Week 8: Agenda

- Course admin
  - Homework 6
  - Gradebook is up! It will show your mid-semester grades
    - Quizzes
      - Quiz 1: 1%
      - After Quiz 1 (up to quiz 6): 9%
      - Quiz 4: optional
      - Quiz 6: It can be replaced by Quiz 11
    - Exam 1: 50%
    - Homework: 35% (20% + TP) **CS academy problems not included**
    - Participation: 5% (Mentor Meetings 2% / CS Academy: 3%) **not included**
    - (60%, 65%) rule
    - No Better-late-than-never (not yet)
  - Mid-semester grades
- What's next?
- This week: Dictionaries & Sets

What’s next?

What's next?

HW#7 Due	W8	Dictionaries, Sets, Efficiency	Quiz #7 - Lists
HW#8 Due	W9	Recursion	Quiz #8 - Dict. Sets, Eff
HW#9 Due	W10	OOP (Term Project Intro)	Quiz #9 - OOP Term Project Season Starts!
Project Proposals Due	W11	Exam 2 (**Thu,** March 27th), OOP	
	W12	Searching and Sorting / Hashing	
	W13	Hashing	
	W14	…	

```python
import copy

def ct1(L):
    a = L
    b = copy.copy(L)
    c = copy.deepcopy(L)
    b[1][1] = c[0][0]
    c[1].append(b[1][0])
    a[0] = b[1]
    a[0][0] += b.pop()[0]
    return a,b,c

L = [[1],[2,5]]
for val in ct1(L):
    print(val)
print(L)
```

Dictionaries

- Why? (Functional)
  - Store relations

key
value

key $\mapsto$ value

Examples:

- SSNs $\mapsto$ Person information data
- Names $\mapsto$ phone numbers, email
- Usernames $\mapsto$ passwords, OS preferences
- ZIP codes $\mapsto$ Shipping costs and time
- Country names $\mapsto$ Capital, demographic info
- Sales items $\mapsto$ Quantity in stock, time to order
- Courses $\mapsto$ Student statistics
- Persons $\mapsto$ Friends in social network
- Animals $\mapsto$ Classification data
- Companies $\mapsto$ Rate, capital, investments
- …

- In all the examples, a **unique label** (***key***) can be associated to a (more or less complex) **piece of data** (the ***value***)
- This motivates the choice of a *dictionary data structure* to represent and manipulate these type of data

Dictionaries: Functionality and Syntax

- Keys vs. Values
- Create / Initialize
- Add (key, values)
- Retrieve/Update values
- Iterate
- Remove (key, values)
- Be aware of aliasing! *(as we saw with lists)*

Sets

- Why? (Functional)
  - Non-duplicate elements

```python
mySet = set()

mySet.add(5)
mySet.add(6)
mySet.add(10)
mySet.add(4)
mySet.add(5)

print(mySet)
```

  - Use set operations not available for lists
    - Union, intersection, difference, …

```python
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)
```

Sets: Functionality and Syntax

- Create / Initialize
- Add values
- ~~Retrieve/Update values~~
- Delete values
- Iterate
- Be aware of aliasing! *(as we saw with dictionaries and lists)*

Write the function `destructiveListInList(a, b, n)` which destructively modifies `a` (without modifying `b`) by adding all the values of `b` between elements `a[n-1]` and `a[n]` and returns the usual value returned by destructive functions. When `n == 0`, it adds all values of `b` onto the front of `a`. You may assume `0 <= n <= len(a)`. When `n == len(a)`, it adds all values of `b` at the back of `a` (the function behaves like `a.extend(b)`).

So this code works:

```
L = [3, 4, 1, 2]
destructiveListInList(L, [7, 6], 2)
assert(L == [3, 4, 7, 6, 1, 2])

L = [1, 3]
destructiveListInList(L, [4, 2], 2)
assert(L = [1, 3, 4, 2])

A = [4, 5]
B = [7, 8, 9]
destructiveListInList(A, B, 1)
assert(A == [4, 7, 8, 9, 5])
assert(B ==  [7, 8, 9])
```

You may not import or use any module other than copy. You may not use any method, function, or concept that we have not covered this semester. We may consider additional test cases not shown here.

diagonalsMatch(L)

Write a function that returns True if the given list L is a square 2D list (i.e., a list of lists with equal row and column lengths) and both its main diagonal (from top-left to bottom-right) and anti-diagonal (from top-right to bottom-left) contain the same values. Otherwise, return False.

Dictionaries & Sets

- They are fast, blazingly fast, at performing certain operations
  - Membership (both)
  - Retrieving/updating a value for a key (dictionaries)
  - Removing elements (both)

```python
myDictionary = {'Qatar':'Doha', 'Oman':'Muscat'}

elem1 = 'Qatar'
elem2 = 'google'
print(elem1 in myDictionary)
print(elem2 in myDictionary)

print(myDictionary[elem1])
print(myDictionary[elem2]) # Don't do that, it fails! why?

mySet = {"google", "microsoft", "apple"}

print(elem1 in mySet)
print(elem2 in mySet)
```

Example: `mostFrequentWord(wordList)`

- Given a list of words from a large text, return the most frequent word together with its frequency. If ties, return any of the most frequent words.

`pairSumsToN(L, n)`

- Find two elements of the list `L` that sum to `n`.
- If no pair exists, return None

Measuring performance

Measuring performance requires that we determine how an algorithm’s execution time increases <u>with respect to</u> the input size

Why would you want to do this?

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

What we do instead:

- Characterize running time as a function of the input size, e.g., n

Why?

- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

How?

- Count the number of operations in our algorithm assuming a **worst case scenario**

Example

```
def foo(L): #L is a list
    uselessVariable = 43
    if len(L) > 0:
        return L[0] * 3
    return 42
```

Example

```
def foo(L): #L is a list
    i = 1
    result = 0
    while i < len(L):
        result += L[i]
        i += 1
    return result
```

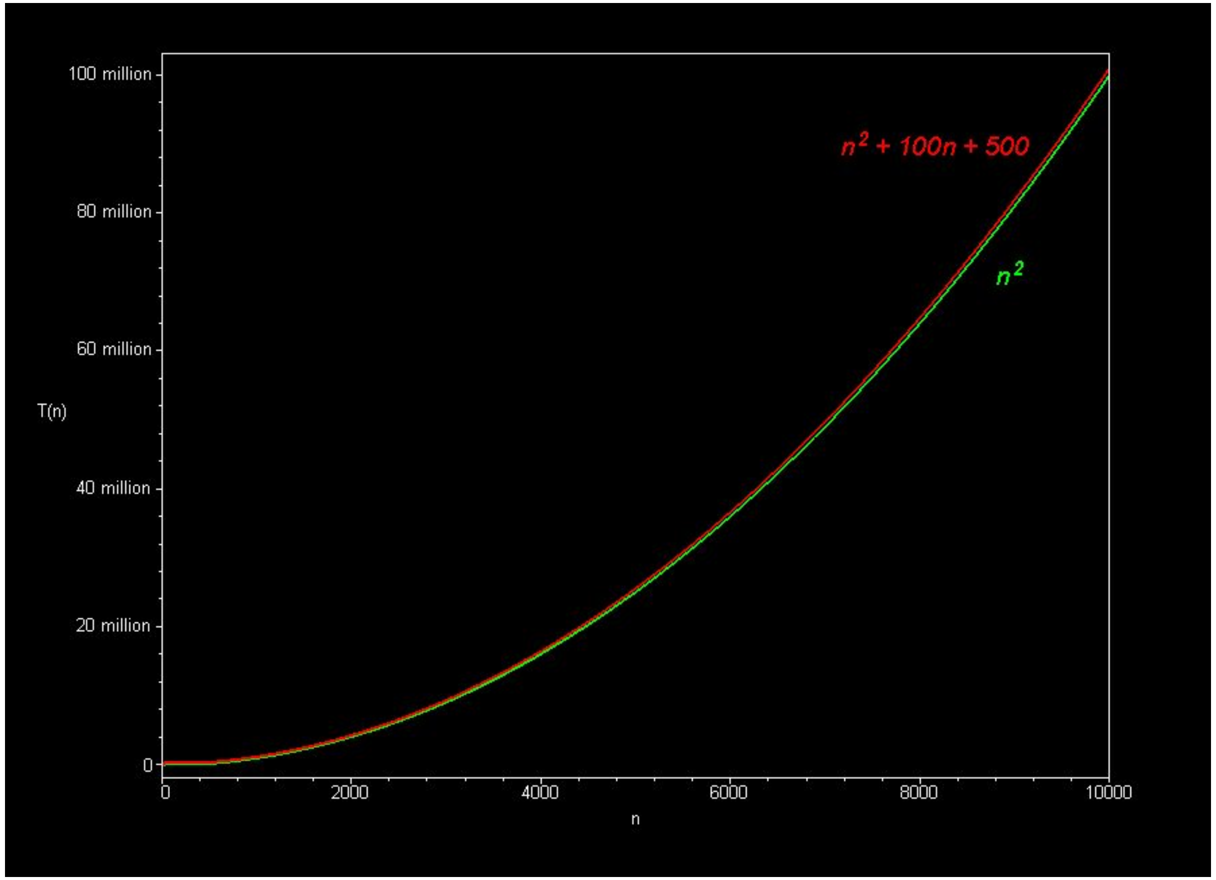
Big-O: Simplify

- **We only care about how the number of steps grows with the input size**
- This *growth rate* is not affected by constant factors or lower-order terms
- Examples (number of operations)
- 102*n + 105 has a linear growth rate (just like n)
- 105*n^2 108*n has a quadratic growth rate (just like n^2)
- 3*n^3 + 20*n^2 has a cubic growth rate (just like n^3)
- 97*log(n) + log(log (n)) has a logarithmic growth rate (just like log(n))

The slower the growth rate, the more efficient the algorithm, so choose an algorithm with a slower growth rate!

Efficiency

- Why we don't care about lower magnitude terms

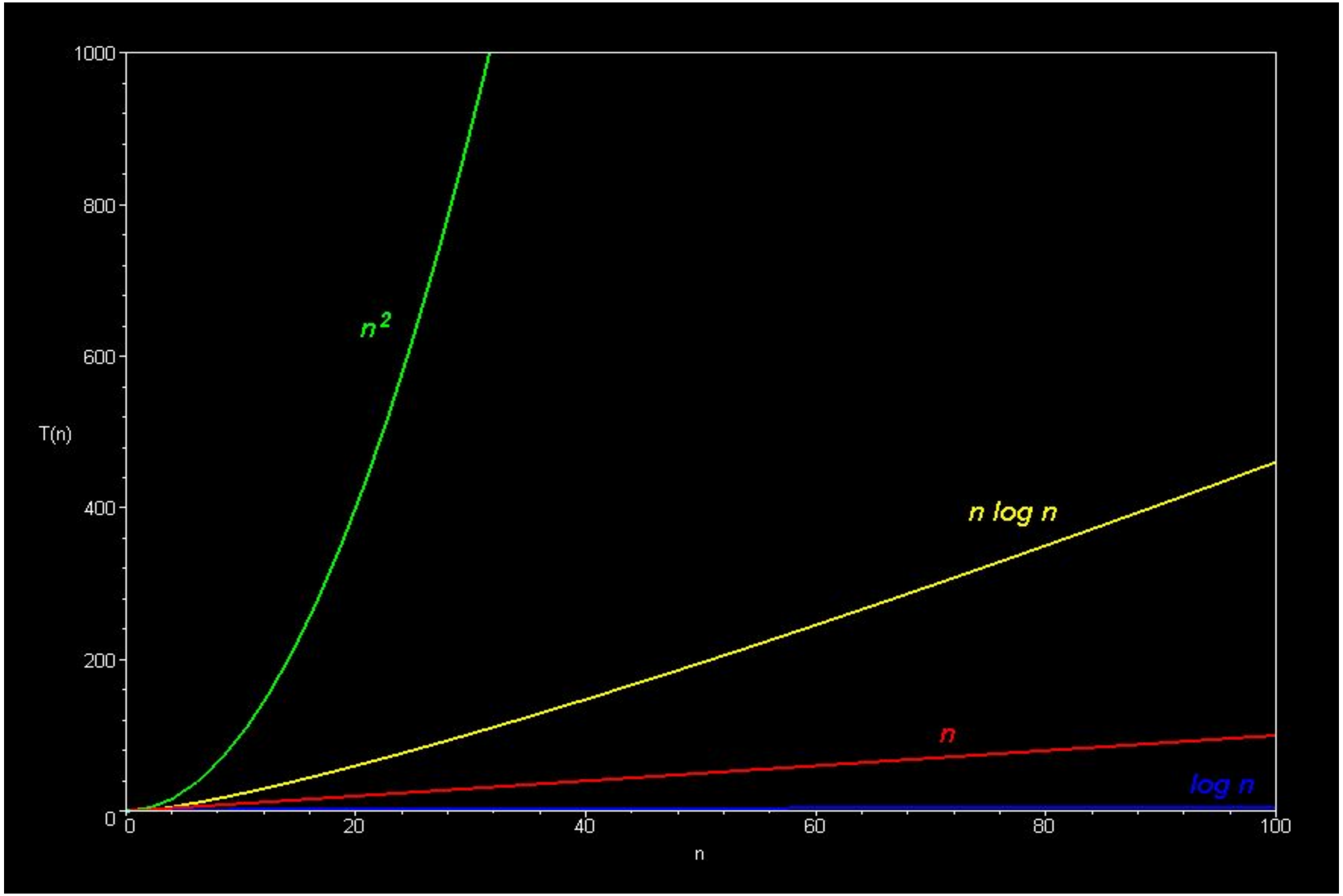


Big-O: Notation

N is the size of our input

- O(1): Constant
- O(N): Linear growth
- O(N^2): Quadratic growth
- O(N^3): Cubic
- …
- O(logN): Logarithmic
- O(2^n): Exponential

Efficiency classes



Example

Calculate the Big-O

```
def foo(n):
  i = 0
  while i < 1000*n:
    print("42")
    i = i + 1
```

Example

Calculate the Big-O

```
def foo(n):
   i = 0
   j = 0
   while i < n:
      while j < n:
         print(“one operation”)
         j = j + 1
      i = i + 1
```

Most Frequent Words (bad)

```
def mostFrequentWordv1(wordList):
    maxword = None
    maxcnt = 0
    for word in wordList:
        cnt = wordList.count(word)
        if cnt > maxcnt:
            maxcnt = cnt
            maxword =word
    return (maxword, maxcnt)
```

. (15 points) **Free Response: Fluke Numbers** A *fluke number* (coined term) is an integer that has a frequency in the list equal to its value

Write the function `findFlukeNumbers(L))` that is given a list `L` of objects (not necessarily integers). The function should return a set containing all the fluke numbers in the list. Your solution should run in $O(N)$ time.

For example,

```
assert(findFlukeNumbers([1,'a','a',[4], 3, False, 3, 3]) == {1, 3})
assert(findFlukeNumbers([1, 2, 2, 3, 3, 3, 4]) == {1, 2, 3})
assert(findFlukeNumbers([0, False, 'hello']) == set())
```