Lecture 3

Algorithms & Data Structures

Goldsmiths Computing

October 15, 2018

Outline

Outline

Lecture

- Pseudocode
    - loops:
        - for, forall, while, until
        - break, continue
        - general loop
    - function calls
    - pre- and post-conditions

Lecture

- Pseudocode
  - loops:
    - for, forall, while, until
    - break, continue
    - general loop
  - function calls
  - pre- and post-conditions
- Data structure building blocks
  - pairs

Lecture

- Pseudocode
  - loops:
    - for, forall, while, until
    - break, continue
    - general loop
  - function calls
  - pre- and post-conditions
- Data structure building blocks
  - pairs
  - vectors (one-dimensional arrays)

Lab

- pseudocode for real algorithms
  1. reading, trying
  2. inferring, proving
- working with the lab environment
  1. makefiles, test libraries
  2. online submission system

VLE activities

Programming language choice

- 126 have made their choice; thank you!
  - contact me directly if you need to change
  - contact me directly (with an apology!) if you haven't made your choice

VLE activities (cont'd)

Pairs and vectors quiz

Statistics so far:

- 177 attempts: average mark 5.68
- 86 students: average mark 6.60
  - 17 under 4.00, 45 over 6.99, 24 at 10.00

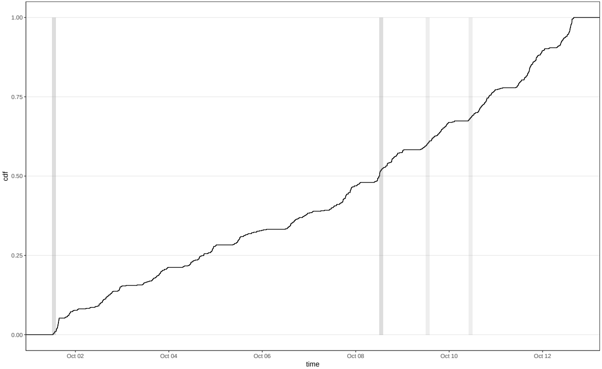
Quiz closes at 16:00 on Friday 19th October

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Pseudocode quiz

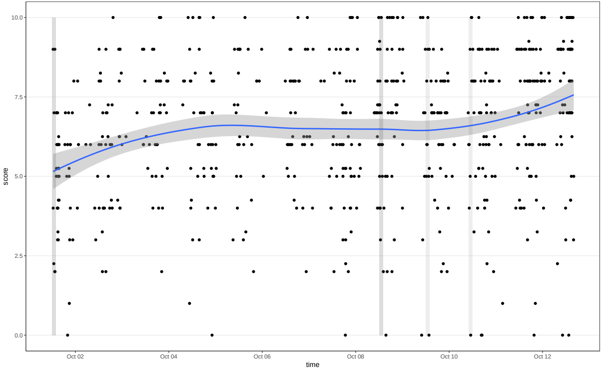
- 650 attempts: average mark 6.53
- 137 students: average mark 8.50
  - 2 under 4.00, 118 above 6.99, 49 at 10



VLE activities (cont'd)

Pseudocode quiz

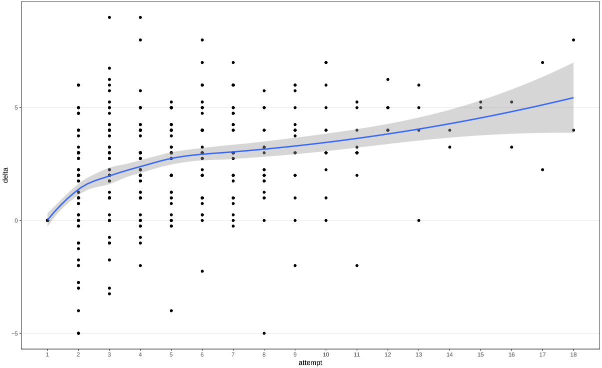
- 650 attempts: average mark 6.53
- 137 students: average mark 8.50
  - 2 under 4.00, 118 above 6.99, 49 at 10



VLE activities (cont'd)

Pseudocode quiz

- 650 attempts: average mark 6.53
- 137 students: average mark 8.50
  - 2 under 4.00, 118 above 6.99, 49 at 10



Outline

Motivation

- constant-time access of (fixed) arrays
- extensibility of linked lists
- Java: `ArrayList`, C++ `std::vector`

We can solve any problem [in Computer Science] by introducing an extra level of indirection. – David J. Wheeler

Definition

A dynamic array is a finite sequential collection of data.
(removal of “fixed-size” from the definition of vector)

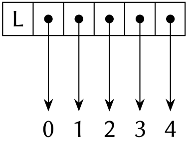
Operations

length	return the current size of the dynamic array
select[k]	return the k^{th} element of the dynamic array
store![o,k]	set the k^{th} element of the array to o

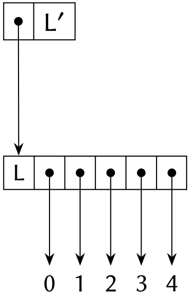
Operations

length	return the current size of the dynamic array
select[k]	return the k^{th} element of the dynamic array
store![o,k]	set the k^{th} element of the array to o
push![o]	increase the length of the dynamic array by 1, and set the endmost element to o
pop!	return the endmost element, decreasing the size of the dynamic array by 1

Implementation

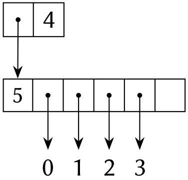


Implementation



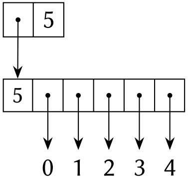
Push!

Require: A :: dynamic array
  function PUSH!(A,k)
    if LENGTH(LEFT(A)) = RIGHT(A) **then**
      EXTEND(A)
    end if
    A[RIGHT(A)] ← k
    RIGHT(A) ← RIGHT(A) + 1
  end function



Push!

Require: A :: dynamic array
function PUSH!(A,k)
  if LENGTH(LEFT(A)) = RIGHT(A) **then**
    EXTEND(A)
  end if
  A[RIGHT(A)] ← k
  RIGHT(A) ← RIGHT(A) + 1
end function



Extend

Require: A :: dynamic array

function EXTEND(A)

  newL ← NEWLENGTH(RIGHT(A))

  new ← **new** Vector(newL)

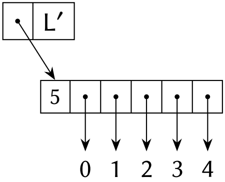
  for $0 \leq i <$ LENGTH(A) **do**

    new[i] ← LEFT(A)[i]

  end for

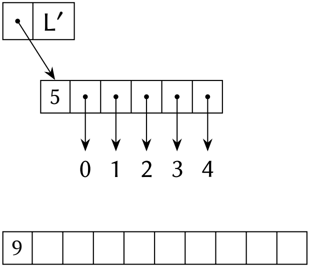
  LEFT(A) ← new

end function



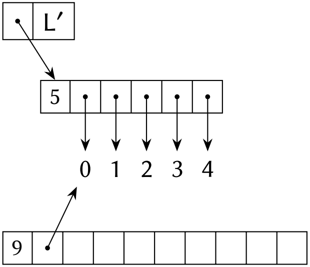
Extend

Require: A :: dynamic array
function EXTEND(A)
  newL ← NEWLENGTH(RIGHT(A))
  new ← **new** Vector(newL)
  for $0 \leq i <$ LENGTH(A) **do**
    new[i] ← LEFT(A)[i]
  end for
  LEFT(A) ← new
end function



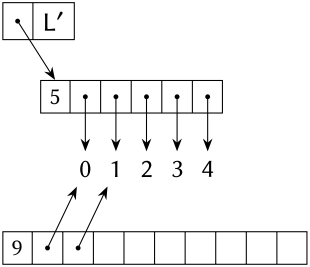
Extend

Require: A :: dynamic array
function EXTEND(A)
  newL ← NEWLENGTH(RIGHT(A))
  new ← **new** Vector(newL)
  for 0 ≤ i < LENGTH(A) **do**
    new[i] ← LEFT(A)[i]
  end for
  LEFT(A) ← new
end function



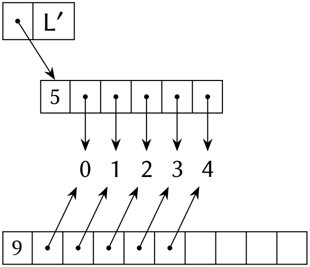
Extend

Require: A :: dynamic array
function EXTEND(A)
  newL ← NEWLENGTH(RIGHT(A))
  new ← **new** Vector(newL)
  for $0 \le i <$ LENGTH(A) **do**
    new[i] ← LEFT(A)[i]
  end for
  LEFT(A) ← new
end function



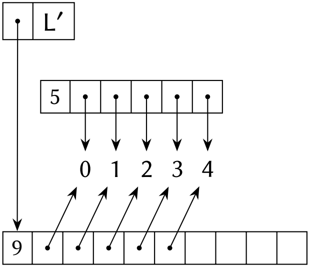
Extend

Require: A :: dynamic array
  function EXTEND(A)
    newL ← NEWLENGTH(RIGHT(A))
    new ← **new** Vector(newL)
    for 0 ≤ i < LENGTH(A) **do**
      new[i] ← LEFT(A)[i]
    end for
    LEFT(A) ← new
  end function



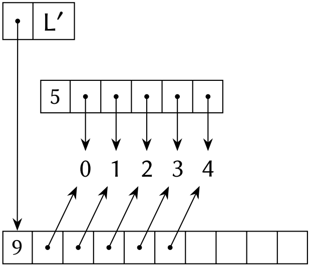
Extend

Require: A :: dynamic array
function EXTEND(A)
  newL ← NEWLENGTH(RIGHT(A))
  new ← **new** Vector(newL)
  for $0 \le i <$ LENGTH(A) **do**
    new[i] ← LEFT(A)[i]
  end for
  LEFT(A) ← new
end function



Extend

Require: A :: dynamic array
function EXTEND(A)
  newL ← NEWLENGTH(RIGHT(A))
  new ← **new** Vector(newL)
  for $0 \le i <$ LENGTH(A) **do**
    new[i] ← LEFT(A)[i]
  end for
  LEFT(A) ← new
end function



What should NEWLENGTH(n) be?

- return n + C (*e.g.* n + 10)?
- return C × n (*e.g.* 2 × n)?
- return n^C (*e.g.* n^2)?

Complexity analysis

length, select, store!

- each is a pointer read (to get the storage array) and a $\Theta(1)$ array operation

$$\Longrightarrow \Theta(1)$$

Complexity analysis

length, select, store!

- each is a pointer read (to get the storage array) and a $\Theta(1)$ array operation

$$\implies \Theta(1)$$

push!

Usual case:

- increment length
- store value in storage array

$$\implies \Theta(1)$$

Complexity analysis

length, select, store!

- each is a pointer read (to get the storage array) and a $\Theta(1)$ array operation

$$\implies \Theta(1)$$

push!

Usual case:

- increment length
- store value in storage array

$$\implies \Theta(1)$$

When extending storage array:

- as above plus...
- ... copy existing contents to new array

$$\implies \Theta(N)$$

Work

1. Reading
   - CLRS, section 17.4
2. Implement a dynamic array using a pair and an array (as shown in these slides). What will you do with the storage array when implementing POP!?

Outline

Motivation

- compare functions in terms of their growth
  - including functions describing algorithm steps
- ignore irrelevant details:
  - lower-order terms
  - constant factors
- basis for informal engineering designs
  - how big will my data grow?
  - will my existing solution still work adequately at scale?

Big-O

$$f(x) = O(g(x)) \text{ or } f(x) \in O(g(x))$$

Informally:

- $f(x)$ grows no faster than $g(x)$

Heuristically:

- as $x \longrightarrow \infty$, $f(x)$ is bounded above by some constant times $g(x)$

Formally:

$$\exists(C \in \mathbb{R}^{+}) : \exists(x_0 \in \mathbb{R}) : \forall(x > x_0) : f(x) < Cg(x)$$

Big-O

Examples:

- $x^2 - 3x + 6 = O(x^2)$ (*e.g.* choose $x_0 = 1$, $C = 5$)
- $x^2 - 3x + 6 = O(x^4 + 3)$ (*e.g.* choose $x_0 = 1$, $C = 2$)
- $x + 2x\log(x) + 3(\log(x))^2 = O(x\log(x))$ (*e.g.* choose $x_0 = 20$, $C = 3$)

Big-Ω

$$f(x) = \Omega(g(x)) \text{ or } f(x) \in \Omega(g(x))$$

Informally:

- $f(x)$ grows no slower than $g(x)$

Heuristically:

- as $x \longrightarrow \infty$, $f(x)$ is bounded below by some constant times $g(x)$

Formally:

$$\exists(C \in \mathbb{R}^+) : \exists(x_0 \in \mathbb{R}) : \forall(x > x_0) : f(x) > Cg(x)$$

Big-Ω

Examples:

- $x^2 - 3x + 6 = \Omega(x^2)$ (*e.g.* choose $x_0 = 3$, $C = \frac{1}{2}$)
- $x^2 - 3x + 6 = \Omega(x)$ (*e.g.* choose $x_0 = 3$, $C = 1$)
- $x + 2x\log(x) + 3(\log(x))^2 = \Omega(\log(x)^2)$ (*e.g.* choose $x_0 = 1$, $C = 1$)

Big-Θ

$$f(x) = \Theta(g(x)) \text{ or } f(x) \in \Theta(g(x))$$

Informally:

- $f(x)$ grows like $g(x)$

Heuristically:

- as $x \longrightarrow \infty$, $f(x)$ is bounded above and below by constants times $g(x)$

Formally:

$$\exists (C_1, C_2 \in \mathbb{R}^+) \ : \ \exists (x_0 \in \mathbb{R}) \ : \ \forall (x > x_0) \ : \ C_1 g(x) < f(x) < C_2 g(x)$$

Big-Θ

Examples:

- $x^2 - 3x + 6 = \Theta(x^2)$ (*e.g.* choose $x_0 = 3$, $C_1 = \frac{1}{2}$, $C_2 = 5$)

Little-o

$$f(x) = o(g(x)) \text{ or } f(x) \in o(g(x))$$

Informally:

- $f(x)$ grows much slower than $g(x)$

Heuristically:

- as $x \longrightarrow \infty$, $\frac{f(x)}{g(x)} \longrightarrow 0$

Formally:

$$\forall(\varepsilon \in \mathbb{R}^+) : \exists(x_0 \in \mathbb{R}) : \forall(x > x_0) : f(x) < \varepsilon g(x)$$

Common complexity classes

$\Theta(1)$	slowest growth
$\Theta(\log(n))$	
$\Theta((\log(n))^{1+c})$	
$\Theta(n^c)$	
$\Theta(n)$	
$\Theta(n\log(n))$	
$\Theta(n^{1+c})$	
$\Theta((1+c)^n)$	
$\Theta(n!)$	
$\Theta(n^n)$	fastest growth

for $0 < c < 1$

Work

1. Reading
   - CLRS, chapter 3
   - DPV, section 0.3
2. Problems from CLRS:
   - 1-1 Comparison of running times
   - 3-2 Relative asymptotic growths
   - 3-3 Ordering by asymptotic growth rates
3. Exercises from DPV: 0.1, 0.2
4. do the big-O quiz on the VLE

Outline

Motivation

- model for real computers
- simple enough to reason about

Definition

A random-access machine is a computer with:

- an unbounded amount of memory
  - addressable by integers
  - each memory access takes a constant time step

Definition

A random-access machine is a computer with:

- an unbounded amount of memory
  - addressable by integers
  - each memory access takes a constant time step
- a program made up of simple instructions
  - executed one-at-a-time
  - each simple instruction takes a constant time step

Definition

A random-access machine is a computer with:

- an unbounded amount of memory
  - addressable by integers
  - each memory access takes a constant time step
- a program made up of simple instructions
  - executed one-at-a-time
  - each simple instruction takes a constant time step
- program combinations using functions, loops, conditionals
  - the combination itself takes a constant time step

Definition

A random-access machine is a computer with:

- an unbounded amount of memory
  - addressable by integers
  - each memory access takes a constant time step
- a program made up of simple instructions
  - executed one-at-a-time
  - each simple instruction takes a constant time step
- program combinations using functions, loops, conditionals
  - the combination itself takes a constant time step
  - the result of combination takes longer

Time and Space

Running time

The number of constant time steps taken

- memory access
- simple instructions executed
- combinations executed

Time and Space

Running time

The number of constant time steps taken

- memory access
- simple instructions executed
- combinations executed

Space used

The number of memory locations used

- in addition to the space used by the input: “additional space used”

Example

```
function EXERCISE1(v)
    a ← 0; b ← 0
    for 0 ≤ i < LENGTH(v) do
        if v[i] > b then
            if v[i] > a then
                b ← a
                a ← v[i]
            else
                b ← v[i]
            end if
        end if
    end for
    return b
end function
```

Example

```
function EXERCISE1(v)
    a ← 0; b ← 0                                    ▷ 2
    for 0 ≤ i < LENGTH(v) do
        if v[i] > b then
            if v[i] > a then
                b ← a
                a ← v[i]
            else
                b ← v[i]
            end if
        end if
    end for
    return b                                        ▷ 1
end function
```

Example

```
function EXERCISE1(v)
    a ← 0; b ← 0                                  ▷ 2
    for 0 ≤ i < LENGTH(v) do                      ▷ n = LENGTH(v)
        if v[i] > b then
            if v[i] > a then
                b ← a
                a ← v[i]
            else
                b ← v[i]
            end if
        end if
    end for
    return b                                      ▷ 1
end function
```

Example

```
function Exercise1(v)
    a ← 0; b ← 0                                  ▷ 2
    for 0 ≤ i < length(v) do                      ▷ n = length(v)
        if v[i] > b then                          ▷ n
            if v[i] > a then
                b ← a
                a ← v[i]
            else
                b ← v[i]
            end if
        end if
    end for
    return b                                      ▷ 1
end function
```

Example

```
function EXERCISE1(v)
    a ← 0; b ← 0                                ▷ 2
    for 0 ≤ i < LENGTH(v) do                    ▷ n = LENGTH(v)
        if v[i] > b then                        ▷ n
            if v[i] > a then                    ▷ up to n
                b ← a
                a ← v[i]
            else
                b ← v[i]
            end if
        end if
    end for
    return b                                    ▷ 1
end function
```

Example

```
function Exercise1(v)
    a ← 0; b ← 0                              ▷ 2
    for 0 ≤ i < length(v) do                  ▷ n = length(v)
        if v[i] > b then                      ▷ n
            if v[i] > a then                  ▷ up to n
                b ← a                         ▷ up to n
                a ← v[i]                      ▷ up to n
            else
                b ← v[i]
            end if
        end if
    end for
    return b                                  ▷ 1
end function
```

The point of all of this

In software design and implementation, we often want to:

- minimize the time the program takes to run
- minimize the resources (*e.g.* memory, disk space) the program consumes

The Random-Access Machine model

- simple enough to compute answers, at least approximately
- realistic enough to be a guide to real computers

Work

1. Reading
   - CLRS, section 2.2
2. Exercises from CLRS: 2.2-1, 2.2-4

Outline

Motivation

- understand consequence of data structure design decisions
- simple example of random-access model and big-O notation

Implementation details

Here we are thinking as the data structure implementor, not the data structure user

- look “behind the curtain” of the data structure
- implementing operations, so we can’t use them!

Implementation details

Here we are thinking as the data structure implementor, not the data structure user

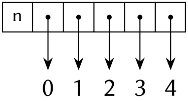
- look "behind the curtain" of the data structure
- implementing operations, so we can't use them!

Primitives:

- $\textsc{mref}(n) \Longrightarrow \mathbb{Z}$
- $\textsc{alloc}(n) \Longrightarrow \mathbb{Z}$ (and allocates memory)
- arithmetic

Constructor

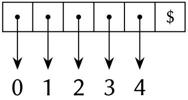
length-data



function VNEW(n)
    v ← ALLOC(n+1)
    MREF(v) ← n
    return v
end function

$\Longrightarrow \Theta(1)$

sentinel

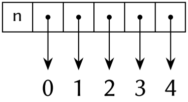


function VNEW(n)
    v ← ALLOC(n+1)
    MREF(v+n) ← $
    return v
end function

$\Longrightarrow \Theta(1)$

Constructor

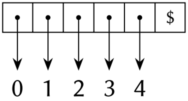
length-data



function VNEW(n)
    v ← ALLOC(n+1) ▷ 2
    MREF(v) ← n ▷ 1
    return v ▷ 1
end function

$\Longrightarrow \Theta(1)$

sentinel

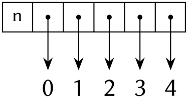


function VNEW(n)
    v ← ALLOC(n+1)
    MREF(v+n) ← $
    return v
end function

$\Longrightarrow \Theta(1)$

Constructor

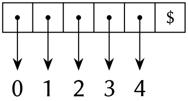
length-data



function VNEW(n)
    v ← ALLOC(n+1) ▷ 2
    MREF(v) ← n ▷ 1
    return v ▷ 1
end function

$\Longrightarrow \Theta(1)$

sentinel

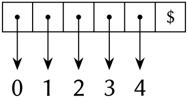


function VNEW(n)
    v ← ALLOC(n+1) ▷ 2
    MREF(v+n) ← $ ▷ 2
    return v ▷ 1
end function

$\Longrightarrow \Theta(1)$

Dereference

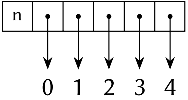
sentinel



function [](v,i)
  return MREF(v+i)
end function

$\implies \Theta(1)$

length-data

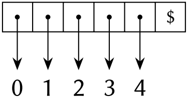


function [](v,i)
  return MREF(v+i+1)
end function

$\implies \Theta(1)$

Dereference

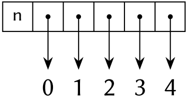
sentinel



function [](v,i)
    return MREF(v+i) $\triangleright$ 2
end function

$\Longrightarrow \Theta(1)$

length-data

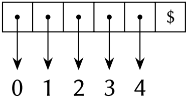


function [](v,i)
    return MREF(v+i+1)
end function

$\Longrightarrow \Theta(1)$

Dereference

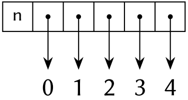
sentinel



function [](v,i)
    return MREF(v+i) ▷ 2
end function

$\Longrightarrow \Theta(1)$

length-data

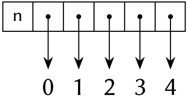


function [](v,i)
    return MREF(v+i+1) ▷ 3
end function

$\Longrightarrow \Theta(1)$

Length

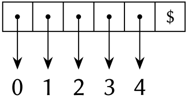
length-data



function LENGTH(v)
    return MREF(v)
end function

$\Longrightarrow \Theta(1)$

sentinel

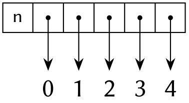


function LENGTH(v)
    $l \leftarrow 0$
    while MREF(v+l) $\neq$ \$ **do**
        $l \leftarrow l + 1$
    end while
    return l
end function

$\Longrightarrow \Theta(n)$

Length

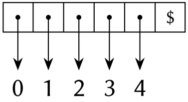
length-data



function LENGTH(v)
    return MREF(v)   ▷ 1
end function

$\Longrightarrow \Theta(1)$

sentinel

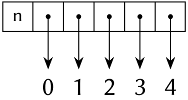


function LENGTH(v)
    l ⟵ 0
    while MREF(v+l) ≠ $ **do**
        l ⟵ l + 1
    end while
    return l
end function

$\Longrightarrow \Theta(n)$

Length

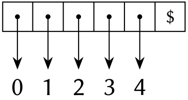
length-data



function LENGTH(v)
    return MREF(v)   ▷ 1
end function

$\Longrightarrow \Theta(1)$

sentinel

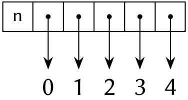


function LENGTH(v)
    l ⟵ 0   ▷ 1
    while MREF(v+l) ≠ $ **do**
        l ⟵ l + 1
    end while
    return l
end function

$\Longrightarrow \Theta(n)$

Length

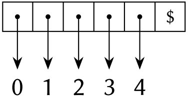
length-data



function LENGTH(v)
    return MREF(v) $\triangleright$ 1
end function

$$\Longrightarrow \Theta(1)$$

sentinel



function LENGTH(v)
    l $\longleftarrow$ 0 $\triangleright$ 1
    while MREF(v+l) $\neq$ \$ **do** $\triangleright$ 3n+3
        l $\longleftarrow$ l + 1 $\triangleright$ 2n+2
    end while
    return l
end function

$$\Longrightarrow \Theta(n)$$

Activities over the next week

1. Finish pairs and vectors quiz (16:00 Friday 20th October)
2. Start big-O quiz
3. Labsheet 03: dynamic arrays
4. Reading and problems
5. Answer and ask questions on the forum