29/03/2023

Time and Space Complexity of Recursion

1) Couting print using recursion

```
void print (n) {
        // Base Case
        cout << n;
        print (n-1);
}
```

print (5) → k time
↓
print (4) → k time
↓
print (3) → k time
↓
print (2) → k time
↓
print (1) → k time

Total calls = 5 i.e n
If one call takes k time, then n calls

take time n * k time.
Time complexity = O(n * k) = O(n)
↳ can be neglected

While calculating space complexity, find the instance taking maximum space & that will be the space complexity.

Instance of max space	print(1)	→ k space
	print(2)	→ k space
	print(3)	→ k space
	print(4)	→ k space
	print(5)	→ k space
	main()	

O(n * k) = O(n) is the space complexity

Note ↳ We don't have to consider the space taken by main as we are finding space complexity of print function.

2) Factorial using recursion

```
int factorial (n) {
        if (n == 0 || n == 1)      } k1
                return 1;
        return n * factorial (n-1);
                (k2)
}
```

$T(n) = k + T(n-1)$
↳ (k1 + k2)

$fact(5) \rightarrow k$

$5 * \downarrow$

$fact(4) \rightarrow k$

$4 * \downarrow$

$fact(3) \rightarrow k$

$3 * \downarrow$

$fact(2) \rightarrow k$

$2 * \downarrow$

$fact(1) \rightarrow k$

Time complexity $= O(n * \underbrace{k}_{\text{neglected}}) = O(n)$

fact(1)	$\rightarrow$ k space
fact(2)	$\rightarrow$ k space
fact(3)	$\rightarrow$ k space
fact(4)	$\rightarrow$ k space
fact(5)	$\rightarrow$ k space
main()	

Space complexity $= O(n * k) = O(n)$

Note $\rightarrow$ $O(10^6)$ is constant space.

3) Power of 2^n

$2^n = 2 \times 2^{n-1}$

$\hookrightarrow 2 \times 2^{n-2}$

$\hookrightarrow 2 \times 2^{n-3}$

$\vdots \ldots 2 \times 2^0$

$f(n) = 2 \times f(n-1)$ where $f(n) = 2^n$

$$
\left.
\begin{array}{ll}
f(5) & \rightarrow k \text{ time} \\
\downarrow & \\
2 \times f(4) & \rightarrow k \text{ time} \\
\downarrow & \\
2 \times f(3) & \rightarrow k \text{ time} \\
\downarrow & \\
2 \times f(2) & \rightarrow k \text{ time} \\
\downarrow & \\
2 \times f(1) & \rightarrow k \text{ time} \\
\downarrow & \\
2 \times f(0) & \rightarrow k \text{ time}
\end{array}
\right\} (n+1)k \text{ time}
$$

TC

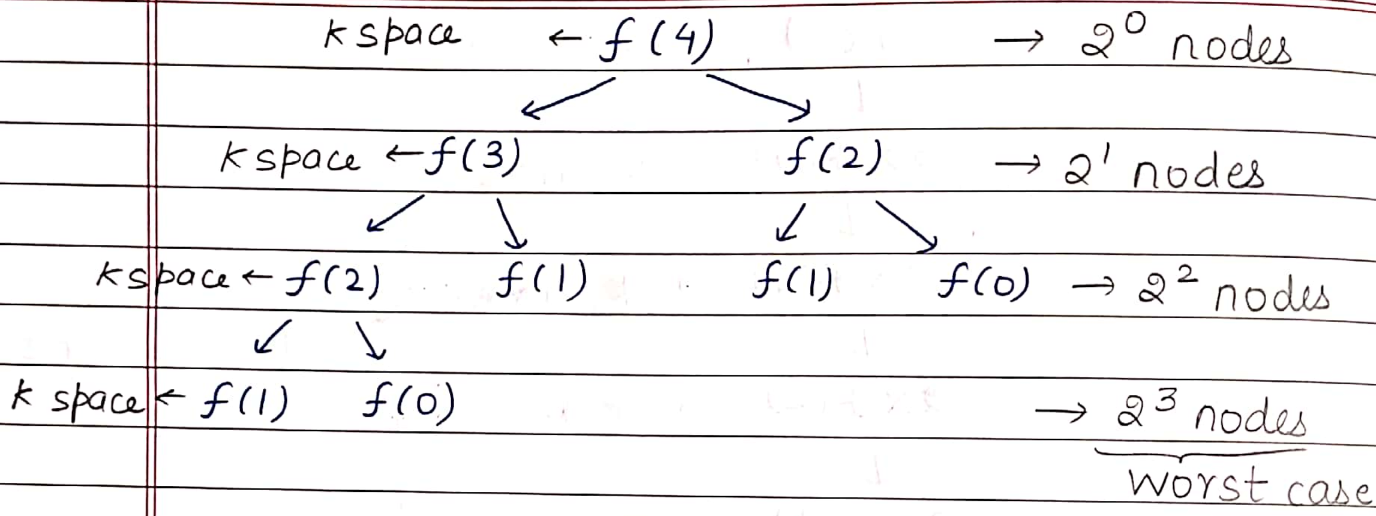
n+1 calls will take $(n+1)k$ time i.e $O(n)$ only.

f(0)
f(1)
f(2)
f(3)
f(4)
f(5)
main()

Each call takes k space, so n+1 calls take $(n+1)k$ space i.e $O(n)$ space only.
SC

Time complexity $= O(n)$
Space complexity $= O(n)$

4) Fibonacci using recursion
$f(n) = f(n-1) + f(n-2)$



Total nodes $= 2^0 + 2^1 + 2^2 + \dots + 2^{n-1} + 2^n$
Solving GP we get time complexity $= O(2^n)$ & this is known as exponential time complexity.

Space complexity $= O(n)$ if the call is taking constant space.

5) Jump stairs
Same time & space complexity as that of fibonacci.

6) Maximum element in the array.
We are linearly traversing the array in the recursion call & hence time taken is $O(n)$.

Space complexity $= O(n * \overset{\text{each call taking k space}}{k}) = O(n)$

7) Minimum element in the array
Time complexity $= O(n)$
Space complexity $= O(n)$

8) Binary search using recursion

n size → k time
↓
n/2 size → k time
↓
n/4 size → k time
↓
⋮
↓
$n/2^a$ size } single block (k time)

(the levels above together: log n)

$\frac{n}{2^a} = 1$

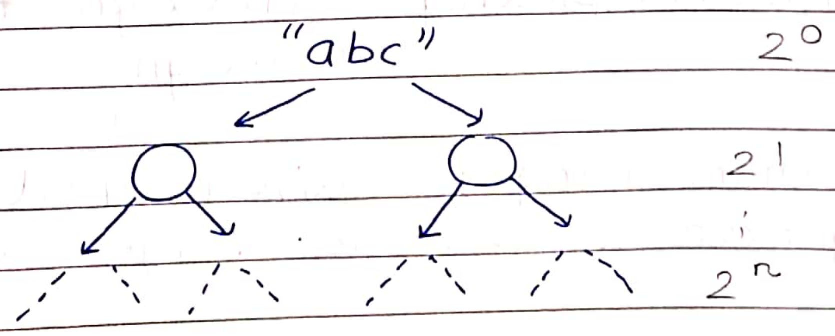
$n = 2^a \Rightarrow a = \log_2 n$

Time complexity = $O(k * \log_2 n) = O(\log_2 n)$
↳ neglect

Space complexity = $O(k * \log_2 n) = O(\log_2 n)$
↳ each call takes constant space

Note → log n is not considered as constant space.
Iterative binary search > Recursive binary search.

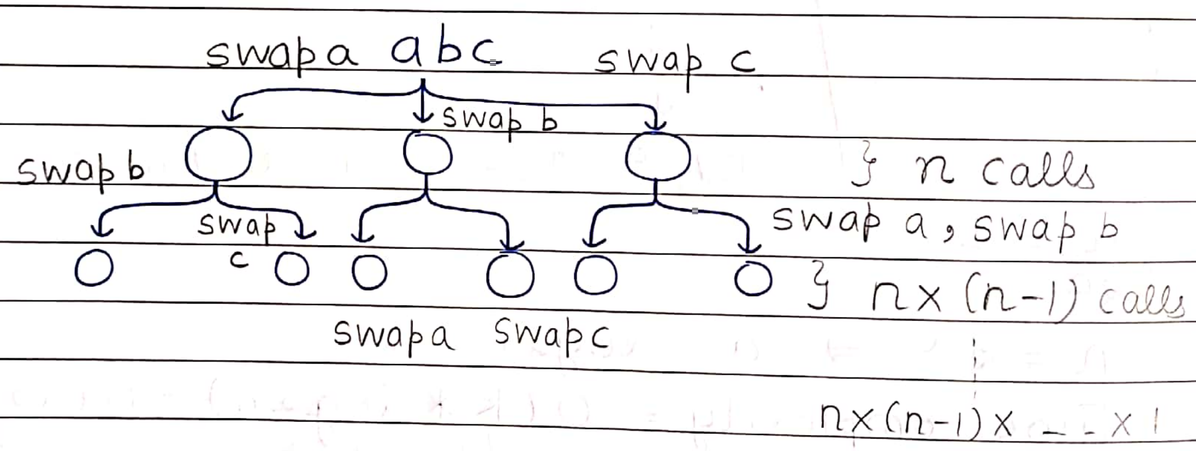
9) Subsequence of string It was based on include exclude pattern



Same as that of fibonacci. Hence time complexity is $O(2^n)$.

No. of levels = length of string (L) } Levels
Space complexity = $O(L)$

10) Permutations of string.



Time complexity = $O(n \times (n-1) \times \ldots \times 1)$
Time complexity = $O(n!)$

Space complexity is the homework.

11) Merge Sort

No. of levels = $\log n$ } Same as binary search
No. of calls = $\log n$
Time to merge $n/2$ sized array = $O(n/2 + n/2) = O(n)$
Hence Time Complexity = $O(n \times \log n)$
= $O(n \log n)$
merge ↲ ↳ calls

The solution using iterative method has already been discussed in merge sort video.

Space complexity = $O(n)$

We have created 2 arrays i.e left & right arrays in the merge function.