SEARCHING AND SORTING ALGORITHMS

(download slides and .py files from Stellar to follow along!)

6.0001 LECTURE 10

Eric Grimson

LAST TIME

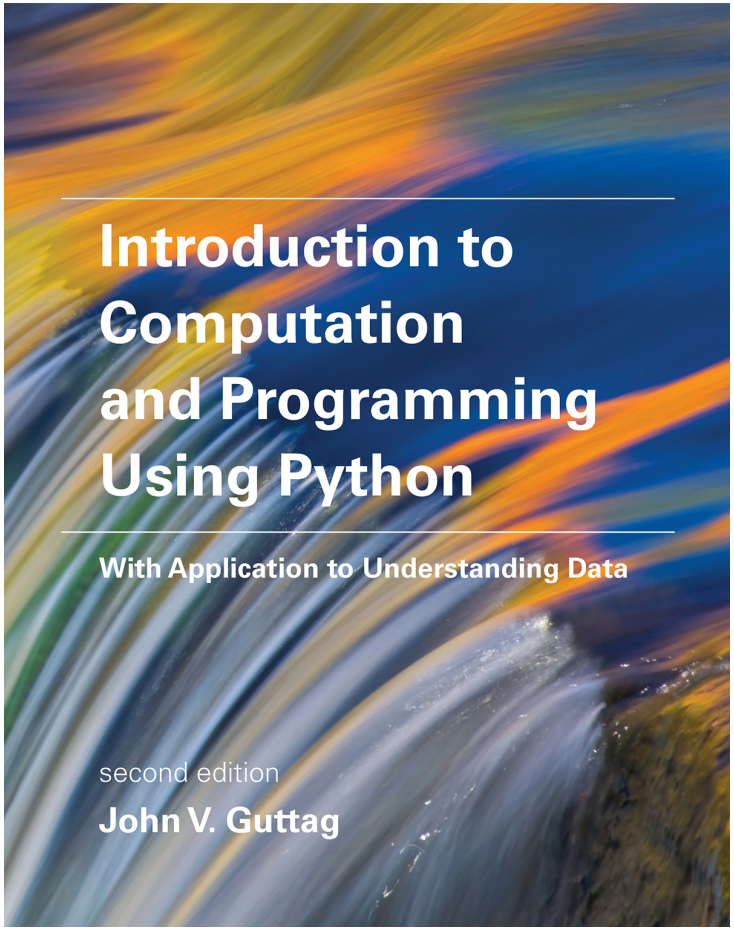
- complexity, order of growth, big oh notation
- best, average, worst case scenario
- linear, logarithmic, polynomial complexity examples

TODAY

- more classes of complexity and examples
- amortized complexity
- sorting algorithms, and their complexity
- **summary of 6.0001**

READING

- Chapter 10



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

MEASURING RUN-TIME

- can **time** it by importing the time module
- can **count** number of operations
- can express the **order of growth**

PROBLEMS WITH TIMING AND COUNTING

- **timing** the exact running time of the program
  - depends on **machine**
  - depends on **implementation**
  - **small inputs** don't show growth
- **counting** the exact number of steps
  - **machine independent**, which is good
  - depends on **implementation**
  - **multiplicative/additive constants** are irrelevant for large inputs

MEASURING ORDER OF GROWTH: BIG OH NOTATION

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth
  - concerned with growth as size of problem gets large

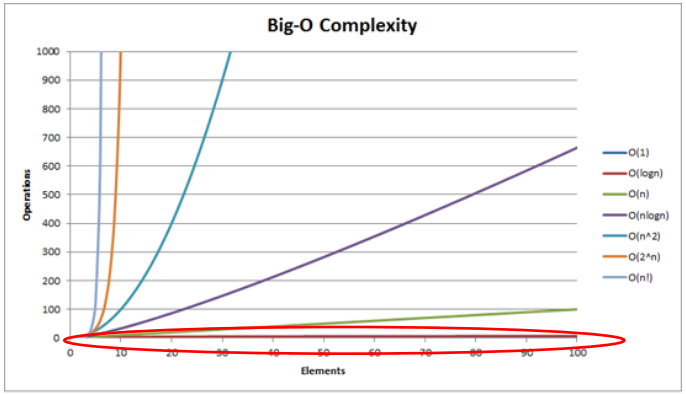
- **Big Oh or O()** is used to describe worst case
  - worst case occurs often and is the bottleneck when a program runs
  - express rate of growth of program relative to the input
  - evaluate algorithm not machine or implementation

COMPLEXITY CLASSES

- *O(1)* denotes constant running time
- *O(log n)* denotes logarithmic running time
- *O(n)* denotes linear running time
- *O(n log n)* denotes log-linear running time
- $O(n^c)$ denotes polynomial running time (c is a constant)
- $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)

CONSTANT COMPLEXITY

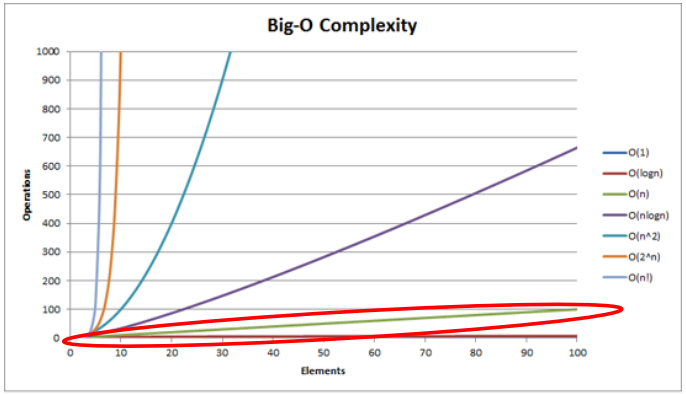
CONSTANT COMPLEXITY – O(1)



- only pass through the code once (or a constant number of times independent of the size of the problem)
- great when we can do it, as very efficient
- examples:
  - list index
  - list append

LINEAR COMPLEXITY

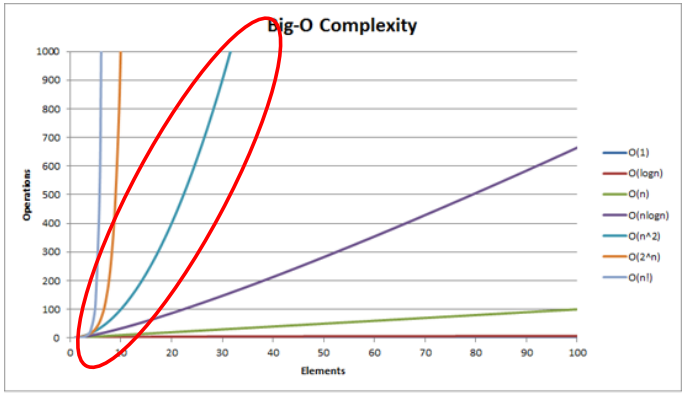
LINEAR COMPLEXITY – O(n)



- often characterized by loops (or recursive calls) that execute once for each element of a data structure or for each increment of an argument
  - single recursive call for each element of list
  - single pass through loop for each increase in argument
- examples:
  - unordered search from last time
  - ordered search from last time

POLYNOMIAL COMPLEXITY

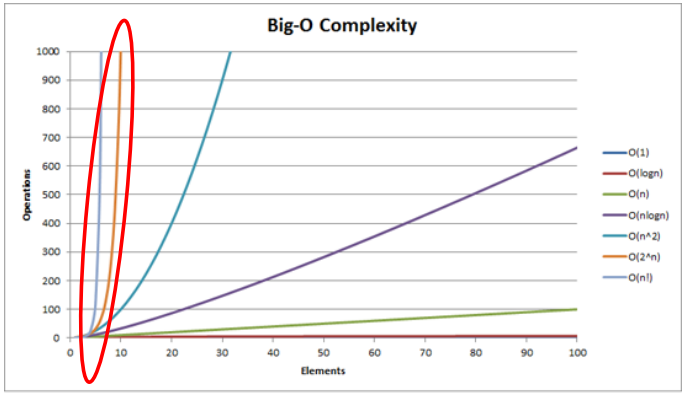
QUADRATIC COMPLEXITY – $O(n^2)$



- often characterized by nested loops
  - inner loop is linear
  - outer loop also linear, but does linear work of inner loop each time
- example
  - intersect code from last time

EXPONENTIAL COMPLEXITY

EXPONENTIAL COMPLEXITY – $O(c^n)$



- Recursive functions where have more than one recursive call for each size of problem
  - Fibonacci

- Many important problems are inherently exponential
  - Unfortunate, as cost can be high
  - Will lead us to consider approximate solutions more quickly

EXAMPLE: Power Set

- given a list of elements, create a list of all the sublists (of size 0, 1, 2, …, n)
- L = [1, 2, 4]
- power set is

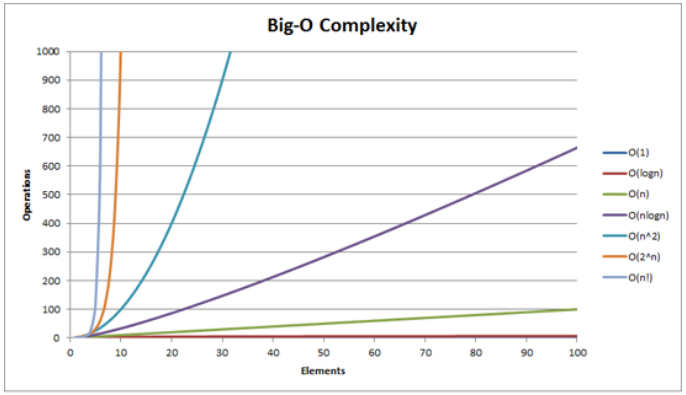
[[],

[1], [2], [4],

[1,2], [1,4], [2,4],

[1,2,4]]

EXPONENTIAL COMPLEXITY




```
def genSubsets(L):
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

Go until reach empty list; return list of empty list

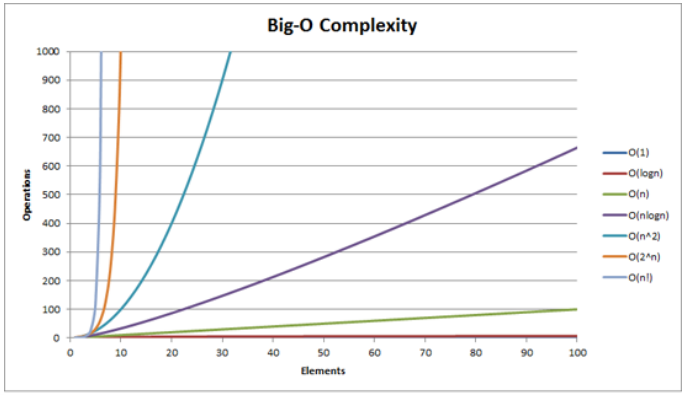
All subsets without last element

Create a list of just last element

For all smaller solutions, make one that also has last element

Combine sublists with last element and those without last element

EXPONENTIAL COMPLEXITY




```
def genSubsets(L):
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

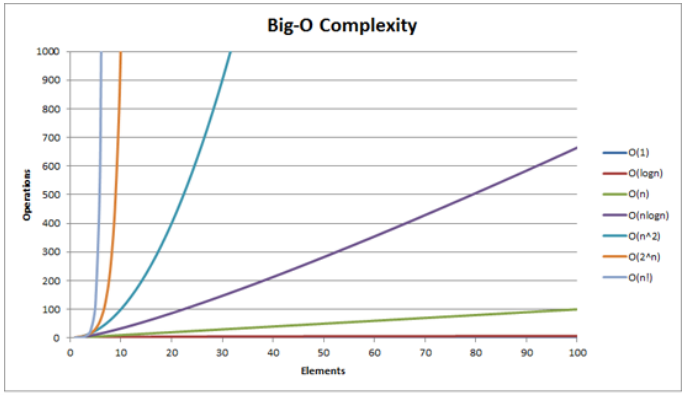
assuming append is constant time

time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

latter step will be linear in size of smaller solution, so cost depends on size of solution

question is how big is smaller solution?

EXPONENTIAL COMPLEXITY




```
def genSubsets(L):
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

important to think about size of smaller

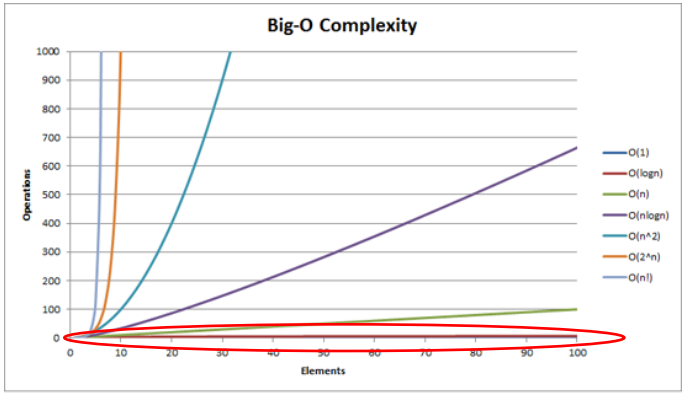
know that for a set of size k there are 2^k cases (note that to generate all subsets have choice of whether to include each element)

so to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps

math tells us this is $O(2^n)$

LOGARITHMIC COMPLEXITY

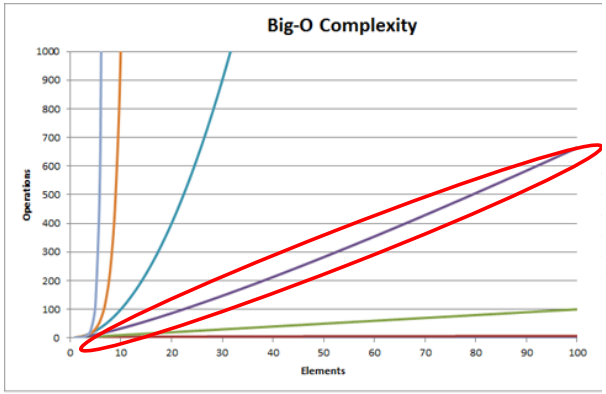
LOG COMPLEXITY – O(log n)



- complexity grows as log of size of one of its inputs
- characterized by algorithms where we cut the size of the problem in half at each step
- example: one **bisection search** implementation
- example: **binary search** of a list

LOG LINEAR COMPLEXITY

LOG LINEAR COMPLEXITY – O(n log n)



- many practical programs are log linear
- these grow as O(n log n) – a bit more costly than linear, but much better than quadratic or exponential
- commonly used log linear algorithm is **merge sort**
- will see this in a few minutes

SEARCHING ALGORITHMS

SEARCH ALGORITHMS

- search algorithm – method for finding an item or group of items with specific properties within a collection of items
- collection could be implicit
  - example – find square root as a search problem
    - exhaustive enumeration
    - bisection search
    - Newton-Raphson
- collection could be explicit
  - example – is a student record in a stored collection of data?
  - example – retrieving a Web page containing a set of key words

SEARCH ALGORITHMS

- linear search
  - **brute force** search (aka British Museum algorithm)
  - list does not have to be sorted
  - saw this last time
  - O(n) where n = len(L)
- bisection search (aka binary search)
  - list **MUST be sorted** to give correct answer
  - saw two different implementations of the algorithm
    - O(n log n) if copy sublists
    - O(log n) if pass in pointers to locations in lists

AMORTIZED COST ANALYSIS

SEARCHING A SORTED LIST – n is len(L)

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
  - **but** assumes the **list is sorted**!
- when does it make sense to **sort first then search**?
  - SORT + O(`log n`) < O(`n`)
  - → SORT < O(`n`) – O(`log n`)
  - so worthwhile when cost of sorting is less than O(`n`)
- **NEVER TRUE!**
  - **To sort a collection of n elements must look at each one at least once!**

AMORTIZED COST – n is len(L)

- why bother sorting first?
- in some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- for K searches, we want

SORT + `K`*O(`log n`) < `K`*O(`n`)

→ for large `K`, **SORT time becomes irrelevant**, if cost of sorting is small enough

- can we find a way to sort list fast enough to make this worthwhile?

SORT ALGORITHMS

SORT ALGORITHMS

- Want to efficiently sort a list of entries (typically numbers, or other entities that can easily be compared)
- Will see a range of methods, including one that is very efficient

MONKEY SORT

- aka bogosort, stupid sort, slowsort, permutation sort, shotgun sort
- to sort a deck of cards
  - throw them in the air
  - pick them up
  - are they sorted?
  - repeat if not sorted

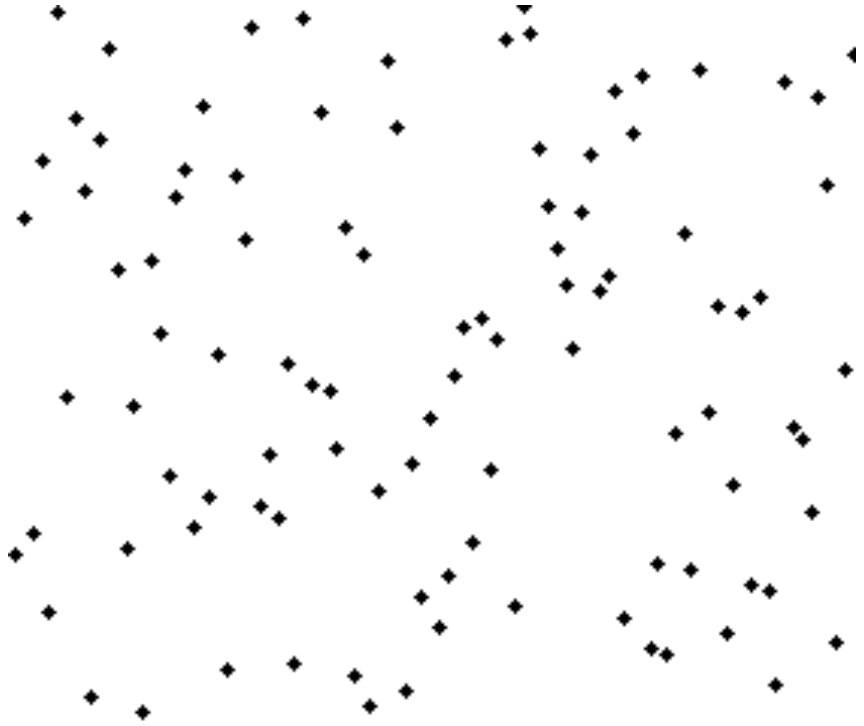
COMPLEXITY OF BOGO SORT

```
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- best case: **O(n) where n is len(L)** to check if sorted
- worst case: **O(?)** – it is **unbounded** if really unlucky; **$O(n^n)$** if no duplicate cases

BUBBLE SORT

- **compare consecutive pairs** of elements
- **swap elements** in pair such that smaller is first
- when reach end of list, **start over** again
- stop when **no more swaps** have been made
- largest unsorted element always at end after pass, so at most n passes



CC-BY Hydrargyrum
https://commons.wikimedia.org/wiki/File:Bubble_sort_animation.gif

COMPLEXITY OF BUBBLE SORT

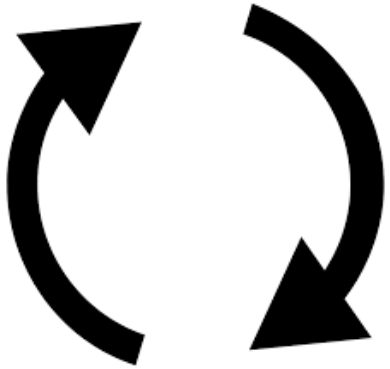
```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

O(len(L))

O(len(L))

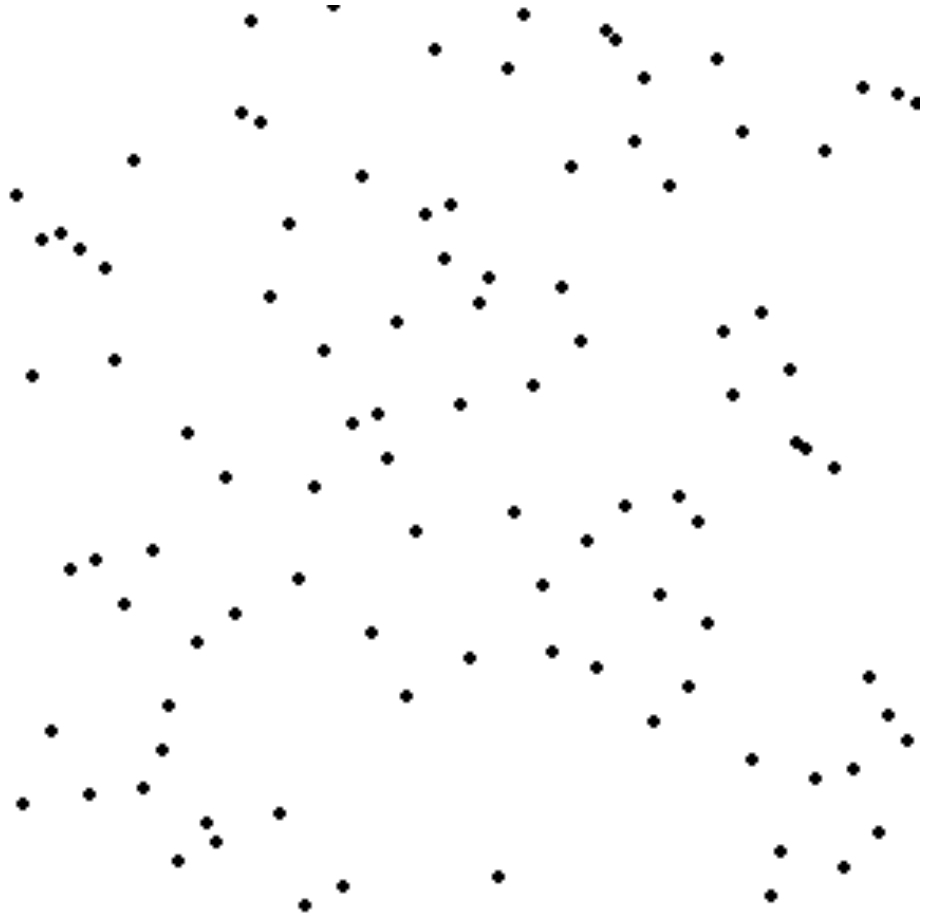
- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **$O(n^2)$ where n is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes

SELECTION SORT

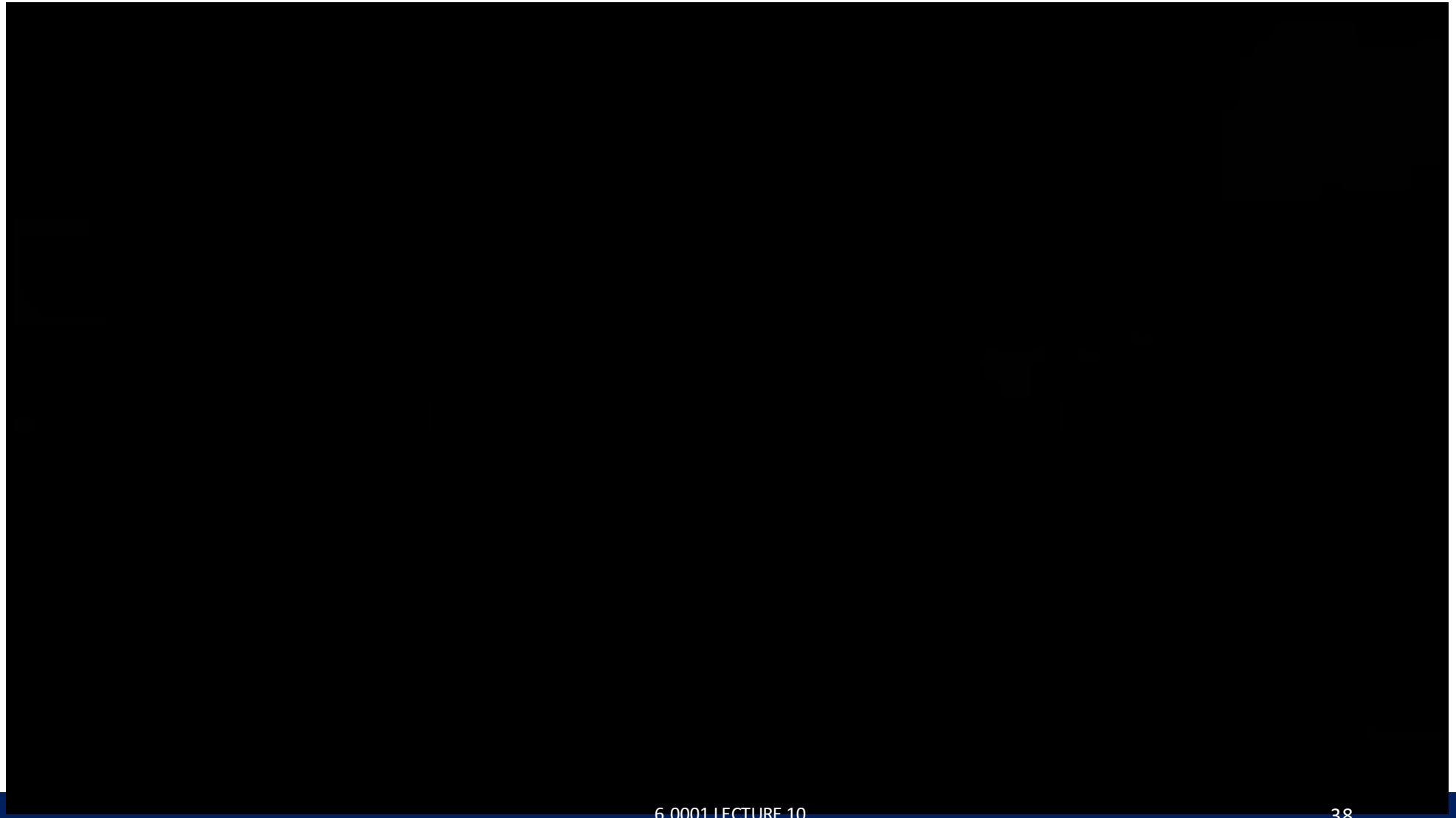


- first step
  - extract **minimum element**
  - **swap it** with element at **index 0**
- subsequent step
  - in remaining sublist, extract **minimum element**
  - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
  - at i^{th} step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

SELECTION SORT DEMO



SELECTION SORT WITH MIT STUDENTS



COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

len(L) times → O(len(L))

len(L) – suffixSt times → O(len(L))

- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

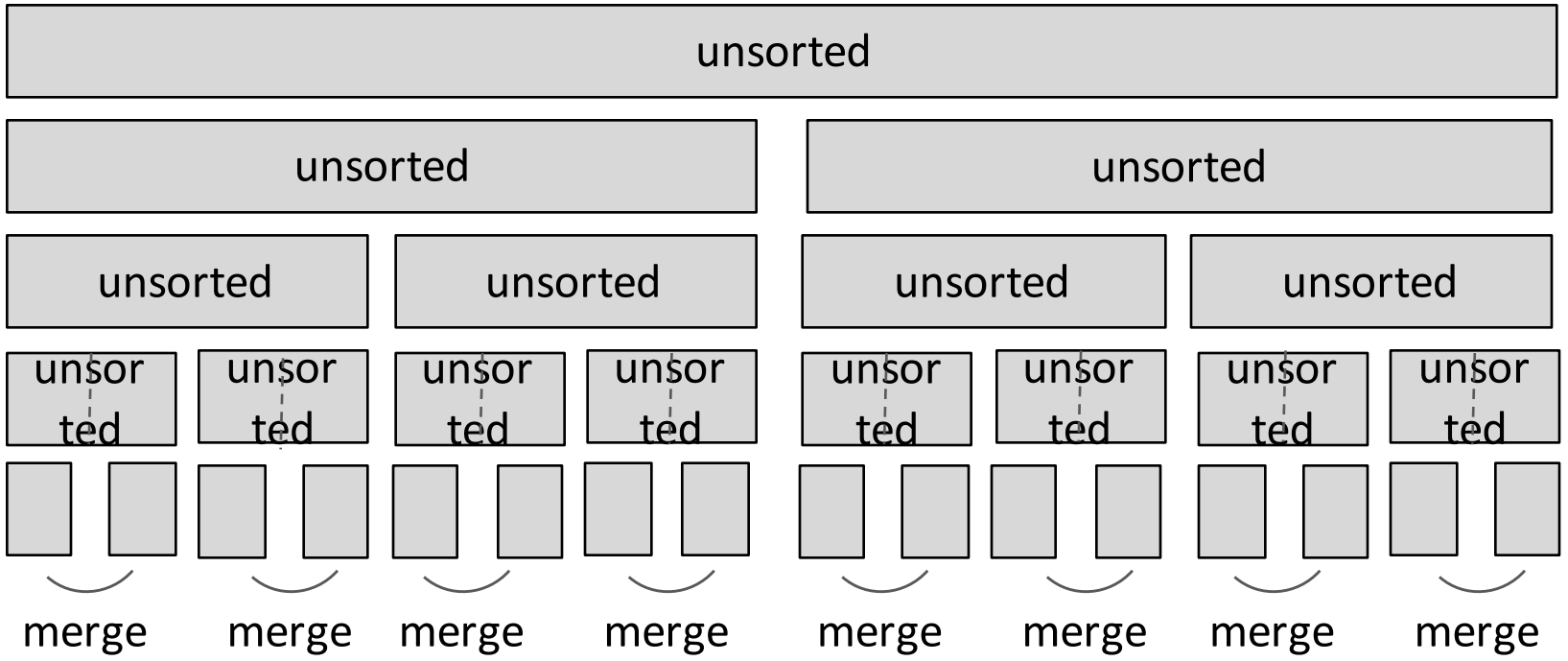
5 minute break

MERGE SORT

- use a divide-and-conquer approach:
  1. if list is of length 0 or 1, already sorted
  2. if list has more than one element, split into two lists, and sort each
  3. merge sorted sublists
     1. look at first element of each, move smaller to end of the result
     2. when one list empty, just copy rest of other list

MERGE SORT

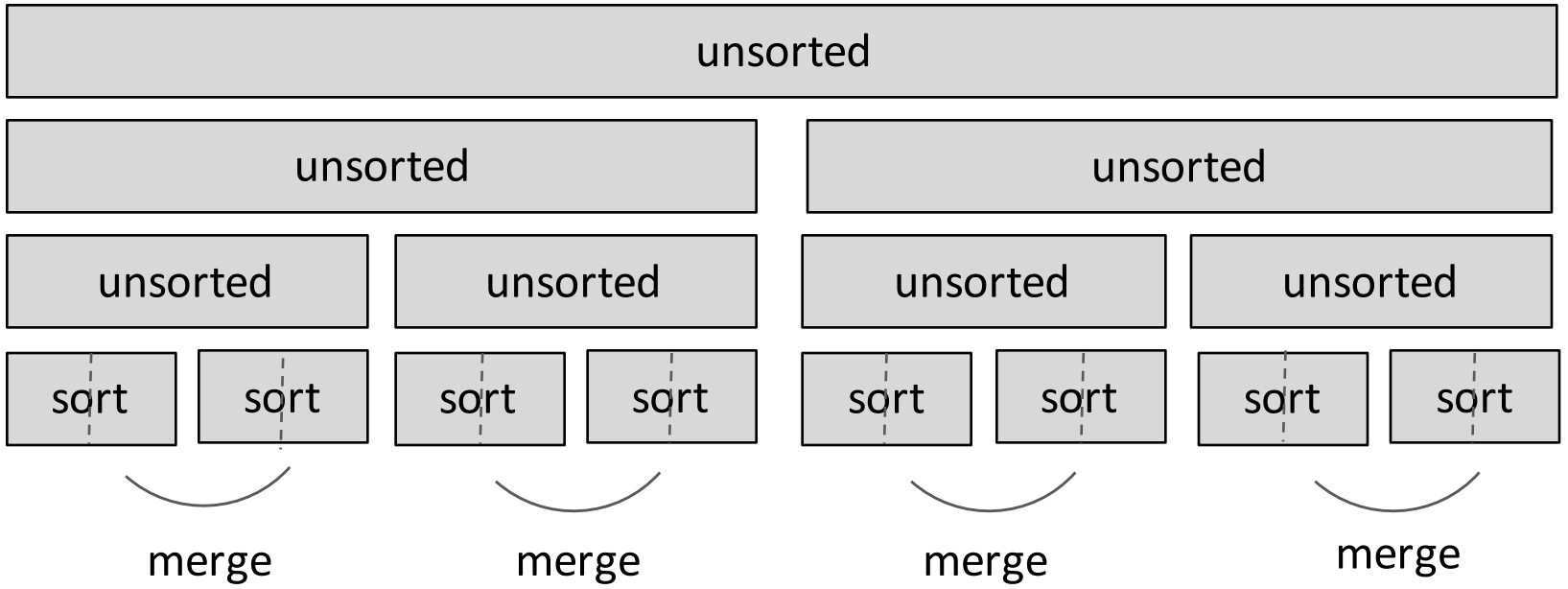
- divide and conquer



- **split list in half** until have sublists of only 1 element

MERGE SORT

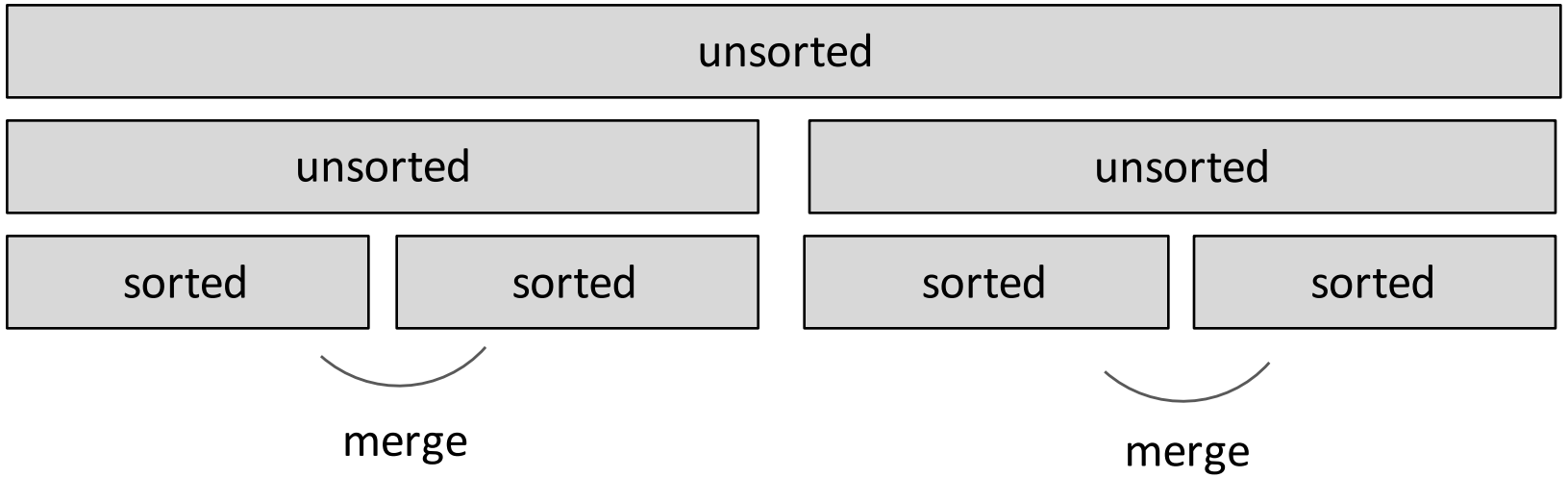
- divide and conquer



- merge such that **sublists will be sorted after merge**

MERGE SORT

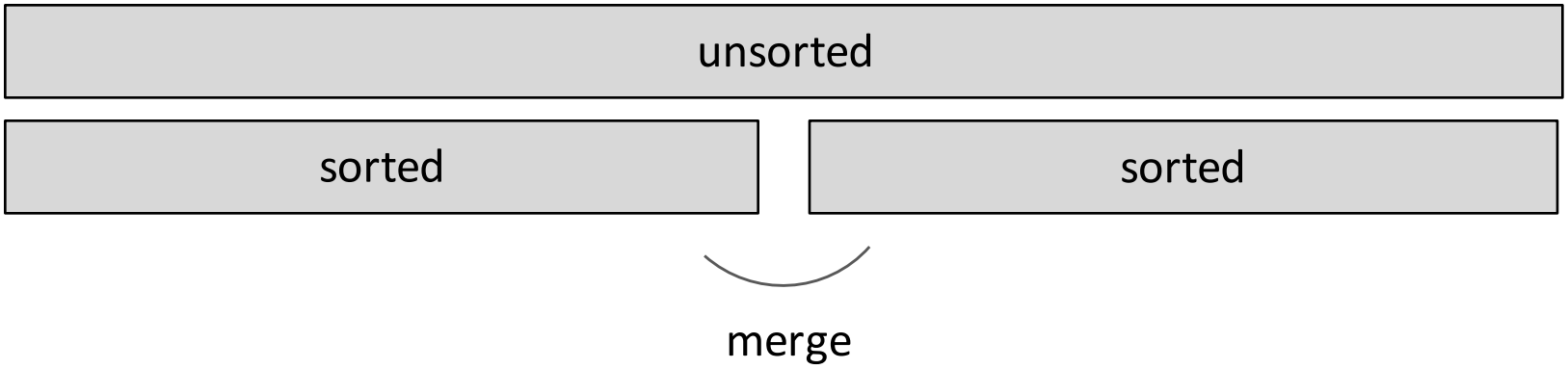
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer – done!

sorted

MERGE SORT DEMO

MERGE SORT WITH MIT STUDENTS

EXAMPLE OF MERGING

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

MERGING SUBLISTS STEP

```
def merge(left, right):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- left and right sublists are ordered
- move indices for sublists depending on which sublist holds next smallest element

when right sublist is empty

when left sublist is empty

COMPLEXITY OF MERGING SUBLISTS STEP

- go through two lists, only one pass
- compare only **smallest elements in each sublist**
- O(len(left) + len(right)) copied elements
- O(len(longer list)) comparisons
- **linear in length of the lists**

MERGE SORT ALGORITHM – RECURSIVE

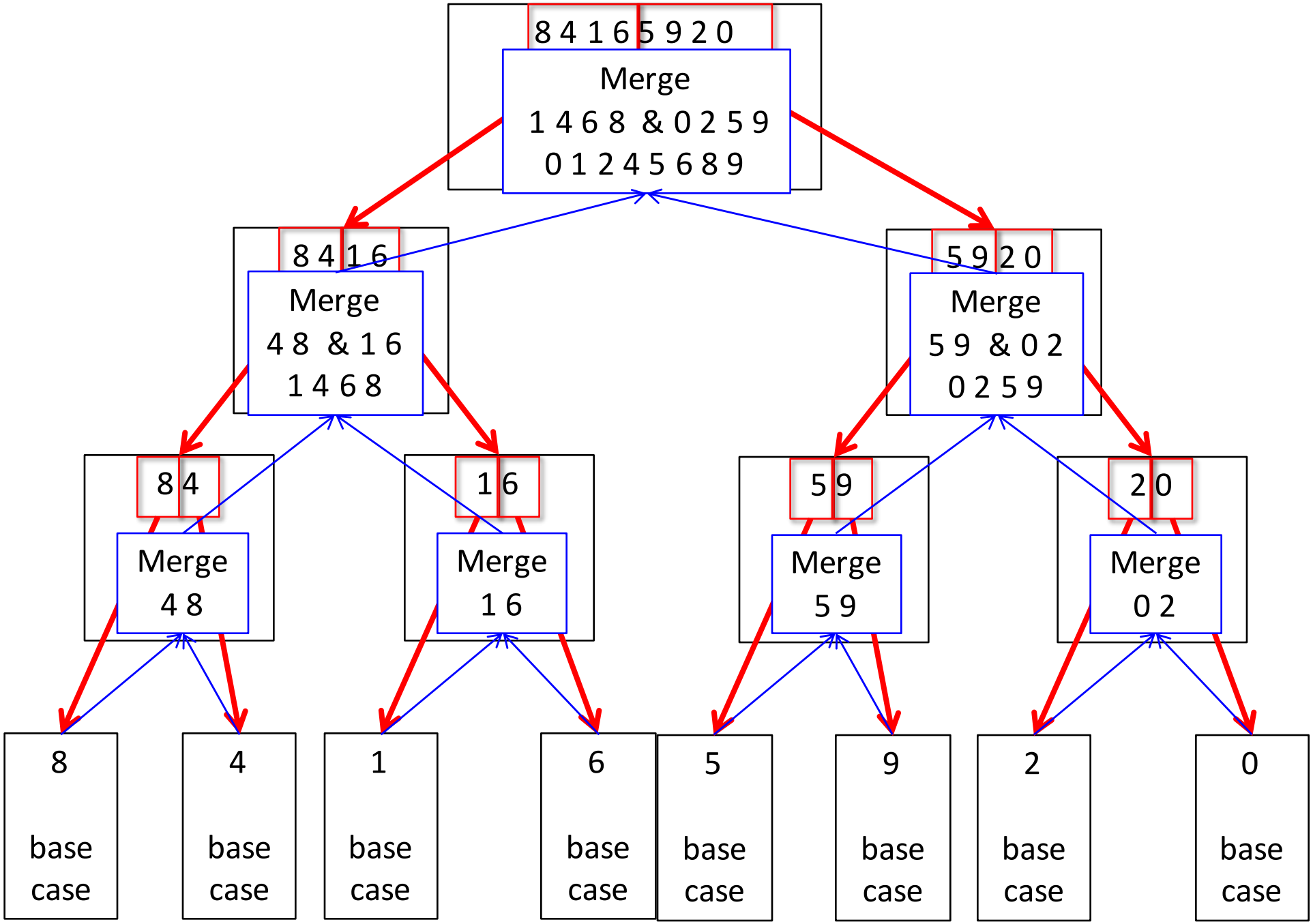
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

base case

divide

conquer with the merge step

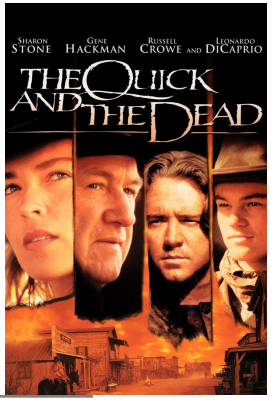
- **divide list** successively into halves
- depth-first such that **conquer smallest pieces down one branch** first before moving to larger pieces


8 4 1 6 5 9 2 0
Merge
1 4 6 8 & 0 2 5 9
0 1 2 4 5 6 8 9
8 4 1 6
Merge
4 8 & 1 6
1 4 6 8
5 9 2 0
Merge
5 9 & 0 2
0 2 5 9
8 4
Merge
4 8
1 6
Merge
1 6
5 9
Merge
5 9
2 0
Merge
0 2
8
base case
4
base case
1
base case
6
base case
5
base case
9
base case
2
base case
0
base case

COMPLEXITY OF MERGE SORT

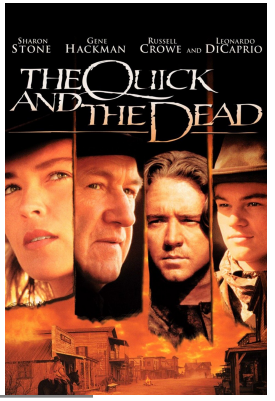
- at **first recursion level**
  - n/2 elements in each list; one merge
  - merge in worst case looks at each element
  - O(n) where n is len(L)
- at **second recursion level**
  - n/4 elements in each list; two merges
  - each merge in worst case looks at n/2 elements
  - two merges → O(n) where n is len(L)
- each recursion level is O(n) where n is len(L) – cost of each level
- **dividing list in half** with each recursive call – number of levels
  - O(log(n)) where n is len(L)
- overall complexity is **O(n log(n)) where n is len(L)**

QUICKSORT



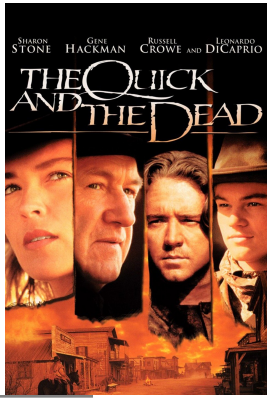
- Select a first element, call it the *pivot*.
- Want to group elements so elements to the left of the pivot are smaller than the pivot and elements to the right of the pivot are larger than the pivot.
- Maintain two pointers *left* and *right*. The *left* pointer points to the first element after the pivot, call this element *lelt.* The *right* pointer points to the farthest element on the right side of the list, call this element *relt*.
- At each step, compare *lelt* with *pivot* and *relt* with *pivot.* Want *lelt* < *pivot* and *relt* > *pivot*. If these conditions are not satisfied, *lelt* and *relt* are swapped.
- Otherwise, *left* pointer is increased by 1 and *right* pointer is decreased by 1.
- When *left* >= *right*, the *pivot* is swapped with either the *lelt* or *relt.* The *pivot* element will be in its correct position.
- Continue to quick sort the left half and the right half of the list.

QUICKSORT



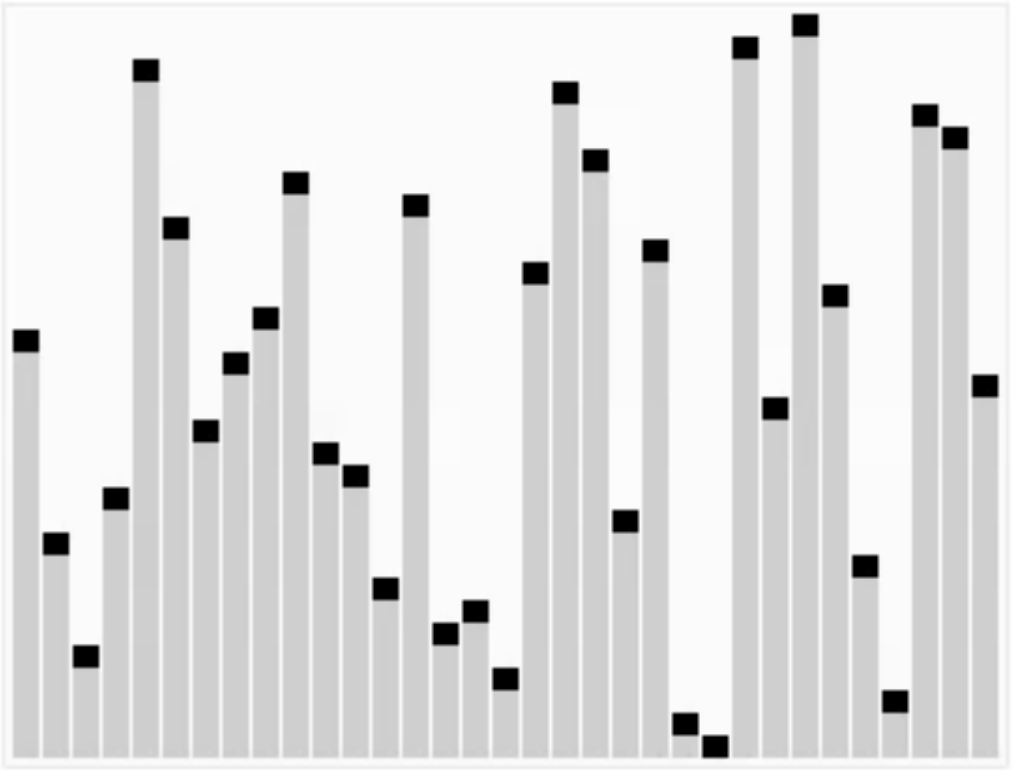
1. Choose a pivot
2. Set a *left* pointer and *right* pointer
3. Compare the *left* pointer element (*lelt)* with the pivot and the *right* pointer element (*relt)* with the *pivot*.
4. Check if *lelt<pivot* and *relt>*pivot:
   1. If yes, increment the *left* pointer and decrement the *right* pointer
   2. If not, swap *lelt* and *relt*
5. When *left >= right,* swap the *pivot* with either *left* or *right* pointer.
6. Repeat steps 1 - 5 on the left half and the right half of the list till the entire list is sorted.

QUICKSORT

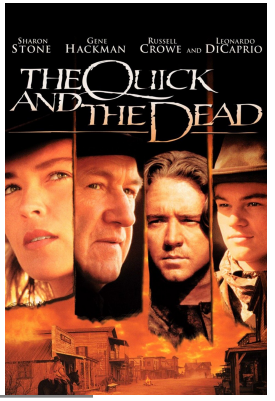


Animated visualization of the quicksort algorithm. This visualization is using the last element of a list as the pivot, but the idea is the same as our description.

The horizontal lines are pivot values.

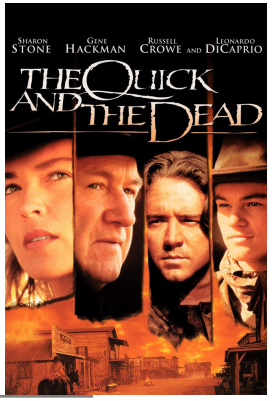


QUICKSORT



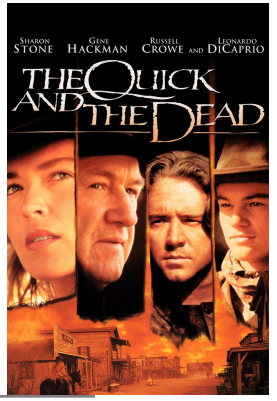

```
def quick_sort(L):
    quick_sort_help(L,0,len(L)-1)

def quick_sort_help(L,first,last):
    if first<last:
        splitpoint = partition(L,first,last)
        quick_sort_help(L,first,splitpoint-1)
        quick_sort_help(L,splitpoint+1,last)
```

QUICKSORT (cont)




```
def partition(L,first,last):
  pivotvalue = L[first]
  leftmark = first+1
  rightmark = last
  done = False
  while not done:
      while leftmark <= rightmark and L[leftmark] <= pivotvalue:
          leftmark = leftmark + 1
      while L[rightmark] >= pivotvalue and rightmark >= leftmark:
          rightmark = rightmark -1
      if rightmark < leftmark:
          done = True
      else:
          temp = L[leftmark]
          L[leftmark] = L[rightmark]
          L[rightmark] = temp
  temp = L[first]
  L[first] = L[rightmark]
  L[rightmark] = temp
  return rightmark
```

QUICKSORT COMPLEXITY



- Choosing the *pivot* is very crucial as it determines the complexity of this algorithm.
- The worst choice for *pivot* is the smallest or the largest value in the list, as one partition is empty, and the other is O(n). If this is true for all sublists, then the complexity is $O(n^2)$.
- To avoid this, several methods are used to pick the *pivot* value. In the median method, we look at the first, last, and middle element, and pick the median value as the pivot.
- Typically in this method, the average complexity is O(n log(n)).
- Unlike Merge Sort, Quicksort doesn't use extra memory or space.

SORTING SUMMARY -- n is len(L)

- bogo sort
  - randomness, unbounded O()
- bubble sort
  - $O(n^2)$
- selection sort
  - $O(n^2)$
- merge sort
  - O(n log(n))
- quick sort
  - $O(n^2)$ – worst case
  - O(n log n) – average case
- **O(n log(n)) is the fastest a sort algorithm can be**

6.0001 SUMMARY

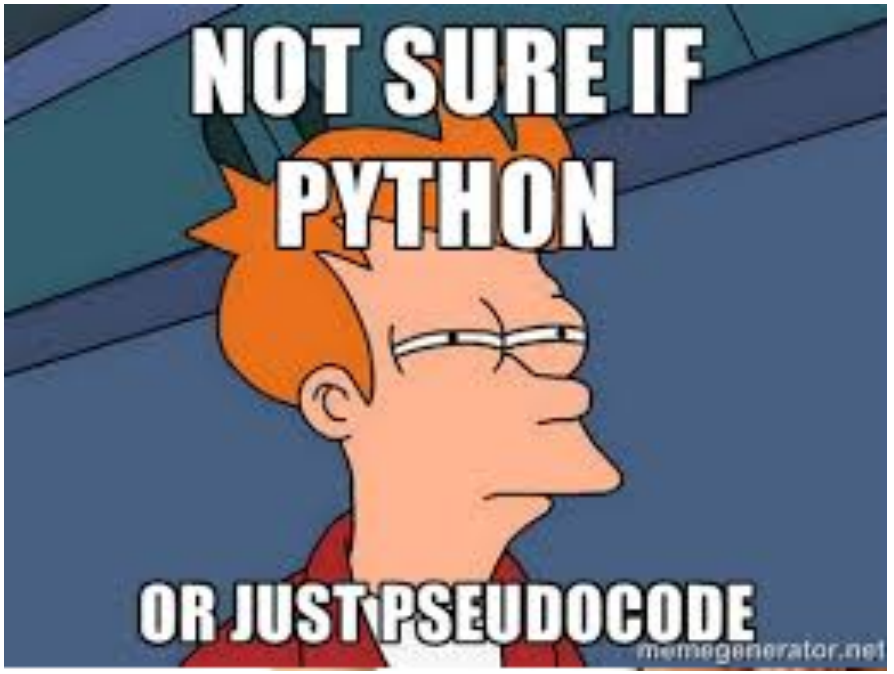
Ana Bell

WRAP-UP OF 6.0001

1. Python **language**
2. How to **write** and **debug** fairly complicated programs
3. How to **analyze** the run-time **complexity** of programs
4. Some simple **algorithms**

PYTHON

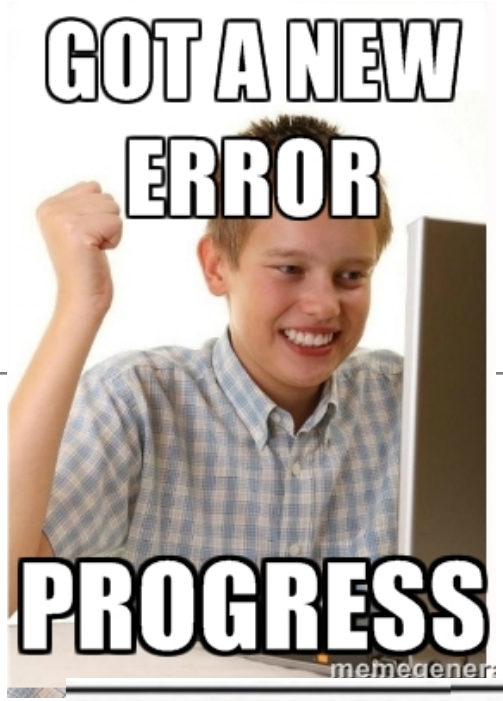
- We used Python to introduce you to programming
  - Simple **syntax**
  - Certain details you do not need to worry about
  - Many **online resources** and debugging tools

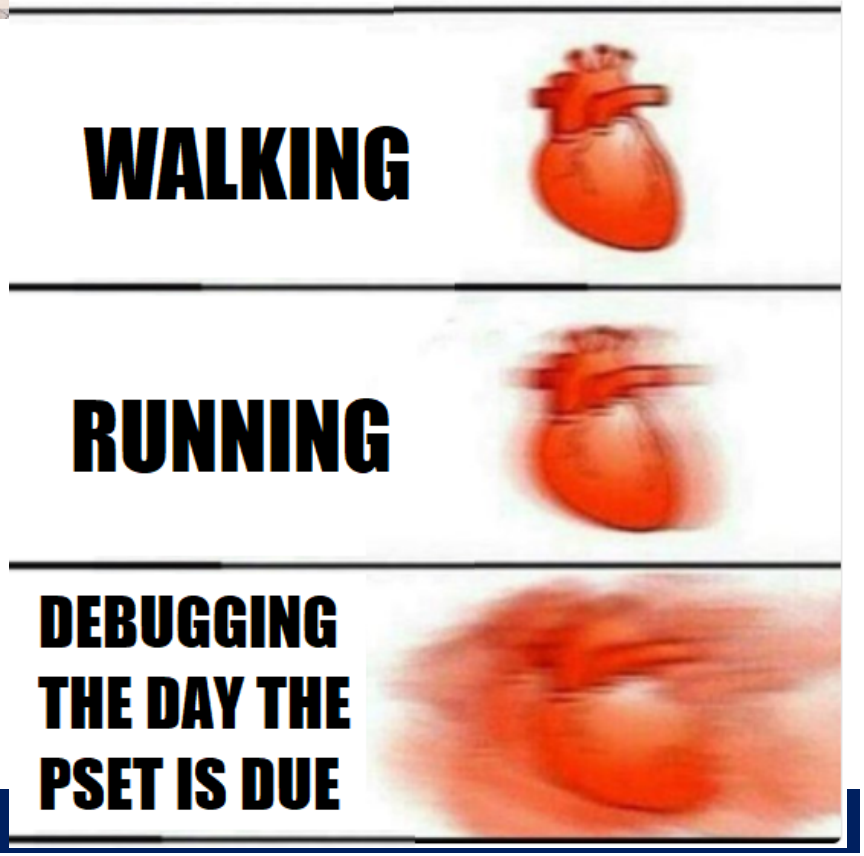


- Python is a **popular** language
  - Used in many MIT **courses**
  - In demand for many **industry** jobs
  - Many people contribute interesting and useful **packages** (bio, finance, video games…)

WRITING AND DEBUGGING

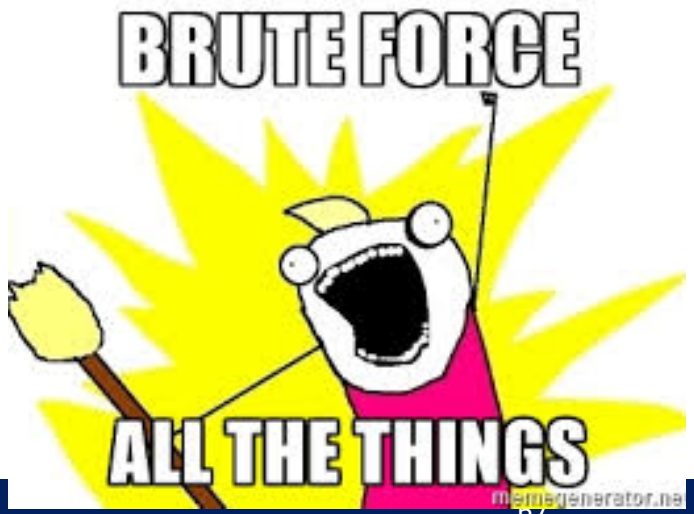
- **Decomposition** and **abstraction**
  - Returned to these ideas time and time again (functions, debugging, classes)
- Structure your code
  - **Think** before you write code
  - Write **modular code** from the start – easy to debug
- Use **comments** and **docstrings**
  - Document your ideas
  - Write code that works then try to improve





ALGORITHMS

- **Different kinds** of algorithms
  - Iterative, recursive
  - Brute force, exhaustive enumeration, guess-and-check, successive approximation
- **Complexity** of algorithms
  - Constant, linear, logarithmic, polynomial
  - Big Oh notation
- Specific algorithms for **searching** and **sorting**
  - Bisection/binary search
  - Selection sort, merge sort



NEXT STEPS

- Final exam next week
  - Motivation → look at the first problem sets, you've come a long way in just a few weeks!
- Continue to practice
  - Stellar->Materials has some websites for fun exercises
  - Write programs to do work for you, for example write a program to schedule your classes, organize group activities

- Other courses:

6.0002 (Comp Thinking and Data Sci)

6.009 (Prog. Fundamentals)
6.031 (Software Constr, Java)
6.036 (Mchine Learning)
6.006 (Algorithms)
6.01 (EECS Intro)
6.004 (Comp. Structures)

Course evaluations

- For those of you enrolled in 6.0001, the web site for course evaluations is now open
- You have until Monday, October 15 at 9 am to complete the surveys.
- You can access the evaluation here: https://registrar.mit.edu/subjectevaluation
- MIT encourages you to be fair, thoughtful, polite and objective in your responses and to adhere to the community standards put forth in the Mind and Hand Book (section II.11) - https://handbook.mit.edu/