31/01/2023

Algorithm Analysis

↗ unambiguous instruction.

Algorithm is a finite set of instruction which if followed to accomplish a particular task.

two main criteria for judging the merits of algorithms:
- correctness &
- efficiency.

Properties of an Algorithm:
- Finiteness
- Absence of ambiguity
- Definition of sequence
- Input / output (should be finite)
- Feasibility.

Algorithm Design Goals:

concern of software Engineering.
↳ easy to understand, code & debug.

concern of data structures and algorithm analysis
↳ algorithm that makes efficient use of resources

Algorithm Analysis:

CODE —(code Modelling)→ RUNTIME FUNCTION —(Asymptotic Analysis)→ COMPLEXITY CLASS

```
for(i=0; i<n; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

f(n) = 2n

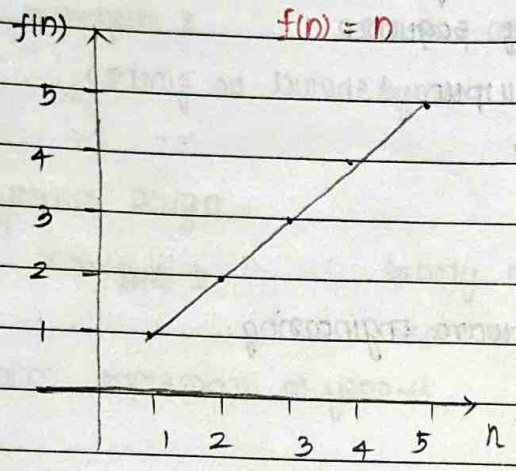
O(n)

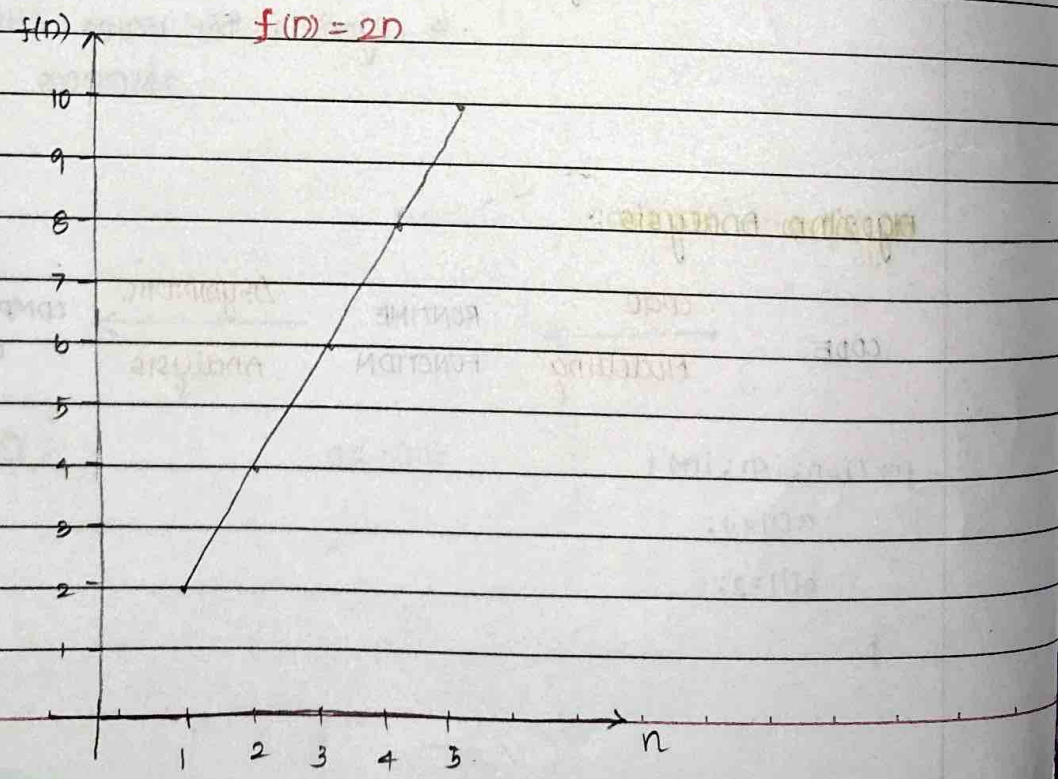
Algorithm analysis :- the overall process of characterizing code with a complexity class, consisting of

Code Modelling :- code → Function describing code's runtime
Asymptotic analysis :- Function → complexity class describing asymptotic behavior.

Draw a graph for $f(n) = n$ and $f(n) = 2n$ for value of $n = 1$ to 5

$n = 1$ to 5





Types of Analysis :-

Worst case

Best case

Average case

Lower bound ≤ Average time ≤ upper bound.

Worst case :-

Defines the input for which the algorithm takes a long time.

Defines the input for which the algorithm runs the slowest.

Upper bound of the Runtime

Best case :-

Defines the input for which the algorithm takes the least time.

Defines the input for which the algorithm runs the fastest.

Lower bound of the Running time.

Average case :-

Provides a prediction about the running time of the algorithm.

Run the algorithm many times (with random input),

Asymptotic Notation :-

Assume that the given algorithm is represented in the form of function f(n)

Three types :-

Big oh (O) Upper bound

Omega (Ω) Lower bound

Theta (Θ) Order Function

the value must be right bound.

$f(n) = n$

Upper bound $O(n) = n^2, n^3, \ldots$

Lower bound $\Omega(n) = \log(n)$ (or) some constant functions

Order function $\theta(n) = n$

Big oh notation :-

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all

$n \geq n_0, \quad f(n) \leq c \cdot g(n)$

Upper bound Function :-

$g(n)$ is greater than $f(n)$ for $n \geq n_0$

$g(n)$ is the upper bound of $f(n)$.

Example :-

$f(n) = 3n^2 - 100n + 6$

Let assume $g(n) = n^2$

$c \cdot g(n) > f(n)$

$c = 3$

$3n^2 > 3n^2 - 100n + 6$

01/02/2023

Page No. 07
Date / /

Given a function that models some piece of code, characterize that function's growth rate asymptotically (as n approaches infinity)

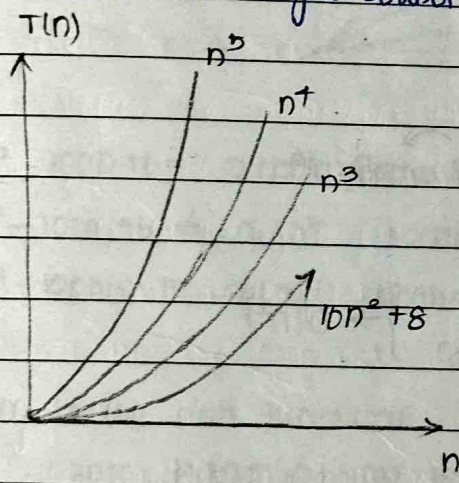
n - size of the input, n is always a non-negative integer.

$f(n) = 100n^2 + 8$

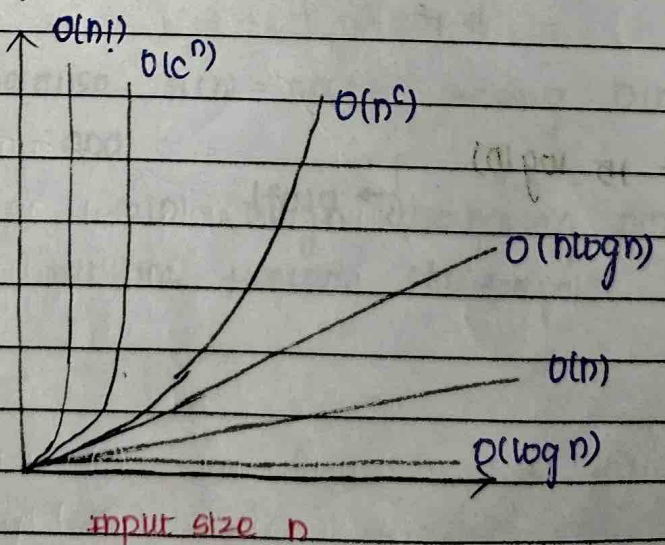
Bigoh (O) is an upper bound on that function's growth rate.

constants and smaller terms ignored

$O(n^2)$ is a tight bound. Function is also $O(n^3)$



Comparison of algorithms:



Running time complexity in terms of Big-O o(f(n))

Complexity classes :

time complexity	name	Example
1	constant	Adding an element to the front of a linked list
log n	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
n log n	Linear Logarithmic	Sorting n items by "divide and conquer". merge sort
n^2	quadratic	shortest path between two nodes in a graph
n^3	cubic	Matrix Multiplication
2^n	Exponential	The towers of Hanoi problem.

↓
(non polynomial algorithms)

$f_1(n) = 30n^3 + 10$

↳ tight bound $O(n^3)$ → $O(n^4)$

$f_2(n) = 10,000,000$

↳ (constant function) } → $O(n^2)$

$f_3(n) = 2n^2 + 5n + 20$

$2n^2$

↳ n^2 } → $O(n^2)$

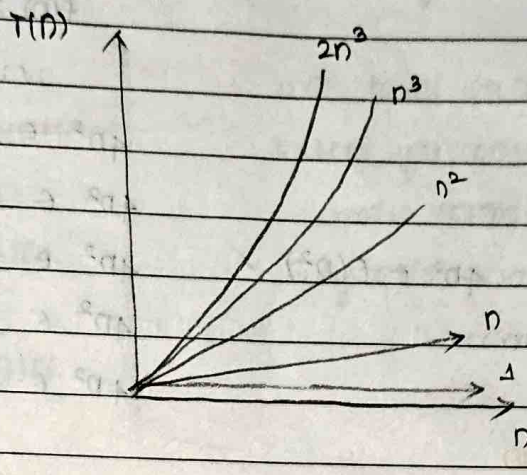
$f_4(n) = 15 \log(n)$ } → $O(n^2)$

$\log n < n^2$

omega- Ω notation

Lower bounding function

The Ω notation can be defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$. $g(n)$ is an asymptotic tight lower bound for $f(n)$.



$3n^2 - 100n + 6 = \Omega(n^2)$ because $2.99\,n^2 < 3n^2 - 100n + 6$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ because $3n^2 - 100n + 6 < n^3$

$3n^2 - 100n + 6 = \Omega(n)$ because $10^{10^{10}}\,n < 3n^2 - 100n + 6$

f1 and f2 are two functions

Both f1 and f2 are having O(n)

Whether f1 and f2 are same?

No because $f1(n) = 5n + 6$ having O(n)

and

$f2(n) = 5 \log n$ also having O(n)

but the function $f1 \neq f2$.

Examples:

Big-Oh

$f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$f(n) \leq c \cdot g(n)$

$4n^2 \in O(1)$ ✗

$4n^2 \in O(n)$ ✗

$4n^2 \in O(n^2)$ ✓

$4n^2 \in O(n^3)$ ✓

$4n^2 \in O(n^4)$ ✓

Big-omega

$f(n) \in \Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$f(n) \geq c \cdot g(n)$

$4n^2 \in \Omega(1)$ ✓

$4n^2 \in \Omega(n)$ ✓

$4n^2 \in \Omega(n^2)$ ✓

$4n^2 \in \Omega(n^3)$ ✗

$4n^2 \in \Omega(n^4)$ ✗

$4n^2 \in \Theta(n^2)$ ✓

theta-Θ Notation (Order Function)

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0 \}$

$3n^2 - 1000n + 6 = \Theta(n^2)$ because O and Ω

$3n^2 - 1000n + 6 \neq \Theta(n^3)$ because O only

$3n^2 - 1000n + 6 \neq \Theta(n)$ because Ω only.

A: $O(n^2)$ ✓

B: $\Theta(n^2)$

↳ exactly more than this

$(n, \log n, 1)$ n^2

Big-Theta

$f(n)$ is $\Theta(g(n))$ if

$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Big Theta is "equal to"

code takes exactly this long to run.

Except for constant factors and lower order terms.

only exists when Big-Oh == Big-omega.

Intuition about the notations :

O (Big-Oh)	$f(n) \leq g(n)$
Ω (Big-Omega)	$f(n) \geq g(n)$
Θ (Theta)	$f(n) = g(n)$

Rates of growth :

$1 < \log \log N < \sqrt{\log N}$

$\log \sqrt{N} < 2^{\log N} < N$

$N < \log(N!) < N \log N$

$N \log N < N^2$

$N^2 < 2^N < 4^N < N! < 2^N < 2^{2N}$

Practically not feasible for implementation

DD case analysis when varying other input properties besides n can change runtime!

(Linear search):

complexity cases:

Worst case

Best-case

Average-case

Amortized (consecutive inputs of n divided by m.
m → "most challenging")

Exercise:

$2n \in O(n)$? ✓

$n \in O(n^2)$? ✓

$n^2 \in O(n)$? ×

$n \in O(1)$? ×

$100 \in O(n)$? ✓

$214n + 34 \in O(2n^2 + 8n)$? ✓

↳ $n \in O(n^2)$ ✓

$f(n) = 5n$ pick tightest bound

$f(n) = O(n^5)$ $f(n) = O(n^3)$

$f(n) = O(n \log n)$ $f(n) = O(n)$ (tightest bound)

Show $f(n) = 2n+6 = O(n)$

claim : $f(n) = 2n+6 = O(n)$

Must find c, n_0 such that for all $n > n_0$,

$$2n+6 <= c*n$$

$\therefore c=3$ and $n_0 = 6$

$\therefore f(n) = 2n+6 = O(n)$

Analysis rules :-

1. Assume an arbitary time unit.
2. Execution of the following operation takes time 1.

   (i) assignment operation
   (ii) single I/O operations
   (iii) single Boolean operations, numeric comparissons.
   (iv) single arithmetic operations
   (v) function return
   (vi) array index operations, pointer dereferences

can also be approximated by taking the time for assignment statement to be unity (except in case of function calls included in statement)

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clausos in the selection.

As an approximation time for the condition evaluation may be ignored, as they are constants

Example :-

```
if (count>10)
    print                    } -> 1
else {
    print;                   } -> 3 ✓
    count=10;
}
```

4. Loop execution time is the sum, over the number of times the loop is executed, of the body time + time for the loop setup + time for the loop check and update operations.

As an approximation time for loop bot check and update and setup may be ignored as they are constants

```
for (count =0 ; count <10; count++)
  {
    x = a[i];
    x = x + count;
  }
```

```
count =0
while (count<10) {
    x = a[i];
    x = x + count;
count = count +i;
}
```

Always assume that loop executes the maximum number of iterations possible.

5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

As an approximation, time for call setup and parameter calculations, may be ignored if they are independent of the input

Example : x = x + fun CA (D)

Analysis of program control statements :-

while loop : Analyze like a for loop

if statement : take greater complexity of then/else clauses.

switch statement : take complexity of most expensive case.

subroutine call : complexity of subroutine.

General rules :-

Basic operations	constant time
consecutive statements	sum of number of statements
conditionals	test + larger branch cost
Loops	sum of iterations
Function calls	cost of function.
Recursive function	solve "recurrence relation"

nested loops :-

single loop over n ⇒ $f(n) = n$

loop within loop ⇒ $f(n) = n^2$

loop within loop within loop ⇒ $f(n) = n^3$

Example :-

```
statement 1;
S 2;
S 3;
for (int i=1 ; i<=N ; i++) {
    S4;
}
for (int i=1 ; i<=N ; i++) {
    S5;
    S6;
    S7;
}
```

3

n(1)

n(3)

$f(n) = 3 + n + 3n$

$= 3 + 4n$

$\in O(n)$

Rule of thumb :-

single programs can be analysed by counting the nested loops of the program.

A single loop over n items yields $f(n) = n$

A loop within a loop yields $f(n) = n^2$

A loop within a loop within a loop yields $f(n) = n^3$

Given a series of for loops that are sequential, the slowest of them determines the asymptotic behaviour of the program.

Two nested loops followed by a single loop is asymptotically

Examples: (Exercise):

1. $f(n) = n^3(4\log n - \log n) + (n^2)^2$

Express the function in terms of O.

$\Rightarrow 4n^3\log n - n^3\log n + (n^4)$

↑ higher order term.

$f(n) \in O(n^4)$

2. $f(n) = \log_8(2^{4n})$ $\because \log_a a^n = n$

$\Rightarrow \log_8(2^n) \Rightarrow O(n)$

$f(n) \in O(n)$

3. $f(n) = 500n\log n + n^2\log n$

$\Rightarrow n\log n + n^2\log n$

↑ higher order term

$\Rightarrow n^2\log n$

$f(n) \in O(n^2\log n)$

4. $f(n) = (n \cdot (n^2+1)^2)$

$\Rightarrow n \cdot (n^2)^2$

$\Rightarrow n \cdot n^4 \Rightarrow n^5$

$\Rightarrow O(n^5)$

$f(n) \in O(n^5)$

5. $f(n) = (n \cdot (n+1)^2)$

$f(n) \Rightarrow O(n^3)$

$f(n) \in O(n^3)$

6. $f(n) = \log n^4 + \log^2 n$

$= 4 \log n + \log^2 n$

↑ Higher term

$= O(\log^2 n)$

$f(n) \in O(\log^2 n)$

7. $f(n) = n + \log n + n^{1/6}$

$= O(n)$

$f(n) \in O(n)$

8. $f(n) = n^{1/4} + (\log^4 n)^2$

$f(n) \in O(n^{1/4})$

9. $f(n) = n^3 \cdot n^2 + (n+2n)^2$

$= n^5 + (3n)^2$

↑ higher term

$= O(n^5)$

$f(n) \in O(n^5)$

10. $f(n) = (n^2+1)^{10}$ Find θ

$= (n^2)^{10} = n^{20}$

$f(n) \in \theta(n^{20})$

11. $f(n) = \sqrt{10n^2+7n+3}$ Find θ

↑ higher term.

$f(n) = \sqrt{10n^2}$

$= \sqrt{n^2} = n$

$= \theta(n)$

$f(n) \in \theta(n)$

12. $f(n) = 2n \log (n+2)^2 + (n+2)^2 \log n/2$ Find θ

$= 4n \log(n+2) + n^2 \log n/2$

$= n \log(n) + n^2 \log n$

↑ higher term

$= \theta\, n^2 \log n$

$f(n) \in \theta(n^2 \log n)$

13. $f(n) = 2^{n+1} + 3^{n-1}$ Find θ

$= 2^n + 3^n \quad = \theta(3^n)$

↑ hr

$f(n) \in \theta(3^n)$

If $t_1(n) = O(g_1(n))$ and $t_2(n) = O(g_2(n))$,

what is $t_1(n) + t_2(n)$?

$= O(\max(g_1(n), g_2(n)))$

$O(n) \leftarrow$ for (i=1; i <= n; i++) print (i)

$O(1) \leftarrow x = y + z$

$f(n) = n+1$

$f(n) \in O(n)$

$t_1(n) = O(g_1(n))$ $\quad$ $t_2(n) = O(g_2(n))$

$t_1(n) \leq c_1 g_1(n)$ for $n > n_{01}$ $\quad$ $t_2(n) \leq c_2 g_2(n)$ for $n > n_{02}$

$t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$ for $n > \max(n_{01}, n_{02})$

Let us consider $c_3 = \max(c_1, c_2)$

$t_1(n) + t_2(n) \leq c_3 g_1(n) + c_3 g_2(n)$

$t_1(n) + t_2(n) \leq c_3 (g_1(n) + g_2(n))$

$t_1(n) + t_2(n) \leq 2c_3 (\max(g_1(n), g_2(n)))$

$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$

Where.

$c = 2c_3$

$= 2 \max(c_1, c_2)$

$n \geq \max(n_{01}, n_{02})$

If $t_1(n) = \Omega(g_1(n))$ and $t_2(n) = \Omega(g_2(n))$,
What is $t_1(n) + t_2(n)$?

$= \Omega(\min(g_1(n), g_2(n)))$

$t_1(n) = \Omega(g_1(n))$ $\qquad$ $t_2(n) = \Omega(g_2(n))$

$t_1(n) \geq c_1 g_1(n)$ for $n \geq n_{01}$ $\qquad$ $t_2(n) \geq c_2 g_2(n)$ for $n \geq n_{02}$

$t_1(n) + t_2(n) \geq c_1 g_1(n) + c_2 g_2(n)$ for $n \geq \min(n_{01}, n_{02})$

Let us consider $c_3 = \min(c_1, c_2)$

$t_1(n) + t_2(n) \geq c_3 g_1(n) + c_3 g_2(n)$

$\geq c_3 (g_1(n) + g_2(n))$

$\geq 2c_3 (\min(g_1(n) + g_2(n)))$

$t_1(n) + t_2(n) \in \Omega(\min(g_1(n), g_2(n)))$

where

$c = 2c_3$

$= 2 \min(c_1, c_2)$

$n \geq \min(n_{01}, n_{02})$

07.02.2023

code complexity :

(i)
```
sum = 0                          ⇒ 1
for (j = 1 ; j <= n ; j++)
    for (i = 1 ; i <= j ; i++)
        sum++ ;
```
n → $\frac{n(n+1)}{2}$

$f(n) = (n^2+n)/2 + 1$

$= n^2 + n$

$= O(n^2)$

When control variable of inner loop is dependent on the outer loop

When n = 4

j	i
1	1
2	1 2
3	1 2 3
4	1 2 3 4

$1 + 2 + 3 + 4 = \frac{4(5)}{2}$

(ii)
```
sum = 0                          ⇒ 1
for (j = 1 ; j <= n ; j++)
    for (i = 1 ; i <= j ; i++)
        sum++ ;
```
⇒ $\frac{n(n+1)}{2}$

```
for (k = 0 ; k < n ; k++)
    A[k] = k ;
```
⇒ n

$f(n) = 1 + \frac{(n^2+n)}{2} + n$

$= n^2 + 2n = O(n^2)$

(iii)
```
sum1 = 0
for (i = 1 ; i <= n ; i++)
    for (j = 1 ; j <= n ; j++)
        sum1++ ;
```
n^2

```
sum2 = 0
for (i = 1 ; i <= n ; i++)
    for (j = 1 ; j <= n ; j++)
        sum2++
```
n^2

$f(n) = n^2 + n^2$

$= 2n^2 = O(n^2) = \Theta(n^2)$

$k = n \Rightarrow k/2$
$k = 1 \Rightarrow k * 2$
} $\log n$

(iv)
```
sum1 = 0;
for (k=1; k<=n; k*=2)
    sum1++;
```

$\log n + 1$

$f(n) = \log n + 1$
$= \log n$
$= O(\log n)$

n	k
1	1
2	1 2
3	1 2
4	1 2 4
5	1 2 4
6	1 2 4
7	1 2 4
8	1 2 4 8

1 2 4 8 16 32 64 128 ....

(v)
```
sum1 = 0
for (k=1; k<=n; k*=2)        ⇒ log n
    for (j=1; j<=n; j++)     } n
        sum2++;
```

$f(n) = n \log n$
$= O(n \log n)$

(vi)
```
sum2 = 0
for (k=1; k<=n; k*=2)
    for (j=1; j<=k; j++)
        sum2++;
```

k	j
1	1
2	1 2 ⇒ 2
4	1 2 3 4 ⇒ 2^2
8	1 2 3 4 5 6 7 8 ⇒ 2^3
⋮	⋮
n	$2^{\log n}$

$1 + 2 + 2^2 + \cdots + 2^{\log n} = 2^{\log n + 1} - 1$

$f(n) = 2^{\log n + 1} - 1$
$= O(\log n)$ $= O(n)$

$2^x = n$
$\log 2^x = \log n$
$x = \log n$

(vii)
```
sum = 0;
for (int i=1; i <= N*N; i++) {          → N^2
    for (int j=1; j <= N*N*N; j++) {    → N^3   } N^5   } N^5 + 1
        sum ++;
    }
}
```

$f(n) = N^5 + 1$

$= O(N^5)$

(viii)
```
sum = 0;
for (int i=1; i <= N; i += C) {   } N/C   } N/C + 1
    sum ++;
}
```

$f(n) = N/C + 1$

$\in O(N)$

N	C	
10	2	⇒ 5
12	4	⇒ 3
16	8	⇒ 2

(ix)
```
sum = 0;
for (int i=1; i <= N; i *= C)   } log_C N   } log_C N + 1
    sum ++;
```

$f(n) = \log_C N + 1 \in O(\log_C N)$

(x)
```
sum = 0;  i = 1;
while ( i <= N )   } N/C   } N/C + 2
{  sum ++;
   i += C;
}
```

similar to for loop.

$f(n) = N/C + 2$

$\in O(N)$

06.02.2023

Page No. 27
Date

Recurrences and solving Recurrence Relations

Analysis

Complexity of Recursive Functions:

→ decide on a parameter indicating size of input
→ identify basic operation
→ check whether the number of times the basic operation is executed may vary on different inputs of the same size.
    * If it may, the worst, average and best cases must be investigated separately.
→ set up a recurrence relation with an appropriate initial condition.
→ solve the recurrence.

Factorial Function - Recursive Definition:

$n! = 1 * 2 * \cdots * (n-1) * n$ for $n \geq 1$

$0! = 1$.

Recursive definition of n!

$F(n) = n * F(n-1)$ for $n \geq 1$

$F(0) = 1$

Algorithm F(n)
// computes n! recursively
// input: A nonnegative integer n
// output: the value of n!
if n=0 return 1.
else return F(n-1) * n.
(recursive call)

Recurrence Relation :

Page No. 28
Date

→ denoting time with size of input n.

$T(n) = T(n-1) + 1 ; \quad n > 0$

↑ To calculate $F(n-1)$

↖ To multiply $F(n-1)$ by n.

$T(0) = 0$ No multiplication when $n = 0$

$$T(n) = T(n-1) + 1 ; \quad n > 0$$
$$T(0) = 0$$

solving Recurrence relation (substitution method)

1. $T(n) = T(n-1) + 1 ; \quad n > 0$

$T(0) = 0$

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

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

$= T(n-3) + 1 + 2 = T(n-3) + 3$

$= T(n-3) + 3$

$T(n) = T(n-i) + i$

when $n - i = 0$ ($\because n = i$)

$T(n) = T(0) + n$

$= 0 + n = n$

$T(n) = n$

$T(n)$ is $O(n)$

2. $T(n) = T(n/2) + 1 \quad ; n > 1$

$T(1) = 1$

$T(n) = T(n/2) + 1$

$= T(n/4) + 1 + 1 = T(n/4) + 2$

$= T(n/8) + 1 + 2 = T(n/8) + 3$

$= T(n/8) + 3$

$T(n) = T(n/2^i) + i$

When $n/2^i = 1 \quad (\because n = 2^i) \Rightarrow i = \log n$

↳ taking log on both sides

$T(n) = T(1) + \log n$ | $\log n = \log 2^i$

$T(n) = 1 + \log n$ | $i = \log n$

$T(n)$ is $O(\log n)$

3. $T(n) = 2 * T(n/2) + 1 \quad ; n \geq 1$

$T(1) = 1$

$T(n) = 2 * T(n/2) + 1$

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

$= 8 * T(n/8) + 3$

$T(n) = 2^i * T(n/2^i) + i$

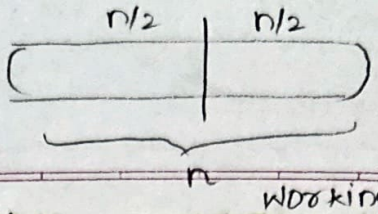
When $n/2^i = 1 \quad n = 2^i$

↳ taking log on both sides

$T(n) = 2^{\log n} * T(1) + \log n$ | $\log n = \log 2^i$

$= 2^{\log n} * (1) + \log n$ | $i = \log n$

$T(n) = 2^{\log n} + \log n$



working on both the roots

3. $T(n) = 2 * T(n/2) + 1 \quad ; \quad n > 1$

$T(1) = 1$

$$T(n) = 2 * T(n/2) + 1$$

$$= 2 \left[2 * T(n/4) + 1 \right] + 1$$

$$= 4 * T(n/4) + 2 + 1$$

$$= 4 * T(n/4) + 3$$

$$= 4 \left[2 * T(n/8) + 1 \right] + 3$$

$$= 8 * T(n/8) + 4 + 3$$

$$= 8 * T(n/8) + 7$$

$$T(n) = i * T(n/i) + (i-1)$$

When $n/i = 1 \Rightarrow n = i$

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

$$= n * 1 + n - 1 = n + n - 1$$

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

$T(n)$ is $O(n)$

Solve recurrence relation :-

1. $T(n) = 2T(n-1) - 1$, if $n > 0$ $n = 1, 2, \ldots$
   $T(0) = 1$ otherwise

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

$$= 2\left[2T(n-2) - 1\right] - 1$$

$$= 4T(n-2) - 2 - 1$$

$$= 4T(n-2) - 3$$

$$= 4\left[2T(n-3) - 1\right] - 3$$

$$= 8T(n-3) - 4 - 3$$

$$= 8T(n-3) - 7$$

$$T(n) = 2^i T(n-i) - (2^i - 1)$$

$n - i = 0$

$\Rightarrow i = n$

when $n - i = 0 \Rightarrow i = n$

$$T(n) = 2^n T(n-i) - (2^n - 1)$$

$$= 2^n T(0) - (2^n - 1)$$

$$= 2^n (1) - (2^n - 1)$$

$$= 2^n - 2^n + 1$$

$$T(n) = 1$$

$$\in O(1)$$

2. $M(n) = 2M(n-1) + 1$ for $n > 1$

$M(1) = 1$

$M(n) = 2\,M(n-1) + 1$

$= 2\left[2\,M(n-2) + 1\right] + 1$

$= 4\,M(n-2) + 2 + 1$

$= 4\,M(n-2) + 3$

$= 4\left[2\,M(n-3) + 1\right] + 3$

$= 8\,M(n-3) + 4 + 3 = 8\,M(n-3) + 7$

$M(n) = 2^i M(n-i) + (2^i - 1)$

when

$n - i = 1 \quad i = n - 1$

$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1} M(n-n+1) + 2^{n-1} - 1$

$= 2^{n-1} M(1) + 2^{n-1} - 1$

$= 2^{n-1} + 2^{n-1} - 1$

$= 2(2^{n-1}) - 1$

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

$M(n) = 2^n - 1 \quad \in O(2^n)$

3. $T(n) = 3T(n-1) - 1$ if $n > 0$

$= 1$ otherwise

$T(n) = 3T(n-1) - 1$

$= 3[3T(n-2) - 1] - 1$

$= 9T(n-2) - 3 - 1$

$= 9T(n-2) - 4$

$= 9[3T(n-3) - 1] - 4$

$= 27T(n-3) - 9 - 4$

$= 27T(n-3) - 13$

$T(n) = 3^i T(n-i) - \left(\frac{3^i - 1}{2}\right)$

when

$n - i = 0 \Rightarrow i = n$

$T(n) = 3^n T(n-n) - \left(\frac{3^n - 1}{2}\right)$

$= 3^n T(0) - \left(\frac{3^n - 1}{2}\right)$

$= 3^n - \left(\frac{3^n - 1}{2}\right)$

$= 3^n - \frac{3^n}{2} + \frac{1}{2}$

$T(n) = \frac{2 \cdot 3^n - 3^n + 1}{2} = \frac{1}{2}(2 \cdot 3^n - 3^n + 1)$

$T(n) \in O(3^n)$

4. $x(n) = x(n/2) + n$ for $n > 1$

$x(1) = 1$

solve for $n = 2^k$

$$x(n) = [x(n/2)] + n$$

$$= x(n/4) + n/2 + n$$

$$= x(n/4) + 3n/2$$

$$= x(n/8) + n/4 + 3n/2$$

$$= x(n/8) + \frac{n+6n}{4}$$

$$= x(n/8) + 7n/4$$

$$x(n) = x\left(\frac{n}{2i}\right) + \frac{(2i-1)n}{i}$$

when $n/2i = 1$ $\quad n/2 = i$

$$x(n) = x\left(\frac{n}{2(n/2)}\right) + \frac{(2(n/2)-1)n}{n/2}$$

$$= x(1) + \frac{(n-1)n}{n/2}$$

$$= 1 + \frac{2*(n-1)*n}{n}$$

$$= 1 + 2n - 2$$

$$x(n) = 2n - 1$$

$\in O(n)$

solve for $n = 2^k$

$$= 2(2^k) - 1$$

$$x(2^k) = 2^{k+1} - 1$$

1.
```
for (i=10; i<40; i++)
    for (j=0; j<n; j++)
        print('*')
```
(inner loop: n) } 30n

$f(n) = 30n$

$\in O(n)$

2.
```
for( i=0; i<n; i++)          ⇒ n
{   sum = 0;
    x = 10;
    y = x + i;               ⇒ 4
    continue;
    for(j=0; j<i; j++)       } not be considered
        sum += j
}
```
always continue stmt ignores the below code.

$f(n) = 4 + n$

$\in O(n)$

3.
```
long power (long x, long y)

    if (n==0) return 1;
    if (n==1) return x;

    if ((n%2==0)==0)
        return power (x*x, n/2);
    else
        return power (x*x, n/2) * x;
```

Find the recurrence relation and then compute the time complexity of the algorithm using substitution method.

Recurrence relation :-

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

$T(1) = x$ (constant)

$T(0) = 1$

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

$= T(n/4) + c + c$

$= T(n/8) + c + c + c$

$= T(n/8) + 3c$

$T(n) = T(n/2^i) + ic$

when $n/2^i = 1$

$n = 2^i$

taking log on both sides

$\log n = \log 2^i \Rightarrow i = \log n$

$T(n) = T(n/2^{\log n}) + \log n * c$

$= T(1) + \log n * c$

$T(n) = x + \log n * c$

↑ constant ↑ constant

$T(n) \in \log n$