STATS 507
Data Analysis in Python

Week5-1: Algorithm thinking

Dr. Xian Zhang

Recap: Iterator as an object

An iterator is an object that represents a "data stream"

Supports method `__next__()`:

returns next element of the stream/sequence

raises `StopIteration` error when there are no more elements left

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
    def __iter__(self):
        return(self)
s = Squares()
for x in s:
    print(x)
```

Iterable means that an object has the `__iter__()` method, which returns an iterator. So `__iter__()` returns a new object that supports `__next__()`.

`__next__()` is the important point, here. It returns a value, the next square.

Now `Squares` supports `__iter__()` (it just returns itself!), so Python allows us to iterate over it.

Recap: Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

```
next(h)
```

```
1.0
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

If/when we run out of `yield` statements (i.e., because we reach the end of the definition block), the generator returns a `StopIteration` error, as required of an iterator (not shown here).

Recap: Handling exceptions

Exception handler in Python

```
try:
    # do some potentially
    # problematic code
except:
    # do something to
    # handle the problem
```

```
if <all potentially problematic code succeeds>:
    # great, all that code
    # just ran fine!
else:
    # do something to
    # handle the problem
```

Besides `except` blocks

- `else`
  - Body will always be executed **when** `try` block competes with no exceptions
- `finally`
  - Useful for cleanup actions

```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    else:
        return f"The result is: {result}"
    finally:
        print("Execution complete.")
```

```
Execution complete.
'The result is: 2.0'
```

Recap: Assertion as a defensive programming

- Check **inputs** to functions, but can be used anywhere
- Check **outputs** to functions to avoid propagating bad values
- Can make it easier to **debug**.

```
assert <statement that should evaluate to be true>, "statement not true"
```

```python
def divide(a, b):
    assert b != 0, "Division by zero is not allowed"
    return a / b

# Test the function
print(divide(10, 2))  # This will work fine
print(divide(10, 0))  # This will raise an AssertionError
```

```python
class Person:
    def __init__(self, age):
        assert age > 0, "Age must be positive"
        self.age = age

# Create a Person object
p = Person(25)  # Works fine
p = Person(-5)  # Raises AssertionError
```

Program efficiency is also important

Besides **correctness**

- Test cases to check the output …

The **efficiency** of the programs is also of great importance

- **Data sets** can be very large
- Problem can get **complex**

What should we really are about?

- **Time** efficiency (how fast)
- Space efficiency (how much memory)
- **Tradeoff** between them (use more memory to save time)
  - Fibonacci recursive v.s Fibonacci with memorization
- Focus on the algorithms not implementations
  - `while` and `for`

1. Timing programs and counting operations

2. Big Oh and Theta

3. More Examples

Measure with a timer

We can evaluate programs by

- Measure with a timer
- Count the operations

- Start clock
- Call function
- Stop clock

```
def my_fun(n):
    total = 0
    for i in range(n):
        total += i
    return total
```

```
import time
def measure_time(func, n):
    start = time.time()
    func(n)
    end = time.time()
    return end - start
input_sizes = [10, 100, 1000, 10000, 100000]
for n in input_sizes:
    execution_time = measure_time(my_fun, n)
    print(f"n = {n:<7}, Time = {execution_time:.6f} seconds")
```

Seconds since the **epoch**: Jan, 01, 1970, where time starts for Unix Systems.

```
n = 10     , Time = 0.000005 seconds
n = 100    , Time = 0.000007 seconds
n = 1000   , Time = 0.000058 seconds
n = 10000  , Time = 0.000624 seconds
n = 100000 , Time = 0.007215 seconds
```

Timer in practice: `time.perf_counter()`

`time.perf_counter()` in practice

- Specifically designed for performance measurement
- More accurate, higher precision, often in nanosecond v.s microsecond for `time.time()`

- Start clock
- Call function
- Stop clock

```python
def perf_measure(func, n):
    start = time.perf_counter()
    func(n)
    end = time.perf_counter()
    return end - start
input_sizes = [10, 100, 1000, 10000, 100000]
for n in input_sizes:
    execution_time = perf_measure(my_fun, n)
    print(f"n = {n:<7}, Time = {execution_time:.6f} seconds")
```

```
n = 10     , Time = 0.000001 seconds
n = 100    , Time = 0.000002 seconds
n = 1000   , Time = 0.000019 seconds
n = 10000  , Time = 0.000166 seconds
n = 100000 , Time = 0.001698 seconds
```

Does the time depend on the input parameters? How?

Potential problem with timing to evaluate efficiency?

Counting operations

Besides measure with a timer, We can also evaluate programs by

- Count the operations

Assume all those steps take **constant time**

- Mathematical operations
- Comparisons
- Assignments
- Accessing objects in memory (indexing)

```python
def array_sum(arr):
    total = 0
    for i in range(len(arr)):
        total += arr[i]
    return total
```

```python
def array_sum(arr):
    total = 0  # 1 operation (assignment)
    for i in range(len(arr)):
        total += arr[i]  # 3 operations per iteration:
                         # 1 for array access
                         # 1 for addition
                         # 1 for assignment back to total
    return total  # 1 operation (return)
```

$1 + (3 * n) + 1 =$ **$3n + 2$**

Counting operations

```
def print_all_pairs(arr):
    n = len(arr)  # 1 operation
    for i in range(n):
        for j in range(n):
            print(f"({arr[i]}, {arr[j]})")  #
print_all_pairs([1,2,3])
```

```
(1, 1)
(1, 2)
(1, 3)
(2, 1)
(2, 2)
(2, 3)
(3, 1)
(3, 2)
(3, 3)
```

1 + (n * n) * 2 = $\mathbf{2n^2 + 1}$

```
import time
def fibonacci(n):
    if n <= 1:
        return 1  # Base case, 1 operation
    # For each n, the function calls itself twice
    return fibonacci(n - 1) + fibonacci(n - 2)

def perf_measure(func, n):
    start = time.perf_counter()
    func(n)
    end = time.perf_counter()
    return end - start

input_sizes = [5, 10, 20, 100]
for n in input_sizes:
    execution_time = perf_measure(fibonacci, n)
    print(f"n = {n:<7}, Time = {execution_time:.6f} seconds")
```

```
n = 5      , Time = 0.000006 seconds
n = 10     , Time = 0.000034 seconds
n = 20     , Time = 0.003413 seconds
```

What is the closed form?

Reference

~2^n (exponential)

Problems with timing and counting

Timing the exact running time of the program:

- Depends on **machine**
- Depends in **implementation**
- Small inputs don't show growth

Counting the exact number of steps:

- Gives a formula
- Do NOT depend on machine
- Depends on the implementation
- Also consider irrelevant operations for largest inputs
  - Initial assignment, addition

Goal:

- Evaluate algorithms (not implementation)
- Evaluate scalability just in terms of input size

1. Timing programs and counting operations

2. Big Oh and Theta

3. More Examples

Asymptotic growth: the order of growth

We can evaluate programs by

- Timer
- Count the operations
- Abstract notion of order of growth

Goal: Describe how run time grows as size of input grows

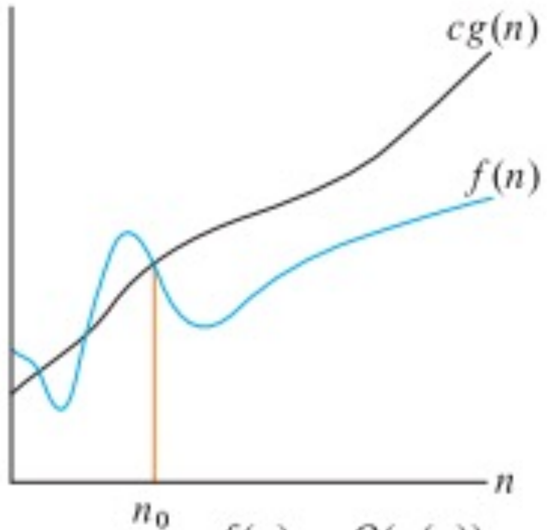
- Want to put a **bound** on growth
- Do **NOT** need to be precise: “order of ” not ”exact” growth
- Want to focus on terms that grows most rapidly
    - Ignore additive and multiplicative constants

This is called order of growth

- Use mathematical notions of “Big Oh(O)” and “Big Theta(Θ)”

Big O definition

O-notation characterizes an **upper bound** on the asymptotic behavior of a function. In other words, it says that a function grows no faster than a certain rate, based on the highest-order term.



$f(n) = O(g(n))$

A function $f(n)$ belongs to the set $O(g(n))$

For example:

$f(n) = 3n^2 + 5$

$g(n)$?

Just an upper bound:

$c_0 g(n) = 4n^2$

$c_0 g(n) = 2n^3$

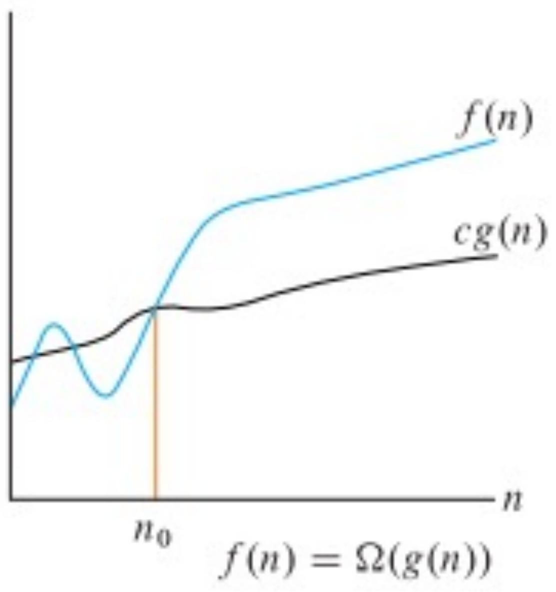
$c_0 g(n) = 1 * 2^n$

$f(n) = O(g(n))$ means there exist constants c_0, n_0 for which

$$c_0 g(n) \geq f(n) \qquad \forall\, n > n_0$$

Big Ω definition

Ω -notation characterizes a **lower bound** on the asymptotic behavior of a function. In other words, it says that a function grows no slower than a certain rate, based on the highest-order term.



For example:

$f(n) = 3n^2 + 5$

$g(n)$?

Just an lower bound:

$c_0 g(n) = 2n^2$

$c_0 g(n) = 2n$

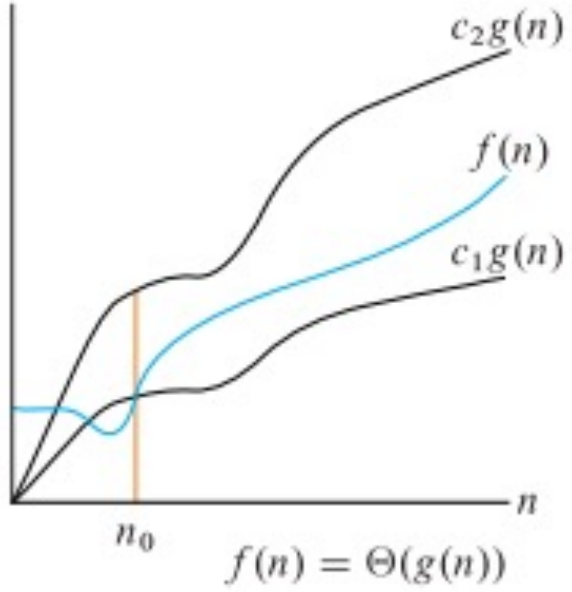
$c_0 g(n) = 5$

$f(n) = \Omega(g(n))$ means there exist constants c_0, n_0 for which

$$f(n) \geq c_0 g(n).\ \forall\, n > n_0$$

Big Θ definition

Θ -notation characterizes a **tight bound** (upper and lower) on the asymptotic behavior of a function.



Now for Θ -notation :

$f(n) = 3n^2 + 5$

$g(n)$?

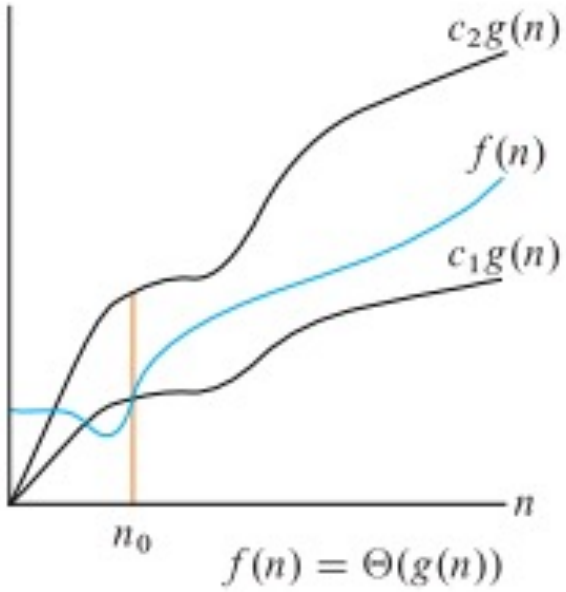
$c_0 g(n) = 4n^2$

$f(n) = \Theta(g(n))$ means there exist constants $c_1,\ c_2,\ n_0$ for which

$$c_2 g(n) \geq f(n) \geq c_1\, g(n).\ \forall n > n_0$$

O v.s Θ

In practice, ***Θ* bounds are preferred**, because they are "tighter"



Now for Θ -notation :

$$f(n) = 3n^2 + 5$$

$$g(n)\ ?$$

$c_0 g(n) = 4n^2$ ✓

$c_0 g(n) = 2n^3$ ✗

$c_0 g(n) = 1 * 2^n$ ✗

Let’s try to find Θ(x)

- Drop **constants** and **multiplicative** factors
- Focus on **dominant** terms

$$1000*\log(x) + \boxed{x}$$

$$n^2\log(n) + \boxed{n^3}$$

$$\log(y) + 0.000001\boxed{y}$$

$$\boxed{2^b} + 1000a^2 + 100*b^2 + 0.0001\boxed{a^3}$$

Θ(x) Complexity Classes

- $\Theta(1)$: denotes constant running time
- $\Theta(\log n)$: denotes logarithmic running time
- $\Theta(n)$: denotes linear running time
- $\Theta(n \log n)$: denotes log-linear running time
- $\Theta(n^c)$: denotes polynomial running time
- $\Theta(c^n)$: denotes exponential running time
- $\Theta(n!)$: denotes factorial running time

In-class practice

Θ(x): Worst case scenario

We'll usually (but not always) concentrating on find the worst-case running time.

- Holds true for any input
- Happen fairly often.
- Sometimes, we use average-case

```python
def is_element_in_list(element, lst):
    for item in lst:
        if item == element:
            return True
    return False
```

```python
numbers = [1, 3, 5, 7, 9, 11, 13, 15]
print(is_element_in_list(1, numbers))
print(is_element_in_list(7, numbers))
print(is_element_in_list(19, numbers))
```

```
True
True
False
```

- $\Theta(1)$: denotes constant running time
- $\Theta(\log n)$: denotes logarithmic running time
- $\Theta(n)$: denotes linear running time
- $\Theta(n \log n)$: denotes log-linear running time
- $\Theta(n^c)$: denotes polynomial running time
- $\Theta(c^n)$: denotes exponential running time
- $\Theta(n!)$: denotes factorial running time

1. Timing programs and counting operations

2. Big Oh and Theta Notation

3. More Examples

Constant and linear running time

- $\Theta(1)$: denotes constant running time
- Independent of input size

```python
def get_element(arr, index):
    return arr[index]
```

```python
def is_even(number):
    return number % 2 == 0
```

- $\Theta(n)$: denotes linear running time

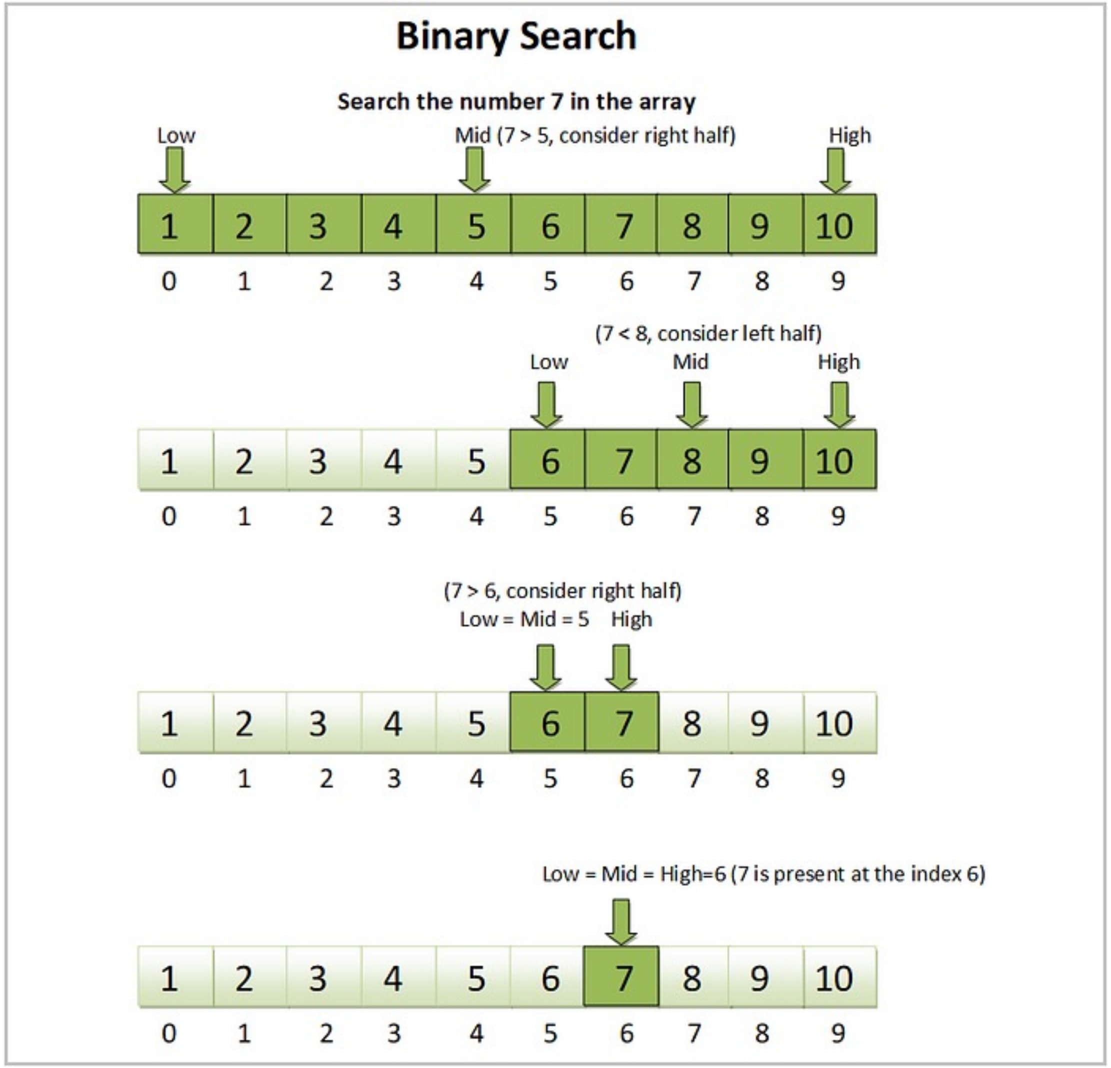
```python
def array_sum(arr):
    total = 0  # 1 operation (assignment)
    for i in range(len(arr)):
        total += arr[i]  # 3 operations per iteration:
                         # 1 for array access
                         # 1 for addition
                         # 1 for assignment back to total
    return total  # 1 operation (return)
```

Logarithmic running time

- $\Theta(\log n)$: denotes logarithmic running time
- Example: takes a **sorted** array and a target value as input.

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid  # Target found, return its index
        elif arr[mid] < target:
            left = mid + 1  # Target is in the right half
        else:
            right = mid - 1  # Target is in the left half

    return -1  # Target not found

# Example usage
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7
result = binary_search(sorted_array, target)

if result != -1:
    print(f"Target {target} found at index {result}")
else:
    print(f"Target {target} not found in the array")
```

```
Target 7 found at index 6
```



Reference: https://dolly-desir.medium.com/algorithms-binary-search-2656c7eb5049

Log-linear running time

- $\Theta(n \log n)$: denotes log-linear running time
  - Divide $(\log n)$
  - Merge (n)

These algorithms are generally preferred for large datasets because they **scale** well.

More on sorting: [Intro to Algorithm]

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
    # Recursively sort both halves
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)
    # Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    left_index, right_index = 0, 0

    # Compare elements from both lists and add the smaller one to the result
    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1
    result.extend(left[left_index:])
    result.extend(right[right_index:])
    return result

# Example usage
unsorted_array = [64, 34, 25, 12, 22, 11, 90]
sorted_array = merge_sort(unsorted_array)
print("Sorted array:", sorted_array)
```

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

Exponential and factorial running time

- $\Theta(c^n)$: denotes exponential running time
- Sometimes can be replaced by faster algorithms via memorization

```python
import time
def fibonacci(n):
    if n <= 1:
        return 1  # Base case, 1 operation
    # For each n, the function calls itself twice
    return fibonacci(n - 1) + fibonacci(n - 2)

def perf_measure(func, n):
    start = time.perf_counter()
    func(n)
    end = time.perf_counter()
    return end - start

input_sizes = [5, 10, 20, 100]
for n in input_sizes:
    execution_time = perf_measure(fibonacci, n)
    print(f"n = {n:<7}, Time = {execution_time:.6f} seconds")
```

```
n = 5      , Time = 0.000006 seconds
n = 10     , Time = 0.000034 seconds
n = 20     , Time = 0.003413 seconds
```

- $\Theta(n!)$: denotes factorial running time
  - For n distinct objects, there are $(n!)$ permutations

```python
def permute(lst):
    # Base case: if the list has only one element
    if len(lst) == 1:
        return [lst]
    # Recursive case
    permutations = []
    for i in range(len(lst)):
        current = lst[i]
        remaining = lst[:i] + lst[i+1:]
        for p in permute(remaining):
            permutations.append([current] + p)
    return permutations

# Example usage
example_list = [1, 2, 3]
result = permute(example_list)
print(f"All permutations of {example_list}:")
for perm in result:
    print(perm)
```

```
All permutations of [1, 2, 3]:
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Other things

HW4 due this week.

Midterm in 2.5 weeks.

Read chap1-3 of Intro to Algorithm

Coming next:
Intro to Numpy.