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

Introduction to Computer Engineering: Programming Applications

Lecture 8
Data Structure and Algorithm

Prof. Junhua Zhao
School of Science and Engineering

Data structure and algorithm

- A data structure is a systematic way of organizing and accessing data

- An algorithm is a step-by-step procedure for performing some task in a finite amount of time.

Why study data structure and algorithm?

- Important for all other branches of computer science
- Plays a key role in modern technological innovation
- Moore's law predicts that the density of transistors in integrated circuits would continue to double every 1 to 2 years
- However, in many areas, performance gains due to the improvements in algorithms have greatly exceeded even the dramatic performance gains due to increased processor speed

Why study data structure and algorithm?

- Provide novel “lens” on processes outside of computer science and technology, such as physics, economic markets, evolution

- Challenging (good for your brain!!) and funny

Example: Integer Multiplication

- Inputs: two n-digits number x and y
- Output: the product of x and y
- Primitive operations: add or multiply 2 single digit numbers

The algorithm designer’s mantra

- “Perhaps the most important principle for the good algorithm designer is to refuse to be content”

Aho, Hopcroft, and Ullman*, The Design and Analysis of Computer Algorithms*, 1974

How do we define a “good” algorithm?

- The primary analysis of algorithms involves characterizing the running times and space usage of algorithms and data structure operations
- Running time is a natural measure of “goodness,” since time is a precious resource—computer solutions should run as fast as possible
- Space usage is another major issue to consider when we design an algorithm, since we only have limited storage spaces

Measuring the running time experimentally

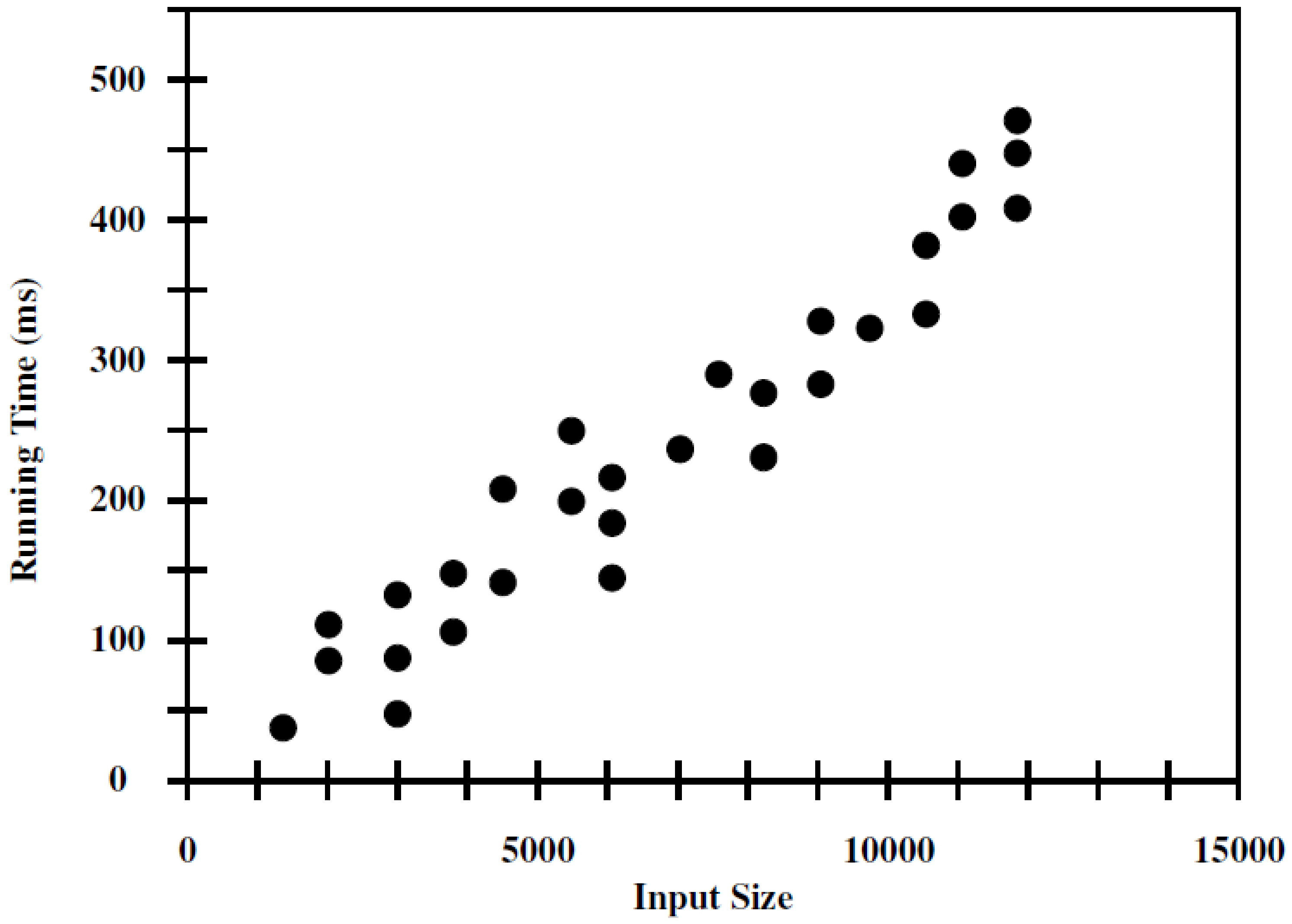
```
from time import time

startTime = time()

for i in range(1,20000):
    if i%10 == 0:
        print(i)

endTime = time()

print(’The time elapsed is:’,endTime - startTime,’seconds’)
```

Visualize the running time

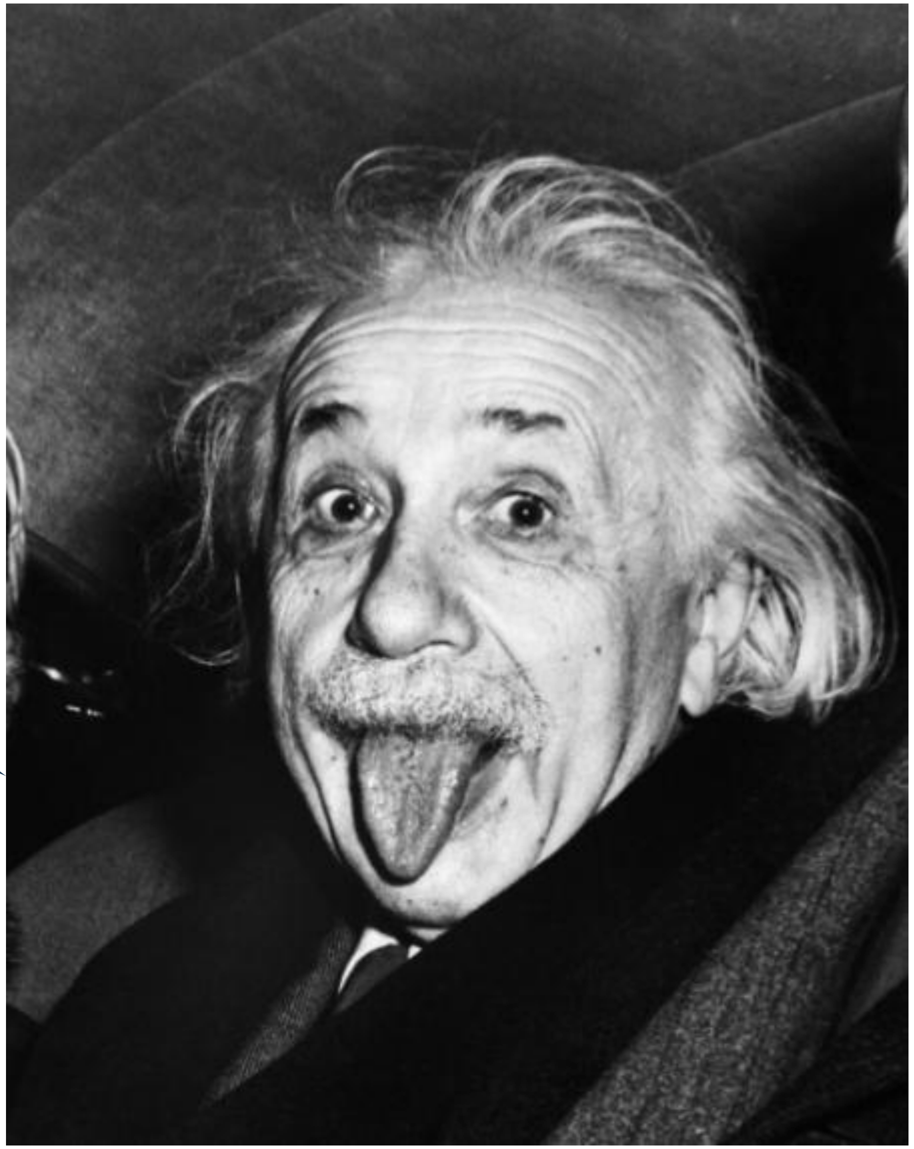
- Running time and space usage are dependent on the size of the input

- Perform independent experiments on many different test inputs of various sizes

- Visualize the results by plotting the performance of each run of the algorithm as a point



When you sit with a beautiful girl for two hours, it feels like you've been sitting for a minute; but if you sit by a hot fire, even for just one minute, it feels like you've been sitting for two hours. This is the theory of relativity.



Challenges of experimental analysis

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments

- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important)

- An algorithm must be fully implemented in order to execute it to study its running time experimentally

Principle of algorithm analysis 1: Counting primitive operations

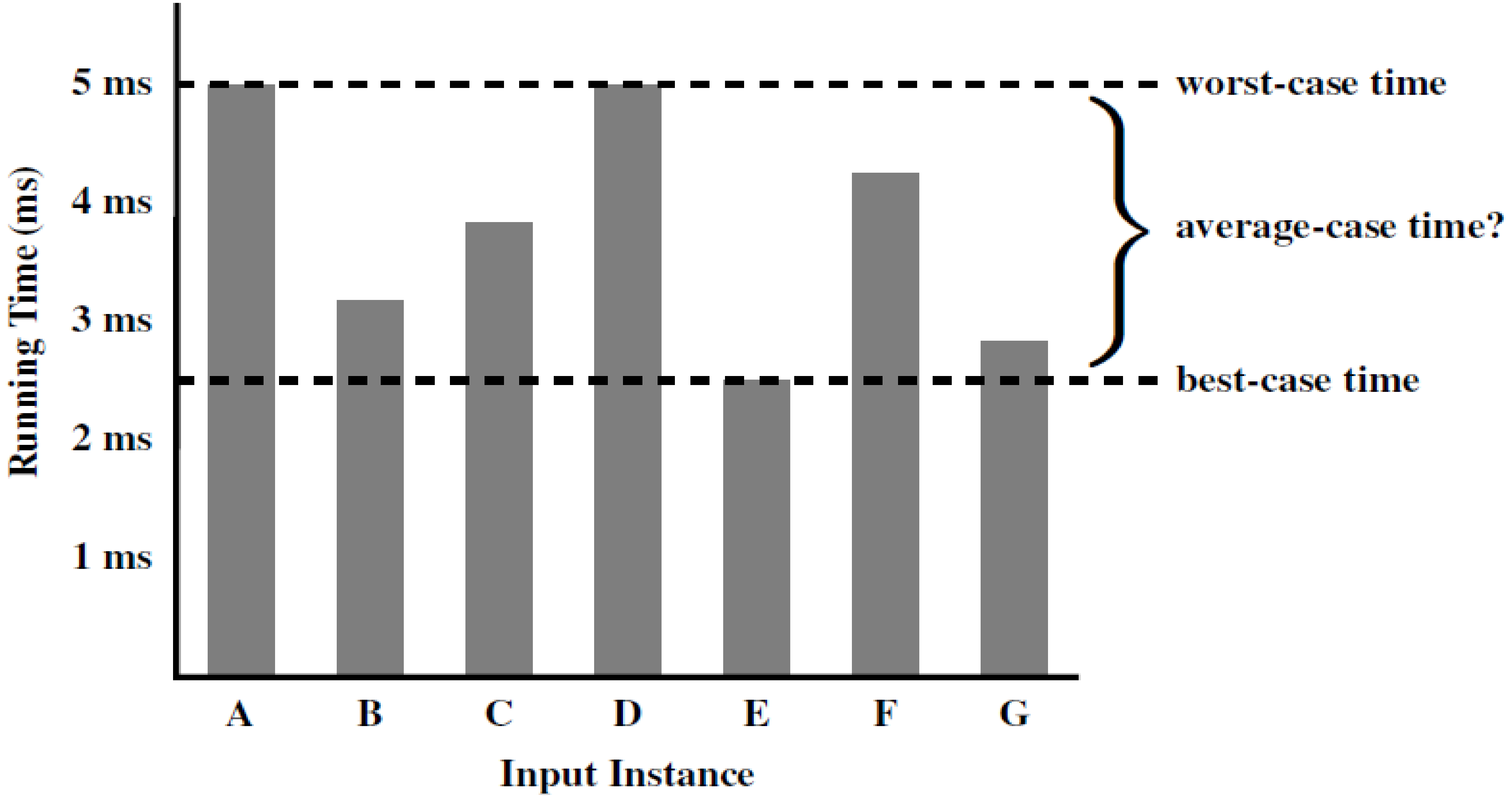
- To analyse the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm

- We define a set of primitive operations such as the following:
  - ✓ Assigning an variable to an object
  - ✓ Determining the object associated with an variable
  - ✓ Performing an arithmetic operation (for example, adding two numbers)
  - ✓ Comparing two numbers
  - ✓ Accessing a single element of a Python list by index
  - ✓ Calling a function (excluding operations executed within the function)
  - ✓ Returning from a function.

Principle of algorithm analysis 2: Measuring Operations as a Function of Input Size

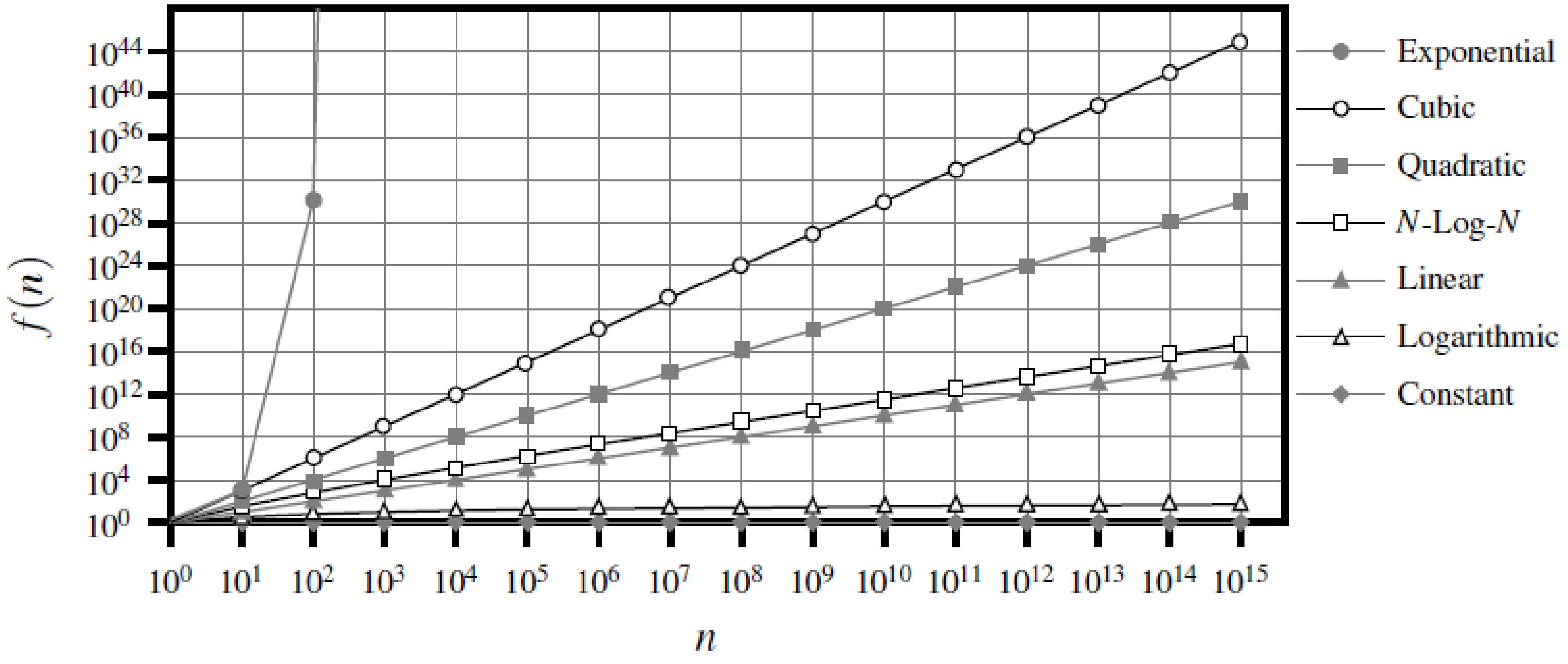
- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function f (n) that characterizes the number of primitive operations that are performed as a function of the problem size n

Principle of algorithm analysis 3: Focusing on the Worst-Case Input



The 7 functions used in algorithm analysis

- We may use the following 7 functions to measure the time complexity of an algorithm: constant, logarithm, linear, N-log-N, quadratic, cubic and other polynomials, exponential



Asymptotic Analysis

- In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a “big-picture” approach
- Vocabulary for the analysis and design of algorithms
- “Sweet spot” for high-level reasoning about algorithms
- Coarse enough to supress unnecessary details, e.g. architecture/language/compiler…
- Sharp enough to make meaningful comparisons between algorithms

The big Oh notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
- We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n), \text{ for } n \geq n_0$$

- This definition is often referred to as the "big-Oh" notation
- Example: The function $8n+5$ is $O(n)$.

The big Oh notation

- The big-Oh notation allows us to say that a function f(n) is “less than or equal to” another function g(n) up to a constant factor and in the asymptotic sense as n grows toward infinity

- The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the “size” of the problem

Some Properties of the Big-Oh Notation

- The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth

- Example: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is O(n^4)
- Example: 2^{n+2} is O(2^n)
- Example: $2n + 100logn$ is O(n)

- In general, we should use the big-Oh notation to characterize a function as closely as possible

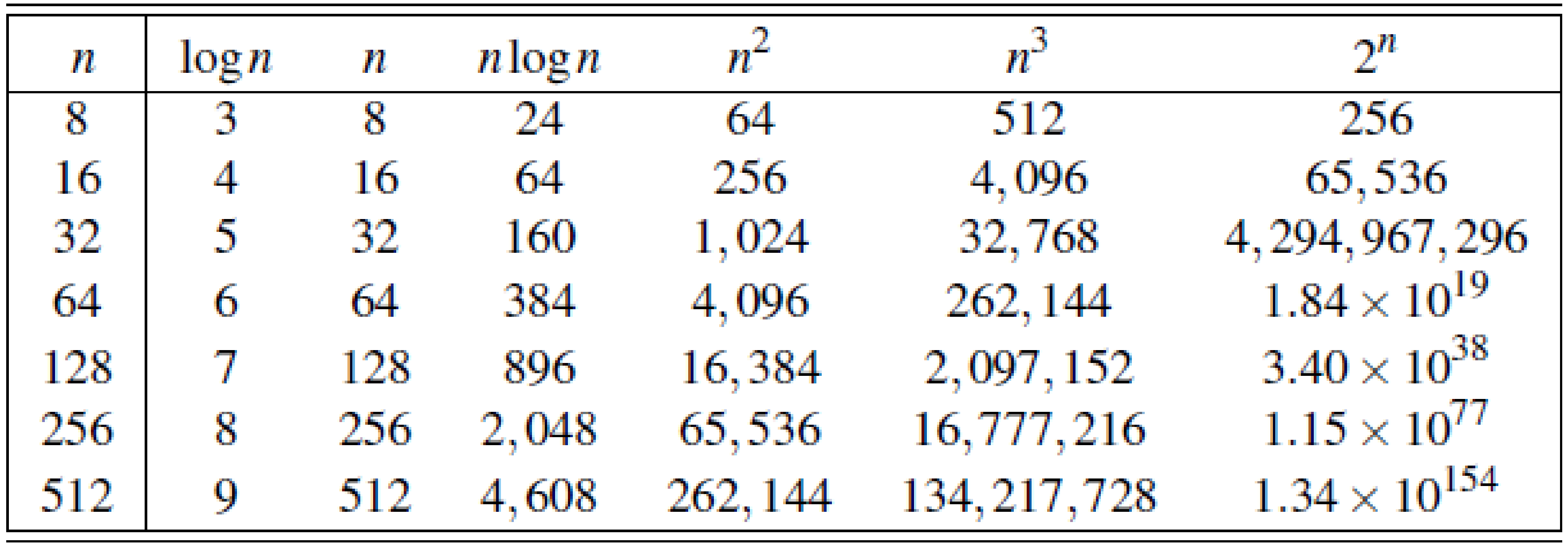
Comparative analysis

Question: Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of O(n), and an algorithm B, which has a running time of O(n^2). Which algorithm is better?

Answer: Algorithm A is asymptotically better than algorithm B

Comparative analysis

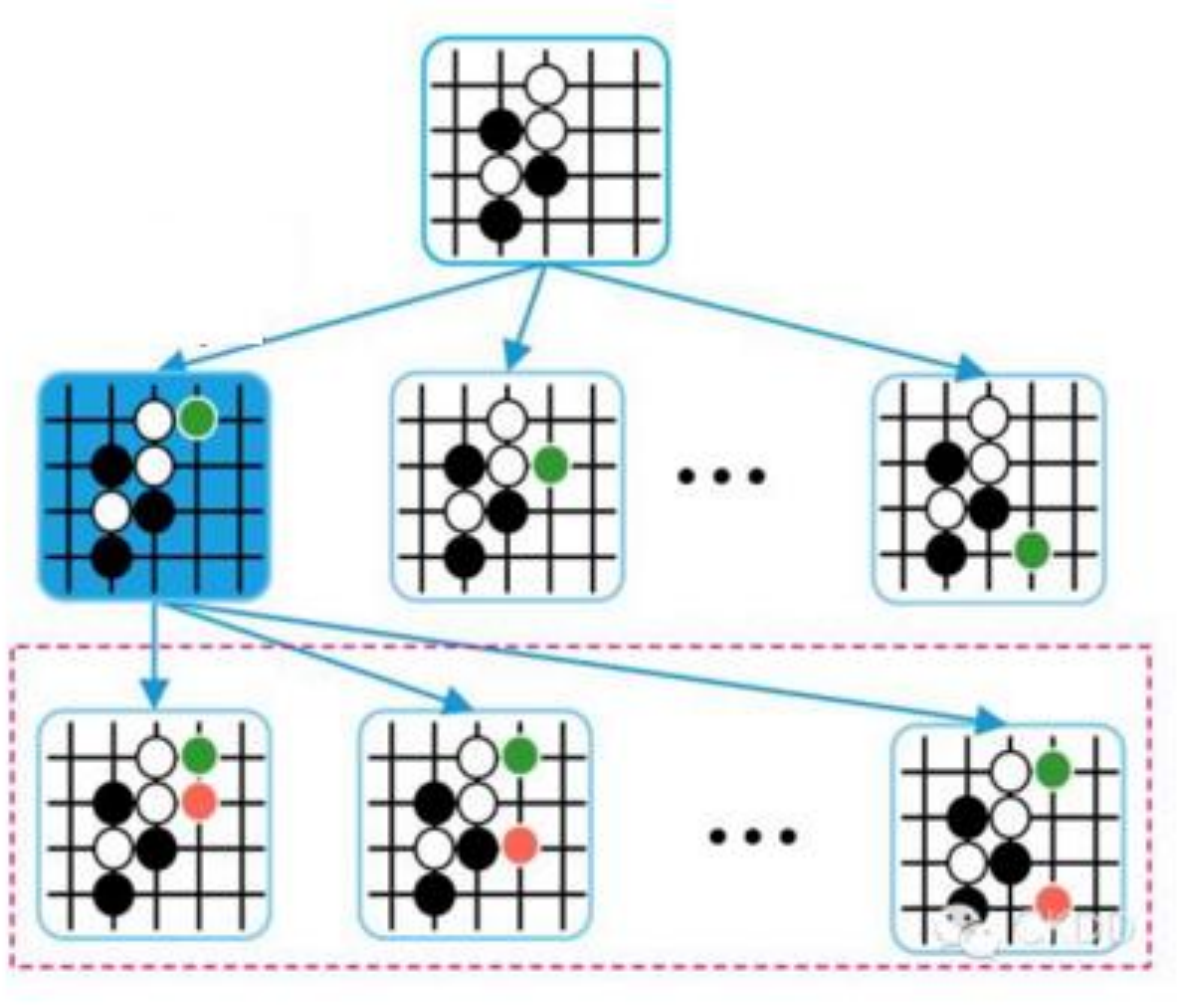
- We can use the big-Oh notation to order classes of functions by asymptotic growth rate
- Our seven functions are ordered by increasing growth rate in the following sequence



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Why AlghaGo is a remarkable achievement?

- If we use brutal-force to search the best move in Go, the time complexity is at the order of $O(10^n)$
- The search space is even larger than the number of atoms in the universe!!!



The line of tractability

- To differentiate efficient and inefficient algorithms, the general line is between polynomial time algorithms and exponential time algorithms

- The distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability

Example: Finding the smallest number in a list

```
smallest_so_far = None
print('Before',smallest_so_far)

for num in [9,39,21,98,4,5,100,65]:
    if smallest_so_far == None:
        smallest_so_far = num
    elif num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far,num)

print('After',smallest_so_far)
```

- What is the time complexity of this algorithm?

Recursion

- Recursion is a technique by which a function makes one or more calls to itself during execution
- Recursion provides an elegant and powerful alternative for performing repetitive tasks
- Recursion is an important technique in the study of data structures and algorithms

Inception

Example: The factorial function

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

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

- The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items

The recursive definition

- 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

The recursive definition of factorial function

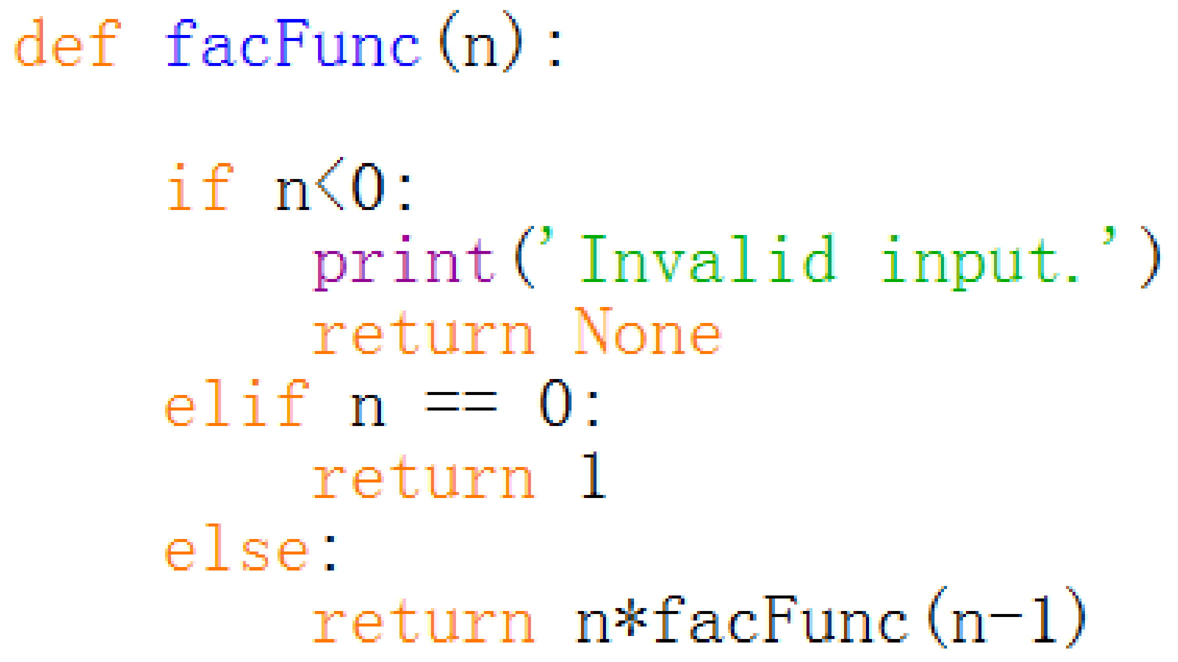
- The factorial function can be naturally defined in a recursive way, for example, 5! = 5 ·(4 · 3 · 2 · 1) = 5 · 4!

- 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

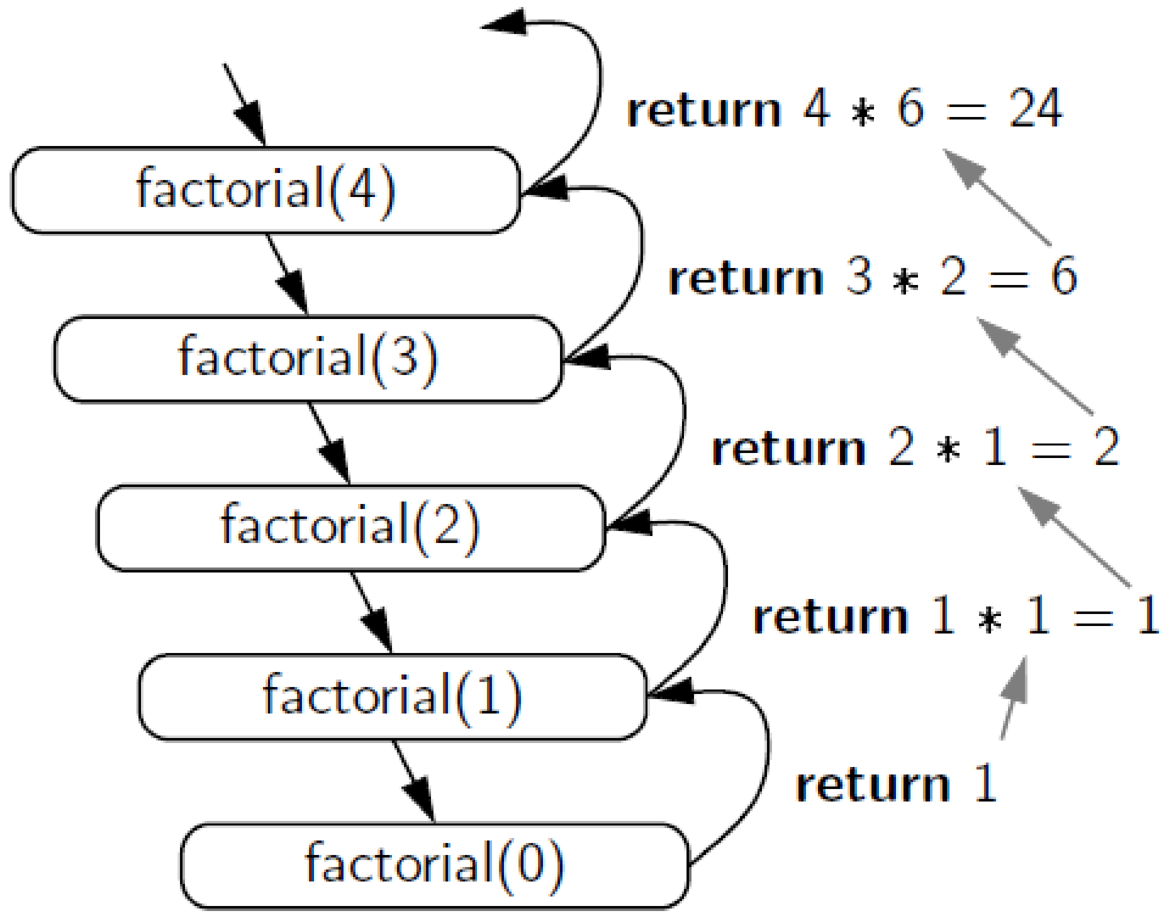


```
def facFunc(n):

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

How Python implements recursion

- 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
- This activation record stores the function call's parameters and local variables
- When the execution of a function leads to a nested function call, the execution of the former call is suspended and 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



Example: Binary search

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

- When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set; This is known as the sequential search algorithm

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

- When the sequence is sorted and indexable, binary search is a much more efficient algorithm

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

The strategy of binary search

- 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

$$\text{mid} = \lfloor(\text{low}+\text{high})/2\rfloor$$

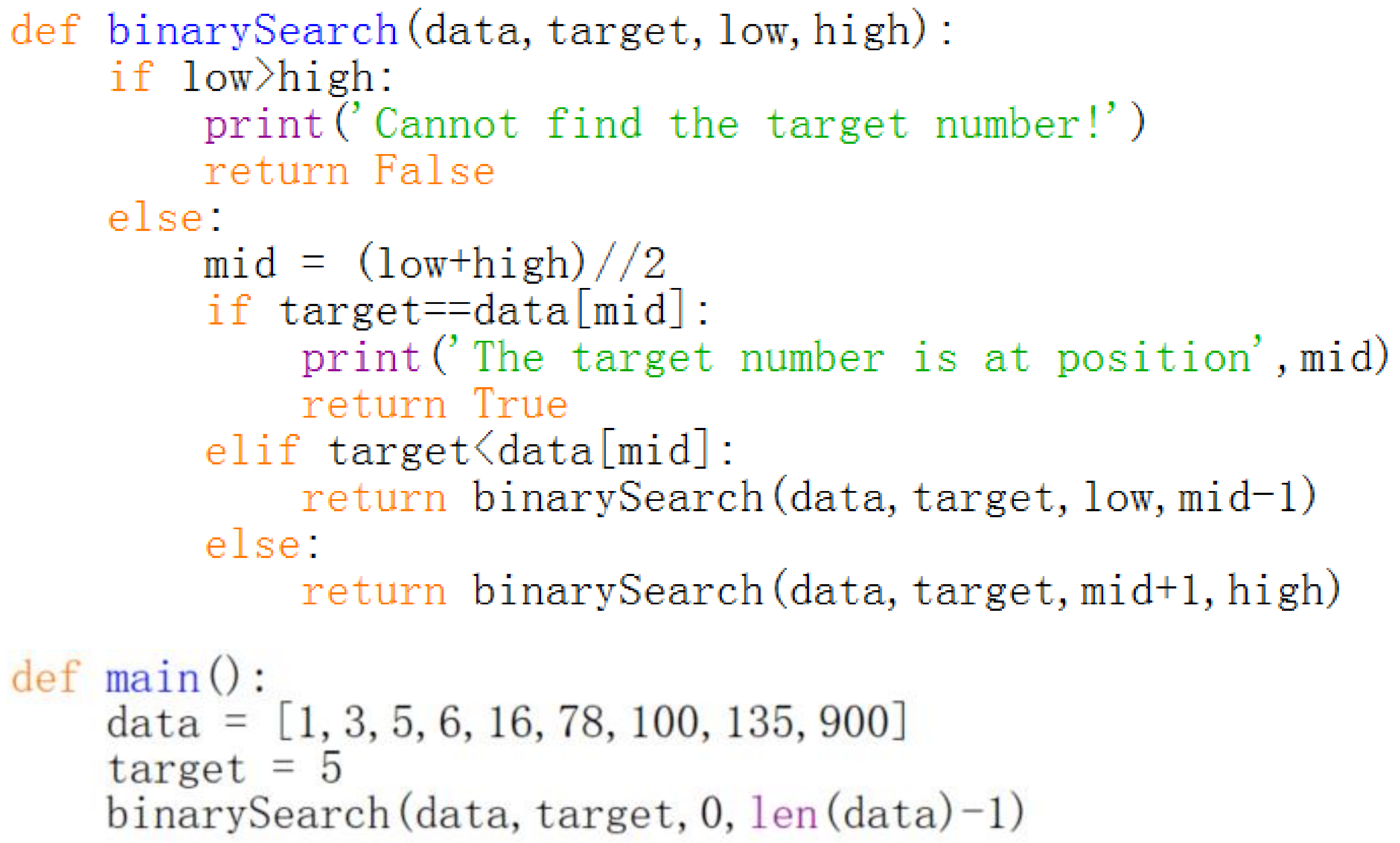
The strategy of binary search

- 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, that is, on the interval of indices from low to mid−1

- If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high

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 = 5
    binarySearch(data, target, 0, len(data)-1)
```

Time complexity of binary search

Proposition: The binary search algorithm runs in O(logn) time for a sorted sequence with n elements

Why ?

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

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

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

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

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. ■

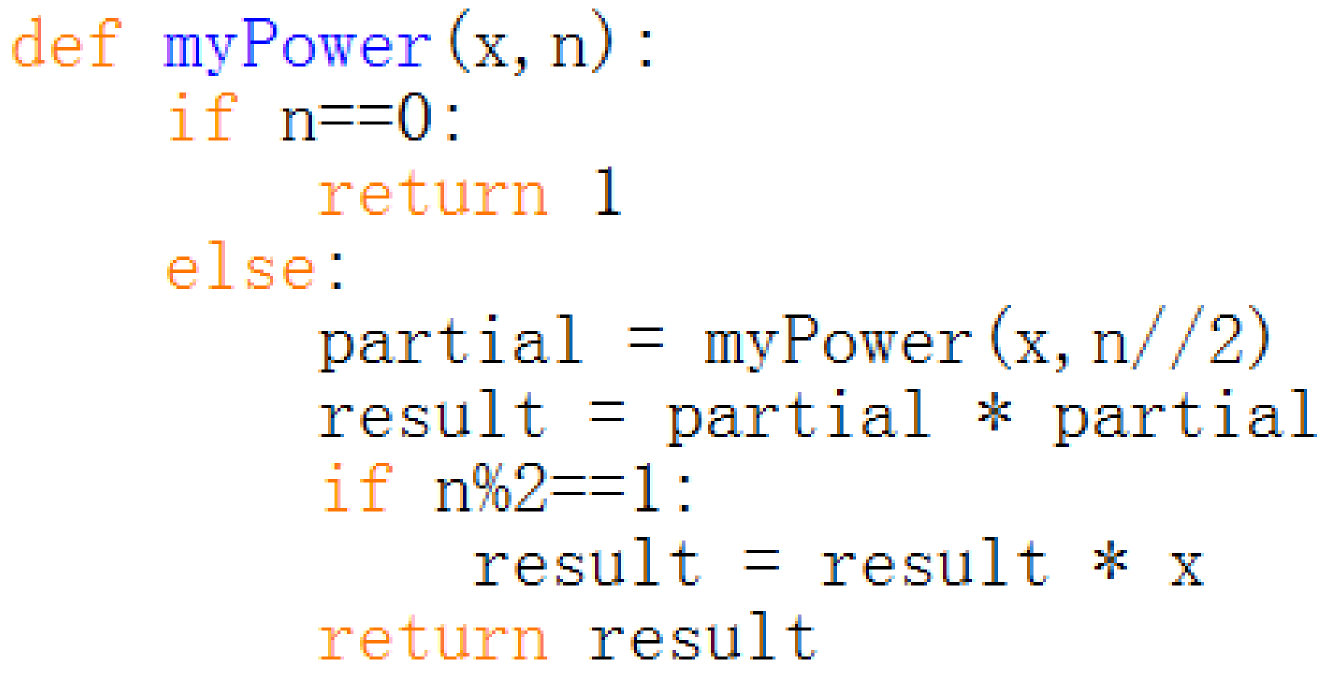
Practice: Power function

- 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)

A better recursive definition of power function

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

Solution:



```
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
```