中国科学院大学
University of Chinese Academy of Sciences

CS101

Overview

Computational Thinking
Without: ABC
Within: Acu-Exams

zxu@ict.ac.cn
zhangjialin@ict.ac.cn

Outline

- CS & CT study computational processes
- Three features without: ABC
- Eight understandings within: Acu-Exams
- CS & CT are a synergy

- Reminder
  - Set up your computer for programming by next week
    - Use programs this week
    - Start to do programming next week

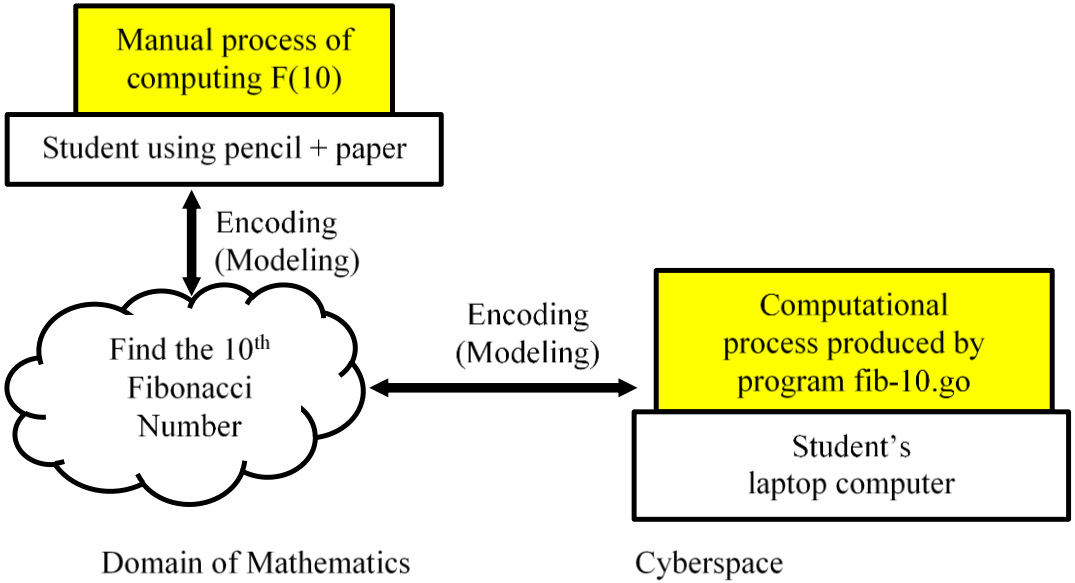
These slides acknowledge sources for additional data not cited in the textbook

1. What are CS and CT?

- Computer science (CS) is the study of computational processes
  - for problem solving and creative expression
  - that are correct, smart, and practical
- CS combines
  - logic, algorithmic, systems thinking, and
  - network thinking
- Computational thinking (CT) is the way of thinking underlying the computer science discipline
  - ABC features without (looking from the outside)
  - Acu-Exams understandings within (looking inside)

1.1 Computational process

- A step-by-step process of information transformation
  - A sequence of symbol manipulation steps
  - Can be done manually or automatically
- Compute Fibonacci number F(10)
  - Given definition
    F(0)=0, F(1)=1,
    F(n)=F(n-1)+F(n-2)
    when n>1,
    Find F(10).
- Manual
  - Tedious
  - Impractical for large n
- Computer
  - Automatic after encoding into cyberspace
  - Practical even for n = 1 billion



1.2 Problem solving by PEPS

- Compute Fibonacci number F(10)
  - **P**roblem:
    - Given definition, find F(10).
  - **E**ncoding:
    - A recursive algorithm
  - Computational **P**rocess
    - Embodied in fib-10.go
  - Computer **S**ystem

```
Output F(10)                          // n and F(n) are natural numbers
where F(n) is defined as
      if (n=0 or n=1) then F(n)=n else F(n)=F(n-1)+F(n-2)
```

(a) An algorithm to find F(10) directly from the mathematical definition

```
package main                          // Program setup
import "fmt"
func main() {
    fmt.Println("F(10)=", fibonacci(10))   // Output F(10)
}
func fibonacci(n int) int {           // fibonacci(10)
    if n == 0 || n == 1 {             // If n=0 OR n=1, (|| means OR)
        return n                      //      return n and exit
    }                                 // Recursively call
    return fibonacci(n-1)+fibonacci(n-2)  //    fibonacci(9) and fibonacci(8)
}
```

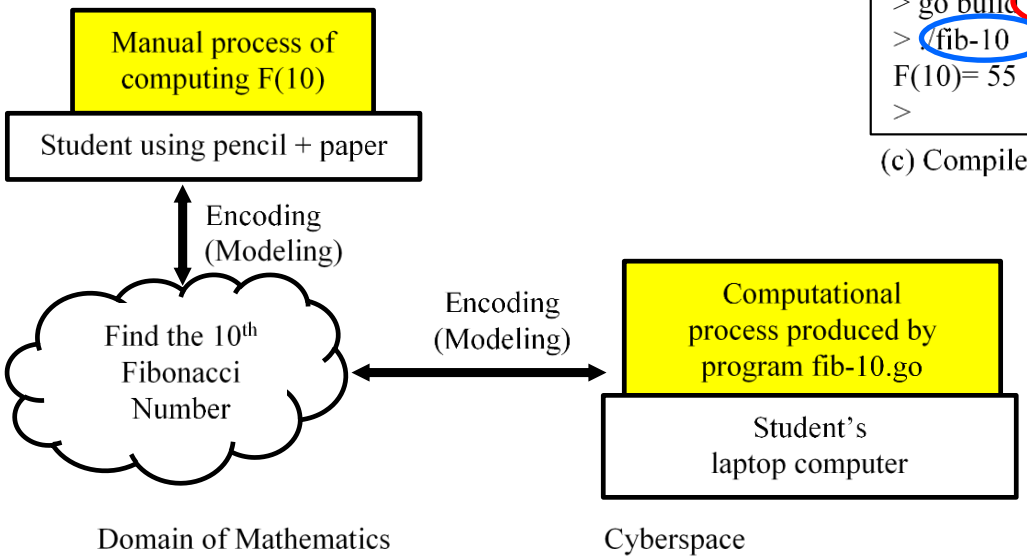
(b) A Go program fib-10.go that implements the algorithm

```
> go build fib-10.go
> ./fib-10
F(10)= 55
>
```

HLL program

Executable program, low-level language program, binary code

(c) Compile fib-10.go and execute fib-10 to produce the output



Demo

2. ABC features without

- Automatic execution
- Bit accuracy
- Constructive abstraction
- They form a synergy of information transformation, thus differs from other disciplines
  - Abstraction in CS is automatically executed and bit accurate abstraction
  - Logic in CS emphasizes automatically executed and bit accurate logic reasoning
  - Algorithm emphasizes automatically executed, bit accurate, and efficient methods of computation

2.1 Automatic execution

- Demo of executing fib-10.go

```
~> cat fib-10.go
package main                          // Program setup
import "fmt"
func main() {
fmt.Println("F(10)=", fibonacci(10))    // Output F(10)
}
func fibonacci(n int) int {           // fibonacci(10)
if n == 0 || n == 1 {                 // If n=0 OR n=1, (|| means OR)
return n                              //     return n and exit
}
return fibonacci(n-1)+fibonacci(n-2)    //   Recursive calls
}
~> go build fib-10.go          compile fib-10.go
~> ./fib-10                    execute binary code fib-10 (immediately finishes)
F(10)= 55                      display output
~>
```

Some details

- "F(10)“
  is changed to
  "F(50)"

- "fibonacci(10)"
  is changed to
  "fibonacci(50)"

- // Output F(10)“
  is changed to
  "// Output F(50)“

- "10“
  is changed to
  "50"

```
package main                                  // Program setup
import "fmt"
func main() {
   fmt.Println("F(10)=", fibonacci(10))      // Output F(10)
}
func fibonacci(n int) int {                   // fibonacci(10)
   if n == 0 || n == 1 {                      // If n=0 OR n=1, (|| means OR)
      return n                                //     return n and exit
   }                                          // Recursively call
   return fibonacci(n-1)+fibonacci(n-2)       //   fibonacci(9) and fibonac-ci(8)
}
```

```
> go build fib-10.go
> ./fib-10
F(10)= 55
>
```

(a) F(10)= 55
(b) F(10)= 12586269025
(c) F(50)= 55
(d) F(50)= 12586269025

Automatic execution

- Demo of executing fib-50.go

```
~> cat fib-50.go
package main                        // Program setup
import "fmt"
func main() {
fmt.Println("F(50)=", fibonacci(50))    // Output F(50)
}
func fibonacci(n int) int {         // fibonacci(50)
if n == 0 || n == 1 {               // If n=0 OR n=1, (|| means OR)
return n                            //     return n and exit
}
return fibonacci(n-1)+fibonacci(n-2)    //   Recursive calls
}
~> go run fib-50.go compile and execute fib-50.go (wait for ~1 minute)
F(50)= 12586269025                  display output
~>
```

Automatic execution

- Demo of executing fib-50.go

```
~> cat fib-50.go
package main                          // Program setup
import "fmt"
func main() {
fmt.Println("F(50)=", fibonacci(50))    // Output F(50)
}
func fibonacci(n int) int {          // fibonacci(50)
if n == 0 || n == 1 {                 // If n=0 OR n=1, (|| means OR)
return n                              //     return n and exit
}
return fibonacci(n-1)+fibonacci(n-2)    //   Recursive calls
}
~> go run fib-50.go compile and execute fib-50.go (wait for ~1 minute)
F(50)= 12586269025                    display output
~>
```

- Repeat for fib-100.go and see what happens
  - Terribly slow, and produce a wrong result
  - due to **overflow** (result too big to be held in a 64-bit integer)

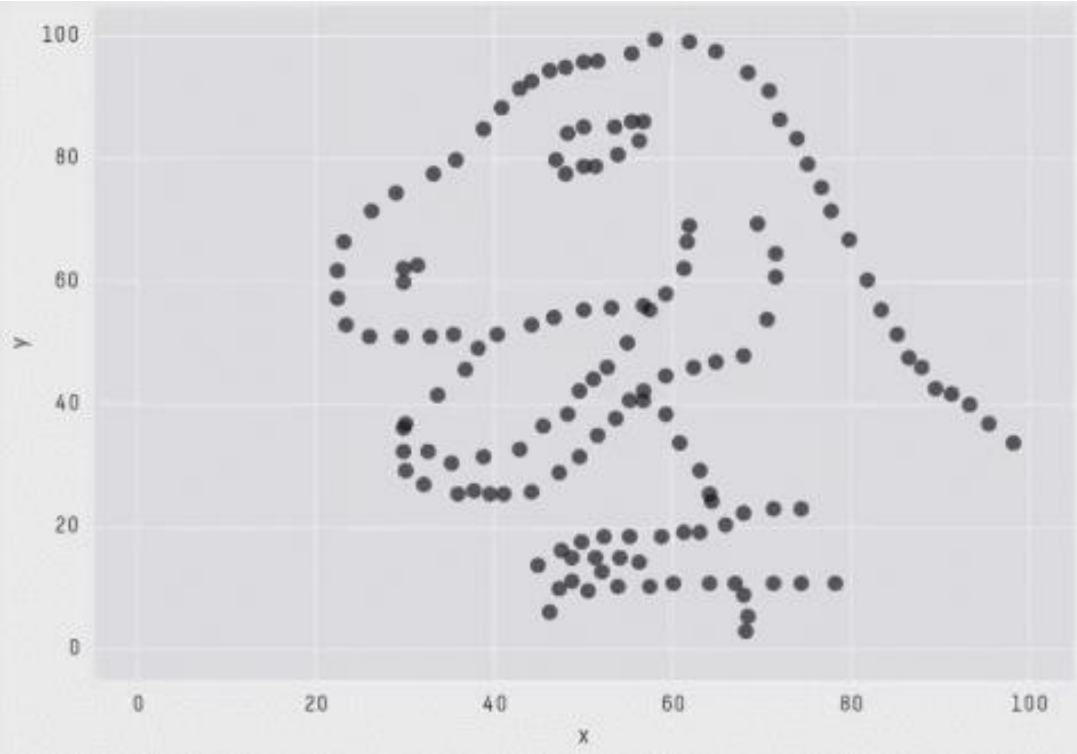
2.2 Bit accuracy

- Other sciences pursue their own scientific rigor
  - Experiments results are statistically significant when the p-value is less than 0.05
  - The error is no more than 3 Angstrom (Å = 0.1 nm)
  - The results are precise up to two digits after the decimal point

- Computer science uses binary values of 0s and 1s
  - One *binary digit* is called a **bit**
- Computer science pursues **bit accuracy**
  - A computational process is accurate and precise up to every bit
  - Any practical computer has finite memory
    - Cannot represent real numbers of arbitrary precision

The “dinosaur data sets” phenomenon

- These data sets are the **same but different**
  - show quite different graphs (varied appearance), but
  - have the same statistics (up to two digits after the decimal point)



X Mean: 54.2659224
Y Mean: 47.8313999
X SD : 16.7649829
Y SD : 26.9342120
Corr. : -0.0642526

Justin Matejka and George Fitzmaurice. (2017). Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17). ACM, New York, NY, USA, 1290-1294.

Bit accuracy

- Fib-50.go is slow
- Use Binet's formula to compute F(50)
- Involve real numbers, or floating-point numbers

Output F(50) // n and F(n) are real numbers

where $F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, and $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

(a) An algorithm to find F(50) directly from Binet's formula

```
package main
import "fmt"
import "math"                  // utilize the math library
func main() {
    fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) float64 {
    sqrt5 := math.Sqrt(5)      // assign the square root of 5 to sqrt5
    phi := (1+sqrt5)/2         // assign the golden ratio to phi
    return      (math.Pow(phi,float64(n))-math.Pow((1-phi),float64(n)))/sqrt5
}
```

(b) A Go program fib.binet-50.go that implements the algorithm

```
> go build fib.binet-50.go
> ./ fib.binet-50
F(50)= 1.2586269024999998e+10
```

(c) Compile fib.binet-50.go and execute fib.binet-50 to produce the output

Bit accuracy

- Demo of fib.binet-50.go, fib.binet-100.go, fib.binet-500.go

~> go run fib.binet-50.go **compile** and **execute** (immediately finishes)
F(50)= 1.2586269024999998e+10 **display** output
~> go run fib.binet-100.go **compile** and **execute** (immediately finishes)
F(100)= 3.542248481792618e+20 **display** output
~> go run fib.binet-500.go **compile** and **execute** (immediately finishes)
F(500)= 1.39423224561697 67e+104 **display** output

The floating-point numbers have only 16 significant digits, resulting in **round-off errors**

F(50) = 1.2586269024999998e+10 = 1258 6269 024.9 99998
F(50) = **1258 6269 025**
F(100) = 3.542 2484 8179 2618 e+20 = 3542 2484 8179 2618 00000
F(100) = **3542 2484 8179 2619 15075**
F(500) = 1.39423224561697 67e+104
= 1394 2322 4561 6976 7000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 00000
F(500) = **1394 2322 4561 6978 8013 9724 3828 7040 7283 9500 7025 6587 6973**
0726 4108 9629 4832 5571 6228 6329 0691 5576 5887 6222 5212 94125

2.3 Constructive abstraction

- Computer science is about digital abstractions
  - constructive, automatically executed abstractions of information transformation
- Three layers of meaning
  - *Abstraction* from concrete instances to the general concept
  - *Constructive*: a step-by-step integration of more primitive symbols and operations
  - *Smart* construction, not ad hoc, arbitrary actions or processes
  - Although may use brute-force actions (e.g., exhaustive enumeration) and seemingly arbitrary random operations (e.g., randomly picking a number)
- Abstraction example: recursive function

```
func fibonacci(n int) int {
   if n == 0 || n == 1 {
            return n
   }
   return fibonacci(n-1)+fibonacci(n-2)
}
```

Layer-1 meaning

Abstraction from concrete instances to the general concept

- From concrete instances

```
func fibonacci(n int) int {
    if n == 0 || n == 1 {
                return n
    }
    return fibonacci(n-1)+fibonacci(n-2)
}


fmt.Println( fibonacci(10) )
```

Instance-1: fibonacci

```
func quicksort(A []int) {
        if len(A) < 2 {
                return
        }
        lowerA, upperA := partition(A)
        quicksort(lowerA)
        quicksort(upperA)
}

quicksort( arrayX )
```

Instance-2: quicksort

to general concept
of **function**

- Function definition →

```
func functionName( parameters )  returnType {
        functionBody
}
```

  - A function takes input and returns output
  - Parameter is also called argument
- Function call →

```
functionName( actualParameters )  // function call
```

Layer-2 meaning

Constructive: a step-by-step integration of more primitive entities

- From concrete instances

```
func fibonacci(n int) int {
    if n == 0 || n == 1 {
                return n
    }
    return fibonacci(n-1)+fibonacci(n-2)
}


fmt.Println( fibonacci(10) )
```

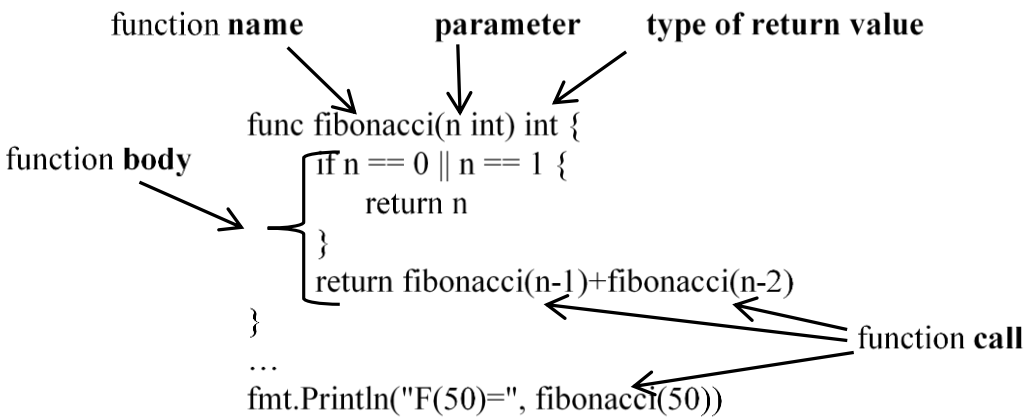
Instance-1: fibonacci

```
func quicksort(A []int) {
        if len(A) < 2 {
                return
        }
        lowerA, upperA := partition(A)
        quicksort(lowerA)
        quicksort(upperA)
}

quicksort( arrayX )
```

Instance-2: quicksort

to general concept
of four-part function



Layer-3 meaning

Smart construction

- From concrete instances

```
func fibonacci(n int) int {
    if n == 0 || n == 1 {
                return n
    }
    return fibonacci(n-1)+fibonacci(n-2)
}


fmt.Println( fibonacci(10) )
```

Instance-1: fibonacci

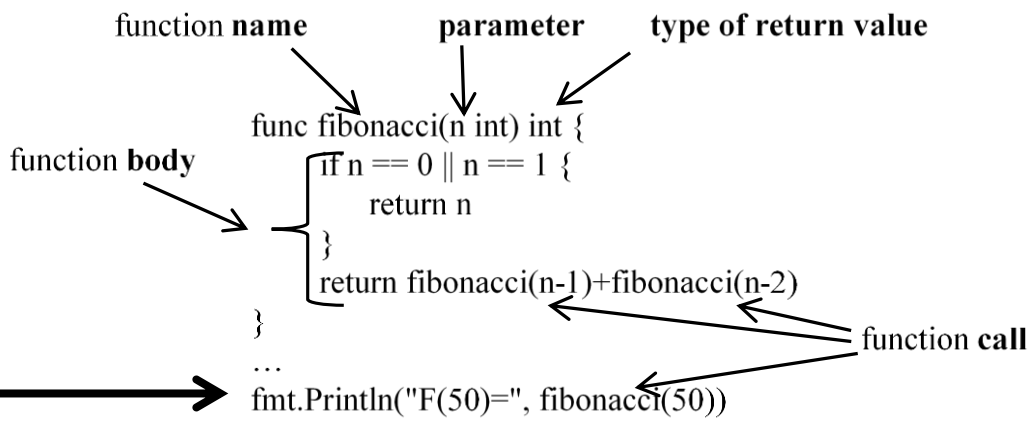
```
func quicksort(A []int) {
        if len(A) < 2 {
                return
        }
        lowerA, upperA := partition(A)
        quicksort(lowerA)
        quicksort(upperA)
}

quicksort( arrayX )
```

Instance-2: quicksort

- Define base cases and the rest
- May contain recursive call

to general concept
of four-part function
and
recursive function



May have side effect →

3. Eight understandings within (Acu-Exams)

- **A**utomatic execution. Computational processes are automatically executed step-by-step on computers.
- **C**orrectness. The correctness of computational processes can be rigorously defined and analyzed by computational models such as Boolean logic and Turing machines.
- **U**niversality. Turing machine compatible computers can be used to solve any computable problems.
- **E**ffectiveness. People are able to construct smart methods to solve problems effectively.
- comple**X**ity. These smart methods, called algorithms, have time complexity and space complexity when executed on a computer.
- **A**bstraction. A small number of carefully crafted systems abstractions can support many computing systems and applications.
- **M**odularity. Computing systems are built by composing modules.
- **S**eamless Transition. Computational processes smoothly execute on computing systems, seamlessly transitioning from one step to the next.

3.1 Automatic execution

- Computational processes are automatically executed step-by-step on computers.
- Automatic execution is common when looking inside or from outside
  - Step-by-step mechanic automatic execution of digital symbol manipulation is the most fundamental characteristic of computational thinking, both without and within.
  - It underlies all the other seven understandings.
  - CS studies logic that is automatic executable logic, algorithms that are automatic executed algorithms, abstractions that are automatic executed abstractions.
- It partially answers the question
  - Why and how trillions of instructions can be automatically executed in a fraction of a second, sometimes across the globe, to produce correct computational results?

3.2 Correct, smart, and practical processes

- **A**utomatic execution. Computational processes are a
  executed step-by-step on computers.
- **C**orrectness. The correctness of computational proce
  rigorously defined and analyzed by computational mo
  Boolean logic and Turing machines.
- **U**niversality. Turing machine compatible computers c
  any computable problems.
- **E**ffectiveness. People are able to construct smart met
  problems effectively.
- comple**X**ity. These smart methods, called algorithms,
  complexity and space complexity when executed on a
- **A**bstraction. A small number of carefully crafted syste
  support many computing systems and applications.
- **M**odularity. Computing systems are built by composin
- **S**eamless Transition. Computational processes smoo
  computing systems, seamlessly transitioning from on

How to make computational processes correct, smart, and practical?

Logic thinking (C, U); Chapter 3

Algorithmic thinking (E, X); Chapter 4

Systems thinking (A, M, S); Chapter 5

3.3 Computing Fibonacci numbers F(n)

- Q: When the input data and the algorithm are correct, will the program execution successfully produce the correct result?
- A: Not necessarily. E.g., compute F(n) when n = 1 billion
- Q: Why not?
- A: There are many possible reasons, such as
  1. The algorithm or program is too slow to finish in reasonable amount of time.
  2. Wrong result due to overflow error.
     - The data type (64-bit integer) used has too small a word length to hold the result.
  3. Other compile-time or run-time errors.
  4. The program and its data are too big for the computer to hold (not enough memory).
- The program fib.go is
  - not smart: performs too many repetitive computations
  - not correct: produces wrong results starting at F(93)
  - **not practical: F(100) needs hundred-thousand years**

3.3 Computing Fibonacci numbers (demo)

- Use program fib.all-40.go and fib.all-41.go to see why the code is slow
  - Which output not only F(n), but also F(0), F(1), …, F(n-1)
  - "time" is a command to measure execution time of the code
- Observation
  - Suppose T(n) is the execution time to compute F(n) by fib.go.
  - Then, $T(n+1) \approx 1.6 \times T(n)$; when n increases by 10 to 50, T increases $1.6^9 = 68$ times

Program fib.all-40.go

```
package main
import "fmt"
func main() {
        for n:=0; n<=40; n++ {
                fmt.Println("F(",n,")=", fibonacci(n))
        }
}
func fibonacci(n int) int {
        if n == 0 || n == 1 {
                return n
        }
        return fibonacci(n-1)+fibonacci(n-2)
}
```

```
> time go run fib-40-all.go
F( 0 )= 0
F( 1 )= 1
F( 2 )= 1
F( 3 )= 2
F( 4 )= 3
…
F( 38 )= 39088169
F( 39 )= 63245986
F( 40 )= 102334155

real    0m1.463s
user    0m1.391s
sys     0m0.141s
```

3.3 Computing Fibonacci numbers (demo)

- Observation
  - Suppose T(n) is the execution time to compute F(n) by fib.go.
  - Then, $T(n+1) \approx 1.6 \times T(n)$
- If T(50)=725 s, then
  - $T(100) \approx 1.6^{100-50} \times T(50)$
  - $T(500) \approx 1.6^{500-50} \times T(50)$
- Both are judged to be impractical

Execution time to compute F(n), in seconds

n	fib.go
50	725 seconds
100	1.17×10^{13} seconds = 369 thousand years
500	3.57×10^{93} seconds = 1.13×10^{86} years
5,000,000	Too big to compute

Values may vary depending on the computer used

3.3 Computing Fibonacci numbers (demo)

- Use program fib.dp.all-50.go to speed up

Program fib.dp.all-50.go

```
package main
import "fmt"
func main() {
for n:=0; n<51; n++ {
   fmt.Println("F(",n,")=", fibonacci(n))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1) // make a slice fib
   fib[0] = 0                       // initialize fib[0] and fib[1]
   fib[1] = 1
   for i := 2; i <= n; i++ {        // iteratively compute fib[i]
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

```
> time go run fib.dp.all-50.go
F( 0 )= 0
F( 1 )= 1
F( 2 )= 1
F( 3 )= 2
F( 4 )= 3
…
F(42)=267914296
F(43)=433494437
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
F(50)=12586269025

real    0m0.207s
user    0m0.094s
sys     0m0.109s
```

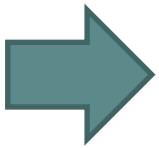
Program fib.dp.go can be simplified

- Program fib.dp.go still produces wrong results starting at F(93)
- Program fib.dp.big.go corrects it by using a new data type big.Int

- Three different definitions of the fibonacci function

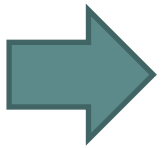
in fib.dp.go

```
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1)
   fib[0] = 0
   fib[1] = 1
   for i := 2; i <= n; i++ {
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```



in simplified fib.dp.go

```
func fibonacci(n int) int {
    var a, b, i int
    a, b = 0, 1
    for i = 1; i < n+1; i++ {
        a = a + b
        a, b = b, a
    }
    return a
}
```



in fib.dp.big.go

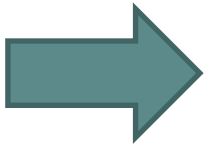
```
func fibonacci(n uint) *big.Int {
        a := big.NewInt(0)
        b := big.NewInt(1)
        for i := uint(1); i < n+1; i++ {
                a.Add(a, b)
                a, b = b, a
        }
        return a
}
```

Program fib.dp.go can be simplified

- Program fib.dp.go still produces wrong results starting at F(93)
  - Program fib.dp.big.go corrects it by using a new data type big.Int.

```
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1)
   fib[0] = 0
   fib[1] = 1
   for i := 2; i <= n; i++ {
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

```
func fibonacci(n int) int {
    var a, b, i int
    a, b = 0, 1
    for i = 1; i < n+1; i++ {
        a = a+b
        a, b = b, a
    }
    return a
}
```



```
func fibonacci(n uint) *big.Int {
        a := big.NewInt(0)
        b := big.NewInt(1)
        for i := uint(1); i < n+1; i++ {
                a.Add(a, b)
                a, b = b, a
        }
        return a
}
```

Execution time to compute F(n), in seconds

n	fib.go	fib.dp.go	fib.dp.big.go	fib.matrix.go
50	725	0.059	0.019	0.000012
500	Error	Error	0.026	0.000022
5,000,000	Error	Error	102	4.13
1,000,000,000	Error	Error	Killed after 2 days	187,160

Values may vary depending on the computer used

3.3 Computing F(1,000,000,000)

- Program fib.dp.big.go is still too slow.
- Program fib.matrix.go uses a smart algorithm, based on a fact
  - $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$, and
  - exponentiation by squaring
- Program fib.matrix.go is correct, smart, and practical for computing F(n), up to n = 1 billion
  - It computes F(1,000,000,000) in 2.17 days.

Execution time to compute F(n), in seconds

n	fib.go	fib.dp.go	fib.dp.big.go	fib.matrix.go
50	725	0.059	0.019	0.000012
500	Error	Error	0.026	0.000022
5,000,000	Error	Error	102	4.13
1,000,000,000	Error	Error	Killed after 2 days	187,160

Values may vary depending on the computer used

4. CS and CT are a synergy & a symphony

- Three different viewpoints of the same CS
  - Georg Gottlob (Oxford)
    - Computer science is the continuation of **logic** by other means
  - Richard Karp (UC-Berkeley)
    - Computational lens (also known as **algorithmic** lens)
  - Joseph Sifakis (CNRS)
    - **System** design science

- Classic: Yang Xiong's Canon of Supreme Mystery
  - 《太玄经·差首》：𝌢 帝由群雍，物差其容。
  - Head *Cha* (Diversity) 𝌢: The way emerges from the multitude of harmonies, where things diverge in their appearances.
  - Computer science is like a musical symphony. Many instruments produce different sounds while playing the same music. Each instrument offers its distinct contribution. The diversity of their differences creates a harmonic whole of the symphony.
  - Logic thinking, algorithmic thinking and systems thinking together produce the totality of computational process, that is correct, smart, and practical.