Coding Interviews

Muchang Bahng

Spring 2024

Contents

1 Nuances of Python

1.1 Pass by Reference vs By Value

There are a lot of parallel characteristics between python variable assignment and C++ pointers. When we assign a variable to an object in python, what we are doing under the hood is creating the object in the heap memory (hence we use `malloc` rather than initializing on the stack) and initializing a pointer to point to that place in memory.

```
# Python
x = 4
print(x) # 4
.
.
```

```
# C
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
printf("%d\n", **x);
```

So far so good. But what if we wanted to reassign `x` to another variable? What it does in the backend is first create a new object in memory and reassign that pointer to the new object. If no other variables points to the original object, then the memory is automatically freed.

```
# Python
x = 4
print(x) # 4
x = 5
print(x) # 5
.
.
.
.
.
.
```

```
# C
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
printf("%d\n", **x);

int *xx_ = malloc(sizeof(int));
*xx_ = 5;
x = &xx_;
free(x_);
printf("%d\n", **x);
```

This is essentially how references work in Python. With this, it is clear why when you reassign a variable to a new object, it does not affect what the other objects are pointing to. There are two ways a programmer can interpret the following iconic example.

```
x = 4
y = x
print(x, y) # obviously prints 4, 4
y = 5
print(x, y) # what about this?
```

1. *Passing By Reference (Left).* The first interpretation is that by setting `y = 5`, we are modifying the value that `y` points to be `5`. Since the pointer `x` also points to the same memory address pointed by `y`, then `x` also should equal `5`.
2. *Passing By Value (Right).* By setting `y = 5`, we create a new `int` object, reassign the pointer `y` to the new object. Therefore `x` still points to `4` and `y` now points to `5`.

```c
// Pass by Reference
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
int** y = &x_;
printf("%d, %d\n", **x, **y); // 4, 4

**y = 5;
printf("%d, %d\n", **x, **y); // 5, 5
.
.
```

```c
// Pass by Value
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
int** y = &x_;
printf("%d, %d\n", **x, **y); // 4, 4

int *y_ = malloc(sizeof(int));
*y_ = 5;
y = &y_;
printf("%d, %d\n", **x, **y); // 4, 5
```

For primitive types (char, string, int, float), Python, along with almost every other major language, passes by value. For nonprimitive types, they are passed by reference. It's a little odd since we call these variables references, even though in C++ references cannot be reassigned. Either way, it is extremely important to know whether a variable is copied by reference or by value, since you'll be able to predict the behavior on one variable if you modify the other one.

To debug and see the memory address that it is pointing to, we can use the `id()` method.

```python
# Pass by value
x = 4
y = x
# Points to same address
print(id(x)) # 4382741696
print(id(y)) # 4382741696
x += 1
# Now it doesn't
print(x)    # 5
print(y)    # 4
```

```python
# Pass by reference
x = []
y = x
# Points to same address
print(id(x)) # 4383459648
print(id(y)) # 4383459648
x.append(1)
# Still points to same address
print(x)     # [1]
print(y)     # [1]
```

Example 1.1 (Common Traps)

To initialize a list of zeros, we can just do

```python
>>> x = [0] * 5
>>> x[0] = 1
>>> x
[1, 0, 0, 0, 0]
```

This is all good since primitive types are passed by value, so modifying one will not affect the others. However, if we are initializing a list of lists, then we get something different.

```python
>>> x = [[]] * 5
>>> print(x)
[[], [], [], [], []]
>>> x[0].append(1)
>>> x
[[1], [1], [1], [1], [1]]
```

This is because the inner list is multiplied and therefore copied *by reference*. This means that all the lists are simply pointing to the same object in memory, and modifying one modifies all.

1.2 Object Caching

The `is` operator returns `True` if two variables point to the same memory address. The `id()` operator is a unique integer representing the memory location of the value. In general, if we initialize two variables to be the same value, they do not point to the same memory address.

```python
# Example of when two variables are
# initialized to be the same value, but
# do not point to the same memory
x = 1000
y = 1000
print(id(x)) # 4385025360
print(id(y)) # 4385026288
.
.
.
```

```c
int* x_ = malloc(sizeof(int));
*x_ = 1000;
int** x = &x_;

int* y_ = malloc(sizeof(int));
*y_ = 1000;
int** y = &y_;

printf("%p\n", *x); 0x600001be8040
printf("%p\n", *y); 0x600001be8050
```

However, we can initialize `y` to be equal to `x`, which tells it to point to the same memory address as `x` is, thus having the same `id`.

```python
x = 1000
y = x
print(id(x)) # 4303203888
print(id(y)) # 4303203888
.
.
.
.
```

```c
int* x_ = malloc(sizeof(int));
*x_ = 1000;
int** x = &x_;

int** y = &x_;

printf("%p\n", *x); 0x600002368040
printf("%p\n", *y); 0x600002368040
```

Usually, just setting the values equal does not have it point to the same memory address, but for integers `[-5, 256]`, Python caches these numbers so that even if we initialize two numbers with the same integer value, they will always point to the same address.

```python
# Don't need to set y = x
x = 200
y = 200
print(id(x)) # 4314934592
print(id(y)) # 4314934592
```

This is a Python-specific fact that you should be aware of.

1.3 Mutable and Immutable Objects

All immutable types are copied by value?

The old trap `x = [[]] * 5, x[0].append(1)`. Primitive types are copied by value which is why the actual lists don't update. The elements are accessed, copied, and the copies are incremented

Primitive values are copied by value. Nonprimitives are copied by reference.

We talk about mutable and immutable objects, but what does that really mean in the backend? Integers are immutable, but we can always do `x = x + 1`. What immutable really means is that you cannot change the value that the pointer is pointing to without changing the actual memory location. When `x = 5` and you set `x = 4`, you have essentially reallocated (malloced again) another space of heap memory and had the variable point to that memory, so you have really "changed" the value by creating an entire new address, initializing it to what the changed value should be, and reassigning the variable pointer to the new address.

```python
# Python
x = 4
print(x) # 4
x = 5
print(x) # 5
.
.
.
.
.
.
```

```c
# C
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
printf("%d\n", **x);

int *xx_ = malloc(sizeof(int));
*xx_ = 5;
x = &xx_;
free(x_);
printf("%d\n", **x);
```

Let's try another example with a string now.

```python
x = "Hello "
print(id(x)) # 4382416384
print(x)     # Hello
x += "World"
print(id(x)) # 4382723056
print(x)     # Hello World
.
.
.
.
.
.
.
```

```c
char* x_ = malloc(sizeof(char) * 6);
strcpy(x_, "Hello ");
char** x = &x_;

printf("%p\n", *x); // 0x600002220040
printf("%s\n", *x); // Hello

char *xx_ = malloc(sizeof(char) * 11);
strcpy(xx_, "Hello World");
*x = xx_;

printf("%p\n", *x); // 0x600002220050
printf("%s\n", *x); // Hello World
```

If we work with mutable objects, like lists, then we do indeed have the same id even after modifying.

```python
x = [1, 2, 3, 4]
print(id(x)) # 4380543296
x.append(5)
print(id(x)) # 4380543296
del x[0]
print(id(x)) # 4380543296
```

```
```

1.4 Lists

Lists are implemented as an array of pointers, which can point to any object in memory which is why Python lists can be dynamically allocated. We should be familiar with the general operations we can do with a list, which are implemented as dunder methods.

Definition 1.1 (Length)

The `list.__len__()` method returns the length of a list, which is stored as metadata and is thus $O(1)$ retrieval time. It is invoked by `len(list) <-> list.__len__()`.

Definition 1.2 (Set Item, Get Item, Del Item)

The following three methods are getter, setter, and delete functions on the `list[T]` array given the index.

1. The `__getitem__(i) -> T` returns the value of the index of the list. Since we can do pointer arithmetic on the array, which is again just 8 byte pointers, we essentially have $O(1)$ retrieval

time. It is invoked by `list[i] <-> list.__getitem__(i)`.

2. The `__setitem__(i, val) -> None` returns `None` and sets the value of the index. It is invoked by `list[i] = val <-> list.__setitem__(i, val)`.
3. The `__delitem__(i) -> None` deletes the value at that index. It is invoked by `del list[i] <-> list.__delitem__(i)`.

The next few definitions are not dunder methods, but are important.

Definition 1.3 (Append, Insert, Pop)

`List.append(val)` is amortized $O(1)$ but is quite slow if we are inserting into the middle with `List.insert(i, val)`. `List.pop()` is great for removing from the back of the list, with $O(1)$, but not so great for removing from the front, where all the elements have to be shifted $O(n)$. Dynamically resizing the array, where all the elements of the previous array gets copied over to a larger array, is slightly different. For example, in an old implementation of Python, the new size is implemented to be `new_size + new_size » 3 + (new_size < 9 ? 3 : 6)`, which approximately doubles the size (like Java, which exactly doubles the list size), giving us amortized $O(1)$.

Definition 1.4 (Extend)

Definition 1.5 (Sort)

List slicing is quite slow since we are copying the references to every element in the list. Note that the values are not copied themselves, but we are creating an array of new pointers.

1.5 Iterators and Loops

For loops and while loops are straightforward enough, but it's important to know the difference between them.

1.5.1 Dynamic Evaluation of Condition During Loop

In while loops, the condition is rechecked and thus any functions called during this is recomputed at each loop, and so when deleting things from a list, the loop already accounts for the new length. However, a for loop evaluates the length of the list only once and leads to index violation errors.

```python
x = [1, 2, 3, 4]
print(x)
i = 0
while i < len(x):
    print(len(x))
    if x[i] == 2:
        del x[i]
    i += 1
print(x)

[1, 2, 3, 4]
4
4
3
[1, 3, 4]
```

```python
x = [1, 2, 3, 4]
print(x)

for i in range(len(x)):
    print(i, x[i])
    if x[i] == 2:
        del x[i]
print(x)

[1, 2, 3, 4]
0 1
1 2
2 4
IndexError: list index out of range
.
```

This can also be a problem when evaluating to a list where you may need to append more elements to it. Here we use the previous initial list. We want to append 5 and 6 since 2 and 4 are even, but the extra 6 added will require us to add 7 as well. In a for loop, this also breaks down. The for loop only accounts up to the length of the original list, which will end with 6 as the last element added. Whether you want the condition to by dynamically evaluated at every loop depends on the problem.

```
x = [1, 2, 3, 4]
print(x)

i = 0
while i < len(x):
    print(x[i])
    if x[i] % 2 == 0:
        x.append(max(x) + 1)
    i += 1

print(x)

[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6, 7]
```

```
x = [1, 2, 3, 4]
print(x)

for i in range(len(x)):
    if x[i] % 2 == 0:
        x.append(max(x) + 1)

print(x)

[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6]
.
.
.
```

1.5.2 Iterators and Enhanced For Loops

A list is an example of an *iterable* object. An `Iterable` class implements an `__iter__()` method that transforms it into an `Iterator` object. An `Iterator` objects allows one to generate some value every time a `__next__()` method is called. It should implement the next function and an `__iter__()` method also, which just returns itself. Here is an example for a list.

```
class Iterator:

  def __init__(self, input: list):
    self.index = 0
    self.input = input
    self.limit = len(input)

  def __iter__(self):
    return self

  def __next__(self):
    if self.index > self.limit:
      raise StopIteration
    self.index += 1
    return self.input[self.index]
```

So far, we have talked about looping through a list by looking at the indices. Another way is to to use an *enhanced for loop* to iterate directly over the values. When we use an enhanced for loop, we are really just creating an iterator object around the list and doing a while loop. Therefore, a for loop is really just a while loop!

```python
x = [1, 2, 3, 4]
for elem in x:
    print(elem)
.
.
.
.
.
```

```python
x = [1, 2, 3, 4]
x_ = iter(x)
while True:
  try:
    item = next(x_)
  except StopIteration:
    break
  print(item)
```

This means that every for loop is really just a while loop. For loops were created early on in programming for convenience. Even when doing for loops over indexes, the `range` is really an iterable, and so you can convert it into an iterator and do the same thing.

Another fact about `range` is that it is *lazy*, meaning that to save memory, calling `range(100)` does not generate a list of 100 elements. The iterator really evaluates the next number on demand, which adds runtime overhead but saves memory.

Example 1.2 (Common Trap)

Look at the following code

```python
>>> x = [1, 2, 3, 4]
>>> for elem in x:
...     elem += 1
...
>>> x
[1, 2, 3, 4]
```

This is clearly not our intended behavior. This is because in the backend, the `elem` is really being returned by calling `next()` on the iterator object. The type being returned is an `int`, a primitive type, and therefore it is passed *by value*. Even though `elem` and `x[i]` points to the same memory address, once we reassign `elem += 1`, elem just gets reassigned to another number, which does not affect `x[i]`. Note that this does not work as well since `elem` is just being copied by value and not by reference, and again further changes to `elem` will decouple it from `x[i]`.

```python
>>> x = [1, 2, 3, 4]
>>> for i, elem in enumerate(x):
...     elem = x[i]
...     elem += 1
...
>>> x
[1, 2, 3, 4]
```

To actually fix this behavior, we must make sure to call the `__setitem__(i, val)` method, which can be done as such.

```python
>>> x = [1, 2, 3, 4]
>>> for i in range(len(x)):
...     x[i] += 1
...
>>> x
[2, 3, 4, 5]
```

Note that if we had nonprimitive types in the list, then the iterator will copy by reference, and we don't have this problem.

```
>>> x = [[1], [2], [3]]
>>> for elem in x:
...     elem.append(4)
...
>>> x
[[1, 4], [2, 4], [3, 4]]
```

1.6 Queues

A `collections.deque` (double ended queue) is implemented as a doubly linked list.

1.7 Hash Sets and Hash Maps

It should be clear that anything that is mutable cannot be hashed, so you cannot have a set of lists, etc. A convenient way to bypass this is to convert into tuples. We should also be familiar with some of the dunder methods.

Definition 1.6 (Get)

There are two ways to access from a dictionary.

1. `dict[key]` retrieves the value and throws a `KeyNotFoundError` if a key does not exist.
2. `dict.get(key, def)` retrieves the value and will return `def` if the key does not exist.

A nice trick is to initialize a `collections.defaultdict`, which allow you to use `dict[key]` and automatically initializes the value to some default value if the key does not exist.

Example 1.3 ()

Again, when we iterate this with an enhanced for loop, we are just calling `next` on the keys or values that may be a copy by value or a copy by reference.

```
# y is copied by value so incrementing
# it reassigns it
>>> x = {"a" : 1, "b" : 2, "c" : 3}
>>> for k in x:
...     y = x[k]
...     y += 1
...
>>> x
{'a': 1, 'b': 2, 'c': 3}
```

```
# v is passed by value, so incrementing
# it reassigns it
>>> x = {"a" : 1, "b" : 2, "c" : 3}
>>> for v in x.values():
...     v += 1
...
>>> x
{'a': 1, 'b': 2, 'c': 3}
.
```

2 Two Pointers

Two pointers refer to methods where you are taking two pointers over some array and are comparing the value of the elements at those indices.

3 Sliding Window

Sliding window is similar to two pointers, but usually we take a window (which can be indexed by two pointers) and compare the actual property of that window (e.g. sum of all values within that subarray).

You basically take the two pointer approach, but now you want to keep an extra variable that stores some quality of that window.

Example 3.1 (Length of Longest Substring)

My thought process was very simple. I make a hashset that keeps all the letters within that sliding window and every time a new letter appears, I either increment `j` if the letter is new or increment `i` to the letter after this letter at `j` if the letter is already in the window. There are two slightly different methods to do this. You set both pointers to 0 and let `j` be the candidate pointer. The first method shows that when you reach a new letter already seen, you increment it once more and remove it from seen, and then the next step will add the new candidate letter back into seen.

```python
class Solution(object):
  def lengthOfLongestSubstring(self, s):
    """
    :type s: str
    :rtype: int
    """
    if len(s) == 0:
      return 0

    i = j = 0
    seen = set()
    res = 0

    while j < len(s):
      if s[j] in seen:
        while s[i] != s[j]:
          seen.remove(s[i])
          i += 1
        seen.remove(s[i])
        i += 1
      else:
        seen.add(s[j])
        j += 1
        res = max(res, len(seen))
    return res
```

However, this method is slightly repetitive since if you have already seen a letter, you are removing it and then adding it back in the next iteration. You can already simulate this by simply not removing from seen in the first place and incrementing `j`.

```python
class Solution(object):
  def lengthOfLongestSubstring(self, s):
    """
    :type s: str
    :rtype: int
    """
    if len(s) == 0:
      return 0

    i = j = 0
    seen = set()
    res = 0

    while j < len(s):
      if s[j] in seen:
```

```
      while s[i] != s[j]:
        seen.remove(s[i])
        i += 1
      i += 1
      j += 1
    else:
      seen.add(s[j])
      j += 1
      res = max(res, len(seen))
  return res
```

Example 3.2 (Permutation in String)

This one has a harder method but I just went through step by step.

1. I just create a dictionary to keep count of the letters and subtract them one by one rather than creating a second dictionary to keep track of the letters in `s2`.
2. To check if the letter counts matches that of `s1`, it's not good to just check through the entire dictionary at every iteration, but this can be avoided if I just take the sum of the counts and see if it is 0.
3. The final if statement in the return is needed since we could get the proper substring at the end but the while loop stops and the sum isn't checked.

```
class Solution(object):
  def createCounts(self, s):
    target = {}
    for char in s:
        if char not in target:
            target[char] = 0
        target[char] += 1
    return target

  def checkInclusion(self, s1, s2):
    """
    :type s1: str
    :type s2: str
    :rtype: bool
    """
    target = self.createCounts(s1)
    sums = sum(target.values())

    i = j = 0
    while j < len(s2):
      if sums == 0:
        return True
      if s2[j] in target:
        if target[s2[j]] > 0:
          target[s2[j]] -= 1
          sums -= 1
          j += 1
        elif target[s2[j]] == 0:
          while s2[i] != s2[j]:
            target[s2[i]] += 1
            sums += 1
            i += 1
```

```
          i += 1
          j += 1
      else:
        while i < j:
          target[s2[i]] += 1
          sums += 1
          i += 1
        i += 1
        j += 1
    return True if sums == 0 else False
```

Example 3.3 (Longest Repeating Character Replacement)

My thought process is that this is another sliding window problem and `k` gives us how many characters we can replace. We should keep track of the longest substring that we can make from replacing up to `k` characters. My thought process.

1. Maybe we can create a dictionary to keep track of the characters between `i` and `j`. We would choose the character with the maximum count and replace the rest. But keeping track of this letter would require us to iterate through the keys. I'll worry about this later.
2. No this wouldn't work since the ordering actually matters.

```


```

4 Binary Search

Before we get into anything, we must know how to do binary search. There are many forms of it, which we will go over here.

What we first want to do is initialize two pointers `i` and `j`, and at each iteration we will focus on `nums[i:j]` (remember `j`) is excluded. Therefore, we should implement a while loop such that `i < j`.

1. We set the midpoint to be `(i + j) // 2` to represent our midpoint.
2. We then check if this midpoint is equal to our target and if so return the index.
3. If `mid` is less than our target, then it means that our target has to be in `nums[mid+1:j]`. Note that it is not `nums[mid:j]` since by (1), target cannot be `nums[mid]`, so we can start from `mid+1`.
4. If `mid` is greater than our target, then we can place `j = mid`. Note that this also excludes `mid` from being a candidate. If we do `j = mid + 1`, then we are still including `nums[mid]` in our calculations and if `j = mid - 1`, then we are excluding the potential candidate `nums[mid-1]`.

While we are talking about this, there is also the problem **Search Insert Position** that asks if the target was in the list, which index would it go? The excluding last element is the best job for this.

Theorem 4.1 (Binary Search Excluding Last Element)

Note that this will converge onto the index that is the smallest number greater than or equal to the target. This is because that since we're updating `i = mid + 1` or `j = mid`, we are guaranteed to converge onto a length 2 subsequence, where the mid would be the latter element. Therefore, we would converge onto the smaller element.

```python
class Solution(object):
  def search(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    i = 0
    j = len(nums)

    while i < j:
      mid = (i + j) // 2
      if nums[mid] == target:
        return mid
      elif nums[mid] > target:
        "should not have a -1 since we haven't checked j yet"
        j = mid
      elif nums[mid] < target:
        "we can increment it to +1 since we know"
        "for sure that since mid isn't the target."
        i = mid + 1
    return -1
```

However, there are times when we will need to implement binary search where we want to start at the last element and include it.

Theorem 4.2 (Binary Search Including Last Element)

It is often the case that this will converge onto the index that is the greatest number less than or equal to the target, but there are a lot of edge cases such as when the target is less than or greater than all the elements of `nums` or if we converge onto a 2 length subsequence or a 3 length subsequence (which the end cases will be different since they can both shrink to a 1 length subsequence).

```python
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        i = 0
        j = len(nums) - 1

        while i <= j:
            mid = (i + j) // 2
            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                i = mid + 1
            elif nums[mid] > target:
                j = mid - 1

        return -1
```

Note that this binary search guarantees to converge onto the either the value that is grater than or equal to than the target.

What if we tried to set `mid = (i + j)// 2 + 1`? Simply changing this would cause an finite loop since if `i = 9` and `j = 10`, then `(i + j) // 2 + 1 = 10`. If `nums[j] > target`, then `j` wouldn't change and it would cause an infinite loop.

Now let's move onto some other problems.

Example 4.1 (Search a 2D Matrix)

This next problem is obviously a binary search problem. There's no problem in recognizing that. This is just the same thing but now we do it for a 2D matrix, which we can imagine as flattening out into an array. The thing is that if we just explicitly flatten out, then we are already using $O(mn)$ memory, which is not good. Rather, we implement two helper functions that help us convert the 1D indexing into 2D indexing.

```python
class Solution(object):

  def twoToOne(self, row, col, columns):
    return row * columns + col

  def oneToTwo(self, x, columns):
    return x // columns, x % columns

  def searchMatrix(self, matrix, target):
    """
    :type matrix: List[List[int]]
    :type target: int
    :rtype: bool
    """
    m = len(matrix)     # rows
    n = len(matrix[0])  # columns

    i = 0
    j = n * m

    while i != j:
        mid = (i + j) // 2
        mid_m, mid_n = self.oneToTwo(mid, n)
        if matrix[mid_m][mid_n] == target:
            return True
        elif matrix[mid_m][mid_n] < target:
            i = mid + 1
        elif matrix[mid_m][mid_n] > target:
            j = mid

    return False
```

Turns out that we don't even need `twoToOne`.

Example 4.2 (Koko Eating Bananas)

This is not an obvious binary search problem, but the constraints being 10^9 should be a sign that you need to solve this in $\log(n)$ time. My thought process.

1. If `h = len(piles)`, then we should return the maximum bananas in the array.
2. Maybe we can linearly increment starting from 1, and it has to be bounded by the maximum in piles. This gives me an aha moment, where I can maybe do a binary search from `1` to

max(piles).

3. But given some candidate k, how would I do this? I guess I can iterate through the list and find the time it takes to eat that whole pile. This linear solver also matches with the maximum length being 10^4.

A bug that I saw was that you have to typecast the integers to floats before you divide so that math.ceil works properly.

```python
class Solution(object):

  def computeEatingTime(self, num_bananas, k):
    return math.ceil(float(num_bananas) / float(k))

  def minEatingSpeed(self, piles, h):
    """
    :type piles: List[int]
    :type h: int
    :rtype: int
    """

    i = 1
    j = max(piles) + 1

    while i != j:
        mid = (i + j) // 2
        total_time = sum([self.computeEatingTime(bananas, mid) for bananas in piles])

        if total_time <= h:
            j = mid
        elif total_time > h:
            i = mid + 1

    return i
```

Example 4.3 (Find Minimum in Rotated Sorted Array)

A linear solution is trivial, but the log solution may indicate that this is binary search. My thought process. Let's say we initialize two pointers i = 0 and j = len(nums). If we choose the midpoint, then there are many possibilities. At first I thought it may seem like there are 8, but since the list is sorted and then rotated, this restricts the possible outcomes. We can divide it by whether i, j, or mid is the greatest.

1. nums[j] < nums[i] < nums[mid]
2. nums[i] < nums[mid] < nums[j]
3. nums[mid] < nums[j] < nums[i]

This actually causes a bug in the code since if we're dealing with say i = 9, j = 10, then mid = 9 and then we would have equalities. Therefore, we should put equalities on the conditions to get the final code.

```python
class Solution(object):
  def findMin(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """

```

```python
    i = 0
    j = len(nums) - 1

    while i != j:
      mid = (i + j) // 2
      if nums[j] <= nums[i] <= nums[mid]:
          i = mid + 1
      elif nums[i] <= nums[mid] <= nums[j]:
          j = mid
      elif nums[mid] <= nums[j] <= nums[i]:
          j = mid
      else:
          print("This should never hit.")

    return nums[i]
```

But one thing to note that if we use j.

Example 4.4 (Search in Rotated Sorted Array)

This next problem took a bit of time for me to solve since I got confused with indexing.

```python
class Solution(object):
  def search(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """

    i = 0
    j = len(nums) - 1

    while i != j:
      mid = (i + j) // 2

      if nums[mid] == target:
        return mid
      elif nums[j] >= nums[mid] >= nums[i]:
        if target > nums[mid]:
            i = mid + 1
        else:
            j = mid
      elif nums[mid] >= nums[i] >= nums[j]:
        if nums[mid] > target >= nums[i]:
            j = mid
        else:
            i = mid + 1
      elif nums[i] >= nums[j] >= nums[mid]:
        if nums[j] >= target > nums[mid]:
            i = mid + 1
        else:
            j = mid
    return i if nums[i] == target else -1

```

Example 4.5 (Time Based Key-Value Store)

My thought process:

1. You should probably create a hashmap, but since this is indexed by time, we can do binary search on this as well. This is easily dont since the timestamps inputted are strictly increasing.
2. Therefore, I can create a hashmap with values that are hashmaps as well.

```
class TimeMap(object):

  def __init__(self):
    self.map = {}
    self.tstamps = {}

  def set(self, key, value, timestamp):
    """
    :type key: str
    :type value: str
    :type timestamp: int
    :rtype: None
    """

    if key not in self.map:
        self.map[key] = {}
        self.tstamps[key] = []
    self.map[key][timestamp] = value
    self.tstamps[key].append(timestamp)

  def get(self, key, timestamp):
    """
    :type key: str
    :type timestamp: int
    :rtype: str
    """
    if key not in self.tstamps:
        return ""
    ts = self.tstamps[key]

    i = 0
    j = len(ts)

    while i != j:
      mid = (i + j) // 2
      if ts[mid] == timestamp:
          return self.map[key][ts[mid]]
      elif ts[mid] > timestamp:
          j = mid
      elif ts[mid] < timestamp:
          i = mid + 1
      else:
          raise Exception("This should not happen")

    if timestamp > ts[-1]:
      return self.map[key][ts[-1]]
    elif timestamp < ts[0]:
      return ""
    else:
      return self.map[key][ts[i-1]]
```

Example 4.6 (Median of Two Sorted Arrays)

This one is quite tricky when I looked at it first. It's trivial to merge the two and then sort them, but this is not log time. My thought process.

1. The fact that it is `log(m + n)` and not even `log(n) + log(m)` means that I can't do something like binary search on each individual list. I have to somehow do binary search over the two lists at once?
2. The solution is found if I can just eliminate the first `(m + n)//2` elements. I can maybe look at the first list, do some binary search, and compare it to the first element of the next list?
3. In linear time, we can just compare the first elements of the two lists and just pluck them off one by one. Maybe there's a faster way to implement this in binary search. I can start off by looking at list that has the greater first element and finding where that is in the other list using binary search.
4. What if I do a binary search over the possible medians? It is bounded by `min(nums1[0], nums2[0])` and `max(nums1[-1], nums2[-1])`, which are bounded by 1000, so this is basically constant time. Then given each median candidate, I can run a binary search over `nums1` and `nums2` to get the number of elements before and after it. Technically this is `O(log(n m))` but I'll go with this for now.

5 Linked Lists

Linked lists can take a bit of getting used to. You just have to keep track of your pointers and which points where, any null pointers, and temporary variables to store nodes in. Let's work with singly linked lists, and you should know how to reverse one and merge to sorted lists by heart.

Theorem 5.1 (Reverse Linked List)

At every iteration, you essentially want to keep track of the previous (so that you can point current to it), current, and next (to keep track of it when you remove the pointer from the current) nodes.

1. You start off with the previous as null. The current to be the head, and the after to be `head.next`. You start off with the degenerate case where the list is null, where you return null.
2. Then you want to have current point to the previous one and just increment everything up one.
3. At the end, you'll have `after = None`, where the while loop will stop running. At the end of the while loop, you will have the second last node point to its previous, and you increment everything such that prev is the 2nd last, curr is the last, and after is null. This will end the while loop. So curr will need to be pointing to prev one more time.

```python
class Solution:
  def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
    if head is None:
        return None

    prev = None
    curr = head
    after = head.next

    while after is not None:
      curr.next = prev
      prev = curr
      curr = after
      after = after.next
    curr.next = prev

    return curr
```

6 Binary Trees

Binary trees can almost always be solved through recursion.

Theorem 6.1 (Traversing a Binary Tree)

There are many ways you can traverse through a binary tree.

1. The *in order traversal* tells it to print everything on the left of the node, then print the node, and then print everything on the right.
2. The *pre order traversal* tells it to print the node first, then all the ones on the left, then all the ones on the right.
3. The `post order traversal` tells it to print all nodes on the left, then all on the right, and then the node itself.
4.

```
def traverse(node):
  if node:
    # print(node.val)
    traverse(node.left)
    # print(node.val)
    traverse(node.right)
    # print(node.val)
```

This makes it simple to simply store the binary tree in a list, since rather than print statements we can initialize a list and append to it.

Theorem 6.2 (Height of a Node)

The height of a node is defined as the number of nodes it takes to get from it to the lowest node.

1. By convention, the null node has a height of -1.
2. Therefore, every node is really just 1 plus the maximum of the heights of the child nodes.

To get the height of a node, it actually doesn't matter what its parents are since we are only looking at how many nodes are below it.

```
def height(node):
  # get the height of a target node
  if not node:
    return -1
  return 1 + max(height(node.left), height(node.right))
```

We can also just traverse this and store all the heights.

```
root = ...
data = []
def height(node):
  if not node:
    return -1
  left_height = height(node.left)
  right_height = height(node.right)
  data.append((node.val, 1 + max(left_height, right_height)))
  return 1 + max(left_height, right_height)
```

```

height(root)
```

Theorem 6.3 (Depth of Node)

The **depth** of a node is the number of edges between the root and the node. To calculate this, we can just add 1 plus the depth of the parent. However, we don't have access to the parent, so we must use an accumulator.

1. If the node is None, then the depth doesn't exist and we return -1.
2. If the node exists, then we can print its depth according to the accumulator and call this same function on its child nodes with the accumulator incremented.

Just getting the depth of a node with only the `node` as a parameter is impossible, since we need access to its parents. To get this information, we must traverse from the root down.

```
root = ...
def depth(node, target, curr_depth):
  if not node:
    return -1

  if node == target:
    return curr_depth

  left_depth = depth(node.left, target, curr_depth + 1)
  if left_depth != -1:
    return left_depth

  right_depth = depth(node.right, target, curr_depth + 1)
  return right_depth

depth(root, 0)
```

We can also just traverse this whole thing and store all the depths.

```
data = []
def depth(node, curr_depth):
  if not node:
    return
  data.append((node.val, curr_depth))
  depth(node.left, curr_depth+1)
  depth(node.right, curr_depth+1)

depth(root, 0)
```

Note that there are a few key differences between finding the height and depth.

1. The recursive approach to height requires us to actually *return* something since that will be used for calculations lower in the stack, while that of depth doesn't actually since the values are in the `curr_depth` accumulator.

What if you wanted to store both the depths and the heights?

```
data = []
def traverse(node, curr_depth):
  if not node:
    return -1
```

```

  left_height = traverse(node.left, curr_depth + 1)
  right_height = traverse(node.right, curr_depth + 1)

  data.append((node.val, 1 + max(left_height, right_height), curr_depth))

  return 1 + max(left_height, right_height)

traverse(root, 0)
```

Example 6.1 (Maximum Depth of Binary Tree)

This can really be found by taking the maximum height and just subtracting one from it.

7 Stacks, Queues, Priority Queues

8 Graphs

The difference between DFS and BFS is really whether you use a stack or a queue, most obviously in the iterative sense.

Definition 8.1 (DFS)

The recursive algorithm is

```
visited = set()
def dfs(start):
  if start not in visited:
    visited.add(start)
    # do something
    neighbors = ...
    for neighbor in neighbors:
      dfs(neighbor)
```

The iterative algorithm uses a stack, which mirrors the function call stack.

```
visited = set()

def dfs(start):
  toExplore = []
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.pop()
    # Do something
    neighbors = ...
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        toExplore.append(neighbor)
```

Definition 8.2 (BFS)

The recursive version of BFS is very nontrivial, so we show only the iterative version here.

```python
visited = set()
def bfs(start):
  toExplore = collections.deque()
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.popleft()
    # Do something
    neighbors = ...
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        toExplore.append(neighbor)
```