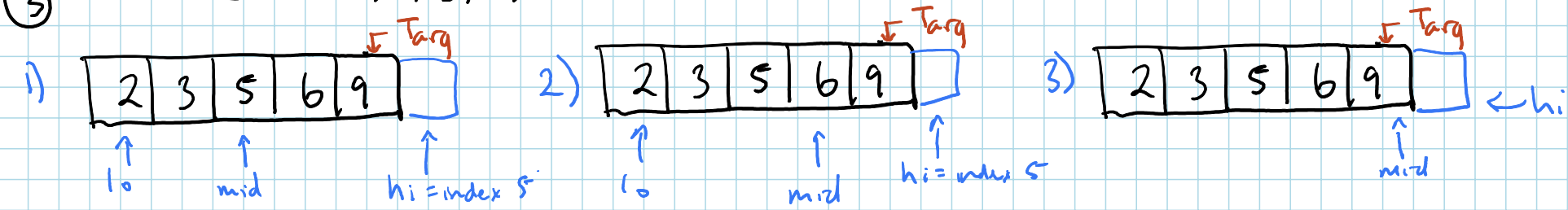
CPE 202
W2020

Lab 1

Richard Hwa
1/13/20

(3) search([2,3,5,6,9], 9)



The search function takes two parameters - a sorted list to search for the target value that it is seeking - and returns the index of the target value if it is found, or None if it does not exist in the list or if the list is empty. This function uses a recursive binary search and gives us a Big-O of O(logn) since the binary aspect halves the search amount every time the function is called.

For the example search([2,3,5,6,9], 9) we have set the low and high indices to be static at the first index of the list and high to be the length of the list, respectively. Setting high to the length of the list allows mid to be floored to the correct value but never exceeding the limits of the list since it is given as (mid = (mid+hi)//2). In this example, we start off mid at the middle of the list but since the function is looking to find 9 and the value at mid is 5, we look at the larger half of the list and move to the right until we find the target 9 and return the index it is at as given by mid in the 3rd diagram.

(4) fib(3)

Fibonacci definition: $F(0)=0$, $F(1)=1$, $F(2)=1$

$F(n) = F(n-1) + F(n-2)$ for $n>1$

$fib(3) = fib(2) + fib(1) = 1+1 = 2$

The fib(n) function is by definition a recursive problem since it requires the previous two values F(n-1) and F(n-2) in the sequence to compute F(n). Knowing this, we recursively step back from our given value of n until we hit n=3 when we trigger our base cases of F(1)=1 and F(2)=1 by Fibonacci's definition. Then the function will back out of the recursion and add all the previous values it stepped over to get to the base case to give us the value for fib(n).

In our example, letting n=3 be our input argument already triggers the base cases once we step into the recursion. So, all our function has to do is take the 2 given definitions for n=1 and n=2 to add together to give us our desired value for the function fib(3) which evaluates to equal 2.

The time complexity of this function turns out to be O(2^n) since the recursion is called twice. Once for F(n-1) and another time for F(n-2). Calling this an n amount of times quickly reduces our ability to compute the Fibonacci value since we are executing two operations every time it is called and bringing our Big-O to O(2^n).

Of course, any cases not caught by this recursion (I'm talking about you n=0), will be defaulted to return 0 which also is determined by a Fibonacci sequence's definition.

5 Factorial

The factorial function works much like the Fibonacci function, except that it requires one base case instead of two and multiplies n*(n-1) instead of adding (n-1)+(n-2).

When I tried to compute 1000!, python gave me a value for the iterative solution although it turns out to be incorrect. The recursive method hits the maximum recursion depth and then stops calculating altogether, giving no value. The reason for this is because Python has a recursion limit in order to prevent searches that go too deep because these types of problems take a significant amount of time and also overload stack memory. A recursive solution should not be used in general for a problem of this nature anyways since it is not good practice to have to go 1000 steps deep. A cleaner and quicker solution should be looked for and Python's limit is there for good reason. However, if you do not want to follow good practices or you really need to implement your solution recursively, you can raise the recursion depth in Python if you so wish.

As mentioned in the previous section, just like how recursion takes up too much stack memory, this can be a possible problem as well for an iterative implementation. The reason that my iterative factorial solution obtains a wrong answer could be due to this as well as rounding errors.

Information has been obtained on this topic from googling "recursion limit on python", looking at the stack overflow page on Wikipedia, an actual Stack Overflow answer, and many other online resources including geeksforgeeks.org