Chapter 3, Algorithms

John M. Morrison

January 6, 2020

Contents

0 Introduction

Recall that an algorithm is a precise recipe for carrying out a task. We have a wide variety of tools at our disposal for accomplishing computational tasks in Python. We have looping, conditional logic and Python's smart objects at our disposal that can easily automate complex procedures. The purpose of this chapter is to allow us to flex these new muscles.

1 Searching

Consider the task of searching a list for a particular item. If the list is unsorted and contains n elements, we will just have to walk through the list, checking if each element is the desired element we seek (the quarry). At most we will have to do n checks. If the item is not present, we end up checking the whole list to no avail.

We see that this procedure takes an amount of time at worst proportional to the size of the list, so it is an $O(n)$ algorithm.

Programming Exercise Use a loop to write `contains(c, quarry)`, where `c` is a list or tuple and `quarry` is an object. Have it return `True` if `quarry` is present in `c` and `False` otherwise.

1.1 Binary Search

Later in this chapter, we will discuss sorting, but here will discuss a smart scheme for searching a sorted list. This method will be quicker than the linear search you just wrote because you will not need to inspect every element in the list.

Suppose we have a list that is in sorted order. Here is a scheme for searching it.

1. Look at the item halfway into the list.
2. if this item is your quarry, report `True` and you are done. If this item is is before your quarry, discard the first half of the list; the quarry, if present, is in the second half. If the item is after the quarry, the item is in the first half of the list.
3. Repeat. If you end up discarding the entire list and never find the quarry, return `False`

This method is called *binary search*.

Suppose your list contained a million elements. After each pruning of the list, we have the following largest size of the remaining list.

```
1 1000000
2 500000
3 250000
4 125000
5 62500
6 31250
```

```
 7 15625
 8 7813
 9 3907
10 1954
11 977
12 489
13 245
14 123
15 62
16 31
17 16
18 8
19 4
20 2
21 1
```

This procedure requires a maximum of 20 steps. Each time, the search field is cut in half. So, all we are really asking is: How many times do we need to divide a number by 2 until it is reduced to 1 or 0? This would be $\log_2(n)$. So the time for this procedure to do its job is $O(log(n))$. You will notice an absence of a base in the logarithm; this is because all logarithms are proportional to each other. The reason? Apply the change-of-base formula for logs.

We all know from Ren and Stimpy that logs are big, heavy and wood. They are also very slow growing functions. So sorting a list makes finding things in it worlds faster. You can search a sorted list with a billion elements in 30 checks. That's a whole lot better than a billion checks.

Programming Exercise Implement this method. Load the Scrabble dictionary into a list using `readlines`. Then write a procedure to search it for a word, returning `True` if the word is present and `False` otherwise.

2 Root Finding

We are going to describe an algorithm for finding a zero (x–intercept) of a continuous function that changes sign on an interval. This uses a "divide and conquer" mechanism similar to that of the binary search algorithm you just wrote. We need a one useful fact, the Weierstraß Intermediate value theorem.

Theorem. Let $[a, b]$ be a closed, bounded interval and f be a continuous real-valued function defined on this interval. If f changes sign on the interval, then there is at least one $x \in [a, b]$ so that $f(x) = 0$.

Figure 1: Fast-Moving Banana Slug

There is an easy way to check if two numbers x and y are of opposite sign; just check to see if $xy < 0$.

Let us begin with an example. Suppose we are trying to approximate $\sqrt{2}$. We know that the function $f(x) = x^2 - 2$ has $\sqrt{2}$ as a root. Also we can see that $f(1) = -1$ and $f(2) = 2$, so f must have a root on $[1, 2]$.

We know that $\sqrt{2} = 1.5 \pm .5$. Now we do this. Check the midpoint; $f(1.5) = .25$ and $f(1) = -1$ so we have now bounded our root to being in $[1, 1.5]$. This means $\sqrt{2} = 1.25 \pm .25$. At each step we see that we can obtain a value that is within a tolerance of $\sqrt{2}$. When that tolerance is small enough, we can stop.

Programing Exercise Continue this calculation until you know $\sqrt{2} \pm .001$. Can you automate the process with a Pyhon program?

Note that the error is at most `(right - left)/2` where [`left`, `right`] is the interval on which we know the function changes sign.

Programming Exercise Write a function `zero(a, b, tol, f)` that finds an approximation for a root of the function `f` on $[a, b]$ with error tolerance `tol`. The preconditon for calling this function is that f changes sign on $[a, b]$.

3 A little number theory

Let us begin by looking at some elementary ideas of number theory. Back in your Wormwood days, you learned about long division. You learned that if a and b are positive integers, you can write

$$b = aq + r,$$

where q is an integer called the *quotient*, r is the *remainder*, and $0 \leqslant r < a$. In fact, these integers q and r are unique. If the remainder r is zero, then a divides into b evenly. In this event, we write $a \mid b$.

Python computes these quantities easily. Let us take $b = 365$ and $a = 7$.

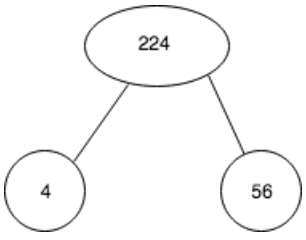
```
>>> b = 365
>>> a = 7
>>> q = b//a
>>> r = b%a
>>> a*q + r == b
True
```

So, the quotient is just obtained by integer division and the remainder is computed with our friend mod, `%`.

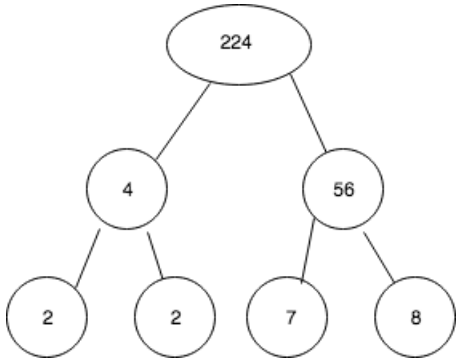
Testing for divisibility of numbers is easy. If you have integers `a` and `b`, then you can test for `a|b` with the predicate `b%a == 0`. For example, to test if a number `n` is even, you use the predicate `n%2 == 0`.

Back in your Wormwood days, you likely learned about prime numbers. The number 1 was defined not to be prime. An integer 2 or larger is prime if its only positive divisors are 1 and itself. For example, 4 is not prime, because $2 \mid 4$. More generally, a number is not prime if it can be factored into two smaller integers.

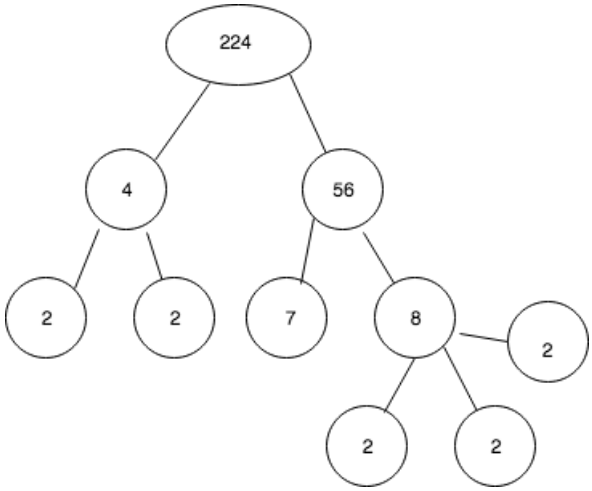
The *prime factoriaztion theorem* states that every number $n \geqslant 2$ can be factored uniquely into primes. Back in the day, Miss Wormwood taught about factor trees, which provide a nifty method for doing this. Here is how they worked. Suppose we want to factor the number 224. It's a cinch that 224 is divisible by 4. We begin by drawing this.



Both of the leaves on the tree can be factored into smaller integers; do so. The wonderful machinery of number theory says that the result at the end is the same no matter how you do this.



Two of the leaves on this tree now have 2s in them; they are prime. Similarly, there is nothing to be done with the 7. We will "cheat" and factor the 8 into three 2s.



Every leaf on the tree is prime; it cannot be factored further. We simply pluck up the numbers inside of the leaves of the tree. We now have this prime factorization for 224.

$$224 = 2 \cdot 2 \cdot 2 \cdot 7 \cdot 2 \cdot 2 = 2^5 \cdot 7.$$

Now we will create a function that tests positive integers for primality.

To begin, observe that the number 2 is prime because it cannot be factored into two smaller integers. However, any even number n larger than 2 satisfies

$2|n$, so even numbers larger than 2 are not prime. So, we can begin writing our `isPrime` function as follows.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2
        return True
    if n%2 == 0:
        return False
    ##we are not done yet.
    return "cows"
```

Test this function out. It will return `False` if `n` is 1 or if it is an even number larger than 2. It will return `True` if `n` is 2. For all other cases, it will punt and return `"cows"`, a ridiculous value we provide to remind ourselves we are not done yet.

We could accomplish the rest as follows. We then take a couple of known primes for a spin.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    k = 3
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
print(isPrime(997))
print(isPrime(10000019))
```

Ugh. We're movin' kinda slow at the junction.

```
unix> time python wasteful.py
True
True

real        0m1.980s
user        0m1.508s
sys        0m0.073s
```

This has to do a whole lot of checks before it resolves; the cost of the procedure is roughly proportional to the number you ask it to check if you pass it a prime and less otherwise.

Now here is an interesting little jewel of a fact that will make our function sweetly efficient. Suppose that you have a positive integer n and you write $n = ab$ where a and b are integers. Then at least one of a or b must satisfy the condition $k * k \leqslant n$.

Let us show an example. Take the number 36 and write $36 = 9 * 4$. Notice that $4 * 4 = 14 \leqslant 36$. If you write $36 = 6 * 6$, then both factors have a square that is at most 36. As you shall see all we care is that at least one factor has a square that is at most n.

So let us improve this. For starters, because of our preliminary tests, we can begin with `k = 3`. Also, we can discard all even numbers, so each time the loop runs let us update with `k += 2`. This will keep us testing with only odd numbers. Et Voila! We have a 50% improvement.

You should take your existing implementation of `isPrime` and integrate this improvement.

But, like the Ginsu Knife Man says, "There is more!" A while back, we learned this: if for a positive integer n we have no k so that $k * k \leqslant n$ and $n\%k = 0$, then n is prime! One thing we know here is that the square root of a large number is a whole lot less than the original number. This becomes more acutely so as the original number gets bigger. Let us put this together. Here is what we had.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2
        return True
    if n%2 == 0:
        return False
    ##we are not done yet.
    return "cows"
```

Now we start at `k = 3` and go up by 2 each time. The cows get the boot, because if the loop ends without finding a prime, we know that `n` is prime.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
```

```
        return False
    k = 3
    while k < n:
        if n%k == 0:
            return False
        k += 2
    return True
```

Next, insert our other improvement.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    k = 3
    while k*k <= n:
        if n%k == 0:
            return False
        k += 2
    return True
```

Now let's run it.

```
unix> time python faster.py
True
True

real        0m0.073s
user        0m0.042s
sys        0m0.017s
```

Va voom!

Programming Exercises

1. Implement a function called `smallestFactor(n)` that does the following. It accepts a positive integer `n`. If `n` is 1 it just returns 1. Otherwise, it finds the first positive integer `k` so that `k|n`. Here are some handy test cases.

   ```
   smallestFactor(997) == 997
   smallestFactor(323) == 17
   ```

```
smallestFactor(1) == 1
smallestFactor(1728) == 2
smallestFactor(1005973) == 997
```

Try hard to minimize the number of times the loop runs. Borrow from the ideas usedin `isPrime`.

2. Why does `smallestFactor` return prime numbers if `n >= 2`?
3. Use `smallestFactor` to implement a function called `primeFactors(n)`, which returns a list of all prime factors of the positive integer `n` passed it. If `n == 1`, return an empty list. Can you use recursion to do this?
4. Run `primeFactors` on several different cases. Why is the list it returns always in numerical order?

4 The Performance of Algorithms

We just saw that there is more than one way to test a number for primality. Both ways worked, but, using a little knowledge of mathematics, we were able to achieve a huge boost in performance.

We are going to lay down a little mathematical vocabulary here that will help us to describe algorithmic efficiency. This language is a universal language among computer scientists.

Mathematically, a *sequence* is a function whose domain is the set of positive integers. If f and g are sequences, we will write

$$f(n) = O(g(n))$$

to mean that there is some constant M so that

$$|f(n)| \leqslant M|g(n)|.$$

To put this is plain English, f is at most proportional to g.

Let us look at a concrete example. The unimproved `isPrime` function used the predicate `k < n` in its `while` loop. This loop could run as many as `(n - 1)//2` times. So, the cost of executing this procedure is at most proportional to `n`, the size of the number we are testing for primality. We would say this algorithm is $O(n)$, or *linear-time*.

The improved algorithm used the predicate `k*k < n` in its `while` loop. It could not possibly run more than $\sqrt{\texttt{n}}$ times. This makes the improved algorithm a $O(\sqrt{n})$ algorithm.

You know that when n is large, $\sqrt{n}$ is far smaller than n. So this improved algorithm is much faster than its unimproved counterpart.

5 Sorting using Iterative Techniques

Python performs sorting on lists as a service. However, it is important to understand how sorting works. We will study several sorting algorithms, starting with the dreadful and moving to the fleet. What we show here is just a small sampling of what is out there.

Imagine you have a deck of cards with numbers on them. Here is a possible way to sort them; it is called the *Bozo sort*. It is named after the character Bozo, who was a clown in a show that was popular from the late 1940s into the 1960s.

1. Check to see if the deck is in order; if so, you are done.
2. If not, shuffle the deck and repeat Step 1.

Let us see that this can work. Create a program called sorts.py. We can shuffle a list using its `shuffle` method.

First, let us write a function to see if a list is in order; let us agree on ascending order. Just reverse the inequalities in the predicates for descending order. To do this, we just walk up to each pair of elements and check if they are in order. If a pair is not, a `False` is returned.

```
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
```

Now let us write code to test our result. This code will shuffle a "deck", which is a list of consecutive integers, then sort with the bozo method. We will stub the bozo method in. and put our test code in.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    pass  #do nothing
def main():
    pass
main()
```

Running this code will not shuffle the deck. Now we add the code to do that.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    while(not isInOrder(deck)):
        random.shuffle(deck)

def main():
    deck = list(range(9))
    random.shuffle(deck)
    bozo(deck)
    print(deck)
main()
```

Here we are shuffling a deck with 9 items in it. Now run it. Even for a handful of elements, this works dreadfully. Remember, $9! = 362880$. This sort works in $O(n!)$ time, which is terrible.

```
Tue Feb 07:13:28:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m4.711s
user    0m4.344s
sys 0m0.080s
Tue Feb 07:13:28:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m7.150s
user    0m6.993s
sys 0m0.060s
Tue Feb 07:13:29:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m5.379s
user    0m5.185s
sys 0m0.055s
```

We can do better.

Let us look at a sort that works in a more reasonable amount of time, the bubble sort. Suppose we have a list, and we walk up to each pair of elements in succession. If the elements are in order, do nothing. If not, switch them. After one pass, you will see that the largest element will have "bubbled" to the top.

Now we can repeat the procedure and skip checking the last element. Then the two largest elements will be at the top. We continue until the list is sorted.

Let us begin to write this procedure in a function named `bubble`. We will have two integer variables, `k` and `end`. The variable `end` will point at the index dividing the sorted and unsorted portions of the list. The variable `k` will do the walking up to the pairs. So here is how we get started.

```
def bubble(x):
    end = len(x)
    #more code
```

Each time the inner loop runs we have a `loop invariant`, which is a set of statements that is true after each execution of the loop. Our loop invariant is this: items to the right of `end` are the largest elements in sorted order and items to the left have no guaranteed order. Yep, it's a jungle out there.

Let us make a general pass of the loop.

```
def bubble(x):
    end = len(x)
    for k in range(end) - 1):
        if x[k] > x[k+1]:
            x[k],x[k+1] = x[k+1],x[k]
    #more code
```

As is, this does one pass of the loop. At the end of each pass, `end` may be reduced by 1; let's put that in.

```
def bubble(x):
    end = len(x)
    for k in range(end - 1):
        if x[k] > x[k+1]:
            x[k],x[k+1] = x[k+1],x[k]
    end -= 1
```

When are we done? How about when `end` gets down to 1; by the process of elimination, the first element must be in place. We will need to enclose our `for` loop in another loop.

```
def bubble(x):
    end = len(x)
    while end > 1:
        for k in range(end) - 1:
            if x[k] > x[k+1]:
                x[k],x[k+1] = x[k+1],x[k]
        end -= 1
```

Let's test this.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    while(not isInOrder(deck)):
        random.shuffle(deck)
def bubble(x):
    end = len(x)
    while end > 1:
        for k in range(end - 1):
            if x[k] > x[k+1]:
                x[k],x[k+1] = x[k+1],x[k]
        end -= 1
def main():
    deck = list(range(9))
    random.shuffle(deck)
    bubble(deck)
    print(deck)
main()
```

That was so fast on 9 elements it was basically instant. Let's try this on 10,000 elements.

```
$ time python sorts.py

real    0m13.159s
user    0m13.011s
sys 0m0.057s
```

Here is the result on 5000 elements. It is about 1/4 as long

```
Tue Feb 07:14:08:ppp> time python sorts.py

real    0m3.509s
user    0m3.305s
sys 0m0.039s
```

If you did this with the bozo sort, the sun would likely nova first and consume the Earth before your list would ever be sorted. Even the bubble sort seems a little slow.

You might ask, "What is the big-O classification for this sort?" Let us think about it. The first time through we perform $n-1$ checks. The second time through, we perform $n-2$ checks. Each time the number of checks diminishes by 1. The total number is

$$(n-1)+(n-1)+\cdots 1=\sum_{k=1}^{n-1} k.$$

Yes, there is a formula for this,

$$\sum_{k=1}^{n-1} k=\frac{n(n-1)}{2}=O(n^2).$$

The bubble sort is an example of a *quadratic sort*.

Now we will look at two other sorting methods, insertion sort and selection sort. Both of these are quadratic sorts, but the constant of proportionality is somewhat smaller. The insertion sort works in a manner similar to that of a player picking up cards and inserting them in his hand. As he picks up each card, he places it in the appropriate place in his hand until all of the cards are inserted.

It is best if we can do our sort in-place without using additional memory. So here is an idea. If the array is empty, do nothing and return immediately. Otherwise, make a variable called `end` and have it point at index 1 in the array. The loop invariant is this: items to the left of `end` are sorted there is no assertaion about items to the right. We now do the following. Add 1 to end. Then, keep swapping the new entry with its neighbor to the left until it is in the right place (a "trickle down") procedure. We keep doing this until `end` reaches the right-hand end of the array.

Let us begin here. If the list is empty, we bail. If not, we mark `end` at `1`. Notice here that everthing to the left of `end` is sorted, because there is only one entry over there.

```
def insertion(x):
    if x == []:
        return
    end = 1
```

Now let's block in some more code.

```
def insertion(x):
    if x == []:
        return
    end = 1
    while end < len(x) - 1:
```

```
        end += 1
        #then trickle down
```

Now let us explain carefully. Suppose we have this array

```
1  3  5  9 |4  0
          (end)
```

We increment end.

```
1  3  5  9  4 |0
            (end)
```

We will keep swapping with the neighbor to the left until 4 is in the right place.

```
1  3  5  9  4 |0
1  3  5  4  9 |0
1  3  4  5  9 |0
            (end)
```

Let us do this again. Begin by incrementing `end`

```
1  3  5  9  4 |0
1  3  4  5  9  0 |
               (end)
```

Now do the swaps. WARNING: We do not want to attempt to check the left-hand neighbor of 0 when it gets to the bottom! If we do so we will be comparing with the item at the end of the list. This will make a horrid mess. We will use short circuiting to prevent this.

```
def insertion(x):
    if x == []:
        return
    end = 1
    while end < len(x) - 1:
        end += 1
        k = end
        #trickle down; notice k >= 1 condition
        while  k >= 1 and x[k] <  x[k - 1]:
            x[k], x[k - 1] = x[k - 1], x[k]
            k -= 1
```

You can put this in `sorts.py` and test it. This method is quite a bit faster than the bubble sort. Typically, unless you are very unlucky, the trickle down procedure only trickles about half-way down on average. This means it is executing far fewer instructions than the bubble sort.

Now we will implement the insertion sort. If the list is empty do nothing. Otherwise we set `end = 0`.

```
def selection(x):
    if x == []:
        return
    end = 0
```

Now we scan the list beyond `end` for the smallest element and swap it with `x[end]`. After this we update `end` by incrementing it.

```
def selection(x):
    if x == []:
        return
    end = 0
    while end < len(x):
        j = end
        for k in range(1 + end, len(x)):
            if x[j] > x[k]:
                j = k
        x[j],x[end] = x[end], x[j]
        end += 1
```

All of the sorting algorithms we used here are quadratic sorts, i.e. they operate in $O(n^2)$ time. Next, we will use recursion to perform sorts and we will realize a significant increase in performance for sorting large lists.

6 A Recursive Sort: Mergesort

Imagine you want to sort a humongous list. You might do this. Break the list in two. Use a quadratic sort on each half. Each half will take 1/4 as long as doing a quadratic sort on the entire list. Then, once you have the two sorted lists, zipper them back together; that procedure is $O(n)$ as we shall soon see. We realize a significant performance boost.

Any handle worth cranking once is probably worth cranking a whole lot. What if we take each of the two smaller lists, cut them in half and . . . ?

That is the idea behind the *merge sort*. To make the merge sort work, we must first write a procedure to zipper two sorted lists together.

So suppose we have sorted lists `x` and `y`. Begin by making variables `j` and `k` and setting them both to be 0. We then make a new list. In each iteration we put the smaller of `x[j]` and `x[k]` on the new list, then iterate the index we used. So some code looking like this might be in order.

```
def zipper(x,y):
    j = 0
    k = 0
    out = []
    while j < len(x) and k < len(y):
        if x[j] < y[k]:
            out.append(x[j])
            j += 1
        else:
            out.append(y[k])
            k += 1
```

Once we get here, we will have arrived at the end of one or both of the two lists. Just append their tails and return our result as follows.

```
def zipper(x,y):
    j = 0
    k = 0
    out = []
    while j < len(x) and k < len(y):
        if x[j] < y[k]:
            out.append(x[j])
            j += 1
        else:
            out.append(y[k])
            k += 1
    out += x[j:]
    out += x[k:]
    return out
```

Now let us write the actual sort. Lists of length 0 or 1 are fully sorted. So we can no proceed as follows.

```
def merge(x):
    if len(x) <= 1:
        return x
```

Now comes the recursive step. Cut the list in two and call the `merge` function on each half and zipper them up!

```
def merge(x):
    if len(x) <= 1:
        return x
    mid = len(x)//2
    first = x[:mid]
    second = x[mid:]
    return zipper(merge(first), merge(second))
```

The disadvantage to this method that is is not an in-place method. You have to create an additional copy of the list. However, you will see that is now feasible to sort a 1,000,000 integer list on your PC.

Programming Exercises In this exercise, you will learn about the *Gnome Sort*, which was named by its inventor Hamid Sarbazi-Azad as the *Stupid Sort*. What is interesting is that our quadratic sorts have an inner and outer loop. This has a single loop. Here is the idea.

1. If the list has fewer than two items, you are done. Otherwise proceed as follows.
   (a) Start at the beginning of the list.
   (b) Move right one.
   (c) Check your value and the value to the left. If they are in order, move right one.
   (d) If they are out of order, swap them and move one to the left. If you are at the beginning, move right one.
   (e) Once you skid off the right hand end of the list, you are done.
2. Shuffle the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` and run this procedure on it, printing it after each iteration. What do you see?
3. Learn about Timsort; this is the sorting method actually used by Python and Java. It is a variant on mergesort that seeks out runs in the data and zips them together.

7 Terminlogy Roundup

- **binary search** This is a divide and conquer technique by which we eliminate half of the search field in each iteration.
- **Bozo sort** This is sorting technique in which elements are shuffled and checked for proper order repeatedly until they are in the proper order.
- **Gnome aka Stupid Sort** This is a sort due to Hamid Sarbazi-Azad that looks like a modified insertion sort
- **linear-time** This describes an algorithm whose computational cost is proportional to the size of the problem.

- **merge sort** This is a recursive sort that keeps splitting the list into pieces until they are of size 1 or zero, and which zips the results together.
- **quotient** This is the result of dividing two numbers.
- **remainder** This is the remainder left by integer division.
- **prime factoriaztion theorem**
- **quadratic sort** This describes an algorithm whose computational cost is proportional to the size of the square of the problem.
- **sequence** This is a data structure that contains a collection indexed by the nonnegative integers.