Algorithmic Strategies 2025/26

Week 2 – Recursion

· U C ·

Universidade de Coimbra

Outline

1. Introduction
2. Examples

Reading about problem solving with recursion

- J. Erickson, Algorithms, Chapter 1
- J. Edmonds, How to think about algorithms, Chapter 8 (or Part II - recursion)
- S.S. Skiena, M.G. Revilla, Programming Challenges, Chapter 6

Problem solving

- In EA, you can solve most of the problems by using reduction techniques. You need to recognize the underlying problem.
- Or use a general strategy: Break the problem down into smaller problems which you can solve, and devise how to recover the solution from the partial solutions found
- This is the main strategy of backtracking, dynamic programming, greedy algorithms and branch-&-bound
- To know how to break the problem in the most effective manner requires a lot of training

Introduction

Recursive program: A program that calls itself.

Main idea: We solve the problem by solving smaller sub-problems.

1. A base case (simple problem, not solved by recursion)
2. A recursive step (uses solutions of sub-problems)

Introduction

Recursive program: A program that calls itself.

Main idea: We solve the problem by solving smaller sub-problems.

1. A base case (simple problem, not solved by recursion)
2. A recursive step (uses solutions of sub-problems)

Proof by mathematical induction:

1. (Base case) True for the base case
2. (Inductive hypothesis) Assume that is true for k
3. (Inductive step) True for $k \implies$ true for $k+1$.

Induction:

Show that $0 + 1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

1. Base case: True for $n = 0$: $0 = \dfrac{0 \cdot (0+1)}{2}$
2. If it holds for k, then it also holds for $k+1$:

$$(0 + 1 + 2 + \cdots + k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Under the induction hypothesis that is true for k:

$$\frac{k(k+1)}{2} + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Induction:

Show that $0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

1. Base case: True for $n = 0$: $0 = \frac{0 \cdot (0+1)}{2}$
2. If it holds for k, then it also holds for $k + 1$:

$$(0 + 1 + 2 + \cdots + k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Under the induction hypothesis that is true for k:

$$\frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

Induction:

Show that $0 + 1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

1. Base case: True for $n = 0$: $0 = \dfrac{0 \cdot (0+1)}{2}$
2. If it holds for k, then it also holds for $k + 1$:

$$(0 + 1 + 2 + \cdots + k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Under the induction hypothesis that is true for k:

$$\frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

Induction:

Show that $0 + 1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

1. Base case: True for $n = 0$: $0 = \dfrac{0 \cdot (0+1)}{2}$
2. If it holds for k, then it also holds for $k+1$:

$$(0 + 1 + 2 + \cdots + k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Under the induction hypothesis that is true for k:

$$\frac{(k+1)((k+1)+1)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

A recursive algorithm to compute the square of a number n

Function $SQ(n)$
  if $n = 0$ **then** {base case}
    $s = 0$
  else
    $s = SQ(n-1) + 2(n-1) + 1$ {recursive step}
  return s

Note that $n^2 = (n-1)^2 + 2(n-1) + 1$.

Correctness proof by induction

- The recursion terminates when $n = 0$
- Base case: After the last recursion, $s = 0$
- Inductive hypothesis: Assume that after returning from $k - 1$ recursions, $s = (k - 1)^2$
- Inductive step: After returning from k recursions, $s = (k - 1)^2 + 2(k - 1) + 1 = k^2$
- Then, after returning from n recursions, $s = (n - 1)^2 + 2(n - 1) + 1 = n^2$

Patterns:

- Handle first or last and recur on remaining
- Divide in half, recur on one/both halves (D&C)

Pros: Smaller code, few or no local variables.

Cons: Less eficient than iterative because of the push and pop operations in the run-time stack. Can have problems of stack overflow.

Problem: Draw a Sierpiński triangle

Examples

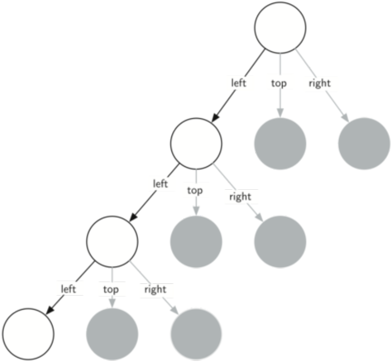
Problem: Draw a Sierpiński triangle

Recursion: Draw smaller triangles at the left, top and right of the large triangle

Base case: The triangle is small enough

Examples

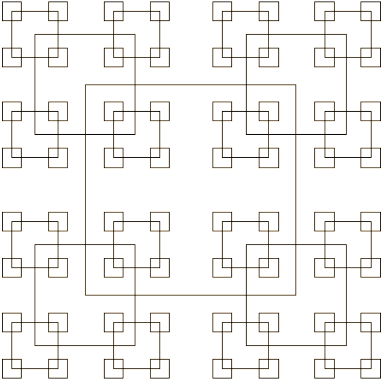
Recursive call tree

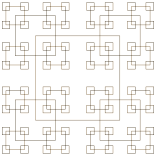


base case →

Examples

Problem: All Squares (modified UVa 155)





Function *Square*(x, y, s)

- *drawSquare*(x, y, s) {(x, y) is the centroid of the square}
- **if** $s/2 \leq 1$ **then** {base case}
  - **return**
- **else** {recursive step}
  - *Square*$(x + s/2, y + s/2, s/2)$ {top-right}
  - *Square*$(x - s/2, y + s/2, s/2)$ {top-left}
  - *Square*$(x + s/2, y - s/2, s/2)$ {bottom-right}
  - *Square*$(x - s/2, y - s/2, s/2)$ {bottom-left}

Examples

Problem: How many squares?

Function *Square*(x, y, s)

  if $s/2 \leq 1$ **then** {base case}

    return 1

  else {recursive step}

    return $1 + Square(x + s/2, y + s/2, s/2) +$ {top-right}

      $Square(x - s/2, y + s/2, s/2) +$ {top-left}

      $Square(x + s/2, y - s/2, s/2) +$ {bottom-right}

      $Square(x - s/2, y - s/2, s/2)$ {bottom-left}

Examples

Problem: How many squares contain a given point (p_x, p_y)?

Function *Square*(x, y, s)

$k = 0$

if $p_x \in [x - s/2, x + s/2]$ **and** $p_y \in [y - s/2, y + s/2]$ **then** $\{$in$\}$

$k = 1$

if $s/2 \leq 1$ **then** $\{$base case$\}$

return k

else $\{$recursive step$\}$

return $k + Square(x + s/2, y + s/2, s/2) +$ $\{$top-right$\}$

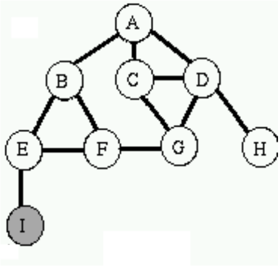
$Square(x - s/2, y + s/2, s/2) +$ $\{$top-left$\}$

$Square(x + s/2, y - s/2, s/2) +$ $\{$bottom-right$\}$

$Square(x - s/2, y - s/2, s/2)$ $\{$bottom-left$\}$

Examples

Problem: Depth First Search (DFS) in a graph $G = (V, E)$



Recursion: Visit neighbors of a node in G that were not yet visited

Base case: All neighbors were already visited

Function $\mathit{dfs}(G, u)$

  $\mathit{color}(u) = \mathit{gray}$ {node u is in progress}

  for each $\{u, v\} \in E$ **and** $\mathit{color}(v) = \mathit{white}$ **do**

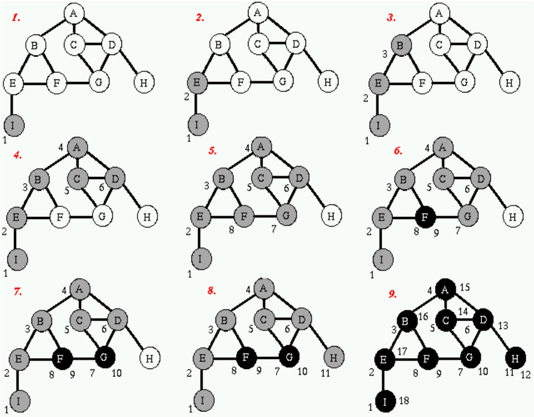
    $\mathit{dfs}(G, v)$ {run dfs on v}

  $\mathit{color}(u) = \mathit{black}$ {node u is visited}

Note: all nodes in G are marked white (unvisited)

Examples

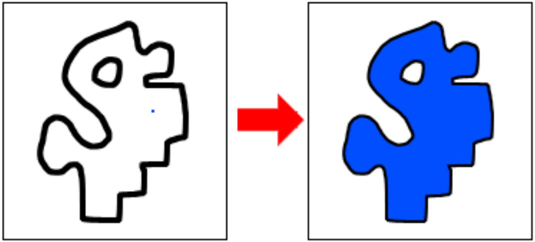
Problem: Depth First Search (DFS)



Problem: Find node with label ℓ with dfs

Function $dfs(G, u, \ell)$
  if $label(u) = \ell$ **then** {base case}
    return `true`
  else {recursive step}
    $color(u) = gray$ {node u is in progress}
    for each $\{u, v\} \in E$ **and** $color(v) = white$ **do**
      if $dfs(G, v, \ell) =$ `true` **then** {if dfs on v found the node}
        return `true` {stop recursion}
    $color(u) = black$ {node u is visited}
    return `false`

Problem: Flood Fill



Examples

Recursion: Visit neighbors of a cell that were not yet colored

Base case: All neighbors were already colored

Function *flood*(M, x, y)

- **if** $color(M[x][y]) = \texttt{true}$ **then** {base case}
  - **return**
- **else** {recursive step}
  - $paint(M, x, y)$ {paint in (x, y)}
  - $flood(M, x, y-1)$ {down}
  - $flood(M, x, y+1)$ {up}
  - $flood(M, x-1, y)$ {left}
  - $flood(M, x+1, y)$ {right}

Examples

Problem: Exploring a maze

	A	B	C	D	E	F	G	H
1	*	*	*	*	*			
2	*				*			
3	*	S	*	*	*			
4	*				*	*	*	*
5	*		*					*
6	*				*			*
7	*	*	*	*	*		E	*
8					*	*	*	*

Examples

```
Function Maze(M, x, y)
  if y > 8 or y < 1 or x <'A' or x >'H' then        {base case: limits}
     return false
  if M[x][y] = '*' then                             {base case: wall}
     return false
  if M[x][y] = 'E' then                             {base case: exit}
     return true
  M[x][y] = "*"
  if Maze(M, x, y − 1) = true then                  {recursive step: down}
     return true
  if Maze(M, x, y + 1) = true then                  {recursive step: up}
     return true
  if Maze(M, x − 1, y) = true then                  {recursive step: left}
     return true
  if Maze(M, x + 1, y) = true then                  {recursive step: right}
     return true
  return false
```

Examples

Generate all binary strings of size n

0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	0
0	0	0	1
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	1

Examples

A recursive algorithm to print all binary strings of size n

```
Function gen(n, S)
  if n = 0 then                          {base case}
    print S
  else
    S[n] = 0                             {put a 0}
    gen(n − 1, S)                        {1st recursive step}
    S[n] = 1                             {put a 1}
    gen(n − 1, S)                        {2nd recursive step}
    S[n] = " "                           {empty}
  return
```

What is the time complexity of this algorithm?