CODEPATH (/compsci)

Search

UMPIRE Interview Strategy

Edit Page (https://github.com/codepath/compsci_guides/wiki/UMPIRE-Interview-Strategy/_edit)

Page History (https://github.com/codepath/compsci_guides/wiki/UMPIRE-Interview-Strategy/_history)

Introduction

During a technical interview, the way you work towards your solution is often more important than the binary result of solving the problem or not. There are several good ways to show and communicate your work as you work through an algorithm problem. The UMPIRE method is another way to state the best practices that underlie the most successful approaches in a catchy name.

UMPIRE stands for:

1. **Understand** what the interviewer is asking for by using test cases and questions about the problem.
2. **Match** what this problem looks like to known categories of problems, e.g. Linked List or Dynamic Programming and strategies or patterns in those categories.
3. **Plan** the solution with appropriate visualizations and pseudocode.
4. **Implement** the code to solve the algorithm.
5. **Review** the code by running specific example(s) and recording values (watchlist) of your code's variables along the way.
6. **Evaluate** the performance of your algorithm and state any strong/weak or future potential work.

UMPIRE Example

Let's practice using the UMPIRE technique on the following problem.

> **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

Understand

Test Cases

One of the best ways to understand a problem is to come up with test cases. E.g.,

```
Input: 3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1; partition = 5
Output: 3 -> 1 -> 2 -> 5 -> 5 -> 8 -> 10
```

Coming up with good test cases can be an art, but I typically like to cover one happy path or an average case. And any interesting edge cases. This can be tricky as it depends on the problem.

```
Input: 1; partition = 1
Output: 1
```

Questions

Remember that the interview is a dialog, so understanding the problem is best done as a dialog as well. Asking questions, like

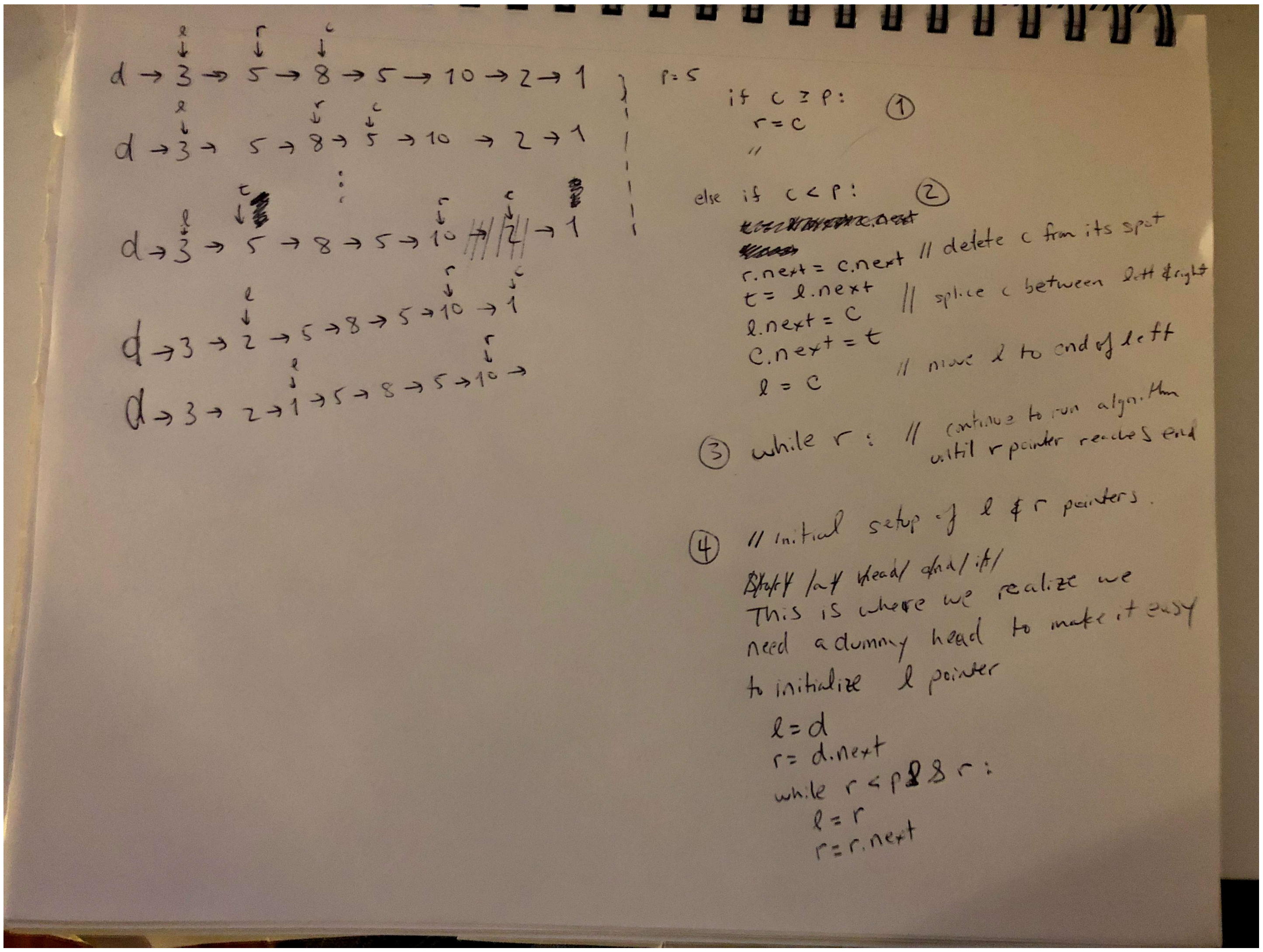
1. Should this be done in place?
2. Does the ordering beyond the left/right of the partition matter?

are very useful components to the dialog and are helpful for this problem.

Match

Linked List problems are usually some of the easiest to spot and so matching this to the Linked List category is a good choice. So we know we would like to employ Pointer Bookkeeping during the algorithm planning. So let's list out all the things that we might consider and our belief of match Likely, Neutral, Unlikely.

1. Linked List = Likely
2. Pointer Bookkeeping = Likely
3. MultiPass = Unlikely
4. Reverse = Unlikely
5. Two-Pointer = Likely
6. Dummy Head = Neutral ### Plan This is the harder part for Linked List problems as dealing with pointer management is where solving this problem can go awry. So we take advantage of our matching of Pointer Bookkeeping tool and perhaps a two-pointer strategy and dummy-head to move forward to visualize the solution.


d → 3 → 5 → 8 → 5 → 10 → 2 → 1
d → 3 → 5 → 8 → 5 → 10 → 2 → 1
d → 3 → 5 → 8 → 5 → 10 → 2 → 1
d → 3 → 2 → 5 → 8 → 5 → 10 → 1
d → 3 → 2 → 1 → 5 → 8 → 5 → 10 →
p = 5

```
if c ≥ p:        ①
  r = c
  //

else if c < p:   ②
  ~~t = c.next~~
  r.next = c.next // delete c from its spot
  t = l.next      // splice c between left & right
  l.next = c
  c.next = t
  l = c           // move l to end of left

③ while r :  // continue to run algorithm
             until r pointer reaches end

④ // initial setup of l & r pointers.
  This is where we realize we
  need a dummy head to make it easy
  to initialize l pointer
    l = d
    r = d.next
    while r < p && r :
        l = r
        r = r.next
```

In this pointer visualization, I started with the following assumption that I would have one pointer `l` at the end of the left side and `r` pointer at the last known right side position. Then `c` or really just `r.next` would test if the next node is greater than or less than the partition value `p`. In the case labeled 1, value of `c` is >= `p` so in that case the `r` pointer moves forward.

I then fast forward the bookkeeping visualization, to the next position where that condition doesn't hold. This is situation 2, in this case, we need to remove the `c` node from its position and place it somewhere on the left which in the written code is after the `l` node, but we recognize that we need to introduce a temporary pointer `t` so that we can move the `c` node into position without orphaning the rest of the list. Since this seems to handle all possible cases, I fast forward the visualization to deal with the end case. This situation 3 is when `r` reaches the end or `r.next`. So most likely the whole algorithm will be in a while loop to forward `r` until this condition fails.

Finally, I revisit the initial condition of setting up the `l` and `r` pointer, in situation 4. Here, however, I realize that it would have been easier to utilize the dummy-head technique because there is a chance that the very first node could be >= `p`, thus I would not be able to get the initial invariant I setup in the beginning with `l` pointing to the last node on the left side. So situation with situation 4, I create a `d` node or dummy-head that points to the actual head of the list and this is a great place to start `l`. However, now that we have the dummy-head and combined with the knowledge that we don't care about the ordering of the nodes within the left or right partitions we realize we could alter the algorithm to not use an `l` pointer and simply place the `c` nodes after the dummy to simplify the algorithm.

Implement

If our plan stage was successful, we should be able to implement the code by simply glueing the plan together while filling out the details of our language specific implementation. I like to write my implementation out in layers, where the first layer is me writing basic setup code and then talking to the pieces of the plan.

```
def partition(list, p):
  d = ListNode('dummy')
  d.next = list
  # 4: Initialize r pointer to first value >= p
  # 3: Loop until we run out of unknown values
  # 1: Handle right side value
  # 2: Handle left side value

  ## Pop off the dummy
  return d.next
```

Then, I proceed to implement the separate parts of the plan:

```
def partition(list, p):
  d = ListNode('dummy')
  d.next = list
  # 4: Initialize r pointer to first value >= p
  r = d.next
  while r and r.val < p:
    r = r.next
  # 3: Loop until we run out of unkown values
  while r.next:
        c = r.next
        if c >= p:
            # 1: Handle right side value
            r = c
        else:
            # 2: Handle left side value
            r.next = c.next
            t = d.next
            d.next = c
            c.next = t
  ## Pop off the dummy
  return d.next
```

As we can see, the mix of understanding the ordering requirements of the algorithm and the dummy-head technique we were able to further simplify our problem by removing the need for the situation 4 or initialization of our `l` and `r` pointers and getting rid of `l` altogether (really it was just renamed to `d`).

Review

The very important, but too often skimped on, review stage is next. Unfortunately, most interviewees just skim through the code to say yeah that looks right, but that IS NOT what review means. It means go through it as if you are debugging it, assuming there is a bug. If you do you woudn't just be saying yep, thats what I'd meant to write. Instead, you would be bringing the big debugging "guns". Watchlist of variables and stepping through the code line by line are the typical tools of choice to go for this stage.

Happy Path

1. Start with an average or happy case example, `list = 1 -> 2 -> 5 -> 8 -> 3` and `p = 5`.

2. After second line `d -> 1 -> 2 ...`
3. `r == node(1)`
4. Examining the `r` while loop puts `r = node(5)`
5. Examine the next while loop condition and indeed we have `r.next` so we enter the loop
6. Set `c = node(8)`
7. Uh oh, I notice a bug on the next line `c >= p` doesn't make sense because `c` is a node and `p` is a value. So we would go up to change our code above to say `c.val >= p` .
8. Ok now `c.val == 8` so the if condition is true and we set `r = node(8)` .
9. Then we go back to the while loop to check `r.next == node(3)` so we go in the while loop
10. We set `c = node(3)`
11. `c.val < p` so we go to the else condition
12. `r.next` is set to `None`
13. `t` is set to `node(1)`
14. We point `d` to `c` so we have `d -> 3`
15. But then we point `c` to t so we have `d -> 3 -> 1 -> 2 -> 5 -> 8`
16. We go back to the top of the while, but fail the while condition because `r.next == None`
17. Exiting the loop the last line to run returns `d.next` or `3 -> 1 -> 2 -> 5 -> 8` , which satisfies the solution.

Edge Cases

1. Next, we check the most common edge case or passing the function an empty list. `partition(None, 5)`
2. We go do the second line without problems and we now have `d` , that is the dummy head is pointing to None
3. `r` is then set to `None`
4. The while loop doesn't proceed as `r` is `None`
5. Now we get to the second while loop and we notice a problem, `r.next` would dereference `None` which is not allowed.
6. We could fix this a number of ways, but we decide to just change the loop condition to be `while r and r.next`
7. After making that change, we continue to step through the code and exit the while loop immediately becase `r == None`
8. At the last line, we return `d.next` or `None` , which satisfies the constraints of our algorithm.

After our review of our algorithm we have the following code:

```
def partition(list, p):
  d = ListNode('dummy')
  d.next = list
  # 4: Initialize r pointer to first value >= p
  r = d.next
  while r and r.val < p:
    r = r.next
  # 3: Loop until we run out of unkown values
  while r and r.next:
        c = r.next
        if c.val >= p:
            # 1: Handle right side value
            r = c
        else:
            # 2: Handle left side value
            r.next = c.next
            t = d.next
            d.next = c
            c.next = t
  ## Pop off the dummy
  return d.next
```

Evaluate

The last step of the UMPIRE strategy is to discuss the pros and cons of your algorithm with your interviewer. An algorithm much like a good interview is not amenable to correct/incorrect label, instead there are often things to discuss about what you like about your algorithm or code and what you would like to change depending on conditions, inputs, expectations, the business, etc. A great place to start this conversation and is almost always required is to discuss the asymptotic performance of the algorithm. Let's evaluate our algorithm. To do this, let's start with any loop statements. We have two loop statements. The first, is iterating the `r` pointer until it reaches the first node on the right side. In the best case, this wouldn't go further than the first position, in the worst case we would go to the end of the list without finding a right node. So this loop has `O(N)` time complexity where `N` is the length of the list.

Now we go to the second loop. Again, we are iterating `r` and if we ignore the rest of the contents of the loop outside of how `r` is manipulated we get

```
while r and r.next:
    if r.next >= p:
        r = r.next
    else:
        r = r.next.next
```

So again, the worst we can do is start at the beginning of the list and go to the end so this loop is `O(N)`. So total time complexity is `O(N)`.
Next, we move on to space complexity. We see that we are only requiring a constant number of pointers in addition to the list, so our space complexity is `O(1)`.