Big O

big O (big O) => big O is upperbound on runtime

big Ω (big Omega) => big Ω is lowerbound of runtime

big Θ (big theta) => big Θ is when algo will be O(N) only when its both O(N) and Ω(N). Θ gives tight bound on runtime

→ Industry uses big theta (tight bound on runtime) as definition of big O.

Best, worst, Expected cases are types of time complexity cases.

eg. For quicksort
worst – $O(N^2)$ best – O(N)
expected – O(log(N))

→ Space Complexity => while recursive code adds ~~up in~~ (stacks) up in space complexity. It is not neccessary. If adjacent elements of array are added space complexity is not O(N) rather O(1) bcz only one array is there. where as time complexity will be O(N).

→ O(2N) = O(N) [drop constants]

Page No.:
Date:

→ Drop Non-Dominant Terms - $O(N^3+N) \rightarrow O(N^3)$

→ ~~Graph~~ Below depicts rate of increase of big (O) order

$$O(x) < O(x\log x) < O(x^2) < O(2^x) < O(x!)$$

→ If we have two loops working simultaneously with runtime A and B then
big O = $O(A+B)$

and if we have both loops nested then
big O = $O(A * B)$

Amortized Time

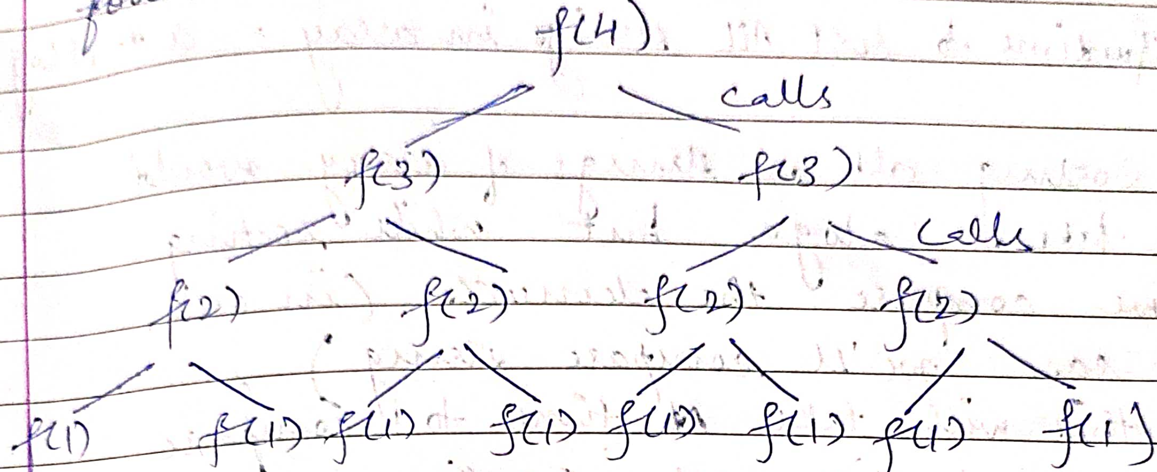
A concept which takes worst case and expected case, both in account.

Q. How do we get log N Runtimes?

Whenever we have a situation where after single step (of algo) we are down to "/i" elements then.
For eg. - in binary search

Recursive Runtime

if a function has such that 2 functions are being called inside it (same function twice) then the way to derive runtime is as follows



Depth of tree = 4 (~~nth~~ basically N of f(N))
and the increase in nodes is as follows

1 → 2 → 4 → 8

∴ runtime = $2^0 + 2^1 + 2^2 + 2^3$ ......

⇒ runtime = $O(\text{branches}^{\text{depth of tree}-1}) = O(2^3)$

Examples

→ for (i < a);
  for (j < a)

BUT

for (i < a):
  for (j < b):

Runtime for above code will be $O(N^2)$

Runtime for above will be $O(ab)$

Page No.:
Date:

(2) Q What is Runtime of sorting each string in array and then sorting array?

→ Runtime to sort string = N log N

→ Runtime to sort ALL strings in array = a * N log N

→ Sorting all a strings of array would take a log a but while sorting we compare the elements (in our case we'll compare string) so this will take N time to compare strings in each iteration.

Hence total for this is N * a log a.

→ Adding them all the runtime for entire process will be

~~N (a log a +~~

RUNTIME = N * a (log a + log N)

8

Page No.:
Date:

(3) What is runtime of following code which sums values of all nodes in balanced binary search tree.

```
int sum (Node node) {
    if (node = NULL) {
        return 0;
    }
    return sum(node, left) + node.value + sum(node, right
```

Ans Way 1 - Look at meaning of code. They are summing values in all nodes.

so O(N).

Way 2 - Another way to look at recursive codes is that we know runtime for recursive code is

$$= O(\text{branches}^{\text{depth}})$$

We have 2 branches here (left and right) and since the tree is a balanced binary search tree depth will be log N for N nodes

hence runtime = $O(2^{\log N})$
since log N has base 2, with log properties

runtime = O(N)

Date:

(4) What is the runtime of a code that counts all ~~pe~~ permutations of a string?

The code is a recursive code calling permutation function throughout each iteration of the string length.

So, runtime for permutation function is $n!$ and length of string is n.

Hence runtime for permutation could be $O(n * n!)$

Since this all will be called n times so overall runtime will be
$= O(n^2 * n!)$

(5) code prints fibonacci numbers from 0 to n. what is time complexity?

```
void allFib (int n) {
    for(i=0; i<n; i++){
        print ( fib(i)) ;

int fib (int n) {
    if (n<=0) return 0;
    else ~~...~~; if (n==1) return 1;
    return fib(n-1) + fib(n-2)
```

Here, runtime of first function is $O(n)$ and runtime of second function is $O(2^n)$

So overall runtime = $O(2^n + n) = O(2^n)$