Recursive Algorithms
and
List Data Structures

Introduction to Algorithms

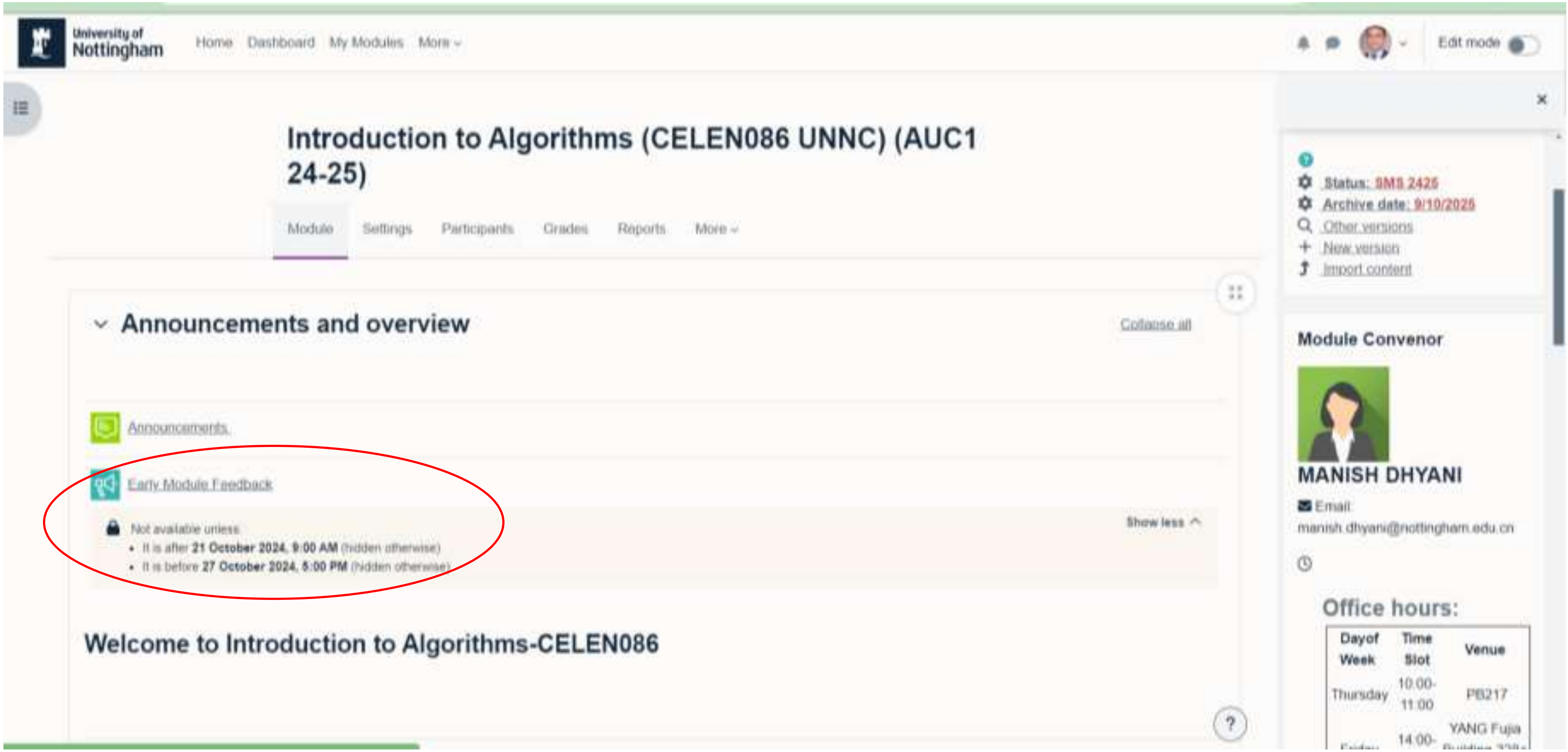
Module Code: CELEN086

Lecture 4

(21/10/24)

Lecturer & Convenor: Manish Dhyani
Email: Manish.Dhyani@nottingham.edu.cn

Early Module Feedback



Available on CELEN086 Moodle page from Monday 21 October 9:00 AM to Sunday 27 October 5:00 PM.

Fibonacci sequence (Recursive Approach)

Fibonacci numbers, commonly denoted by F_n, form a well-known sequence. The following is a Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

- **Fibonacci Numbers: The** Fibonacci sequence is defined recursively as:

$$F_n = F(n-1) + F(n-2), \quad n \geq 3$$

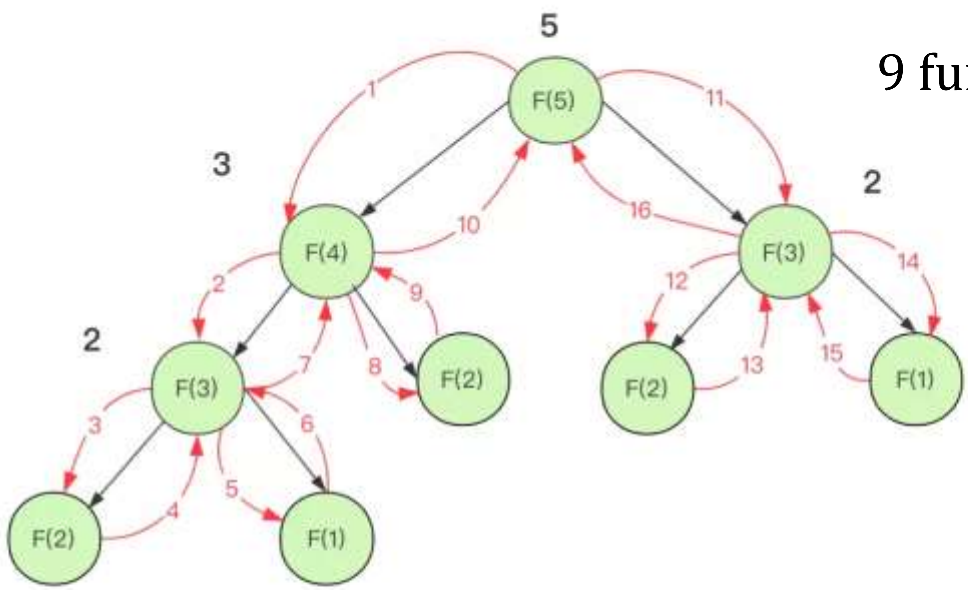
```
Algorithm: F(n)
Requires: a positive integer n
Returns: the n-th number in the Fibonacci sequence
---------------------------------------------------
if  n==0 then
return 0 //base case
elseif  n== 1 then
        return 1 // base case
else
        return F(n-1)+F(n-2) // recursive step
endif
```

Two base cases (combined as one)

Two recursive calls onto function F itself at each recursive step

Recursion: pros and cons

To compute F(5), how many function calls are there?



9 function calls of F(n)

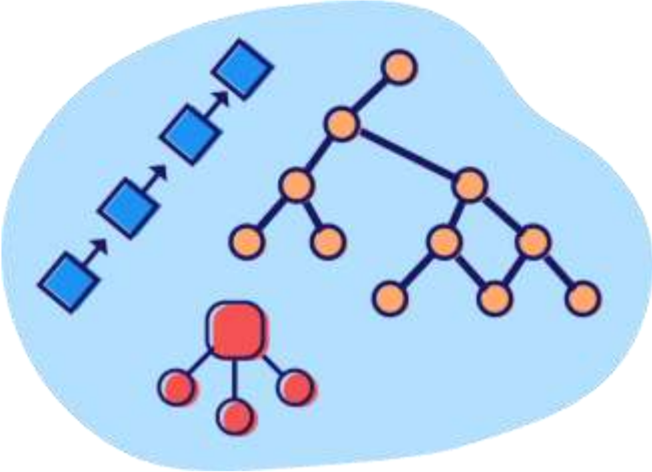
- Pros: recursive algorithm is well-structured with short length of codes.
- Cons: time complexity may grow exponentially if there are considerable numbers of recursive calls.

What is a Data Structure?

- Data Structure is a systematic way to store and organize data so that it can be used efficiently. It is useful in every aspect of computer science: operating system, compiler design, artificial intelligence, etc.

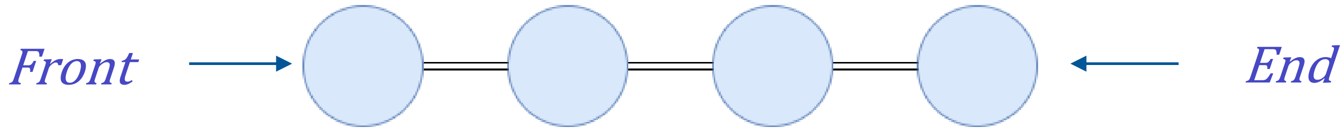
- Data structure is associated with algorithms that allows programmers to handle data in efficient ways and enhance the performance of the program.

- Common types of data structure
  - Linear data structure: list, array, stack, queue
  - Non-linear data structure: tree, graph



List

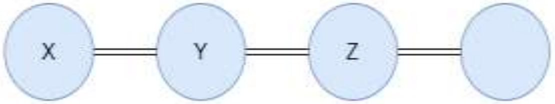
A list is a linear data structure.



How to access elements (data) stored in a list?

In this module, we restrict basic operations in list as follows:

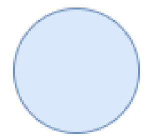
- Add an element to a list from the front
- Delete an element from the front
- Retrieve an element from the front



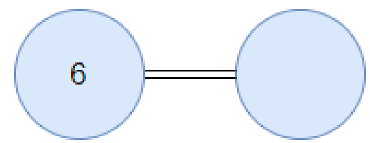
Examples of list

We can use one dimensional arrays* for representing lists in this module.

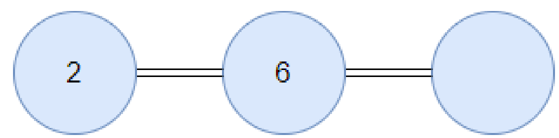
The empty list [ ]



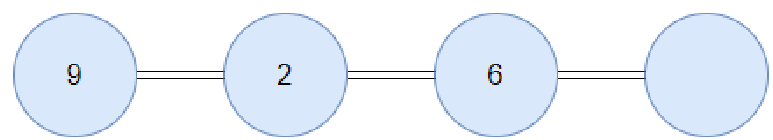
The list [6]



The list [2,6]



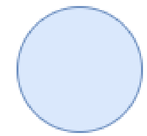
The list [9,2,6]



**Note: Arrays and lists are stored differently in computer memory.*

Creating a list

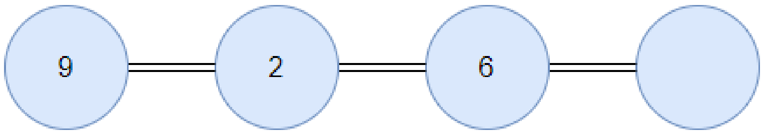
There are two building blocks to create a list:

- Nil — to create/represent an empty list 
- cons(x, list) — to construct a longer list by adding an element x to the front of a given list

Every list is either empty or built by adding a number to a shorter list.

Example

How to build the list [9,2,6]?



Start from an empty list Nil, then add elements one by one in order using cons(x, list).

Nil = [ ]

cons(6,Nil) = [6]

cons(2,[6]) = [2,6]

cons(9,[2,6]) = [9,2,6]

Note:
6 and [6] are not the same. They are specified by different data structures.

cons(x, list)
The first input argument is a number;
the second input argument is a list.

The above statements can be written in a composite form

cons(9, cons(2, cons(6, Nil))) = [9,2,6]

Decomposing a list

Before starting algorithm design on lists, we need three more functions that can help decomposing the lists.

- isEmpty(list) — to return Boolean value True if the list is empty (Nil or [ ]); False if the list is non-empty
- value(list) — to return the first element stored in the list
- tail(list) — to return a list without the first element, i.e., value(list)

Example: isEmpty(tail(list)) can check if a non-empty list has single element

Example

isEmpty(Nil) = True

isEmpty([1,2,3]) = False

value([1,2,3]) = 1

tail([1,2,3]) = [2,3]

value(tail([1,2,3])) = 2

tail(tail([1,2,3])) = [3]

Not valid!

value (value([1,2,3]))

(not a list)

tail (value([1,2,3]))

value (tail (tail([1,2,3]))) = 3

tail (tail (tail([1,2,3]))) = [ ]

By making composite calls of value() and tail(), we are able to move alone with the list and retrieve any element from it.

List Length

Trace length([4,2,7])

list=[4,2,7]

Line 1: False

Line 4: return 1+length([2,7]) =1+2=3

list=[2,7]

Line 1: False

Line 4: return 1+length([7])=1+1=2

list=[7]

Line 1: False

Line 4: return 1+length([ ]) =1+0=1

list=[ ]

Line 1: True

Backtracking

Line 2: return 0 (base case)

Algorithm: length(list)
Requires: a list
Returns: total number of elements in the list

```
1.  if isEmpty(list)
2.          return 0
3.  else
4.          return 1+length(tail(list))
5.  endif
```

Approaching base case

> Note:
> The value() and tail() functions only work on non-empty list.
>
> Therefore, we may need to check if the list is empty or not before calling these two functions, using isEmpty().

List Sum

Trace sum([4,2,7])

list=[4,2,7]

Line 1: isEmpty([2,7])? False

Line 4: return 4+sum([2,7]) =4+9=13

list=[2,7]

Line 1: isEmpty([7])? False

Line 4: return 2+sum([7]) =2+7=9

list=[7]

Line 1: isEmpty([ ])? True

Line 4: return 7

(base case)

Backtracking

Algorithm: sum(list)
Requires: a non-empty list
Returns: sum of all elements in the list

```
1.  if isEmpty(tail(list))
2.          return value(list)
3.  else
4.          return value(list)+sum(tail(list))
5.  endif
```

Note:
Normally in a recursive algorithm on lists, the simplest situation will be that the list is empty, or it has single element.

We can use isEmpty(list) or isEmpty(tail(list)) respectively, to specify the base case.

Minimum of a list

```
Algorithm: minimum(list) [main algorithm]
Requires: a non-empty list
Returns: smallest element of the list

1.  if isEmpty(tail(list))
2.          return value(list)
3.  else
4.          return min(value(list),minimum(tail(list)))
5.  endif
```

```
Algorithm: min(x,y) [sub algorithm]
Requires: two numbers x and y
Returns: the minimum of x and y

1.  if x<y
2.          return x
3.  else
4.          return y
5.  end
```

minimum([4,2,7])

list = [4,2,7]

Line 4: return min(4, minimum([2,7])) =min(4,2) = 2

list = [2,7]

Computed by sub-algorithm

Line 4: return min(2, minimum([7])) =min(2,7) = 2

list = [7]

Backtracking

Line 2: return 7

Minimum of a list (alternative ver.)

Algorithm: minList(L) [main function]
Requires: a non-empty list
Returns: smallest element of the list

```
1. return minListHelper(tail(L),value(L))
```

Algorithm: minListHelper(list, ref_Val) [helper function]
Requires: a non-empty list and a number
Returns: smallest element of the list

Stores the minimum value we have found so far.

```
1.  if isEmpty(list)
2.          return ref_Val
3.  elseif value(list)<ref_Val
4.          return minListHelper(tail(list), value(list))
5.  else  // ref_Val <= value(list)
6.          return minListHelper(tail(list), ref_Val)
7.  endif
```

minList([4,2,7]) [main]

L = [4,2,7]

return minListHelper([2,7],4)

---

list = [2,7], ref_Val = 4 [helper]

Line 1: isEmpty? False. Line 3: 2<4? True. Line 4: return minListHelper([7],2)

list = [7], ref_Val = 2

Line 1: isEmpty? False. Line 3: 7<2? False. Line 6: return minListHelper([ ],2)

list = [ ], ref_Val = 2

Backtracking? No. This is a tail recursion. No other actions need to be performed. (Seminar 3)

Line 1: isEmpty? True. Line 2: return 2

Review Activity

https://forms.office.com/r/30wPYX9PYn

Review Quiz -CELEN086(ITA)