Lecture 9 Algorithm: Recursion

§1 Recursion

1. The concept of recursion

1° Recursion provides an elegant and powerful alternative for performing repetitive tasks.

2° Implementation: A function makes one or more calls to itself during execution.

3° Recursion is one of the central ideas of computer science.

EXAMPLE: THE FACTORIAL FUNCTION

- The factorial of a positive integer n, denoted n!, is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- Permutations of n items
  - the number of ways in which n distinct items can be arranged into a sequence

THE RECURSIVE DEFINITION OF FACTORIAL FUNCTION

- The factorial function can be naturally defined in a recursive way
  - for example, 5! = 5 ·(4 · 3 · 2 · 1) = 5 · 4!
- More generally, for a positive integer n, we can define n! to be n · (n−1)!
- Therefore, the recursive definition of factorial function is:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

SOLUTION

THE RECURSIVE DEFINITION

```
def facFunc(n):
    if n<0:
        print('Invalid input.')
        return None
    elif n == 0:
        return 1
    else:
        return n*facFunc(n-1)
```

First, a recursive definition contains one or more base cases, which are defined non-recursively in terms of fixed quantities

Second, it also contains one or more recursive cases, which are defined by appealing to the definition of the function being defined

2. How python implements recursion

1° In Python, each time a function (recursive or otherwise) is called, a structure known as an activation record or frame

is created to store information about the progress of that invocation of the function.

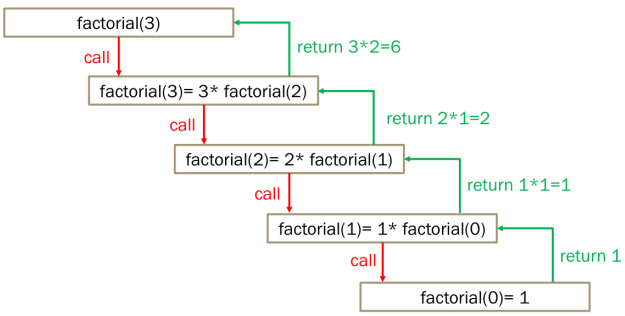
2° This activation record stores the function call's parameters and local variables

3° When the execution of a function leads to a nested function call

① The execution of the former call is suspended.

② its activation record stores the place in the source code at which the flow of control should continue upon return of the nested call.

THE RECURSIVE TRACE



3. Binary search

- Sequential Search Algorithm
  - use a loop to examine every element, until either finding the target or exhausting the data set
  - for unsorted sequence
- How about sorted sequence?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- Binary Search
  - A classic and very useful recursive algorithm
  - can be used to efficiently locate a target value within a sorted sequence of n elements

THE STRATEGY OF BINARY SEARCH

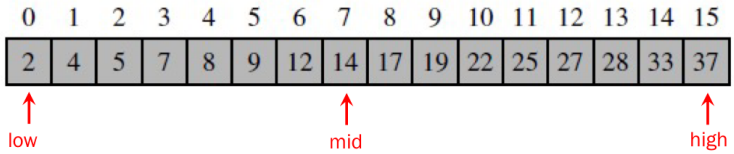
- For any index j, we know that all the values stored at indices 0, . . . , j−1 are less than or equal to the value at index j, and all the values stored at indices j+1, . . . ,n−1 are greater than or equal to that at index j

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

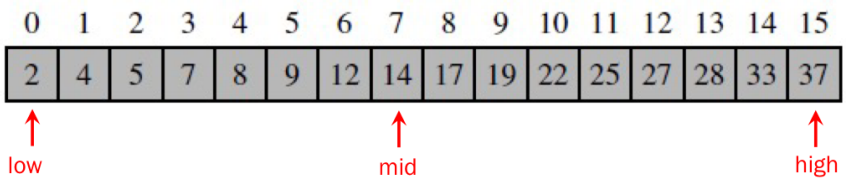
j

We call an element of the sequence a candidate if, at the current stage of the search, we cannot rule out that this item matches the target

- The algorithm maintains two parameters, low and high, such that all the candidate entries have index at least low and at most high
- Initially, low = 0 and high = n−1. We then compare the target value to the median candidate, that is, the item data[mid] with index mid = ⌊(low+high)/2⌋

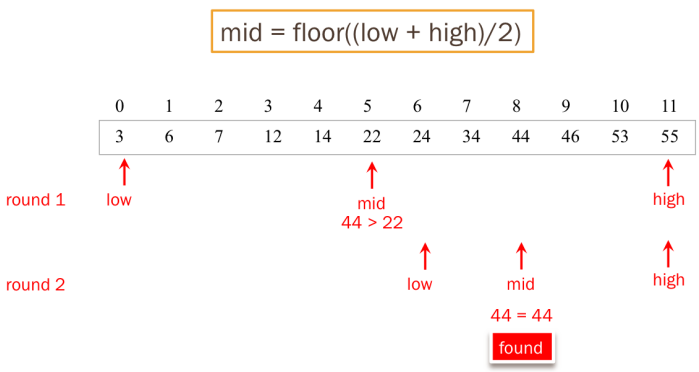


- If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully
  - If target < data[mid], then we recur on the first half of the sequence: from low to mid−1
  - If target > data[mid], then we recur on the second half of the sequence: from mid+1 to high
- But when should the algorithm stop?
- A common mistake for recursive algorithm is infinite self-recursion.
- The condition that we should stop:
  - We find that low > high



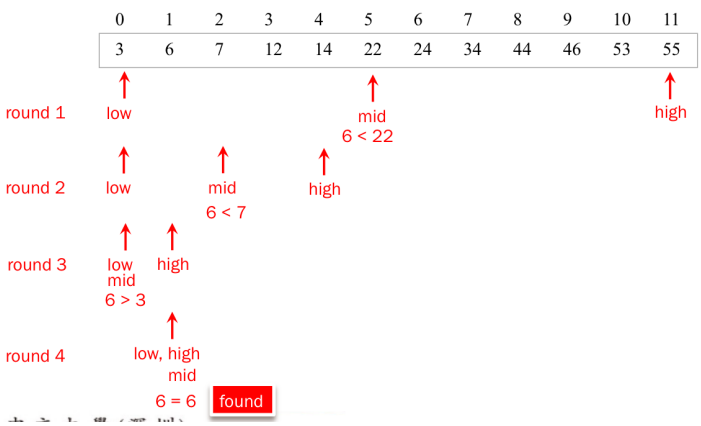
BINARY SEARCH: EXAMPLE

Searching for 44 from following sorted array



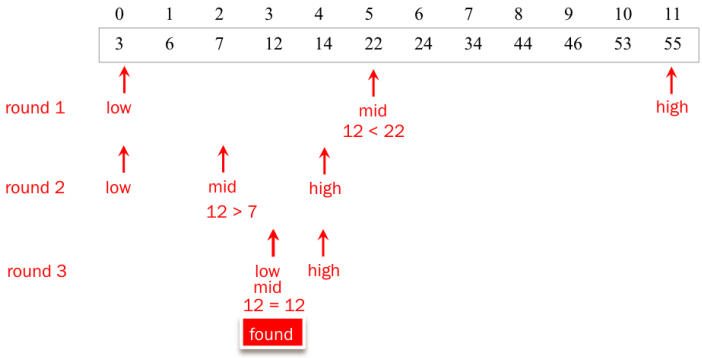
BINARY SEARCH: EXAMPLE

Searching for 6 from following sorted array

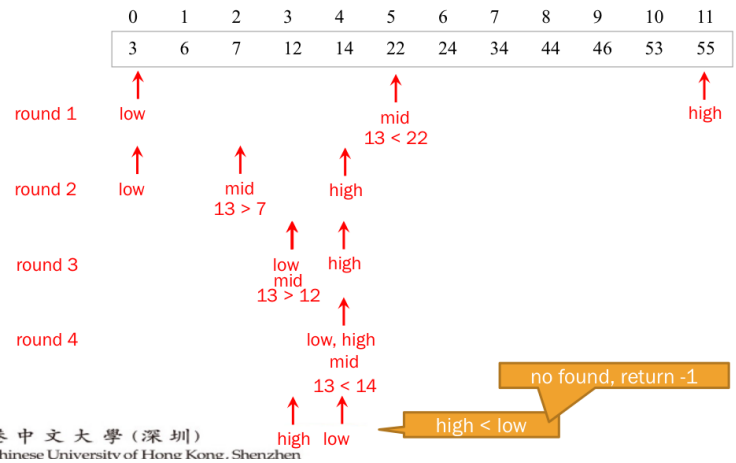


BINARY SEARCH: EXAMPLE

Searching for 12 from following sorted array



Searching for 13 from following sorted array



香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SOLUTION

```
def binarySearch(data, target, low, high):
    if low>high:
        print('Cannot find the target number!')
        return False
    else:
        mid = (low+high)//2
        if target==data[mid]:
            print('The target number is at position', mid)
            return True
        elif target<data[mid]:
            return binarySearch(data, target, low, mid-1)
        else:
            return binarySearch(data, target, mid+1, high)

def main():
    data = [1, 3, 5, 6, 16, 78, 100, 135, 900]
    target = 16
    binarySearch(data, target, 0, len(data)-1)
```

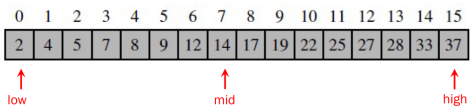
In python 3.0, / means integer division, // means floating division

The binary search algorithm runs in $O(\log n)$ time for a sorted sequence with n elements.

PROOF

Justification: To prove this claim, a crucial fact is that with each recursive call the number of candidate entries still to be searched is given by the value

$$\text{high} - \text{low} + 1.$$



Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

or

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}.$$

PROOF

Initially, the number of candidates is n; after the first call in a binary search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the j^{th} call in a binary search, the number of candidate entries remaining is at most $n/2^j$.

In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

$$\frac{n}{2^r} < 1$$

number of candidates

In other words (recalling that we omit a logarithm's base when it is 2), $r > \log n$. Thus, we have

$$r = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time. ■

§2 Linear Recursion

1. Linear recursion

If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is known as linear recursion.

```
def facFunc(n):

    if n<0:
        print('Invalid input.')
        return None
    elif n == 0:
        return 1
    else:
        return n*facFunc(n-1)
```

Finding the factorial and binary search are both linear recursive algorithms

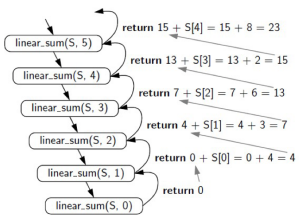
```
def binarySearch(data, target, low, high):
    if low>high:
        print('Cannot find the target number!')
        return False
    else:
        mid = (low+high)//2
        if target==data[mid]:
            print('The target number is at position',mid)
            return True
        elif target<data[mid]:
            return binarySearch(data, target, low, mid-1)
        else:
            return binarySearch(data, target, mid+1, high)

def main():
    data = [1, 3, 5, 6, 16, 78, 100, 135, 900]
    target = 16
    binarySearch(data, target, 0, len(data)-1)
```

PRACTICE: SUM OF A LIST

- Given a list of numbers, write a program to calculate the sum of this list using recursion

SOLUTION:

```
def linearSum(L, n):
    if n==0:
        return 0
    else:
        return linearSum(L, n-1)+L[n-1]

def main():
    L = [4, 3, 6, 2, 8]
    print('The sum is:', linearSum(L, len(L)))
```



CASE STUDY: POWER FUNCTION

- Write a program to calculate the power function $f(x, n) = x^n$ using Recursion
- The time complexity of the program should be O(logn)

$$power(x, n) = \begin{cases} 1 & if\ n = 0 \\ x \cdot power(x, n-1) & if\ n \neq 0 \end{cases}$$

O(n)

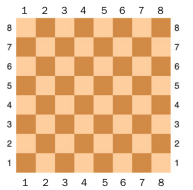
A RECURSIVE DEFINITION OF POWER FUNCTION WITH O(logn)

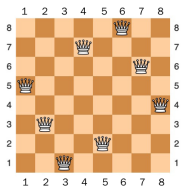
$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

```
def myPower(x, n):
    if n==0:
        return 1
    else:
        partial = myPower(x, n//2)
        result = partial * partial
        if n%2==1:
            result = result * x
        return result
```

CASE STUDY:

- How to solve the 8 queens problem using Recursion?





a solution requires that no two queens share the same row, column, or diagonal

SOLUTION:

```
15863724
16837425
17468253
17582463
24683175
25713864
25741863
26174835
26831475
27368514
27581463
...
Total Number = 92
```

```
def find_Queen(row):

    if row>7:
        global count
        count+=1
        print_queen()
        return

    for column in range(8):
        if check(row,column):
            Queen[row][column]=1
            find_Queen(row+1)
            Queen[row][column]=0
```

```
def check(row,column):

    # 检查行列
    for k in range(8):
        if Queen[k][column]==1:
            return False

    # 检查主对角线

    for i,j in zip(range(row-1,-1,-1),range(column-1,-1,-1)):
        if Queen[i][j]==1:
            return False

    # 检查副对角线
    for i,j in zip(range(row-1,-1,-1),range(column+1,8)):
        if Queen[i][j]==1:
            return False

    return True
```


§3 Multiple Recursion

1. Multiple recursion

When a function makes two or more recursive calls, we say that it uses multiple recursion.

EXAMPLE: DRAWING AN ENGLISH RULER

- We denote the length of the tick designating a whole inch as the *major tick length*.
- Between the marks for whole inches, the ruler contains a series of *minor ticks*, placed at intervals of 1/2 inch, 1/4 inch, and so on.
- As the size of the interval decreases by half, the tick length decreases by one

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
(a)
```

Inch: 2
Max Stick: 4

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
(b)
```

Inch: 1
Max Stick: 5

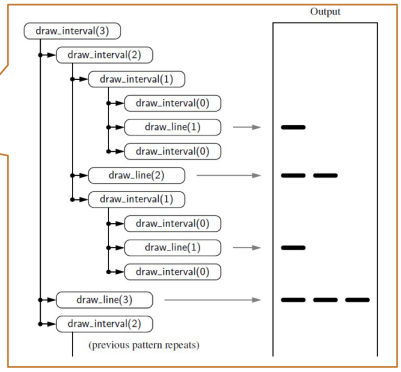
```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
(c)
```

Inch: 3
Max Stick: 3

- An interval with a central tick length $L \geq 1$ is composed of:
  - ✓ An interval with a central tick length $L-1$
  - ✓ A single tick of length L
  - ✓ An interval with a central tick length $L-1$

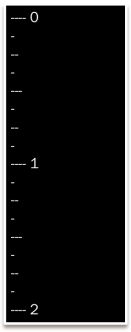
SOLUTION

```
def draw_line(tickLen, tickLabel=''):
    line = '-'*tickLen
    if tickLabel:
        line+=' '+tickLabel
    print(line)

def draw_interval(centerLen):
    if centerLen>0:
        draw_interval(centerLen-1)
        draw_line(centerLen)
        draw_interval(centerLen-1)

def draw_ruler(numInch, majorLen):
    draw_line(majorLen,'0')

    for j in range(1, 1+numInch):
        draw_interval(majorLen-1)
        draw_line(majorLen, str(j))
```



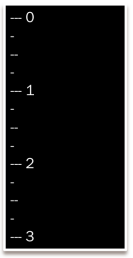
香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

draw_ruler(2, 4)



draw_ruler(1, 5)

draw_ruler(3, 3)



PRACTICE: BINARY SUM

- Write a function binarySum() to calculate the sum of a list of numbers.
- Inside binarySum() two recursive calls should be made

SOLUTION:

```
def binarySum(L, start, stop):
    if start>=stop:
        return 0
    elif start==stop - 1:
        return L[start]
    else:
        mid = (start+stop)//2
        return binarySum(L,start,mid)+binarySum(L,mid,stop)

def main():
    L = [1, 2, 3, 4, 5, 6, 7]
    print(binarySum(L, 0, len(L)))
```

FIBONACCI NUMBERS

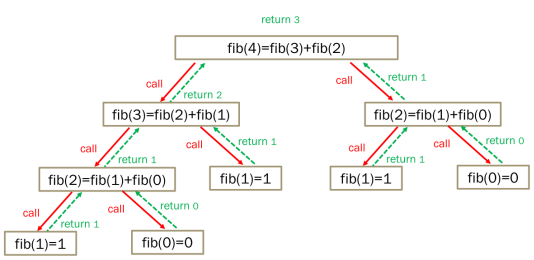
- In mathematics, the Fibonacci numbers, commonly denoted Fn form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- Recursive Approach:

F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)

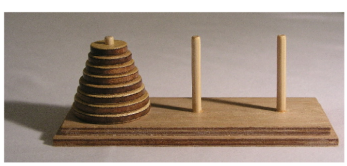
FIBONACCI NUMBERS



TOWER OF HANOI

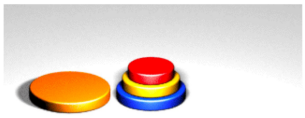
To move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

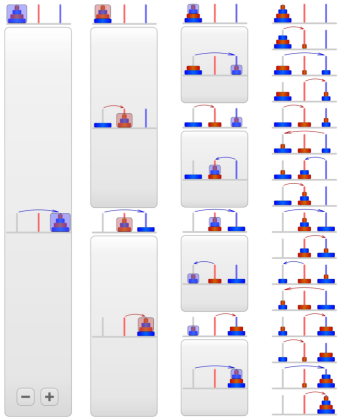


SOLVING TOWER OF HANOI

What is the base case?



香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen



TOWER OF HANOI

```
def hanoi(from_rod, to_rod, help_rod, n):
      if n==1:
            print("move from",from_rod,"to",to_rod)
      else:
            hanoi(from_rod, help_rod, to_rod, n-1)
            print("move from",from_rod,"to",to_rod)
            hanoi(help_rod, to_rod, from_rod, n-1)

hanoi('left','right','middle',5)
```