PROGRAM EFFICIENCY

John Guttag

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

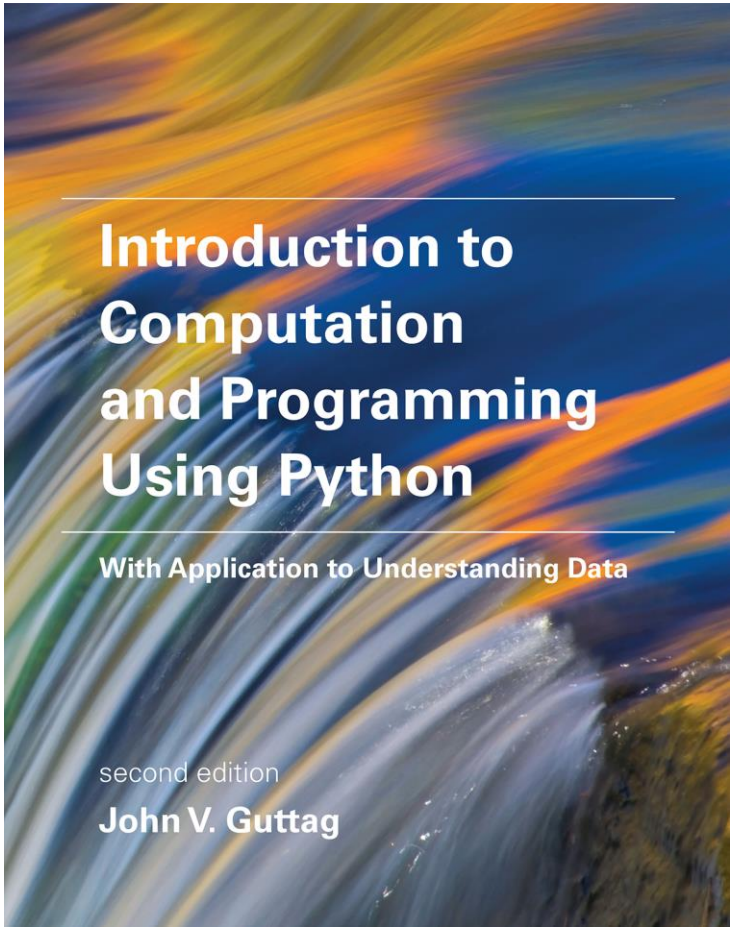
6.0001 LECTURE 9

Today

- Formally evaluate programs
- Efficiency in time
- Orders of growth
- Examples of different complexity cases
- Some important algorithms
- Lists and indirection

Assigned Reading

- Today
  - Chapter 9
  - 10.1 – 10.2
- Monday
  - 10.3
  - Chapter 11

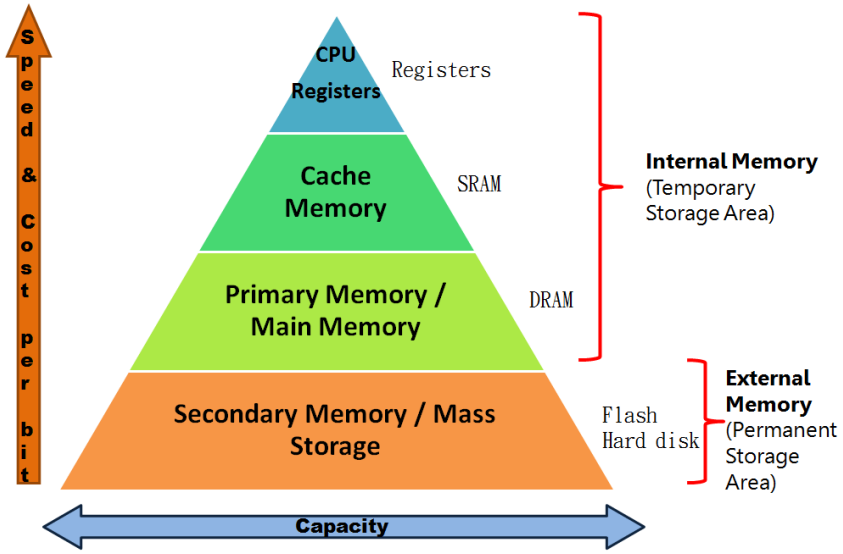


Efficiency Is Important

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - Want to separate implementation from choice of more abstract **algorithm**
  - Not always easy to separate memory usage from time

Time and Space Are Related

- Often a tradeoff, can decrease one at the expense of the other
- From a practical perspective, space way more complicated
  - Often a threshold function
    - Doesn't matter how much use as long as it less than X
- Time more straightforward, and usually more important
  - Modern machines have memory hierarchies
    - Often a dramatic impact on performance



Cache: 10's cycles
Main memory: 100's cycles
Secondary memory: 1,000,000's cycles

EVALUATING ALGORITHMS

- Focus on idea of counting operations in an algorithm, but **not worry about small variations in implementation**
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- Look at the **worst case asymptotic run time** of a program, as the input grows to a large value

Measuring Order of Growth

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth
- **Big Oh or O()** is used to describe worst case
  - Worst case tends to occur 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
- Based on **counting operations**
  - Need to know how much to "charge" for each builtin operation

Complexity of Some Python Operations

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

A Technicality

- When we say that the complexity of f is O(n), we mean that its asymptotic growth is **not worse than** linear in n
- It is an upper bound, not necessarily a **tight bound**
- In practice, we are usually looking for something close to a tight bound
  - **Upper bound**, worst case not worse than
  - **Lower bound**, worst case not better than

Big $\boldsymbol{\theta}$ (Theta)

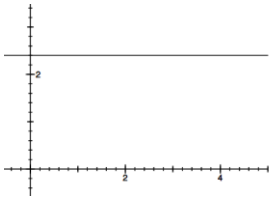
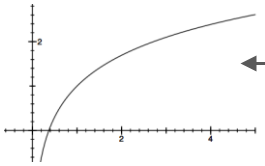
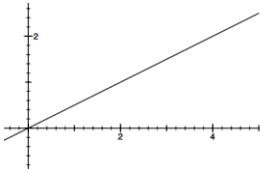
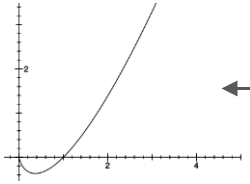
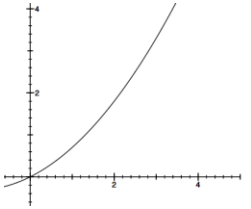
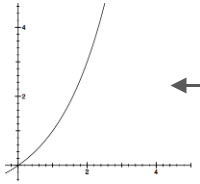
- When we have a tight bound, f(n), we can say that an algorithm is order $\boldsymbol{\theta}$(f(n))
- add_digits is **in** O(len(s))
  - No worse than linear in len(s)
  - Perhaps faster
  - Also technically correct to say in O(len(s)^2) (but not helpful)
- add_digits **is** order $\boldsymbol{\theta}$(len(s))
  - No worse than linear in len(s)
  - No better than linear in len(s)

```
def add_digits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

Most people a bit sloppy

Interpret is "is O(n)" to Imply upper and lower bound

COMPLEXITY CLASSES ORDERED LOW TO HIGH

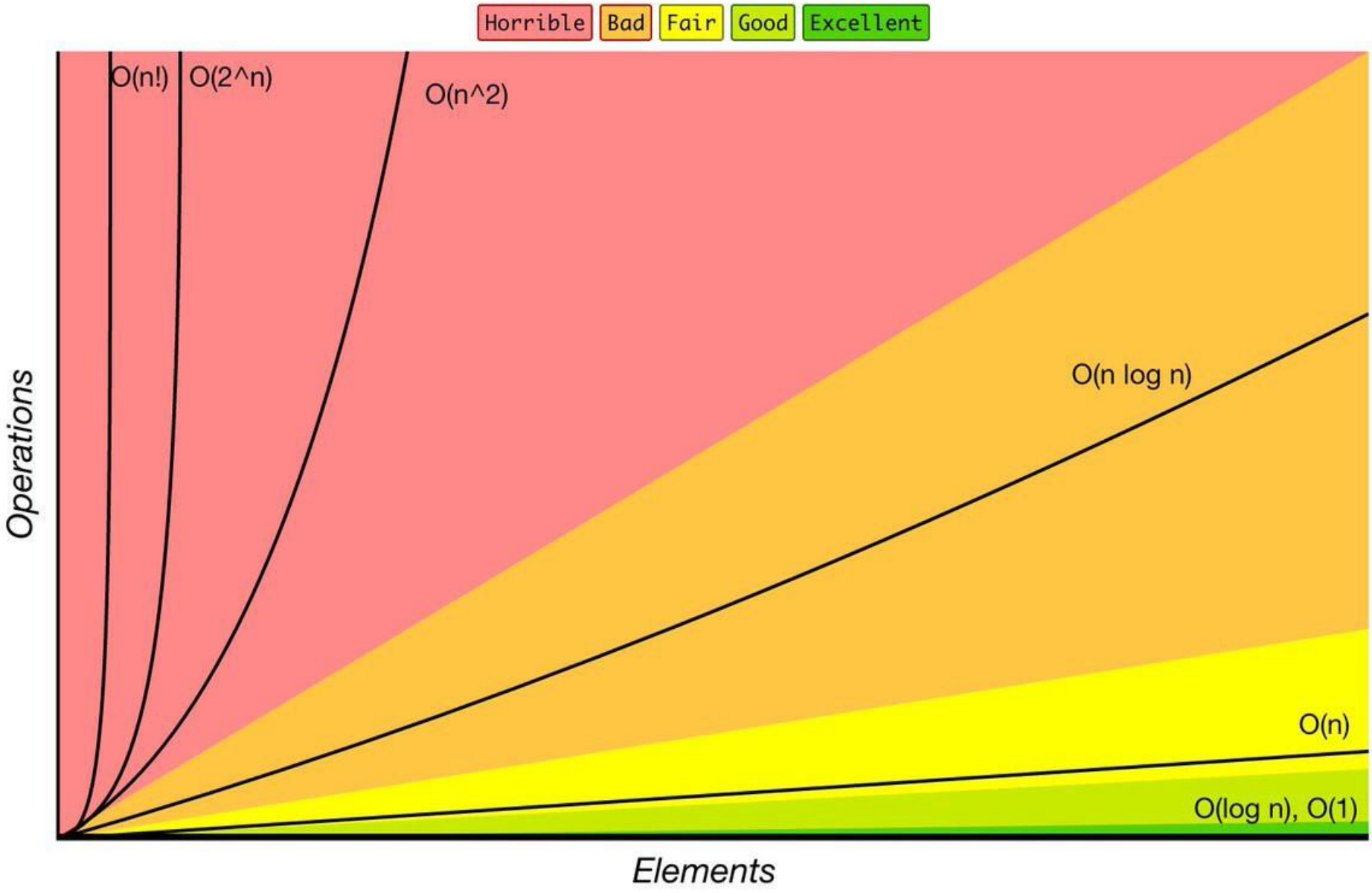
`O(1)`	:	constant →	
`O(log n)`	:	 ← logarithmic	
`O(n)`	:	linear →	
`O(n log n)`	:	 ← loglinear	
$O(n^c)$	:	polynomial →	
$O(c^n)$	:	 ← exponential	

c is a constant

COMPLEXITY GROWTH

CLASS	N = 10	N = 100	N = 1000	N = 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
$O(n^2)$	100	10000	1000000	1000000000000
$O(2^n)$	1024	1267650600228229401496703205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493193783881569585812759467291755314682518714528569231404359845775746985748039345677748242309854210746050623711418779541821530464749835819412673987675591655439460770629145711964776865421676604298316526243868372056680693 76	Good Luck!!

Big-O Complexity Chart



CONSTANT COMPLEXITY

CONSTANT COMPLEXITY

- Complexity independent of inputs
- Very few interesting algorithms in this class, but can often have pieces that fit this class
- Can have loops or recursive calls, but number of iterations or calls independent of size of input

CONSTANT COMPLEXITY: EXAMPLE 1

- Add x to y

```
def add(x, y):
    return x+y
```

- **O(1)**

LINEAR COMPLEXITY

LINEAR COMPLEXITY

- Simple **iterative loop** algorithms
- Loops must be a **function of input**
- Linear search a list to see if an element is present
- Recursive functions with one recursive call and constant overhead for call

LINEAR COMPLEXITY: EXAMPLE 1

- Add characters of a string, assumed to be composed of decimal digits

```
def add_digits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

Complexity expressed in terms of arguments to function

Not internal variables

Not n, unless n is an argument

- **O(len(s))**
- **O(n) where n is len(s)**

LINEAR COMPLEXITY: EXAMPLE 2

- Multiply x by y

```
def mul(x, y):
    tot = 0
    for i in range(y):
        tot += x
    return tot
```

What is complexity of mul?

- complexity in terms of x: **O(1)**
- complexity in terms of y: **O(y)**

LINEAR COMPLEXITY: EXAMPLE 3

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

fact_recur **O(n)** because the number of function calls is linear in n

- **Iterative and recursive factorial** implementations are the **same order of growth**
- If you time them, notice that fact_recur is slower than iterative version because of memory allocation

LINEAR SEARCH

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

speed up a little by returning True here, but speed up doesn't impact worst case

- Must look through all elements to decide it's not there
- O(len(L)) for the loop * O(1) to test if e == L[i]
- Overall complexity is **O(n) – where n is len(L)**
- **O(len(L))**

LINEAR SEARCH ON **SORTED** LIST

```
def search(L, e):
    for i in L:
        if i == e:
            return True
        if i > e:
            return False
    return False
```

- Must only look until reach a number greater than e
- O(len(L)) for the loop * O(1) to test if e == L[i]
- Overall complexity is **O(n) – where n is len(L)**

POLYNOMIAL COMPLEXITY

POLYNOMIAL COMPLEXITY (OFTEN QUADRATIC)

- Most **common polynomial algorithms are quadratic**, i.e., complexity grows with square of size of input
- Commonly occurs when we have **nested loops** or recursive function calls

QUADRATIC COMPLEXITY: EXAMPLE 1

```
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

- Computes n^2 very inefficiently
- When dealing with for loops, **look at the ranges**
- Nested loops, **each iterating n times**
- **$O(n^2)$**

QUADRATIC COMPLEXITY: EXAMPLE 2

- Find if L1 is a subset of L2 (if all elements in L1 are in L2)

```
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

QUADRATIC COMPLEXITY: EXAMPLE 2

```
def is_subset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Outer loop executed len(L1) times

Each iteration will execute inner loop up to len(L2) times

O(len(L1)*len(L2))

QUADRATIC COMPLEXITY: EXAMPLE 3

- Find intersection of two lists, return a list with each element appearing only once

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique
```

QUADRATIC COMPLEXITY: EXAMPLE 3

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    unique = []
    for e in tmp:
        if not(e in unique):
            unique.append(e)
    return unique
```

First nested loop takes **O(len(L1)*len(L2))** steps.

Second loop takes at most **O(len(L1)*len(L2))** steps. Typically not this bad.

Overall **O(len(L1)*len(L2))**

EXPONENTIAL COMPLEXITY

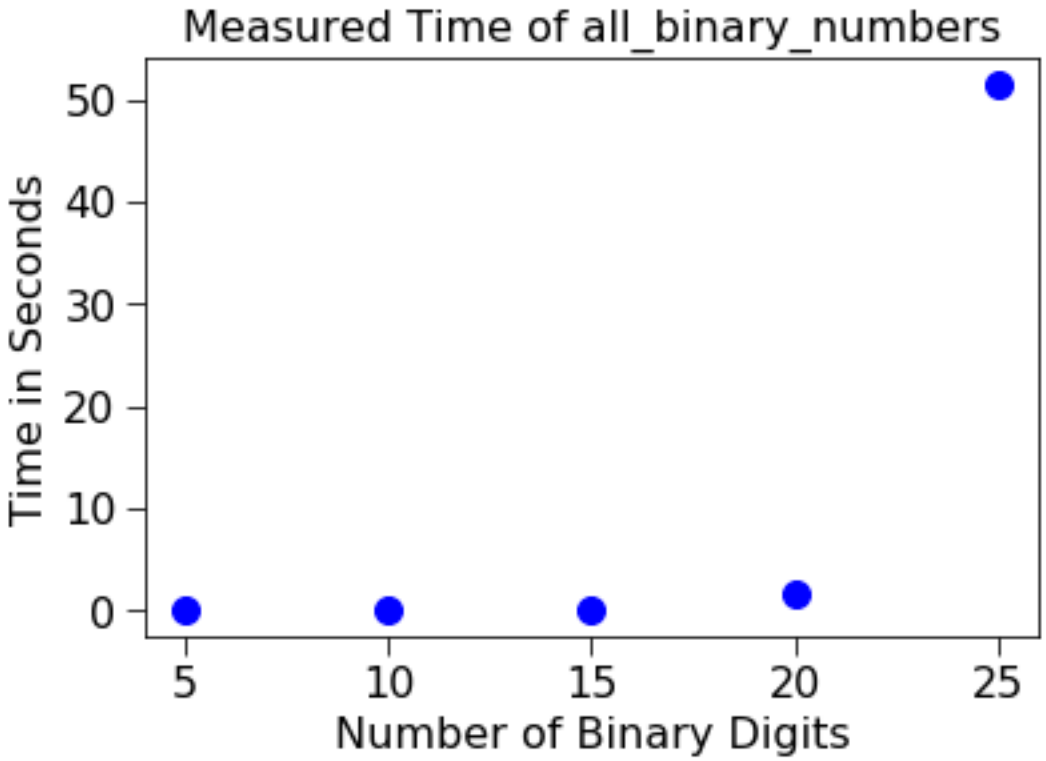
EXPONENTIAL COMPLEXITY

```
def all_binary_numbers(N):
    def helper (prefix, N):
        if N == 0:
            return [prefix]
        return helper(prefix+'0', N-1) + helper(prefix+'1', N-1)
    return helper('', N)
```

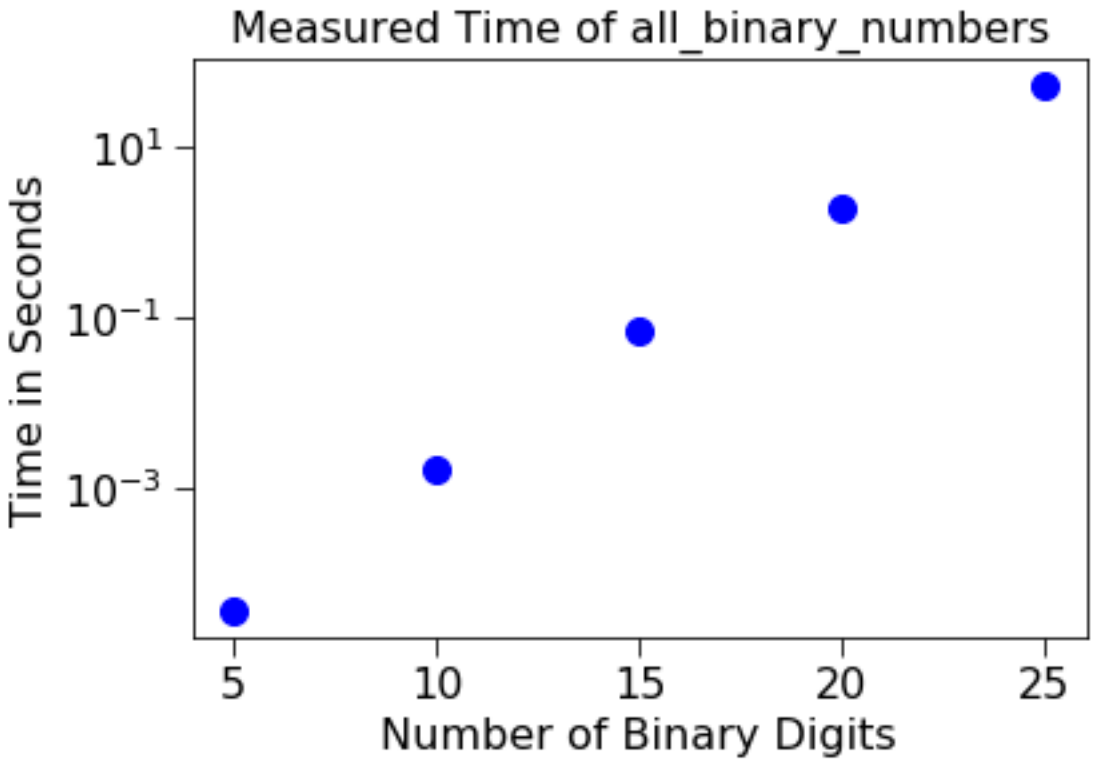
binary numbers of 5 digits took 3.504939377307892e-05 s
binary numbers of 10 digits took 0.0016635740175843239 s
binary numbers of 15 digits took 0.068629020359367 13 s
binary numbers of 20 digits took 1.9542624829337 s
binary numbers of 25 digits took 53.613449009135365 s

binary numbers of 5 digits took 3.523286432027817e-05 s
binary numbers of 10 digits took 0.0010585528798401356 s
binary numbers of 15 digits took 0.0496662026271224 s
binary numbers of 20 digits took 1.7410518480464816 s
binary numbers of 25 digits took 51.55056634778157 s

Exponential Complexity



Exponential Complexity



EXPONENTIAL COMPLEXITY

- Many important problems are inherently exponential
  - Will lead us to consider (mostly in 6.0002)
    - Approximate solutions
    - Algorithms that work well on special cases

TEST YOURSELF

```
def all_digits(nums):
    """ nums is a list of numbers """
    digits = [0,1,2,3,4,5,6,7,8,9]
    for i in nums:
        is_in = False
        for j in digits:
            if i == j:
                is_in = True
                break
        if not is_in:
            return False
    return True
```

O(1)?

O(len(nums))?

O(len(nums)*len(digits)?

O(len(nums)**2?

None of the above

BIG OH SUMMARY

- Compare efficiency of algorithms
  - notation that describes **growth relative to size of inputs**
  - **lower order of growth** is better
  - independent of machine or specific implementation

- Using Big Oh (or Θ)
  - describe order of growth
  - **asymptotic notation**
  - **upper bound**
  - **worst case** analysis

Not whole story
Sometimes constants do matter

5 Minute Break

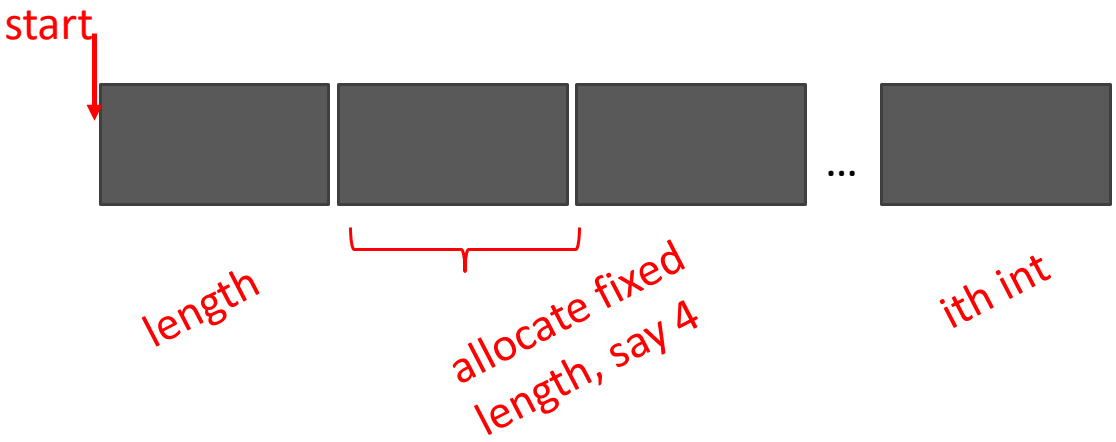
Complexity of Some Python Operations

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

Constant Time Indexing Into a List

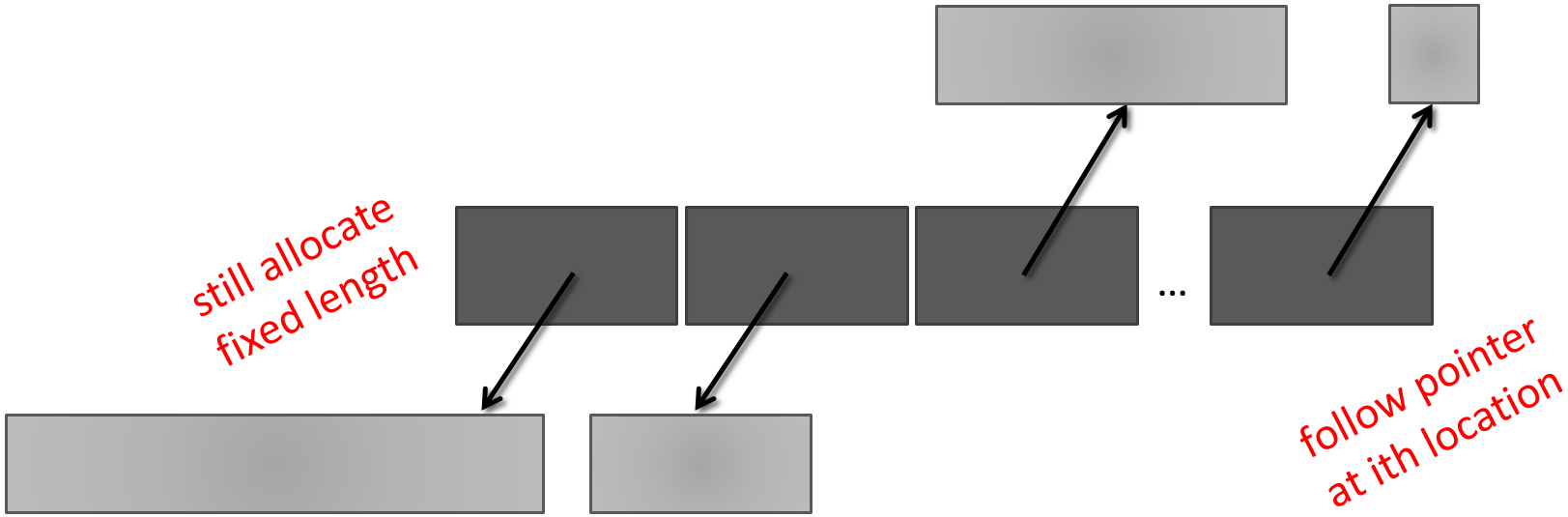
- If list is all 32 bit ints



- i^{th} element at location start + 4*i

Constant Time Indexing Into a List

- If list is heterogeneous
  - **Indirection**
  - References to other objects



What About Bisection Search?

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1(L[:half], e)
        else:
            return bisect_search1(L[half:], e)
```

constant O(1)

constant O(1)

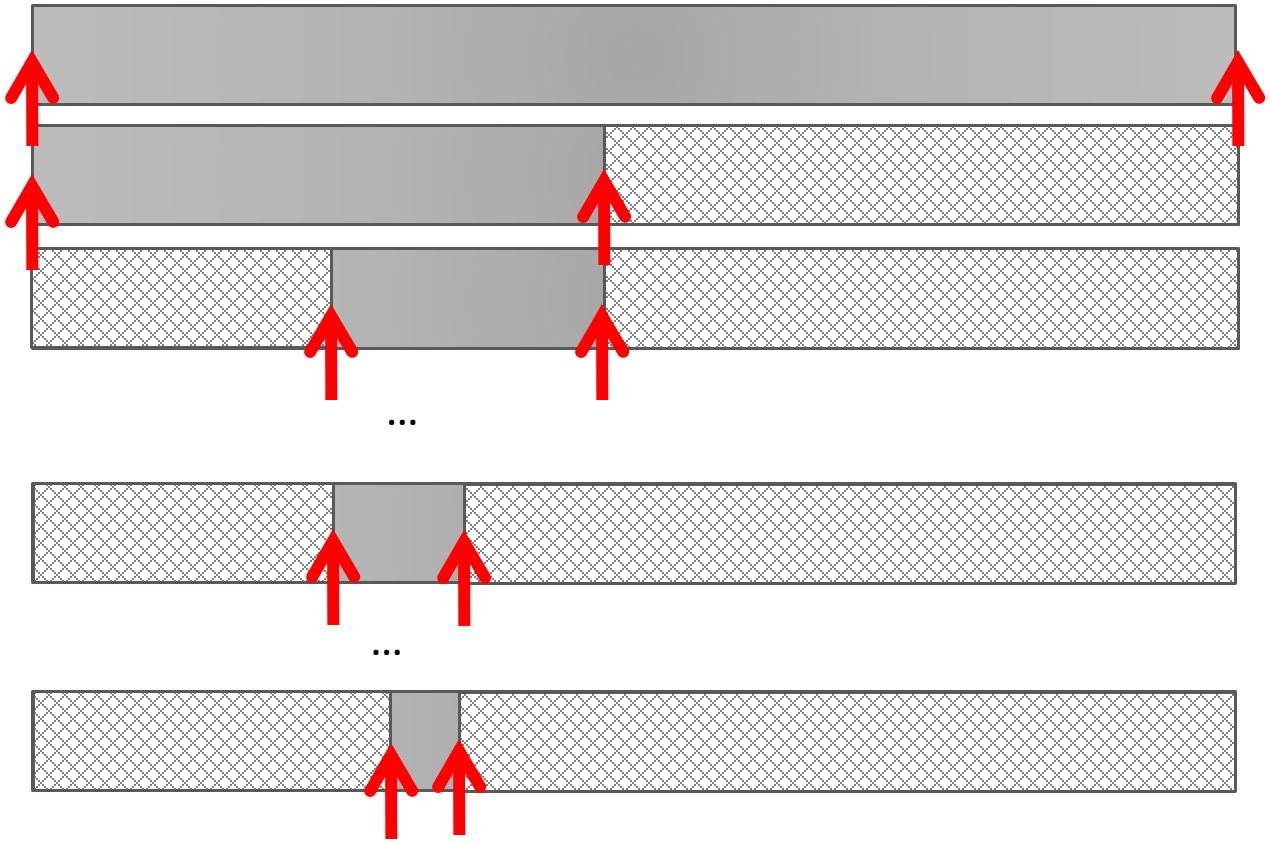
constant O(1)

NOT constant, copies list

NOT constant

NOT constant

A Better Way to Implement Bisection Search



- reduce size of problem by factor of 2 each step
- keep track of low and high indices to search list
- avoid copying list
- complexity of recursion is **O(log n) – where n is len(L)**

Amortized Analysis

- Why bother sorting first, considering that complexity of sorting is more than linear?
- In some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over `K` searches
- We want (sort time + `K`*O(`log n`)) < `K`*O(`n`)

  → for large `K`, **SORT time becomes irrelevant**
- How big does `K` have to be?
  - Depends upon complexity of sorting

Stupid Sort

```
def is_sorted(L):
    result = True
    for i in range(len(L) - 1):
        if L[i] > L[i+1]:
            result = False
    return result

def stupid_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

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 ith step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

Why It Works

- Maintains a **loop invariant**
  - given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix
    1. base case: prefix empty, suffix whole list – invariant true
    2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
    3. when exit, prefix is entire list, suffix empty, so sorted

Complexity Analysis

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

len(L) times → O(len(L))

len(L) – suffixSt times → O(len(L))

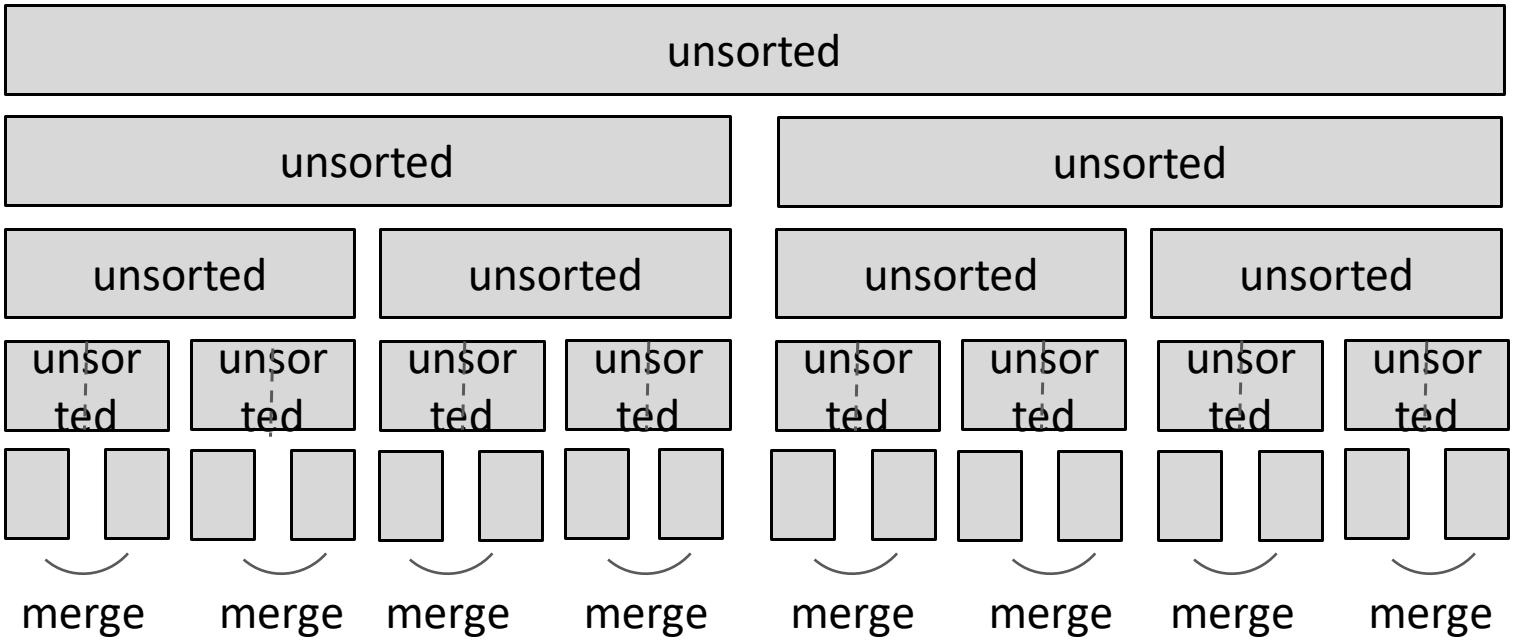
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

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

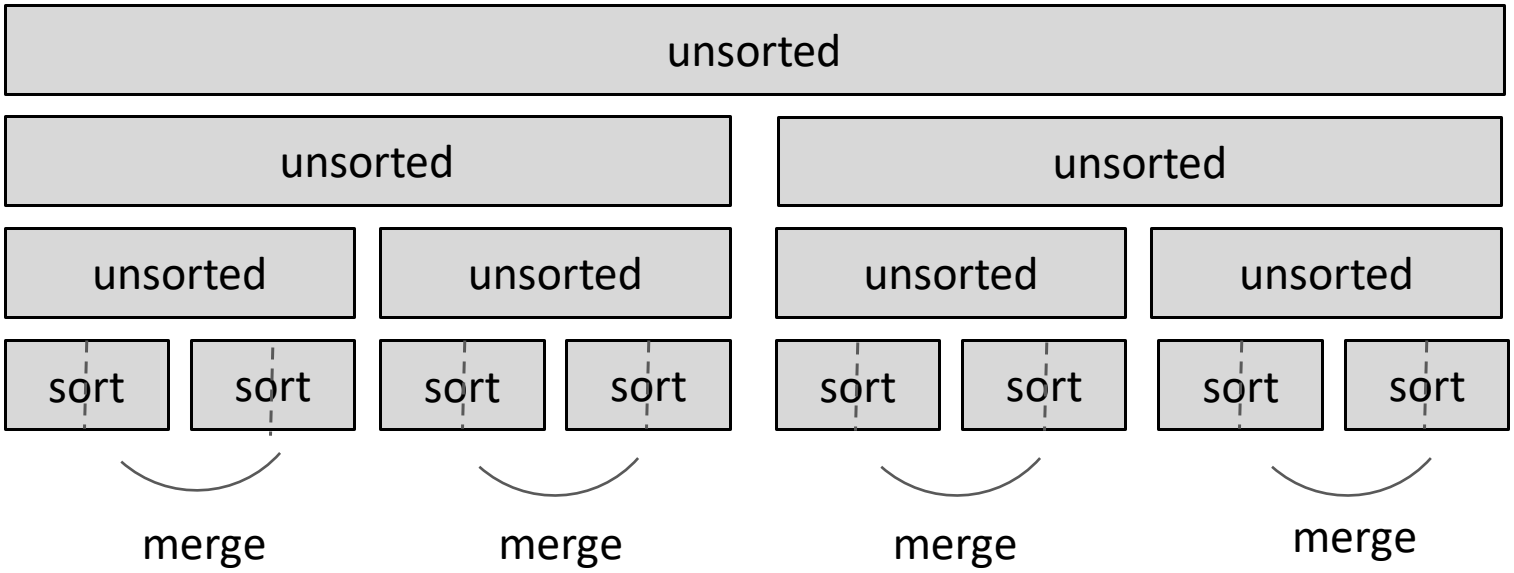
Merge Sort

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



MERGE SORT

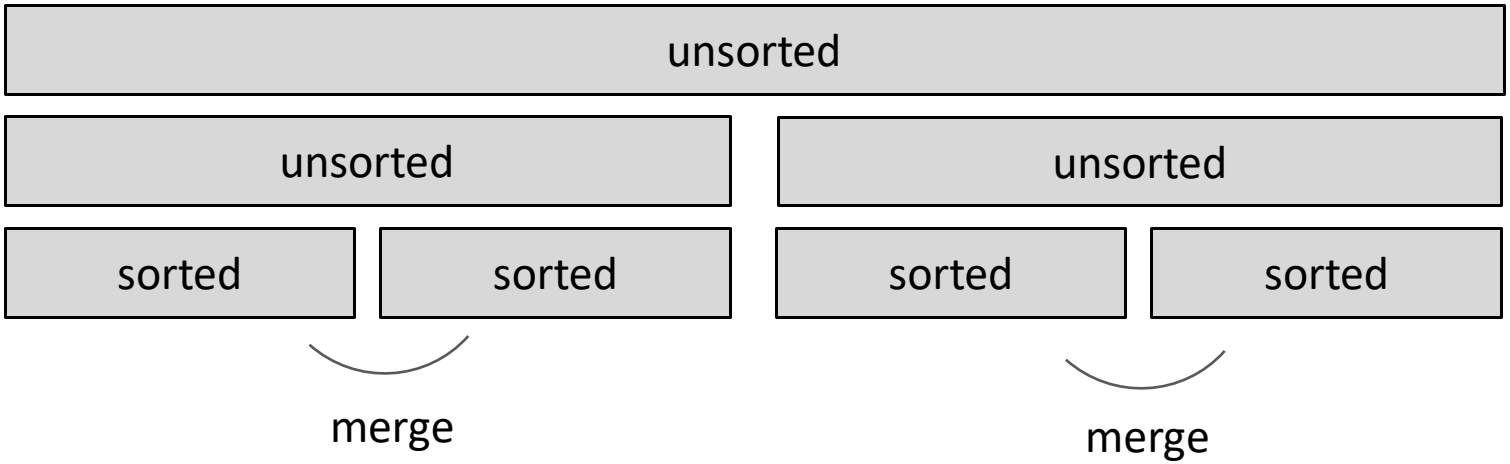
- divide and conquer



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

MERGE SORT

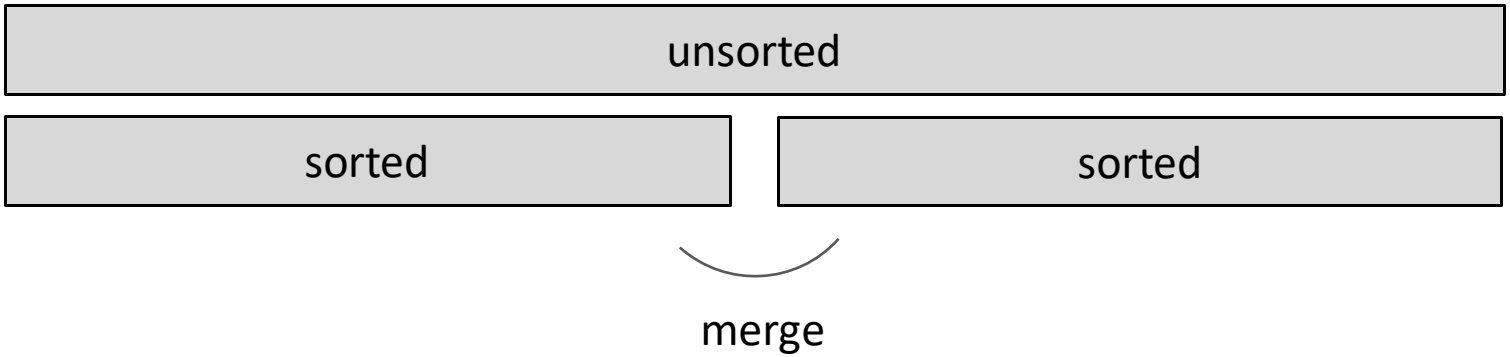
- divide and conquer



- 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!

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]

Complexity of Merge

- 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**

Recursive Merge Sort Implementation

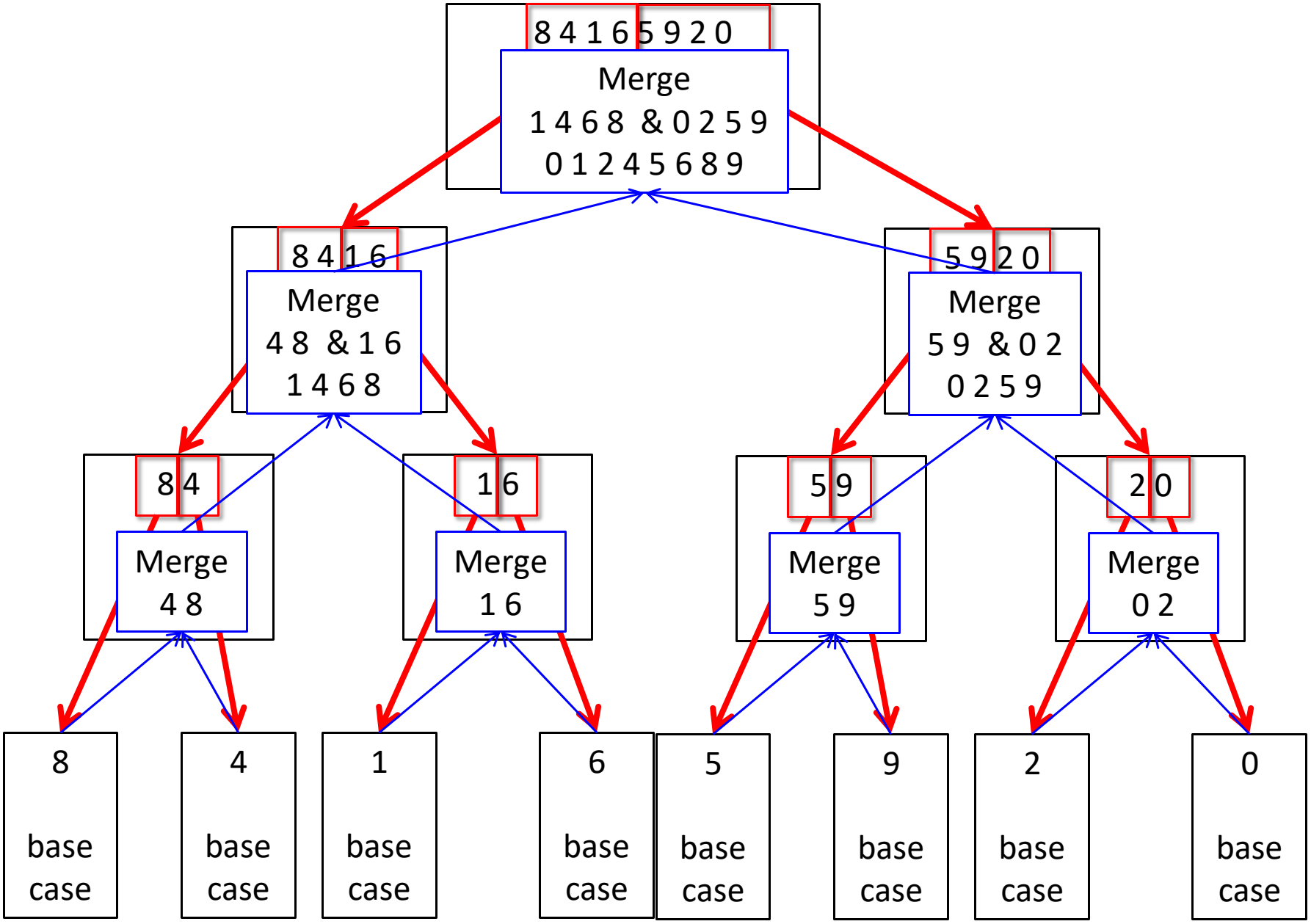
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

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 Analysis

- At **first recursion level**
  - n/2 elements in each list
  - O(n) + O(n) = O(n) where n is len(L)
- At **second recursion level**
  - n/4 elements in each list
  - two merges → O(n) where n is len(L)
- Each recursion level is O(n) where n is len(L)
- **Dividing list in half** with each recursive call
  - O(log(n)) where n is len(L)
- Overall complexity is **O(n log(n)) where n is len(L)**

O(len(L) * log(len(L)) is optimal

Other algorithms with faster ave. times
e.g., one used in Python