,C,D,E]
  = 1 + 1 + length [C,D,E]
  = 1 + 1 + 1 + length [D,E]
  = 1 + 1 + 1 + 1 + length [E]
  = 1 + 1 + 1 + 1 + 1
  = 1 + 1 + 1 + 2
  = 1 + 1 + 3
  = 1 + 4
  = 5
```

Figure 4: Recursive process for computing the length of a sequence

- A problem together with its input is an instance of the problem.
- If we solve the same problem with a different input, we are solving another instance of the problem — the problem is the same, only the input differs.
- The problem with an input which is a part of the given input is called an subinstance of the problem.

5. Recursive problem solving/Recursive algorithm

- 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.
- Construct the solution to the problem from the subsolution.

```
solve (input)
  if input is small enough
     construct solution
  else
     deconstruct input to substructures of smaller size
     subsolutions = for each substructure solve (substructure)
     construct solution from subsolutions
```

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

A solver can assume that sub-solver outputs the solution to the sub-problem, and construct the solution to the given problem.

A recursive algorithm 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 sub-problems, strictly smaller in size than the given problem. Call sub-solvers to solve the sub-problems. Assume that the sub-solver outputs the solutions to the sub-problems. Construct the solution to the given problem.

6. Recursion — Examples

6.1. Length of a list

A recursive algorithm for `length` of a list

```
length(s)
# input:  s
# output: length of s
  if s has one item        # base case
    1
  else
    1 + length(tail(s)) # recursion step
```

6.2. Power of a number

A recurrence relation to compute a^n.

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

The recurrence relation can be expressed as a recursive algorithm for computing `power(a, n)`.

```
power(a, n)
# input:  n is an integer, n ≥ 0
# output: a^n
  if n = 0 # base case
    1
  else       # recursion step
    a × power(a, n−1)
```

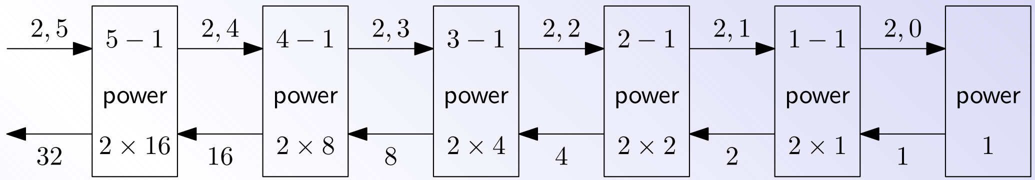


Figure 5: Recursive process for calculating `power(2, 5)`

```
  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 6: Recursive process for `power(2, 5)`

6.3. Corner-covered board

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 7). Cover the corner-covered board with the L-shaped triominoes without overlap. Triominoes can be rotated as needed.

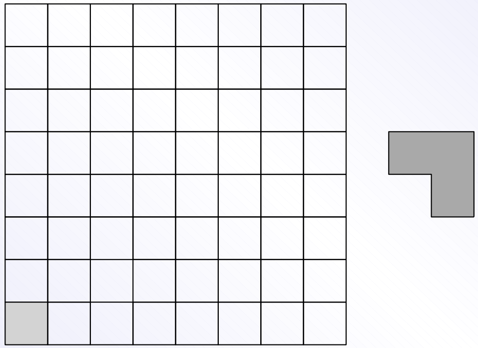


Figure 7: Corner-covered board and triominoe

- The size of the problem is n (board of size $2^n \times 2^n$).
- Base case: $n = 1$. It is a 2×2 corner-covered board. Cover it with one

triominoe and solve the problem.

- 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. Now, we have four corner-covered boards, each of size $2^{n-1} \times 2^{n-1}$.

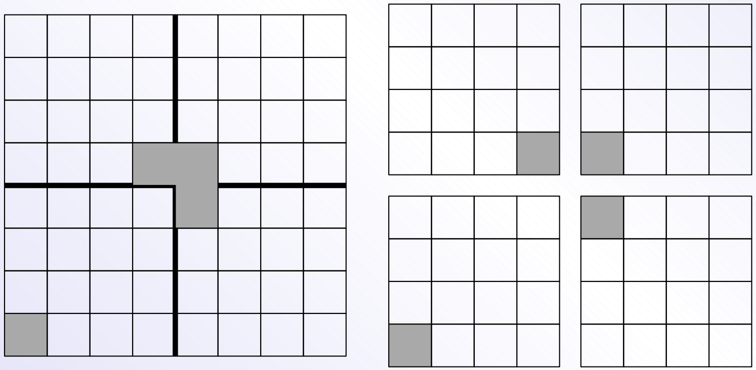


Figure 8: Recursion step for covering a corner-covered board of size $2^3 \times 2^3$

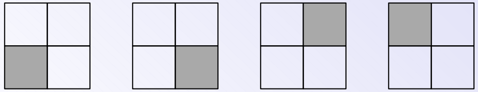


Figure 9: Base case for covering a corner-covered board of size $2^3 \times 2^3$

```
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
```

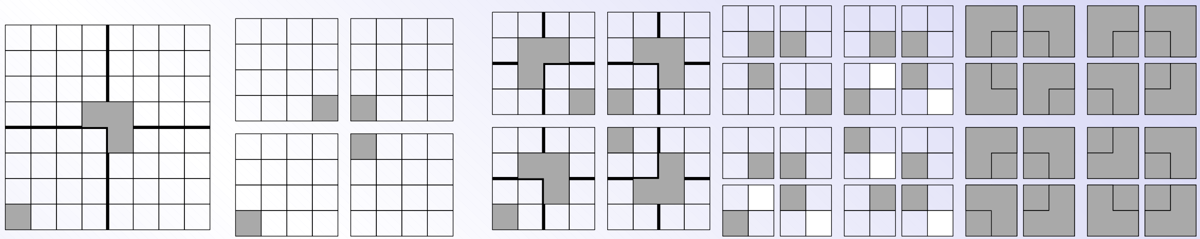


Figure 10: Recursive process of covering a corner-covered board of size $2^3 \times 2^3$

6.4. Tower of Hanoi

There are three poles fixed in the ground. On the first of these poles, 8 discs are placed, each of different size, in decreasing order of size (see Figure 11). How will you move the discs from its pole to the clockwise pole (`cw_pole`) according to the rule that no disc may ever be above a smaller disc.

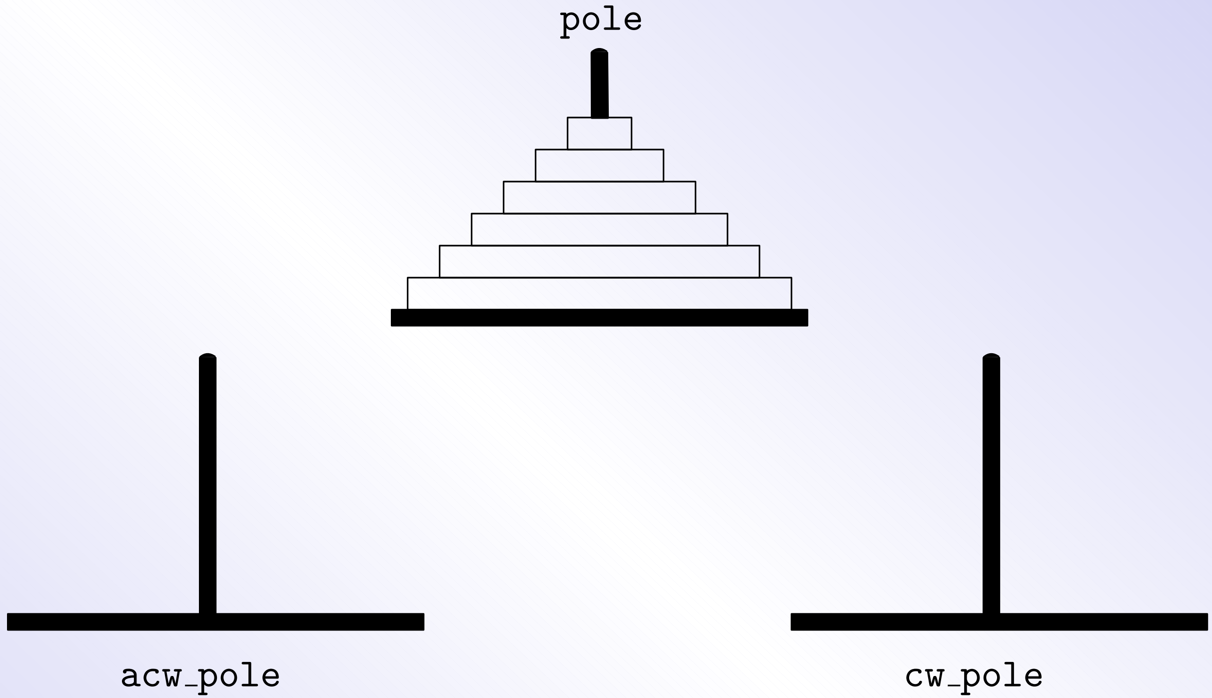


Figure 11: Tower of Hanoi, pole, clockwise pole, anti-clockwise pole

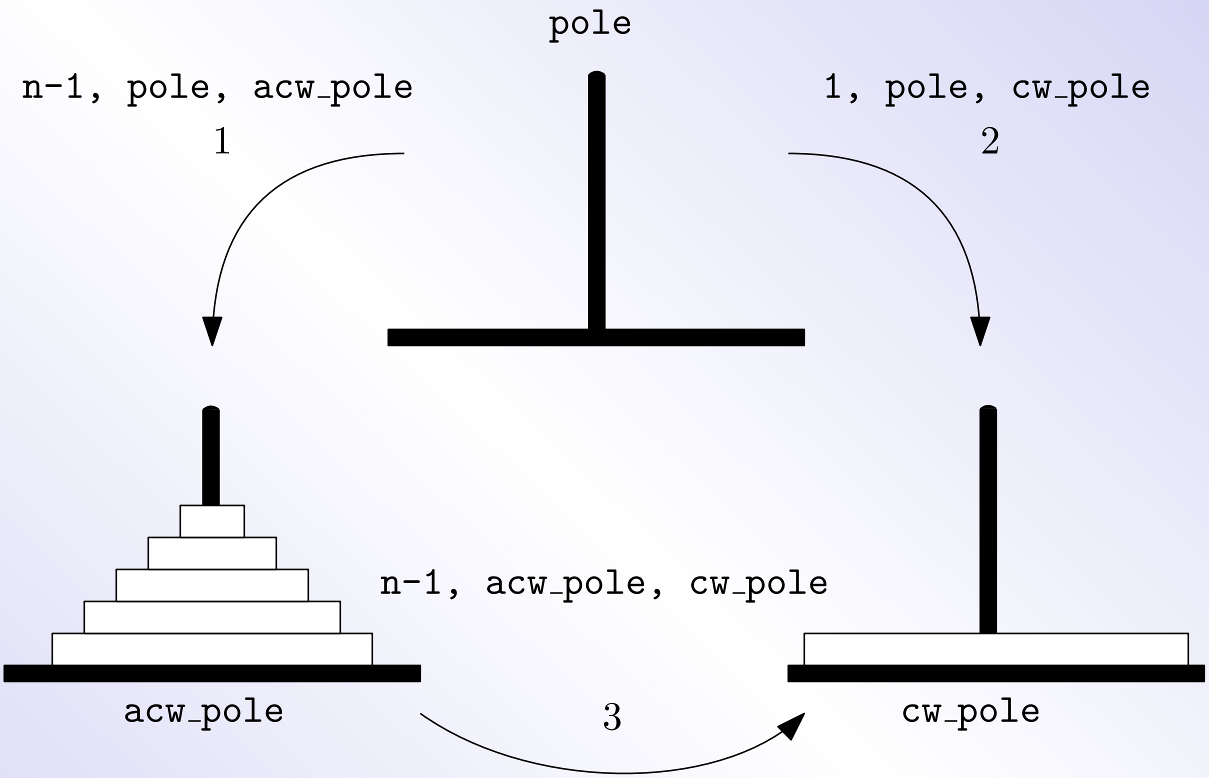


Figure 12: Tower of Hanoi: move tower in two recursive steps

- Base case: There is no disc in the pole.

- Recursion step: Reduce the size of the tower to $n-1$ discs. Move the tower of top $n-1$ discs to the anti-clockwise pole. Move the exposed disc (n) on the pole to the clockwise pole. Then, move the tower of $n-1$ discs from anti-clockwise pole to the clockwise pole.

```
move_tower  (n, pole, cw pole, acw pole)
# input:  tower of size n on pole,
#         towers in cw and acw poles are larger than in pole
# output: tower of size n on cw_pole
  if n > 0
     move_tower (n-1, pole, acw pole, cw pole)
     move_disk (pole, cw pole)
     move_tower (n-1, acw pole, cw pole, pole)
```

6.5. Summation of a list

Input (precondition) is a list `a[0:N]` of `N` addable items. Output (postcondition) is the sum of the items in the array.

$$\text{sum [i:j]} = \begin{cases} 0 & \text{if } i > j \\ \text{[i]} + \text{sum [i+1:j]} & \text{otherwise} \end{cases}$$

or, if a is a list

$$\text{sum a} = \begin{cases} 0 & \text{if } a = [] \\ \text{head a} + \text{sum (tail a)} & \text{otherwise} \end{cases}$$

Algorithm: Sum a[i:j]

Input: An array `[i:j]`

Output: $\sum$ a[i:j]

if `i > j` **then** **return** 0

return a[i] + Sum a[i+1:j]

$\sum$ `[i:j]`

Algorithm: Sum a

Input: A list `a`

Output: $\sum$ a

if `a = []` **then** **return** 0

return a[0] + Sum (a[1:])

$\sum$ `a`

7. Recursive process vs iterative process

7.1. Recursive process

Algorithm: Sum a

Input: List a

Output: $\sum$ a

```
if a = [] then return 0
return a[0] + Sum (a[1:])
# ∑ a
```

sum [2, 9, 1, 6]
2 + sum [9, 1, 6]
    9 + sum [1, 6]
        1 + sum [6]
            6 + sum []
                0
            6 + 0
        1+6
    9 + 7
2 + 16
18

7.2. Iterative process

Algorithm: Sum s, a[i:j]

Input: An array `[i:j]`,

$s = \sum[0:i]$

Output: $\sum$ [0:j]

1 **if** `i > j` **then** **return** s

\# $s = \sum[0:i], [i+1, j]$

2 **return** Sum (s+a[i], [i+1:j])

\# $\sum$ `[0:j]`

0, sum [2, 9, 1, 6]
0 + 2, sum [9, 1, 6]
2 + 9, sum [1, 6]
11 + 1, sum [6]
12 + 6, sum []
18
18
18
18
18

Algorithm: FactRecur n

Input: A nonnegative integer n

Output: $n!$

```
if n = 0 then return 1
return n × FactRecur (n − 1)
```

Algorithm: FactIter f, i, n

Input: A nonnegative integer n,
$f = i!, i \leq n$

Output: $n!$

```
if i = n then return f
return FactIter (f × (i + 1), i + 1, n)
```

8. Tree recursion

8.1. Fibonacci number

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

Algorithm: Fib n

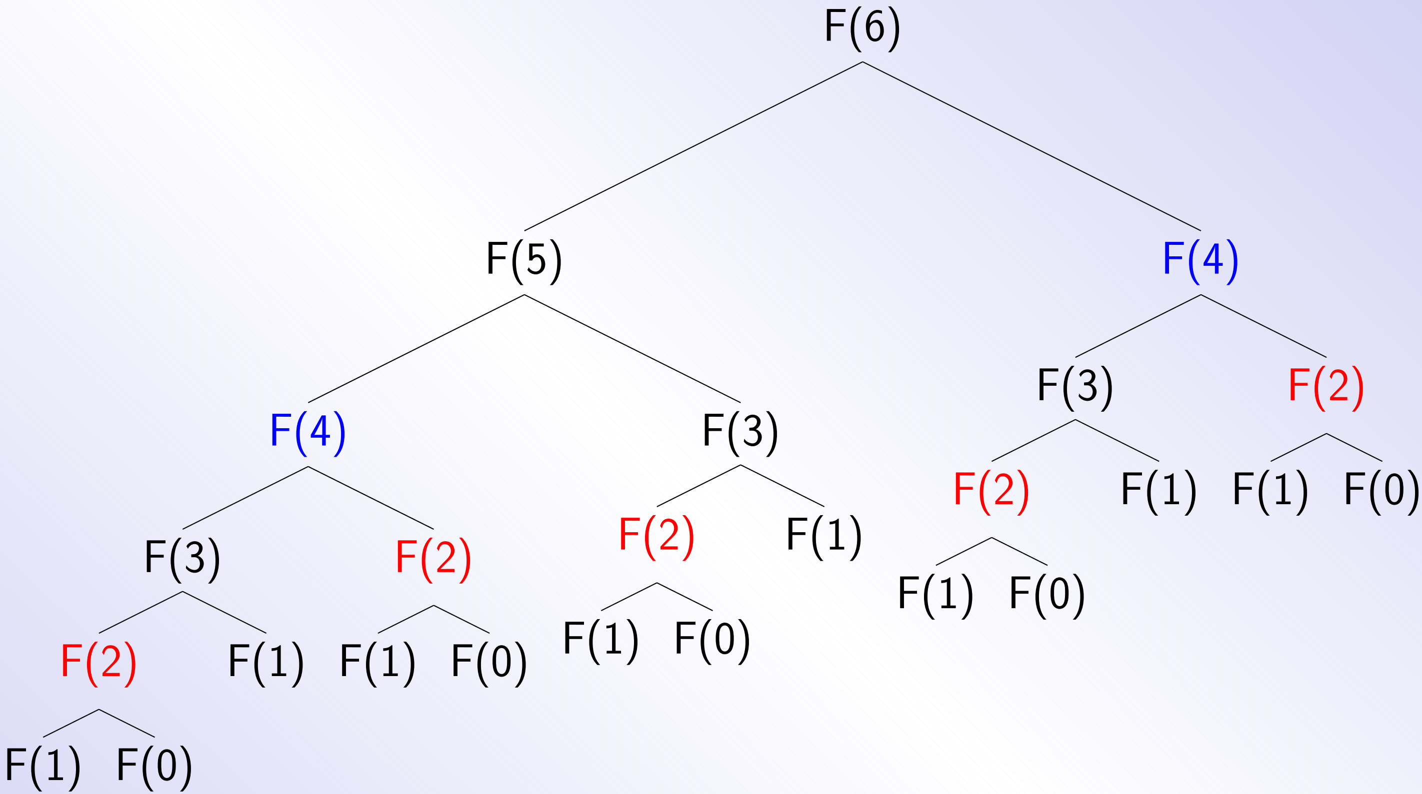
Input: A nonnegative integer n

Output: f_n

if $n = 0$ **then** **return** 0

if $n = 1$ **then** **return** 1

return Fib$(n-1)$ + Fib$(n-2)$


F(6)
F(5)
F(4)
F(4)
F(3)
F(3)
F(2)
F(3)
F(2)
F(2)
F(1)
F(1)
F(0)
F(2)
F(1)
F(1)
F(0)
F(1)
F(0)
F(1)
F(0)
F(1)
F(0)

]

8.2. Binary trees

8.2.1. Traverse

Pre-order: Visit the root before; then traverse the left subtree and right subtree.
Root–Left–Right

Algorithm: PreOrder (r)

Input: BST r

Output: Items of r in pre-order

```
if r ≠ ∅ then
    print (r.key)
    PreOrder (r.left)
    PreOrder (r.right)
end
```

8.2.2. Count

Algorithm: Count (r)

Input: Binary tree r

Output: Number of items in r

```
if r = ∅ then
    return 0
return Count (r.left) + Count (r.right)
```

8.2.3. Search

Search a BST `r` for a target `t`: `Search (r, t)`.

$$\text{Search (r, t)} = \begin{cases} r & \text{if r} = \text{null} \\ r & \text{if t} = \text{r.key} \\ \text{Search (r.left, t)} & \text{if t} < \text{r.key} \\ \text{Search (r.right, x)} & \text{if t} > \text{r.key} \end{cases}$$

Algorithm: Search(r, t)

Input: BST r, search key t

Output: Node with the key t or null

```
if r = null or r.key = t then
    return r
if t < r.key then
    return Search(r.left, t)
else
    return Search(r.right, t)
end
```

8.3. Graphs

Algorithm: DepthFirstSearch (v)

Input: v is a vertex.

Output: v is explored $=$ v and all its neighbors w are discovered.

```
if not discovered (v) then
    discovered (v) ← true
    # discovered (v)
    foreach edge (v,w) do
        DepthFirstSearch (w)
        # discovered (w)
    end
end
# explored (v) = for all w, discovered (w)
```

9. Summary

- Recursion is more expressive than iteration.
- Recursive problem solving
  1. Deconstruct the input structure to smaller substructure
  2. Solve the problem for the substructures
  3. Construct the solution from the subsolutions
- Recursive algorithm
  1. Base case(s)
  2. Recursion step: input size strictly smaller
- Tail recursion is iteration.
- Input structures: numbers, lists, trees, graphs, sets