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

CS101

Algorithmic Thinking

Dynamic Programming, Randomization

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

Outline

- What is algorithmic thinking
- Divide-and-conquer paradigm
- Other interesting paradigms
  - Dynamic Programming
  - Randomization
  - Greedy***
- P vs. NP

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

3. Paradigms other than divide-and-conquer

- Dynamic programming
  - What if subproblems have overlapping elements
- Randomization
  - Avoid being trapped in a bad situation
- Greedy***
  - Try the obviously best from the possible next steps
- Hashing***
  - Search algorithms

3.1 Dynamic programming

- The divide and conquer paradigm desires a problem partition
  - Partition: subproblems do not overlap
- What does "subproblems overlap" mean?
  - Subproblems have repetitive common computations
    - F(5) = F(4) + F(3), where the subproblems F(4) and F(3) have common computation F(2)
    - F(4) = F(3) + F(2)
    - F(3) = F(2) + F(1)

Divide and conquer in computing F(n)

Divide a problem into
Base case
and
Two subproblems

Conquer by solving the subproblems

Combine by addition

fib-5.go

```
package main
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1) + fibonacci(n-2)
}
```

3.1 Dynamic programming

- What if subproblems do overlap?
  - Could have repetitive and unnecessary computation and drastically increase time complexity
    - Recursive Fibonacci computation F(*n*) needs O(2^n)
    - Add **fmt.Println(“F(”,n,“)”)** to reveal unnecessary computation, shown in red

fib-5.go

```
package main
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1)+fibonacci(n-2)
}
```

```
> go run fib-5.go
F( 5 )
F( 4 )
F( 3 )
F( 2 )
F( 1 )
F( 0 )
F( 1 )
F( 2 )
F( 1 )
F( 0 )
F( 3 )
F( 2 )
F( 1 )
F( 0 )
F( 1 )
F(5)= 5
>
```

3.1 Dynamic programming

- What if subproblems do overlap?
  - Could have repetitive and unnecessary computation and drastically increase time complexity
    - Recursive Fibonacci computation F(n) needs O(2^n)
    - When n = 5, there are 9 repetitive and unnecessary function calls
      - ⑦⑧⑨⑩⑪⑫⑬⑭⑮

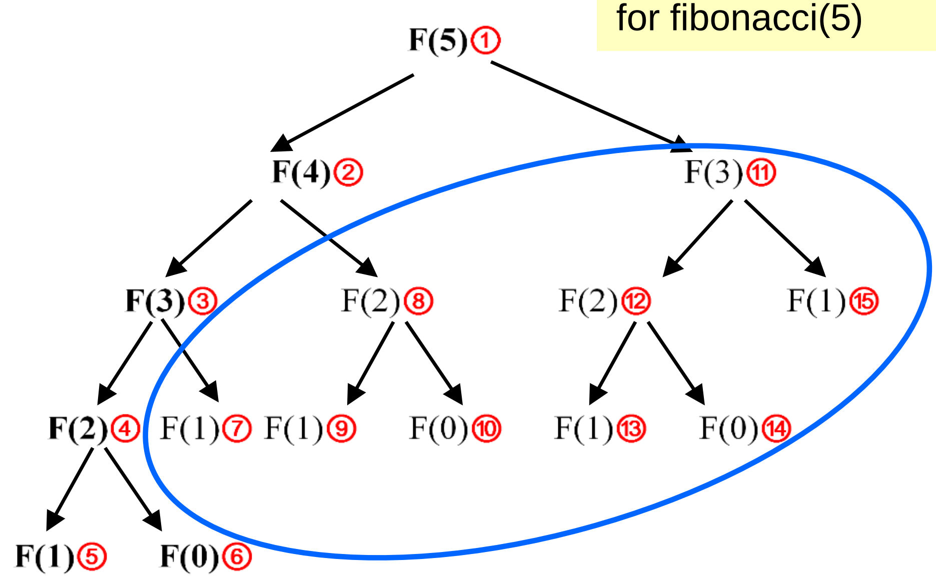
Use a call-graph to make it clearer

F(5) is shorthand for fibonacci(5)

fib-5.go

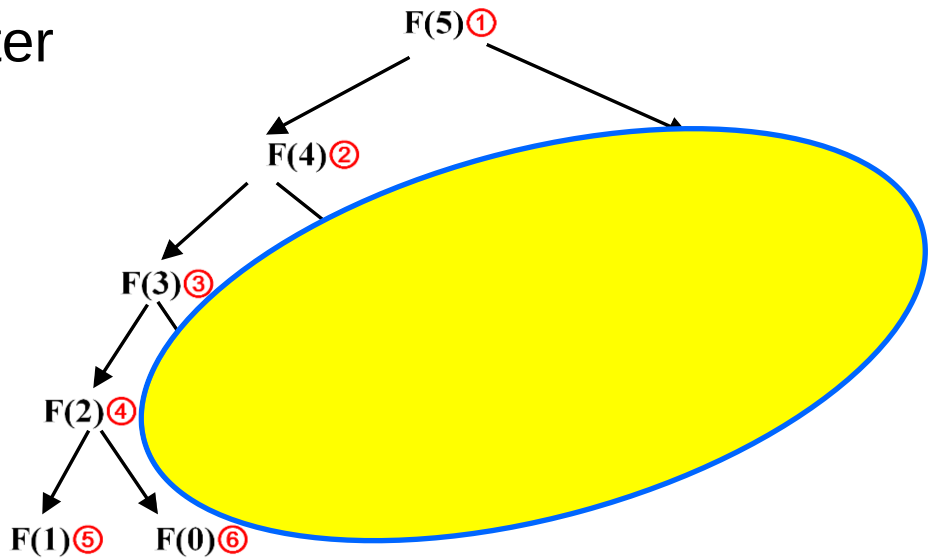
```go
package main
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1)+fibonacci(n-2)
}
```

```
> go run fib-5.go
F( 5 )     ①
F( 4 )     ②
F( 3 )     ③
F( 2 )     ④
F( 1 )     ⑤
F( 0 )     ⑥
F( 1 )     ⑦
F( 2 )     ⑧
F( 1 )     ⑨
F( 0 )     ⑩
F( 3 )     ⑪
F( 2 )     ⑫
F( 1 )     ⑬
F( 0 )     ⑭
F( 1 )     ⑮
F(5)= 5
>
```



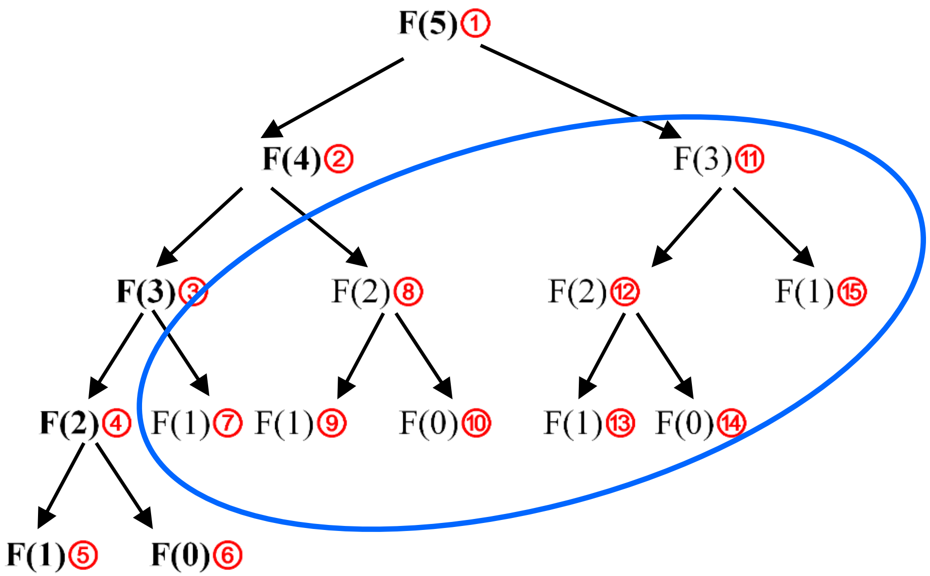
3.1 Dynamic programming

- What if subproblems do overlap?
  - Could have repetitive and unnecessary computation and drastically increase time complexity
    - Recursive Fibonacci computation F(n) needs O(2^n)
    - When n = 5, there are 9 repetitive and unnecessary function calls
- Dynamic programming to the rescue
  - Memorize and reuse to
    avoid repetitive computations   (fib.dp-5.go)
- Computation becomes much faster
  - Computing F(n) only needs O(n)
  - When n = 5, avoid all the 9 repetitive and unnecessary function calls
  - Only need to execute
    the 6 function calls in bold



Note：instead of “memorize”, standard texts use the word “memoize”, i.e., store solutions of subproblems in a memo

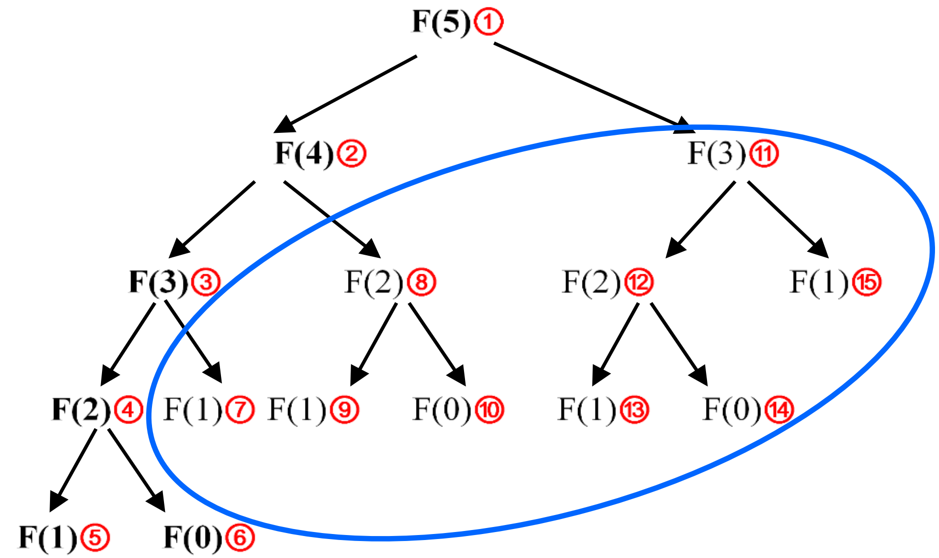
- fib.dp-5.go vs. fib-5.go
  - Use an array mem to store the computed values
  - Initialize mem[i] to -1




```
package main    fib-5.go
import "fmt"

func main() {

   fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
   fmt.Println("F(",n,")")


   if n == 0 || n == 1 {

      return n
   }

   return fibonacci(n-1)+fibonacci(n-2)
}
```

```
package main      // fib.dp-5.go
import "fmt"
var mem [6]int
func main() {
   for i := 0; i < 6; i++ { mem[i] = -1 }
   fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
   fmt.Println("F(",n,")")
   if mem[n] != -1 {  // immediately return to avoid repetitive operations
      return mem[n]
   }
   if n == 0 || n == 1 {
      mem[n] = n
      return mem[n]
   }
   mem[n] = fibonacci(n-1) + fibonacci(n-2)
   return mem[n]
}
```

- Add printing statements to show the sequence of execution steps
  - Avoid all repetitive calls
    - Immediate returns




```
package main      // fib.dp-5.go
import "fmt"
var mem [6]int
func main() {
   for i := 0; i < 6; i++ { mem[i] = -1 }
   fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
   fmt.Println("F(",n,")")
   if mem[n] != -1 {  // immediately return
      fmt.Println("Immediate Return: F(",n,")=",mem[n])
      return mem[n]
   }
   if n == 0 || n == 1 {
      mem[n] = n
      fmt.Println("Return: F(",n,")=",mem[n])
      return mem[n]
   }
   mem[n] = fibonacci(n-1) + fibonacci(n-2)
   fmt.Println("Return: F(",n,")=",mem[n])
   return mem[n]
}
```

```
> go run fib.dp-5.go
F( 5 )
F( 4 )
F( 3 )
F( 2 )
F( 1 )
Return: F( 1 )= 1
F( 0 )
Return: F( 0 )= 0
Return: F( 2 )= 1
F( 1 )
Immediate Return: F( 1 )= 1
Return: F( 3 )= 2
F( 2 )
Immediate Return: F( 2 )= 1
Return: F( 4 )= 3
F( 3 )
Immediate Return: F( 3 )= 2
Return: F( 5 )= 5
F(5)= 5
>
```

Repetitive calls avoided

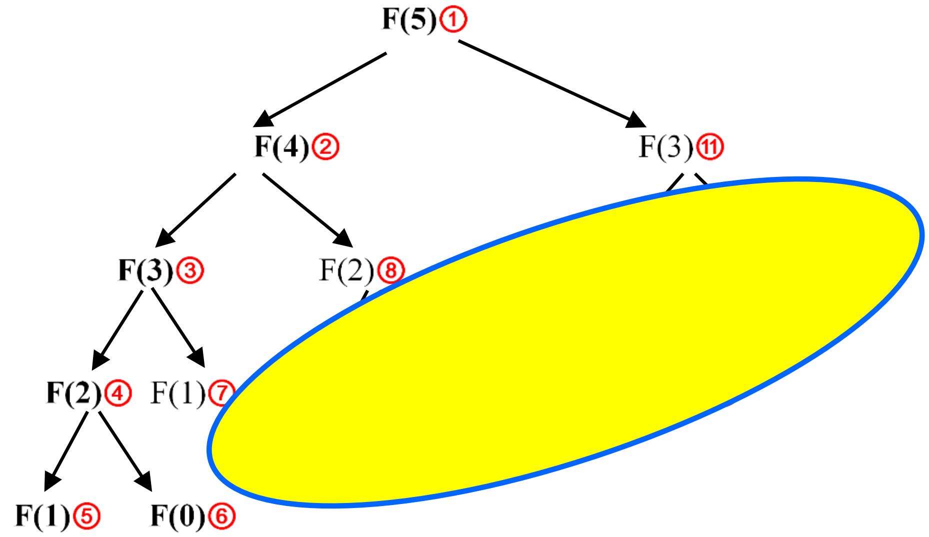
Actually, the code repetitively calls F(1), F(2), F(3) but immediately returns

7

8 9 10

11 12 13 14 15

- Add printing statements to show the sequence of execution steps
  - Avoid all repetitive calls
    - Immediate returns




```
package main      // fib.dp-5.go
import "fmt"
var mem [6]int
func main() {
  for i := 0; i < 6; i++ { mem[i] = -1 }
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if mem[n] != -1 {  // immediately return
    fmt.Println("Immediate Return: F(",n,")=",mem[n])
    return mem[n]
  }
  if n == 0 || n == 1 {
    mem[n] = n
    fmt.Println("Return: F(",n,")=",mem[n])
    return mem[n]
  }
  mem[n] = fibonacci(n-1) + fibonacci(n-2)
  fmt.Println("Return: F(",n,")=",mem[n])
  return mem[n]
}
```

```
> go run fib.dp-5.go
F( 5 )
F( 4 )
F( 3 )
F( 2 )
F( 1 )
Return: F( 1 )= 1
F( 0 )
Return: F( 0 )= 0
Return: F( 2 )= 1
F( 1 )
Immediate Return: F( 1 )= 1
Return: F( 3 )= 2
F( 2 )
Immediate Return: F( 2 )= 1
Return: F( 4 )= 3
F( 3 )
Immediate Return: F( 3 )= 2
Return: F( 5 )= 5
F(5)= 5
>
```

Two fib.dp programs

Top down

Compute F(n), F(n-1), F(n-2), …, F(0)

```
package main      // fib.dp-5.go
import "fmt"
var mem [6]int
func main() {
  for i := 0; i < 6; i++ { mem[i] = -1 }
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if mem[n] != -1 {  // immediately return
    fmt.Println("Immediate Return: F(",n,")=",mem[n])
    return mem[n]
  }
  if n == 0 || n == 1 {
    mem[n] = n
    fmt.Println("Return: F(",n,")=",mem[n])
    return mem[n]
  }
  mem[n] = fibonacci(n-1) + fibonacci(n-2)
  fmt.Println("Return: F(",n,")=",mem[n])
  return mem[n]
}
```

Bottom up

Compute F(0), F(1), …, F(n)

```
package main   // fib.dp.bu-5.go
import "fmt"
func main() {
   fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
   a :=0
   b :=1
   for i :=1; i < n+1; i++ {
      a = a + b
      a, b = b, a
   }
   return a
}
```

3.2 Randomization in quicksort

- Why random pivoting in quicksort?
  - In the Partition subroutine:
  - Randomly extracts a pivot element x=A[q] from the array A[p,…, r], and then partition A[p,…, r] into A[p,…,q-1], x, A[q+1,…,r]; such that element in lower array <= x element in upper array >= x
  - What if we do not do this randomization?
    - Random selection of a pivot

$p, r = n, 1$

QuickSort(A, p, r)

If $p < r$

1. q = Partition(A, p, r)
2. QuickSort(A, p, q-1)
3. QuickSort(A, q+1, r)

Without randomization

e.g., using A[1] as pivot

N=8 elements

[1,2,3,5,6,8,9,10]	
[], 1, [2,3,5,6,8,9,10]	O(N)
[2,3,5,6,8,9,10]	
[], 2, [3,5,6,8,9,10]	O(N-1)
[3,5,6,8,9,10]	
[], 3, [5,6,8,9,10]	O(N-2)
…	
[9,10]	
[], 9, [10]	1
[10]	stop

Need $O(N^2)$ in total

3.2 Randomization in quicksort

- Why random pivoting in quicksort?
  - In Partition: randomly extracts a pivot element x=A[q] from array A[p,…, r], and then partition A[p,…, r] into A[p,…,q-1], x, A[q+1,…,r]; such that element in lower array <= x element in upper array >= x

With randomization
N=8 elements

[1,2,3,5,6,8,9,10]	pivot = 6
[1,2,3,5], 6, [8,9,10]	O(N)
[1,2,3,5]	pivot = 2
[1], 2, [3,5]	O(N/2)
[3,5]	pivot = 3
[], 3, [5]	O(N/4)
[8,9,10]	pivot = 9
[8], 9, [10]	O(N/2)

Need $O(N \log N)$ in total

$p, r = n, 1$

QuickSort(A, p, r)

  If $p < r$

  1. q = Partition(A, p, r)
  2. QuickSort(A, p, q-1)
  3. QuickSort(A, q+1, r)

For more details

- Code of a Go function for quicksort
  - with randomization
  - the for loop examines the array, going from left to right

```
func quicksort(A []int) {
        if len(A) < 2 { return }
        lowerA, upperA := partition(A)
        quicksort(lowerA)
        quicksort(upperA)
}
func partition(A []int) ([]int, []int) {            // return lower and upper slices as output
        pivotIndex := rand.Intn(len(A))             // randomly select a pivot
        pivotValue := A[pivotIndex]
        lower := 0
        A[pivotIndex], A[len(A)-1] = A[len(A)-1], A[pivotIndex]
        for i:= 0; i<len(A); i++ {
            // insert your code here to realize the following logic:
            //      if A[i]<pivot then {exchange A[i] and A[lower],and update lower}
        }
        A[lower], A[len(A)-1] = A[len(A)-1], A[lower]
        return A[0:lower], A[lower+1:len(A)]
}
```