Master of Science in
Genomics and Bioinformatics

Lecture 12
Introduction to Program Complexity and Basic Algorithms

GNBF5010

Instructor: Jessie Y. Huang

Understanding the program complexity

How to evaluate efficiency/complexity of programs

- measure with a **timer**
- **count** the operations
- $\boxed{\text{measure the \textbf{order of growth}}}$
  - An order of growth is a set of functions whose asymptotic growth behavior is considered equivalent.
  - For example, **2n, 100n** and **n+1** belong to the same order of growth, which is written **O(n)** in Big-O notation and often called **linear** because every function in the set grows linearly with n.

The Big O notation

- Big O or O( ) expresses rate of growth of program relative to the input size (n)

- evaluates the algorithm **NOT** machine or implementation

- Big O is used to describe the worst case
  - different inputs change how the program runs
  - worst case occurs often and is the bottleneck when a program runs

The "worst case"

- Example: A function that searches for an element in a list

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

- when `e` is the **first element** in the list → BEST CASE
- when `e` is **not in list** → WORST CASE

Complexity classes

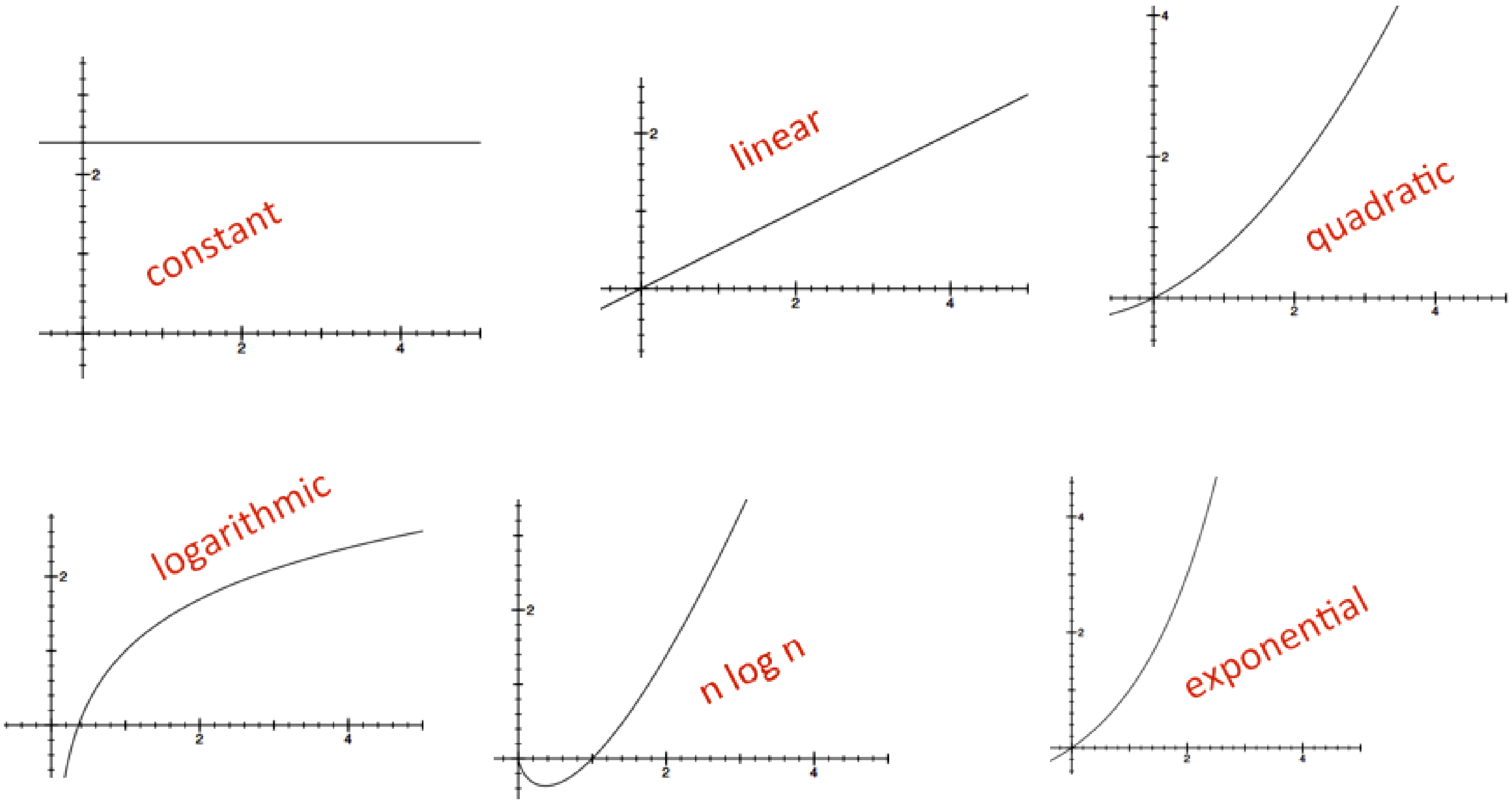
- Orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness:

Order of Growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any *b*)
$O(n)$	linear
$O(n \log_b n)$	"en log en"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any *c*)

Complexity growth

CLASS	n=10	= 100	= 1000	= 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	12676506 00228229 40149670 3205376	107150860718626732094842504906 000181056140481170553360744375038 837035105112493612249319837881 569585812759467291755314682518 7 145285692314043598457757469857 4 803934567774824230985421074605 0 623711418779541821530464749835 8 194126739876755916554394607706 2 914571196477686542167660429831 6 5262438683720566806937 6	**Good luck!!**

Types of orders of growth



Linear complexity

- Simple iterative loop algorithms are typically linear in complexity
- complexity often depends on number of iterations

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

- number of times around loop is n
- number of operations inside loop is a **constant** (in this case, 3: set i, multiply, set prod)
  - $O(1 + 3n + 1) = O(3n + 2) = O(n)$
- overall just *O(n)*

Quadratic complexity

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)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

- first nested loop takes *len(L1)*len(L2)* steps
- second loop takes at most *len(L1)* steps
- if we assume lists are of roughly same length, then $O(len(L1)^2)$

O( ) for nested loops

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

- when dealing with nested loops, **look at the ranges**
- in nested loops, **each iterating n times**
- ***$O(n^2)$***

INTRODUCTION TO BASIC SEARCHING ALGORITHMS

Searching algorithms

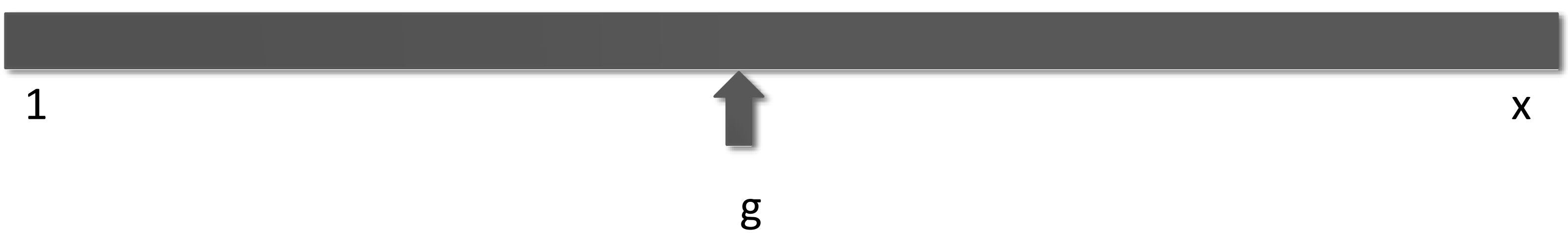
- searching algorithm – method for finding an item or group of items with specific properties within a collection of items
- collection could be implicit
  - example – find square root as a search problem
    - exhaustive enumeration
    - bisection search
- collection could be explicit
  - example – is a student record in a stored collection of data?

Searching algorithms

- linear search
  - **brute force** search (aka British Museum algorithm)
  - list does not have to be sorted
- Bisection/binary search
  - list **MUST be sorted** to give correct answer
  - saw two different implementations of the algorithm

Bisection/Binary search

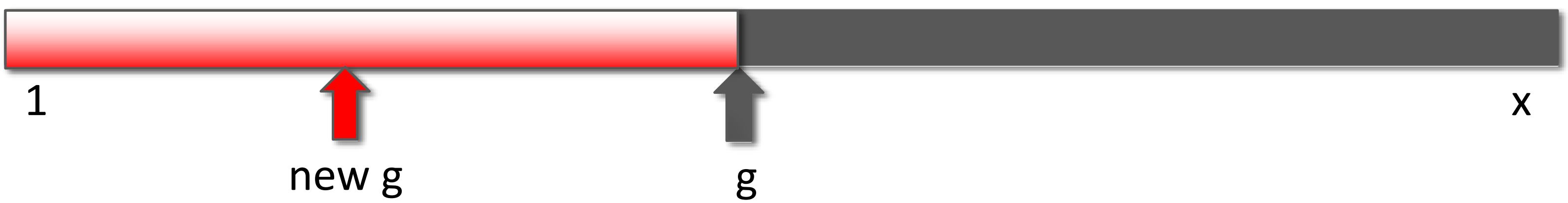
- We know that the square root of x lies between 1 and x, from mathematics
- Rather than exhaustively trying things starting at 1, suppose instead we pick a number in the middle of this range



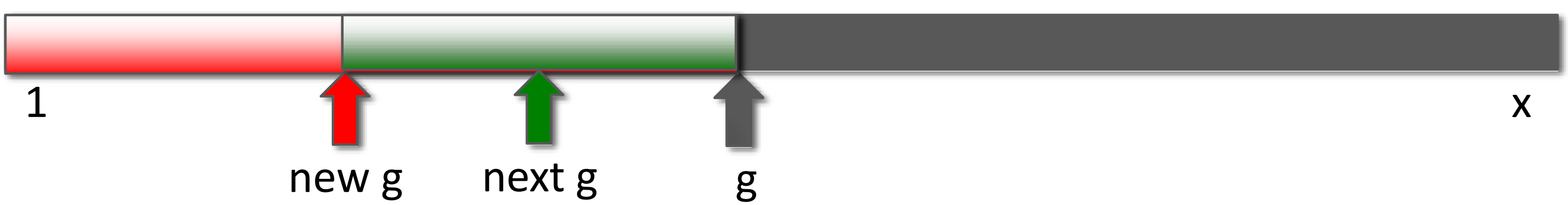
- If we are lucky, this answer is close enough

Bisection/Binary search

- If not close enough, is guess too big or too small?
- If $g^2 > x$, then know g is too big; but now search



- And if, for example, this new g is such that $g^2 < x$, then know too small; so now search



- At each stage, reduce range of values to search by half

Example of square root

```
x = 25
epsilon = 0.01
numGuesses = 0
low = 1.0
high = x
ans = (high + low)/2.0

while abs(ans**2 - x) >= epsilon:
print(f'low = {low} high = {high} ans = {ans}')
numGuesses += 1
if ans**2 < x:
    low = ans
else:
    high = ans
ans = (high + low)/2.0

print(f'numGuesses={numGuesses}')
print(f'{ans} is close to square root of {x}.')
```

Some observations

- Bisection search radically reduces computation time – being smart about generating guesses is important
- Should work well on problems with "ordering" property – value of function being solved varies monotonically with input value
  - Here function is g^2, which grows as g grows

Program complexity

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
  - assumes the **list is sorted**!

INTRODUCTION TO BASIC SORTING ALGORITHMS

Sorting ALGORITHMS

- Want to efficiently sort a list of entries (typically numbers)
- Will see a range of methods, including one that is quite efficient

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

Complexity of bubble sort

```
def bubble_sort(L):
    swap = True
    while swap:
        swap = False
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = True
                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

Analyzing selection sort

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
2. induction step: move minimum element from suffix to end of prefix.
3. when exit, prefix is entire list, suffix empty, so sorted

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))

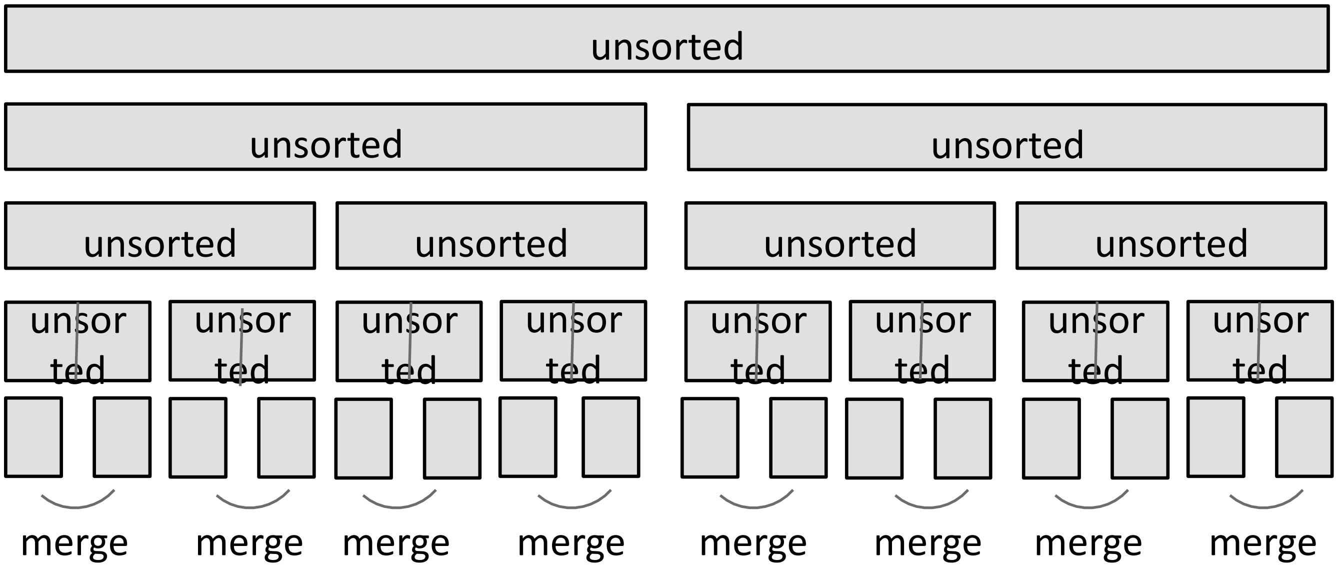
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

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

Merge sort

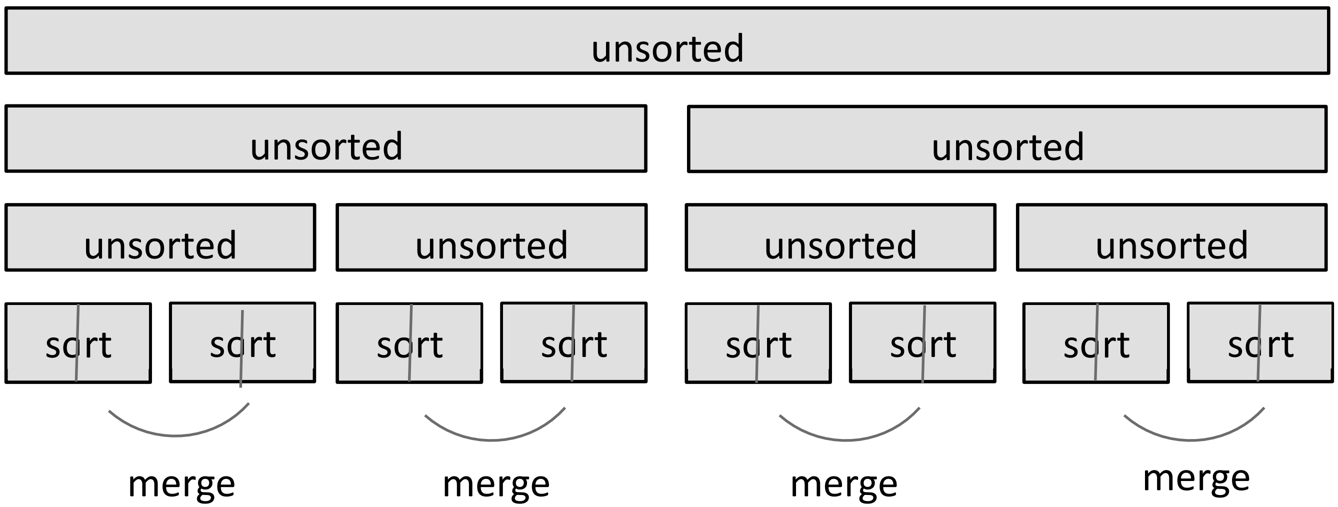
- divide and conquer



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

Merge sort

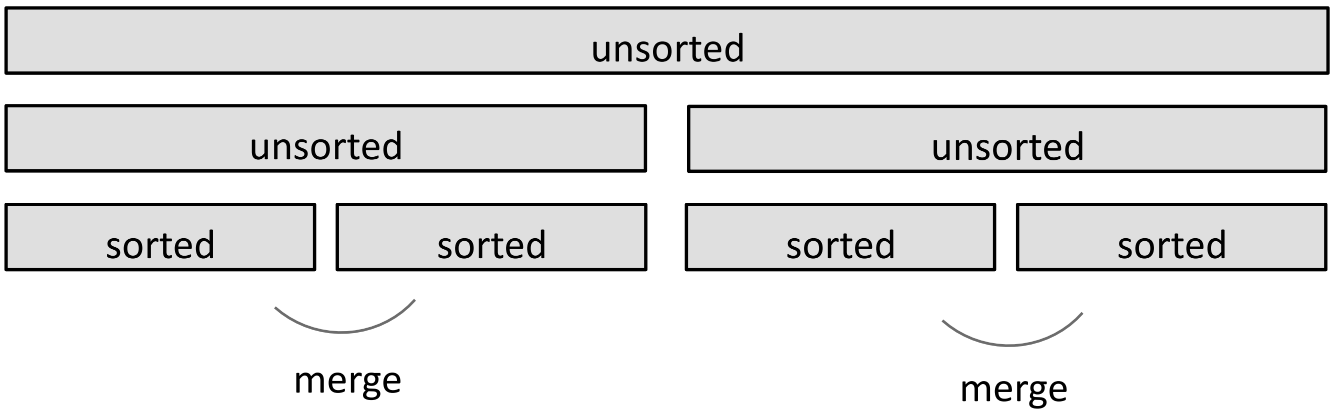
- divide and conquer



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

Merge sort

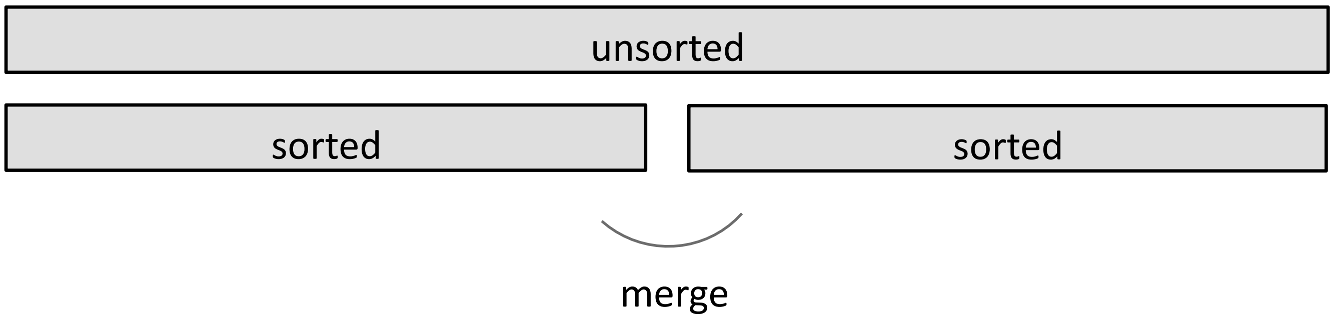
- divide and conquer



- 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

Example of merging

1. look at first element of each, move the smaller to end of the result
2. when one list empty, just copy rest of other list

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 the sublists

```
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 the 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

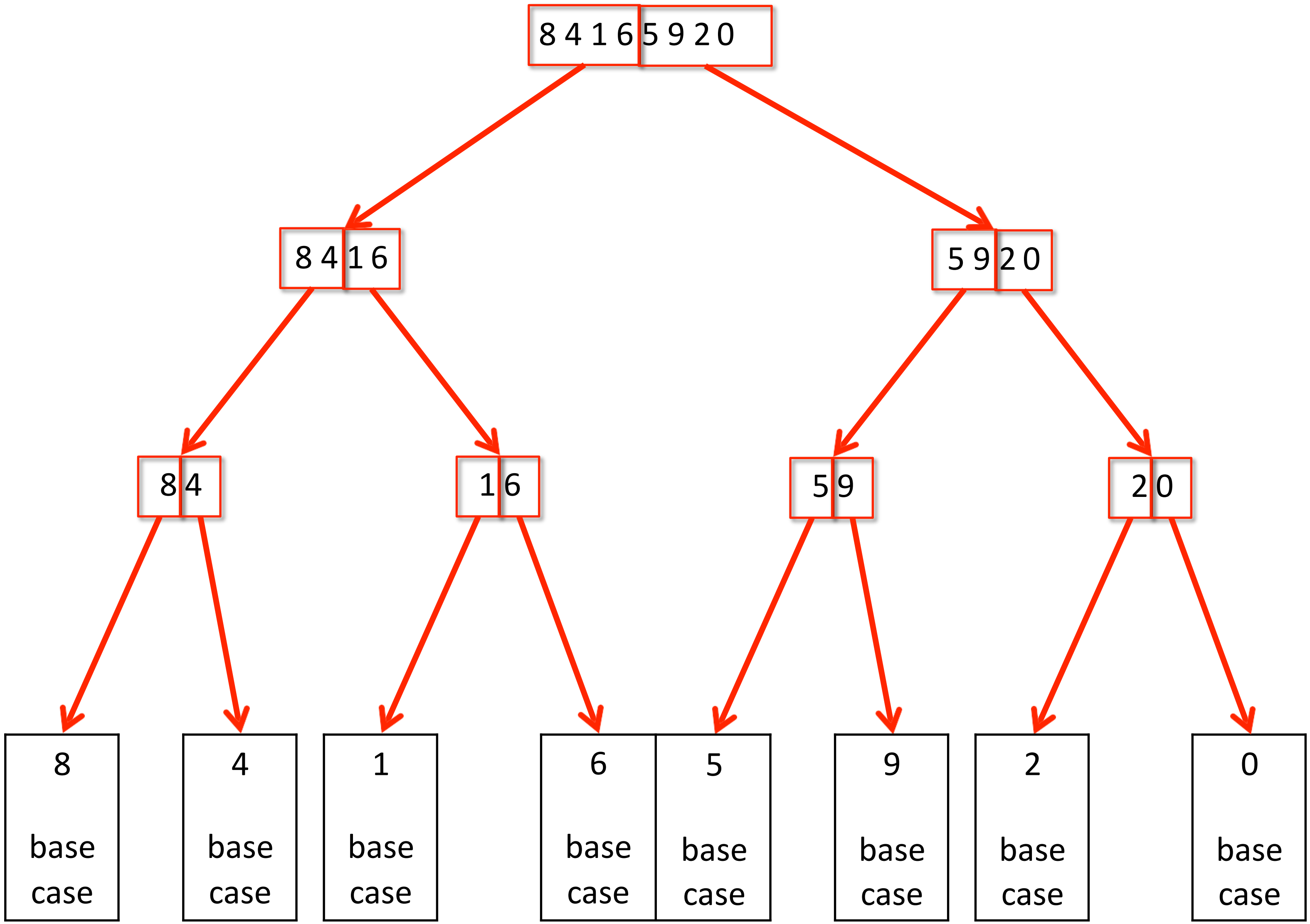
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

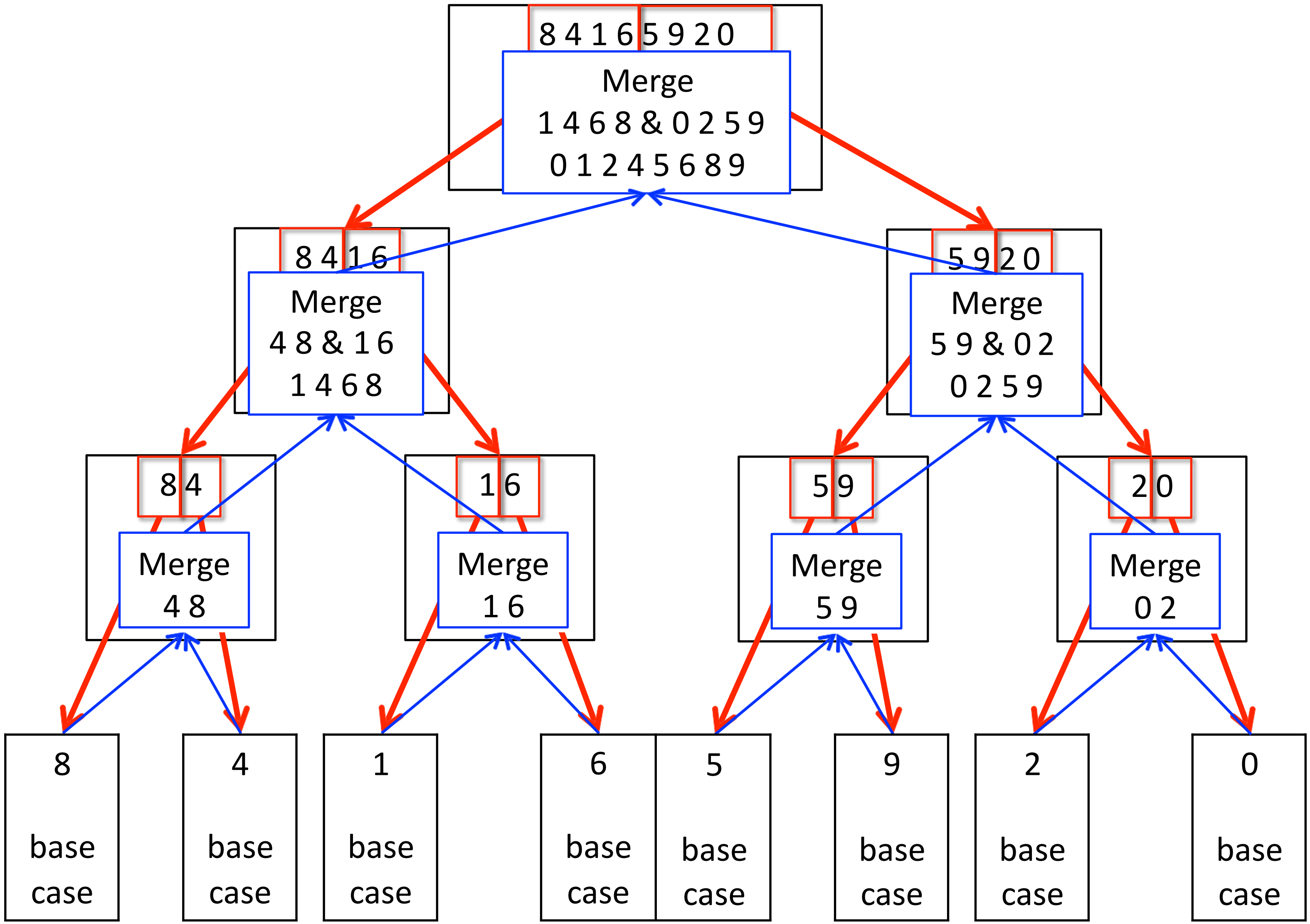
base case

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


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
  - 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)**

Sorting summary

- bubble sort
  - $O(n^2)$
- selection sort
  - $O(n^2)$
  - guaranteed the first i elements were sorted
- merge sort
  - $O(n \log(n))$
- $O(n \log(n))$ is the fastest a sort can be

Reference

- The Practice of Computing using Python, 2nd Edition. Punch & Enbody.