Week 10 - Algorithms and Recursion

Overview

In this lecture we look at how to compare algorithms in relation to performance.

We look at some common algorithms in relation to some common classifications for performance, in particular computational complexity.

We also look at recursion as an appropriate approach to some types of problems.

What is an algorithm?

An algorithm is formally defined as (Knuth, 1968):

1. Finite (it stops);

2. Definite (precise, not subjective);

3. An Effective procedure (i.e can be carried out); and

4. Produces output (has some result or effect)

5. Takes input.

Knuth, D.E 1968, Fundamental Algorithms: The Art of Computer Programming, Vol 1, Addison-Wesley, London

Examples of algorithms (or not?)

Question

Is each of the following an algorithm:

- ☐ Test each integer to see if it is prime
- ☐ Count each grain of sand on Bondi beach.
- ☐ Beat the eggs until fluffy
- ☐ Calculate the constant PI to 5 decimal places.

Computational Complexity I

- We are interested in how efficient programs are.
- Inefficiency might be ok for small data sets, but a big problem for large data sets.
- It depends on how ‘quickly’ or seriously performance degrades as the data set increases.

Computational Complexity II

There are some general categories for classifying how different algorithms performance compares as the data set size increases. These are as follows:

$O(n^2)$ Algorithms

$O(n)$ Algorithms

$O(log_n)$ Algorithms

Computational Complexity III

- $O(n^2)$ **Algorithms**
    - the time (number of lines of code executed) increases quadratically in proportion to the size of the data set (N). This type of pattern can extend to exponential eg: $O(c^n)$ - where c is a constant.
- $O(n)$ **Algorithms**
    - The time (number of lines of code executed) increases in direct proportion to the size of the data set to N.
- $O(logn)$ **Algorithms**
    - The time (number of lines of code executed) increases logarithmically in proportion to the size of the data set.

An Example

Lets try bubble sort with 100 items, then change NUM_ITEM to 1,000, then 10,000.

For each of the above check the number of comparisons - what pattern do you see?

Run RUBY

```ruby
# Code for bubble sort
NUM_ITEM = 100
# Max VALUE of 100%
MAX_VALUE = 100
num_compare = 0
arr = Array.new(NUM_ITEM)

# Randomly put some final exam VALUEs into arr

for i in (0..NUM_ITEM - 1)
  arr[i] = rand(MAX_VALUE + 1)
end

# Output randomly generated array
```

Complexity Quiz

Question

Look at the complexity graphs - rank the list below so as to go from most efficient (at the top) to least efficient (at the bottom):

$O(n^2)$

$O(n)$

$O(n \log n)$

Recursion

Recursion is a process of looping where a program/procedure/function calls itself.

Recursion Structure

Recursion needs to have the elements of a loop:

1. initialisation
2. repetition/incrementing
3. Termination condition

Loops (revision)

A loop like the following has a control variable:

Loop Example

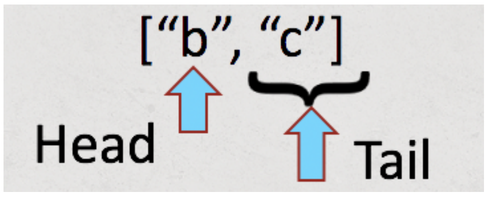
```ruby

def print_array(array)
  index = 0
  while index < array.length
      puts array[index]
      index += 1
  end
  index
end

def main
  array = [10, 2, 5, 7, 6]
  count = print_array(array)
  puts "The array contained " + count.to_s + " elements."
```

Lists

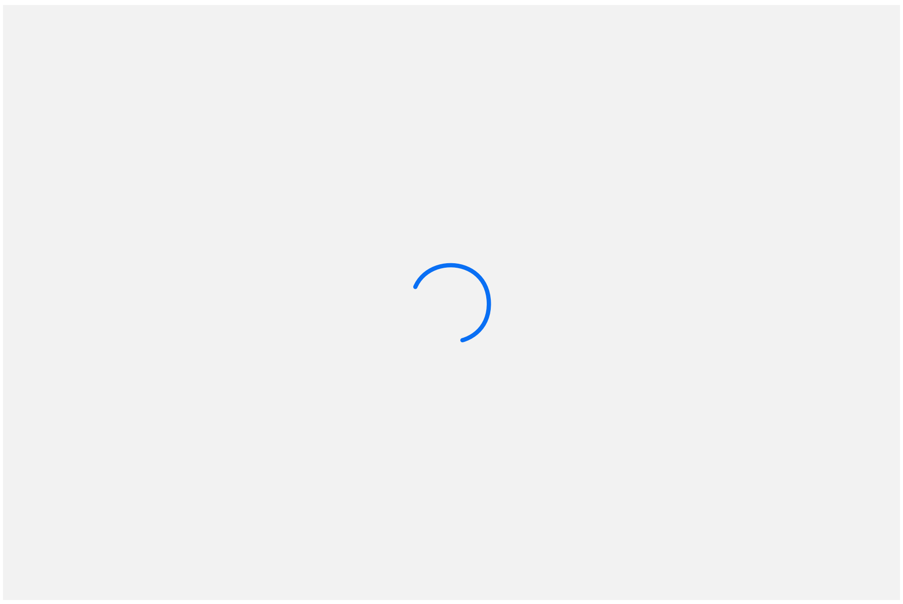
In Ruby we can look at arrays as lists.

In computing lists are considered as having a head and a tail eg:

Then the tail can be seen as another ‘sub-list”:



Recursion Example using a list



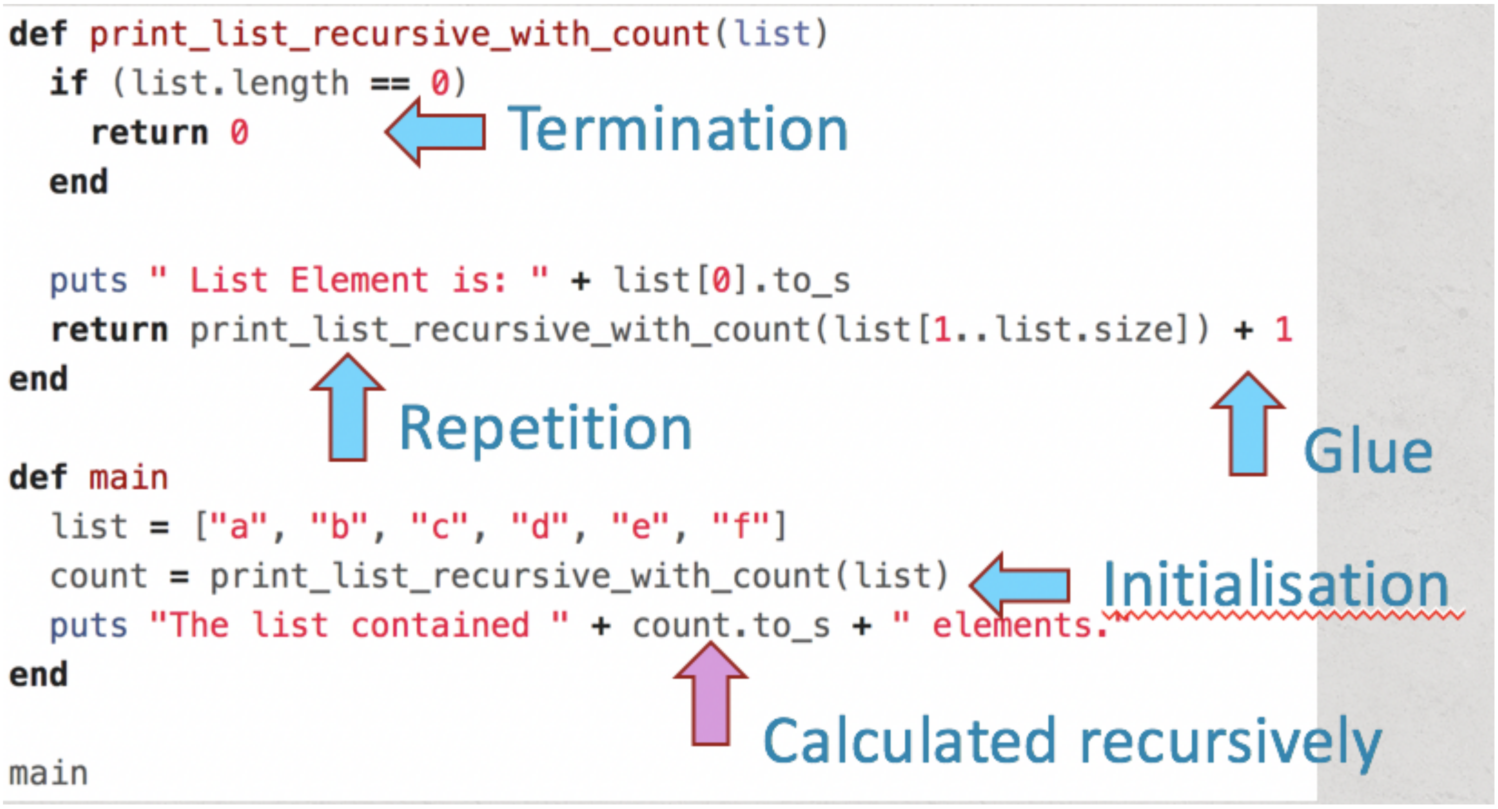
Run RUBY

```ruby
# Print array and return count of elements calculated
# as we print them

def print_list_recursive(list)
  if (list.length == 0)
    return
  end

  puts " List Element is: " + list[0].to_s
  print_list_recursive(list[1..list.size])
end

def main
  list = ["a", "b", "c", "d", "e", "f"]
```

Recursive Calculations

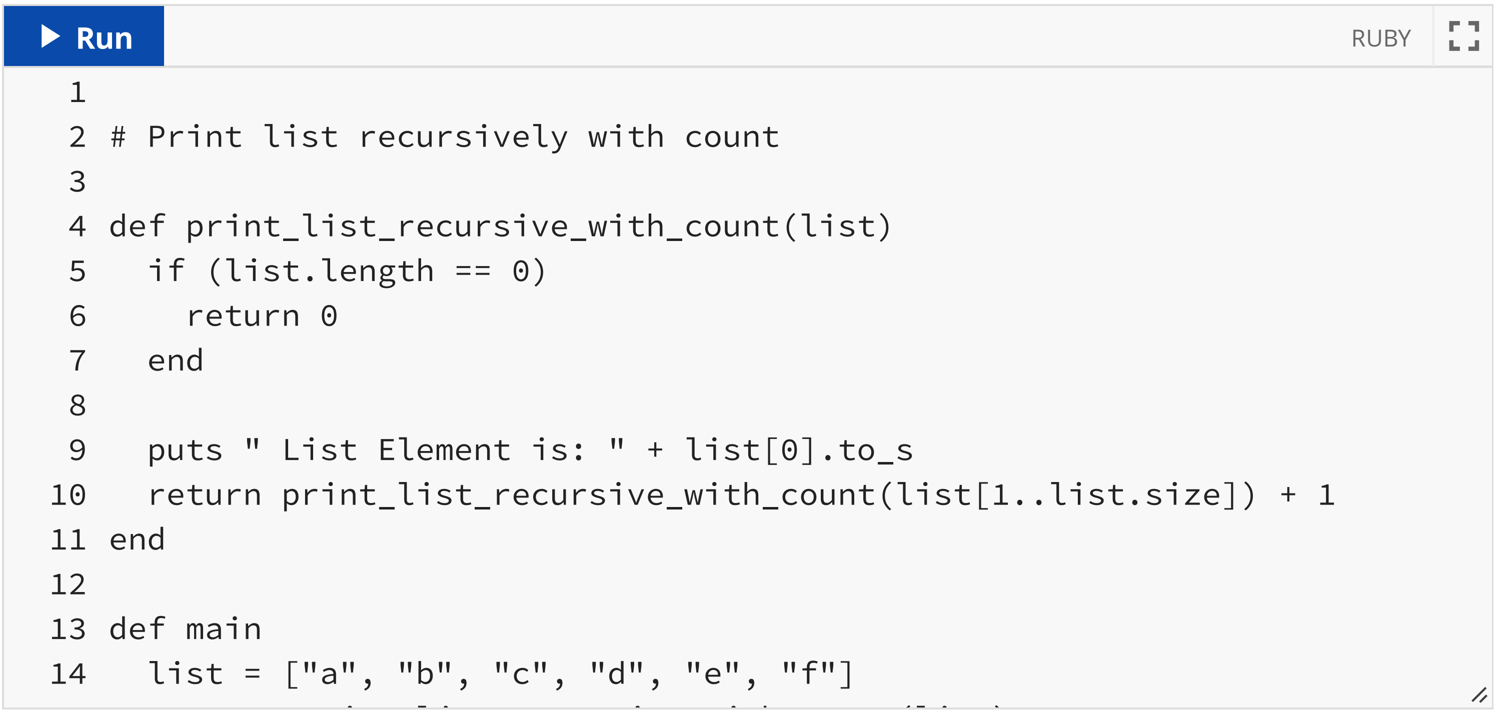
But printing is just output – when doing calculations or generating something (like a sum) we need an additional element for our recursion:

"the glue":



```
def print_list_recursive_with_count(list)
  if (list.length == 0)
    return 0
  end

  puts " List Element is: " + list[0].to_s
  return print_list_recursive_with_count(list[1..list.size]) + 1
end

def main
  list = ["a", "b", "c", "d", "e", "f"]
  count = print_list_recursive_with_count(list)
  puts "The list contained " + count.to_s + " elements."
end

main
```

Recursion Example - function



```ruby

# Print list recursively with count

def print_list_recursive_with_count(list)
  if (list.length == 0)
    return 0
  end

  puts " List Element is: " + list[0].to_s
  return print_list_recursive_with_count(list[1..list.size]) + 1
end

def main
  list = ["a", "b", "c", "d", "e", "f"]
```

Example: Factorials

Factorial:

`f(1)` = 1

`f(2)` = 1 x 2 = 2

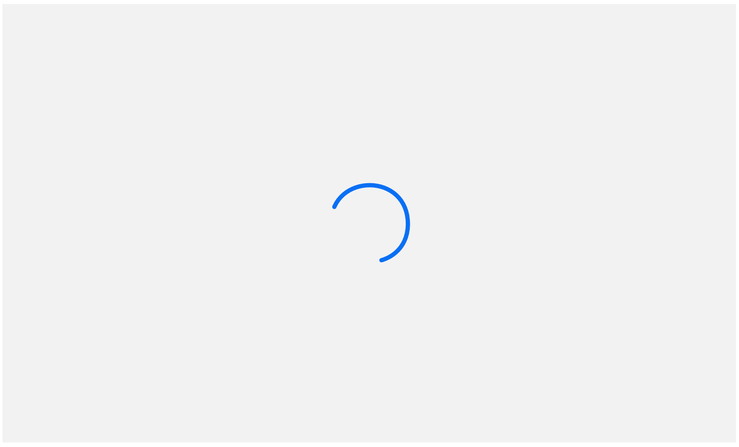
`f(3)` = 1 x 2 x 3 = 6

`f(4)` = 1 x 2 x 3 x 4 = 24

`f(5)` = 1 x 2 x 3 x 4 x 5 = 120

Example: Factorials II

A Lisp program:



Lets try this with Lisp. What if we try a really large number?

What happens? Why?

Run LISP

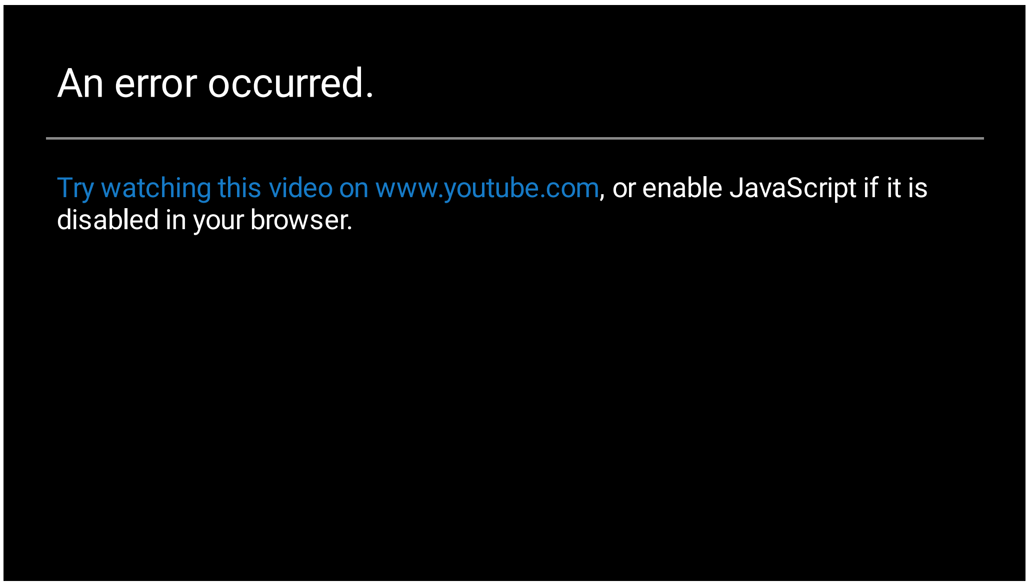
```lisp
(defun factorial (N) (if (= N 1) 1 (* N (factorial (- N 1)))))
(trace factorial)
(factorial 10000)
```

Impacts on the Stack

- Each time we call a recursive function we add a level to the program stack.
- Remember the stack stores the state of each function or procedure that is called.
- Tail recursion like print_list_recursive() above allows the possibility for a smart interpreter to never run out of stack space.
- Thus if you pass all the data to the next call, you can potentially save stack space

When to use recursion?

The Towers of Hanoi (5 disk version):



An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Induction

Recursion is related to a method of mathematical proof called ”induction”.

Induction: Recurrence Relations

Recurrence relations are equations that define the i th value in a sequence of numbers in terms of the preceding $i – 1$ values.

Eg: factorials
$0! = 1, n! = (n – 1)! * n$

Another function

Cumulative Sum:

$c(2) 1 + 2 = 3$

$c(3) 1 + 2 + 3 = 6$

$c(4) 1 + 2 + 3 + 4 = 10$

$c(5) 1 + 2 + 3 + 4 + 5 = 154$

Lets try two new Lisp functions

```
(defun sum (N) (if (= N 1)  1 (+ N (sum (- N 1)))))
```

```
(defun cumsum (N) (if (= N 1)  1 (/ (* N (+ N 1)) 2)))
```

Note: `(trace sum)` will trace the first function.

Run LISP

```
(defun sum (N) (if (= N 1)  1 (+ N (sum (- N 1)))))
(trace sum)
(sum 10)
```

Run LISP

```
(defun cumsum (N) (if (= N 1) 1 (/ (* N (+ N 1)) 2)))
(trace cumsum)
(cumsum 10)
```

Intuitive Proof

Eg: $n = 9$, then add up all the n sums to get 45, which is the same as $9 \times 10/2$

Eg: $n = 3$, then add up all the n sums to get 6 which is $3 \times 4/2$

Inductive Proof I

Induction involves proof through recursive relations using the following steps:

1.Show the proposition is true for an initial value (k) (base step)

2.Assume the proposition is true for $n \geq k$ show it is true for $n+1$ (induction step)

Inductive Proof II:

Eg: Prove that:
$1+2+3+\ldots+n=\frac{n(n+1)}{2}(1.0)$

Base Step: P(1) show the statement is true for the initial value:

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

Inductive Proof III

So keep in mind that:

$$1 + 2 + 3 + \ldots = n = \frac{n(n+1)}{2}(1.0)$$

And for $n = 1$ we have:

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

Induction Step I

So to generalise we add a ‘recursive’ step, moving to the next number in the ‘chain’.

We start with:

$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$ (1.0)

Then the induction step assumes that our equation holds for some n + 1.

$1 + 2 + 3 + ... + n + (n+1) = \frac{(n+1)(n+2)}{2}$ (2.0)

i.e add one to each n.

Induction Step II

So building on our induction step that our equation holds for some $n+1$.

$$1+2+3+\ldots+n+(n+1)=\frac{(n+1)(n+2)}{2}(2.0)$$

then the left side of the equation (2.0) becomes (note the recursive use of the RHS of 1.0 here):

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

and so we have verified conditions 1 and 2 of the Principle of Mathematical Induction.

HD Task: Recursive Maze Search: Hint

Try running the code with the following command:

Then insert the missing code:

The last lines of each `if` statement need to set `visited` to `true`, and then recursively call `search()`

C Compiling Task for Week 10

Let us see a demonstration of the C compiling task and talk a bit about what is in that file:

This week’s CREDIT task

Write a simple recursive factorial program

You can use the code above as a basis

It takes an argument on the command line.

Summary

- Programs may be written to be simple/elegant and/or efficient.
- For larger data sets efficiency becomes more important, the growth can be so large that machine time costs become significant.
- Recursion is an alternative to loops.
- Recursion may allow code to be both EFFICIENT and ELEGANT – if the problem is suited to it.

References

Freider, O, Frieder, G & Grossman, D. 2013 Computer Science Programming Basics in Ruby, O’Reilly Media Inc. See Section 7.2 Available online from the Swinburne Library

Stephens, R 2013 Essential Algorithms: A Practical Approach to Computer Algorithms, John Wiley and Sons (Chapt 6)

Test Revision

Code - make sure you up to at least week 6 with your tasks and you should be fine.

Quizzes and theory - based on the lessons and the lesson quizzes up to Week 6.

D and HD Tasks

- Check The Rubric (attached)
- Some the examples in the recorded video 'Custom Project Levels' under Modules->Assessments in Canvas.

MarkingRubricSummary.pdf

Note: we want discussion of design and evidence of design.