

Berner Fachhochschule - Technik und Informatik

Object-Oriented Programming 2

Topic 8: Recursion

Rolf Haenni & Andres Scheidegger

FS 2018

Outline

Recursive Algorithms

Examples

MiniMax Algorithm for Two-Player Games

ExpectedMax Algorithm for One-Player Games with Chance

Outline

Recursive Algorithms

Examples

MiniMax Algorithm for Two-Player Games

ExpectedMax Algorithm for One-Player Games with Chance

Divide and Conquer

- ▶ **Divide and conquer** is a powerful algorithm design paradigm
 - Break down a problem recursively into two or more sub-problems of the same (or related) type
 - Continue until the problem becomes simple enough to be solved directly
 - Merge the solutions of the sub-problems to get a solution for the original problem
- ▶ Benefits of divide and conquer
 - Powerful tool for solving conceptually difficult problems
 - Often provides a natural way to design efficient algorithms
 - Sub-problems can be executed in parallel
- ▶ **Decrease and conquer** is a variation of divide and conquer, where the problem is simplified into a single sub-problem

Recursive Algorithm

Recursive Algorithm

Algorithm that uses itself as part of the solution

Recursive Call

A method call in which the method being called is the same as the one making the call

Direct Recursion

Recursion in which a method directly calls itself

Indirect Recursion

Recursion in which a chain of two or more method calls returns to the method that originated the chain, e.g. A calls B, B calls C, and C calls A

Example: Factorial (Recursion)

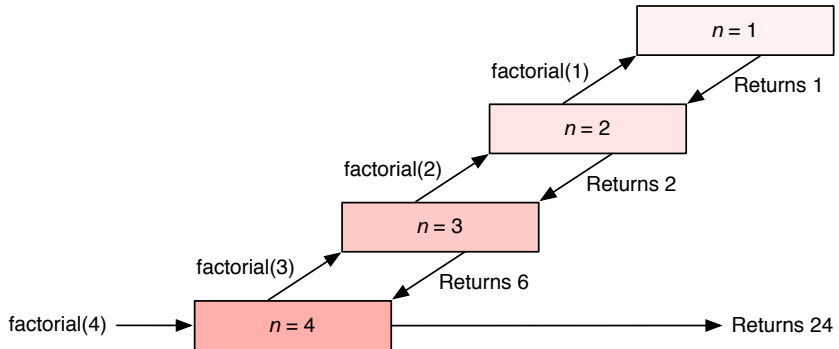
Compute the factorial of integer n :

$$n! = \underbrace{1 * 2 * \dots * n - 1}_{(n-1)!} * n = \begin{cases} 1 & n = 1 \\ n * (n - 1)! & n > 1 \end{cases}$$

```
algorithm factorial( $n$ )  
if  $n = 1$  then  
  return 1 // base case of the recursion  
else  
  return  $n * \text{factorial}(n - 1)$  // recursive call
```

Example: Factorial (Recursion)

The execution of a recursive algorithm uses a **call stack** to keep track of previous method call



Remarks

- ▶ Each recursion must have a **base case** to stop
- ▶ Each recursive call must reduce the problem size, i.e. leading inescapably to the base case
- ▶ A recursion is called **tail-recursive**, if the recursive call is always the last executed instruction
- ▶ Each recursive algorithm can be transformed into an iterative algorithm (and vice versa)
 - trivial for tail-recursions
 - not so easy in general (requires a **stack** to replace the call stack)
- ▶ In a recursive algorithm, the stack is hidden from the user

Iterative Algorithm

- ▶ In general, an **iterative** algorithm is a non-recursive one
- ▶ A typical iterative algorithm contains one or several, possibly nested loops (repetitions of instructions):
 - For ... Do
 - While ... Do
 - Repeat ... Until
- ▶ Iteration also stands for the style of programming used in **imperative** programming languages (e.g. Assembler, Basic, Pascal, C++, Java, ...)
- ▶ This contrasts with recursion, which is more declarative
 - **Functional** programming languages (e.g. Scheme, Haskell, ...)
 - **Logical** programming languages (e.g. Prolog)

Example: Factorial (Iteration)

The tail-recursive factorial algorithm can be replaced by a simple iterative loop

```
algorithm factorial(n)  
  result  $\leftarrow$  1  
  while n > 0 do  
    result  $\leftarrow$  result * n  
    n  $\leftarrow$  n - 1  
  return result
```

Recursive vs. Iterative Algorithms

Recursive algorithms are sometimes less efficient than their iterative counterparts

- ▶ The overhead of managing the call stack is non-negligible
- ▶ The space complexity is often $O(n)$ or $O(\log n)$ instead of $O(1)$
- ▶ Risk to get "stack_overflow" errors
- ▶ Some (functional) programming languages such as Scheme, Lisp, or Haskell detect tail-recursions automatically

Nevertheless, recursion has been used to solve some of the biggest and hardest algorithmic problems in computer science !!!

Outline

Recursive Algorithms

Examples

MiniMax Algorithm for Two-Player Games

ExpectedMax Algorithm for One-Player Games with Chance

Example 1: Fibonacci

Compute the Fibonacci number

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

Input Integer $n \geq 0$

Output $F(n)$

Examples

`fibonacci(10)` \rightarrow 55

`fibonacci(21)` \rightarrow 10946

Example 1: Fibonacci (cont.)

Pseudo-code solution:

```
algorithm fibonacci( $n$ )  
  if  $n \leq 1$  then  
    return  $n$   
  else  
    return fibonacci( $n - 1$ ) + fibonacci( $n - 2$ )
```

Example 2: Palindrome

Detect whether a string is a palindrome

Input

String s

Output

true or *false*

Examples

`palindrome("amanaplanacanalpanama")` \rightarrow *true*

`palindrome("recursioniscomplicated")` \rightarrow *false*

Auxiliary Functions

`size(s)`, `first(s)`, `last(s)`, `removeFirst(s)`, `removeLast(s)`

Example 2: Palindrome (cont.)

Pseudo-code solution:

```
algorithm palindrome(s)
  if size(s)  $\leq$  1 then
    return true
  else if first(s) = last(s) then
    return palindrome(removeFirst(removeLast(s)))
  else
    return false
```


Example 3: Integer Printing

Print integers relative to various bases

Input

Integer $n \geq 0$, base $b \geq 2$

Output

Screen output of $[d_k \cdots d_1 d_0]_b$ for $n = d_k b^k + \cdots + d_1 b + d_0$

Examples

`printInteger(61, 2)` \rightarrow 111101

`printInteger(61, 10)` \rightarrow 61

`printInteger(61, 16)` \rightarrow 3D

Auxiliary Function

`printDigit(d)`

Example 3: Integer Printing (cont.)

Pseudo-code solution:

```
algorithm printInteger( $n, b$ )  
  if  $n < b$  then  
    printDigit( $n$ )  
  else  
    printInteger( $\lfloor n/b \rfloor, b$ )  
    printDigit( $n \bmod b$ )
```

Example 4: Line Plotter

Plot a line between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$

Input

Points x_1, y_1, x_2, y_2

Output

Plot the connecting line on the screen

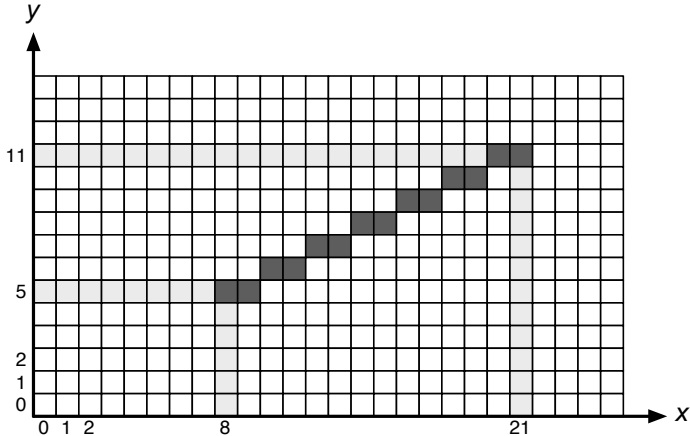
Example

`plotLine(8, 5, 21, 11)` → see next slide

Auxiliary Functions

`plotPixel(x, y)`

Example 4: Line Plotter (cont.)



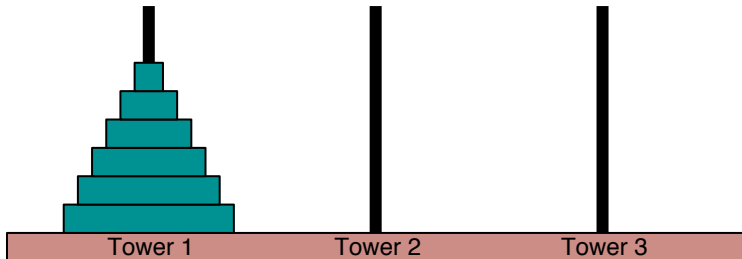
Example 4: Line Plotter (cont.)

Pseudo-code solution:

```
algorithm plotLine( $x_1, y_1, x_2, y_2$ )  
  if  $x_1 = x_2$  and  $y_1 = y_2$  then  
    plotPixel( $x_1, y_1$ )  
  else if  $|x_1 - x_2| \leq 1$  and  $|y_1 - y_2| \leq 1$  then  
    plotPixel( $x_1, y_1$ )  
    plotPixel( $x_2, y_2$ )  
  else  
     $x \leftarrow \lfloor (x_1 + x_2) / 2 \rfloor$   
     $y \leftarrow \lfloor (y_1 + y_2) / 2 \rfloor$   
    plotLine( $x_1, y_1, x, y$ )  
    plotLine( $x, y, x_2, y_2$ )
```

Example 5: Towers of Hanoi

Move all the blocks from the first to the third tower, one after another, and without ever putting a block on a smaller one



Example 5: Towers of Hanoi (cont.)

Input

Integer n (number of blocks)

Output

Print sequence of instructions on screen

Example

$\text{hanoi}(3) \rightarrow 1 \Rightarrow 3, 1 \Rightarrow 2, 3 \Rightarrow 2, 1 \Rightarrow 3, 2 \Rightarrow 1, 2 \Rightarrow 3, 1 \Rightarrow 3$

Auxiliary Functions

$\text{print}(t_1, t_2)$

Example 5: Towers of Hanoi (cont.)

Pseudo-code solution:

```
algorithm hanoi(n)  
  recHanoi(n, 1, 3)  
  
algorithm recHanoi(n, from, to)  
if n = 1 then  
  print(from, to)  
else  
  other  $\leftarrow 6 - (\textit{from} + \textit{to})$   
  recHanoi(n - 1, from, other)  
  recHanoi(1, from, to)  
  recHanoi(n - 1, other, to)
```


Outline

Recursive Algorithms

Examples

MiniMax Algorithm for Two-Player Games

ExpectedMax Algorithm for One-Player Games with Chance



The MiniMax-Algorithm

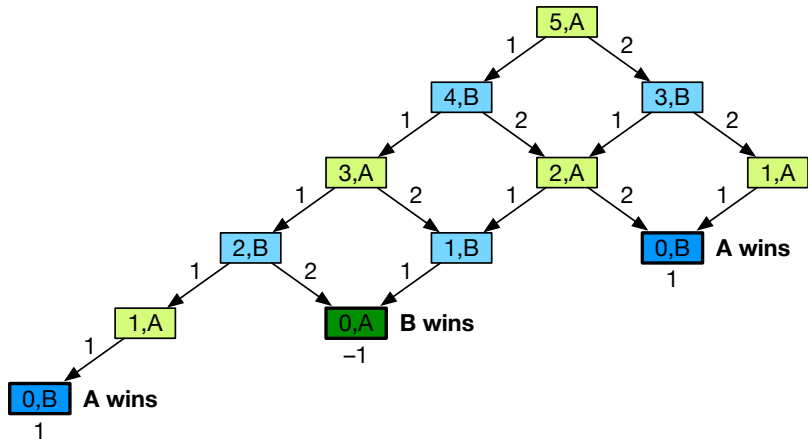
- ▶ The **MiniMax algorithm** is an optimal game playing algorithm for two player games such as Tic-Tac-Toe, Chess, Go, Uril, ...
- ▶ Let A and B be the two players and S the finite set of possible **game states**, where ...
 - $s_0 \in S$ is the **initial state**
 - $S^* \subseteq S$ are **final states**, in which the game ends
- ▶ For all final states $s^* \in S^*$, the winner of the game is defined by a function $E : S^* \rightarrow [-1, 1]$, where

$$E(s^*) = \begin{cases} 1, & \text{if } A \text{ wins} \\ 0, & \text{if the game ends as draw} \\ -1, & \text{if } B \text{ wins} \end{cases}$$

Example: Subtraction Game

- ▶ In a (k, n) -subtraction game, A and B take turns in removing up to k objects from a pile with initially n objects
 - A begins
 - Whoever removes the last object from the pile wins
- ▶ Example: $(2, 5)$ -subtraction game
 - States: $S = \{(5, A), (3, A), \dots, (0, A), (4, B), \dots, (0, B)\}$
 - Initial state: $s_0 = (5, A)$
 - Final states: $S^* = \{(0, A), (0, B)\}$
 - $$E(s^*) = \begin{cases} 1, & \text{for } s^* = (0, B) \\ -1, & \text{for } s^* = (0, A) \end{cases}$$

Example: Subtraction Game



MiniMax Algorithm: General Idea

- ▶ Let $next(s) \subseteq S$ be the reachable states from $s \in S$
- ▶ Example: (2,5)-subtraction game
 - $next((5, A)) = \{(4, B), (3, B)\}$
 - $next((4, B)) = \{(3, A), (2, A)\}$
 - \vdots
 - $next((0, A)) = next(0, B) = \{\}$
- ▶ The MiniMax algorithm extends E from $E : S^* \rightarrow [-1, 1]$ to $E' : S \rightarrow [-1, 1]$ by computing $E'(s)$ recursively for all $s \in S$

$$E'(s) = \begin{cases} E(s), & \text{if } s \in S^* \\ \max\{E'(s') : s' \in next(s)\}, & \text{if it is } A\text{'s turn} \\ \min\{E'(s') : s' \in next(s)\}, & \text{if it is } B\text{'s turn} \end{cases}$$

MiniMax Algorithm: Pseudocode

Algorithm: MiniMax(s, d)

if $s \in S^*$ **then** // game ends
 | **return** $E(s)$

if $d \bmod 2 = 0$ **then** // A's turn

 | $m \leftarrow -1$

 | **for** $s' \in next(s)$ **do**

 | $m \leftarrow \max(m, \text{MiniMax}(s', d + 1))$

else // B's turn

 | $m \leftarrow 1$

 | **for** $s' \in next(s)$ **do**

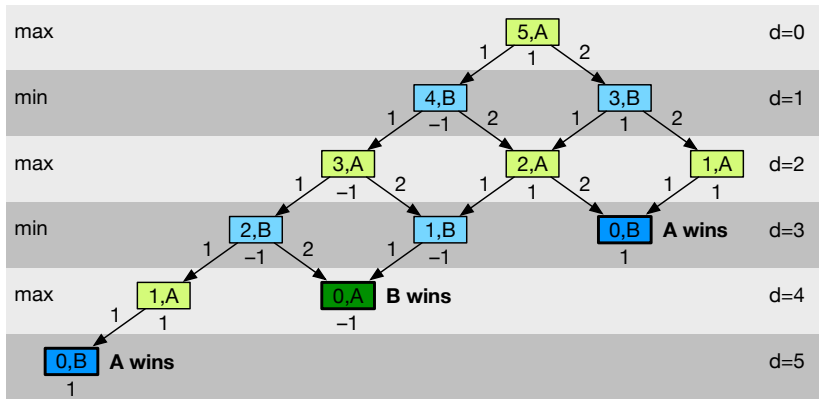
 | $m \leftarrow \min(m, \text{MiniMax}(s', d + 1))$

return m

Example: Subtraction Game

Initial call: $\text{MiniMax}((5, A), 0)$

Return value: $E'((5, A)) = 1$, i.e. A wins by removing 2 objects



MiniMax Algorithm: Performance

- ▶ The **branching factor** b of a game is the average number legal moves (children in the game tree)
- ▶ Examples:
 - (k, n) -subtraction game: $b \leq k, h \leq n$
 - Chess: $b \approx 35, h \approx 80$
 - Go: $b \approx 250$
- ▶ The MiniMax algorithms runs in $O(b^h)$ time, where h denotes the height (maximal depth) of the game tree
- ▶ In other words, exploring the full game tree is impossible for most non-trivial games

Pruned MiniMax Algorithm

- ▶ To apply the MiniMax algorithm to non-trivial games, the game tree exploration must be pruned
- ▶ The simplest pruning method is to stop the recursion when a **maximal depth** d_{\max} is reached
- ▶ When the recursion stops at state $s \in S$, then a **evaluation function** $\tilde{E} : S \rightarrow [-1, 1]$ is applied to s
 - For $s \in S^*$, let $\tilde{E}(s) = E(s)$
 - Otherwise, let $\tilde{E}(s)$ be an estimate of the **advantage** of state s relative to A and B , such that $\tilde{E}(s) = 1$ means maximal advantage for A and $\tilde{E}(s) = -1$ maximal advantage for B
- ▶ The quality of the estimate $\tilde{E}(s)$ and d_{\max} determine the quality and accuracy of the final MiniMax return value

Pruned MiniMax Algorithm: Pseudocode

Algorithm: MiniMax(s, d, d_{\max})

if $s \in S^*$ **or** $d = d_{\max}$ **then** // game ends

return $\tilde{E}(s)$

if $d \bmod 2 = 0$ **then** // A's turn

$m \leftarrow -1$

for $s' \in \text{next}(s)$ **do**

$m \leftarrow \max(m, \text{MiniMax}(s', d + 1, d_{\max}))$

else // B's turn

$m \leftarrow 1$

for $s' \in \text{next}(s)$ **do**

$m \leftarrow \min(m, \text{MiniMax}(s', d + 1, d_{\max}))$

return m

Defining the Evaluation Function

- ▶ One popular strategy for constructing an evaluation function $\tilde{E} : S \rightarrow [-1, 1]$ is as a **weighted sum**

$$\tilde{E}(s) = \frac{1}{W} \sum_{i=1}^k w_i \cdot \tilde{E}_i(s)$$

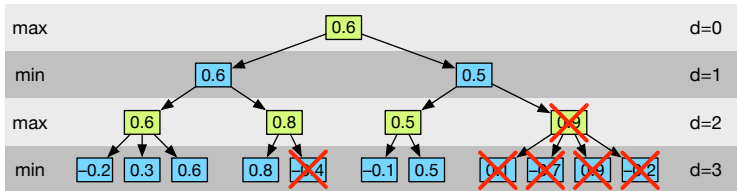
of k individual evaluation criteria

- ▶ The value $w_i \in [0, 1]$ denotes the **weight** of criterion i and $W = \sum_{i=1}^k w_i$ denotes the total weight of all criteria
- ▶ $\tilde{E}_i : S \rightarrow [-1, 1]$ defines the evaluation function of criterion i
- ▶ Example from chess:

$$\tilde{E}(s) = 9 \cdot (Q - Q') + 5 \cdot \frac{R - R'}{2} + 3 \cdot \frac{B - B'}{2} + 3 \cdot \frac{N - N'}{2} + \frac{P - P'}{8} + \dots$$

Alpha-Beta Pruning

- ▶ There are many ways of optimizing the MiniMax algorithm
- ▶ The general idea is to prune branches of the game tree that will not influence the final MiniMax return value
- ▶ The simplest optimization is known as **alpha-beta pruning**



- ▶ The following version of the MiniMax algorithm is initially called with $MiniMax(s_0, 0, d_{\max}, -1, 1)$

Alpha-Beta Pruning: Pseudocode I

Algorithm: MiniMax($s, d, d_{\max}, \alpha, \beta$)

if $s \in S^*$ **or** $d = d_{\max}$ **then** // game ends

return $\tilde{E}(s)$

if $d \bmod 2 = 0$ **then** // A's turn

$m \leftarrow -1$

for $s' \in \text{next}(s)$ **do**

$m \leftarrow \max(m, \text{MiniMax}(s', d + 1, d_{\max}, \alpha, \beta))$

$\alpha \leftarrow \max(\alpha, m)$

if $\alpha \geq \beta$ **then**

return m // β cutoff

Alpha-Beta Pruning: Pseudocode II

```
else // B's turn
   $m \leftarrow 1$ 
  for  $s' \in \text{next}(s)$  do
     $m \leftarrow \min(m, \text{MiniMax}(s', d + 1, d_{\max}, \alpha, \beta))$ 
     $\beta \leftarrow \min(\beta, m)$ 
    if  $\alpha \geq \beta$  then
      return  $m$  //  $\alpha$  cutoff
  return  $m$ 
```

Outline

Recursive Algorithms

Examples

MiniMax Algorithm for Two-Player Games

ExpectedMax Algorithm for One-Player Games with Chance

One-Player Games

- ▶ In one-player games, the goal is often to maximize the value $E(s^*)$ when reaching a final state $s^* \in S^*$, for

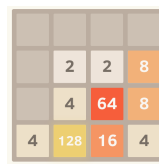
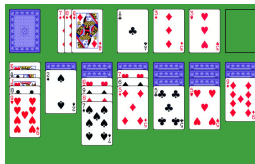
$$E : S^* \rightarrow \mathbb{R}^+$$

- ▶ The highest such value reached in all previously played games is called **high score**
- ▶ A **puzzle** is a one-player game with complete information and no chance
- ▶ Examples: Peg Solitaire, Rubik's Cube, Rush Hour, ...



One-Player Games with Chance

- ▶ Adding chance (or hidden information) makes one-player games more interesting
- ▶ Examples: Minesweeper, Solitaire, 2048, ...



- ▶ Probabilistic events can be seen as an auxiliary player playing **chance moves**, each leading to a new state $s' \in \text{next}(s)$
- ▶ For $n = |\text{next}(s)|$ different chance moves, let $0 < p(s'_i) \leq 1$ be the probability of $s'_i \in \text{next}(s)$ to occur, i.e. $\sum_{i=1}^n p(s'_i) = 1$

ExpectedMax Algorithm

- ▶ Let's assume that a chance move takes place after every move by the player (if not, think of a chance move with $p(s') = 1$)
- ▶ The ExpectedMax algorithm extends E from $E : S^* \rightarrow \mathbb{R}^+$ to $E' : S \rightarrow \mathbb{R}^+$ by computing $E'(s)$ recursively for all $s \in S$

$$E'(s) = \begin{cases} E(s), & \text{if } s \in S^* \\ \max\{E'(s') : s' \in \text{next}(s)\}, & \text{if it is the player's turn} \\ \sum\{p(s') \cdot E'(s') : s' \in \text{next}(s)\}, & \text{if it is a chance move,} \end{cases}$$

where $\sum p(s') \cdot E'(s')$ is the **expected value** of the game state before a chance move

- ▶ Note that chance moves can also be added to two-player or multi-player games

ExpectedMax Algorithm: Pseudocode

Algorithm: ExpectedMax(s, d)

```
if  $s \in S^*$  then                                     // game ends
  | return  $E(s)$ 
if  $d \bmod 2 = 0$  then                                   // Player's turn
  |  $m \leftarrow 0$ 
  | for  $s' \in next(s)$  do
  | |  $m \leftarrow \max(m, \text{ExpectedMax}(s', d + 1))$ 
else                                                  // Chance move
  |  $e \leftarrow 0$ 
  | for  $s' \in next(s)$  do
  | |  $e \leftarrow e + p(s') \cdot \text{ExpectedMax}(s', d + 1)$ 
return  $m$ 
```

Pruned ExpectedMax Algorithm: Pseudocode

```

Algorithm: ExpectedMax( $s, d, d_{\max}$ )

if  $s \in S^*$  or  $d = d_{\max}$  then                                // game ends
  | return  $\tilde{E}(s)$ 
if  $d \bmod 2 = 0$  then                                           // Player's turn
  |  $m \leftarrow 0$ 
  | for  $s' \in next(s)$  do
  | |  $m \leftarrow \max(m, \text{ExpectedMax}(s', d + 1, d_{\max}))$ 
else                                                            // Chance move
  |  $e \leftarrow 0$ 
  | for  $s' \in next(s)$  do
  | |  $e \leftarrow e + p(s') \cdot \text{ExpectedMax}(s', d + 1, d_{\max})$ 
return  $m$ 
  
```