Data Structures and Algorithms

Lecture 02 – Josephus

（Array、linked-list and Recursive）

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

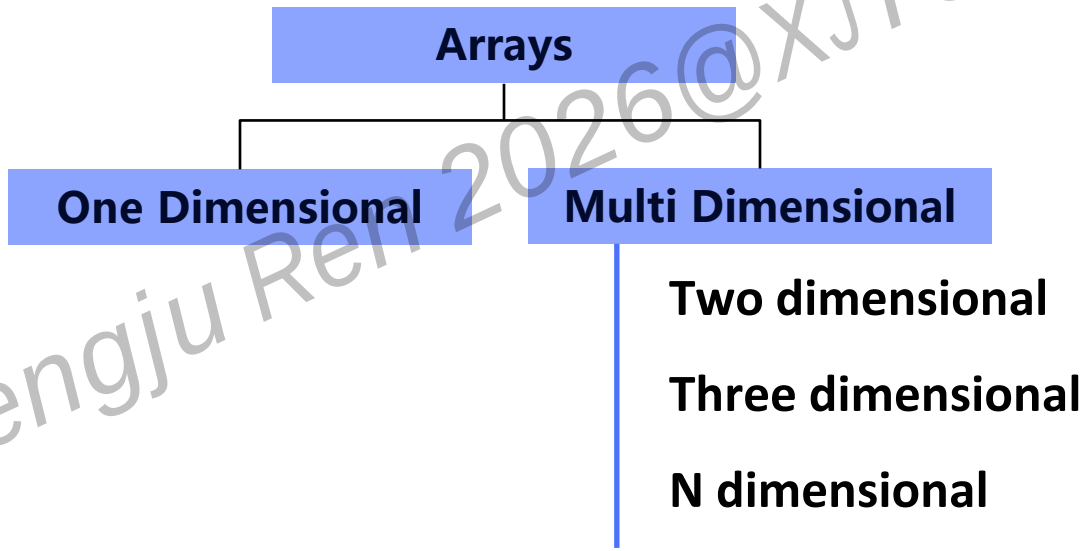
Problem Solving

How to survive in the Josephus ?

Intro of Array and Linked-list

Array

- Array can store data of specified type
- Elements of an array are located in a contiguous manner
- Each element of an array has a unique index



Types of Array

One dimensional array: an array with a bunch of values having been declared with a single index

`a[i]` → `i between 0 and n`

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	**1**	**2**	**3**	**5**	**8**	**13**	**21**

Two dimensional array: an array with a bunch of values having declared with double index

`a[i][j]` → `i and j between 0 and n`

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**
[1]	**34**	**55**	**89**	**144**	**233**	**377**	**610**	**987**

Arrays in Memory

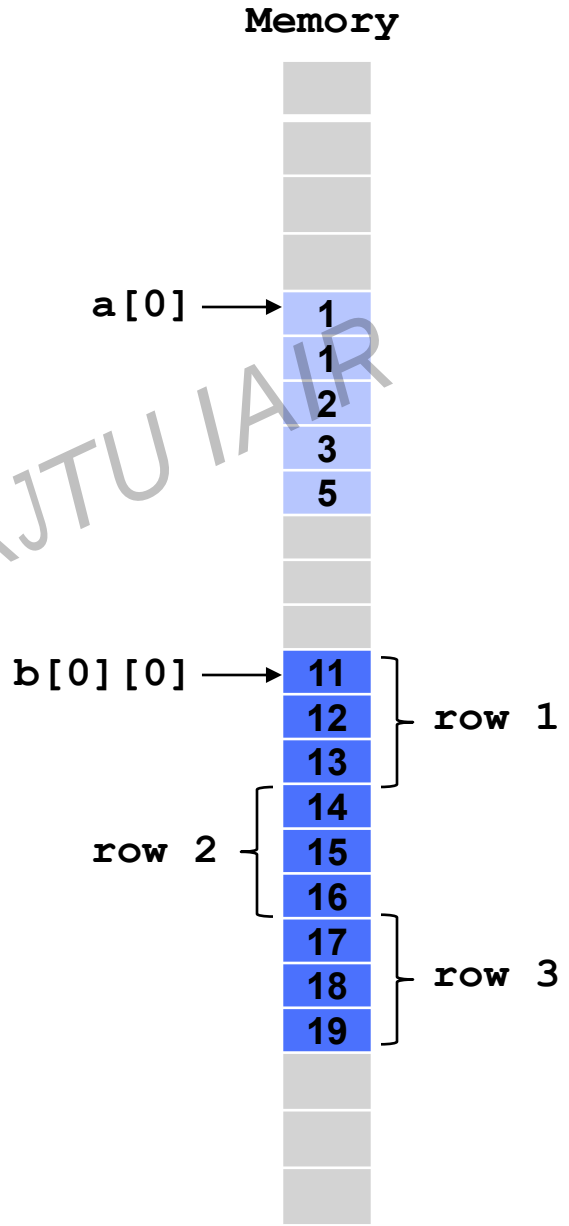
One dimensional: a [ ]

1	1	2	3	5

Two dimensional: b [ ] [ ]

11	12	13
14	15	16
17	18	19

Three dimensional: c [ ] [ ] [ ]

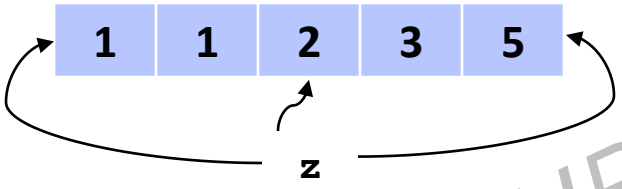


Examples of Array in Python

```
# 1-D  Array
arr = [10, 20, 30, 40, 50]
# 2-D  Array a.k.a Matrix
matrix = [
   [1, 2, 3],
   [4, 5, 6]
]
# 3-D Cubic
depth = 2
rows = 3
cols = 4
cubic = [[[0 for _ in range(cols)] for _ in range(rows)] for _ in range(depth)]
```

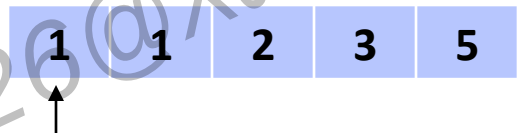
Basic Op in Arrays

- ***Insertion*** **to an Array**



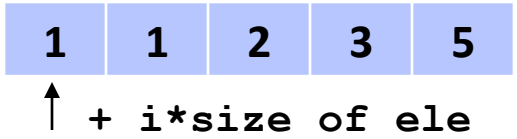
Time Complexity: ? , Space Complexity: ?

- ***Traversal*** **an Array**



Time Complexity: ? , Space Complexity: ?

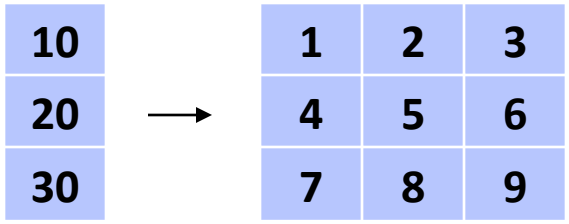
- ***Access*** **Array element (read/write element at an index)**



Time Complexity: ? , Space Complexity: ?

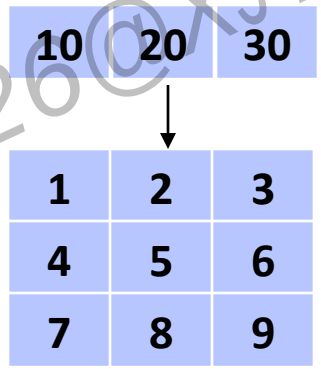
Basic Ops in 2D-Array

- Insertion a column



Time Complexity: ? , Space Complexity: ?

- Insertion a row



Time Complexity: ? , Space Complexity: ?

- Delete a row/column
- Search an element

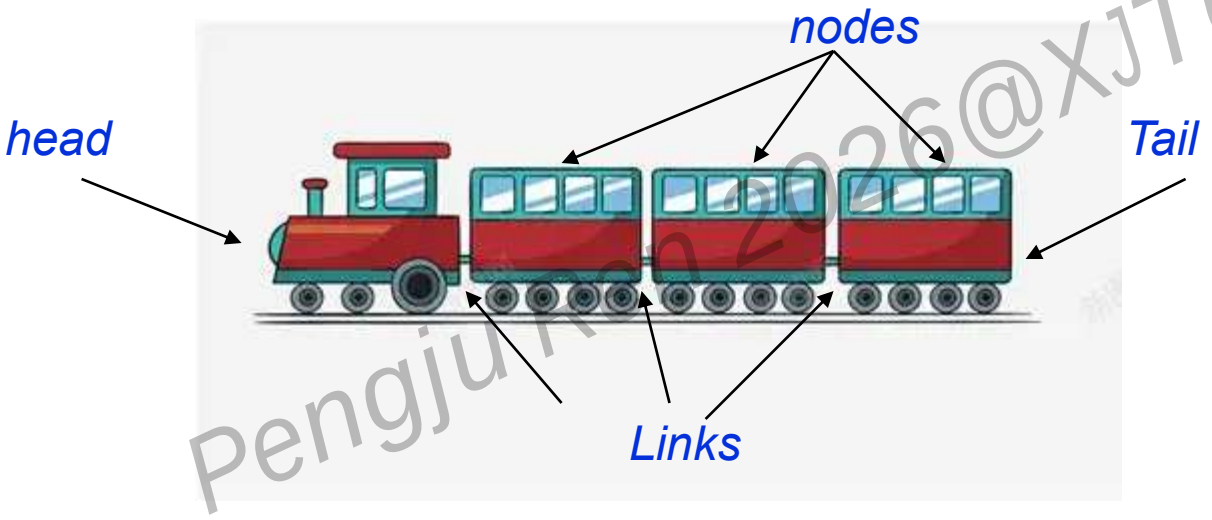
Why do we need arrays?

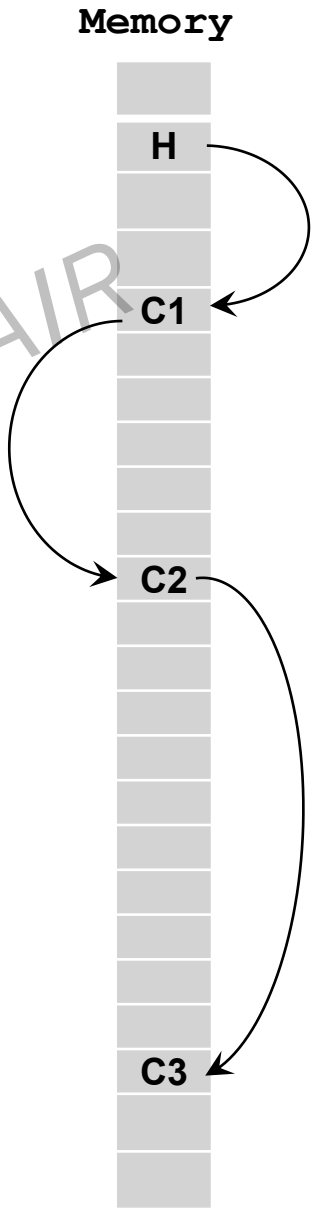
3 variables: num1, num2, num3

What if 5000 integer? Are we going to use 5000 variables?

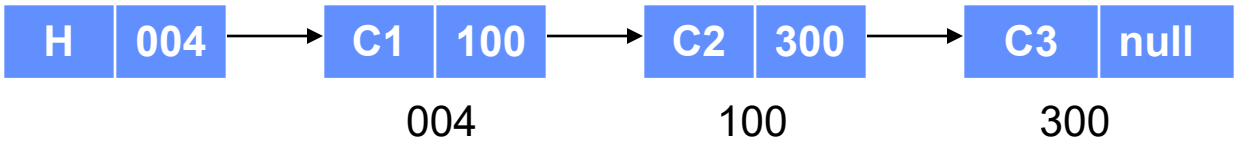
What is a linked list?

Linked list **is a form of sequential collection and it does not have to be in order. A** ***linked list*** **is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link**





Each ***node*** **is independent,** ***node*** **contains** ***data*** **and** ***links***



Why do we use linked list?

Problems of Arrays:

- **Arrays are of fixed size**
- **Arrays stores data sequentially in the memory**

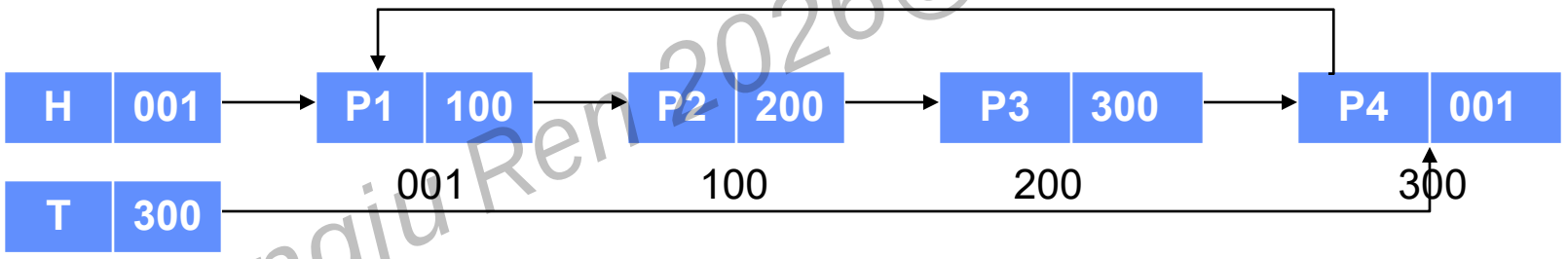
In reality we don't know how many elements we need to store

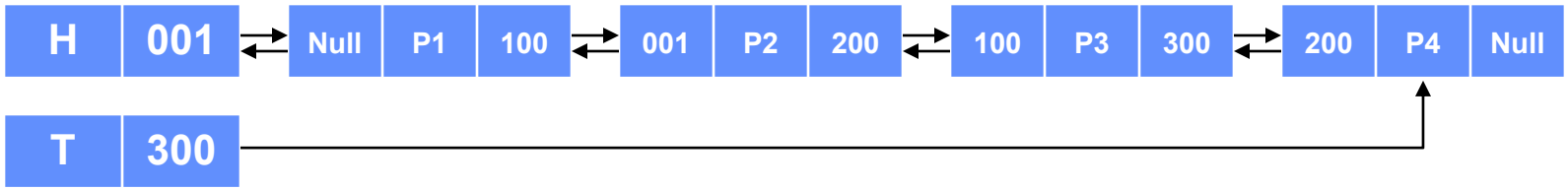
Need of a Data Structure that grows and shrinks

Efficient utilization of Memory

Types of Linked Lists

- Singly linked list
- Circular Single linked list
- Doubly linked list
- Circular Doubly linked list





Examples of Linked-list in Python

```
class SinglyNode:
    def __init__(self, value):
        self.value = value
        self.next = None
```

```
class DoublyNode:
    def __init__(self, value):
        self.value = value
        self.prev = None
        self.next = None
```

Basic Ops in Linked list

- **Insertion to linked list**

  At the beginning/In the middle/At the end

- **Prepend to the linked list**

  Adding new node in the front of the linked list

- **Append to linked list**

  Adding new node at the end of the linked list

Dynamic Array Allocation

Linked lists can be easily expanded; how about arrays?

Dynamic Arrays (Python's *list*)

- **Relax constraint of size of array #items in sequence**
- **Allocate new array of 2x size, and copy the first part**
- **The cost is amortized cost = $O(n)$**

$$1+2+4+8+\ldots n = \sum_{i=0}^{log_2 n} 2^i = 2^{log_2 n+1} = n+1$$

The core principle of Recursion

What is Recursion?

Recursion : a way of solving a problem by having a function *calling itself*

- **Performing the same operation multiple times with *different inputs***
- **In every step we try smaller inputs to make the *problem smaller***
- **Base condition is needed to *stop the recursion*, otherwise infinite loop will occur**

Why is Recursion?

Recursion thinking is really important in programming and it helps you break down problems into smaller ones and easy to use

- **The prominent usage of recursion in data structures like *trees* and *graphs***
- **It is used in many algorithms (*divide and conquer*, *greedy* and *dynamic programming*)**

How to write recursion in 3 steps?

Step1: Base case – the stopping criterion

$1! = 1$

$0! = 1$

Step2: Recursive case – the working flow problem become small

$n! = n * (n-1) * (n-2) * \ \ldots * 2 \ * 1$

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

```
factorial(n) = n*factorial(n-1)
```

Step3: Unintentional case – the constraint

```
factorial(-10)?
factorial(3.14)?
```

Example: factorial

How to write recursion in 3 steps?

Step1: Base case – the stopping criterion

`fib(0)` and `fib(1)` both are `1`

Step2: Recursive case – the working flow problem become small

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

```
fib(n) = fib(n-1)+fib(n-2)
```

Step3: Unintentional case – the constraint

```
fib(-10)?
fib(3.14)?
```

Example: fibonacci

How Recursion works?

1. A method calls itself
2. Exit from infinite loop

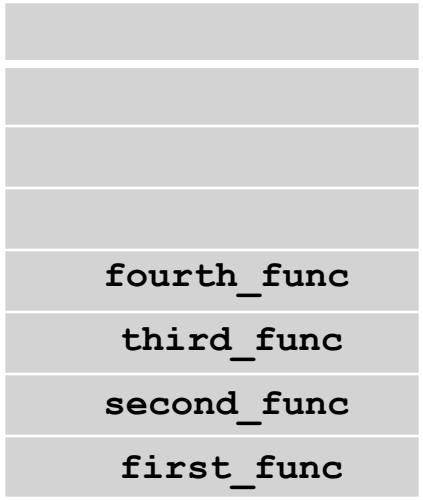
```
def recursiveMethod(parameters)
    if exit from condition satisfied:
       return some value
    else
       recursionMethod(Modified parameters)
```

```
def first_iteration():
    second_iteration()
    print("I am the first iteration")

def second_iteration():
    third_ iteration()
    print("I am the second iteration")

def third_iteration():
    fourth_iteration()
    print("I am the third iteration")

def fourth_iteration():
    print("I am the fourth iteration")
```

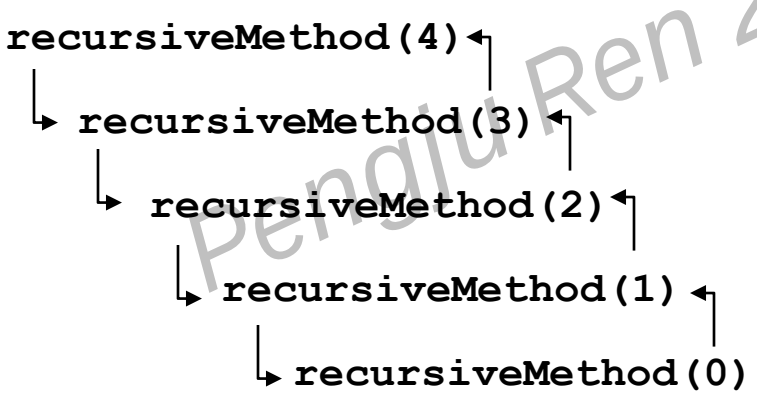


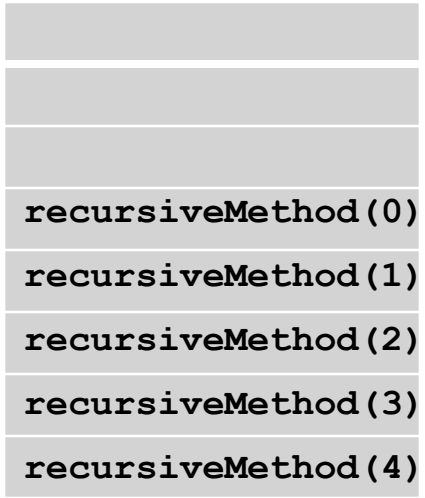
Memory Stack

How Recursion works?

1. A method calls itself
2. Exit from infinite loop

```
def recursiveMethod(n)
    if n<1:
       print("n is less than 1")
    else
       recursionMethod(n-1)
       print(n)
```





Memory Stack

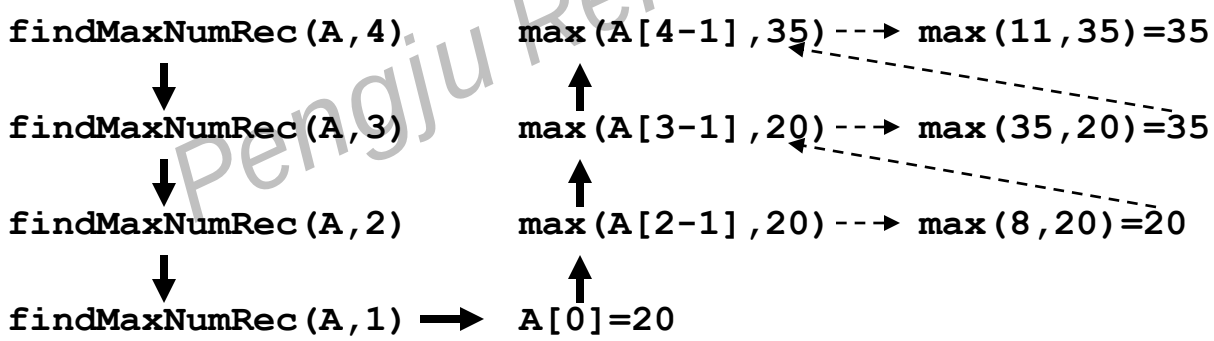
How to measure Recursive Algorithm?

A sampleArray

20	8	35	11	58	43	60	21	110

```
def findMaxNumRec(sampleArray, n):
    if n==1:
       return sampleArray[0]
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1)
```

20	8	35	11



$T(n)=T(n-1)+O(1)$

$=T(n-2)+O(1)+O(1)$

...

$=T(1)+(n-1)O(1)$

$=nO(1)$

$=O(n)$

What about the time complexity of Fibonacci ?

Types of Recursion

Tail Recursion

Head Recursion

Tree Recursion

Indirect Recursion

```
def cal1(n):
    if n>0:
        k=n**2
        print(k)
        cal1(n-1)
```

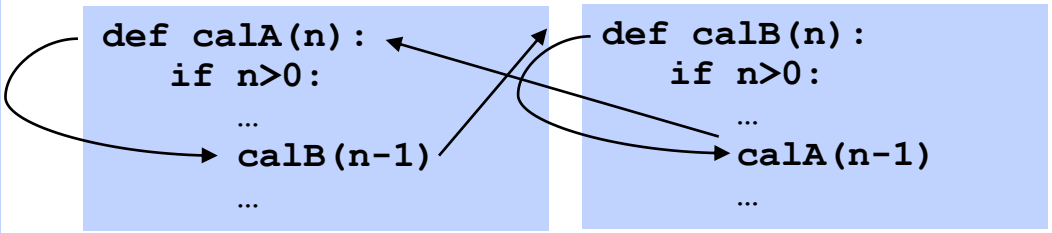
```
def cal2(n):
    if n>0:
        cal2(n-1)
        k=n**2
        print(k)
```

```
cal1(10): 100,81,64,49,36,25,16,9,4,1
cal2(10): 1,4,9,16,25,36,49,64,81,100
```

```
def is_even(n):
    if n == 0:
        return True
    else:
        return is_odd(n-1)

def is_odd(n):
    if n == 0:
        return False
    else:
        return is_even(n-1)
```

```
fib(int n)
{
 if n<=2;
      return 1;
  else
      return fib(n-1)+fib(n-2);
}
```



Recursive v.s Iterative

```
def PowerofTwo(n)
    if n==0:
       return 1
    else
       power=PowerofTwo(n-1)
       return power*2
```

```
def PowerofTwo(n)
    i=0
    power=1
    while(i<n):
      power = power*2
      i += 1
    return power
```

Recursive v.s Iterative

```
def bin_search_iter(arr, target):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
      mid = (lo + hi) // 2
      if arr[mid] == target: return mid
      if arr[mid] < target: lo = mid + 1
      else: hi = mid - 1
    return -1
```

```
def bin_search_recur(arr, target, lo=0, hi=None):
    hi = hi or len(arr) - 1
    if lo > hi: return -1
    mid = (lo + hi) // 2
    if arr[mid] == target: return mid
    if arr[mid] < target:
       return bin_search_recur(arr,target,mid + 1,hi)
    return bin_search_recur(arr,target,lo,mid - 1)
```


Points	Recursion	Iteration	notes
Space Efficient	N	Y	No Stack Memory require in case of iteration
Time Efficient	N	Y	Recursion: needs more time for pop and push ops to stack memory, which makes it less time efficient
Easy to code?	N	Y	Recursion especially preferred in the cases we know that a problem can be divided into similar sub-problems
Safety?	N	Y	For big problem set, the stack might overflow

When to Use/Avoid Recursion?

When to use it?

- When we can easily breakdown a problem into similar subproblem
- When we are fine to extra overhead(both time and space) that comes with it
- When we need a quick solution instead of efficient one
- When we use memorization in recursion (*Dynamic Programming*)

When to avoid it ?

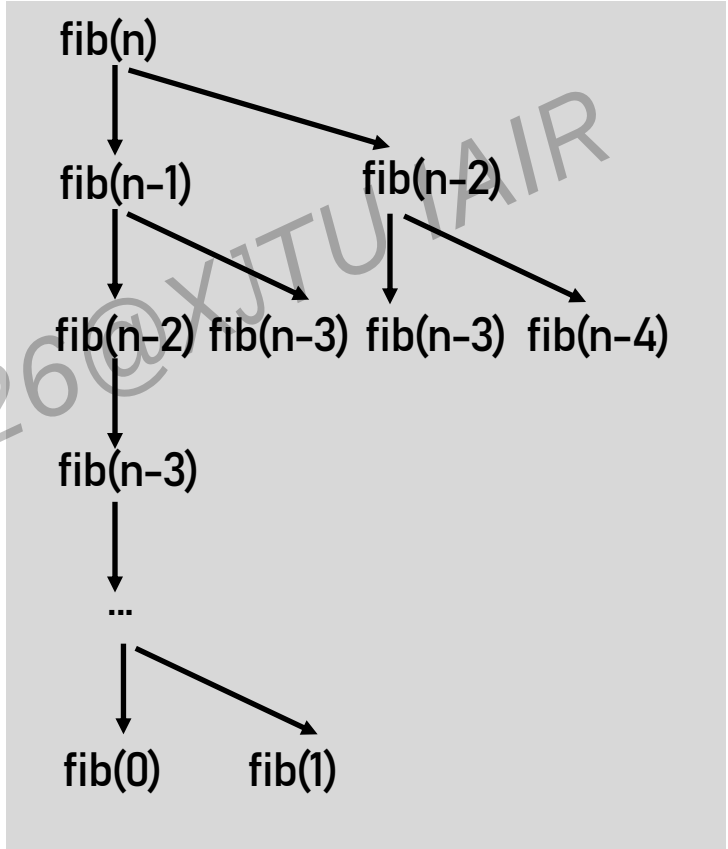
- If time and space complexity matters for us
- Recursion uses more memory (*Embedded Processor*)

Call Chain Example

```
fib(int n)
{
  if n<=2;
     return 1;
  else
     return fib(n-1)+fib(n-2);
}
```

Procedure fib (*) is recursive

Example Call Chain

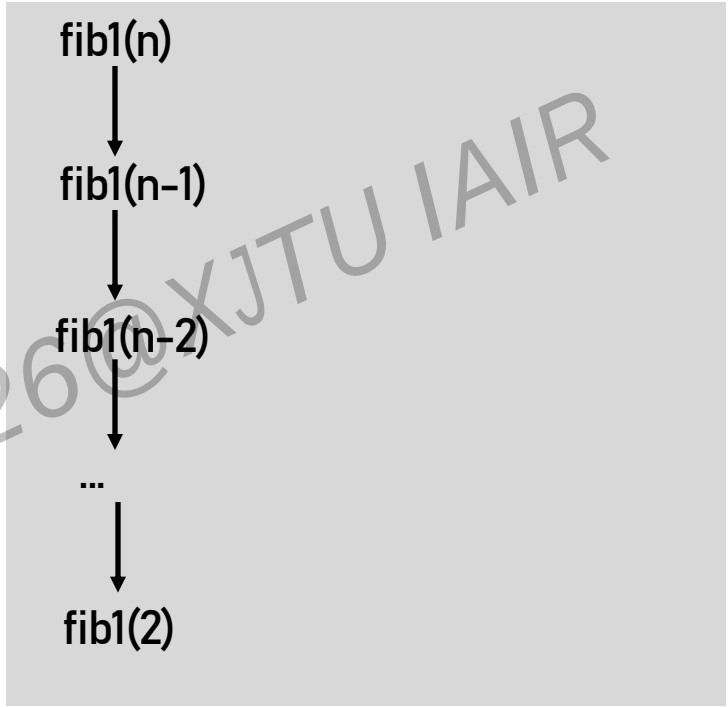


Call Chain Example

```
fib1(int n)
{
  if n<=2;
     return 1, 1;
  else
     a, b = fib1(n-1);
     return a+b, a;
}
```

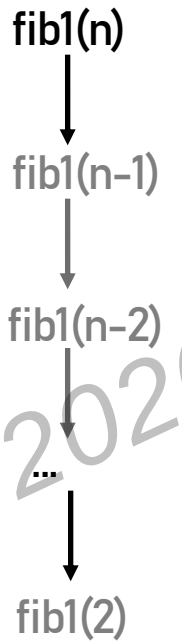
Procedure fib1 (*) is recursive

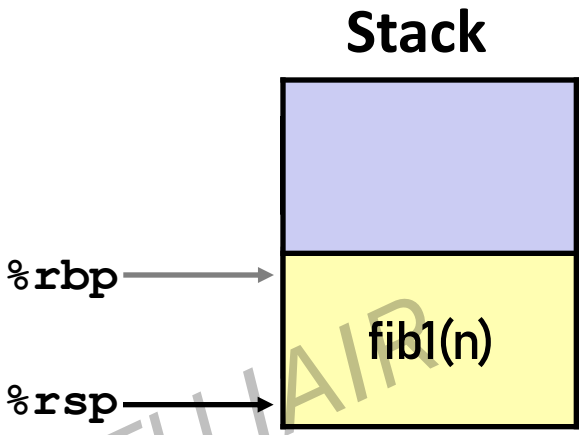
Example Call Chain



Example

```
fib1(int n)
{
  if n<=2;
     return 1, 1;
  else
     a, b = fib1(n-1);
     return a+b, a;
}
```





Example

```
fib1(int n)
{
  if
  el
}
```

```
fib1(int n)
{
  if n<=2;
    return 1, 1;
  else
    a, b = fib1(n-1);
    return a+b, a;
}
```

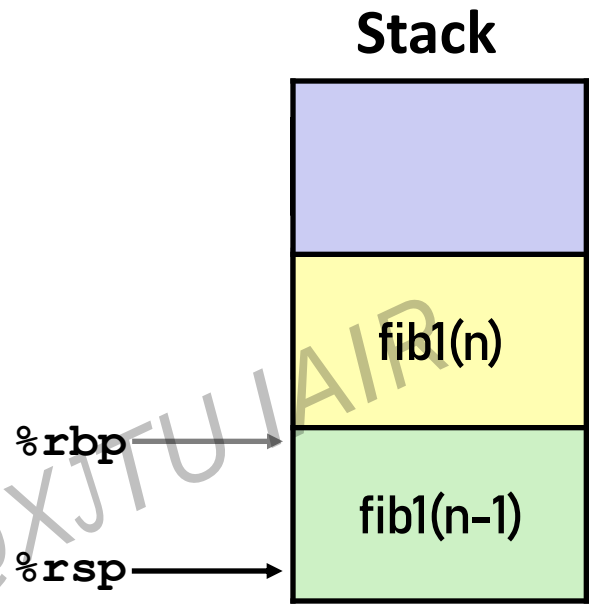
fib1(n)

fib1(n-1)

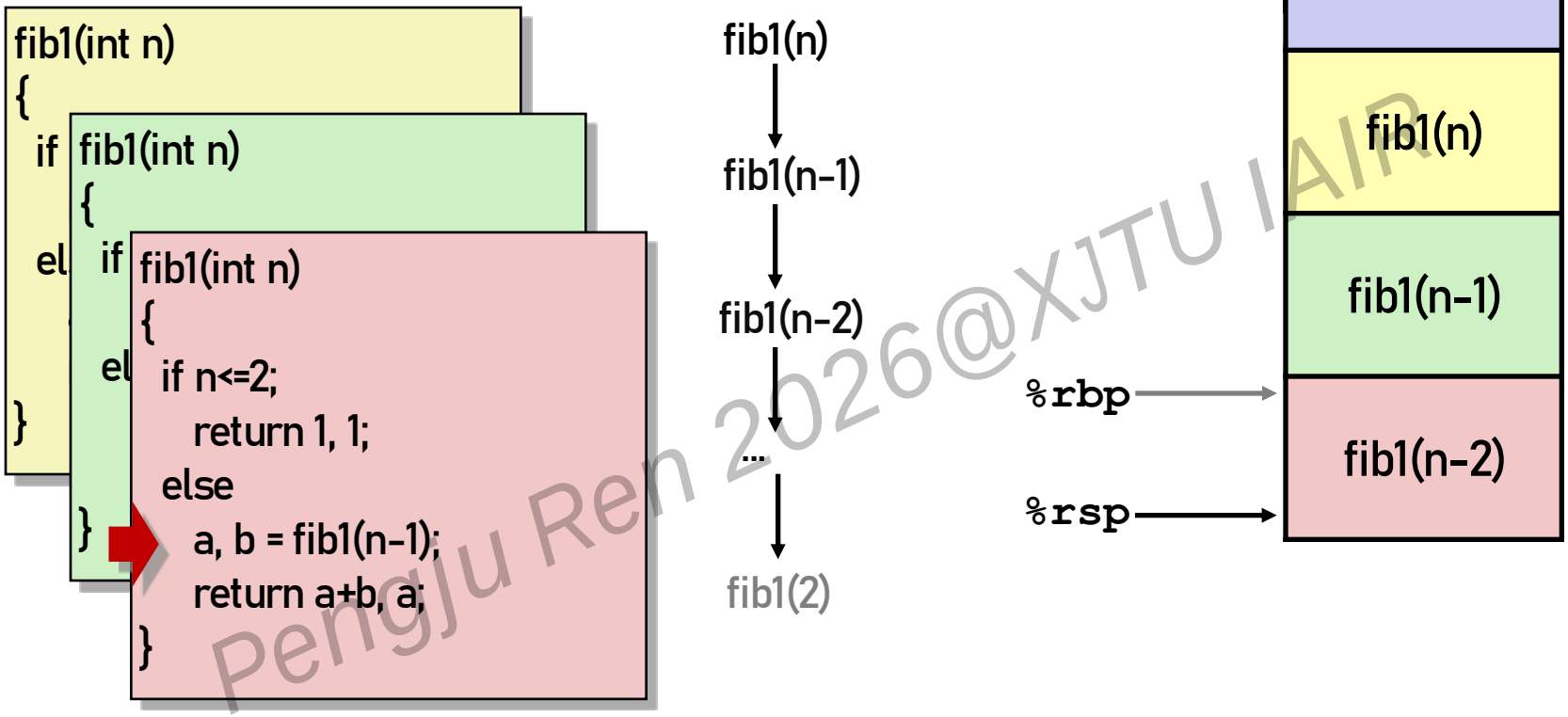
fib1(n-2)

...

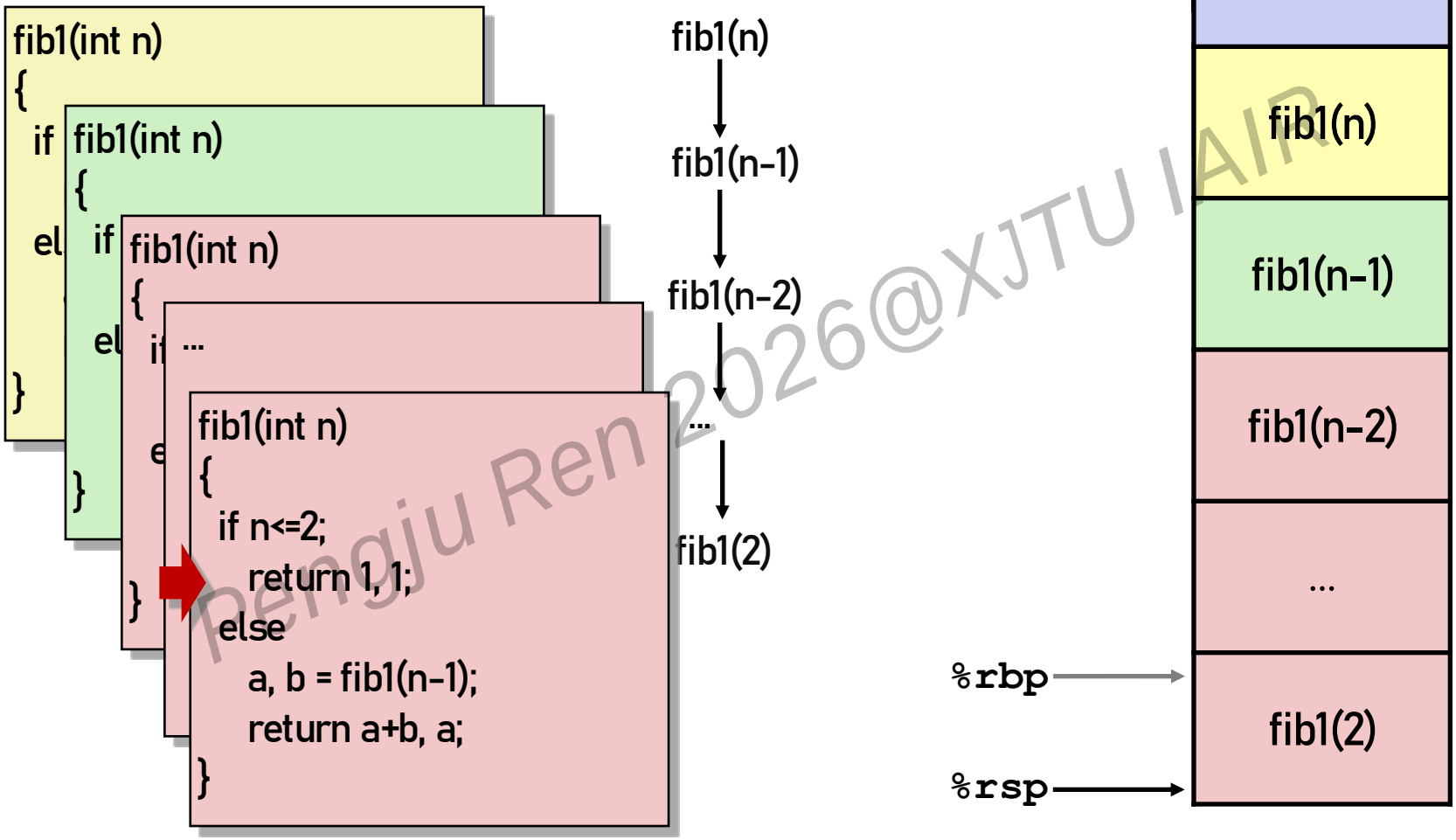
fib1(2)



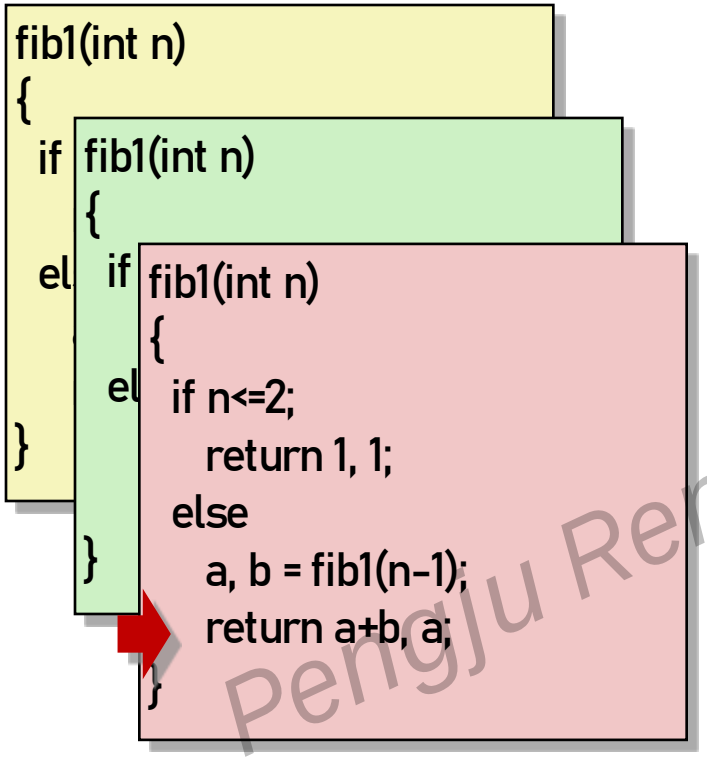
Example

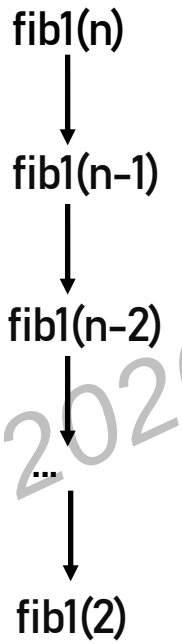


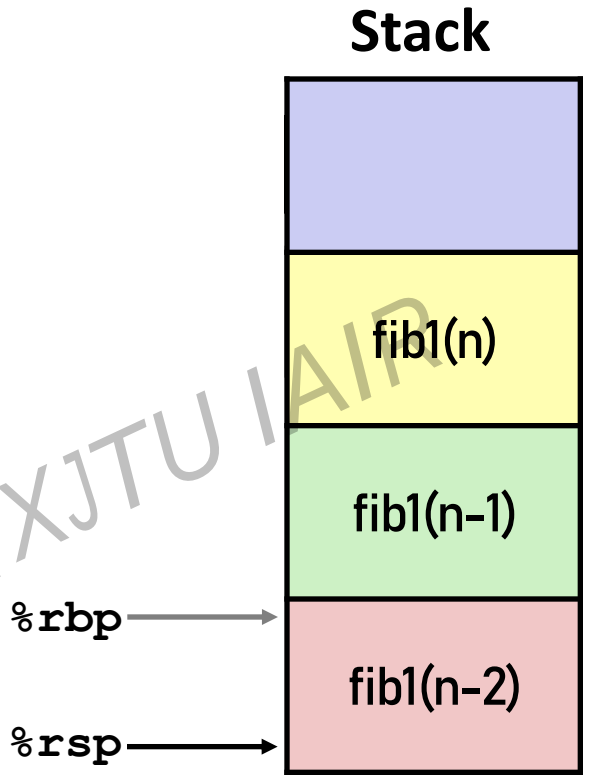
Example



Example



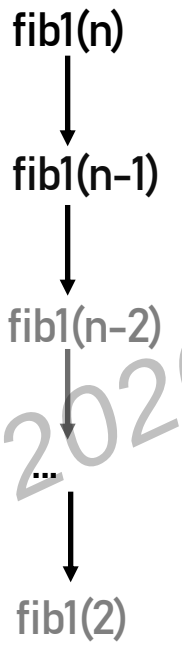


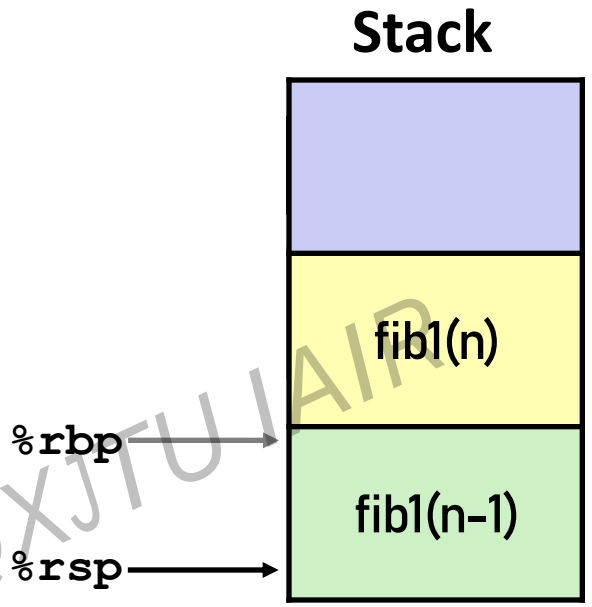


Example

```
fib1(int n)
{
  if
  el
}
```

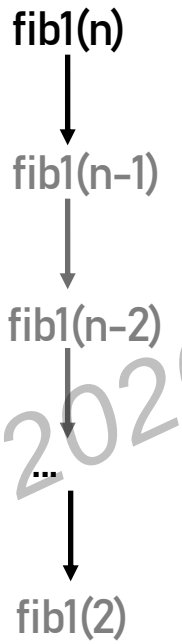
```
fib1(int n)
{
  if n<=2;
    return 1, 1;
  else
    a, b = fib1(n-1);
    return a+b, a;
}
```

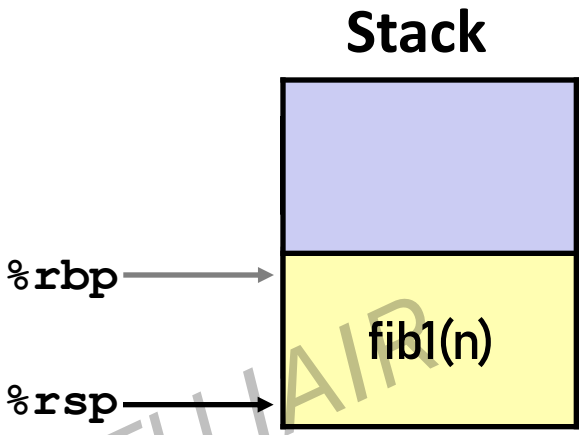




Example

```
fib1(int n)
{
  if n<=2;
     return 1, 1;
  else
     a, b = fib1(n-1);
     return a+b, a;
}
```





Mechanisms in Procedures/Programs

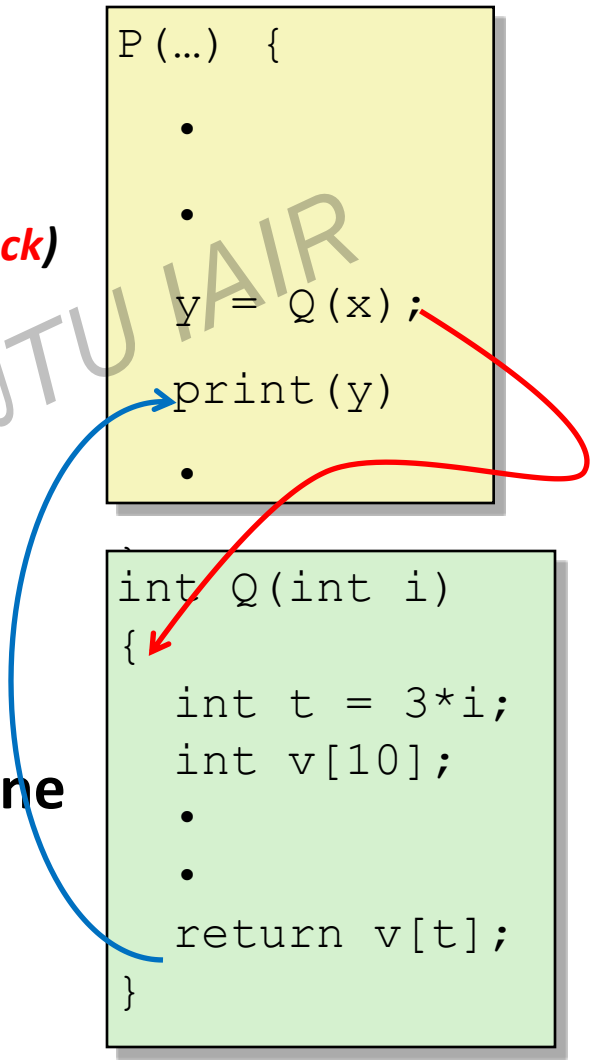
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Mechanisms in Procedures/Programs

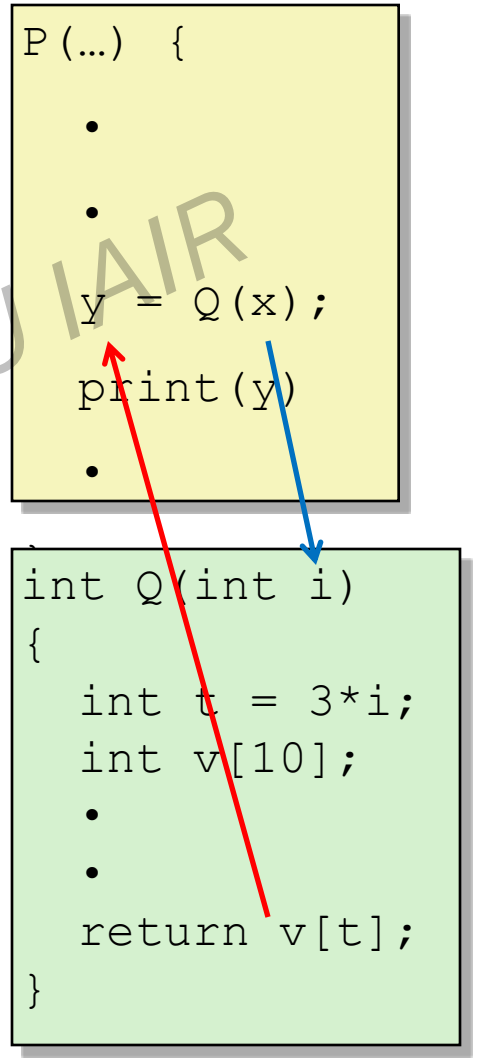
- **Passing control**
  - **To beginning of procedure code *(jump to label)***
  - **Back to return point *(jump to addr poped from stack)***
- **Passing data**
  - **Procedure arguments**
  - **Return value**
- **Memory management**
  - **Allocate during procedure execution**
  - **Deallocate upon return**
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**



Mechanisms in Procedures/Programs

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments *(specific regs)*
  - Return value *(specific regs)*
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

Only allocate stack space when needed



Mechanisms in Procedures/Programs

- **Passing control**
  - **To beginning of procedure code**
  - **Back to return point**
- **Passing data**
  - **Procedure arguments (Caller to Callee)**
  - **Return value (Callee to Caller)**
- **Memory management**
  - **Allocate during procedure execution**
  - **Deallocate upon return**
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

```
P(…) {
   •
   •
  y = Q(x);
  print(y)
   •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
   •
   •
  return v[t];
}
```

Procedure Control Flow

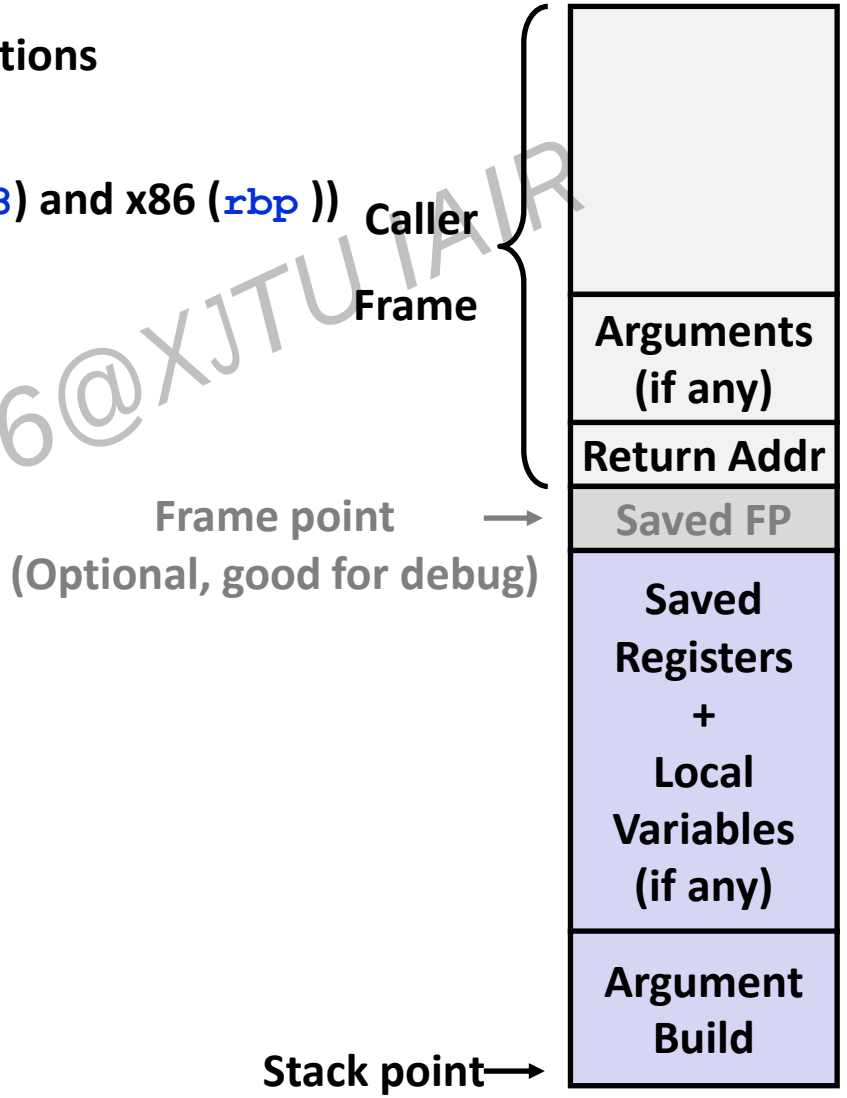
- Use stack to support procedure *call* and *return*
- Procedure call: `call label`
  - Push return address on stack
  - Jump to *label*
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address
- Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the Application Binary Interface (ABI).

Understanding Function Calls by Stack

Stack divided into frames
– Each frame stores locals and args to called functions

Stack pointer points to the top of the stack
Frame pointer points to caller's frame (RISC-V (`x8`) and x86 (`rbp`))

- **Calling a function**
  - **Caller**
    - **Pass arguments**
    - **Call and save return address**
  - **Callee**
    - **Save old frame pointer**
    - **Set frame pointer = stack pointer**
    - **Allocate stack space for local storage**
- **When returning**
  - **Callee**
    - **Pop local storage**
    - **Set stack pointer = frame pointer**
    - **Pop frame pointer**
    - **Pop return address and return**
  - **Caller**
    - **Pop arguments**



Summary

- **Array:** Contiguous memory for homogeneous elements
- **Linked-list:** Non-contiguous nodes connected by pointers
- **Recursion v.s Iteration:** Function calling itself to solve problems , ***termination condition*** 、***Recursive Call*** 、***Problem Reduction*** are 3-essentials.
- **Function Call Process:**
  - Each function call creates a stack frame (activation record)
  - Frame contains: parameters, local variables, return address
  - Recursion is a classic stack application

Homework 1

1: Using recursion to solve problems

1) return the Greatest Common Divisor of number n and m, *GCD(n,m)*
2) Converting a Decimal to Binary (Divide the number by 2)

- **Test data:**
  - Randomly generate 100 test cases for both 1) and 2)

Note：Euclidean Algorithm *GCD(n,m)=GCD(m, n%m)*

Submission requirements:

1: Description of the implementation approach
2: Performance test results and analysis
3: Difficulties encountered and solutions

Homework naming: HW1-Class X-stu ID-name, e.g. HW1-Class01-12345678-张三

Submission email: chengyuma@stu.xjtu.edu.cn
Submission ddl: 3.15 24：00