Iteration and Recursion

- The theoretically equivalent (but notationally different) concepts of *iteration* and *recursion* provide a crucial algorithmic ingredient — without them, we would be unable to define algorithms that can vary in size based on the algorithm's input
  - Try writing an algorithm to calculate the mean of *any* size list of numbers, without iteration nor recursion (well, try it later when we've gone over both of them)
- Note how these concepts are *independent* of any particular programming language; though every language has its own way of doing them, the underlying ideas remain the same

Searches and Sorts

- As in the textbook, we will introduce these concepts by example, using some classic algorithms that we use everyday — searches and sorts
- Presumably we all have an intuitive notion of what searches and sorts are; more formally, searches and sorts are viewed as algorithms performed over *ordered collections* of data
  - A search locates or identifies the item(s) in an ordered collection that match certain criteria (e.g., value equals something, values less than or greater than something, etc.)
  - A sort modifies the order within the collection so that the items within that collection adhere to some kind of sequencing rule (e.g., alphabetical, numerical, time-based, etc.)

Sequential Search

- Let’s take the simplest type of search algorithm: given some value and an ordered collection of values, determine whether that value is in that collection
  - Is Harry Potter in the student list?
  - Is *Last Year at Marienbad* a movie in IMDB?
  - Is there an F-19 stealth fighter for sale on eBay?
- A *sequential* approach for performing this search, in plain English, would say something like:

Go through each item in the collection. If that item is the value that we are looking for, then yes, the value is in the collection. If we go through the entire collection without finding the item, then no, the value isn’t in the collection.

- Let’s take this to pseudocode: notice the key phrase “each item in the collection” — this typically implies that some kind of *iteration* is necessary
  - We need to *name* the objects that we’re dealing with
  - We need some kind of *condition* to determine whether we’ve found the value that we want — in this case, it is simply some notion of *equality*
- Here’s a version of sequential search — intentionally different from the text, to show how algorithms can accomplish the same thing in different ways:

```
procedure search(list, target)
    while (there are more items to consider in list)
       set test to be the next item in list
       if (test = target) then
          target is in list; we’re done with the algorithm
    (If we get here, then we never found the item)
    target is not in list; we’re done with the algorithm
```

A Variation on the Theme

- Note that this version of sequential search makes no assumptions as to the initial ordering of the items in the list — this means we can't declare that the target value isn't in the list until we've looked at *every* item
- *Sorting* the list first, so that it follows a known sequence which we can use to tell whether we're "past" the item that we're seeking, helps us bail out of the loop a little earlier
  - ◇ And this is a good thing because…? Can you state why in a quantitative manner?
  - ◇ So how do we sort the list? Hold that thought…

Inside the Sequential Search Loop

- Look at what would *actually* happen in the algorithm's *loop* or *iteration* section for a specific *list* and *target*, say *list* = [9, 2, 4, 8, 1] with *target* = 4:
  - ◇ There are items to consider: [9, 2, 4, 8, 1]
  - ◇ Set *test* to the next item, which is 9
  - ◇ *test* ≠ *target*, so we aren't done yet
  - ◇ There are items to consider: [2, 4, 8, 1]
  - ◇ Set *test* to the next item, which is 2
  - ◇ *test* ≠ *target*, so we aren't done yet
  - ◇ There are items to consider: [4, 8, 1]
  - ◇ Set *test* to the next item, which is 4
  - ◇ *test* = *target*, so we are done with the algorithm

Technically, this is called unrolling *the loop*

- The loop's *body* is mentioned just once in the algorithm, but when *executed*, it is done multiple times

Loop Control in General

- The sample loop in our sequential search algorithm exhibits the same general structure of all loops:
  - *Initialize* — Set up an initial state (e.g., establish the collection to loop over, set up a counter variable, etc.)
  - *Test* — Check to see if a termination condition holds; this is usually related to whatever is set up in the initialization component
  - *Modify* — Change something about the state of the program; this change should take us closer to the termination condition

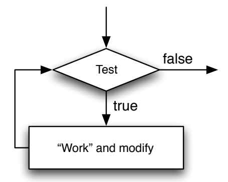
- Note how these three components have been present in the loops that we've seen so far, though of course in different variations

	Initialize	Test	Modify
Batch Operating System	Set up the list of jobs to perform; set the job number (index) to zero	Has the job index reached the last job in the list?	Add one to the job index
Mean of *n* Numbers	Set up the list of numbers to add	Have we added all of the numbers?	Move to the next number to add to the total
Best-of-Three Rock-Scissors-Paper	Set the scores of both players to zero	Has one of the players' scores reached 2?	If a player wins a "hand," add one to their score

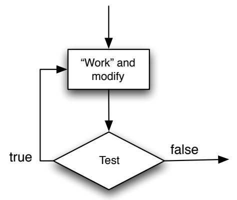
Loop Variations

- An implicit *fourth* component of all loops is to "do some work" — what should we actually *do* for each iteration of the loop?
  - In sequential search, we check to see if we've found what we're looking for
  - In a batch operating system, we perform the job
  - In the mean of *n* numbers, we add to the total
  - And so on…
- Loops may vary in terms of the *order* that these components take place; in particular, the *test*, *modify*, and "work" components may be done at different points in the algorithm

- A *pretest* or *while* loop tests the termination condition *before* possibly doing any work or modification



- A *posttest* or *repeat* loop (sometimes also called a *do-while* loop) tests the termination condition *after* doing at least one iteration of work and modification



- Both types of loops may also test for a termination condition *inside* the work/modification/activity section — in this case, the loop also has one or more *midtests*
  - Our sequential search algorithm has a midtest — the moment we find what we're looking for, we exit the loop, even if we haven't gone through all of the items in the list yet
- Other variations exist: sometimes, we leave the loop when a condition becomes *true* rather than false (this is a *repeat-until* or *do-until* loop)

Insertion Sort

- Now back to that issue of sorting…as you've probably realized, this is an algorithm in itself
- As with searches, we establish some specific initial conditions, since the final goal of these exercises is to get a computer to perform them:
    - A sort modifies the order of items in a collection — note how an *ordered* collection may not necessarily be *sorted* the way we wish
    - We want to perform the sort *in place* — that is, we want to use as little "extra space" as possible (reflective of how computers work with main memory)
    - To help quantify *which* item in the list we are talking about (without having to know the specific *value* of that item), we establish a numeric *index* corresponding to where in the list that item appears (i.e., the first item has index = zero, the second has index = one, etc.)

- An *insertion* approach to sorting may be phrased in this manner (recall that getting here — i.e., getting a "feel" for the algorithm — remains a bit of an "art")

Think of the collection as having two parts: the first part is already sorted, while the remainder isn't. If we start with the "first part" initially containing just the first item, then that one-item list is already sorted. We then take the second item, "take it out" of the list, and insert it back either before or after the first item, depending on whether or not that second item is greater than the first. The new "first part" of our list now consists of two items, and that is now sorted. Go down the list, removing then reinserting each item in the correct place within the sorted part of the list, until we have processed the entire list.

- Note how this algorithm works by taking each item in the list — there's that key word "each" again — and *inserting* it at a better location; this is essentially the "work" that we will perform in our loop
    - The initialize phase establishes the list to sort, and sets the current item — technically called the *pivot* — to the first item in that list
    - The test phase checks whether we've processed every item in the list, while the modification phase sets the pivot item to the next item in the list

Insertion Sort Pseudocode

In this version, we "calibrate" our index so that the first item in the list has an index of zero — note how this is fine as long as we are consistent in how we use the index

```
procedure sort(list)
   pivotIndex := 1 (the second item in the list)

   while (pivotIndex < the number of items in list)
        pivot := the item at pivotIndex
        move pivot out, leaving a "hole" at pivotIndex

        while (there is an item above the hole and that item is
        greater than pivot)
           move the item into the hole, shifting the hole "up"

        place pivot in the hole
        pivotIndex := pivotIndex + 1
```

Looping the Loop

- Note an interesting property of this algorithm — there are *two* loops, one inside another:
  - The loop that moves through each item as the pivot
  - The loop that shifts the "sorted part" of the list until the correct location is found for the pivot
- Loops within loops are not uncommon in algorithms — the "inner" loop is called a *nested* loop
- Can you write out the actual work performed by the algorithm with this nested loop is *unrolled*?

Binary Search

- You may have noticed that in real life, we don't always search for things in a sequential manner — for example, we don't search a dictionary, phone book, or textbook index this way
- Think for a moment…what's the substantive difference between sequential search and the way we search for a word in a dictionary (or an entry in a phone book)?

(yes, you are doing algorithm discovery now)

- Finished? Here's the trick — note that dictionary, phone book, and textbook index searches take advantage of the fact that these lists are *sorted*
- This allows us to "jump" directly to an item that's "close" to what we're looking for, simply because their order gives us an idea of where to look, instead of having to start from the beginning
- Depending on where our "jump" goes, we either jump backward or forward, either until we find the item that we want or we conclude that the item isn't there
    - For example, let's say we're looking for "recursion" in a dictionary — first, we open the dictionary to somewhere past its middle, because we know that the "r" words are in the latter part of the dictionary
    - If we see words beginning with "t," then we know that we overshot, so we jump backwards; this time, we might see words beginning with "p," so we jump forward again, but by a little bit less than before…we may then see "ro" words, so we jump back, etc.

When We Say "Jump," the Computer Asks "How High?"

- We use a certain intuition when skipping around a dictionary or phone book…and unfortunately, intuition is something that machines don't have (yet)
- So we need a more precise way of specifying how far to jump when we search — something that we can eventually state in precise pseudocode
- Since we want our algorithm to be completely general-purpose, we can't make any assumptions about what our sorted collection contains — thus, instructions like "go to the S section" will not do

- So here's an idea — why don't we just jump to the *middle* of the list? If there is an even number of items in the collection, then we jump to the first item of the list's second or latter half
- If this "middle" item is greater than the item that we're looking for, then we know that the item that we want is in the *first* half of the list; otherwise, the item must be in the second half
- Now, we can jump to the middle of our "chosen" half, and the process repeats again — ultimately, we will come down to a "half" consisting of *just one item*, and this item is either the one we're looking for, or it isn't, meaning that the item we want is *not* in the overall list
- Seems like we're ready for some pseudocode now…

Binary Search Pseudocode

Since this algorithm is perhaps the most sophisticated one that we have seen so far, let's build it slowly; here's a first cut, almost like a top-level outline:

```
procedure search(list, target)
  if list is empty then
    target is not in list; we're done with the algorithm
  else
    test := "middle" item in list
    if test = target then
      target is in list; we're done with the algorithm
    else if test < target then
      search the "bottom" half of list for target
    else if test > target then
      search the "top" half of list for target
```

- Note how the procedure's heading is *identical* to that of sequential search, emphasizing that both algorithms accomplish *the same final result* with *the same initial information* — they just use different approaches
- Stare at the last two cases a bit, where either *test* < *target* or *test* > *target*: observe that the action to take in these situations is *also* to search a list for *target*, with the sole difference being that we'll use a *different* list
- In a sense, the *search* procedure ***needs to invoke itself*** to do its work — this is called *recursion*
- A nice visual for this is the idea of mutually reflecting mirrors ⟶

Recursion Pseudocode

The refined pseudocode below emphasizes the recursive nature of the algorithm by showing that the *search* procedure invokes itself:

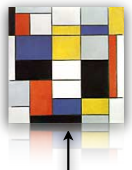
```
procedure search(list, target)
  if list is empty then
    target is not in list; we're done with the algorithm
  else
    test := "middle" item in list
    if test = target then
      target is in list; we're done with the algorithm
    else if test < target then
      search("bottom" half of list, target)
    else if test > target then
      search("top" half of list, target)
```

"Drilling Down"

- The approach we have taken, which is to start with an "outline" of the algorithm then gradually work out the details, is known as a *top-down approach* to solving the problem (which, in this case, is to specify the binary search algorithm in pseudocode)
- Observe how the pseudocode above doesn't quite say everything yet — there are three sub-algorithms that must be filled out…which we will leave as an exercise:
  - How does one extract the "middle" item of a list
  - How does one extract the "bottom half" of a list
  - How does one extract the "top half" of a list

Recursive Control

- As with loops/iteration, proper recursive expression of an algorithm requires certain common components:
  - *Initial call* — Analogous to loop *initialization*, this sets up the initial information required by the algorithm
  - *Base or degenerative case* — Analogous to the loop *test* component, this is the condition that indicates when further recursion is no longer necessary
  - *Recursive call* — Combines *modification* and the work to be done by invoking the same algorithm with slightly different (typically reduced) input



- Binary search is the first recursive algorithm that we have seen; these other algorithms also lend themselves to a recursive approach:
  - Creating a Mondrian-style painting
  - Counting the number of leaves in a tree (think of a tree as having branches, which themselves have branches, and so on until we reach the leaves)
  - Finding the prime factorization of a number (e.g., 60 has prime factors 2 x 2 x 3 x 5)
- While recursion *looks* like a set of infinite reflections, the reality is that it isn't infinite — it *should* terminate, since after all it is used in an algorithm…the general trick for this is to ensure that the gradually reducing recursive calls eventually lead to the base case

Quicksort: Recursive Sort

- Sorts can be recursive, too!
- Think about this approach to a sort:

  Given an ordered collection of items, pick an item somewhere on that list. Then, rearrange the list so that everything *before* that item is *less than* that item, while everything *after* that item is *greater than* that item. Take this “lesser half” of the list and do the same thing to it (i.e., pick an item, then place everything less before it, with everything greater after it), then take the “greater half” and do the same thing as well. Keep on doing this until reducing the list any further results in an empty collection.

- Quite a mouthful, so it might need to sink in…here’s an example, with simple numbers:
  - Start with [9, 2, 4, 7, 1] — pick 4
  - Rearranging this list around 4 results in [2, 1, 4, 7, 9]
  - We then do the same thing to [2, 1] and [7, 9], resulting in [1, 2] and [7, 9], respectively
  - Putting the results together, we get [1, 2, 4, 7, 9] — we’re done

- The helper algorithm, which places all values lesser than the chosen item before it in the list, with all greater values after it, is called a *partition* operation — note that the partitioning doesn’t *sort* the values before and after the chosen item; it just makes sure that everything prior is less, and everything after is greater
- Assuming we have a *partition* operation, quicksort pseudocode would look like this:

```
procedure sort(list)

 if list is empty then
    we’re done; list is sorted
 else
    select an item in list
    partition the list according to item
    sort(part of list before item) ←         Recursion, divide
    sort(part of list after item) ←            and conquer!
```