Recurrence Relations

May 18, 2020

0.1 Problem Statement

Previously, we considered the following problem:

> Given a positive integer `n`, write a function, `print_integers`, that uses recursion to print all numbers from `n` to `1`.
>
> For example, if `n` is `4`, the function shuld print `4 3 2 1`.

Our solution was:

```
In [1]: def print_integers(n):
            if n <= 0:
                return
            print(n)
            print_integers(n - 1)
```

```
In [2]: print_integers(5)
```

```
5
4
3
2
1
```

We have already discussed that every time a function is called, a new *frame* is created in memory, which is then pushed onto the *call stack*. For the current function, `print_integers`, the call stack with all the frames would look like this:

Note that in Python, the stack is displayed in an "upside down" manner. This can be seen in the illustration above—the last frame (i.e. the frame with `n = 0`) lies at the top of the stack (but is displayed last here) and the first frame (i.e., the frame with `n = 5`) lies at the bottom of the stack (but is displayed first).

But don't let this confuse you. The frame with `n = 0` is indeed the top of the stack, so it will be discarded first. And the frame with `n = 5` is indeed at the bottom of the stack, so it will be discarded last.

We define time complexity as a measure of amount of time it takes to run an algorithm. Similarly, the time complexity of our function `print_integers(5)`, would indicate the amount of time taken to exceute our function `print_integers`. But notice how when we call `print_integers()` with a particular value of `n`, it recursively calls itself multiple times.

In other words, when we call `print_integers(n)`, it does operations (like checking for base case, printing number) and then calls `print_integers(n - 1)`.

Therefore, the overall time taken by `print_integers(n)` to execute would be equal to the time taken to execute its own simple operations and the time taken to execute `print_integers(n - 1)`.

Let the time taken to execute the function `print_integers(n)` be $T(n)$. And let the time taken to exceute the function's own simple operations be represented by some constant, k.

In that case, we can say that

$$T(n) = T(n-1) + k$$

where $T(n-1)$ represents the time taken to execute the function `print_integers(n - 1)`.

Similarly, we can represent $T(n-1)$ as

$$T(n-1) = T(n-2) + k$$

We can see that a pattern is being formed here:

1. $T(n) = T(n-1) + k$
2. $T(n-1) = T(n-2) + k$
3. $T(n-2) = T(n-3) + k$
4. $T(n-3) = T(n-4) + k \ldots\ldots$
5. $T(2) = T(1) + k$
6. $T(1) = T(0) + k$
7. $T(0) = k1$

Notice that when `n = 0` we are only checking the base case and then returning. This time can be represented by some other constant, $k1$.

If we add the respective left-hand sides and right-hand sides of all these equations, we get:

$$T(n) = nk + k1$$

We know that while calculating time complexity, we tend to ignore these added constants because for large input sizes on the order of 10^5, these constants become irrelevant.

Thus, we can simplify the above to:

$$T(n) = nk$$

We can see that the time complexity of our function `print_integers(n)` is a linear function of n. Hence, we can say that the time complexity of the function is $O(n)$.

0.2 Binary Search

Overview Given a **sorted** list (say `arr`), and a key (say `target`). The binary search algorithm returns the index of the `target` element if it is present in the given `arr` list, else returns -1. Here is an overview of how the the recursive version of binary search algorithm works:

1. Given a list with the lower bound (`start_index`) and the upper bound (`end_index`).
2. Find the center (say `mid_index`) of the list.
3. Check if the element at the center is your `target`? If yes, return the `mid_index`.
4. Check if the `target` is greater than that element at `mid_index`? If yes, call the same function with right sub-array w.r.t center i.e., updated indexes as `mid_index + 1` to `end_index`

5. Check if the `target` is less than that element at `mid_index`? If yes, call the same function with left sub-array w.r.t center i.e., updated indexes as `start_index` to `mid_index - 1`
6. Repeat the step above until you find the target or until the bounds are the same or cross (the upper bound is less than the lower bound).

Complexity Analysis Let's look at the time complexity of the recursive version of binary search algorithm.

> Note: The binary search function can also be written iteratively. But for the sake of understanding recurrence relations, we will have a look at the recursive algorithm.

Here's the binary search algorithm, coded using recursion:

```
In [4]: def binary_search(arr, target):
            return binary_search_func(arr, 0, len(arr) - 1, target)

        def binary_search_func(arr, start_index, end_index, target):
            if start_index > end_index:
                return -1

            mid_index = (start_index + end_index)//2

            if arr[mid_index] == target:
                return mid_index
            elif arr[mid_index] > target:
                return binary_search_func(arr, start_index, mid_index - 1, target)
            else:
                return binary_search_func(arr, mid_index + 1, end_index, target)


In [28]: arr = [0, 1, 2, 3, 4, 5, 6, 7, 8]
         print(binary_search(arr, 5))

5
```

Let's try to analyze the time complexity of the recursive algorithm for binary search by finding out the recurrence relation.

Our `binary_search()` function calls the `binary_search_func()` function. So the time complexity of the function is entirely dependent on the time complexity of the `binary_search_func()`.

The input here is an array, so our time complexity will be determined in terms of the size of the array.

Like we did earlier, let's say the time complexity of `binary_search_func()` is a function of the input size, `n`. In other words, the time complexity is $T(n)$.

Also keep in mind that we are usually concerned with the worst-case time complexity, and that is what we will calculate here. In the worst case, the `target` value will not be present in the array.

In the `binary_search_func()` function, we first check for the base case. If the base case does not return `True`, we calculate the `mid_index` and then compare the element at this `mid_index` with

the `target` values. All the operations are independent of the size of the array. Therefore, we can consider all these independent operations as taking a combined time, k.

Apart from these constant time operations, we do just one other task. We either make a call on the left-half of the array, or on the right half of the array. By doing so, we are reducing the input size by $n/2$.

> Note: Remember that we usually consider large input sizes while calculating time complexity; there is no significant difference between 10^5 and $(10^5 + 1)$.

Thus, our new function call is only called with half the input size. We said that $T(n)$ was the time complexity of our original function. The time complexity of the function when called with half the input size will be $T(n/2)$.

Therefore:

$$T(n) = T(n/2) + k$$

Similarly, in the next step, the time complexity of the function called with half the input size would be:

$$T(n/2) = T(n/4) + k$$

We can now form similar equations as we did for the last problem:

1. $T(n) \quad = T(n/2) + k$
2. $T(n/2) = T(n/4) + k$
3. $T(n/4) = T(n/8) + k$
4. $T(n/8) = T(n/16) + k \ldots\ldots$
5. $T(4) = T(2) + k$
6. $T(2) = T(1) + k$
7. $T(1) = T(0) + k1$ [(1)]

8. $T(0) = k1$

[(1)] If we have only one element, we go to 0 elements next

From our binary search section, we know that it takes $log(n)$ steps to go from $T(n)$ to 1. Therefore, when we add the corresponding left-hand sides and right-hand sides, we can safely say that:

$$T(n) = log(n) * k + k1$$

As always, we can ignore the constant. Therefore:

$$T(n) = log(n) * k$$

Thus we see that the time complexity of the function is a logarithmic function of the input, n. Hence, the time complexity of the recursive algorithm for binary search is $log(n)$.