SEARCHING AND SORTING 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?

SEARCHING ALGORITHMS

- linear search
  - **brute force** search
  - list does not have to be sorted
- bisection search
  - list **MUST be sorted** to give correct answer
  - will see two different implementations of the algorithm

LINEAR SEARCH ON UNSORTED LIST

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

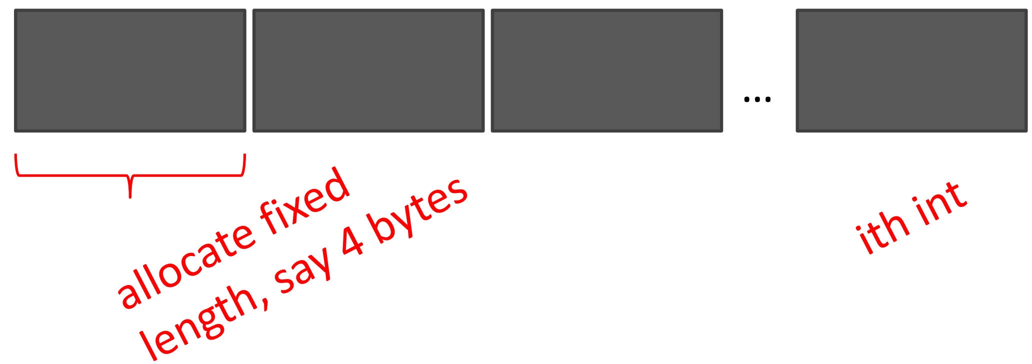
speed up a little by returning True here, but speed up doesn't impact worst case

- 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)**

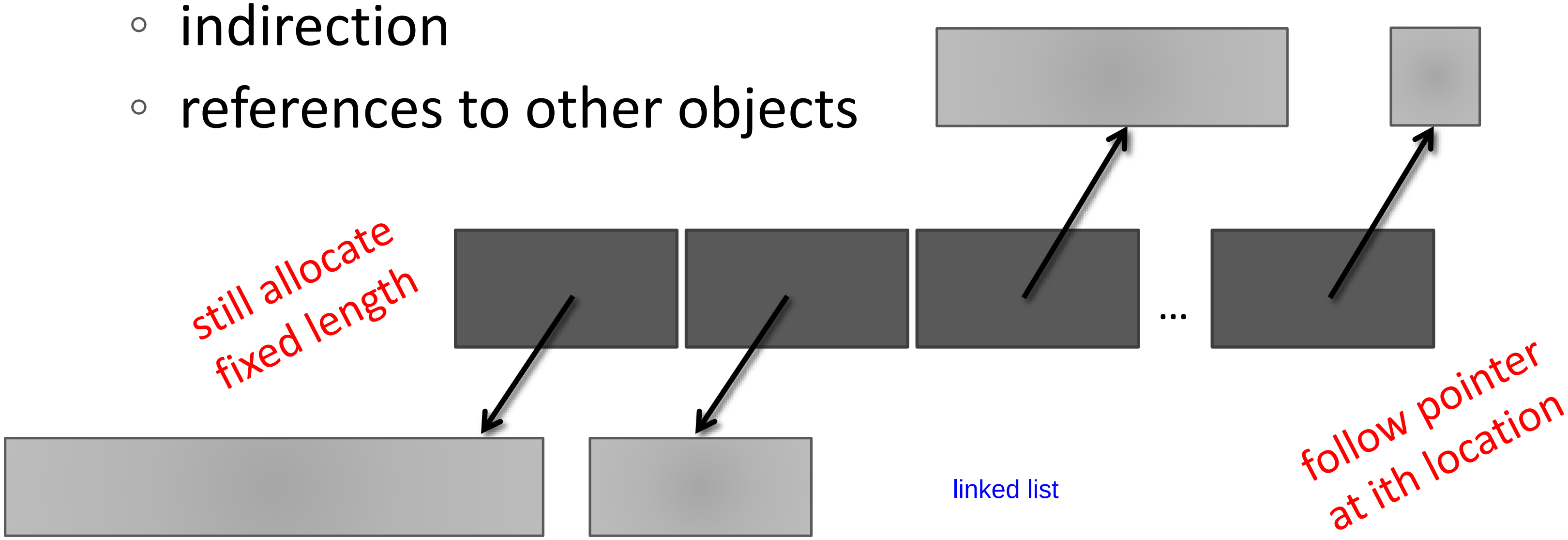
Assumes we can retrieve element of list in constant time

CONSTANT TIME LIST ACCESS

- if list is all ints
  - i^{th} element at
    - base + 4*I



- if list is heterogeneous
  - indirection
  - references to other objects



LINEAR SEARCH ON SORTED LIST

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[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)**

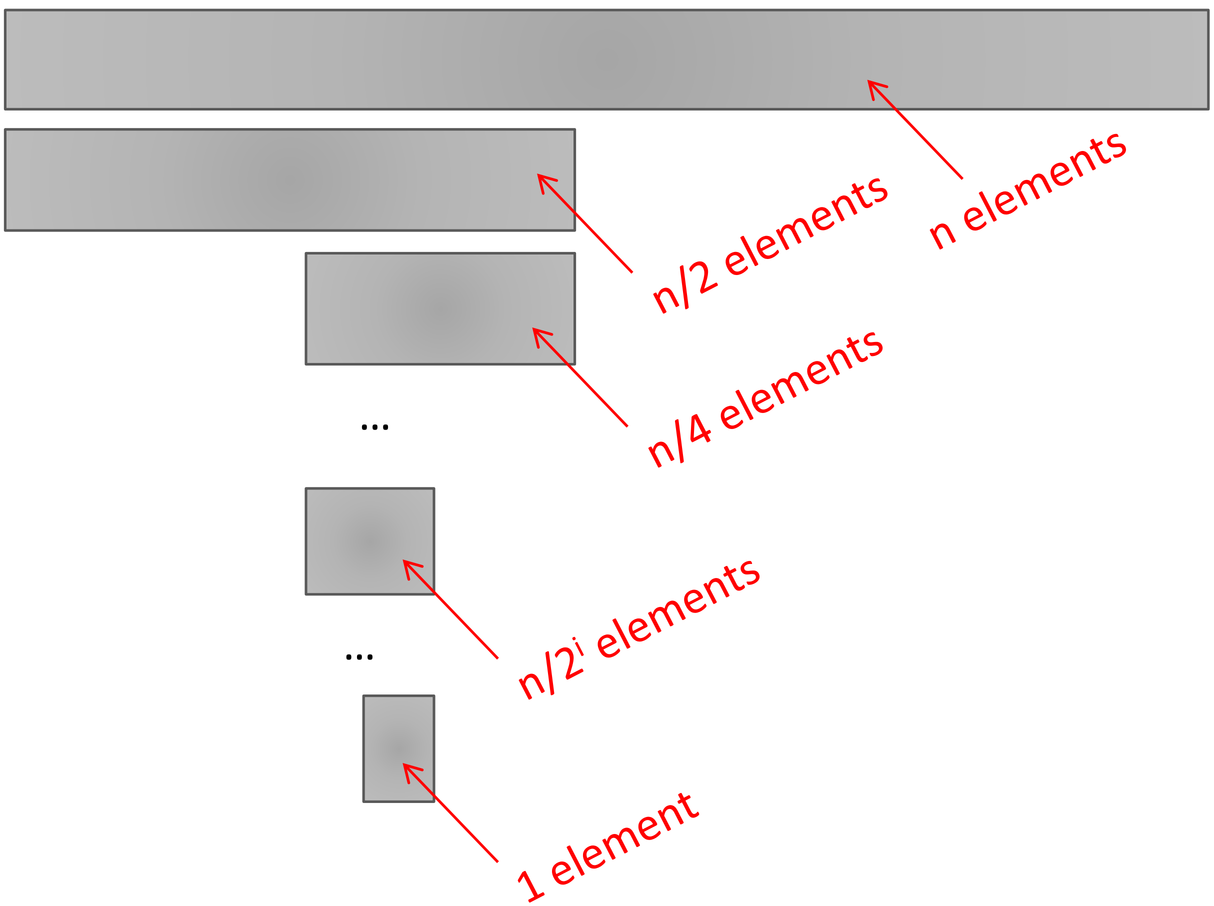
USE BISECTION SEARCH

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 a divide-and-conquer algorithm

- Break into smaller version of problem (smaller list), plus some simple operations
- Answer to smaller version is answer to original problem

BISECTION SEARCH COMPLEXITY ANALYSIS



- finish looking through list when

  $1 = n/2^i$

  so $i = \log n$

- complexity is **O(log n) – where n is 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

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 THE TWO BISECTION SEARCHES

- **Implementation 1 – bisect_search1**
  - O(log n) bisection search calls
  - O(n) for each bisection search call to copy list
  - → **O(n log n)**
  - → O(n) for a tighter bound because length of list is halved each recursive call
- **Implementation 2 – bisect_search2** and its helper
  - pass list and indices as parameters
  - list never copied, just re-passed
  - → **O(log n)**

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(logn)**
  - 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`)
  - when sorting is less than O(`n`) → never true!

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

  → for large `K`, **SORT time becomes irrelevant**

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

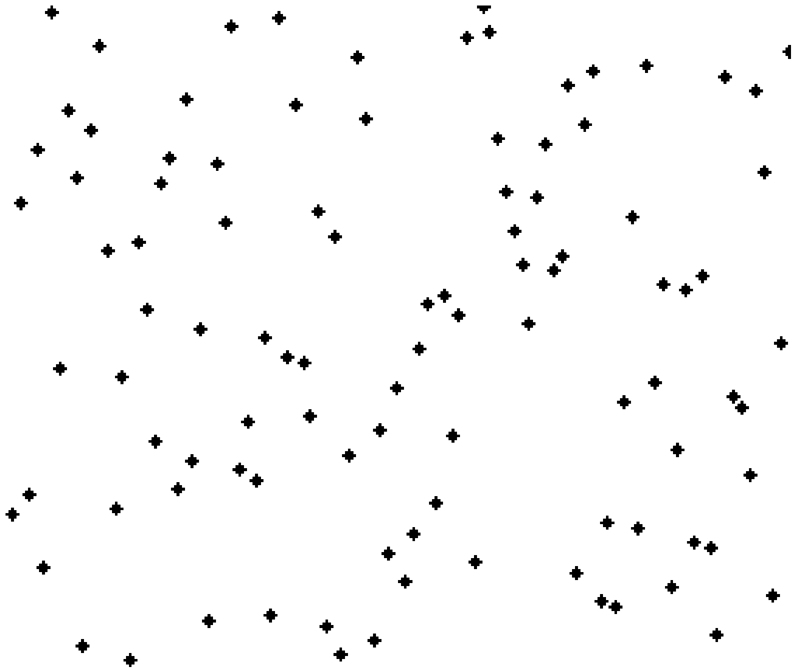
COMPLEXITY OF BOGO SORT

```
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

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



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))

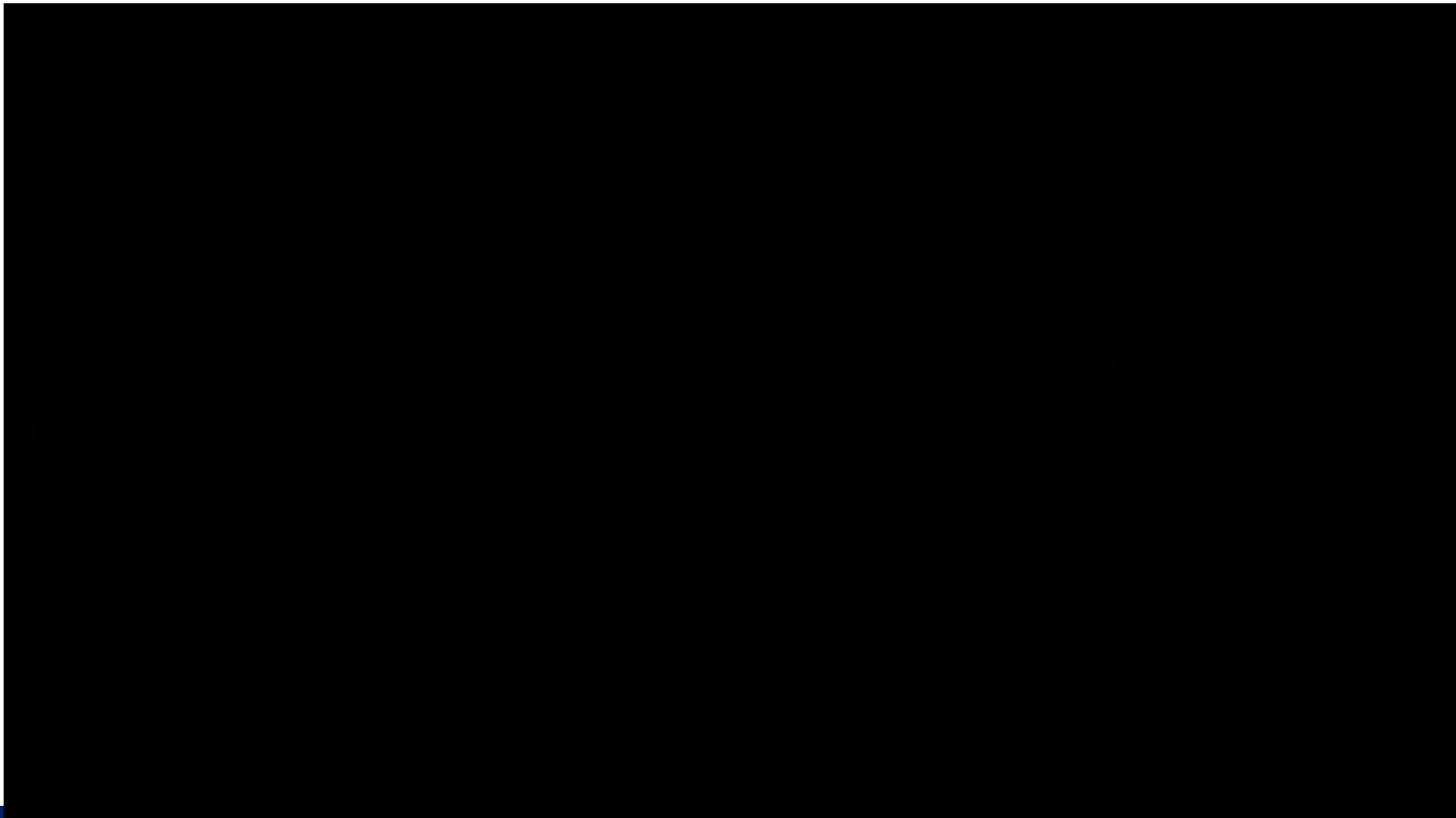
O(len(L))

- 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

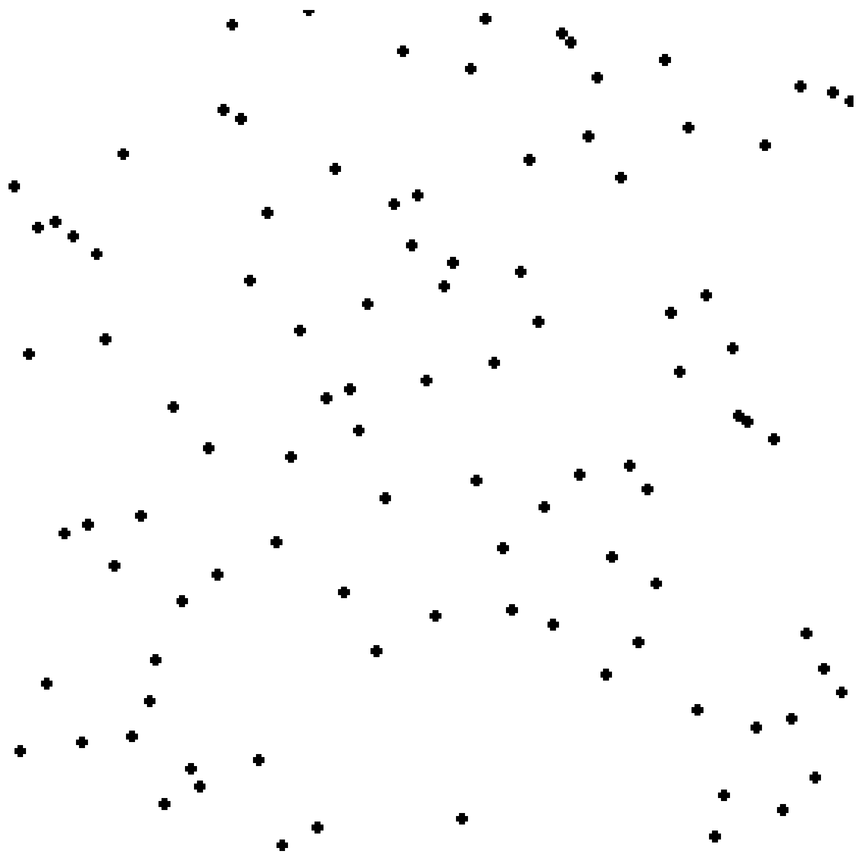
SELECTION SORT

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

SELECTION SORT WITH MIT STUDENTS



SELECTION SORT DEMO



ANALYZING SELECTION SORT

- 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 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))

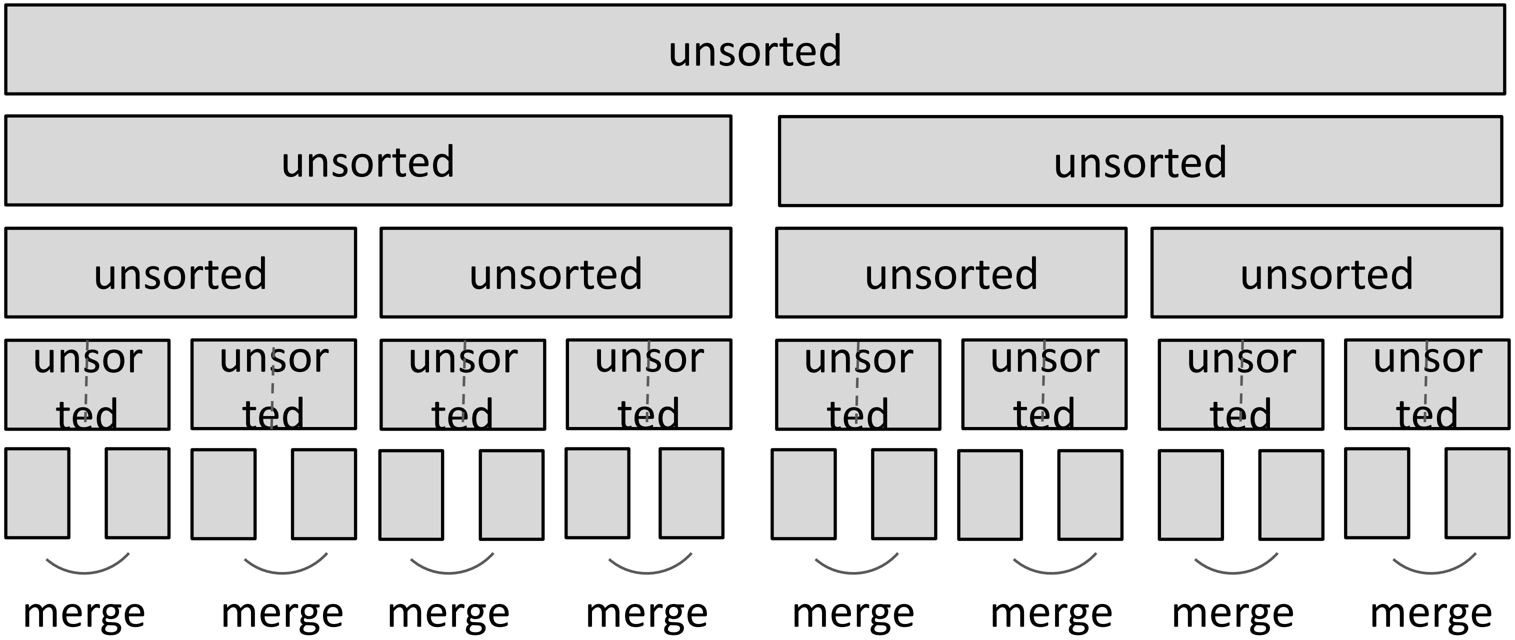
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **O(n^2) where n is len(L)**

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
     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

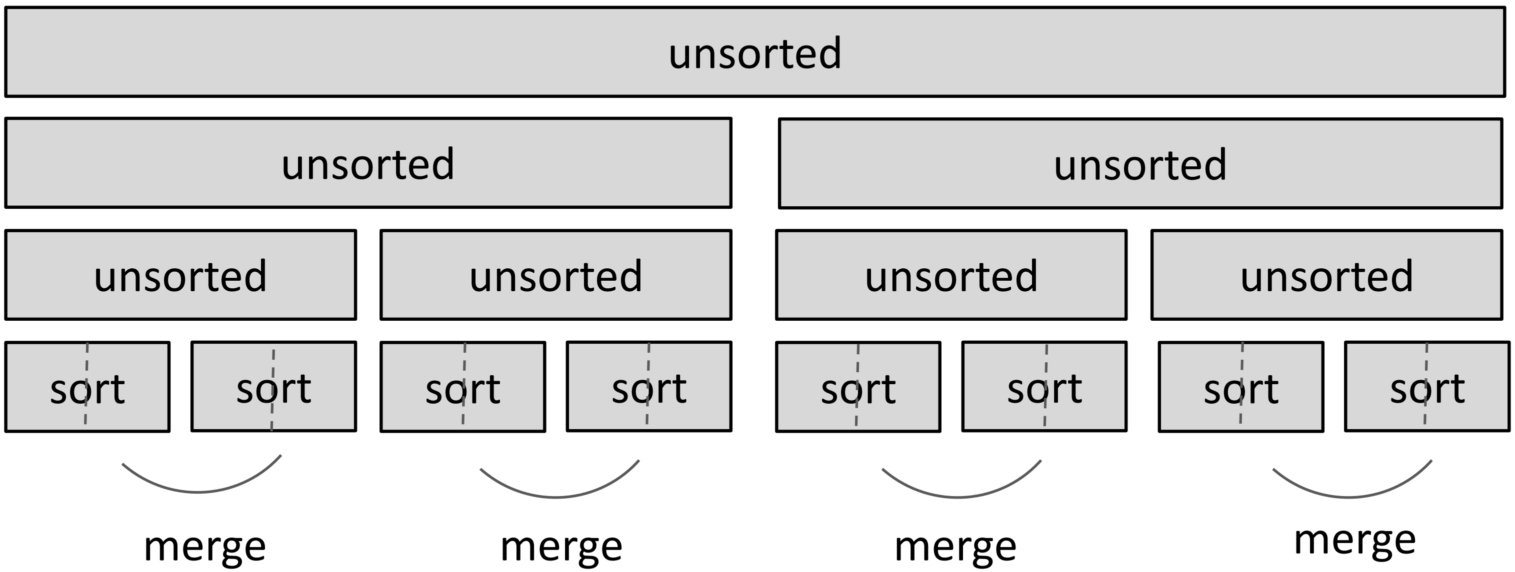
- divide and conquer



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

MERGE SORT

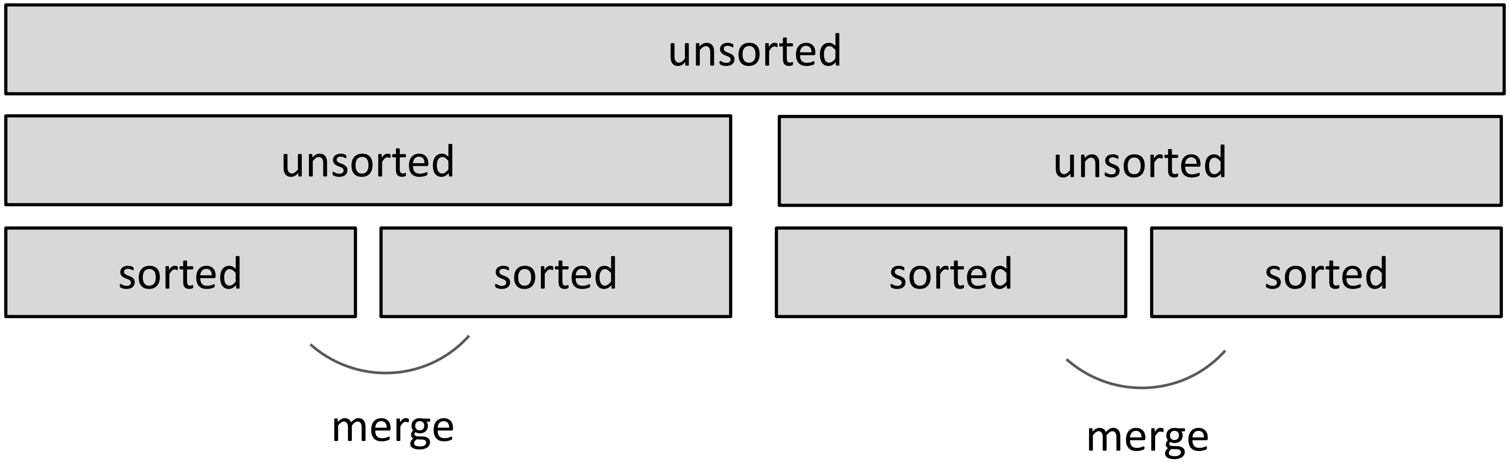
- divide and conquer



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

MERGE SORT

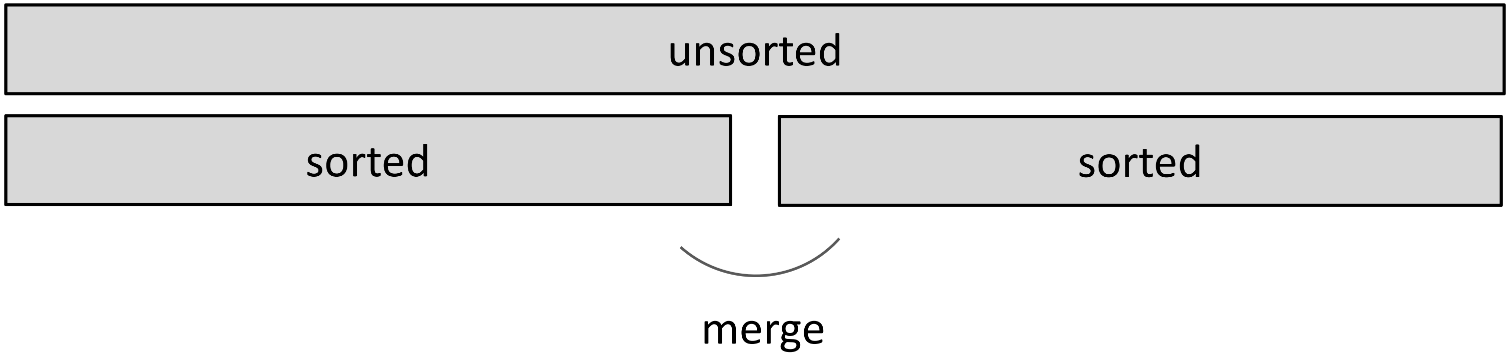
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



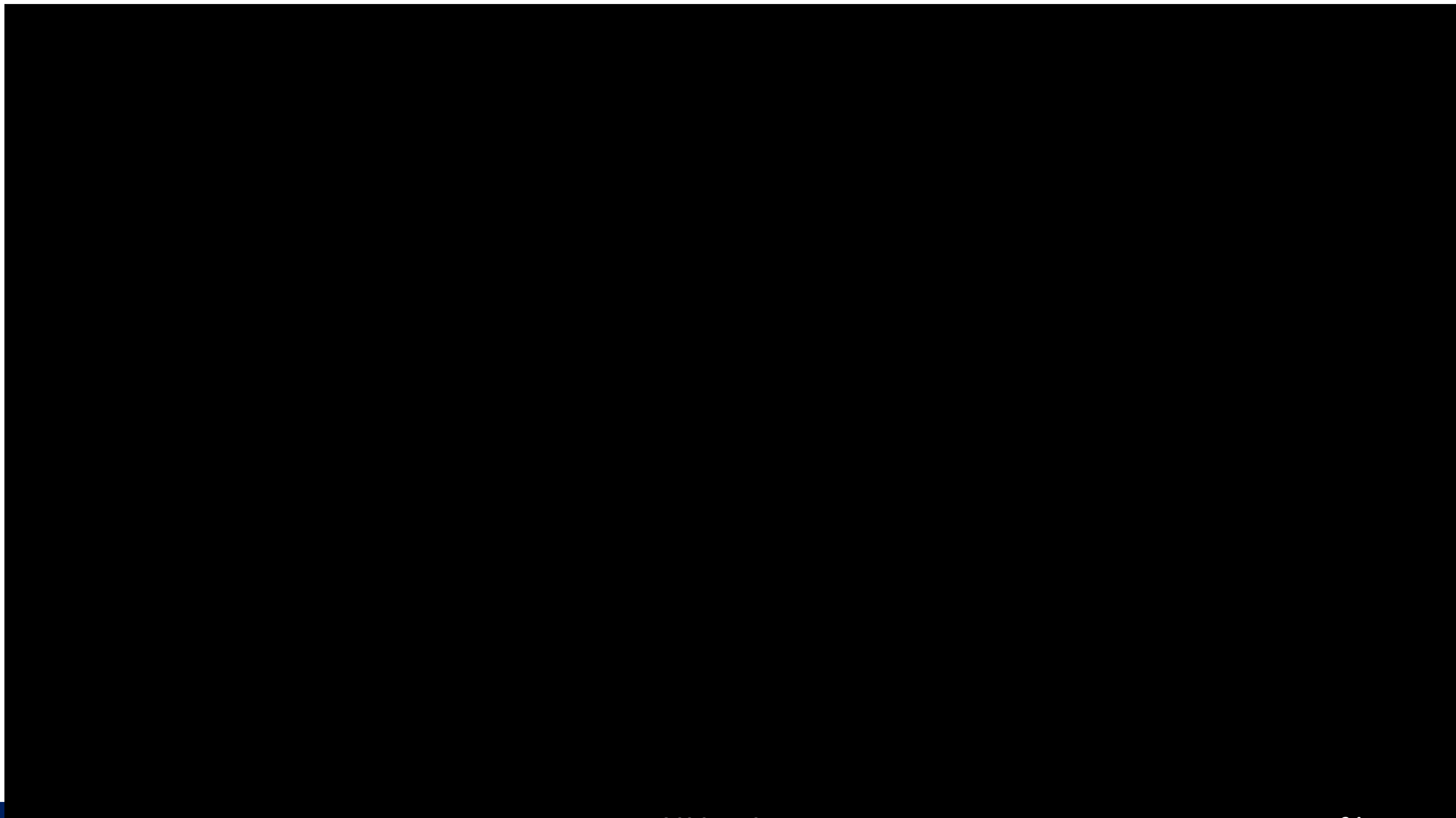
- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

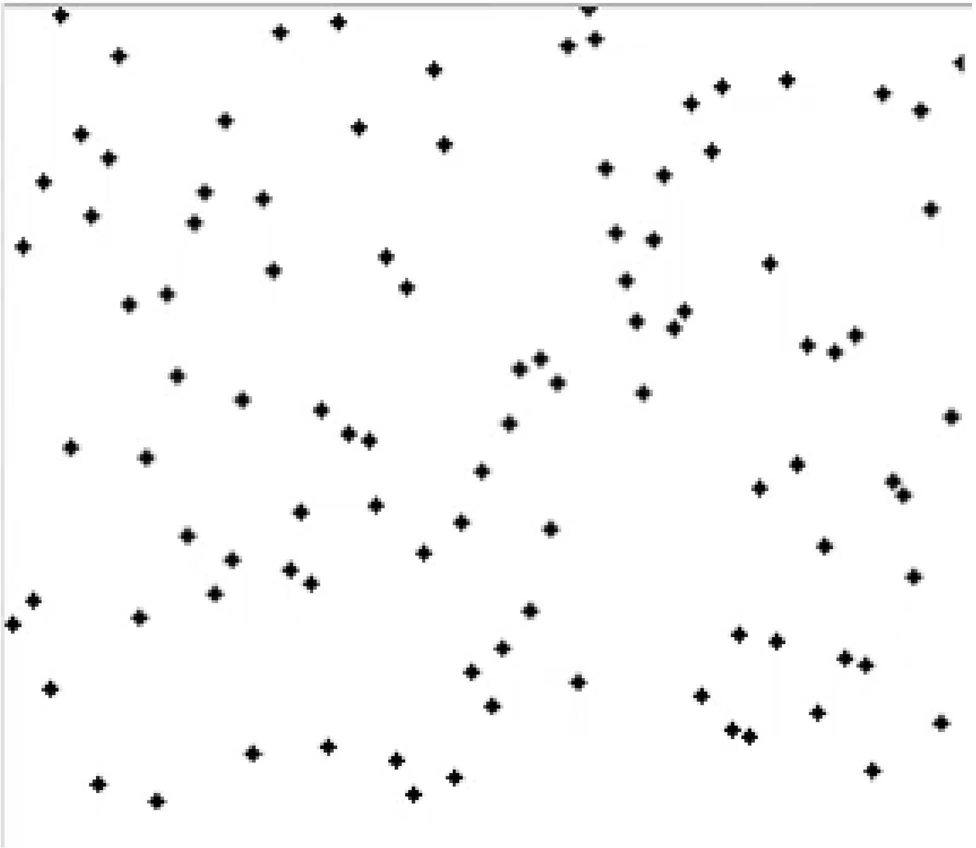
- divide and conquer – done!

sorted

MERGE SORT WITH MIT STUDENTS



MERGE SORT DEMO



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

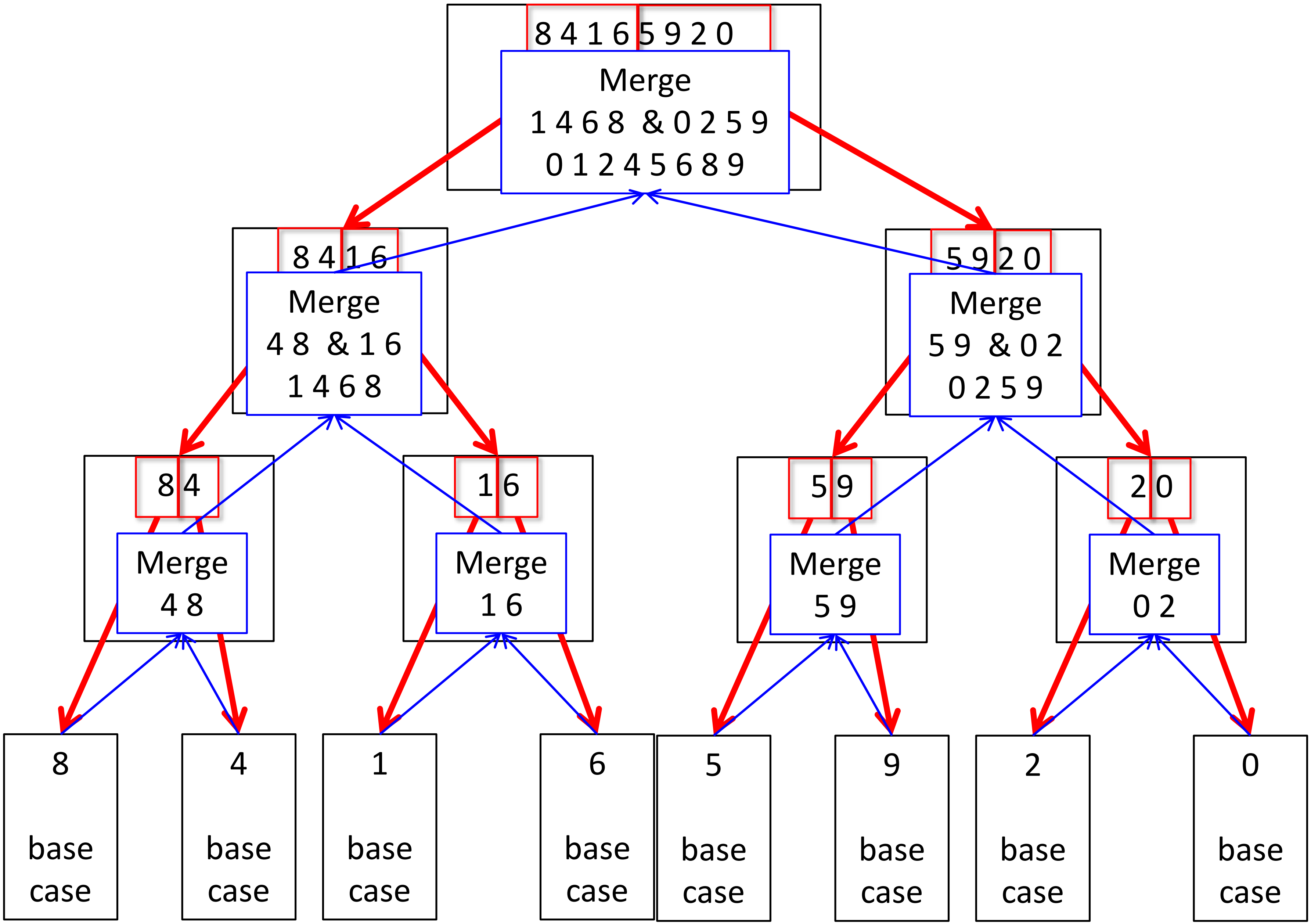
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

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
  - 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 -- n is 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