Recursion

Saikrishna Arcot
M. Hudachek-Buswell

July 18, 2020

Recursion Definition

Recursion is a programming process where a method calls itself repeatedly until it reaches a defined point of termination.

Recursive Example

Classic example - the factorial function:

- $n = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$

Recursive Example

Classic example - the factorial function:

- $n = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$
- Recursive definition:
$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & \text{else} \end{cases}$$

Recursive Example

Classic example - the factorial function:

- $n = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & \text{else} \end{cases}$$

- As a Java method:

```java
public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("arg must be nonnegative");
    } else if (n == 0) {
        return 1;                        // base case
    } else {
        return n * factorial(n - 1);     // recursive case
    }
}
```

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
  - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
    - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.
    - Every recursive method must have at least one base case.

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
    - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.
    - Every recursive method must have at least one base case.
    - Every possible chain of recursive calls **must** eventually reach a base case.

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
    - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.
    - Every recursive method must have at least one base case.
    - Every possible chain of recursive calls **must** eventually reach a base case.
- Recursive calls

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
  - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.
  - Every recursive method must have at least one base case.
  - Every possible chain of recursive calls **must** eventually reach a base case.
- Recursive calls
  - Explicit calls to the current method.

Construction of a Recursive Method

The recursive method will have selection statements for termination and recursive calls that involve a parameter advancing towards the base case.

- Base case(s)
  - Values of the parameter(s) for which there is no recursive call are referred to as **base case(s)**.
  - Every recursive method must have at least one base case.
  - Every possible chain of recursive calls **must** eventually reach a base case.
- Recursive calls
  - Explicit calls to the current method.
  - Each recursive call **must** be defined so that it advances towards a base case. These calls can occur without any parameter(s).

Visualizing Recursion

Recursion Trace

- A box for each recursive call

Visualizing Recursion

Recursion Trace

- A box for each recursive call
- An arrow from each caller to callee

Visualizing Recursion

Recursion Trace

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing the return value.

Visualizing Recursion

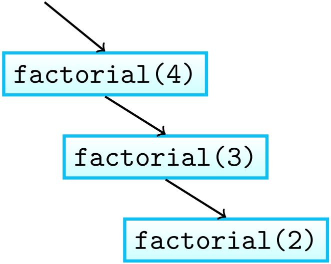
`factorial(4)`

Visualizing Recursion

factorial(4)

factorial(3)

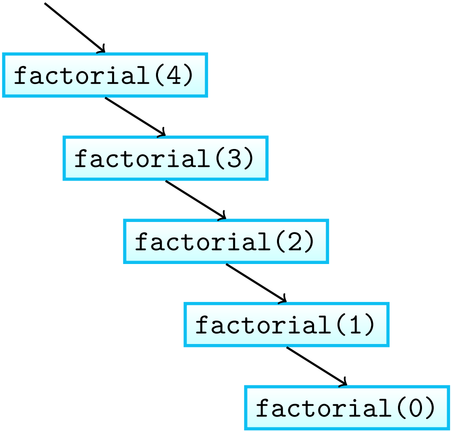
Visualizing Recursion



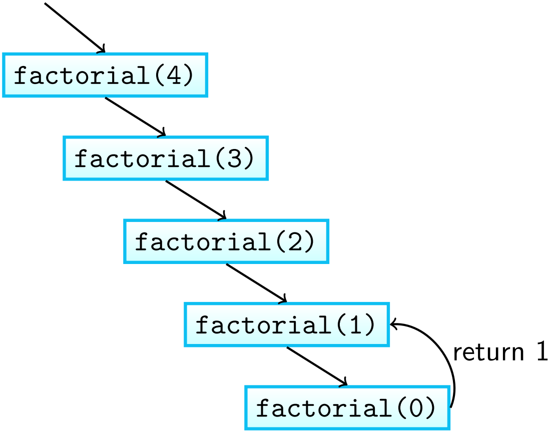
Visualizing Recursion

```
factorial(4)
  factorial(3)
    factorial(2)
      factorial(1)
```

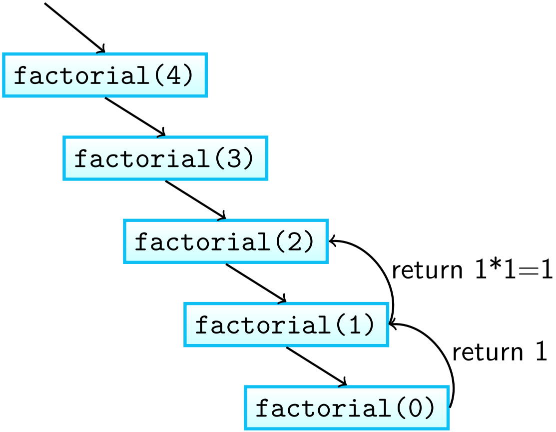
Visualizing Recursion



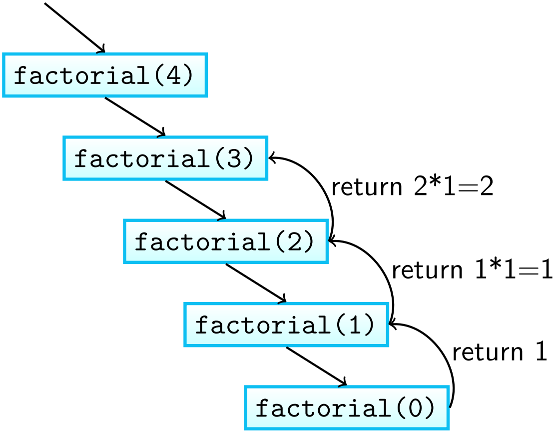
Visualizing Recursion



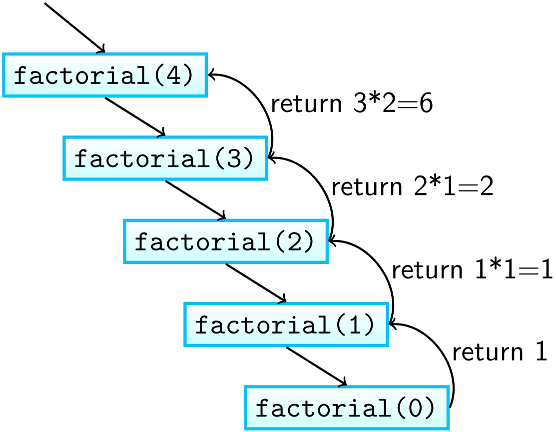
Visualizing Recursion



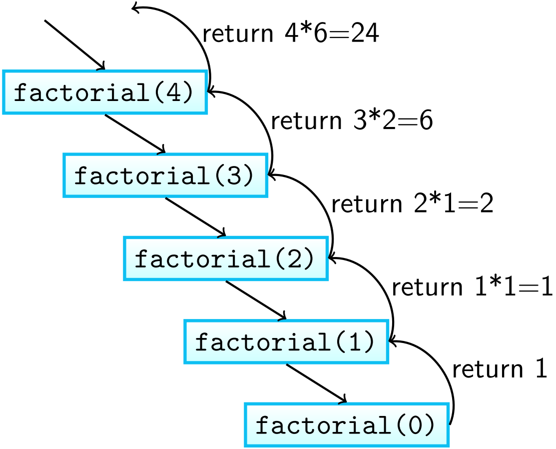
Visualizing Recursion



Visualizing Recursion



Visualizing Recursion



Binary Search Method

Search for an integer in an ordered list

```
procedure BinarySearch(data, target, low, high)
    if low > high then
        return False
    else
        mid ← (low + high)/2
        if target = data[mid] then
            return True
        else if target < data[mid] then
            return BinarySearch(data, target, low, mid − 1)
        else
            return BinarySearch(data, target, mid + 1, high)
        end if
    end if
end procedure
```

Binary Search Example

- We consider three cases:

Binary Search Example

- We consider three cases:
  - If the target equals `data[mid]`, then we have found the target.

Binary Search Example

- We consider three cases:
  - If the target equals `data[mid]`, then we have found the target.
  - If `target < data[mid]`, then we recursively call the method on the first half of the sequence.

Binary Search Example

- We consider three cases:
  - If the target equals `data[mid]`, then we have found the target.
  - If `target < data[mid]`, then we recursively call the method on the first half of the sequence.
  - If `target > data[mid]`, then we recursively call the method on the second half of the sequence.

Visualizing Binary Search Example

(Blue represents the area being considered, while green represents the element that is checked.)

Searching for 19:

2	4	5	7	8	9	12	14	17	19	22	25	27

Visualizing Binary Search Example

(Blue represents the area being considered, while green represents the element that is checked.)

Searching for 19:

2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27

Visualizing Binary Search Example

(Blue represents the area being considered, while green represents the element that is checked.)

Searching for 19:

2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27

Visualizing Binary Search Example

(Blue represents the area being considered, while green represents the element that is checked.)

Searching for 19:

2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27
2	4	5	7	8	9	12	14	17	19	22	25	27

Analyzing Binary Search

- Runs in $O(\log n)$ time.

Analyzing Binary Search

- Runs in $O(\log n)$ time.
  - The portion of the list being searched is of size `high - low + 1`.

Analyzing Binary Search

- Runs in $O(\log n)$ time.
  - The portion of the list being searched is of size `high - low + 1`.
  - After one comparison, this becomes one of the following:

Analyzing Binary Search

- Runs in $O(\log n)$ time.
  - The portion of the list being searched is of size `high - low + 1`.
  - After one comparison, this becomes one of the following:
    - $(mid - 1) - low + 1 = \lfloor \frac{low+high}{2} \rfloor - low \leq \frac{high-low+1}{2}$

Analyzing Binary Search

- Runs in $O(\log n)$ time.
  - The portion of the list being searched is of size `high - low + 1`.
  - After one comparison, this becomes one of the following:
    - $(mid - 1) - low + 1 = \lfloor \frac{low+high}{2} \rfloor - low \leq \frac{high-low+1}{2}$
    - $high - (mid - 1) + 1 = high - \lfloor \frac{low+high}{2} \rfloor \leq \frac{high-low+1}{2}$

Analyzing Binary Search

- Runs in $O(\log n)$ time.
  - The portion of the list being searched is of size `high - low + 1`.
  - After one comparison, this becomes one of the following:
    - $(mid - 1) - low + 1 = \lfloor \frac{low+high}{2} \rfloor - low \leq \frac{high-low+1}{2}$
    - $high - (mid - 1) + 1 = high - \lfloor \frac{low+high}{2} \rfloor \leq \frac{high-low+1}{2}$
  - Thus, each recursive call divides the search region in half; hence, there can be at most $\log n$ levels.

Computing Powers Example

- The power function, $p(a, n) = a^n$, can be defined recursively:
$$p(a, n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times p(a, n-1) & \text{else} \end{cases}$$

Computing Powers Example

- The power function, $p(a, n) = a^n$, can be defined recursively:
$$p(a, n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times p(a, n-1) & \text{else} \end{cases}$$
- This leads to an power function that runs in $O(n)$ time (because we make n recursive calls).

Computing Powers Example

- The power function, $p(a, n) = a^n$, can be defined recursively:
$$p(a, n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times p(a, n-1) & \text{else} \end{cases}$$
- This leads to an power function that runs in $O(n)$ time (because we make n recursive calls).
- We can do better than this, however.

Recursive Squaring Example

- We can derive a more efficient recursive algorithm by using repeated squaring:

$$p(a,n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times p(a,(n-1)/2)^2 & \text{if} a > 0 \text{ is odd} \\ p(a,(n-1)/2)^2 & \text{if} a > 0 \text{ is even} \end{cases}$$

Recursive Squaring Example

- We can derive a more efficient recursive algorithm by using repeated squaring:

$$p(a,n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times p(a,(n-1)/2)^2 & \text{if} a > 0 \text{ is odd} \\ p(a,(n-1)/2)^2 & \text{if} a > 0 \text{ is even} \end{cases}$$

- For example (knowing beforehand that $2^2 = 4$),

$$2^4 = 2^{(4/2)^2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)^2} = 2 \times (2^{4/2})^2 = 2 \times (2^2)^2 = 2 \times 4^2 = 32$$
$$2^6 = 2^{(6/2)^2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)^2} = 2 \times (2^{6/2})^2 = 2 \times (2^3)^2 = 2 \times 8^2 = 128$$

Recursive Squaring Method

procedure $\textsc{Power}(a, n)$
    if $n = 0$ **then**
        return 1
    end if
    if n is odd **then**
        $y \leftarrow \textsc{Power}(a, (n-1)/2)$
        return $a \times y \times y$
    else
        $y \leftarrow \textsc{Power}(a, n/2)$
        return $y \times y$
    end if
end procedure

Tail Recursion

This is a special form of recursion, that can be interpreted as iteration, avoiding inefficiencies.

- Tail recursion is when every recursive method call is the very last step before returning from the method.

Tail Recursion

This is a special form of recursion, that can be interpreted as iteration, avoiding inefficiencies.

- Tail recursion is when every recursive method call is the very last step before returning from the method.
- There is no need to store anything on the call stack because the result is returned to the original method call.

Tail Recursion

This is a special form of recursion, that can be interpreted as iteration, avoiding inefficiencies.

- Tail recursion is when every recursive method call is the very last step before returning from the method.
- There is no need to store anything on the call stack because the result is returned to the original method call.
- Such methods might receive some optimization benefits at runtime.

Tail Recursion

This is a special form of recursion, that can be interpreted as iteration, avoiding inefficiencies.

- Tail recursion is when every recursive method call is the very last step before returning from the method.
- There is no need to store anything on the call stack because the result is returned to the original method call.
- Such methods might receive some optimization benefits at runtime.
  - Note that Java doesn't perform any optimizations for tail recursion.