LAB 10

Midterm 2 Review

Mingxiao Wei
mingxiaowei@berkeley.edu

Apr 3, 2023

LOGISTICS 🏡

- Homework 07 due Thu 04/06
- Midterm 👀
  - Lectures this week are in scope 🤔
  - The midterm 2 study guide will be out soon - make sure to check it beforehand! (check Ed 2069 for updates)
  - Check Ed for more resources!!

TODAY'S OUTLINE ☀️

- Topical Review
  - Efficiency + Lab Q11
  - Iterators and generators + Lab Q4
  - Recursion and tree recursion + Lab Q1-2
  - Lab Q10 (trees) walkthrough, if time allows
- If you want individual help in the meantime, just walk to one of our AIs!

ORDER OF GROWTH

ORDER OF GROWTH

- Order of growth / efficiency - runtime as a <u>**function**</u> of the input size
- Input size
  - numeric - magnitude of the number
  - Python lists - length of the list
  - linked list/trees - number of nodes
- Runtime
  - the number of operations
- Theta notation for approximate runtime
  - $\Theta(f(n))$ - approximately $f(n)$ by a constant factor
  - Only consider the term that grows fastest
- constant < logarithmic < linear < quadratic < exponential

ORDER OF GROWTH - CONSTANT

- Constant ↔ $\Theta(1)$
- Runtime does not change as the input size changes
- For example:

```
def square (x):
    return x * x
```

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

ORDER OF GROWTH - LOGARITHMIC

- Logarithmic $\leftrightarrow$ $\Theta(\log n)$
- Often when we keep dividing the input by a constant

```
def foo(x):
    while x > 0:
        print('hey')
        x //= 2
```

Let's say the while loop runs n times before x reaches 0.

divide x by 2 for n times before it gets to 1

$$\implies \frac{x}{2^n} = 1 \implies 2^n = x \implies n = \log_2 x$$

ORDER OF GROWTH - LINEAR

- Linear $\leftrightarrow \Theta(n)$
- Often when a loop runs n times, each time doing work in constant time

```
def factorial(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i  # execute x times in total
    return prod
```

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

ORDER OF GROWTH - QUADRATIC

- Quadratic $\leftrightarrow \Theta(n^2)$
- Often when a nested loop runs n^2 times, each time doing work in constant time

```python
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a, b)  # print n * n times in total
```

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

ORDER OF GROWTH - EXPONENTIAL

- Exponential $\leftrightarrow \Theta(c^n)$, where c is a constant
- Often in tree recursion

```python
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
…	…	…	…
10	`rec(10)`	1024	1023
…	…	…	…
n	`rec(n)`	`2^n`	`2^n`

ORDER OF GROWTH - TREE RECURSION

```python
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

- Try drawing out the recursion tree diagram
- $n + 1$ levels
- The i^{th} level has 2^i nodes (root at level 0)
- Each node does constant work (addition)
- Total = $(1 + 2 + \ldots + 2^{n+1}) \cdot constant = \Theta(2^n)$

ORDER OF GROWTH - TREE RECURSION

What about this? 🤔

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n // 2) + rec(n // 2)
```

- Draw out the recursion tree diagram
- $\log_2 n$ levels
- The i^{th} level has 2^i nodes (root at level 0)
- Each node does constant work (addition)
- Total = $(1 + 2 + \ldots + 2^{\log_2 n}) \cdot constant = \Theta(n)$

ORDER OF GROWTH - LOOPS

- In general, (# times the loops run) * (work done each time)

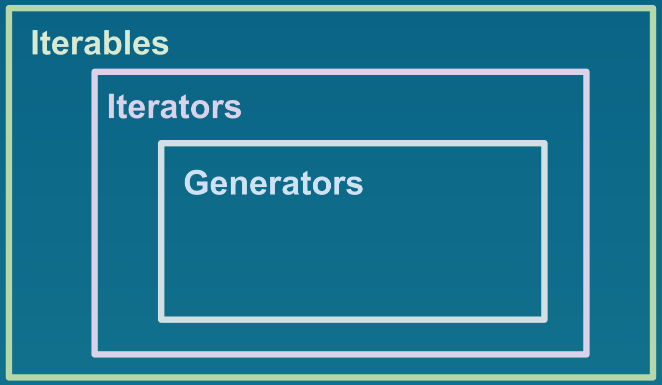
```
def factorial(n):
    # returns n! in linear time
    ...

def foo(n):
    # nested loop runs n^2 times
    for i in range(n):
        for j in range(n):
            # each time this takes theta(n) time
            print(factorial(n))
```

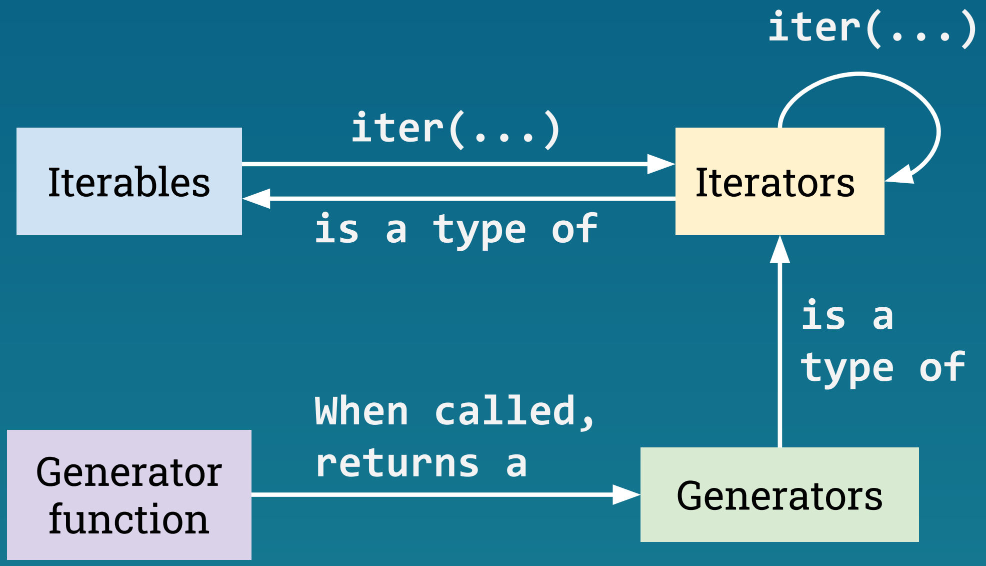
- `foo(n)` runs in $\Theta(n^3)$ time

LAB Q11

ITERATORS AND GENERATORS 🛸

OVERVIEW


Iterables
Iterators
Generators


iter(...)
Iterables
iter(...)
Iterators
is a type of
is a
type of
When called,
returns a
Generator
function
Generators

ITERABLES AND ITERATORS

- Iterables
  - Can be iterated through using a for loop:

```
for elem in iterable:
    # do something
```

  - E.g., sequences (lists, tuples, dictionaries, ranges, etc.)
  - `iter(sequence)` - returns a *new* iterator
- Iterators
  - "Tracker" for an iterable
  - `next(iterator)` gives the next element
    - cannot go back
  - `iter(iterator)` - returns the iterator *itself*, without resetting the position

MORE ON ITERATORS

- Iterators can be "depleted" implicitly with:
  - for loop
  - list comprehension
  - `list(iterator)`
- To iterate through an iterator:
  - use a for loop

```
for elem in iterator:
    # do sth with elem
```

  - use a while loop

```
while ...:
    elem = next(iterator)
    # do sth with elem
```

GENERATOR FUNCTIONS AND GENERATORS

- Generator functions
    - Have by at least one `yield` or `yield from` statement
    - When called, return a **new** generator, without evaluating the function body yet
    - Define customized iterators
- Generators
    - `next(generator)` - "lazy evaluation"
        - Keep evaluating until a value is yielded or the function teriminates

YIELD VS YIELD FROM

- Both defines elements in the generator
- `yield` only one element at a time
- `yield from` any iterables (sequences, iterators, generators, etc.)

```
for elem in iterable:
    yield elem
# is equivalent to
yield from iterable
```

- for loop and `yield` - can modify the element before yielding
- `yield from` - can only yield the exact elements from the iterable

RECURSIVE GENERATORS

- Base case
  - No more elements to yield - often just `return`
- Recursive case
  - Often iterate through the recursive call (generator) with a for loop
- Combination step
  - What does the recursive generator contain? - recursive leap of faith!
  - Given that, how to yield for the current call?

```
def yield_paths(t, value):
    """Yields all possible paths from the root of t
    to a node with the label value as a list."""
    if t.label == value:
        yield [value]
    for b in t.branches:
        for path in yield_paths(b, value):
            yield [t.label] + path
```

LAB Q4

RECURSION,
TREE RECURSION 🌀

RECURSION

- Recursive function - a function calls itself
  - Base case
  - Recursive case
  - Combination step
- Tree recursive function - a function makes more than one call to itself
  - Often times one recusive call represents one choice
  - What choices do we have and how do we combine them together?

BASE CASE

- Simplest input - give an answer without further computation
- Often hinted in the doctest
- Sometimes may need an "overshot" case
    - Could it overshoot in the first place?
    - E.g., input `n` is positive - may need to check if `n <= 0`
- Some common base cases *
    - positive integer - `n <= 0` or `n == 1`
    - sequence - `len(seq) == 0`
    - Trees - `t.is_leaf()`
    - Linked lists - `lnk is Link.empty`

* The goal of these is to help you develop some intuition. Please do NOT take these for granted on an exam - it really depends on the specific problem!

RECURSIVE CASE

- Break down the problem into a smaller one
  - How should each argument change in the recursive call, if they can change at all?
- Make a recursive call for the smaller problem
- Some common recursive cases *
  - positive integer - `n - some_number` or `n // 10`
  - sequence - `seq[1:]`
  - Trees - often one recursive call on each branch
  - Linked lists - `lnk.rest`

* The goal of these is to help you develop some intuition. Please do NOT take these for granted on an exam - it really depends on the specific problem!

COMBINATION STEP

- What does the recursive call return?
    - based on the problem description, NOT your implementation
    - recursive leap of faith - assume that the recursive call always gives the correct result
- Given the solution to a smaller problem, how to solve the bigger problem?

HELPER FUNCTIONS FOR RECURSION

- When to use:
  - Need to keep track of more variables than the given parameters of the outer function
- Where
  - usually nested within the original function
- How
  - define the helper function, and return a call to it with appropriate initial arguments

LAB Q1-2

AND THAT'S IT... 🤞

GOOD LUCK with the midterm!

We got this >:)