

SEARCHING AND SORTING

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

6.0001 LECTURE 10

HALF-TERM EVALUATIONS

- You have until Thursday 9am to evaluate 6.0001
 - <http://web.mit.edu/subjectevaluation/evaluate.html>



6.0001 FINAL

Wednesday Oct 16

- In **lecture time** 3pm to 4:30pm
- **2 sheets of paper** allowed as aid
- **Paper part**, no electronic devices open
 - If you finish early, work on programming problems with pencil and paper
- **Programming part**, Python IDE and MITx only electronic access
- Wednesday's lecture is a review session

LAST TIME

- Efficiency (memory and time)
- Complexity, order of growth, big oh notation
- Best, average, worst case scenario
- Linear, polynomial, exponential complexity examples

TODAY

- More classes of complexity and examples
- Searching and sorting algorithms

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
- **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)

COMPLEXITY OF COMMON PYTHON FUNCTIONS

■ 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

- 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 $O(n)$

LOGARITHMIC COMPLEXITY

TRICKY COMPLEXITY

```
def digit_add(n):  
    """ assume n an int >= 0 """  
    answer = 0  
    s = str(n)  
    for c in s:  
        answer += int(c)  
    return answer
```

linear $O(\text{len}(s))$
but what in terms
of input n ?

- Adds digits of a number together
- Tricky part
 - Convert integer to string
 - Iterate over **length of string**, not magnitude of input n
 - Think of it like dividing n by 10 each iteration
- **$O(\log n)$** – base doesn't matter

LOGARITHMIC COMPLEXITY

- Complexity grows as log of size of one of its inputs
- Example: one **bisection search** implementation
- Example: **binary search** of a list

LOG LINEAR COMPLEXITY

LOG-LINEAR COMPLEXITY

- Many practical programs are log-linear
- Commonly used log-linear algorithm is **merge sort**
- Will see this in a few slides

SEARCHING ALGORITHMS

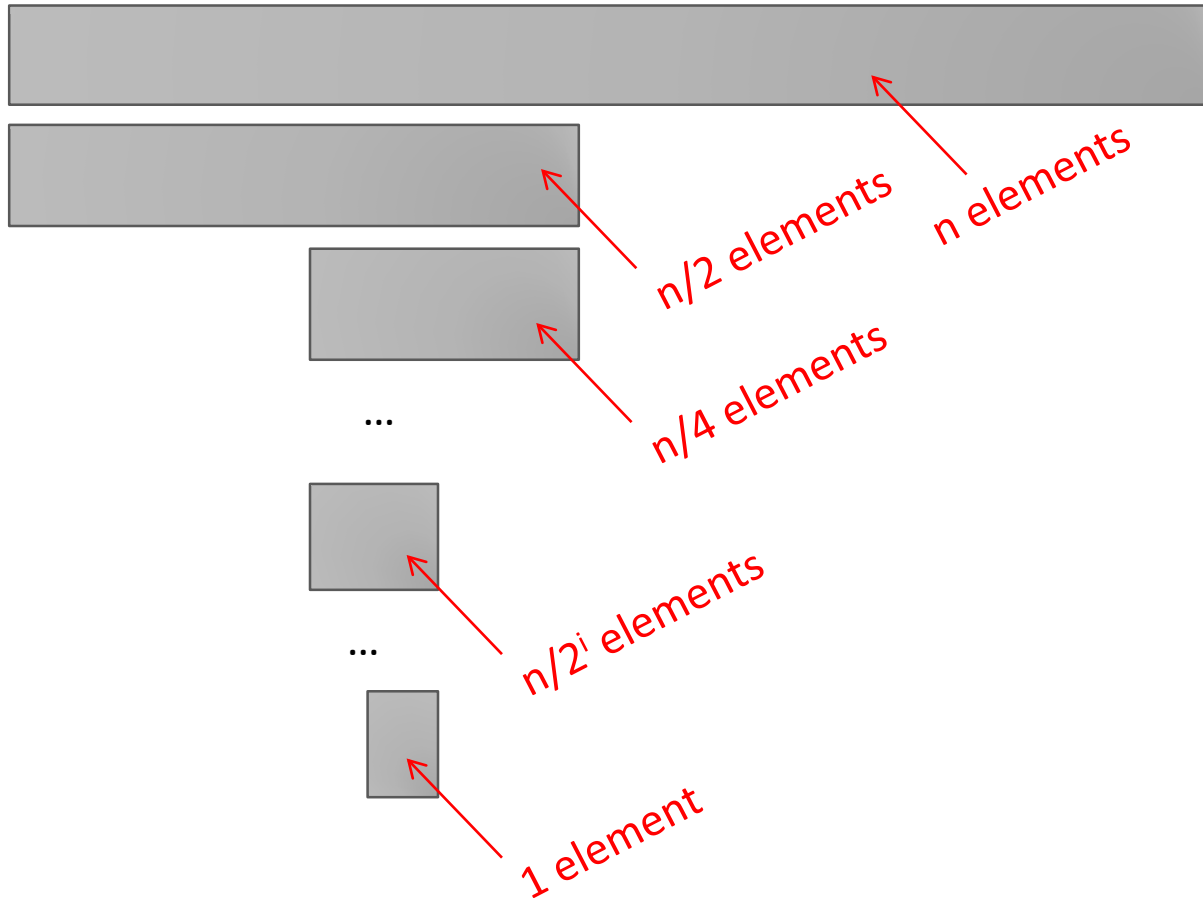
SEARCHING ALGORITHMS

- Linear search
 - **Brute force** search
 - List does not have to be sorted
 - Saw this last time
 - $O(\text{len}(L))$
- Bisection search
 - List **MUST be sorted** to give correct answer
 - Will see two different implementations of the algorithm

BISECTION SEARCH

- 1) Pick an index, i , that divides list in half
 - 2) Ask if $L[i] == e$
 - 3) If not, ask if $L[i]$ is larger or smaller than e
 - 4) Depending on answer, search left or right half of L for e
-
- A new version of divide-and-conquer
 - Break into smaller versions of problem (smaller list), plus simple operations
 - Answer to smaller version is answer to original version

BISECTION SEARCH COMPLEXITY ANALYSIS



- Finish looking through list when

$$1 = n/2^i$$

$$\text{so } i = \log n$$

- Complexity is **$O(\log n)$** –
where n is $\text{len}(L)$

BISECTION SEARCH IMPLEMENTATION 1

```
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

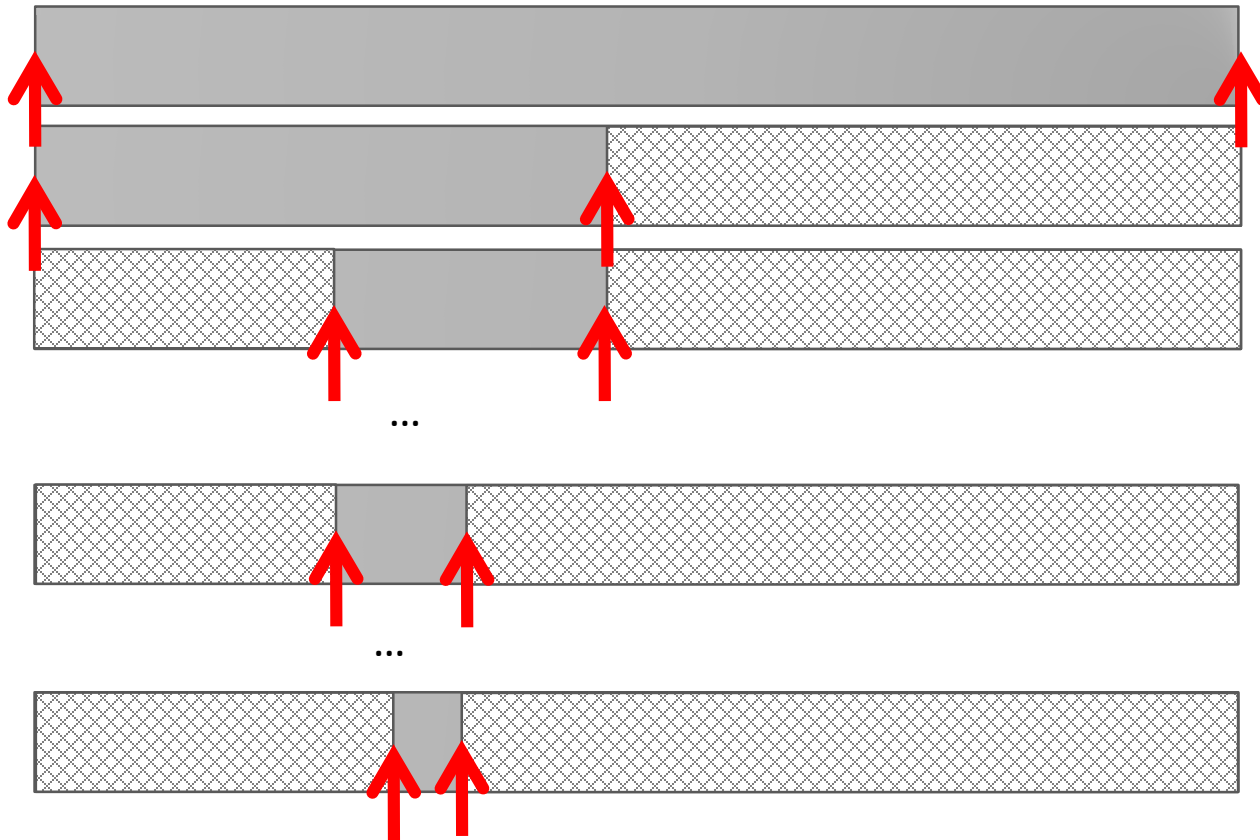
NOT constant

NOT constant

COMPLEXITY OF bisect_search1

- $O(\log n)$ bisection search calls
 - Each recursive call cuts range to search in half
 - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
 - We do this to get an expression relating k to n
- $O(n)$ for each bisection search call to copy list
 - Cost to set up recursive call at each level of recursion
- $O(\log n) * O(n) = \mathbf{O(n \log n)}$ <- this is the answer in this class
- If careful, notice list is also halved on each recursive call
 - Infinite series
 - $\mathbf{O(n)}$ is a tighter bound because copying list dominates $\log n$

BISECTION SEARCH ALTERNATE IMPLEMENTATION



- 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 $\text{len}(L)$

BISECTION SEARCH IMPLEMENTATION 2

```
def bisect_search2(L, e):  
    def bisect_search_helper(L, e, low, high):  
        if high == low:  
            return L[low] == e  
        mid = (low + high)//2  
        if L[mid] == e:  
            return True  
        elif L[mid] > e:  
            if low == mid: #nothing left to search  
                return False  
            else:  
                return bisect_search_helper(L, e, low, mid - 1)  
        else:  
            return bisect_search_helper(L, e, mid + 1, high)  
    if len(L) == 0:  
        return False  
    else:  
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

NOT constant

NOT constant

COMPLEXITY OF bisect_search2 and helper

- $O(\log n)$ bisection search calls
 - Each recursive call cuts range to search in half
 - Worst case to reach range of size 1 from n is when $n/2^k = 1$ or when $k = \log n$
 - We do this to get an expression relating k to n
- Pass list and indices as parameters
 - List never copied, just re-passed
 - $O(1)$ on each recursive call
- $O(\log n) * O(1) = \mathbf{O(\log n)}$

SEARCHING A SORTED LIST

-- n is $\text{len}(L)$

- 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!**
- When does it make sense to **sort first then search**?
 - $\text{SORT} + O(\log n) < O(n) \quad \rightarrow \text{SORT} < O(n) - O(\log n)$
 - when sorting is less than $O(n)$ \rightarrow never true!

AMORTIZED COST

-- n is $\text{len}(L)$

- Why bother sorting first?
- **Sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- $\text{SORT} + K * O(\log n) < K * O(n)$
 - for large K , **SORT time becomes irrelevant**

SORTING ALGORITHMS

BOGO/RANDOM/MONKEY SORT

- aka bogosort,
stupidsort, slowsort,
randomsort,
shotgunsort
- 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 $\text{len}(L)$** to check if sorted
- Worst case: $O(?)$ it is **unbounded** if really unlucky

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

Donald Knuth, in “The Art of Computer Programming”, said:

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"

COMPLEXITY OF BUBBLE SORT

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

- 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 $\text{len}(L)$**
to do $\text{len}(L)-1$ comparisons and $\text{len}(L)-1$ passes

SELECTION SORT

- First step
 - Extract **minimum element**
 - **Swap it** with element at **index 0**
- Second 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

COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
```

```
    for i in range(len(L)):
```

```
        for j in range(i, len(L)):
```

```
            if L[j] < L[i]:
```

```
                L[i], L[j] = L[j], L[i]
```

*len(L) times
→ $O(\text{len}(L))$*

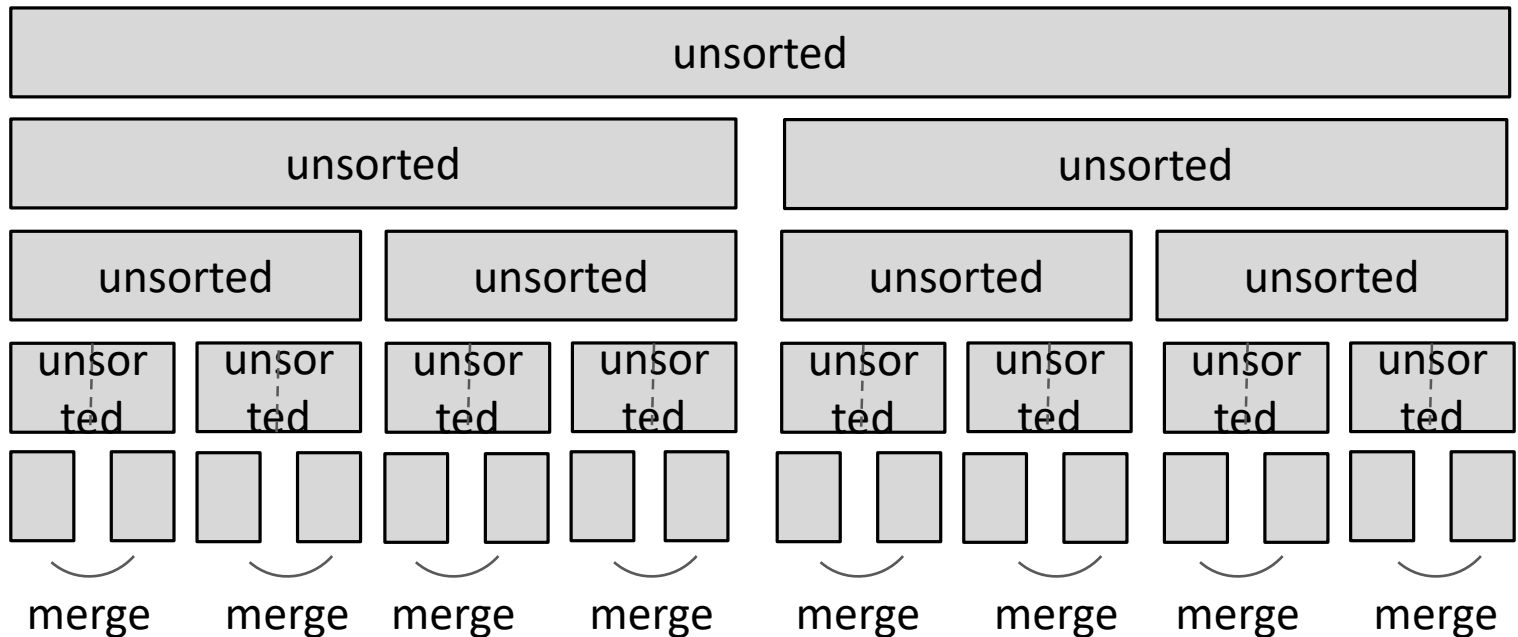
*len(L) - i times
→ $O(\text{len}(L))$*

- Outer loop executes $\text{len}(L)$ times
- Inner loop executes $\text{len}(L) - i$ times
- Complexity of selection sort is **$O(n^2)$ where n is $\text{len}(L)$**

5 Minute Break

MERGE SORT

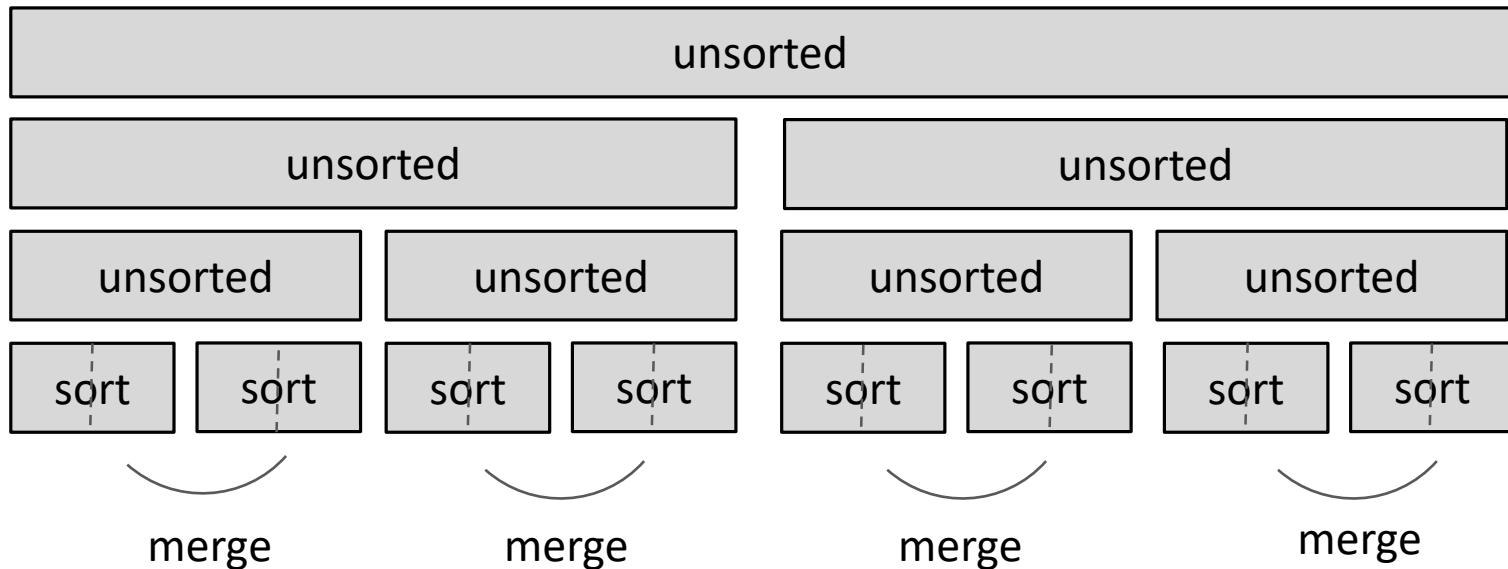
- Divide and conquer



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

MERGE SORT

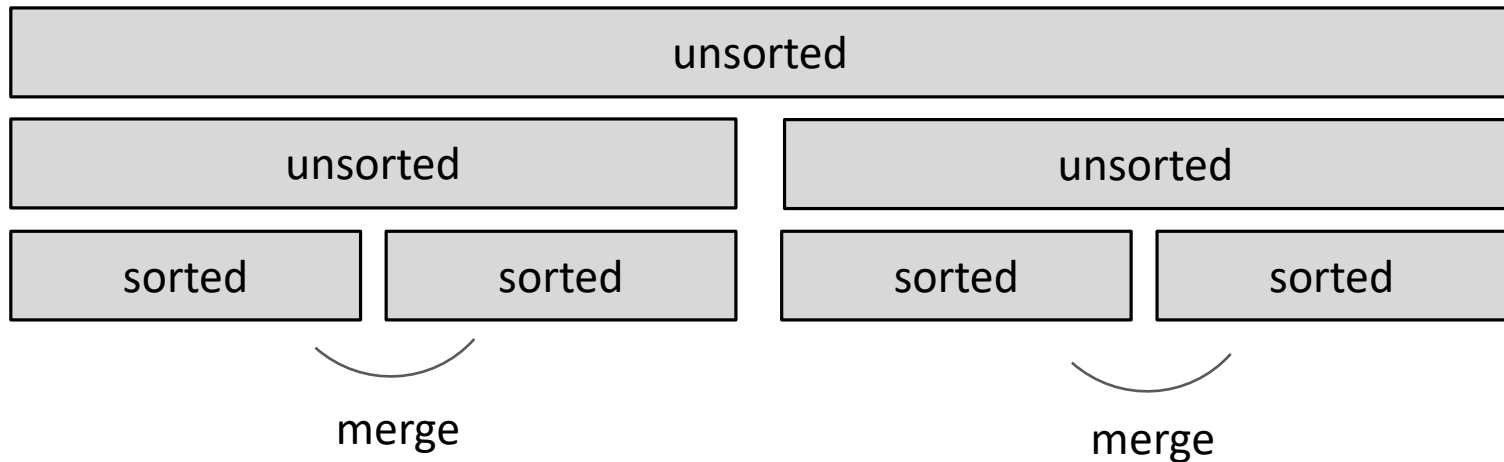
- Divide and conquer



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

MERGE SORT

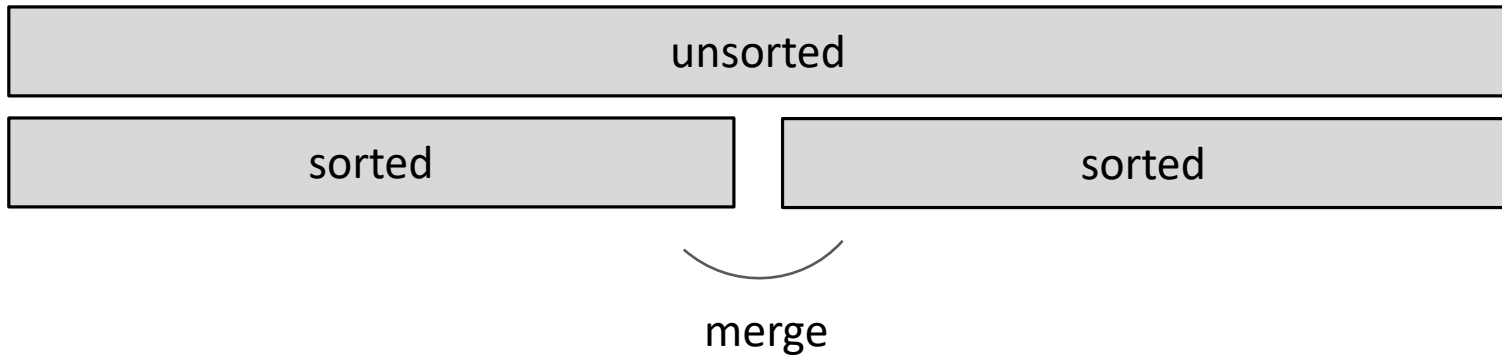
- Divide and conquer



- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT

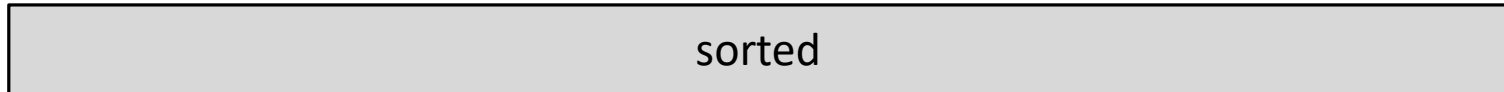
- Divide and conquer



- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT

- Divide and conquer – done!



MERGING EXAMPLE

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**
- $O(\text{len}(\text{left}) + \text{len}(\text{right}))$ copied elements
- $O(\text{len}(\text{longer list}))$ 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:])
```

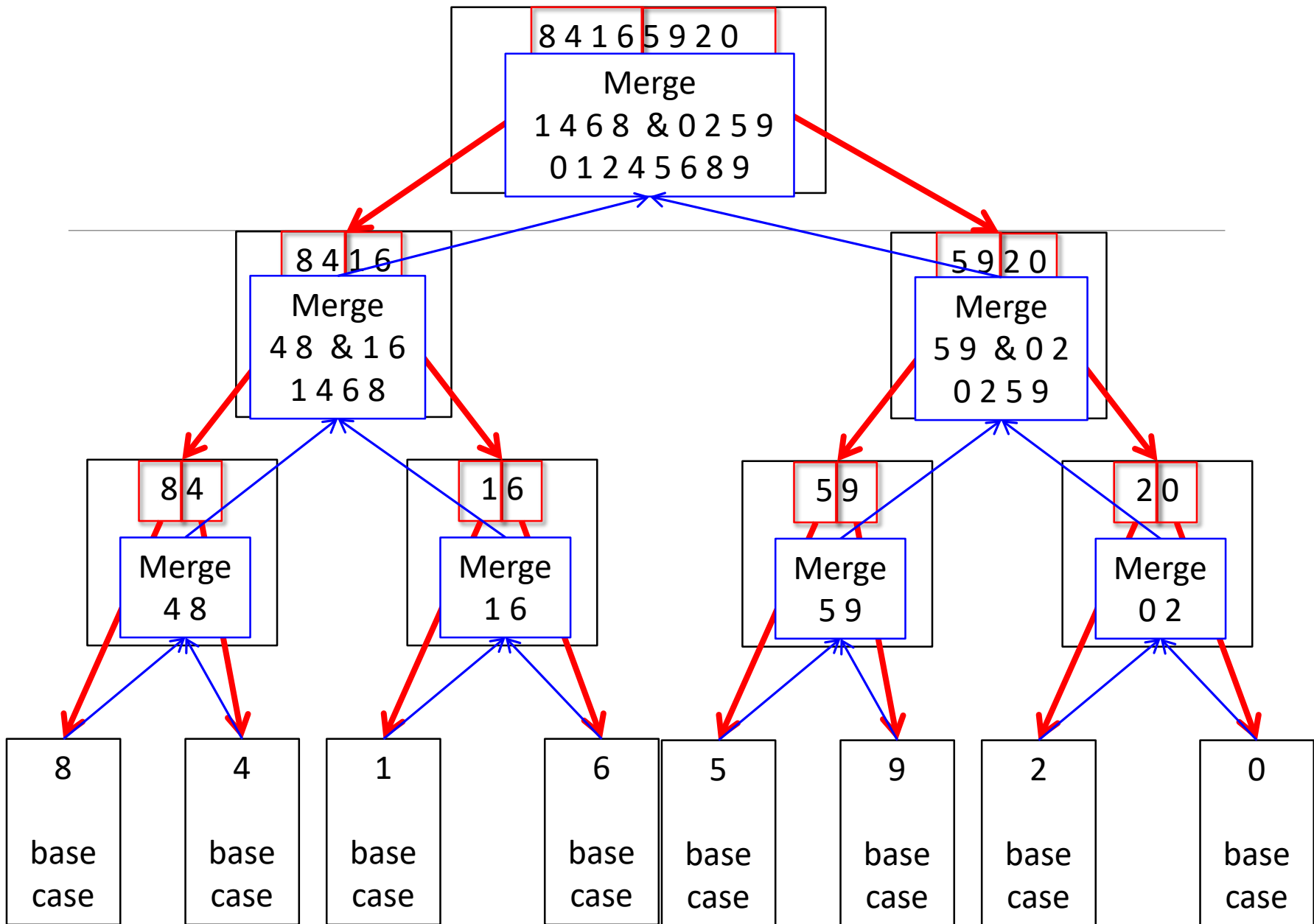
```
        return merge(left, right)
```

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



COMPLEXITY OF MERGE SORT

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

SORTING SUMMARY

-- n is $\text{len}(L)$

- bogo sort
 - randomness, unbounded $O()$
- 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

SUMMARY

WHAT DID YOU LEARN?

- Python syntax
- Flow of control
 - Loops, branching, exceptions
- Data structures
- Organization, decomposition, abstraction
 - Functions
 - Classes
- Algorithms
- Computational complexity
 - Big Oh notation
 - Searching and sorting

HOME STRETCH OF 6.0001

- Wednesday Oct 9
 - TAs will conduct a review session for test
- Wednesday Oct 16
 - Test in class
 - Exam lectures are mandatory, no conflict exams are offered
- Monday Oct 21
 - Start 6.0002