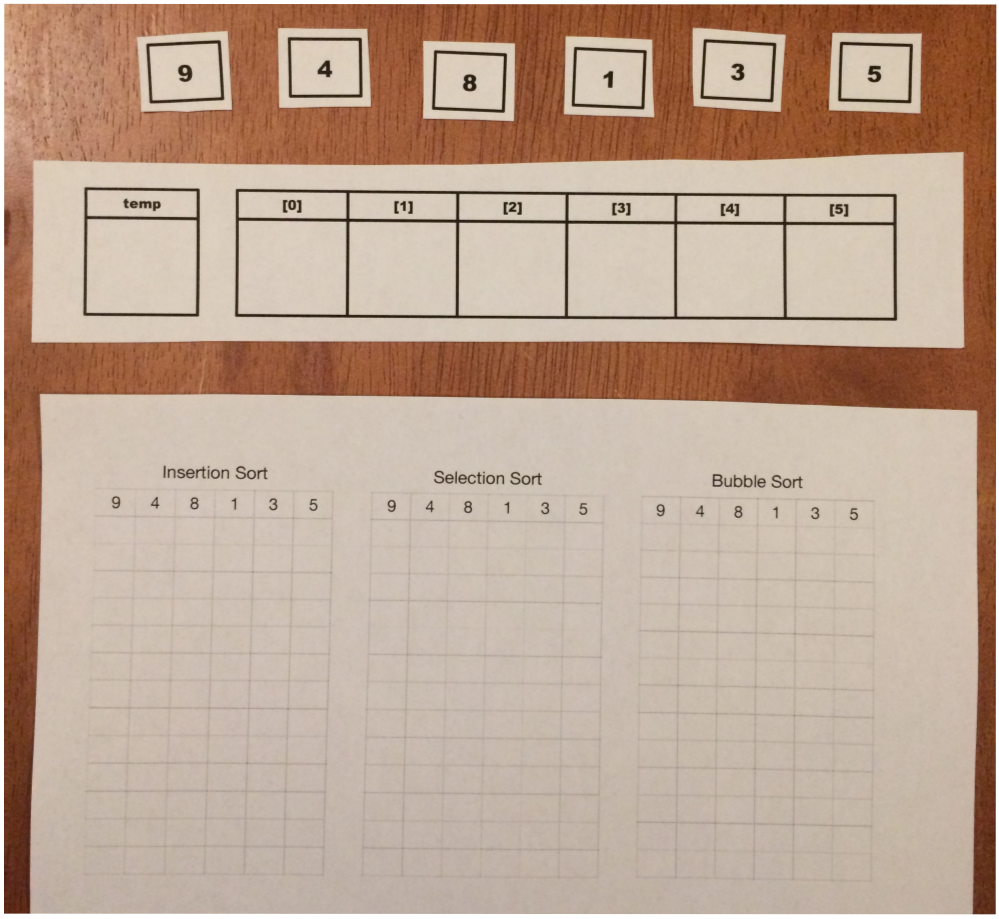
Topic: Basic Algorithms -> Sorting

Goals: By the end of this topic, we will…

- *introduce basic algorithms and analysis*
- *discuss computational sorting and sorting algorithms*

Acknowledgements: These class notes build on the content of my previous courses as well as the work of R. Jordan Crouser, and Jeffrey S. Castrucci.

(Optional) Pre-Class Prep. Arts and Crafts Fun



Opening Activity

Sort these numbers from smallest to largest:

9 4 8 1 3 5

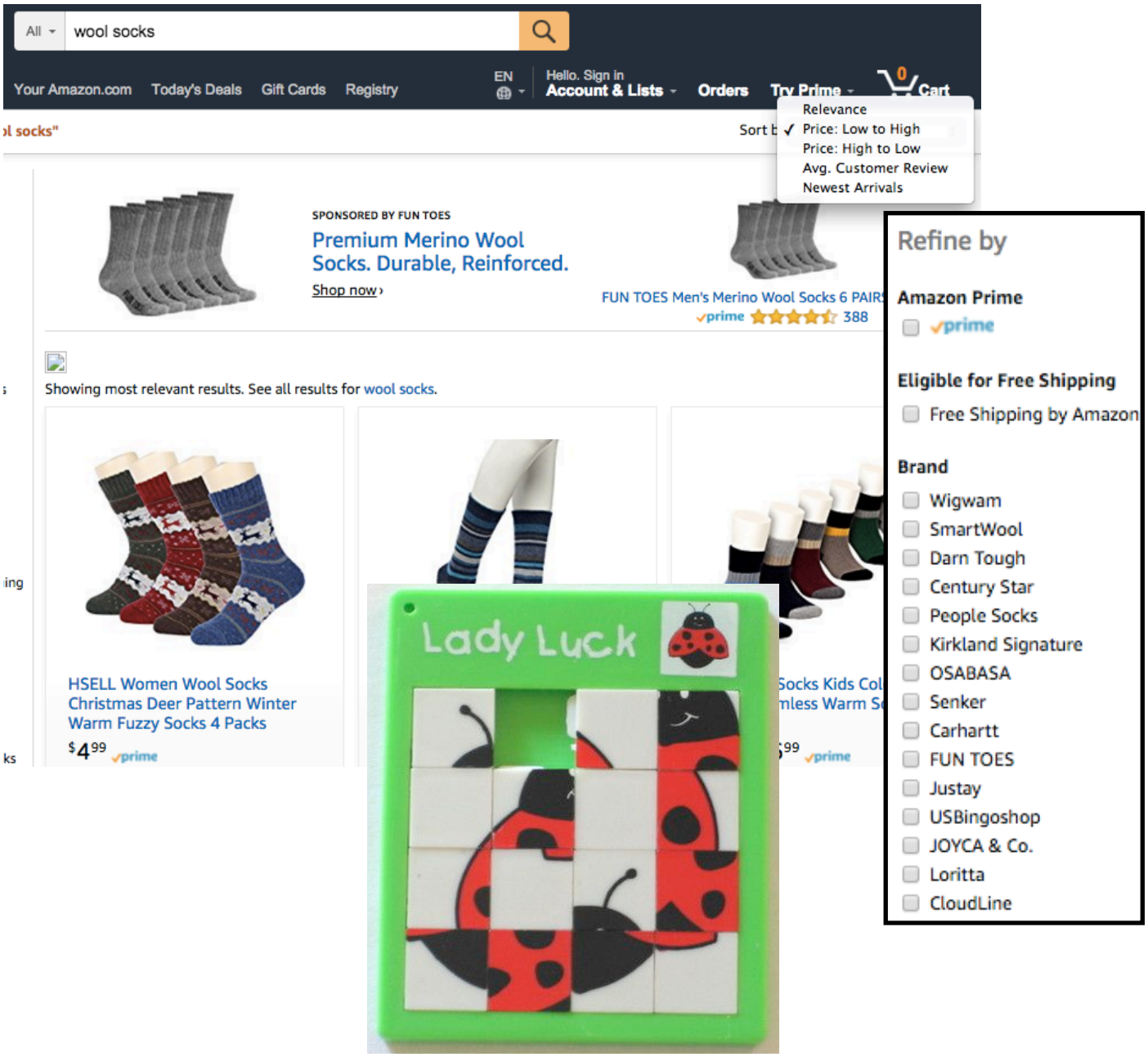
Answer: 1 3 4 5 8 9

How did you do that?? What makes you different than a computer?

Why Sorting???

Barack Obama on Sorting @Google:
http://www.youtube.com/watch?v=k4RRi_ntQc8

Everything involves (searching) and sorting.

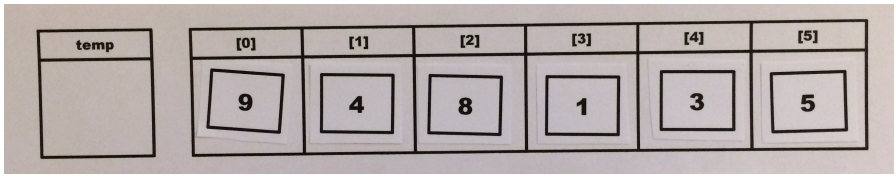


Exercise: Discover Sorting

Sort the numbers using only the boxes available.

You can only move one number at a time between boxes.
On the back of the handout, write down the steps you used to sort the numbers.

9 4 8 1 3 5



What observations did you make?

Popular Sorting Algorithms

Simple: **Selection, Insertion**
Efficient: Merge, Heap, Quick
Bubbling: **Bubble**, Shell, Comb
Distributed: Counting, Bucket, Radix

Selection Sort

Sorts the list incrementally, by **finding the smallest value among the unsorted elements**. The initial list is assumed to be unsorted.

Wiki Insertion Sort Visual:
https://en.wikipedia.org/wiki/Selection_sort#/media/File:Selection-Sort-Animation.gif

Selection Sort:

```
def SSort(L):
    for i in range(len(L)):
        index_of_smallest = i

          # find the smallest element.
        for j in range(i + 1, len(L)):
            if L[j] < L[index_of_smallest]:
                index_of_smallest = j

        temp = L[index_of_smallest]
        L[index_of_smallest] = L[i]
        L[i] = temp
        print(L)

L = [9,4,8,1,3,5]
SSort(L)
```

Arrange the set of numbers in ascending order using a “selection sort”.
Every time you finish one iteration write down the list.
Start with: [9, 4, 8, 1, 3, 5]
End with: [1, 3, 4, 5, 8, 9]

Solution:
[9, 4, 8, 1, 3, 5]
[1, 4, 8, 9, 3, 5]
[1, 3, 8, 9, 4, 5]
[1, 3, 4, 9, 8, 5]
[1, 3, 4, 5, 8, 9]
[1, 3, 4, 5, 8, 9]
[1, 3, 4, 5, 8, 9]

Insertion Sort

Sorts the list **incrementally**, starting with a sublist of length 1 and adding an additional elements by “inserting” it into the already sorted list.

Wiki Insertion Sort Visual:
https://en.wikipedia.org/wiki/Insertion_sort#/media/File:Insertion-sort-example-300px.gif

Insertion Sort:

```
def ISort(L):
    for i in range(len(L)):
        temp = L[i]

        # find the position, i, where value should go
        while i > 0 and L[i - 1] > temp:
            L[i] = L[i - 1]
            i = i - 1

        L[i] = temp
        print(L)

L = [9,4,8,1,3,5]
ISort(L)
```

Arrange a set of numbers in ascending order using a “insertion sort”.
Every time you finish one iteration write down the list.
Start with: [9, 4, 8, 1, 3, 5]
End with: [1, 3, 4, 5, 8, 9]

Solution:
[9, 4, 8, 1, 3, 5]
[9, 4, 8, 1, 3, 5]
[4, 9, 8, 1, 3, 5]
[4, 8, 9, 1, 3, 5]
[1, 4, 8, 9, 3, 5]
[1, 3, 4, 8, 9, 5]
[1, 3, 4, 5, 8, 9]
[1, 3, 4, 5, 8, 9]

Bubble Sort:

Compare each pair of values (repeatedly).

Design:

- pass through the list num-1 times
- each pass, compare each pair of successive entries
- if a pair is in descending order, swap the pair

Wiki Bubble Sort Visual:
https://en.wikipedia.org/wiki/Bubble_sort#/media/File:Bubble-sort-example-300px.gif

```
def BSort(L):
    # Repeat Bubble for each possible trade.
    for h in range(len(L) - 1):

        # Bubble once through the unsorted section
        #   to move the largest item to index end.
        for i in range(len(L) - 1):
            if L[i] > L[i + 1]:
                temp = L[i]
                L[i] = L[i+1]
                L[i+1] = temp
                print(L)

L = [9,4,8,1,3,5]
BSort(L)
```

Arrange a set of numbers in ascending order using a “bubble sort”.
Every time you swap two numbers write down the order.
Start with: [9, 4, 8, 1, 3, 5]
End with: [1, 3, 4, 5, 8, 9]

Answer:
[9, 4, 8, 1, 3, 5]
[4, 9, 8, 1, 3, 5]
[4, 8, 9, 1, 3, 5]
[4, 8, 1, 9, 3, 5]
[4, 8, 1, 3, 9, 5]
[4, 8, 1, 3, 5, 9]
[4, 1, 8, 3, 5, 9]
[4, 1, 3, 8, 5, 9]
[4, 1, 3, 5, 8, 9]
[1, 4, 3, 5, 8, 9]
[1, 3, 4, 5, 8, 9]
[1, 3, 4, 5, 8, 9]

Python comparison trick....

```
temp = L[i]                              #L[i], L[i + 1] = L[i + 1], L[i]
L[i] = L[i+1]
L[i+1] = temp
```

Observations:

- at the end of the first pass (of the outer loop)
  - the largest number is at the end of list
- at the end of the second pass (of the outer loop)
  - the two largest numbers are ordered correctly
- at the end of the third pass (of the outer loop)
  - the three largest numbers are ordered correctly

Can be take advantage of the above observations?

Bubble Sort (Optimized):

```
def OBSort(L):
    end = len(L) - 1
    while end != 0:
        for i in range(end):
            if L[i] > L[i + 1]:
                temp = L[i]
                L[i] = L[i+1]
                L[i+1] = temp
                print(L)
        end = end - 1

L = [9,4,8,1,3,5]
OBSort(L)
```

Bubble Sort	Optimized Bubble Sort
[9, 4, 8, 1, 3, 5]	[9, 4, 8, 1, 3, 5]
[4, 9, 8, 1, 3, 5]	[4, 9, 8, 1, 3, 5]
[4, 8, 9, 1, 3, 5]	[4, 8, 9, 1, 3, 5]
[4, 8, 1, 9, 3, 5]	[4, 8, 1, 9, 3, 5]
[4, 8, 1, 3, 9, 5]	[4, 8, 1, 3, 9, 5]
[4, 8, 1, 3, 5, 9]	[4, 8, 1, 3, 5, 9]
[4, 1, 8, 3, 5, 9]	[4, 1, 8, 3, 5, 9]
[4, 1, 3, 8, 5, 9]	[4, 1, 3, 8, 5, 9]
[4, 1, 3, 5, 8, 9]	[4, 1, 3, 5, 8, 9]
[1, 4, 3, 5, 8, 9]	[1, 4, 3, 5, 8, 9]
[1, 3, 4, 5, 8, 9]	[1, 3, 4, 5, 8, 9]

The output is the same, how do we know this is more efficient?
How do we define efficient?
What similarities / differences did you notice between the three approaches?

Next time: Analyzing Runtimes and Recursive Sorting Algorithms

Algorithmic Analysis

- **What we want**: a way to get a rough bound (limit) on how many steps an algorithm could take, given input of a particular size
  - If the list has 5 items?
  - What about 10?
  - What about n?
- Often care about: “what’s the worst that could happen?”

Asymptotic (“big-O”) notation

- We can avoid details when they don’t matter, and
  they don’t matter when input size (n) is big enough
- For polynomials: only the leading term matters
- Ignore coefficients, talk about degree: linear, quadratic

  $y = 3x$ vs. $y = 3x^2$

SelectionSort

Given some list of comparable items:

Find the smallest item.
Swap it with the item in position 0.
Repeat finding the next-smallest item, and swapping it
into the correct position until the list is sorted.

BubbleSort

```
While the list is still unsorted:
  For p in [0, ..., n-2]:
    If item in position p is greater than item in position p+1:
      Swap them
```

Takeaways

- Big-O notation gives us a language to talk about and compare algorithms before we implement them (smart!)
- InsertionSort, SelectionSort, and BubbleSort:
  - all the same in the worst case
  - different in the best case
  - what about the average case?
  - is there any algorithm that could do better in the worst case? (yes!)
- As your programs get more complex, it’s helpful to think about the algorithm before you start coding

Divide and Conquer Sorting (***Optional Advanced Topic***)

Let's create a better algorithm using the foundations of bubbleSort.

Recall our `swap` function and our `bubble` function.

```
def swap(iList, i1, i2):
    temp = iList[i1]
    iList[i1] = iList[i2]
    iList[i2] = temp
```

```
def bubble(iList, iLen):
    L = iList
    # Bubble once through the unsorted section to move
    #  the largest item to index end.
    for i in range(iLen - 1):
        if L[i] > L[i + 1]:
            temp = L[i]
            L[i] = L[i+1]
            L[i+1] = temp
```

Discussion: What happens if we divide the list into two and merge the two sorted lists?

Merge Function

- build up a sorted list from the two original lists
- after every comparison add an element to the sorted list.

```
# Return a sorted list with the elements in <lst1> and <lst2>.
# Precondition: <lst1> and <lst2> are sorted.
def merge(lst1: list, lst2: list) -> list:
    index1 = 0
    index2 = 0
    merged = []
    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index1] <= lst2[index2]:
            merged.append(lst1[index1])
            index1 += 1
        else:
            merged.append(lst2[index2])
            index2 += 1
    # Now either index1 == len(lst1) or index2 == len(lst2)
    assert index1 == len(lst1) or index2 == len(lst2)

    return merged + lst1[index1:] + lst2[index2:]
```

Note: The remaining elements of the other list can all be added to the end of <merged>.
At most ONE of lst1[index1:] and lst2[index2:] is non-empty, but to keep the code simple, we include both.

Let's run bubbleSort with half the list.

```
def bubbleSort(L):
    for i in range(len(L) - 1):
        bubble(L, len(L))

def main():
    l = [76,454,12,87,41,1,77,98,34,6,23,86]
    half = len(l)//2
    l1 = l[:half]
    l2 = l[half:]
    bubbleSort(l1)
    bubbleSort(l2)
    lM = merge(l1,l2)
    print(l)
    print(lM)
main()
```

We divide the list into two, run sort, then merge.

[76, 454, 12, 87, 41, 1, 77, 98, 34, 6, 23, 86]
[76, 454, 12, 87, 41, 1] - [77, 98, 34, 6, 23, 86]
[1, 6, 12, 23, 34, 41, 76, 77, 86, 87, 98, 454]

What is we repeat this...and divide the list again.

[76, 454, 12, 87, 41, 1, 77, 98, 34, 6, 23, 86]
[76, 454, 12, 87, 41, 1] - [77, 98, 34, 6, 23, 86]
[76, 454, 12] - [87, 41, 1] - [77, 98, 34] - [6, 23, 86]
[12, 76, 454] - [1, 41, 87] - [34, 77, 98] - [6, 23, 86]
[1, 12, 41, 76, 87, 454] - [6, 23, 34, 77, 86, 98]
[1, 6, 12, 23, 34, 41, 76, 77, 86, 87, 98, 454]

What is we repeat this again....and divide it up further....what does this look like?

Mergesort

Basic idea:

- split list up into two halves, and then sort each half
- then assume two sorted lists, and merge them into a single sorted list.

Wiki Mergesort Visual [https://en.wikipedia.org/wiki/Merge_sort]:

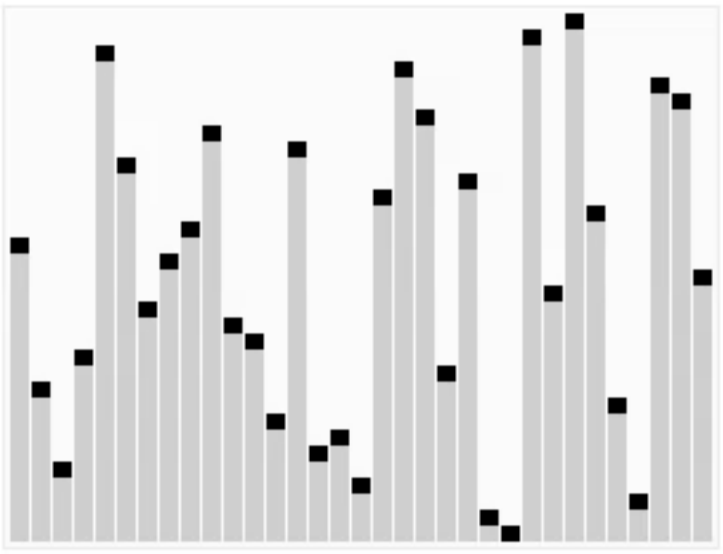

```
def mergesort(lst: list) -> list:
    # Return a sorted list with the same elements as <lst>.
    # This is a *non-mutating* version.
    if len(lst) < 2:
        return lst[:]
    else:
        # Divide the list into two parts, and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two sorted halves. Need a helper here!
        return merge(left_sorted, right_sorted)
```

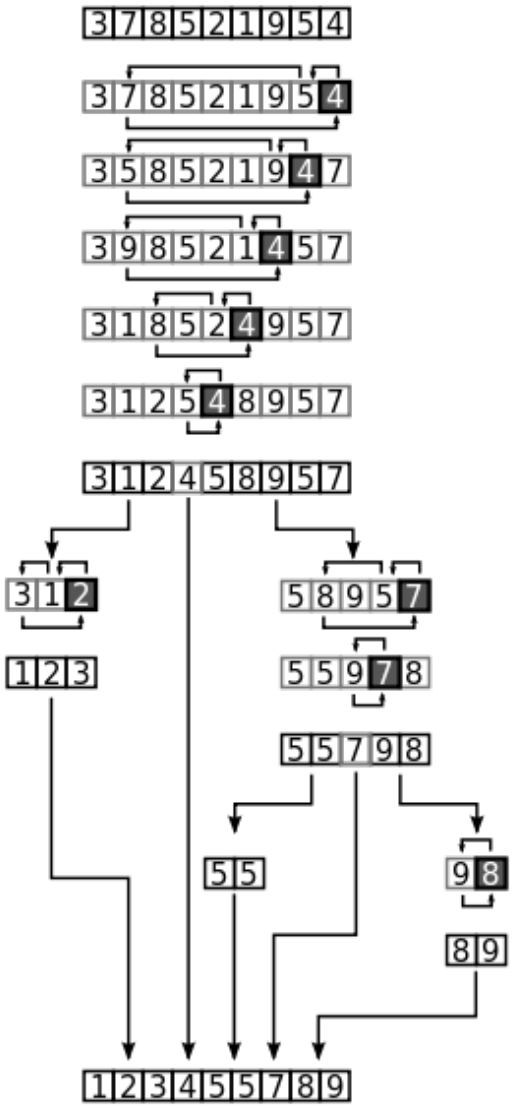
Quicksort

Basic idea:

- pick some arbitrary element in the list and call it the pivot,
- split up the list into two parts:
  - the elements less than (or equal to) the pivot, and those greater than the pivot
  - sort each part separately, and then combine (easy).

Wiki Quick Sort Visual:






```
def quicksort(lst: list) -> list:
    Return a sorted list with the same elements as <lst>.
    This is a *non-mutating* version of quicksort.
    """
    if len(lst) < 2:
        return lst[:]
    else:
        # Pick pivot to be first element.
        # Could make lots of other choices here
          #    (e.g., last, random)
        pivot = lst[0]

        # Partition rest of list into two halves
        smaller, bigger = partition(lst[1:], pivot)

        # Recurse on each partition
        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        # Return! Notice the simple combining step
        return smaller_sorted + [pivot] + bigger_sorted
```

Partition Helper Function:

Goal: Append elements to bigger and smaller lists using a loop.

```
def partition(lst: list, pivot: object) -> Tuple[list, list]:
    """Return a partition of <lst> with the chosen pivot.
    Return two lists, where the first contains the items in <lst>
     that are <= pivot,  and the second is the items
     in <lst> that are > pivot.
    """
    smaller = []
    bigger = []
    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

Efficiency of recursive sorting algorithms (***Optional Advanced Topic***)

For recursive code you need to analyze the running time of the non-recursive part of the code and you must also factor in the cost of each recursive call made.
Mergesort: recall the algorithm:

```
def mergesort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2 # This is the midpoint of lst
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        return merge(left_sorted, right_sorted)
```

Suppose we call mergesort on a list of length n.
- merge operation takes linear time, approximately n steps
- "divide" step itself also takes linear time, since the list slicing operations lst[:mid] and lst[:mid] each take roughly *n/2* steps to make a copy of the left and right halves of the list
- the fact that the "divide" and "combine" steps together take linear time means we can represent them together as taking roughly n steps.

So far we have ignored the cost of the two recursive calls.
- we split the list in half
    - each new list on which we make a has length *n/2*
    - each of those merge and slice operations will take approximately *n/2* steps, and
    - then these two recursive calls will make four recursive calls, each on a list of length *n/4*, etc.

We can represent the total cost as a big tree, where at the top level we write the cost of the merge operation for the original recursive call, at the second level are the two recursive calls on lists of size *n/2*, and so on until we reach the base case (lists of length 1).

Note that even though it looks like the tree shows the size of the lists at each recursive call, what it's actually showing is the running time of the non-recursive part of each recursive call, which just happens to be (approximately) equal to the size of the list!
The height of this tree is the **recursion depth**: the number of recursive calls that are made before the base case is reached.

Since for mergesort we start with a list of length *n* and divide the length by 2 until we reach a list of length 1, the recursion depth of mergesort is the number of times you need to divide *n* by 2 to get 1.

Put another way, it's the number k such that $2^k \approx n$ (remember logarithms?).
This is precisely the definition: $k \approx \log n$, and so there are approximately *log n* levels (note: when we omit the base in computer science we assume the base is 2, not 10). Finally, notice that at each level, the total cost is n.

This makes the total cost of mergesort *O(n log n)*, which is much better than the quadratic n^2 runtime of insertion and selection sort when n gets very large!
You may have noticed that this analysis only depends on the size of the input list, and not the contents of the list; in other words, the same work and recursive calls will be made regardless of the order of the items in the original list.
The worst-case and best-case asymptotic running times for mergesort are the same: O(n log n).

The Perils of Quicksort

What about quicksort? Is it possible to do the same analysis? Not quite. The key difference is that with mergesort we know that we're splitting up the list into two equal halves (each of size n/2); this isn't necessarily the case with quicksort!

Suppose we get lucky, and at each recursive call we choose the pivot to be the median of the list, so that the two partitions both have size (roughly) n/2. Then problem solved, the analysis is the same as mergesort, and we get the (n log n) runtime. (Something to think about: what do we need to know about_partition for the analysis to be the same? Why is this true?)

But what if we're always extremely unlucky with the choice of pivot: say, we always choose the smallest element? Then the partitions are as uneven as possible, with one having no elements, and the other having size n – 1.

Here, the recursion depth is n (the size decreases by 1 at each recursive call), so adding the cost of each level gives us the expression

(n – – – 1)+[n + (n – 1)+(n – 2)+...+1]=(n – – – 1)+n(n + 1)/2,

making the runtime be quadratic.

This means that for quicksort, the choice of pivot is extremely important, because if we repeatedly choose bad pivots, the runtime gets much worse! Its best-case running time is O(log n), while its worst-case running time is $O(n^2)$.

Note about practicality

You might wonder if quicksort is truly used in practice, if its worst-case performance is so much worse than mergesort's. Keep in mind that we've swept a lot of details under the rug by saying that both_merge and)partition take n steps; in practice, the non-recursive parts of quicksort can be significantly faster than the non-recursive parts of mergesort. This means that for an "average" input, quicksort actually does better than mergesort, even though its worst-case performance is so poor!

In fact, with some more background in probability theory, we can even talk about the performance of quicksort on a random list of length n; it turns out that the average performance is essentially (n log n), indicating that the actual "bad" inputs for quicksort are quite rare.