SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to Programming

Week 10,

Algorithms and Recursion

KNOWING

Algorithms and Recursion

- 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?)

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

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(N 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)

- **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(N log N) Algorithms**

  The time (number of lines of code executed) increases logarithmically in proportion to the size of the data set.

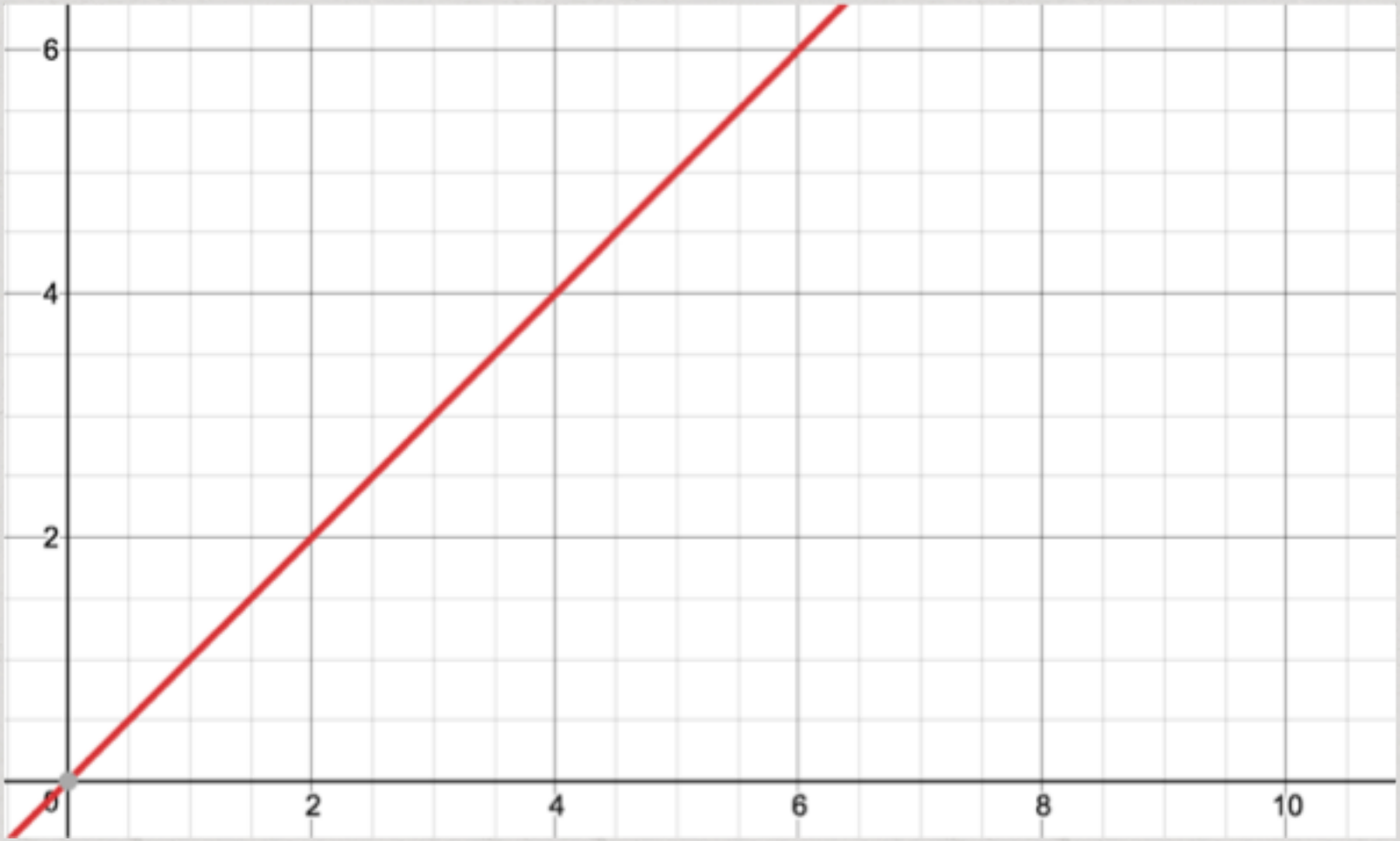
How would we rank these from most efficient to least efficient?

Lets try bubble sort with 10,000 elements.

$O(N^2)$ Algorithms

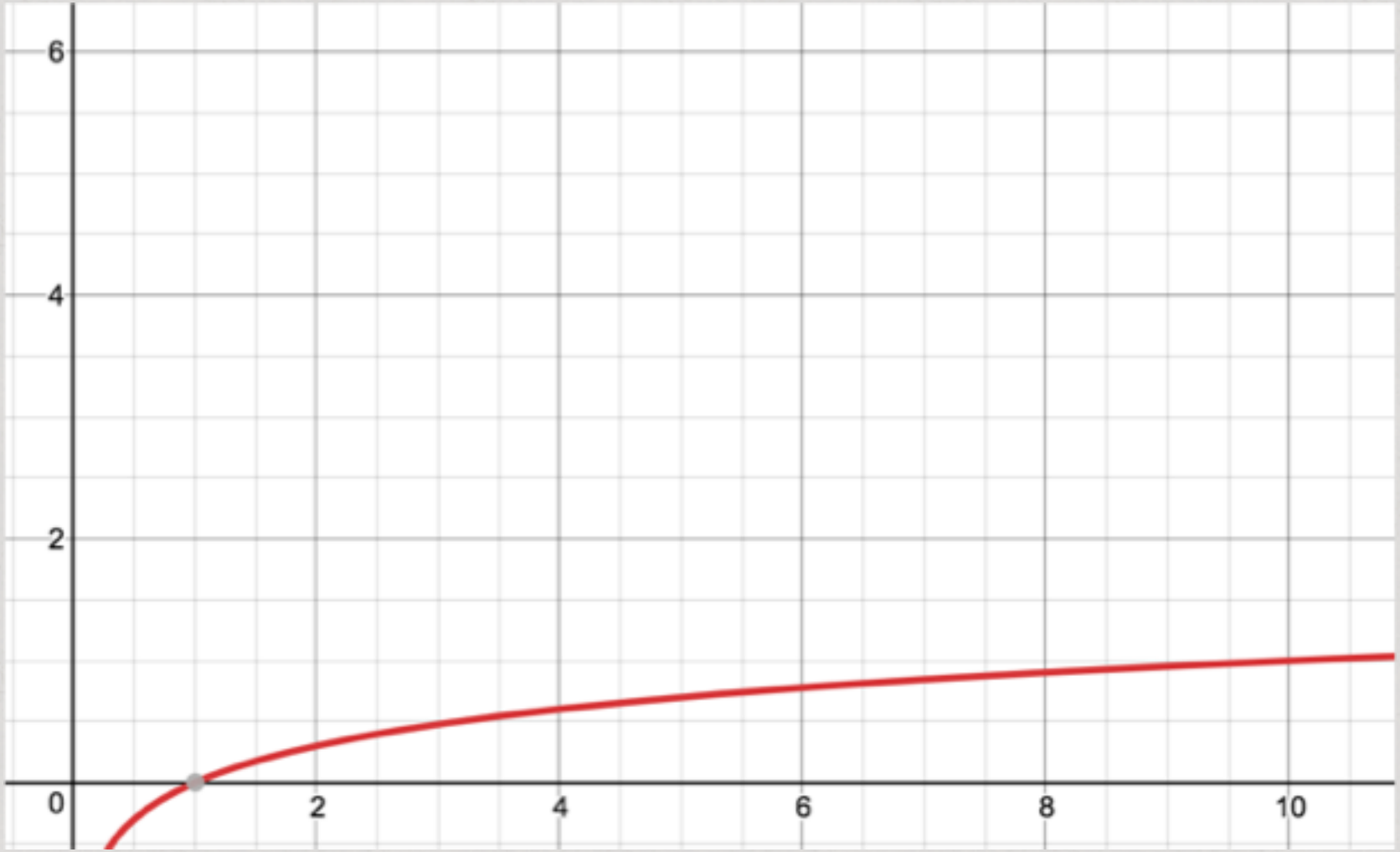
Produced using: https://www.desmos.com/calculator

O(N) Algorithms



Produced using: https://www.desmos.com/calculator

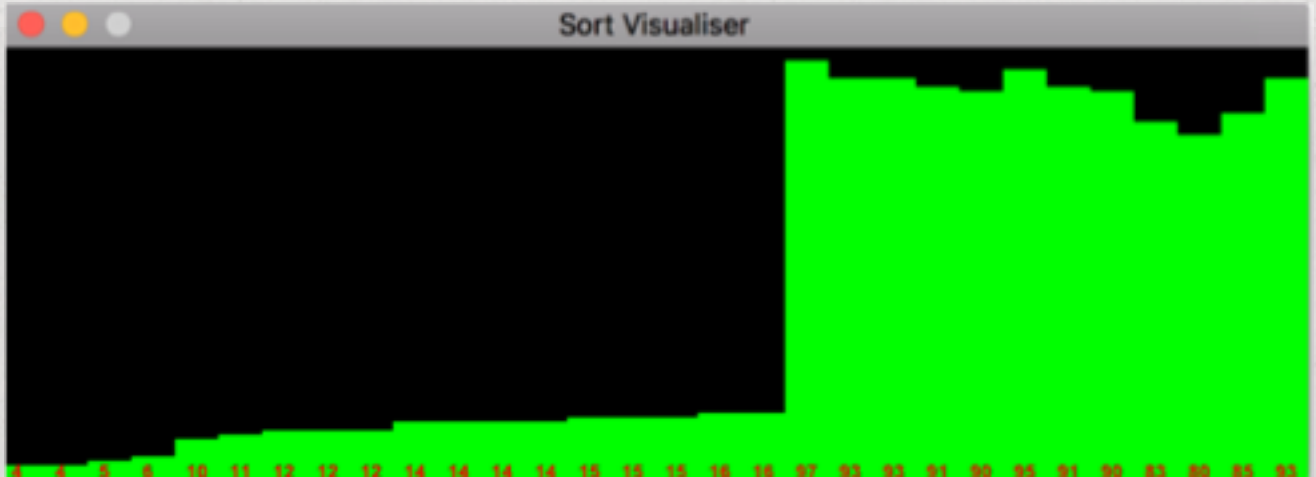
O(N log N) Algorithms



Produced using: https://www.desmos.com/calculator

This Week's Credit Task

- Use the provided code which already displays one sort algorithm running.
- Extend the code to allow the user to select another sort algorithm.
- One way is to left click or right click depending on the algorithm you want to view. Eg: Bubble Sort:



Recursion Description

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

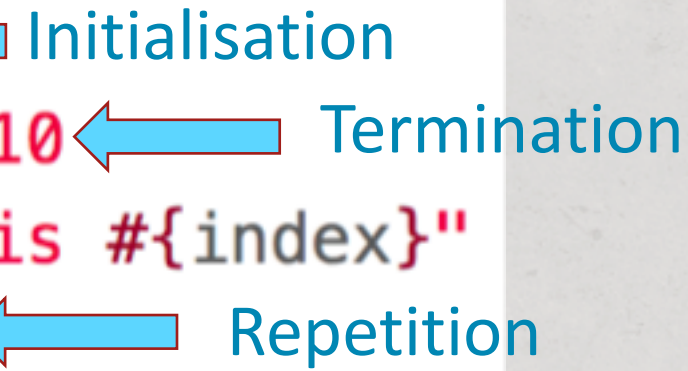
Recursion Structure

Recursion needs to have the elements of a loop:

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

Loops (revision)

```
def loop
   index = 0
   while index < 10
      puts "Index is #{index}"
      index += 1
   end
end
```



A loop like this has a control variable.

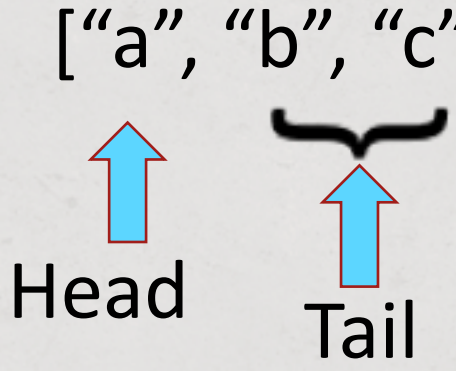
Loops

```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."
end

main
```

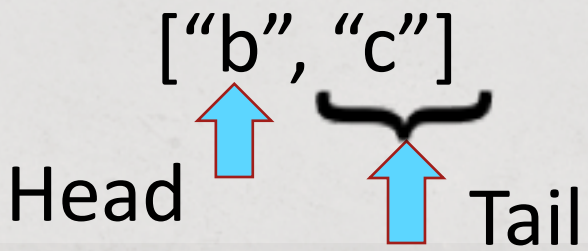
```
MacBook-Pro:Code mmitchell$ ruby array_example.rb
10
2
5
7
6
The array contained 5 elements.
MacBook-Pro:Code mmitchell$
```

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

```ruby
# Print list recursively

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"]
  print_list_recursive(list)
  puts "The list contained " + list.size.to_s + " elements."
end

main
```

Termination

Repetition

Initialisation

Recursion Example: Running

```ruby
# Print list recursively

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"]
  print_list_recursive(list)
  puts "The list contained " + list.size.to_s + " elements."
end

main
```

```
MacBook-Pro:Code mmitchell$ ruby list_example_recursive.rb
 List Element is: a
 List Element is: b
 List Element is: c
 List Element is: d
 List Element is: e
 List Element is: f
The list contained 6 elements.
MacBook-Pro:Code mmitchell$
```

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”.

Recursive Calculations

```ruby
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
```

Termination

Repetition

Glue

Initialisation

Calculated recursively

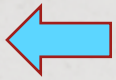
```
MacBook-Pro:Code mmitchell$ ruby list_example_recursive_with_count.rb
 List Element is: a
 List Element is: b
 List Element is: c
 List Element is: d
 List Element is: e
 List Element is: f
The list contained 6 elements.
MacBook-Pro:Code mmitchell$
```

Example: Factorials

```
(defun factorial (N)
        (if (= N 1) 1
                    (* N (factorial (- N 1)))))
```

Termination

Repetition

factorial(10)  Initialisation

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

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?

- Towers of Hanoi
  - See example code.

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

$0! = 1$, $n! = (n - 1)! * n$

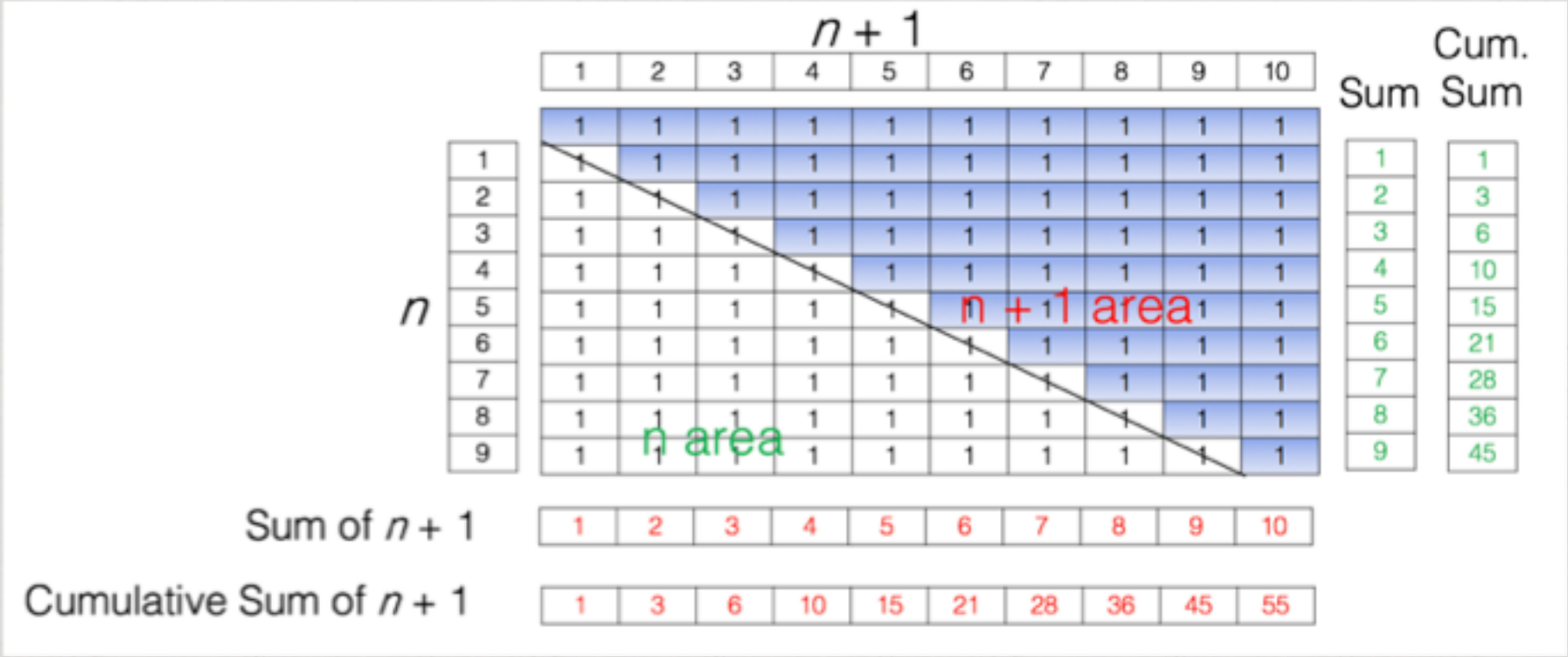
Lets try two new Lisp functions:

- (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.

Intuitive Proof



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

Eg: n = 3, then add up all the n sums to get 6 which is 3 x 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:

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

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:

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

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:

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

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

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

i.e add one to each *n*.

Induction Step II

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

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

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:

```
MacBook-Pro:Code mmitchell$ ruby gosu_maze_search.rb debug
```

- Then insert the missing code:

```
if (ARGV.length > 0) # debug
  puts "Searching. In cell x: " + cell_x.to_s + " y: " + cell_y.to_s
end

# INSERT MISSING CODE HERE!! You need to have 4 'if' tests to
# check each surrounding cell. Make use of the attributes for
# cells such as vacant, visited and on_path.
# Cells on the outer boundaries will always have a nil on the
# boundary side

# pick one of the possible paths that is not nil (if any):
if (north_path != nil)
```

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

C Compiling Task for Week 10

- Ok, let me demonstrate the C compiling task and talk a bit about what is in that file:

```c
#include <stdio.h>

void print_message(char *message)
{
    printf("%s", message);
}

int main()
{
  print_message("Hello World!\n");
  return 0;
}
```

This week's PASS task

- Write a simple recursive factorial program
- You can use the code above as a basis
- It takes an argument on the command line
- We will see a demo

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

- Structure charts – demo
- Loops – Demo
- Programming principles - revise

D and HD Tasks

- The Rubric
- Some more examples
- Note: we want discussion of design and evidence of design.