Removing recursion

- We can remove recursion by using an explicit stack
- Some recursions are very easy to remove
  - fibonacci, factorial etc
  - tail recursion
- Others are a bit more difficult
- All can be removed by a stack
- Function calls and the execution stack

Execution stack

- Executing code:
  - `do_this ;`
  - `x := 10 ;`
  - `...`
  - `n := square( n );`
  - `...`
  - `do_that ;`

- Where the function `square( )` is:
  - `square( n ):`
  - `x := n*n`
  - `return x`

Function calls

- Any function calls requires
  - set the values of the parameters
  - save the "environment"
  - execute the function
  - restore the environment
  - continue execution

Function calls

- Function call
  - save the "environment" -- PUSH operation
  - execute the function
  - restore the environment -- POP operation
  - continue execution
- We need a stack for nested function calls

Factorial example

- The 'environment' here is just 'n'
- Execute fact(3)
Factorial example

```plaintext
fact(3):
  if (n=1) return 1
  return fact(2) * 3
```

Factorial example

```plaintext
fact(n:=3):
  if (n=1) return 1
  return fact(n:=2) * n
```

Factorial example

```plaintext
fact(n:=3):
  if (n=1) return 1
  return fact(n:=2) * n
```

Factorial example

```plaintext
fact(n:=3):
  if (n=1) return 1
  return fact(n:=2) * n
```
Factorial example

```
fact(n:=3):
    if (n=1) return 1
    return fact(n:=2) * n
```

Execution stack

```
fact(n)
n=3
```

Current environment

```
fact(n)
n=2
```

Removing recursion

- Recursive code
  ```
  base case;
  some code;
  recursion;
  more code;
  ```
- Code without recursion, using a stack
  ```
  repeat {
    some code;
    push environment;
    until base case is reached;
  }
  base case;
  repeat {
    pop environment;
    more code;
    until stack is empty;
  }
```
Example of removing recursion

- Recursive program: reversing a string

```java
backward(string):
    if (string is empty) return
    first := first character of string
    rest := string minus its first character
    backward(rest)
    print first
```

- Environment: < first, rest >

Example of removing recursion

```java
backward(string):
    if (string is empty) return
    first := first char
    rest := string minus first
    backward(rest)
    print first
```

- Environment: < first, rest >

Example of removing recursion

```java
backward(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    } /* base case: do nothing */
    while (stack is not empty) {
        <first, rest> = pop()
        print first
    }
```

Example of removing recursion

```java
backward(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    } /* base case: do nothing */
    while (stack is not empty) {
        <first, rest> = pop()
        print first
    }
```
Example of removing recursion

```
Example of removing recursion
backwards(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    }
/* base case: do nothing */
while (stack is not empty) {
    <first, rest> = pop()
    print first
}
```

Example of removing recursion

```
Example of removing recursion
backwards(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    }
/* base case: do nothing */
while (stack is not empty) {
    <first, rest> = pop()
    print first
}
```

Example of removing recursion

```
Example of removing recursion
backwards(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    }
/* base case: do nothing */
while (stack is not empty) {
    <first, rest> = pop()
    print first
}
```

Example of removing recursion

```
Example of removing recursion
backwards(string):
    while (string is not empty) {
        first := first char
        rest := string minus first
        string := rest
        push( <first, rest> )
    }
/* base case: do nothing */
while (stack is not empty) {
    <first, rest> = pop()
    print first
}
```

Tail recursion

- Recursive call is the last action
- We do not need a stack to remove tail recursion
- There is nothing to be done when we return from a recursive call
- No need to save the environment
Example of tail recursion

- This is NOT tail recursion
  \[\text{fact}(n):\]
  \[
  \begin{align*}
  &\text{if } n = 1 \text{ return } 1 \\
  &\text{return } \text{fact}(n-1) \times n
  \end{align*}
  \]
- This is tail recursion
  \[\text{fact}(m,n):\]
  \[
  \begin{align*}
  &\text{if } n = 1 \text{ return } m \times 1 \\
  &\text{return } \text{fact}(m \times n, n-1)
  \end{align*}
  \]

Example of tail recursion

- This is not tail recursion
  \[\text{backwards}(\text{string}):\]
  \[
  \begin{align*}
  &\text{if } \text{string is empty} \text{ return } \\
  &\text{first} := \text{first char} \\
  &\text{rest} := \text{string minus first} \\
  &\text{backwards}(\text{rest}) \\
  &\text{print first}
  \end{align*}
  \]
- This is tail recursion
  \[\text{backwards}(\text{string}):\]
  \[
  \begin{align*}
  &\text{if } \text{string is empty} \text{ return } \\
  &\text{last} := \text{last char} \\
  &\text{rest} := \text{string minus last} \\
  &\text{print last} \\
  &\text{backwards}(\text{rest})
  \end{align*}
  \]

Run-time complexity

- Running time, scalability, complexity
- Worst case running time
- Worst case number of steps
- Example: bubble sort

Analyzing bubble sort

\[\text{array elements}[1..N]\]
\[\text{for } j := 1 \text{ to } N \text{ do} \]
\[\text{for } k := j+1 \text{ to } N \text{ do} \]
\[\text{if } \text{elements}[k] > \text{elements}[k+1] \text{ then} \]
\[\text{swap}(k,k+1, \text{elements}) \]
\[\text{end // if} \]
\[\text{end // for } k \text{ loop} \]
\[\text{end // for } j \text{ loop} \]

Different bubble-sort

\[\text{array elements}[1..N]\]
\[\text{swapDone} = \text{true}\]
\[\text{while } \text{swapDone} \text{ do} \]
\[\text{swapDone} = \text{false}\]
\[\text{for } k := 1 \text{ to } N-1 \text{ do} \]
\[\text{if } \text{elements}[k] > \text{elements}[k+1] \text{ then} \]
\[\text{swap}(k,k+1, \text{elements}) \]
\[\text{swapDone} = \text{true} \]
\[\text{end // if} \]
\[\text{end // for } k \text{ loop} \]
\[\text{end // while loop} \]
Bubble-sort

• Need to look at the algorithm
• Find properties for finishing the loop
  – no swaps done
• Why is the worst case complexity still $O(n^2)$?
• Is this a better solution than nested for loops?