SORTING ALGORITHMS

(download slides and .py files to follow along)

6.100L Lecture 24

Ana Bell

SEARCHING A SORTED LIST
-- n is len(L)

- Using **linear search**, search for an element is **Θ(n)**
- Using **binary search**, can search for an element in **Θ(logn)**
  - assumes the **list is sorted**!
- When does it make sense to **sort first then search**?

Time to sort | Time for binary search | Time for linear search

$\boxed{\text{SORT}} + \boxed{\Theta(\texttt{log n})} < \boxed{\Theta(\texttt{n})}$ implies $\text{SORT} < \Theta(\texttt{n}) - \Theta(\texttt{log n})$

When sorting is less than Θ(n)!?!? This is never true!

AMORTIZED COST
-- n is len(L)

- Why bother sorting first?
- **Sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches

Only once!

Do K searches

- SORT + K * Θ(log n) < K * Θ(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

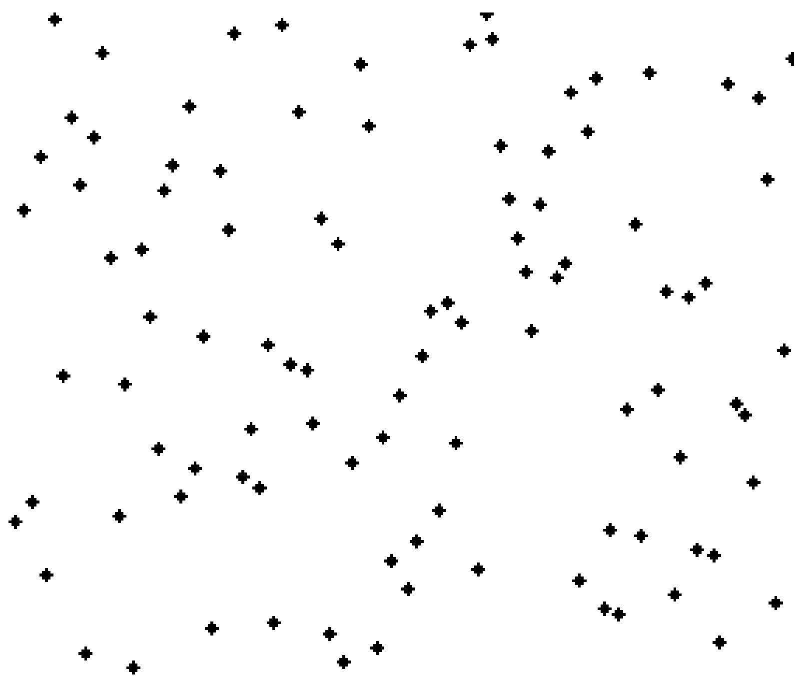
COMPLEXITY OF BOGO SORT

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

- Best case: **$\Theta(n)$ where n is len(L)** to check if sorted
- Worst case: $\Theta(?)$ 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

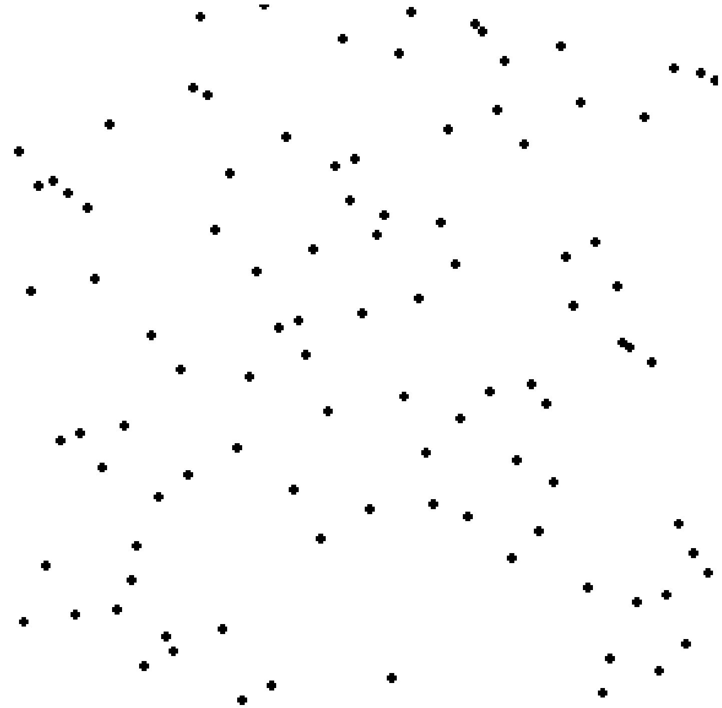
```
def bubble_sort(L):
    did_swap = True
    while did_swap:
        did_swap = False
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                did_swap = True
                L[j],L[j-1] = L[j-1],L[j]
```

Θ(len(L)) (annotating `while did_swap:`)

Θ(len(L)) (annotating `for j in range(1, len(L)):`)

- Inner for loop is for doing the **comparisons**
- Outer while loop is for doing **multiple passes** until no more swaps
- **$\Theta(n^2)$ 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**
- 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 ith 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 → Θ(len(L))

len(L) – i times → Θ(len(L))

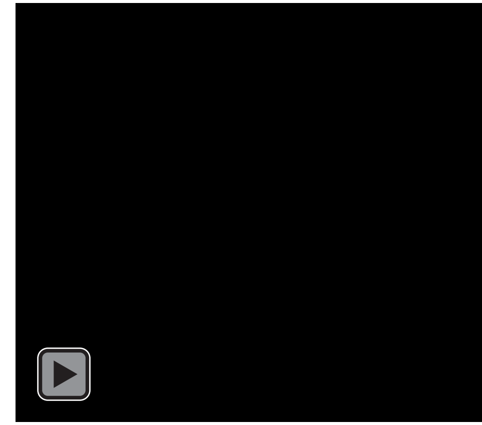
- Complexity of selection sort is **$\Theta(n^2)$ where n is len(L)**
    - Outer loop executes len(L) times
    - Inner loop executes len(L) – i times, on avg len(L)/2
- Can also think about how many times the comparison happens over both loops: say n = len(L)
    - Approx $1+2+3+\ldots+n = (n)(n+1)/2 = n^2/2+n/2 = \Theta(n^2)$

VARIATION ON SELECTION SORT: don't swap every time

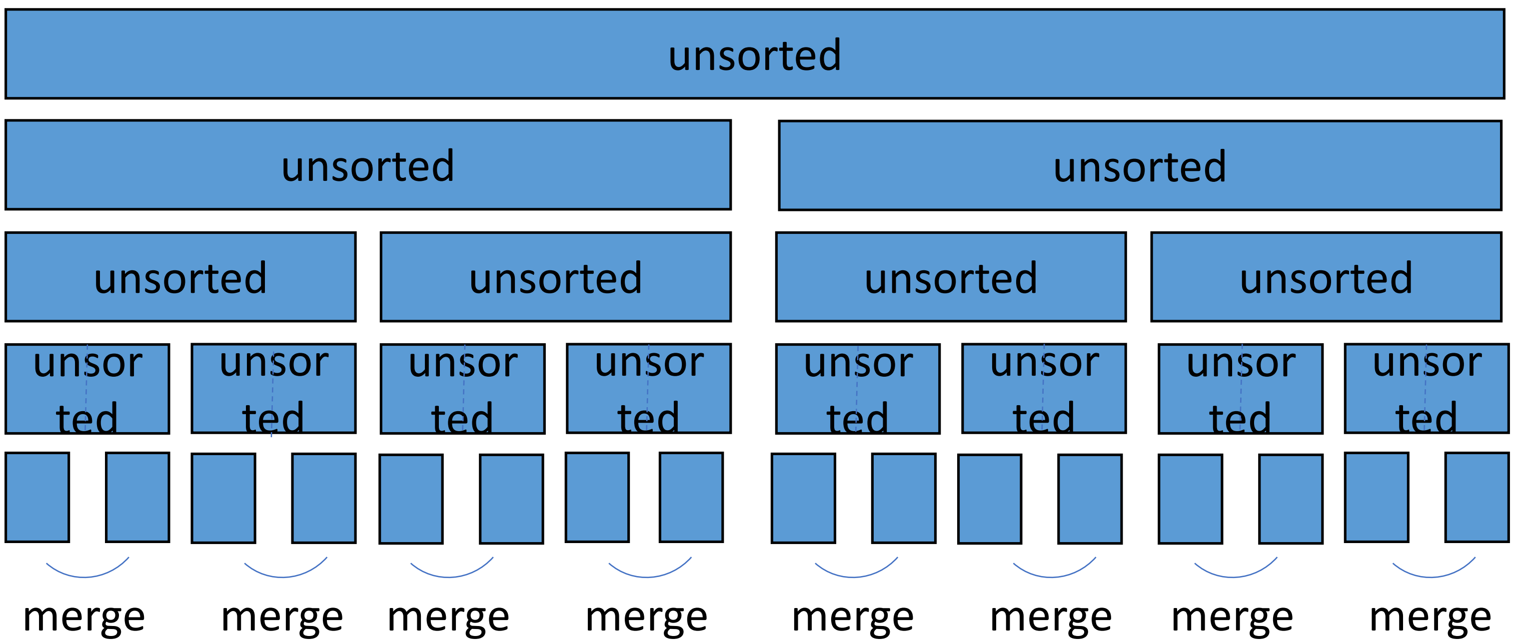
MERGE SORT

- Use a **divide-and-conquer** approach:
  - If list is of length 0 or 1, already sorted
  - If list has more than one element, split into two lists, and sort each
  - Merge sorted sublists
    - Look at first element of each, move smaller to end of the result
    - When one list empty, just copy rest of other list

MERGE SORT



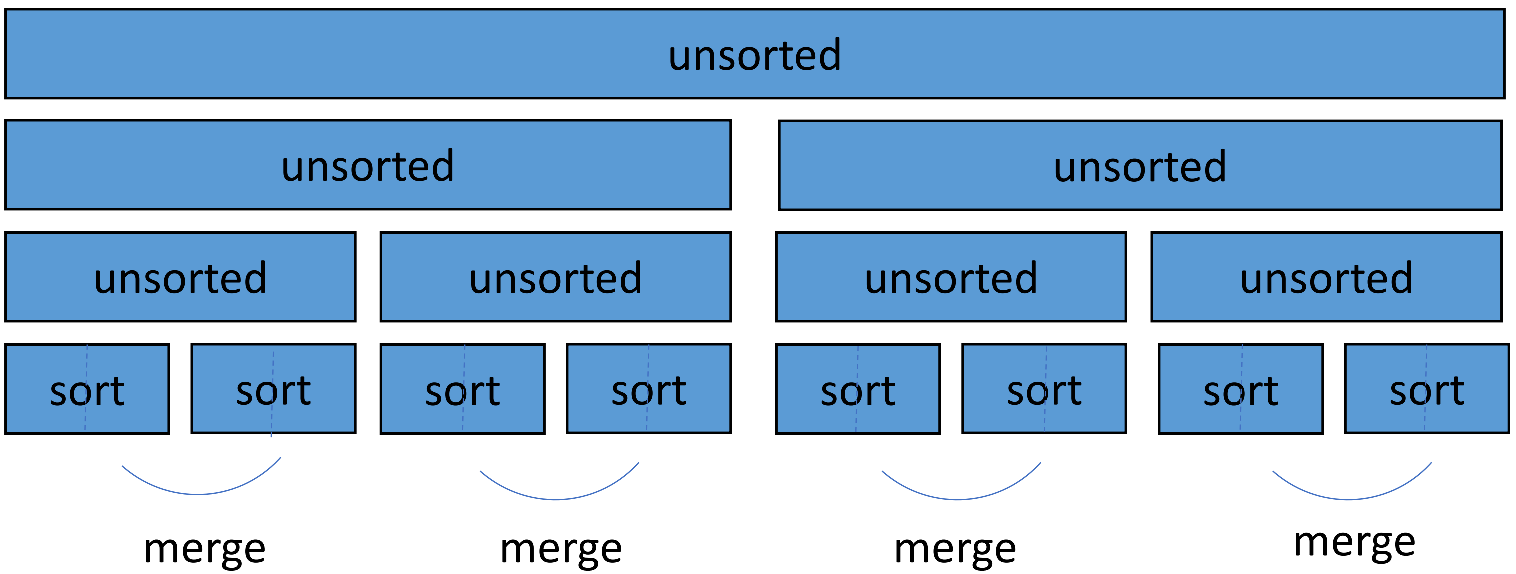
- Divide and conquer



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

MERGE SORT

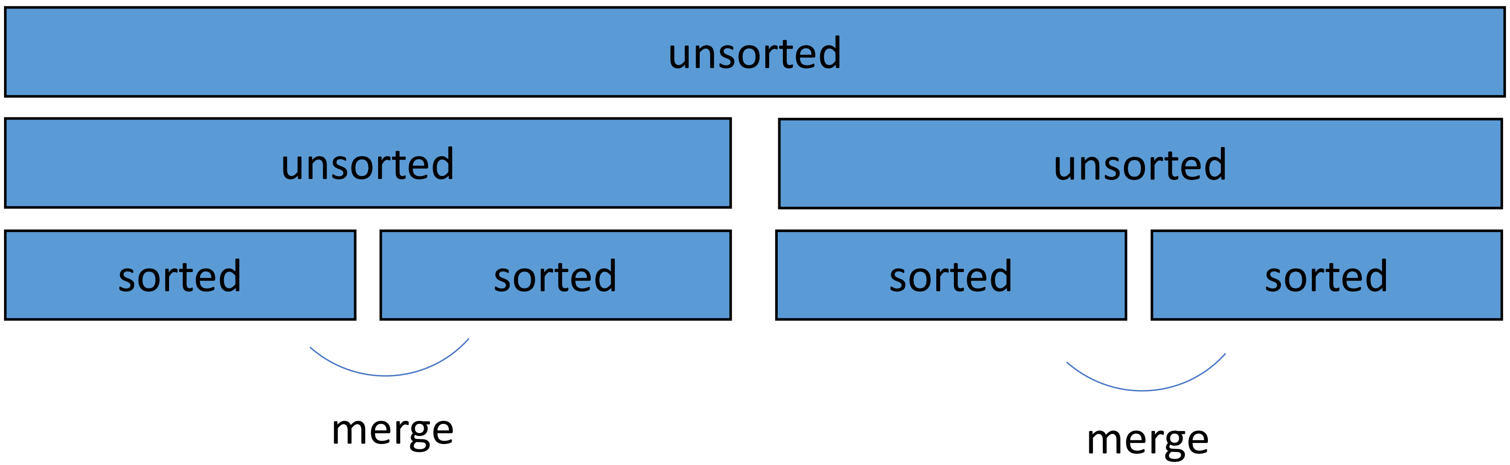
- Divide and conquer



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

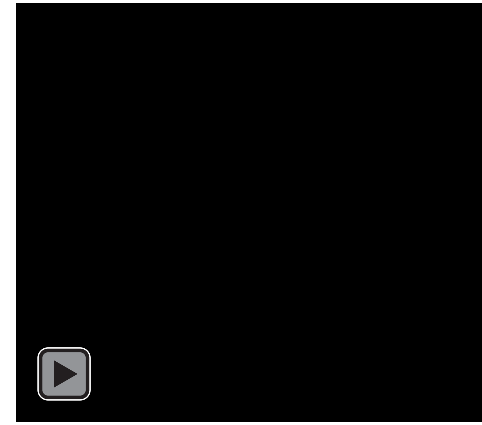
MERGE SORT

- Divide and conquer

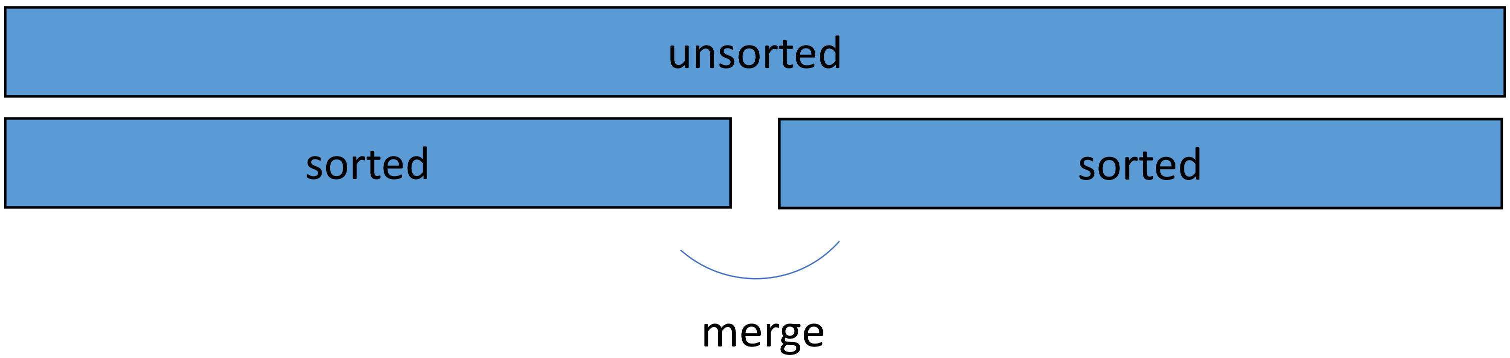


- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT



- Divide and conquer



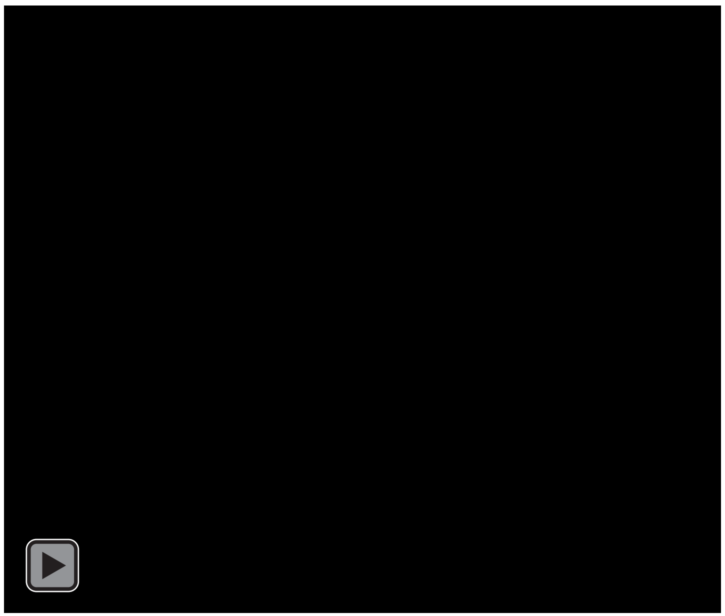
- Merge sorted sublists
- Sublists will be sorted after merge

MERGE SORT

- Divide and conquer – done!

sorted

MERGE SORT DEMO



1. Recursively divide into subproblems
2. Sort each subproblem using linear merge
3. Merge (sorted) subproblems into output list

CLOSER LOOK AT THE MERGE STEP (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 STEP

- Go through two lists, only one pass
- Compare only **smallest elements in each sublist**
- Θ(len(left) + len(right)) copied elements
- Worst case Θ(len(longer list)) comparisons
- **Linear in length of the lists**

FULL MERGE SORT ALGORITHM -- RECURSIVE

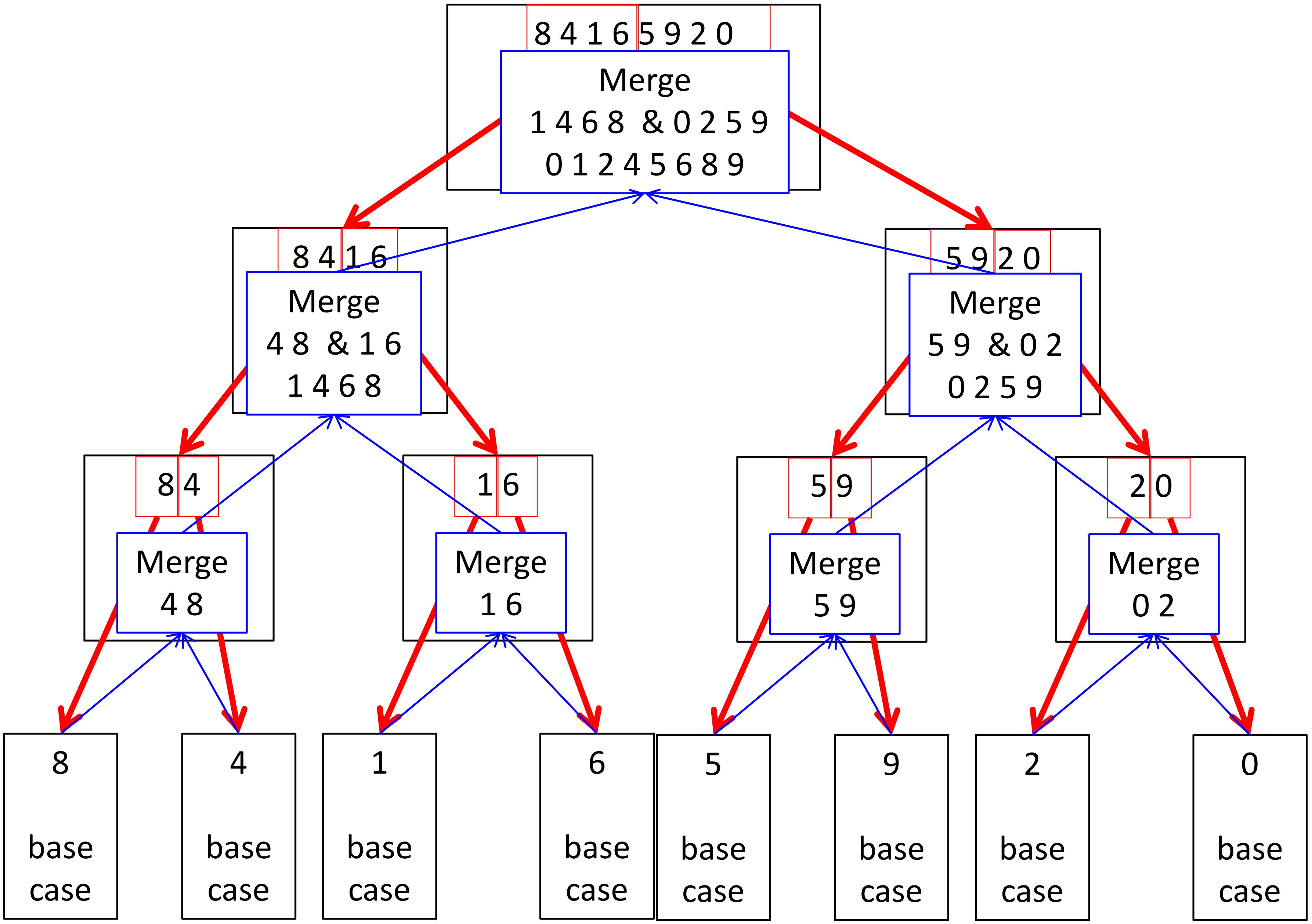
```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

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

- Each level
  - At **first recursion level**
    - n/2 elements in each list, 2 lists
    - One merge → $\Theta(n) + \Theta(n) = \Theta(n)$ where n is len(L)
  - At **second recursion level**
    - n/4 elements in each list, 4 lists
    - Two merges → $\Theta(n)$ where n is len(L)
  - And so on…
- **Dividing list in half** with each recursive call gives our levels
  - $\Theta(\log n)$ where n is len(L)
  - Like bisection search: $1 = n/2^i$ tells us how many splits to get to one element
- Each recursion level does $\Theta(n)$ work and there are $\Theta(\log n)$ levels, where n is len(L)
- Overall complexity is **$\Theta(n \log n)$ where n is len(L)**

SORTING SUMMARY
-- n is len(L)

- Bogo sort
  - Randomness, unbounded $\Theta()$
- Bubble sort
  - $\Theta(n^2)$
- Selection sort
  - $\Theta(n^2)$
  - Guaranteed the first i elements were sorted
- Merge sort
  - $\Theta(n \log n)$
- **$\Theta(n \log n)$ is the fastest a sort can be**

COMPLEXITY SUMMARY

- Compare **efficiency of algorithms**
  - Describe **asymptotic** order of growth with Big Theta
  - **Worst case** analysis
- Saw different classes of complexity
  - Constant
  - Log
  - Linear
  - Log linear
  - Polynomial
  - Exponential
- A priori evaluation (before writing or running code)
- Assesses algorithm independently of machine and implementation
- Provides direct insight to the **design** of efficient algorithms