* Step by step procedure to a given problem which has the following characters is called 'Algorithm'.

* Input: Algorithm can have 0 or more input
* Output: Algorithm can have 1 or more output
* Finiteness: It should have finite no. of steps
* Definiteness: Every step must be precise and clear without any confusions
* Effectiveness: Each step should terminate in finite amount of time

* Analysis of algorithms

→ Why?

For a given problem there might be multiple soln. To know which soln works faster in terms of space and time we analyze algorithm

Algorithm is independent of architecture

* What is running time?

Running time is dependent on architecture. It is the process of determining how processing time increases as the input is increased

* How to compare algorithms?

i) Execution time: Time taken by a program to run
The execution time of algorithm depends on system architecture and programming language: we reject the use of execution time as a parameter to compare algorithms.

(ii) No. of statements:
It depends on programming language.
Ex: 'C' - Linear Search
python - a.find(5)

Ex: pow(2,n)
-OR-
1<<n
-OR-
y=1
for(i=1; i<=n; i++)
y=y*2

It depends on programmer's intelligence
All give same output.

(iii) Ideal Solution:
Expressing running as a function of i/p size n. It is represented by F(n) and T(n)
It is independent of machine architecture and programming language. ✓

(iv) Rate of growth: The rate at which running time ↑ as the function of i/p size n

Rate of Growth	Name
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	linear logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
$n!$	factorial

Asymptotic Notations:

* $O(g(n))$ (Big oh) - For worst case
* $\Theta(g(n))$ (thita) - For average case
* $\Omega(g(n))$ (Omega) - For best case.

$O(g(n))$:

Let our function be $t(n)$, then

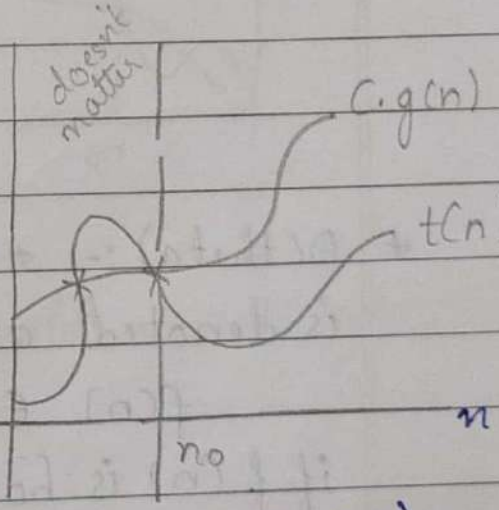
$$t(n) \leq c \cdot g(n) \quad \forall\ n \geq n_0$$

c - some constant

g(n) - maximum limit

- $n^2 \leq n^3 \quad \forall\ n > 0$

  $O(n^3)$



$f(n) = O(g(n))$

* Asymptotic notations:
* O(big-oh): A function $f(n)$ is said to be in $O(g(n))$ is denoted as $f(n) \in O(g(n))$ if $f(n)$ is bounded above by some constant multiple of $g(n)$ $\forall$ larger

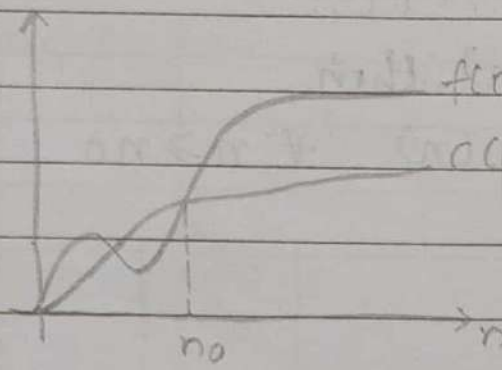
$$f(n) \leq c \cdot g(n) \quad \forall\ n \geq n_0$$

* Ω (big-omega): A function $f(n)$ is said to be in $\Omega(g(n))$ is denoted as

$$f(n) \in \Omega(g(n))$$

if $f(n)$ is bounded below by some constant multiple of $g(n)$ $\forall$ large n

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



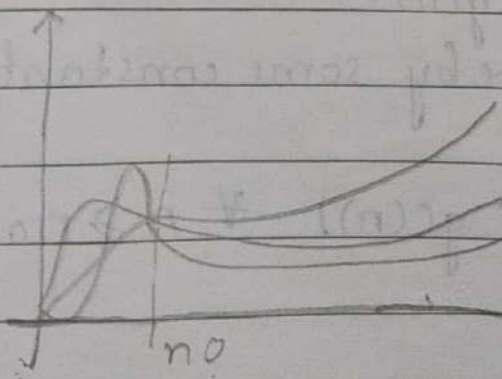
$f(n) = \text{Omega}(g(n))$

* Θ (theta) :- A function $f(n)$ is said to be in $\Theta(g(n))$ is denoted as

$$f(n) \in \Theta(g(n))$$

if $f(n)$ is bounded both above and below by some constant multiples of $g(n)$ $\forall$ large n

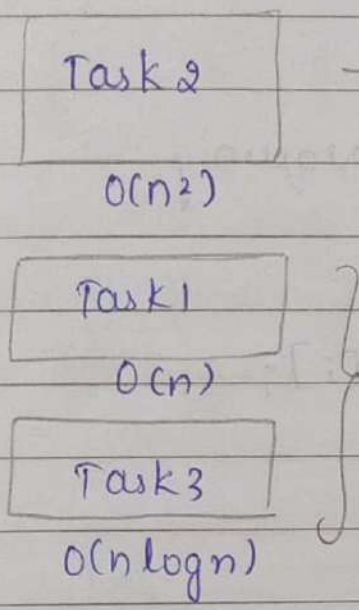
$$c_2\,g(n) \leq f(n) \leq c_1\,g(n) \quad \forall\, n \geq n_0$$



$f(n) = \text{theta}(g(n))$

* Amortized Analysis:-

In amortized analysis the task which is executed min no. of times can be neglected



Overall TC
$= O(n^2)$

Amortized TC
$O(n \log n)$

```
for (i=0; i<t; i++)          → t
{   int x, y;                                   O(t(x+y))
    printf("Input the indexes\n");               or
    scanf("%d %d", &x, &y);                      O((x+y))

for(int m=0; m<=x; m++)      → x     Max of x is n-1
  {  sum1 = sum1 + arr[m];           Max of y is n-1
  }

for(int n=0; n<=y; n++)      → y.    O(n+n)
  {  sum2 = sum2 + arr[n];           O(n)
  }

diff = abs(sum1 - sum2);
    printf("diff");
}
```

```
// #include <stdio.h>
// { int arr[100];
//     printf("Enter the size of array\n");
//     scanf("%d",&n);
// }
```

Using cf or cumulative frequency

```
cf[0] = a[0]
for(i=1; i<=n; i++)        → n
{ cf[i] = cf[i-1] + a[i];  } → once
}
for(i=0; i<t; i++)         → t
{ read x, y,
  print abs(cf[x] - cf[y]);   → 1
}
```

In overall case $n+1 = O(n)$

In ATC $= O(1)$

$= O(t+1) = O(1)$

Arrange the following function in increasing order of time complexity

(1) $f_1(n) = 2^n$ (power) $n^{3/2}$ $n\log n$

$f_2(n) = n^{3/2}$ (power)$^{\frac{3}{2}}$ $\log n$ $\log(n\log n)$

$f_3(n) = n\log n$ $\longrightarrow$ least.

$f(4)(n) = n^{\log n}$ (power)$^{\frac{3}{2}}$ $\log n$ $< \log n + \log(\log n)$

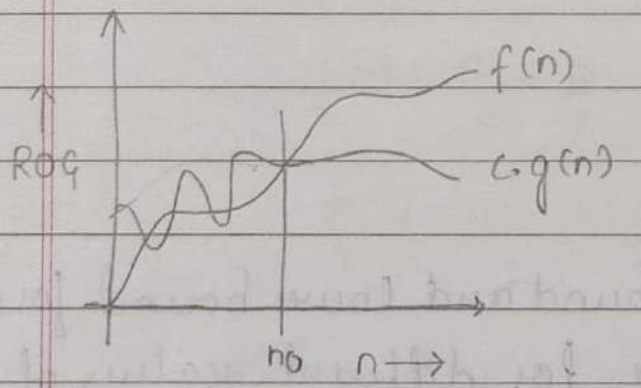
$\log f_1(n) = n \log 2 \quad n = O(n)$

$\log f_2(n) \log n = \log \frac{3}{2} \log n \quad \frac{3}{2} \log n = O(\log n)$

$\log f_4(n) = \log n \log n = O(\log n)^2$

$f_3 \leq f_2 < f_4 < f_1$

* Omega Notation (Ω) It gives the tight lower bound of the function. It is represented as $f(n) = \Omega(g(n))$ It means $g(n)$ is lower bound for $f(n)$



$0 <= C.g(n) <= f(n), \quad \forall n \geq n_0$

* Prove $f(n) = 100n + 5$ doesn't belongs to $\Omega(n^2)$

$f(n) = \Omega(g(n))$

By defn,

$0 <= C.g(n) <= f(n), \quad \forall n \geq n_0$

Given, $f(n) = 100n + 5 \quad \cong 100n$

OR

$100n + 5 <= 100n + 5n$

$100n + 5 <= 105n \rightarrow (1)$

$C.n^2 <= 100n + 5 \rightarrow (2)$

From (1) and (2)

$C.n^2 <= 105n$

$n <= \frac{105}{C}$ // It is true for specific values of n.

* $f(n) = n.1 \in \Omega(n)$

By dfn.

$0 <= c.g(n) <= f(n), \forall n \geq n_0$

$= C.n <= n$ $\therefore f(n) \in \Omega(n)$

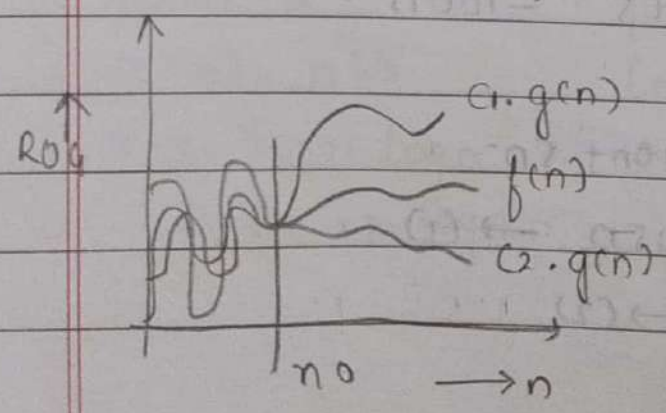
Theta Notation.

It says if the upper bound and lower bound for a given function is same for different values of constant.

$f(n) = \Theta(g(n))$. It means $g(n)$ is both lower and upper bound.

-OR-

$f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$



$0 <= c_2.g(n) <= f(n) <= c_1.g(n)$, $\forall n \geq n_0$

Find theta of $f(n) = \frac{n^2}{2} - \frac{n}{2}$.

→ $f(n) = \frac{n^2}{2} - \frac{n}{2}$

$$\frac{1}{5} n^2 <= \frac{n^2}{2} - \frac{n}{2} <= 1 \cdot n^2 \quad , \forall\ n \geq 3.$$

$c_1 = 1$; $c_2 = \frac{1}{5}$; $g(n) = n^2$.

Prove $n \notin \Theta(n^2)$

Consider

$$c_2 \cdot g(n) <= f(n)$$

$$c_2 \cdot n^2 <= n$$

$$c_2 \cdot n <= 1$$

$$n <= \frac{1}{c_2}$$

This is true for $n <= 1/c_2$ It is not true for large values of n. Hence it is not belonging to $\Omega(n^2)$

Consider

$$f_n <= c_1 . g(n)$$

$$n <= c_1 . n^2$$

$$1 <= c_1 n .$$

This is true for large set of values. Hence $f(n) = O(n^2)$ Since O and Ω notadiontions aren't same it doesn't belong to $\Theta(n^2)$

Analysis of recursive functions:

(i) Backward substitution:

Ex: $T(n) = 3T(n-1) + 1, \quad n > 0$ (Recursive)

$T(n) = 1, \quad n = 0$ (Base condition)

Ex: $T(3) = ?$

$T(3) = 3T(2) + 1 = 40$

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

$T(1) = 3T(0) + 1$

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

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

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

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

$= 3^3 T(n-3) + [3^2 + 3^1 + 3^0]$

for some 'k'

$= 3^k T(n-k) + [3^{k-1} + 3^{k-2} + \dots + 3^1 + 3^0]$

Put $k = n$,

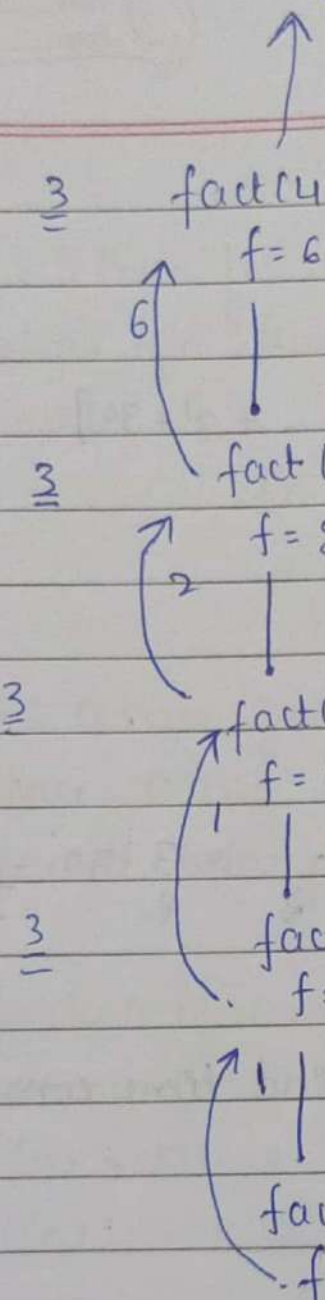
$= 3^n \cdot T(0) + \left[\frac{3^{k-1+1} - 1}{(3-1)}\right]$

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

$= 3^n + \frac{3^n - 1}{2} = 3^n\left[1 + \frac{1}{2}\right] - \frac{1}{2} = \frac{3}{2} \cdot 3^n - \frac{1}{2} = O(3^n)$

* Write a recursive function to find time complexity of factorial.

```
int fact (int n)
{   int fi;
    if (n==0)
        return 1;
    else
    { f= fact(n-1);
      return n*f;
    }
}
```



$T(4) = T(3) + 1$

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

- - - -

// Recursive

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

$T(n) = 1, \quad n = 0$

// Base

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

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

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

$T(n) = (T(n-4) + 1) + 1 + 1 + 1$.

$n = 0$ // Base.

'for some k'

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

$n - k = 0$.

$k = n$

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

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

$T(n) = 1 + n$

$\therefore O(n)$

* int fun (int n)
{ if (n==1)

```
    return 1;
  else
    { S=fun(n-1);
      return n+5;
    }
}
```

2)
```
int fun(int a, int b)
{  if(a==0)
     return b;
   else
     { m=fun(b%a, a);
       return m;
     }
}
```

(i) $T(n) = T(n-1) + 1$ (Recurrence), $n > 1$

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

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

$T(n) = T(n-4)+1+3$

.....

for some 'k'

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

Base if $n==1$

$n-k=1 \therefore k=n-1$

$T(n) = 1, \quad n=1$ (Base)

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

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

$T(n+1) = n$

$\approx T(n) = n$

$O(n)$

Consider $n = 5$

$T(5) = T(4) + 1 = 5$

$T(4) = T(3) + 1 = 4$

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

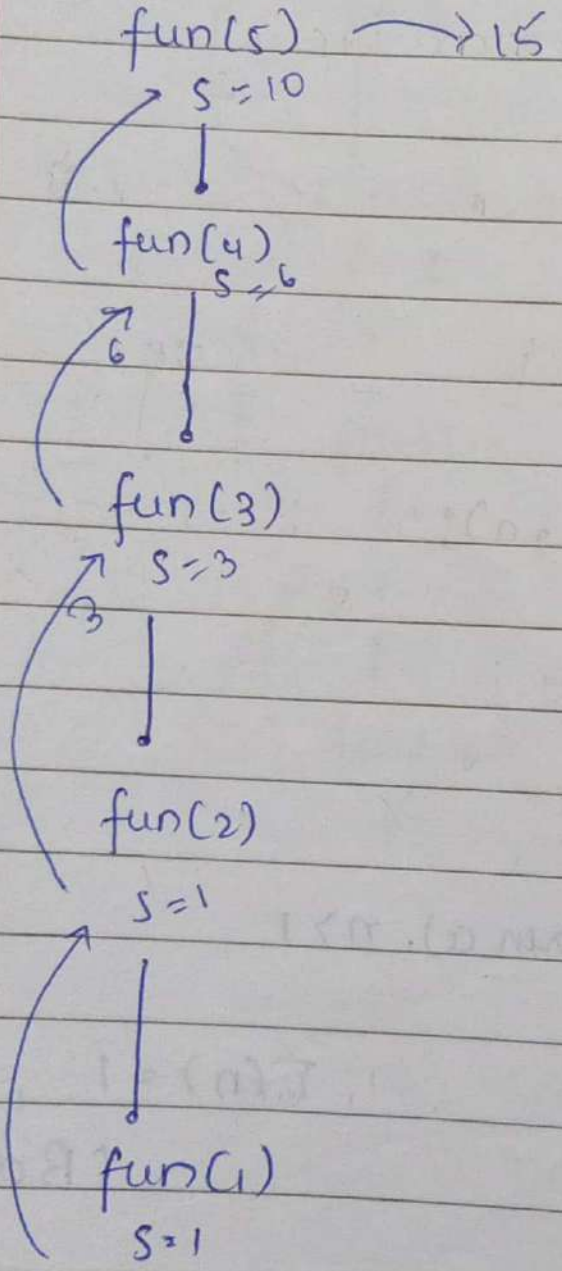
$T(2) = T(1) + 1 = 2$

$T(1) = 1$

for $n = 5$;

$T(5) = 5$

Recursion tree:



```
void TOH (int n, char source, char aux, char dst)
{ 1. — if (n == 0)
  2. —    return;
3. — TOH (n-1, source, dest, aux);
4. — printf ("move disk %d from %c to %c\n",
            n, source, dst);
5. — TOH (n-1, aux, source, dst);
}

main()
{ int n;
   read n;
   TOH (n, 'S', 'A', 'D');
}
```

o/p:- move disk 1 from S → D
move disk 2 from S → A
move disk 1 from D → A
move disk 3 from S → D
move disk 1 from A → S
move disk 2 from A → D
move disk 1 from S → D

$T(n-1)+1$

$T(n) = T(n-1) + 1 + T(n-1)$, $n > 0$

$T(n) = 1$, $n = 0$

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

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

$T(n) = 2[2^2 \cdot T(n-2) + [2^1 + 2^0]]$

$T(n) = 2^2[2T(n-3)+1] + [2^1 + 2^0]$

$T(n) = 2^3 T(n-3) + [2^2 + 2^1 + 2^0]$

for some 'k'

$T(n) = 2^k T(n-k) + (2^{k-1} + 2^{k-2} + \dots + 2^0)$

$n - k = 0$

$k = n$

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

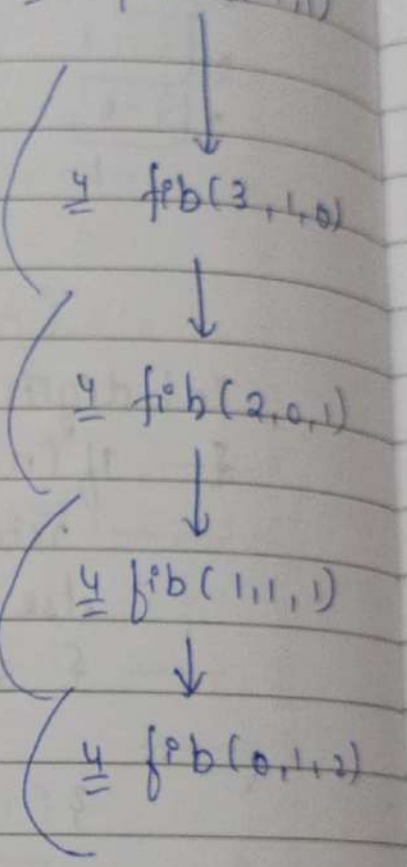
$$= 2^n(1) + \left(\frac{2^n - 1}{1}\right)$$

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

$T(n) = O(2^n)$ Exponential.

* Print fibonacci series

```
main()
{ int n;
  read n;
  fib(n,-1,1);
}
void fib (int N, int a, int b)
{  if(n>0)
   { printf("%d\n", a+b);
     fib (N-1, b, a+b);
   }
}
```



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

$T(n) = 1, \quad n = 0.$

for some k $\quad T(n) = T(n-k) + k$ $\qquad n-k=0$, $k=n+0$

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

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

* Nth term of fibonacci

$fib(1) = 0, \quad n = 1$

$fib(2) = 1, \quad n = 2$

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

$fib(4) = fib(3) + fib(2)$

$fib(n) = fib(n-1) + fib(n-2), \quad n > 2$

```
int fib (int n)
{  if (n==1)
     return 0;
   if (n==2)
     return 1;
   a = fib (n-1);
   b = fib (n-2);
   return a+b;
}
```

$T(n) = T(n-1) + T(n-2) + 1, \quad n \geq 2$

$T(n) = 1, \quad n <= 2$

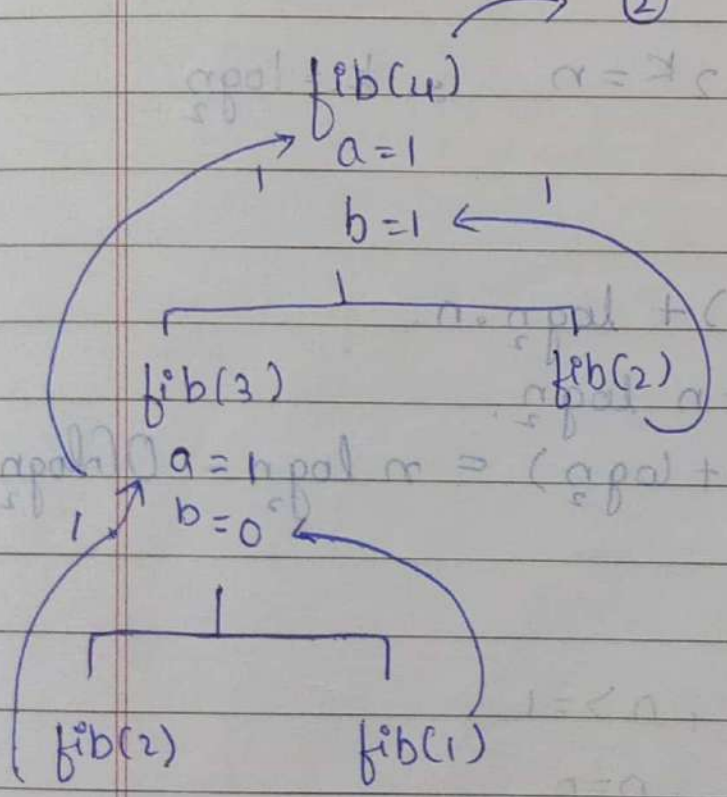
For asymptotic analysis for large values of n, $n-1 \cong n-2$ so

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

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

. . .

$T \cdot O(2^n)$ // Towers of Hanoi



* $T(n) = 2 \cdot T(n/2) + n, \quad n > 1$ } Merge Sort

$T(n) = 1, \quad n = 1$ }

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

$T(n) = 2\left[2T(n/4) + \frac{n}{2}\right] + n$

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

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

$= 2^2\left[2T(n/8) + \frac{n}{4}\right] + 2n$

$= 2^3 T(n/8) + 4n + 2n$

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

for some 'k'

$T(n) = 2^k \cdot T(n/2^k) + kn$.

$\frac{n}{2^k} = 1 \qquad 2^k = n \qquad \therefore k = \log_2 n$.

$T(n) = nT(1) + \log_2 n \cdot n$.

$T(n) = n + n\log_2 n$.

$= n(1 + \log_2 n) = n\log_2 n = O(n\log_2 n)$

* $T(n) = 2T(n-1) - 1 \quad , n >= 1$

$= 1 \quad , n = 0$.

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

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

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

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

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

'for some "k"'

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

Base $n = 0$

$n - k = 0$

$k = n$

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

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

$= O(1)$

Summation formulas:

(a) $\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$

(b) $\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$

(c) $\sum_{i=1}^{n} i^k = 1^k+2^k+\cdots+n^k = \frac{n^{k+1}}{k+1}$

(d) $\sum_{i=0}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1} \quad (a \neq 1)$

(e) $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$

(f) $\sum_{i=1}^{n} i \cdot 2^i = 1\times 2 + 2\times 2^2 + \cdots + n\times 2^n$
$= (n-1)2^{n+1}+2$

(g) $\sum_{i=1}^{n} \log i = n \log n$

(h) $\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{n} = \ln n + \gamma$

$\gamma = 0.5772$, Euler's constant nth harmonic number

5. $$T_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ 2T(n-1)+1 & \text{otherwise} \end{cases}$$

$T(n) = 2T(n-1)+1$ // Recurrence relation

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

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

$= 2^2 [2T(n-3)+1] + 2^1 + 2^0$

$= 2^3 T(n-3) + 2^2 + 2^1 + 2^0$

For some 'K'

$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} - \cdots + 2^0$

Base condition

$n-K=0$

$K=n$

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

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

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

$O(2^n)$

-OR-

$T(n) = 2^4 T(n-4) + 2^3 + 2^2 + 2^1 + 2^0$

- - - - -

$T(n) = 2^n T(n-n) + 2^{n-1} + 2^{n-2} + \cdots + 2^3 + 2^2 + 2^1 + 2^0$

$$T(n) = 2^n T(0) + 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0$$

$$= 2^{n-1} + 2^{n-2} + \dots + 2^0$$

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

It is a G.P. series

$$S = \frac{a(r^n - 1)}{r - 1}$$

a = first term, r = common ratio
n - no. of terms.

$a = 1; r = 2; n = n$

* $$C(n) = \begin{cases} 0 & n = 0 \\ 1 + C(n-1) & \text{otherwise} \end{cases}$$

Taking the recurrence relation:-

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

$$C(n) = 1 + [1 + C(n-2)]$$

$$= 2 + C(n-2)$$

$$= 2 + [C(n-3) + 1)$$

$$= 3 + C(n-3)$$

for 'n'

$$C(n) = n + C(n-n)$$

$$= n + 0$$

$$= n$$

* $T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2)+1 & \text{otherwise} \end{cases}$

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

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

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

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

$= T(n/2^3) + 3$

'for some K'

-OR- $T(n) = T(K) = T(n/2^K) + K$

for 'n'

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

$\frac{n}{2^n} = 1 \quad \therefore T(1) = 0.$

$n = 2^n$

$n = \log_2 n$

$n = 2^K$

$K = \log_2 n.$

$T(2^K) = T(1) + \log_2 n$

$= \log_2 n$

$T(n) = T\left(\frac{2^n}{2^n}\right) + \log_2 n$

$= 0 + \log_2 n$

$T(n) = \log_2 n$

// $n = 2^K$

$\log n = \log_2 K$

$\log n = K \log 2$ // Assume base is 2.

$\log_2 n = K(1)$

* $x(n) = x(n-1) + 5$, $n > 1$

0 , 1

Take the recurrence relation,

$x(n) = x(n-1) + 5$ // for $n = 1$

$= [x(n-2) + 5] + 5$ $n-(n-1)$

$= [x(n-2)] + 10$

$= [x(n-3) + 5] + 10$

$= x(n-3) + 15.$

$= x(n-(n-1)) + 5 * (n-1)$

$= x(1) + 5(n-1)$

$= 5(n-1)$

* $x(n) = 3x(n-1)$, $n > 1$

$= 4$, $n = 1$

Take the recurrence relation

Following the backward substitution,

$x(n) = 3 \cdot x(n-1)$

$x(n) = 3(3 \cdot x(n-2))$

$= 3^2 x(n-2)$

$= 3^2 (3 \cdot x(n-3)) = 3^3 x(n-3)$

$= \ldots \quad 3^{n-1} x(n-(n-1))$

$= \quad 3^{n-1}\, 4 = \frac{3^n \times 4}{3}$

1 $\sum_{i=1}^{n-1} 1 = (n-1-1+1) = n-1$

Upperbound - Lowerbound + 1

2 $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-(i+1)+1) = \sum_{i=0}^{n-2} (n-1-i-1+1)$

$= \sum_{i=0}^{n-2} (n-(i)-1) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$

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

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

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

$= \frac{(n-1)}{2}\left[2(n-1) - (n-2)\right]$

$= \frac{(n-1)}{2}\left[2n - 2 - n + 2\right] = \frac{(n-1)(n)}{2}$

$= \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2} \approx n^2$

3. $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n+1} \sum_{j=0}^{n-1} (n-1+1) = n \sum_{i=0}^{n-1} (n-1+1)$

$= n^2 \sum_{i=0}^{n-1} 1$

$= n^2 (n-1-0+1) = n^3$

Stacks are nothing but linked lists with the restrictions that insertion and deletion can happen at only one end i.e. top

Only top is accesible. or associatid with top

Operations: push
pop
Peek

It is LIFO OR FILO

Conditions:- Overflow
Underlow

```
#define MAX 100
struct stack
{   int items[MAX];
    int top;
};
typedef struct stack STACK;

STACK S;
S.top = -1;

//Overflow   top= MAX-1
```

Ex:-
function call
undo, redo
mirror-polish notation
In XHTML

Queues: First in first out structure.

* $f(n) = 3\log n + 100$ $g(n) = \log n$ is $3\log n + 100$ $O(\log n)$?

$n = 1$ $f(n) = 100$ $g(n) = 0$

$n = 2$ $f(n) = 103$ $g(n) = 1$

$n = 3$ $f(n) = 1.43 + 100$ $g(n) = 4.75$

$f(n) = 104.75$

$\therefore f(n) \geq g(n)$

So $f(n)$ is $\Omega(g(n))$

* $f(n) = 100n + 5$ $g(n) = 100n + n$ is $f(n)$ $O(g(n))$?

is $100n + 5$ $O(n)$?

$n = 1$ $105 = f(n)$ $g(n) = 101$

$n = 2$ $f(n) = 205$ $g(n) = 202$

$n = 3$ $f(n) = 305$ $g(n) = 303$

$n = 4$ $f(n) = 405$ $g(n) = 404$

$n = 5$ $f(n) = 505$ $g(n) = 505$

$n = 6$ $f(n) = 605$ $g(n) = 606$

for all $n \geq n_0$

$n_0 = 5$

$f(n) = O(g(n))$

$= O(n)$

* $f(n) = 6n^2 - 8n$

$g(n) = 100n^3$

$n=1$	$f(n) = -2$	$g(n) = 100$
$n=2$	$f(n) = 8$	$g(n) = 400 \times 2 = 800$
$n=3$	$f(n) = 30$	$g(n) = 2700.$

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

so $f(n) = O(g(n))$

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

* $f(n) = n^3$ and $g(n) = n^2$

$n=1$	$f(n) = 1$	$g(n) = 1$
$n=2$	$f(n) = 8$	$g(n) = 4$
$n=3$	$f(n) = 27$	$g(n) = 9$

$f(n) \geq g(n)$ $\quad n_0 = 1$

$f(n) = \Omega(n^2)$

* $f(n) = 3 * n^2$ $\quad g(n) = n$

$n=1$	$f(n) = 3$	$g(n) = 1$
$n=2$	$f(n) = 3 * 2^2$	$g(n) = 2$
	$f(n) = 12$	$g(n) = 2.$
$n=3$	$f(n) = 3 * 9$	$g(n) = 3.$
	$= 27$	

$f(n) \geq g(n)$ $\quad f(n) = \Omega(g(n)) = \Omega(n)$

$n \geq n_0$ $\quad n_0 = 1$

Hashing:-

A hash table is a datastructure used to implement the associative array, that maps key with value.

Hash: It is a value that is computed by a hash function based on the i/p key.

Hashing is a technique that is used to implement the ADT dictionary or dictionary

Set of operations:

→ Insert
→ delete
→ search

* Define hash:-

Hash: It is a value computed by hash f'n based on the input key

* Define hashing:-

Hashing is a technique used to implement the dictionary i.e set of operations like, insert, delete and search

- OR

Hashing is -OR- hash table is datastructure that is used to implement the associative array

a structure that maps keys with values.

Apps:-

1. Symbol table
2. Databases (extended hashing) on disks.
3. Search engines
4. Ticket booking.
5. Bank acounts maintainence.

Idea:

→ Use the hash function and compute the slot for each key.

→ store the info in h(K)

A hash function 'h' transforms key 'k' to an index in hash table

A hash function 'h' transforms key 'k' to an index in hash table.

$T[0 \cdots m-1]$

$H: U \rightarrow \{0, 1 \cdots m-1\}$

Direct address

One to one mapping b/w key and value (index)

$T[0 \cdots m-1]$ $H: U - \{0, 1 \cdots m-1\}$

Hashing is a technique which is used to search as fast as possible. If the input is in range

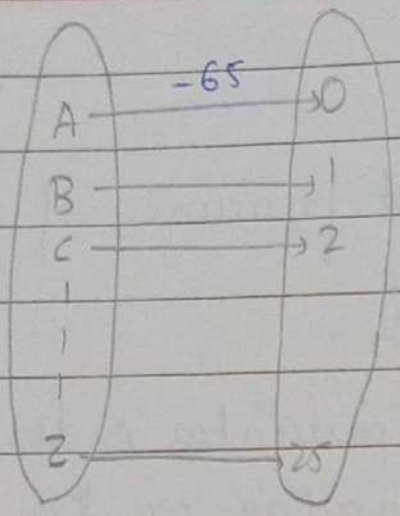
Ex: set of character - 256,

Natural no's b/w 1-100, etc

Then we use direct address table

DAT : It is a one-one function from set of keys to indices (values)

One to one



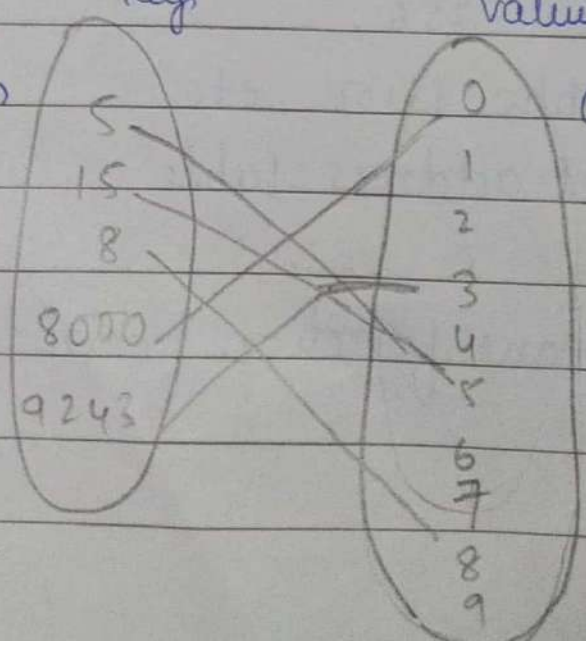
Keys — values

Domain — Codomain.

For a set of natural numbers we can't use DAT (an array) of size infinite.
In such cases we use generalized hash functi

Hash function: A hash function takes string of arbitrary length and gives fixed length output

A hash function is a many to one function



Two or more keys mapping to the same index is called collision.

* Write a C program to find the first repeated characters in a given string

```
for (i=0; str[i]; i++)
{   count[str[i] -65]++;
      if (count [str[i] -65] >1)
                  break;
}
print str [i];
```

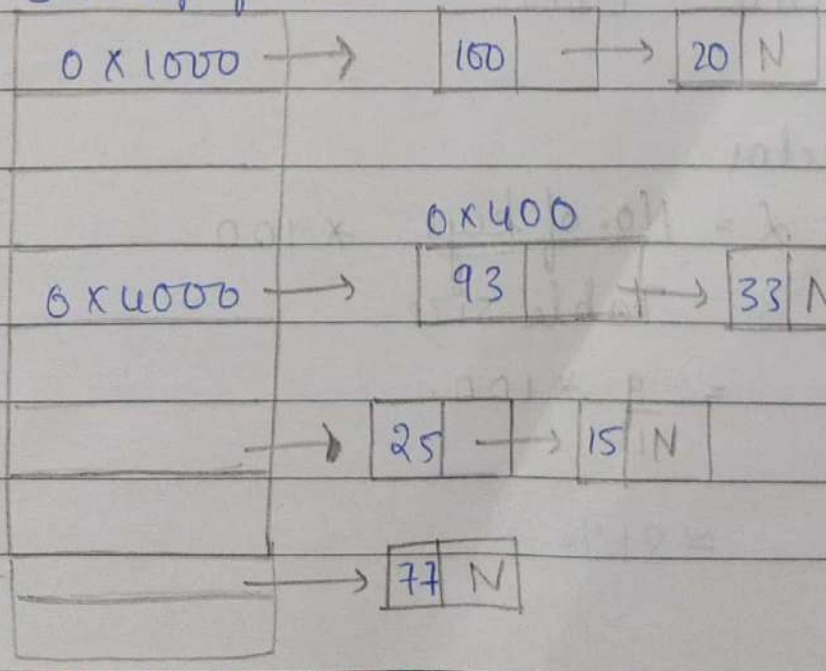
Collision Resolution techniques:-

1. Separate chaining

Keys : 100, 25, 77, 93, 33, 20, 15

$\mu(K) = K\%10$.

(Array of address pointer)



list with one node ↓ best

list with max len ↓ worst

O(1) - best

2) Linear Probing:

keys: 31, 19, 2, 13, 25, 24, 21, 9, 30

$H(K) = K \% 11$ → usually large no (107)

// Increment by one more scatter

0	9
1	30
2	2
3	13
4	25
5	24
6	
7	
8	19
9	31
10	21

rehash (key) = (H(key) + 1) % Table size

H(24) = 24 % 11 = 2

rehash (24) = (2 + 1) % 11 = 3

(24) = (3 + 1) % 11 = 4

(25) = (4 + 1) % 11 = 5

When are multiple of 11 - worst case

Best case O(1).

Load factor

$$\alpha = \frac{\text{No. of keys}}{\text{Table size}} * 100$$

$$= \frac{9}{11} * 100.$$

$$\simeq 81\%$$

If 'α' is greater ≥ 'T' we double the size of the hash table & rehash all the keys acc to new hash function.

T = 70% if T is more then more rehash

Key	α
31	9%
19	18%
2	27%
13	36%
25	45%
24	54%
21	63%
9	72%
30	

When threshold is reached then we double the size of the array then we delete the previous one. then rehash the values.

H(K) = K% 24

" 21

9 — Threshold reached.

H(K) = K % 22 but prime no. is prefered

= K % 23

$$\alpha = \frac{9}{23} * 100$$

$$= 39\%.$$

	24	2	25	30	31	9	13	19	21
⓪	1	2	3	7	8	9	13	19	21

Quadratic Probing:

rehash (k) = (K + i²) % Tablesize

where, i ∈ z+

keys = 31, 19, 2, 13, 25, 24, 21, 9, 30

H(K) = k % 11 // Increment square less scatter

0	
1	30
2	2
3	13
4	25
5	
6	24
7	9
8	19
9	31
10	21

rehash(13) = (13 + 1²) % 11 = 3

(25) = (25 + 1²) % 11 = 4

rehash(24) = (24 + 1²) % 11 = 3

(24) = (24 + 2²) % 11 = 6

rehash(9) = (9 + 1²) % 11 = 10

= (9 + 2²) % 11 = 2

= (9 + 3²) % 11 = 7

rehash(30) =

Double hashing/rehash

the rehash is a function. The rehash function h_2, $h_2 \neq 0$ or $h_2 \neq h_1$.

keys : 58, 14, 91, 25

$H_1(K) = K \% 11$

$H_2(K) = 7 - (K \% 7)$

if greater than table size % tablesize

rehash (key) = $h_1(key) + i\, h_2(key)$

where, $i \in z^+$

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	25.
10	14

rehash(key^{14}) =

$3 + 1(7 - (14 \% 7))$

$3 + 7 = 10$

rehash (91) =

$H_1(91) + 1[7 - (91 \% 7)]$

$3 + 1(7 - 91 \% 7)$

$3 + 1(7 - 0)$

$= 10.$

rehash(91)

$= H_1(91) + 2\, H_2(91)$

$= 3 + 2(7)$

$= 17 >$ Tablesize

So % tablesize $= 17 \% 11 = 6.$

rehash(25) = 3 + 1 (7 - (25%.7))

= 3 + 1 (7 - 4)

= 3 + 3 = 6.

rehash(25) = 3 + 2 × 3 = 9