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 if the list is really large, that's a problem
  - **brute force** search Given a collection, just walk through each element of it one at a time trying to see if I found the solution or not
  - list does not have to be sorted
- bisection search
  - list **MUST be sorted** to give correct answer I don't have to look at all of the list to find it, but can divide up the work
  - 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

even breaking out of this loop, in the worst case, is still linear

- 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)**

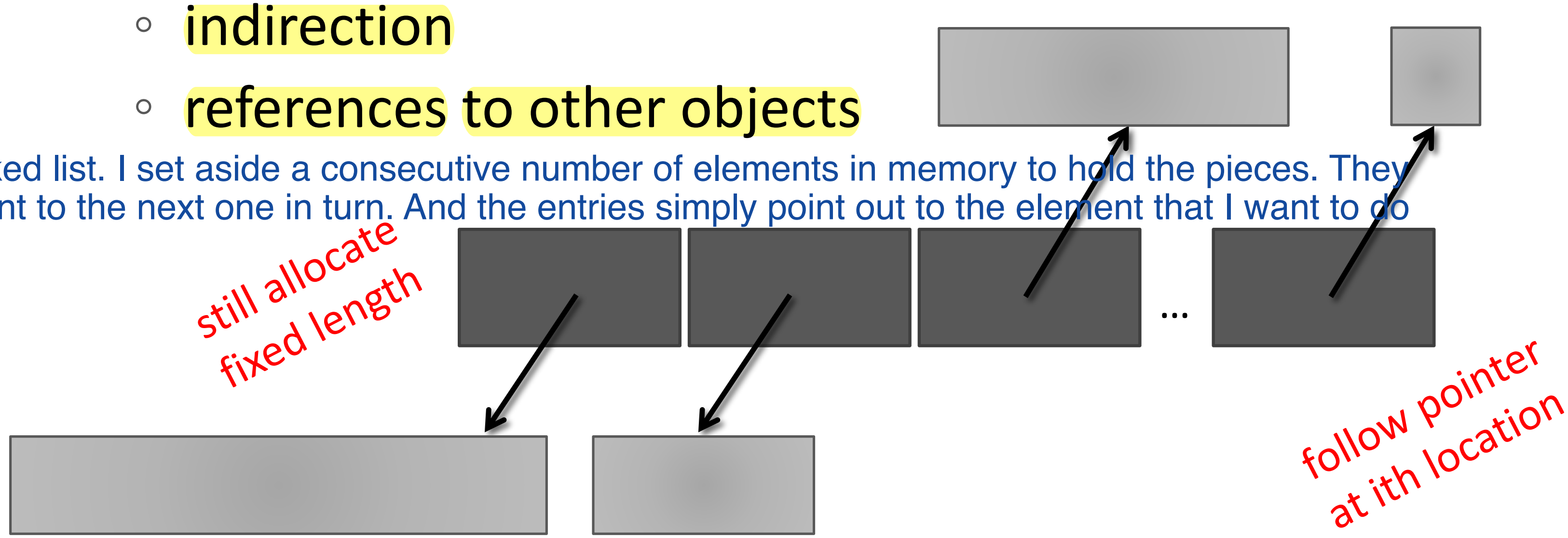
Assumes we can retrieve element of list in constant time

If my list is all integers, then they're smaller than some overall size. I could reserve, say, 4 bytes of memory to store each integer. to represent the list, I would just set aside 4 times the length of the list of consecutive elements of memory to store things in. then if I want to get to the i-th element, I know that I've allocated that 4 bytes, for example, to each one. I know that the i-th element is that whatever the location of the base element, or first element is, plus 4 times i elements down. So I can go exactly to that location in memory to pull it out

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



linked list. I set aside a consecutive number of elements in memory to hold the pieces. They point to the next one in turn. And the entries simply point out to the element that I want to do indirection (also called dereferencing): is the ability to reference something using a name, reference, or container instead of the value itself. The most common form of indirection is the act of manipulating a value through its memory address. For example, accessing a variable through the use of a pointer.

Object-oriented programming makes use of indirection extensively

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
```

Running time wise, this is actually going to be better than the brute force kind of search

- 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)**

breaking out of that loop, as we've seen, doesn't prevent the analysis that in the worst case this is still going to be linear.

USE BISECTION SEARCH

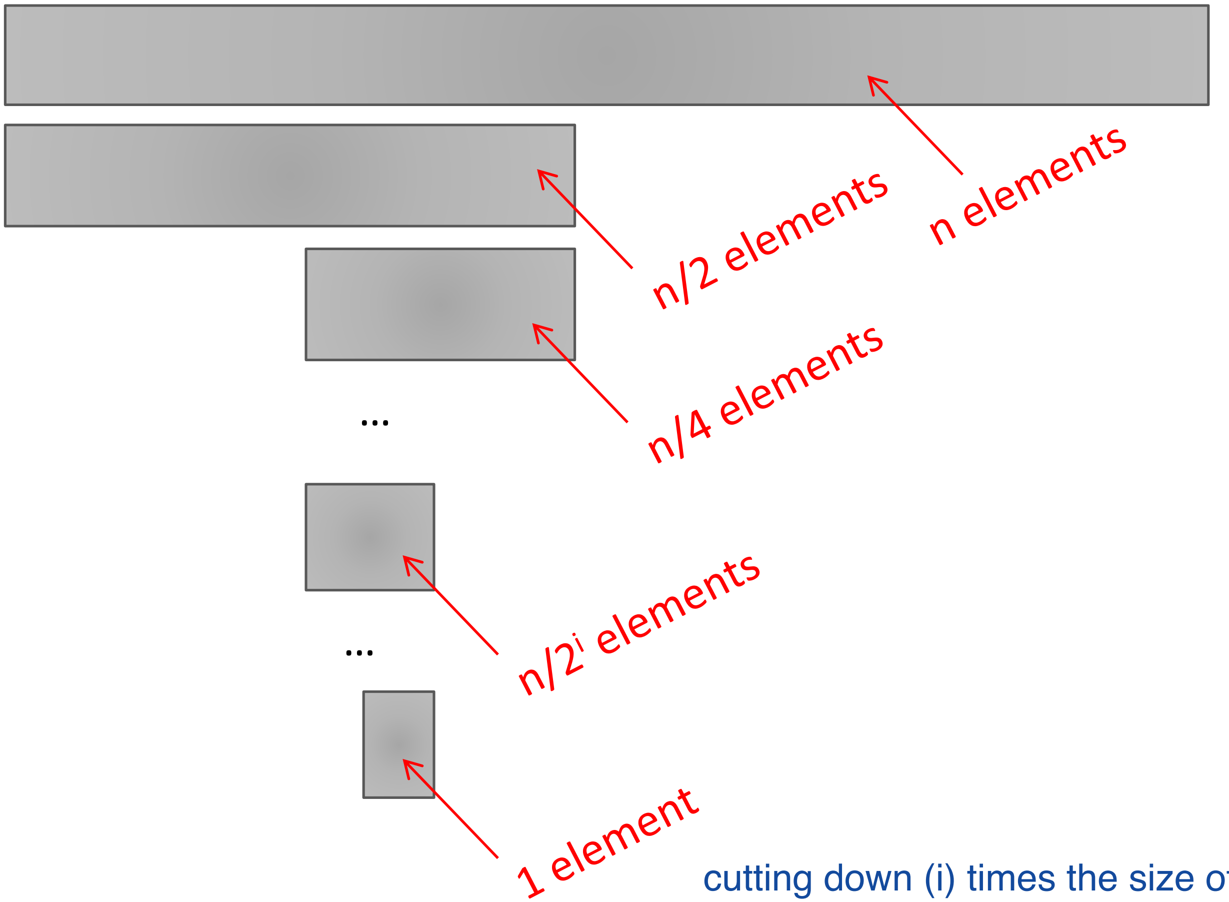
1. Pick an index, `i`, that divides list in half
2. Ask if `L[i] == e`
3. If not, ask if `L[i]` is larger or smaller than `e`
4. Depending on answer, search left or right half of `L` for `e`

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

  Once I found it, I return true.
  If I can't find it, I return false

BISECTION SEARCH COMPLEXITY ANALYSIS



- finish looking through list when

  $1 = n/2^i$

  so i = log n

  i = steps

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

cutting down (i) times the size of the problem (n) by a constant factor (2) at each stage and finish looking through the list when n/ 2**i is just 1 (have reduced size of problem to only one element)

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

base cases

find the halfway point through sorted list, ask: is it bigger than the thing I'm searching?

recursive calls

if it is, I'm going to search only 1/2 of the list. And if it isn't, I'm going to search a different half of the list (until get down to base case)

that's going to add up to a larger complexity, that is bigger than I want because of that cost of actually copying the list (each recursive call)

BISECTION SEARCH IMPLEMENTATION 2

I'm cutting down half the search, but not copying the list, I'm just changing some numbers

```
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)
```

arguments-- the lower half of the list I'm searching and the higher half of the list I'm searching

if the two pointers are at the same place, there's only one element there, check if it's the thing I'm looking for.

Otherwise, I find the midpoint

If, at the midpoint, I have the thing I'm looking for, return true

Otherwise, if the thing at midpoint is bigger than what I'm looking for: if there's nothing left to search, I just return false.

Otherwise, I call it again with the same low point, but now my high point is reduced to the midpoint

NOT constant

Otherwise, change my low point and keep the same high point

NOT constant

that's not constant because of the recursive call. But within it, it is a constant amount of work

base case

call helper function

low pointer at the beginning of the list and the high pointer at the upper end of the list

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)

size of the problem

- 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!

if I'm only going to do it once, it's probably not worth doing the sort and then the search. I might as well just do a linear search.
but I can amortize the cost of the sort over many searches. in many cases, I might simply want to sort the list once, but then do multiple searches

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

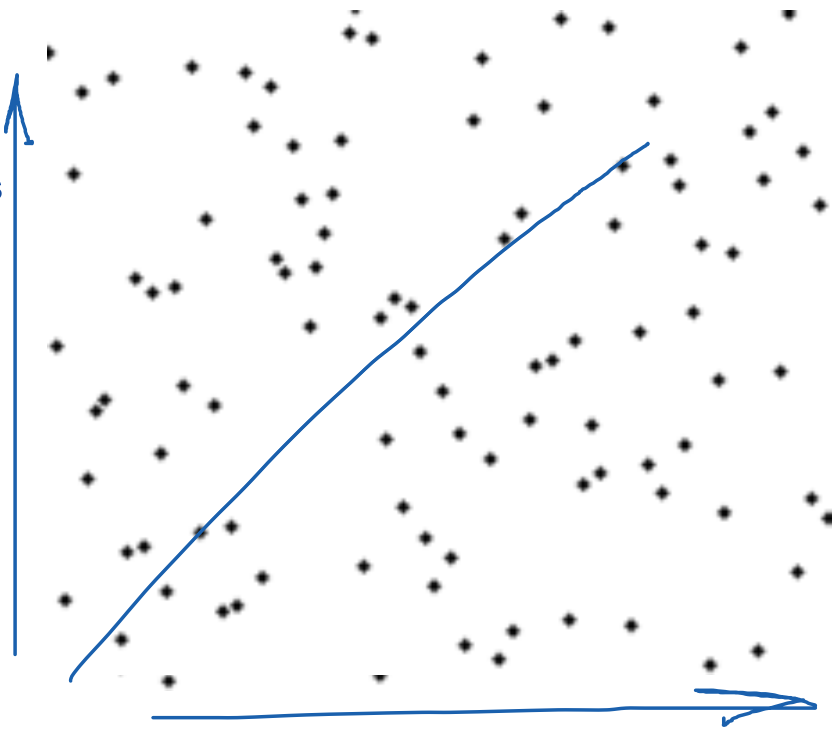
cost of doing k binary searches on presorted list just doing k linear searches

- 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



randomly assign the elements into the list.
And then, I look at them and say are they in order?
And if they aren't, I randomly assign them again

COMPLEXITY OF BOGO SORT

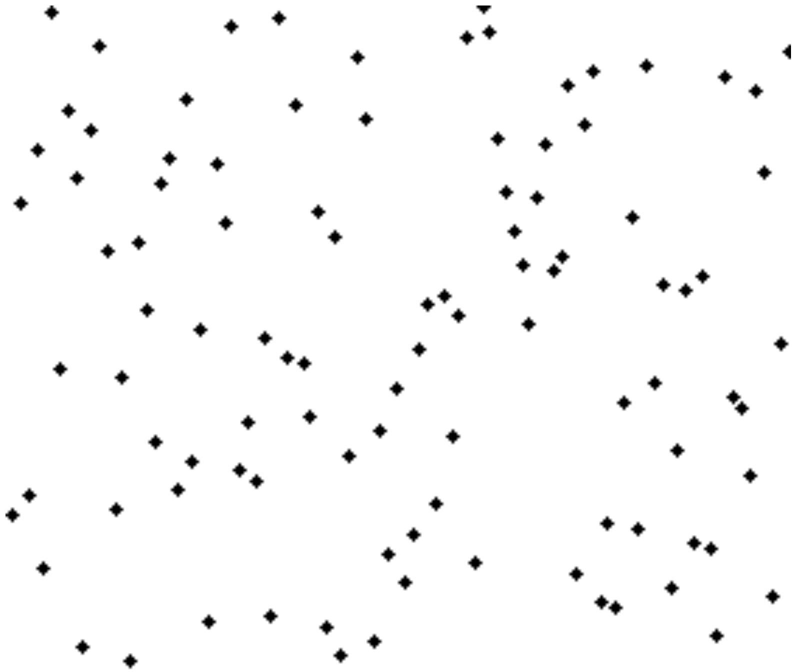
```
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- best case: **O(n) where n is len(L)** to check if sorted
- worst case: O(?) it is **unbounded** if really unlucky

because there's no guarantee I will ever
come up with a random solution that comes up with this

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
```

set up a flag

O(len(L))

let j go from one up to the length of the list

O(len(L))

run through loop while this flag is false

compare successive elements

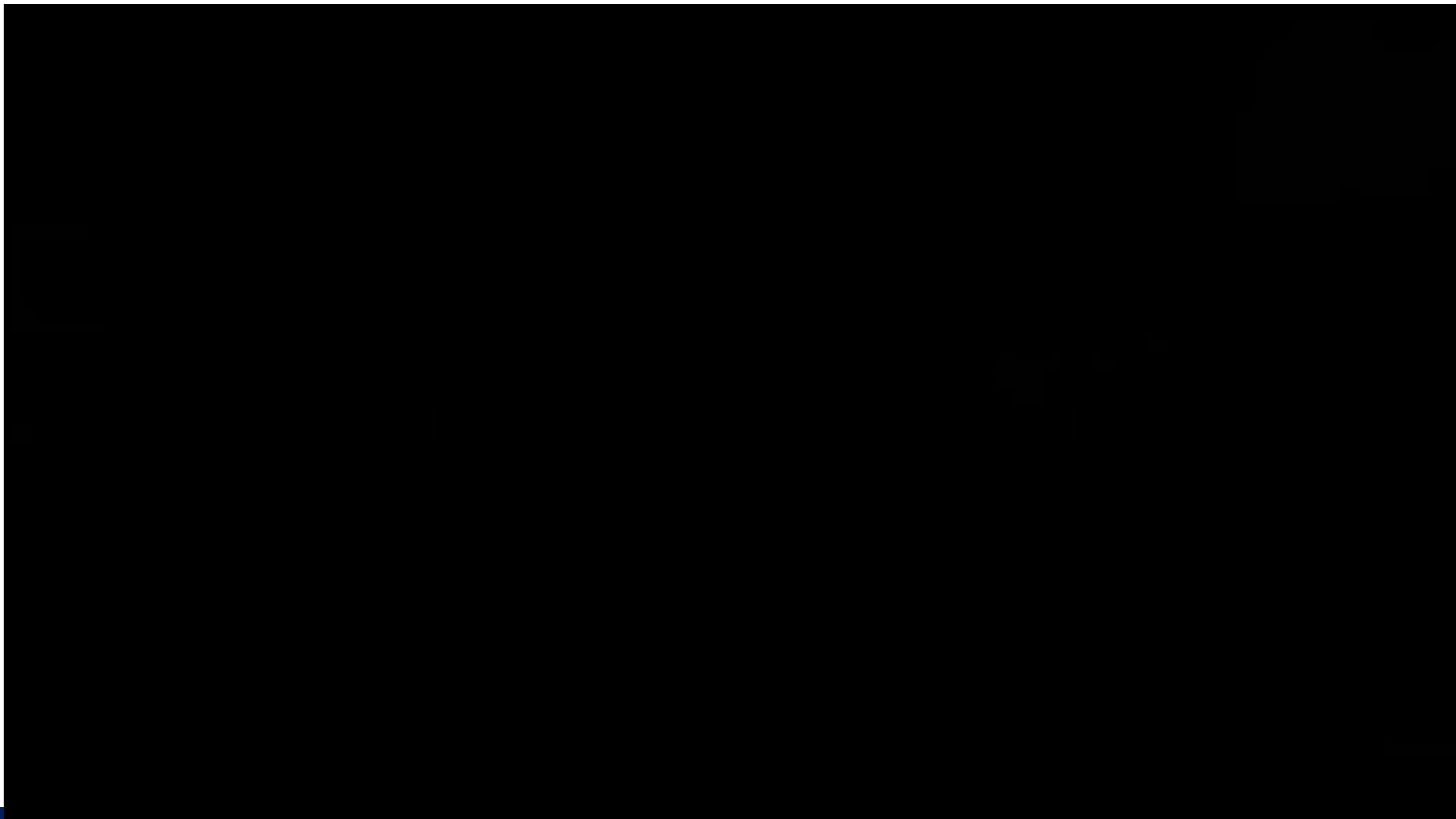
if first element j-1 of consecutive pair is bigger, set the flag to false and swap elements
And then I look at next comsecutive pair

- 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

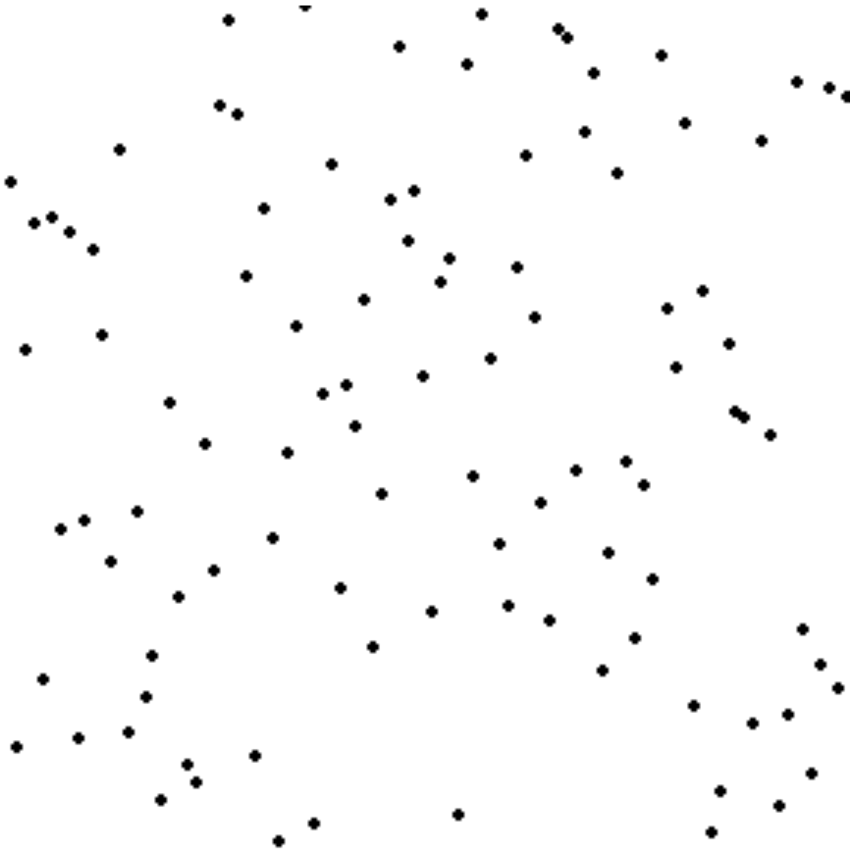
SELECTION SORT

- 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 property that will hold true at each stage of this algorithm
  - 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
```

as long as I don't get to the length of the list, I'm going to run through the loop

len(L) times → O(len(L))

len(L) – suffixSt times → O(len(L))

ranging from the end of the things that are already sorted to the end of the list, compare. And if the thing I'm looking at at that point is less than the thing I'm looking at at the end of my suffix, I'm just going to do a little swap. increase my index by one and move on

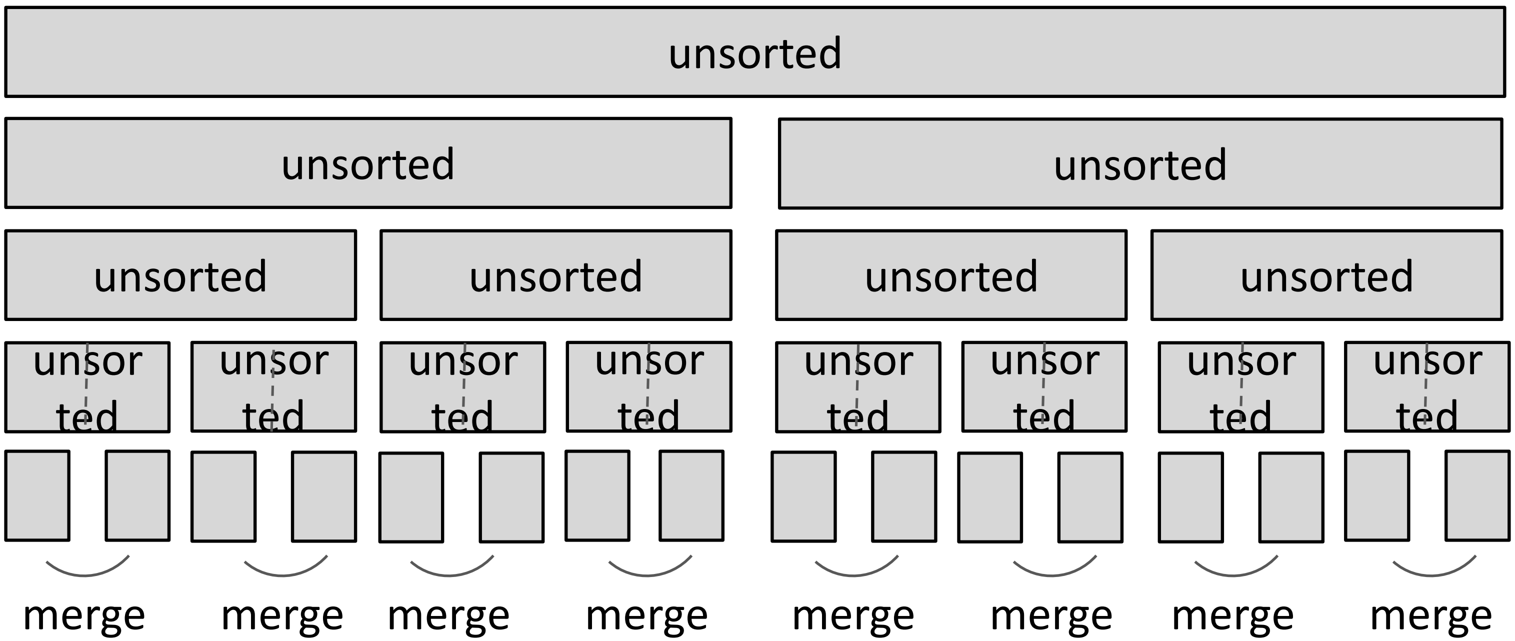
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

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

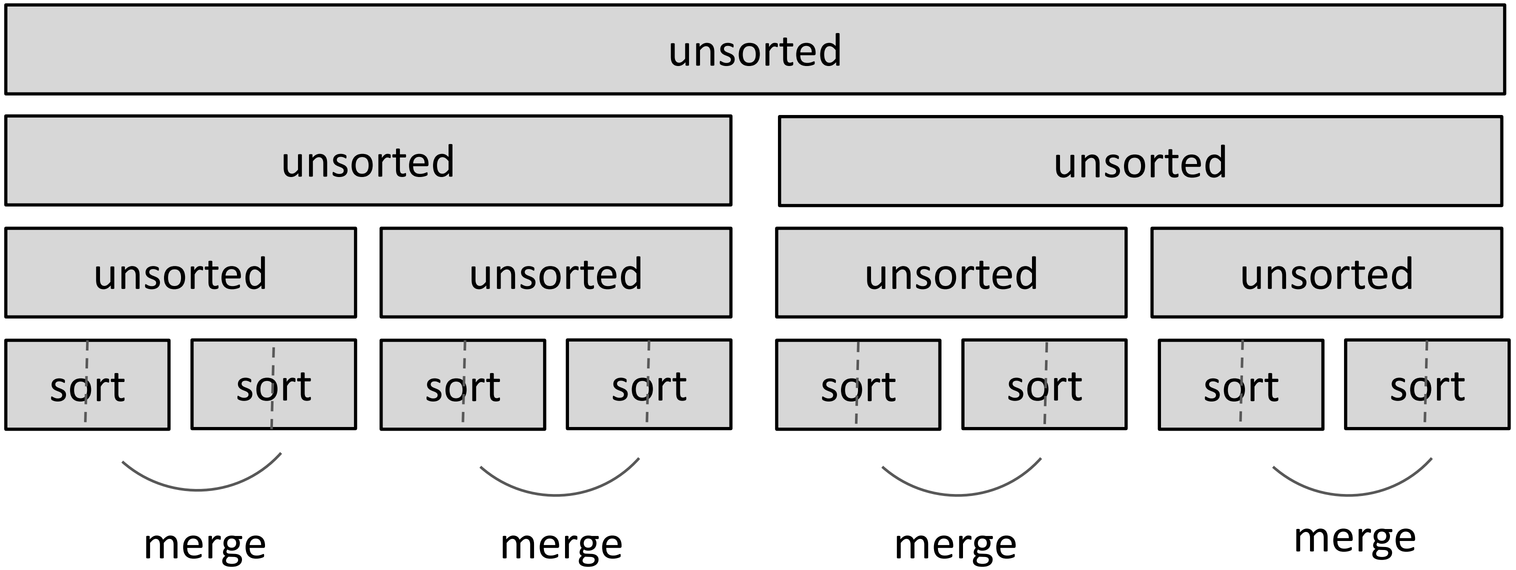
- divide and conquer



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

MERGE SORT

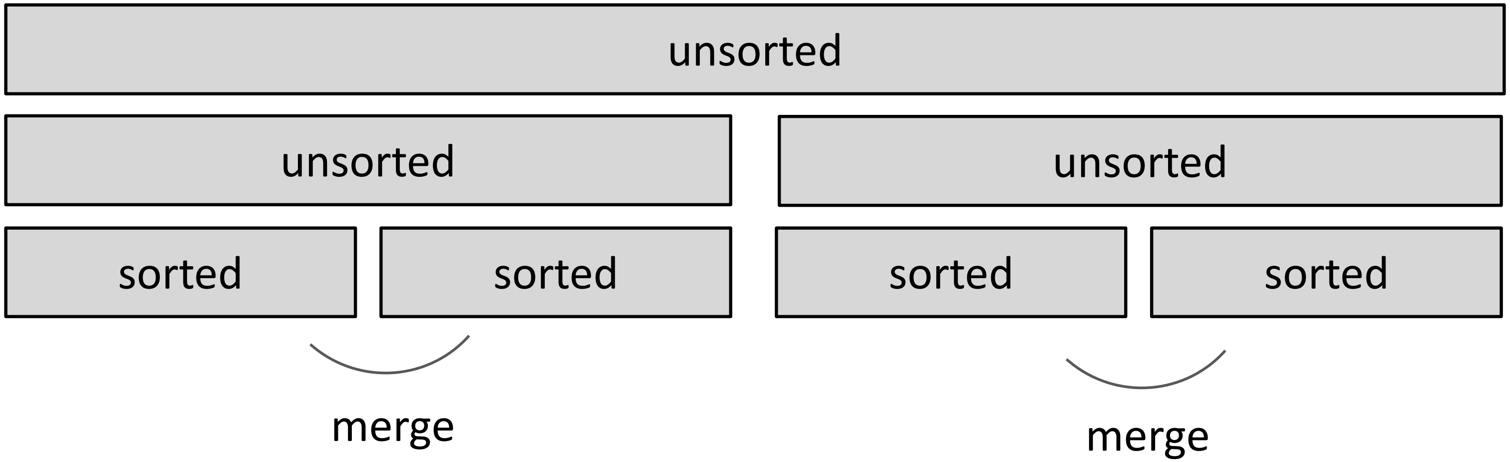
- divide and conquer



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

MERGE SORT

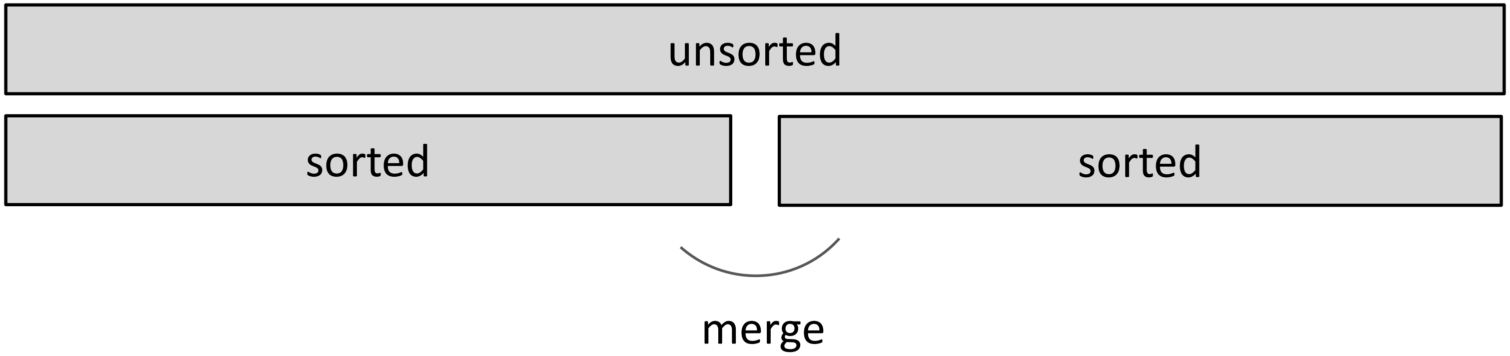
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



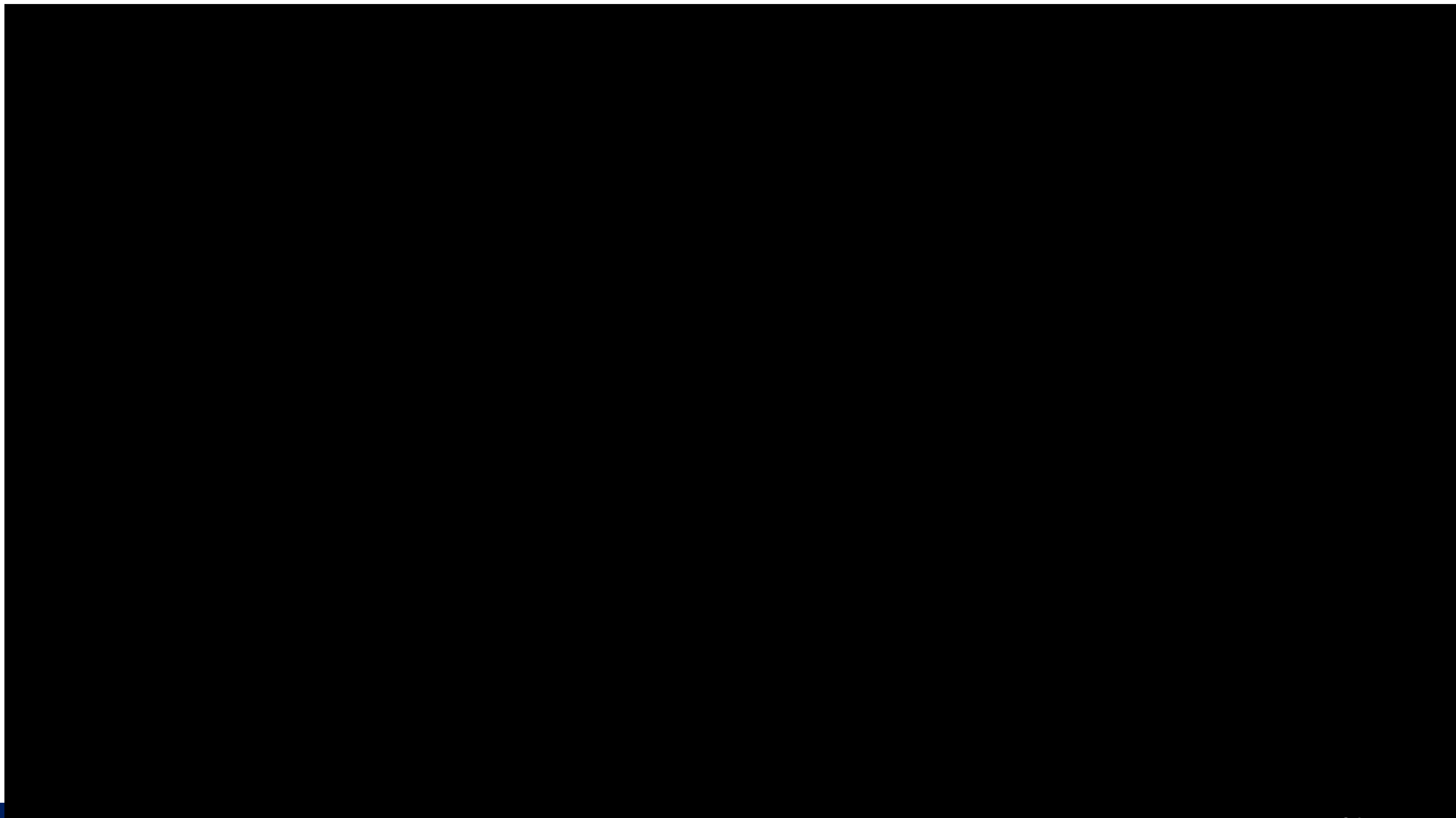
- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer – done!

sorted

MERGE SORT WITH MIT STUDENTS



MERGE SORT DEMO

starts with smaller pieces, does a merge sort on those smaller pieces, and keeps doing that until it can do a merge sort on the entire element

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]

look at the first element in each and put smallest one into the result.

Having done that, one of the lists is now smaller

until I get to a stage where one of the lists is empty, at which case I simply copy the remainder of the list that's not empty to the end of the list.

MERGING SUBLISTS STEP

given a left and right sublist
that we know are sorted

I'm not making copies of the list.
I'm simply looking at the elements

```
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
```

indices that walk down the list

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

keep doing that until one of the lists is exhausted

in either of those cases, just add in the remainder of that list

if the left one is smaller, I'm going to add that element into the result (append is a constant) and change index i

if the right element is smaller, I add it in, and I change j

when right sublist is empty

when left sublist is empty

When the right sublist is empty, I copy the rest of the left sublist.
-
When the left sublist is empty, I copy the remaining elements in the right sublist.

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)
```

if there's only 0 or 1 elements there, I'm done. Just return a copy of the list and I'm all set

base case

Find the midpoint. Break this in half.

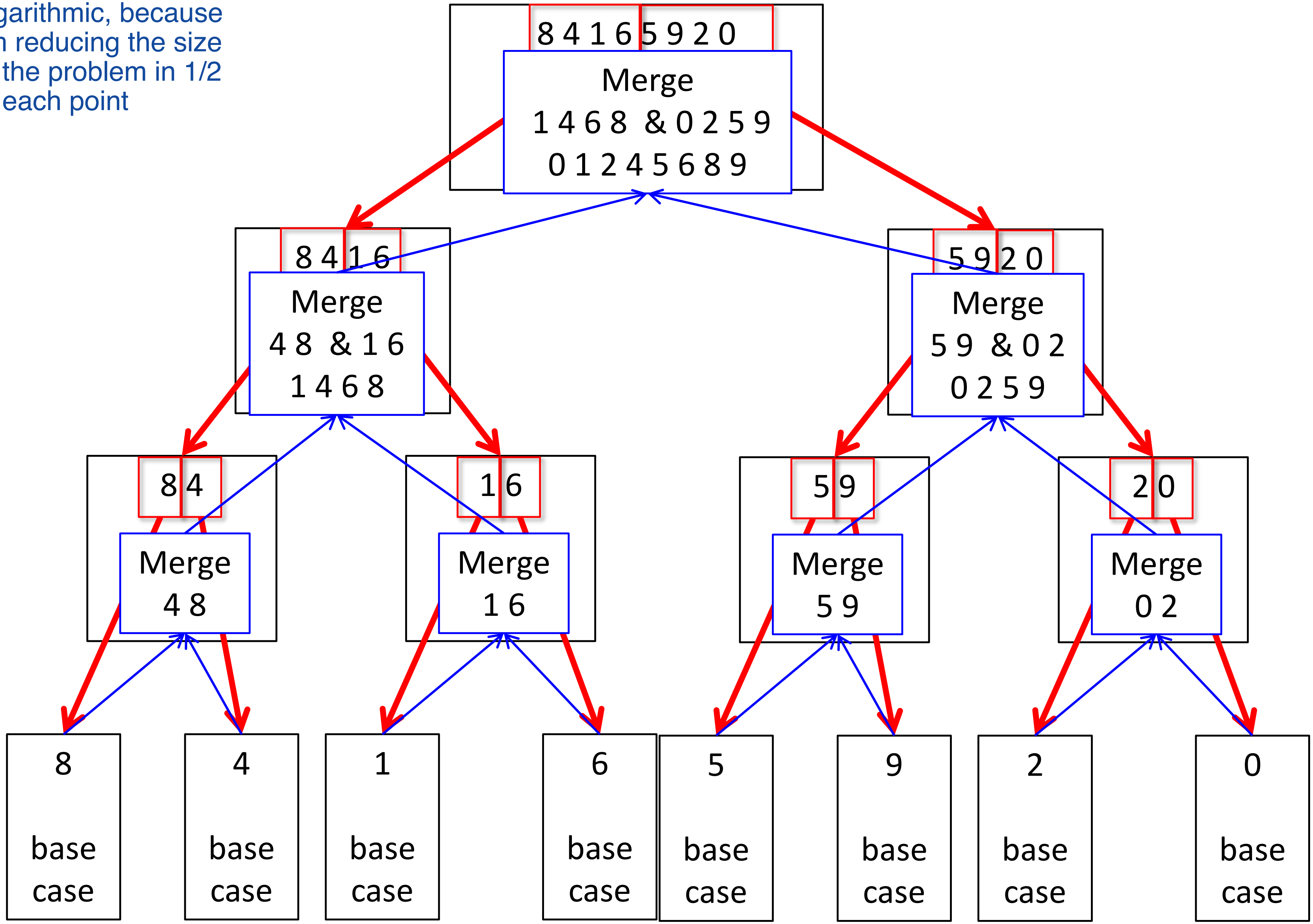
do a merge sort on half of the list. do a merge sort on the left side of the list, then the right side of the list

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

logarithmic, because
I'm reducing the size
of the problem in 1/2
at each point



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 k recursive calls is order n to the k.
I'm done when n to the k is of size 1.
That's when k is log n

- at **second recursion level**
  - n/4 elements in each list
  - two merges → O(n) where n is len(L)

The size of the list is smaller,
but I got more lists

- 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)**

it's not quite as nice as logarithmic. It's not quite as nice as linear.
But it's a lot better than quadratic, or certainly than exponential

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

if I want to stop the computation after I
get some number of the best elements out,
I could do that without having to sort the rest of the list.
And that's better than doing bubble sort